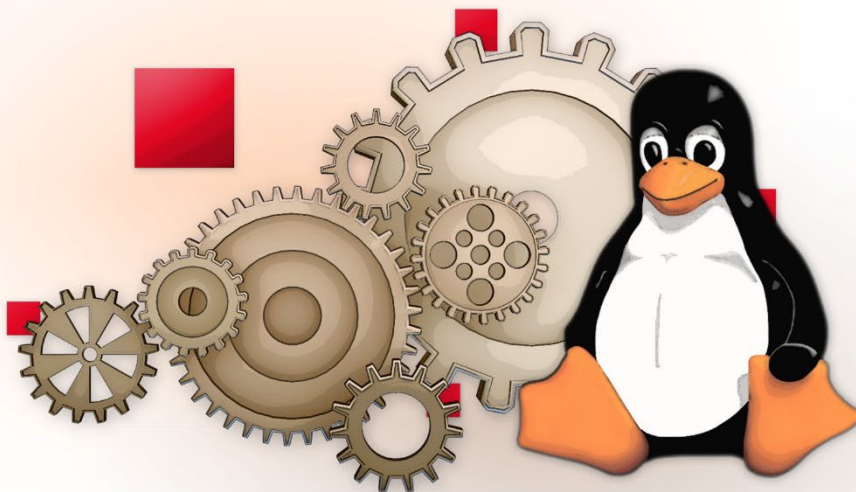


PCAN Driver for Linux v9

CAN Driver and Library API for Linux

User Manual



Relevant Products

Product Name	Version	Part number
PCAN Driver for Linux	9.x.x	not applicable

PCAN® is a registered trademark of PEAK-System Technik GmbH.

Other product names in this document may be the trademarks or registered trademarks of their respective companies. They are not explicitly marked by ™ or ®.

© 2026 PEAK-System Technik GmbH

Duplication (copying, printing, or other forms) and the electronic distribution of this document is only allowed with explicit permission of PEAK-System Technik GmbH. PEAK-System Technik GmbH reserves the right to change technical data without prior announcement. The general business conditions and the regulations of the license agreement apply. All rights are reserved.

PEAK-System Technik GmbH
Leydheckerstraße 10
64293 Darmstadt
Germany

Phone: +49 6151 8173-20
Fax: +49 6151 8173-29

www.peak-system.com
info.peak@hms-networks.com

Document version 3.12.0 (2026-01-06)

Contents

1 Disclaimer	4
2 Introduction	5
2.1 Features	5
2.2 System Requirements	6
2.3 Scope of Supply	6
3 Installation	7
3.1 Build Binaries	7
3.2 Install Package	9
3.3 Configure Software	10
3.4 Configure Non-PnP-Hardware	12
4 Usage of the Driver	14
4.1 Driver loading	14
4.2 Udev Rules	15
4.3 /proc Interface	19
4.4 /sysfs Interface	21
4.5 lspcan Tool	25
4.6 pcanosdiag.sh Tool	27
4.7 read/write Interface	27
4.8 CAN 2.0/CAN FD test Directory	29
4.8.1 receivetest	30
4.8.2 transmitest	31
4.8.3 pcan-settings	32
4.8.4 bitratetest	33
4.8.5 pcanfdtst	34
4.9 CAN XL utils-xl Directory	39
4.9.1 pcanxl-loop	40
4.9.2 pcanxl-ping / pcanxl-pong	41
4.9.3 pcanxl-gen / pcanxl-dump	44
4.9.4 pcanxl-write / pcanxl-read	45
4.9.5 CAN CC/FD/XL Compatibility	46
4.10 netdev Mode	54
4.10.1 assign Parameter	55
4.10.2 defclk Parameter	55
4.10.3 ifconfig/iproute2	55
4.10.4 can-utils	58
4.11 USB Mass Storage Device Mode	58
5 Developer Guide	62
5.1 chardev Mode	62
5.1.1 CAN 2.0 libpcan API	64
5.1.2 CAN FD libpcanfd API	66
5.1.3 CAN XL libpcanxl API	80
5.2 netdev Mode	85

1 Disclaimer

The files provided are part of the PCAN Driver for Linux package.

This is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 3 of the License, or (at your option) any later version.

The software is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY, without even the implied warranty for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with the software package. If not, see www.gnu.org/licenses/.



IMPORTANT: It is strictly prohibited to use intellectual property from the provided source code for developing or producing compatible hardware. All rights are reserved by PEAK-System Technik GmbH.

2 Introduction

With the PCAN Driver for Linux, you can use CAN CC (CAN Classic, a.k.a. CAN 2.0), CAN FD, and, since v9, CAN XL interface hardware products from PEAK-System under Linux-based systems. Even if the use of Linux 2.6 kernels is declining, the canonical age of the driver ensures compatibility with some versions of this kernel line and with older PEAK-System CAN interface hardware products.

The driver is also compatible with the latest versions of well-known real-time (RT) extensions like Xenomai¹ and RTAI², by interfacing to the common "Real Time Driver Mode" model.

Historically, the PCAN Driver for Linux provides an application programming interface called *chardev* by implementing the character mode device driver system calls (open, read, write, close, poll, ioctl). Since version 20070306_n, the driver also provides a *netdev* interface which, by integrating the Kernel SocketCAN network sub-layer, provides applications with access to the PEAK-System CAN channels via the socket interface of the Linux kernel. The choice of the selected interface is exclusively done when building the driver; the driver cannot run offering both interfaces at the same time.

 **Note:** Since the Linux kernel v3.6, PEAK-System has worked to include the support of their most-used PC CAN interfaces in the mainline Kernel. Thus, if you plan to get access to the CAN bus with a PC CAN interface made by PEAK-System from a socket-based application, there is no need to install this PCAN Driver for Linux package anymore. The so-called *netdev* interface is however kept for backward compatibility.

Version 9 of the PCAN Driver for Linux is a major evolution since it mainly includes the support of the new CAN XL specification. Because of the new features CAN XL proposes, the historical *chardev* API has had to evolve, too. Time has come for PCAN to propose a more modern and scalable new *chardev* interface, while the "old" good ones are obviously always supported.

The package is always evolving, because of the constant support of some new CAN interface hardware products made by PEAK-System, some new versions of tools and Kernels, or because of some bug fixing. The latest version can be downloaded from the PEAK-System website:
www.peak-system.com/linux/

2.1 Features

- └ Support of all CAN CC (CAN Classic, a.k.a. CAN 2.0), CAN FD, and CAN XL PC CAN interfaces made by PEAK-System
- └ Support of all 2.6.x, 3.x, 4.x, 5.X and 6.x Linux Kernels in 32-bit and 64-bit environments
- └ DESTDIR and cross-compilation supported
- └ Udev system support
- └ Enhanced *sysfs* integration
- └ Optimized character mode device driver interface (*chardev*) supporting CAN CC, CAN FD, and CAN XL standard and multiple messages transfers between applications and the driver
- └ SocketCAN device driver interface (a.k.a. *netdev*) supporting CAN CC, CAN FD, and CAN XL features (since Linux v6.19), with improved NETLINK integration (*ip link* support).

¹ Website Xenomai: xenomai.org

² Website RTAI: www.rtai.org

- └ Real-time Linux extensions like Xenomai 3.x and RTAI 4.x and 5.x are supported by the driver, as well as by the user space library and the test and examples applications (*chardev* interface only)
- └ Full binary compatibility with existing CAN CC and CAN FD *chardev* applications that run over older versions of the driver (8.x and older)

2.2 System Requirements

- └ Linux-based system running a 32-bit or a 64-bit Kernel
- └ PC CAN interface from PEAK-System
- └ make, gcc
- └ The kernel headers (or Linux headers) package of the running Linux or the sources tree of a cross-compiled Kernel
- └ g++ and libstdc++
- └ libpopt-dev package



Note: The g++ compiler as well as the libpopt-dev package are only required for building some user space applications from the `test` directory.

2.3 Scope of Supply

- └ PCAN Driver for Linux installation including
 - device driver module sources and Makefile
 - user libraries sources and Makefile
 - test and tools applications sources and Makefile
 - Udev rules
 - libpcanbasic for Linux library and examples sources and Makefile
- └ Documentation (this user manual) in PDF format

3 Installation

The PCAN Driver for Linux is an *out-of-tree* driver module, and because of the GPL, it is provided in a (compressed) *tarball* package including the source files of the driver as well as the user libraries and some test utilities and tools (see 2.3 *Scope of Supply* on page 6).

This chapter covers the setup of the whole driver package under non-RT and RT Linux systems (*root* privileges are required for the installation part). Also, cross-compilation options are explained.

3.1 Build Binaries

▶ Do the following to install the package:

1. Untar the compressed tarball file from your `$HOME` (for example) directory:

```
$ tar -xzf peak-linux-driver-X.Y.Z.tar.gz
$ cd peak-linux-driver-X.Y.Z
```

2. Clean the world:

```
$ make clean
```

▶ To build non-real time binaries with default configuration:

```
$ make
```

 **Note:** This behavior is new from v8.x of the driver! In former versions, the global `make` command did build enabling the *netdev* interface rather than the *chardev* one. The main reason for that change is that a great number of PEAK-System CAN interface hardware products are now natively supported by the mainline kernel as SocketCAN interfaces³. Thus, driver users are supposed to prefer using the *chardev* interface instead. But of course, the *netdev* interface can always be selected by rebuilding the package with:

```
$ make netdev
```

▶ To build real-time binaries running in a Xenomai kernel:

```
$ make xeno
```

▶ To build real-time binaries running in an RTAI kernel:

```
$ make rtai
```

³ Kernel code: elixir.bootlin.com/linux/v3.4/source/drivers/net/can/usb/peak_usb/pcan_usb_core.c

Note: Selecting one of the above real-time compilations also removes the support of some of the non-RT PC CAN interfaces (like USB adapters, for example).

▶ To cross-compile binaries:

```
$ make KERNEL_LOCATION=/where/are/the/kernel/headers
```

Making something from the package's root directory recursively makes this thing into the following directories:

```
└─ driver
└─ lib
└─ test
└─ utils-xl
└─ libpcanbasic
```

It is equivalent to the following commands:

```
$ make -C driver
$ make -C lib
$ make -C test
$ make -C utils-xl
$ make -C libpcanbasic
```

▶ Making the 32-bit version of the library:

Since driver version 8.5, the 32-bit version of the `libpcan` library is automatically built (and installed) when running a 64-bit Kernel if the current C compiler can.

Note: The `gcc-multilib` package must be installed.

The default configuration of the PCAN Driver for Linux in non-RT configuration is to handle the support of all PC CAN interfaces. However, to save memory or to fix some cross-compilation and/or loading issues, it is possible to remove the support of some of these interfaces. The driver's Makefile handles the following set of switches from the `make` command line:

Variable	Value	Description
DNG	DONGLE_SUPPORT	Include the support of the parallel port CAN interfaces from PEAK-System in the driver
	NO_DONGLE_SUPPORT	Remove the support of the parallel port CAN interfaces from the driver (default)
USB	USB_SUPPORT	Include the support of the USB CAN interfaces from PEAK-System in the driver (default)
	NO_USB_SUPPORT	Remove the support of the USB CAN interfaces from the driver
PCI	PCI_SUPPORT	Include the support of the PCI/PCIe CAN interfaces from PEAK-System in the driver (default)
	NO_PCI_SUPPORT	Remove the support of the PCI/PCIe CAN interfaces from the driver
PCIEC	PCIEC_SUPPORT	Include the support of the ExpressCard CAN interfaces from PEAK-System in the driver (default). Note that loading the driver built with <code>PCIEC_SUPPORT</code> will automatically load the "i2c_algo_bit" module too.
	NO_PCIEC_SUPPORT	Remove the support of the ExpressCard CAN interfaces from the driver
ISA	ISA_SUPPORT	Include the support of the ISA/PC104 CAN interfaces from PEAK-System in the driver (default)
	NO_ISA_SUPPORT	Remove the support of the ISA/PC104 CAN interfaces from the driver

Variable	Value	Description
PCC	PCCARD_SUPPORT	Include the support of the PCCard CAN interfaces from PEAK-System in the driver
	NO_PCCARD_SUPPORT	Remove the support of the PCCard CAN interfaces from the driver (default)

Table 1: Supported PC CAN interfaces switches

For example, to build the driver including the support of the PCAN-Dongle and the PCAN-PC Card CAN interfaces:

```
$ make -C driver DNG=DONGLE_SUPPORT PCC=PCCARD_SUPPORT
```

To know which variant of the driver (*chardev*, *netdev* or *RT*) has been built, type in the "driver" directory:

```
$ modinfo pcan.ko | grep -e ^description:
```

3.2 Install Package

Once binaries are built, do the following to install the package:

1. Be sure to be in the driver package root directory:

```
$ cd peak-linux-driver-X.Y.Z
```

2. Install everything (root privileges are required):

- a) On Debian-based systems, users can use the `sudo` command:

```
$ sudo make install
```

- b) Otherwise, installation is done with:

```
$ su -c "make install"
```

The above setup will build and install the driver, the user libraries, and the test programs on the running system.

The driver can also be installed with DKMS support. DKMS is a software system that handles the rebuilding of the driver when a new Kernel has been installed in the running Linux based host. To take advantage of DKMS, one must install the driver with:

```
$ sudo make install_with_dkms
```

Calling the Makefile target "install_with_dkms" can be done from the root of `peak-linux-driver-x.y.z` or from the subdirectory "driver". In the first case, the other components (libs and programs) will be installed in the same way as if "make install" had been called.

3.3 Configure Software

The PCAN Driver for Linux runs with some default settings. Some of them can be changed by passing parameters to the module when it is loaded:

Parameter	Type	Description								
type	List of characters strings, separated by “,” (comma).	Gives the list of (maximum) 8 PC CAN interfaces that can't be detected by the plug-and-play system. Known types are: <table><tr><th>type</th><th>PC CAN interface</th></tr><tr><td>isa</td><td>ISA and PC/104</td></tr><tr><td>sp</td><td>Standard parallel port</td></tr><tr><td>epp</td><td>Enhanced parallel port</td></tr></table>	type	PC CAN interface	isa	ISA and PC/104	sp	Standard parallel port	epp	Enhanced parallel port
type	PC CAN interface									
isa	ISA and PC/104									
sp	Standard parallel port									
epp	Enhanced parallel port									
io	List of hexadecimal values, separated by “,” (comma).	Gives the list of I/O ports to use to dialog with the corresponding PC CAN interface (see <code>type</code>).								
irq	List of decimal values, separated by “,” (comma).	Gives the list of IRQ levels to connect to dialog with the corresponding PC CAN interface (see <code>type</code>).								
btr0btrl	Hexadecimal value.	Change the default (nominal) bitrate value set to every CAN/CAN FD channel when it is opened. The hexadecimal value is interpreted as a BTR0BTR1 value (see SJA1000 specifications). If this parameter is not provided when the module is loaded, the default bitrate value is 0x1c (500 kbit/s).								
bitrate	Numeric value. The ending <code>k</code> is interpreted as factor 1,000, while the ending <code>M</code> is interpreted as factor 1,000,000.	Change the default (nominal) bitrate value set to every CAN/CAN FD channel when it is opened. If this parameter is not provided when the module is loaded, the default bitrate value is 0x1c (500 kbit/s). See also the note below.								
dbitrate	Numeric value. The ending <code>k</code> is interpreted as factor 1,000, while the ending <code>M</code> is interpreted as factor 1,000,000.	Change the default CAN FD data bitrate value set to every CAN FD channel when it is opened. If this parameter is not provided when the module is loaded, the default data bitrate value is 2,000,000 (2 Mbit/s). Since v8.11, if <code>dbitrate</code> is 0, then the CAN FD device initializes in CAN 2.0 a/b mode only.								
xbitrate	Numeric value. An ending <code>k</code> is interpreted as factor 1,000, while an ending <code>M</code> is interpreted as factor 1,000,000.	Change the default CAN XL data bitrate value set to every CAN XL channel when it is opened. If this parameter is not provided when the module is loaded, the default CAN XL data bitrate value is 5,000,000 (5 Mbit/s). If <code>xbitrate</code> is 0, then the CAN XL device initializes according to <code>dbitrate</code> value.								
assign	Characters string	Change the default name assignment between PCAN and SocketCAN layer (see 4.10.1 <i>assign Parameter</i> on page 55). This parameter is only used when the <i>netdev</i> interface is selected.								
usemsi ⁴	Numeric value	This parameter controls usage of MSI for the PCIe-based CAN 2.0 cards: <table><tr><td>0</td><td>INTA mode (no MSI)</td></tr><tr><td>1</td><td>Full MSI mode (one IRQ per channel)</td></tr><tr><td>2</td><td>Shared MSI (one IRQ per device)</td></tr></table> 0 is the default mode.	0	INTA mode (no MSI)	1	Full MSI mode (one IRQ per channel)	2	Shared MSI (one IRQ per device)		
0	INTA mode (no MSI)									
1	Full MSI mode (one IRQ per channel)									
2	Shared MSI (one IRQ per device)									
irqmaxloop	Numeric value	Set the maximum number of read loops performed by the SJA1000 interrupt handler. Its default value is 6.								
irqmaxmsg	Numeric value	Set the maximum number of messages read by the SJA1000 interrupt handler per interrupt. Its default value is 8.								
fdusemsi ⁴	Numeric value	This parameter controls usage of MSI for the PCIe-based CAN FD cards: <table><tr><td>0</td><td>INTA mode (no MSI)</td></tr><tr><td>1</td><td>Full MSI mode (one IRQ per channel)</td></tr><tr><td>2</td><td>Shared MSI (one IRQ per device)</td></tr></table> 0 is the default mode.	0	INTA mode (no MSI)	1	Full MSI mode (one IRQ per channel)	2	Shared MSI (one IRQ per device)		
0	INTA mode (no MSI)									
1	Full MSI mode (one IRQ per channel)									
2	Shared MSI (one IRQ per device)									

⁴ Since the kernel allows MSI interrupt levels to be shared, the driver no longer distinguishes between values 1 and 2. These are only retained to ensure compatibility with older installations. Thus, if `(fd)usemsi != 0`, the driver requests as many interrupt levels as there are CAN channels to manage but accepts that the kernel may only offer one for all channels.

Parameter	Type	Description
<code>fdirqcl</code>	Numeric value	Define the number of frames received after which the CANFD firmware of the PCIe family cards will generate an interruption. Its default value is 16.
<code>fdirqtl</code>	Numeric value	Define the delay in 1/10th ms after which the CANFD firmware of the PCIe family cards will generate an interruption. Its default value is 10.
<code>fast_fwd</code>	Numeric value	Tell the USB CANFD interfaces to transfer the received frames as soon as they arrive rather than waiting a while to agglomerate them and thus minimize the USB traffic. Its default value is 0 (no fast forward).
<code>rxqsize</code>	Numeric value	Define the maximum number of messages that can be saved by the driver into the channel Rx queue. Once the Rx queue is full, next incoming CAN frames are discarded by the driver, and the <code>CAN_ERR_OVERRUN</code> (0x0002) flag is set. The default value is 2000.
<code>txqsize</code>	Numeric value	Define the maximum number of messages that can be stored by the application into the channel Tx queue. Once the Tx queue is full, next write will block the application or, if <code>O_NONBLOCK</code> was set, will fail with <code>errno = EAGAIN</code> . The default value is 500.
<code>rxqprealloc</code>	Numeric value	When the value is 1, the driver allocates the channel's message receive queue once and for all when it is loaded and releases it when it is removed from memory, as opposed to the default operation (value 0) where the receive queue is allocated every time the channel is opened and is released when the channel is closed.
<code>txqprealloc</code>	Numeric value	When the value is 1, the driver allocates the channel's message transmit queue once and for all when it is loaded and releases it when it is removed from memory, as opposed to the default operation (value 0) where the transmit queue is allocated every time the channel is opened and is released when the channel is closed.
<code>txqhiwat</code>	Numeric value	Define the maximum level beyond which the driver will no longer wake up the task waiting to write in the Tx queue. Its value varies from 50.00% (5000) to 100.00% (10000). The default value is 10000 (8000 in <i>netdev</i> mode)
<code>deftsmode</code>	Numeric value	Control how hardware timestamps are handled by the driver, if the PC CAN interface is able to provide such timestamps: 0 Timestamps are host (software) timestamps. Timestamp of a received CAN frame corresponds to the time the frame has been saved into the driver Rx queue. 1 Timestamps are based on host time + an offset made of hardware timestamps. The host time base is periodically updated by the driver when receiving notifications from the PC CAN interface. The offset corresponds to the hardware time found in the received CAN frames. 2 Same as 1 except that the hardware offset is cooked to handle any possible clock drift between the CPU and the PC CAN interface quartz. 3 Timestamps are made of hardware timestamps received from the PCAN interface. This means that this timestamp IS NOT a host time, but a count of seconds and μs. since the PC CAN interface has been initialized. 4 Reserved 5 Same as 1, except that the timestamp measurement is triggered on SOF instead of EOF (if device allows it). 6 Same as 2, except that the timestamp measurement is triggered on SOF instead of EOF (if device allows it). 7 Same as 3, except that the timestamp measurement is triggered on SOF instead of EOF (if device allows it). Default mode depends on the PC CAN interface: 0 SJA1000 based internal bus PCAN interfaces default and unique mode. 1 PCAN-USB and PCAN-USB Pro default mode. 2 CAN FD PCAN interfaces.
<code>defclk</code>	Characters string	Define the default clocks values for the PCAN channels, when the driver is built in <i>netdev</i> mode (see 4.10.2 <i>defclk</i> Parameter on page 55).

Parameter	Type	Description
defblperiod	Numeric value	Defines the period in ms. of the message informing the application of the bus load rate, when it has changed. The default value is 500
drvclkref	Numeric value	Define which clock reference the driver is based on, to compute host time timestamps it gives to the applications. Valid values can be: 0 Real-time clock 1 Monotonic clock 4 Monotonic raw clock 7 Boot time clock See also "Table 6: clock reference used by the driver for the timestamps" on page 21.

Table 2: Driver module parameters

Note: The `bitrate=` parameter has changed since v8.x of the driver. In previous versions, this parameter allowed modification to the default nominal bitrate, but by following the coding format of the BTR0BTR1 SJA1000 register only.

To ensure the best backward compatibility with the existing configurations, the `bitrate=` parameter is now parsed as follows:

- If the two first characters of the given value are `0x` or `0X` and if the hexadecimal value is smaller than 65536, then the value is always interpreted as a BTR0BTR1 bitrate specification (as the driver did in previous versions).
- Otherwise, and if the value is obviously a numeric value, then it is used as a bit-per-second (bit/s) bitrate specification.

These parameters and their values can be given on the `insmod` command line or can be written in the `/etc/modprobe.d/pcan.conf` file. The system administrator must edit this file, then to uncomment the options `pcan` line, and to write his own settings.

3.4 Configure Non-PnP-Hardware

Note: This paragraph only concerns the users of some non-plug-and-play PC CAN interfaces (like the PCAN-ISA and PC/104 PC CAN interfaces family). The configuration of the driver for the PCI/PCle and USB PC CAN interfaces families is entirely handled by the system.

When using some non-plug-and-play PC CAN interfaces, the driver must be informed of the IRQs and I/O ports configured for these boards (see the provided hardware reference and the corresponding jumpers' usage). The installation procedure of the PCAN Driver for Linux has already created a configuration text file which enables to define some optional arguments that are passed to the driver (see 3.3 *Configure Software* on page 10), when it is loaded.

For example, if the Linux host is equipped with a two channels ISA PC CAN interface board, and if IRQ 5 (resp. IRQ 10) and I/O port 0x300 (resp. 0x320) is the configuration selected by the dedicated jumpers on the board, then the `/etc/modprobe.d/pcan.conf` file must be changed like this:

```
$ sudo vi /etc/modprobe.d/pcan.conf
# PCAN - automatic made entry, begin -----
# if required add options and remove comment
options pcan type=isa,isa irq=10,5 io=0x300,0x320
install pcan /sbin/modprobe --ignore-install pcan
# PCAN - automatic made entry, end -----
```

The standard assignments for ISA and PC/104 PC CAN interfaces are (io/irq): 0x300/10, 0x320/5. The standard assignments for the PCAN-Dongle in SP/EPP mode are (io/irq): 0x378/7, 0x278/5.

4 Usage of the Driver

Once installed, and if the Udev system is running on the target system, the driver is automatically loaded by the system at the next boot for internal PC CAN interfaces, like the PCI/PCIe boards, or when the external PC CAN interface (like the USB adapters) is plugged into the system.

4.1 Driver loading

Being a module, the driver, however, can be loaded without rebooting the system by asking the system to probe for the PCAN module (root privileges are required):

```
$ sudo modprobe pcan
```

Note: The `modprobe` system command manages to load all the other modules the driver depends on. When using `insmod` instead, you must load all these modules manually:

```
$ modinfo pcan.ko | grep -e "^depends:"
depends:      i2c-algo-bit

$ sudo modprobe i2c-algo-bit
$ sudo insmod pcan.ko
```

The driver is reasonably verbose for the kernel: it logs one or several messages in the kernel logs buffer for each PC CAN interface it enumerates. Next, it will save messages only when something wrong has been detected.

Here are the messages it logs when it just has been loaded, for example:

```
$ dmesg | grep pcan
[495033.661956] pcan: Release_YYYYMMDD_n (1e)
[495033.661960] pcan: driver config [mod] [isa] [pci] [pec] [usb]
[495033.662103] pcan 0000:01:00.0: PCAN-PCI Express FD sub-system id 14h (4
channels)
[495033.674420] pcan: PCB v3h FPGA v3.5.7 (mode 3) Flash: Ok
[495033.674436] pcan: - pcie fd device minor 0 found
[495033.674445] pcan: - pcie fd device minor 1 found
[495033.674453] pcan: - pcie fd device minor 2 found
[495033.674462] pcan: - pcie fd device minor 3 found
[495033.674565] pcan 0000:04:00.0: PCAN-PCI Express FD sub-system id 5h (2
channels)
[495033.687966] pcan: PCB v2h FPGA v2.1.3 (mode 3) Flash: Ok
[495033.687982] pcan: - pcie fd device minor 4 found
[495033.687990] pcan: - pcie fd device minor 5 found
[495033.688073] pcan 0000:05:00.0: PCAN-PCI Express sub-system id 5h (2
channels)
[495033.706572] pcan: PCB v4h FPGA v1.4.0 (mode 3) Flash: Ok
[495033.706584] pcan: - pci/pcie device minor 6 found
[495033.706606] pcan: - pci/pcie device minor 7 found
[495033.707510] pcan 3-1.2:1.0: PCAN-USB Pro FD (MCU01h PCB02h) fw v3.4.4 b1
v2.1.0
[495033.707758] pcan: - usb fd device minor 32 number 1 found
[495033.708163] pcan: - usb fd device minor 33 number 2 found
[495033.709484] pcan 3-1.1.3:1.0: PCAN-USB (MCU0ah) fw v8.7.0
```

```
[495033.759745] pcan: - usb device minor 34 number 3 found
[495033.760826] pcan 3-1.1.4:1.0: PCAN-USB FD (MCU01h PCB01h) fw v3.5.1 bl
v2.1.0
[495033.761194] pcan: - usb fd device minor 35 number 7 found
[495033.774741] pcan 3-1.1.2:1.0: PCAN-USB XL (MCU01h PCB04h) fw v1.0.0 bl
v1.0.4
[495033.781704] pcan: - usb xl device minor 36 number 1344405832 found
[495033.795302] pcan 3-1.1.1:1.0: PCAN-USB XL (MCU01h PCB04h) fw v1.0.0 bl
v1.0.4
[495033.802243] pcan: - usb xl device minor 37 number 1344405831 found
[495033.802258] usbcore: registered new interface driver pcan
[495033.802877] pcan: major 511.
```

The driver enumerates each PC CAN interface according to its type. Up to version 8.5.1, each type had the following range of device minor numbers:

Hardware type	Minor number range
PCI/PCle	[0 ... 7]
ISA and PC/104	[8 ... 15]
SP mode	[16 ... 23]
EPP mode	[24 ... 31]
USB	[32 ... 39]
PC-CARD	[40 ... 47]

Table 3: Device minor number ranges

Because the number of CAN channels required has increased, since v8.6.0, the driver enumerates the PC CAN interfaces according to a different scheme:

Hardware type	Minor number range
PCI/PCle	[0 ... 31]
USB	[32 ... 63]
PC-CARD	[64 ... 71]
ISA and PC/104	[72 ... 79]
SP mode	[80 ... 87]
EPP mode	[88 ... 95]

Table 4: Device minor number ranges

This v8.6.0 new scheme gives more space to most used PC CAN interfaces, while always booking slot 32 for the first USB device channel.

4.2 Udev Rules

The Udev mechanism loads the non-RT driver when the system recognizes one of the devices it handles, at boot time or when the CAN interface hardware device is plugged into the system.

 **Note:** No device nodes files are created when running the real-time version of the driver module because it creates real-time (only) devices which are not connected in any way to the Udev system.

The installation of the driver package also adds some default rules to Udev, for helping the system to create the device nodes that implement the CAN channels handled by the driver (see `peak-linux-driver-x.y.z/driver/udev/45-pcan.rules`). By default, Udev creates one (character) device node under the `/dev` directory per CAN/CAN FD channel. The name of this device node is made of:

└─ pcan prefix

- └ PC CAN interface bus type (pci, isa, usb ...),
- └ fd suffix if the CAN channel is CAN-FD-capable, or xl suffix if the CAN channel is CAN-XL-capable,
- └ unique minor number

For example:

```
$ ls -l /dev/pcan* | grep "^c"
crw-rw-rw- 1 root root 511,  6 sept.  8 10:16 /dev/pcanpci6
crw-rw-rw- 1 root root 511,  7 sept.  8 10:16 /dev/pcanpci7
crw-rw-rw- 1 root root 511,  0 sept.  8 10:16 /dev/pcanpcifd0
crw-rw-rw- 1 root root 511,  1 sept.  8 10:16 /dev/pcanpcifd1
crw-rw-rw- 1 root root 511,  2 sept.  8 10:16 /dev/pcanpcifd2
crw-rw-rw- 1 root root 511,  3 sept.  8 10:16 /dev/pcanpcifd3
crw-rw-rw- 1 root root 511,  4 sept.  8 10:16 /dev/pcanpcifd4
crw-rw-rw- 1 root root 511,  5 sept.  8 10:16 /dev/pcanpcifd5
crw-rw-rw- 1 root root 511, 34 sept.  8 10:16 /dev/pcanusb34
crw-rw-rw- 1 root root 511, 32 sept.  8 10:16 /dev/pcanusbfd32
crw-rw-rw- 1 root root 511, 33 sept.  8 10:16 /dev/pcanusbfd33
crw-rw-rw- 1 root root 511, 35 sept.  8 10:16 /dev/pcanusbfd35
crw-rw-rw- 1 root root 511, 36 sept.  8 10:16 /dev/pcanusbxl36
crw-rw-rw- 1 root root 511, 37 sept.  8 10:16 /dev/pcanusbxl37
```

The Udev rules that the driver installs enable the system to create some symbolic links that give much more information about the CAN channel:

1. Udev rules create one `/dev/pcanX` per CAN channel
2. Udev rules group CAN channels according to their PC CAN interface into the same subdirectory whose name is made of the PC CAN interface product name
3. Udev default rules also create some other symbolic links if the CAN channel exports a `devId` property (different from -1) under `/sys` (as USB devices can, as well as PCIe devices since v8.10 of the driver).

The example below demonstrates the complete list of `/dev/pcan*` nodes, symbolic links, and subdirectories the Udev rules provided with the driver might create.

```
$ ls -l /dev/pcan*
lrwxrwxrwx 1 root root 10 sept.  8 10:16 /dev/pcan0 -> pcanpcifd0
lrwxrwxrwx 1 root root 10 sept.  8 10:16 /dev/pcan1 -> pcanpcifd1
lrwxrwxrwx 1 root root 10 sept.  8 10:16 /dev/pcan2 -> pcanpcifd2
lrwxrwxrwx 1 root root 10 sept.  8 10:16 /dev/pcan3 -> pcanpcifd3
lrwxrwxrwx 1 root root 11 sept.  8 10:16 /dev/pcan32 -> pcanusbfd32
lrwxrwxrwx 1 root root 11 sept.  8 10:16 /dev/pcan33 -> pcanusbfd33
lrwxrwxrwx 1 root root  9 sept.  8 10:16 /dev/pcan34 -> pcanusb34
lrwxrwxrwx 1 root root 11 sept.  8 10:16 /dev/pcan35 -> pcanusbfd35
lrwxrwxrwx 1 root root 11 sept.  8 10:16 /dev/pcan36 -> pcanusbxl36
lrwxrwxrwx 1 root root 11 sept.  8 10:16 /dev/pcan37 -> pcanusbxl37
lrwxrwxrwx 1 root root 10 sept.  8 10:16 /dev/pcan4 -> pcanpcifd4
lrwxrwxrwx 1 root root 10 sept.  8 10:16 /dev/pcan5 -> pcanpcifd5
lrwxrwxrwx 1 root root  8 sept.  8 10:16 /dev/pcan6 -> pcanpci6
lrwxrwxrwx 1 root root  8 sept.  8 10:16 /dev/pcan7 -> pcanpci7
crw-rw-rw- 1 root root 511,  6 sept.  8 10:16 /dev/pcanpci6
crw-rw-rw- 1 root root 511,  7 sept.  8 10:16 /dev/pcanpci7
crw-rw-rw- 1 root root 511,  0 sept.  8 10:16 /dev/pcanpcifd0
crw-rw-rw- 1 root root 511,  1 sept.  8 10:16 /dev/pcanpcifd1
crw-rw-rw- 1 root root 511,  2 sept.  8 10:16 /dev/pcanpcifd2
crw-rw-rw- 1 root root 511,  3 sept.  8 10:16 /dev/pcanpcifd3
```



```
crw-rw-rw- 1 root root 511, 4 sept. 8 10:16 /dev/pcanpcifd4
crw-rw-rw- 1 root root 511, 5 sept. 8 10:16 /dev/pcanpcifd5
crw-rw-rw- 1 root root 511, 34 sept. 8 10:16 /dev/pcanusb34
crw-rw-rw- 1 root root 511, 32 sept. 8 10:16 /dev/pcanusbfd32
crw-rw-rw- 1 root root 511, 33 sept. 8 10:16 /dev/pcanusbfd33
crw-rw-rw- 1 root root 511, 35 sept. 8 10:16 /dev/pcanusbfd35
lrwxrwxrwx 1 root root 11 sept. 8 10:16 /dev/pcanusbpf32 -> pcanusbfd32
lrwxrwxrwx 1 root root 11 sept. 8 10:16 /dev/pcanusbpf33 -> pcanusbfd33
crw-rw-rw- 1 root root 511, 36 sept. 8 10:16 /dev/pcanusbxl36
crw-rw-rw- 1 root root 511, 37 sept. 8 10:16 /dev/pcanusbxl37

/dev/pcan-pci:
total 0
drwxr-xr-x 2 root root 80 sept. 8 10:16 0

/dev/pcan-pcie_fd:
total 0
drwxr-xr-x 2 root root 120 sept. 8 10:16 0
drwxr-xr-x 2 root root 80 sept. 8 10:16 1

/dev/pcan-usb:
total 0
drwxr-xr-x 2 root root 60 sept. 8 10:16 0
lrwxrwxrwx 1 root root 12 sept. 8 10:16 'devid=3' -> ../pcanusb34

/dev/pcan-usb_fd:
total 0
drwxr-xr-x 2 root root 60 sept. 8 10:16 0
lrwxrwxrwx 1 root root 14 sept. 8 10:16 'devid=7' -> ../pcanusbfd35

/dev/pcan-usb_pro_fd:
total 0
drwxr-xr-x 2 root root 80 sept. 8 10:16 0
lrwxrwxrwx 1 root root 14 sept. 8 10:16 'devid=1' -> ../pcanusbfd32
lrwxrwxrwx 1 root root 14 sept. 8 10:16 'devid=2' -> ../pcanusbfd33

/dev/pcan-usb_xl:
total 0
drwxr-xr-x 2 root root 60 sept. 8 10:16 0
drwxr-xr-x 2 root root 60 sept. 8 10:16 1
lrwxrwxrwx 1 root root 14 sept. 8 10:16 'devid=1344405831' -> ../pcanusbxl36
lrwxrwxrwx 1 root root 14 sept. 8 10:16 'devid=1344405832' -> ../pcanusbxl37
```

Here is the content of the subdirectories created by these Udev rules, one per PC CAN interface. The tree representation provides a better way of showing which CAN channel is connected to which PC CAN interface:

```
$ tree /dev/pcan-pci
/dev/pcan-pci
├── 0
│   ├── can0 -> ../../pcanpci6
│   └── can1 -> ../../pcanpci7
└── 1 directory, 2 files
```

```
$ tree /dev/pcan-pcie_fd
/dev/pcan-pcie_fd
├── 0
│   ├── can0 -> ../../pcanpcifd0
│   ├── can1 -> ../../pcanpcifd1
│   ├── can2 -> ../../pcanpcifd2
│   └── can3 -> ../../pcanpcifd3
├── 1
│   ├── can0 -> ../../pcanpcifd6
│   └── can1 -> ../../pcanpcifd7
├── devid=10 -> ../pcanpcifd0
├── devid=11 -> ../pcanpcifd1
├── devid=12 -> ../pcanpcifd2
└── devid=13 -> ../pcanpcifd3
2 directories, 10 files
```

```
$ tree /dev/pcan-usb
/dev/pcan-usb
├── 0
│   └── can0 -> ../../pcanusb34
└── devid=3 -> ../pcanusb34
1 directory, 2 files
```

```
$ tree /dev/pcan-usb_fd
/dev/pcan-usb_fd
├── 0
│   └── can0 -> ../../pcanusbfd35
└── devid=7 -> ../pcanusbfd35
1 directory, 2 files
```

```
$ tree /dev/pcan-usb_pro_fd
/dev/pcan-usb_pro_fd/
├── 0
│   ├── can0 -> ../../pcanusbfd32
│   └── can1 -> ../../pcanusbfd33
├── devid=1 -> ../pcanusbfd32
└── devid=2 -> ../pcanusbfd33
1 directory, 4 files
```

```
$ tree /dev/pcan-usb_xl
/dev/pcan-usb_xl
├── 0
│   └── can0 -> ../../pcanusbxl36
├── 1
│   └── can0 -> ../../pcanusbxl37
├── devid=1344405831 -> ../pcanusbxl36
└── devid=1344405832 -> ../pcanusbxl37

2 directory, 4 files
```

In the above configuration, a user application that wants to access to the CAN bus through the 2nd CAN port of the PCAN-USB Pro FD plugged to the host will be able to open indifferently:

- └ /dev/pcanusbfd33
- └ /dev/pcan33
- └ /dev/pcan-usb_pro_fd/devid=2
- └ /dev/pcan-usb_pro_fd/0/can1



Note: With a properly configured and running Udev system, all these devices' files and directories are generated on the fly. If the target non-RT system does not have a running Udev system, you must create the device files manually each time after driver installation. The driver package provides the shell script `driver/pcan_make_devices` for this. For example, to create a maximum of 2 devices of each type:

```
$ cd driver
$ sudo ./pcan_make_devices 2
```

4.3 /proc Interface

One of the first tests to do is to check whether the driver module is correctly loaded and runs as expected. To do so, read the `/proc/pcan` pseudo file.

Example:

```
$ cat /proc/pcan

*----- PEAK-System CAN interfaces (www.peak-system.com) -----
*----- Release_YYYYMMDD_n (9.Y.Z) MMM DD YYYY HH:MM:SS -----
*----- [mod] [isa] [pci] [pec] [usb] -----
*----- XX interfaces @ major MMM found -----
*n -type- -ndev- --base-- irq --btr- --read-- --write- --irqs-- -errors- status
0 pcifd -NA- 423a1000 039 0x001c 00000000 00000000 00000000 00000000 0x0000
1 pcifd -NA- 423a2000 039 0x001c 00000000 00000000 00000000 00000000 0x0000
2 pcifd -NA- 423a3000 039 0x001c 00000000 00000000 00000000 00000000 0x0000
3 pcifd -NA- 423a4000 039 0x001c 00000000 00000000 00000000 00000000 0x0000
4 pcifd -NA- 42451000 040 0x001c 00000000 00000000 00000000 00000000 0x0000
5 pcifd -NA- 42452000 040 0x001c 00000000 00000000 00000000 00000000 0x0000
6 pci -NA- f7900000 041 0x001c 00000000 00000000 00000000 00000000 0x0000
7 pci -NA- f7900400 041 0x001c 00000000 00000000 00000000 00000000 0x0000
32 usbfd -NA- ffffffff 001 0x001c 00000000 00000000 00000000 00000000 0x0000
33 usbfd -NA- ffffffff 002 0x001c 00000000 00000000 00000000 00000000 0x0000
34 usb -NA- ffffffff 003 0x001c 00000000 00000000 00000000 00000000 0x0000
35 usbfd -NA- 0 007 0x001c 00000000 00000000 00000000 00000000 0x0000
36 usbx1 -NA- ffffffff 071 0x001c 00000000 00000000 00000000 00000000 0x0000
37 usbx1 -NA- ffffffff 072 0x001c 00000000 00000000 00000000 00000000 0x0000
```

The `/proc/pcan` file contains:

- └ the driver version (release date and version numbers) with build date and time
- └ the list of the PC CAN interfaces the driver is able to handle (see Table 1 on page 9)
- └ the count of PC CAN interfaces detected by the driver and the major number the Linux kernel has assigned to the driver
- └ the table of all the CAN devices the driver has detected (one per line)

The columns of the PC CAN interfaces table are properties that are common to each interface:

Column	PC CAN interface property description	
n	decimal value	The minor number the driver has assigned to that PC CAN interface
type	pci	PCI/PCIe/PCC/EC based interface equipped with a physical or FPGA SJA1000 or controller
	isa	ISA based interface equipped with a SJA1000 controller
	sp	Standard Parallel interface equipped with a SJA1000 controller
	epp	Enhanced Parallel interface equipped with a SJA1000 controller
	usb	USB interface equipped with a SJA1000 controller (PCAN-USB, PCAN-USB Pro)
	usbfd	USB interface equipped with a CAN FD core (PCAN-USB FD, PCAN-USB Pro FD, PCAN-USB X6, PCAN-Chip USB)
	pcifd	PCI/PCIe-based interface equipped with a CAN FD FPGA (PCAN-PCI Express FD, PCAN-miniPCIe FD)
	usbx1	USB interface equipped with a CAN XL core (PCAN-USB XL)
ndev	canx	If the <i>netdev</i> interface has been selected when building the driver, this column contains the name of the PC CAN interface for the SocketCAN layer
	not applicable	When the driver has been built to run in <i>chardev</i> mode (default mode), then this column is meaningless
base	hexadecimal value	The I/O port used to access the PC CAN interface hardware, if it is a Parallel or an ISA interface
		The I/O base address to access the PC CAN interface hardware in the other cases
		The serial number of the adapter if the PC CAN interface is an USB interface
irq	decimal value	The IRQ number attached to the PC CAN interface, if any
		The device number <i>devid</i> set to the PC CAN interface, if the PC CAN interface is an USB interface

Column	PC CAN interface property description	
btr	hexadecimal value	The nominal bitrate set to the PC CAN interface, following the BTR0BTR1 format of the SJA1000 bitrate register
read	hexadecimal value	Number of CAN/CAN FD frames read from the driver by the applications that have opened this interface
write	hexadecimal value	Number of CAN/CAN FD frames written to the driver by the applications that have opened this interface
irqs	hexadecimal value	Number of interrupts counted by the driver for that PC CAN interface (when the driver has connected a handler to an IRQ level) Number of packets received by the driver from the USB subsystem, in case of a USB CAN interface
errors	hexadecimal value	Number of errors encountered by the driver for this interface. This counter handles all kinds of errors (controller errors as well as driver internal errors). Some more information about errors is given in the <code>status</code> column
status	bit mask	The signification of each error bit is defined by the <code>CAN_ERR_xxx</code> constants defined in <code>/usr/include/pcan.h</code> .

Table 5: `/proc/pcan` columns

4.4 `/sysfs` Interface



Note: This feature is new since v8.x of the driver.

For historical reasons, v9.x of the driver always handles the `/proc/pcan` file, but it should be considered as deprecated and for CAN 2.0 usage only. Since v8.x, the driver also exports all the `/proc/pcan` properties (and some more) to the `/sysfs` interface.

- a) The `/sys/class/pcan/version` attribute exports the driver version number:

```
$ cat /sys/class/pcan/version
9.0.0
```

- b) Since v8.11.0, the `/sys/class/pcan/clk_ref` attribute exports the clock reference used by the driver:

```
$ cat /sys/class/pcan/clk_ref
0
```

This numeric value corresponds to some `CLOCK_xxx` values defined in `/usr/include/linux/time.h`:

Value	Mnemonic	Description
0	<code>CLOCK_REALTIME</code>	This clock is affected by discontinuous jumps in the system time (e.g., if the system administrator manually changes the clock), and by the incremental adjustments performed by <code>adjtime(3)</code> and NTP.
1	<code>CLOCK_MONOTONIC</code>	This clock is not affected by discontinuous jumps in the system time (e.g., if the system administrator manually changes the clock), but is affected by the incremental adjustments performed by <code>adjtime(3)</code> and NTP.
4	<code>CLOCK_MONOTONIC_RAW</code>	Similar to <code>CLOCK_MONOTONIC</code> , but provides access to a raw hardware-based time that is not subject to NTP adjustments or the incremental adjustments performed by <code>adjtime(3)</code> .
7	<code>CLOCK_BOOTTIME</code>	Identical to <code>CLOCK_MONOTONIC</code> , except it also includes any time that the system is suspended. This allows applications to get a suspend-aware monotonic clock without having to deal with the complications of <code>CLOCK_REALTIME</code> , which may have discontinuities if the time is changed using <code>settimeofday(2)</code> .

Table 6: clock reference used by the driver for the timestamps

- c) The `/sys/class/pcan` directory exports the list of all the CAN interfaces it handles:

```
$ tree -a /sys/class/pcan
/sys/class/pcan
├── clk_ref
├── pcanpci6 -> ../../devices/virtual/pcan/pcanpci6
├── pcanpci7 -> ../../devices/virtual/pcan/pcanpci7
├── pcanpcifd0 -> ../../devices/virtual/pcan/pcanpcifd0
├── pcanpcifd1 -> ../../devices/virtual/pcan/pcanpcifd1
├── pcanpcifd2 -> ../../devices/virtual/pcan/pcanpcifd2
├── pcanpcifd3 -> ../../devices/virtual/pcan/pcanpcifd3
├── pcanpcifd4 -> ../../devices/virtual/pcan/pcanpcifd4
├── pcanpcifd5 -> ../../devices/virtual/pcan/pcanpcifd5
├── pcanusb34 -> ../../devices/.../pcan/pcanusb34
├── pcanusbfd32 -> ../../devices/.../pcan/pcanusbfd32
├── pcanusbfd33 -> ../../devices/.../pcan/pcanusbfd33
├── pcanusbfd35 -> ../../devices/.../pcan/pcanusbfd35
├── pcanusbxl36 -> ../../devices/.../pcan/pcanusbxl36
├── pcanusbxl37 -> ../../devices/.../pcan/pcanusbxl37
└── version
```

- d) These entries have been extended to export some PCAN devices private properties, as shown (**bold**) in the example below (**bold-green** lines properties are the same as the columns of `/proc/pcan`):

```
$ ls -l /sys/class/pcan/pcanpci6/
total 0
-r--r--r-- 1 root root 4096 nov. 6 12:34 adapter_name
-r--r--r-- 1 root root 4096 nov. 6 12:34 adapter_number
-r--r--r-- 1 root root 4096 nov. 6 12:34 adapter_partnum
-r--r--r-- 1 root root 4096 nov. 6 12:34 adapter_version
-r--r--r-- 1 root root 4096 nov. 6 12:34 base
-r--r--r-- 1 root root 4096 nov. 6 12:34 btr0btr1
-r--r--r-- 1 root root 4096 nov. 6 12:34 bus_state
-r--r--r-- 1 root root 4096 nov. 6 12:34 clk_drift
-r--r--r-- 1 root root 4096 nov. 6 12:34 clock
-r--r--r-- 1 root root 4096 nov. 6 12:34 ctrlr_number
-r--r--r-- 1 root root 4096 nov. 6 12:34 dev
-rw-r--r-- 1 root root 4096 nov. 6 12:34 devid
-rw-r--r-- 1 root root 4096 nov. 6 12:34 dev_name
-r--r--r-- 1 root root 4096 nov. 6 12:34 errors
-r--r--r-- 1 root root 4096 nov. 6 12:34 hwtype
-r--r--r-- 1 root root 4096 nov. 6 12:34 init_flags
-r--r--r-- 1 root root 4096 nov. 6 12:34 irq
-r--r--r-- 1 root root 4096 nov. 6 12:34 irqs
-r--r--r-- 1 root root 4096 nov. 6 12:34 minor
-r--r--r-- 1 root root 4096 nov. 6 12:34 ndev
-r--r--r-- 1 root root 4096 nov. 6 12:34 nom_bitrate
-r--r--r-- 1 root root 4096 nov. 6 12:34 nom_brp
-r--r--r-- 1 root root 4096 nov. 6 12:34 nom_sjw
-r--r--r-- 1 root root 4096 nov. 6 12:34 nom_tq
-r--r--r-- 1 root root 4096 nov. 6 12:34 nom_tseg1
-r--r--r-- 1 root root 4096 nov. 6 12:34 nom_tseg2
drwxr-xr-x 2 root root 0 nov. 6 12:34 power
-r--r--r-- 1 root root 4096 nov. 6 12:34 read
-r--r--r-- 1 root root 4096 nov. 6 12:34 rx_error
-r--r--r-- 1 root root 4096 nov. 6 12:34 rx_fifo_ratio
-r--r--r-- 1 root root 4096 nov. 6 12:34 rx_frames
-r--r--r-- 1 root root 4096 nov. 6 12:34 rx_irqs
```

```
-r--r--r-- 1 root root 4096 nov. 6 12:34 serialno
-r--r--r-- 1 root root 4096 nov. 6 12:34 status
lrwxrwxrwx 1 root root 0 nov. 6 12:34 subsystem -> ../../../../class/pcan
-r--r--r-- 1 root root 4096 nov. 6 12:34 ts_mode
-r--r--r-- 1 root root 4096 nov. 6 12:34 tx_errors
-r--r--r-- 1 root root 4096 nov. 6 12:34 tx_fifo_ratio
-r--r--r-- 1 root root 4096 nov. 6 12:34 tx_frames
-r--r--r-- 1 root root 4096 nov. 6 12:34 tx_irqs
-r--r--r-- 1 root root 4096 nov. 6 12:34 type
-rw-r--r-- 1 root root 4096 nov. 6 12:33 uevent
-r--r--r-- 1 root root 4096 nov. 6 12:34 write
```

e) Reading the content of all the above files will display something like that:

```
$ for f in /sys/class/pcan/pcanpci6/*; do [ -f $f ] && echo -n "`basename $f` = " && cat $f; done
adapter_name = PCAN-PCI Express
adapter_number = 0
adapter_partnum = IPEH-003027
adapter_version = 1.4.0
base = 0xf7900000
btr0btrl = 0x001c
bus_state = 0
clk_drift = 0
clock = 8000000
ctrlr_number = 0
dev = 508:6
devid = 4294967295
dev_name = /dev/pcanpci6
errors = 0
hwtype = 10
init_flags = 0x00000000
irq = 17
irqs = 0
minor = 6
nom_bitrate = 500000
nom_brp = 1
nom_sjw = 1
nom_sp = 8750
nom_tq = 125
nom_tseg1 = 13
nom_tseg2 = 2
read = 0
rx_errors = 0
rx_fifo_ratio = 0.00
rx_frames = 0
rx_irqs = 0
serialno = 4294967295
status = 0x0000
ts_mode = 0
tx_errors = 0
tx_fifo_ratio = 0.00
tx_frames = 0
tx_irqs = 0
type = pci
uevent = MAJOR=MMM
MINOR=6
DEVNAME=pcanpci6
write = 0
```

Note: Depending on the CAN interface hardware and/or the mode the driver has been compiled, the device node might export some more properties. For example, a CAN FD PCIe device will export the following properties (specific properties are **bold**):

```
$ for f in /sys/class/pcan/pcanpcifd1/*; do [ -f $f ] && echo -n "`basename $f`
= " && cat $f; done
adapter_name = PCAN-PCIE FD
adapter_number = 0
adapter_partnum = IPEH-004040
adapter_version = 3.5.7
base = 0x423a2000
btr0btr1 = 0x001c
bus_load = 0.00
bus_state = 0
clk_drift = 0
clock = 80000000
ctrlr_number = 1
dev = 508:1
devid = 4294967295
dev_name = /dev/pcanpcifd1
errors = 0
fd_bitrate = 2000000
fd_brp = 1
fd_sjw = 5
fd_sp = 7500
fd_tq = 12
fd_tseg1 = 29
fd_tseg2 = 10
hwtype = 19
init_flags = 0x00000004
irq = 39
irqs = 0
minor = 1
nom_bitrate = 500000
nom_brp = 1
nom_sjw = 1
nom_sp = 8750
nom_tq = 12
nom_tseg1 = 139
nom_tseg2 = 20
read = 0
rx_dma_laddr = 0x110da4000
rx_dma_vaddr = 00000000946b6e66
rx_errors = 0
rx_fifo_ratio = 0.00
rx_frames = 0
rx_irqs = 0
serialno = 4294967295
status = 0x0000
ts_mode = 2
tx_dma_laddr = 0x1080d1000
tx_dma_vaddr = 0000000074077ef3
tx_errors = 0
tx_fifo_ratio = 0.00
tx_frames = 0
tx_irqs = 0
type = pcifd
uevent = MAJOR=MMM
MINOR=1
DEVNAME=pcanpcifd1
```



```
write = 0
```

- f) Some of the entries are exposed with Write permission. These entries can be written but with root privileges only:

For example, attaching his own device number to a CAN channel is (also) possible through `sysfs`:

```
$ cat /sys/class/pcan/pcanusb34/devid
4294967295
$ echo 12 | sudo tee /sys/class/pcan/pcanusb34/devid
[sudo] password for user:
12
$ cat /sys/class/pcan/pcanusb34/devid
12
```

- Note:** Since v8.10, it is also possible to identify a CAN channel through `sysfs`: by writing a millisecond delay to the “led” property, then the channel LED will blink during this delay. For example, to switch on the LED of “/dev/pcanusb34” during 3 s:

```
$ echo 3000 | sudo tee /sys/class/pcan/pcanusb34/led
```

- g) Since driver version 8.19, and for devices that support it, it is also possible to read the GUID attached to the device. This GUID is a 128-bit value which, if not 0, uniquely identifies the device itself. The GUID is read-only value:

```
$ cat /sys/class/pcan/pcanusb32/device_guid
1132154b-148a-42f7-b070-c3997b923e14
```

- Note:** If the “device_guid” entry is not found, please contact support to update your device with the latest firmware.

4.5 lspcan Tool

- Note:** This feature is new since v8.x of the driver.

The `lspcan` tool is a shell script based on the `/sysfs` interface that can be used to get an overview of the PC CAN interfaces and CAN channels of the host.

```
$ lspcan --help
lspcan: list PEAK-System CAN 2.0/CANFD/CANXL devices handled by the PCAN driver

Option:
-a | --all           equivalent to: -i -s
-f | --forever       forever loop on devices (^C to stop)
-h | --help          display this help
-i | --info          information about PCAN devices
-s | --stats         statistics about PCAN devices
-t | --title         display a title line over columns
-T | --tree          tree version
-v | --version       display PCAN version
```

The "-i" option displays static properties of devices nodes:

```
$ lspcan -T -t -i
dev name      port  irq  clock      bus      btrs
[PCAN-PCI Express 0]
|_ pcanpci6    CAN1  41   8MHz      CLOSED   500k
|_ pcanpci7    CAN2  41   8MHz      CLOSED   500k
[PCAN-PCI Express FD 0]
|_ pcanpcifd0  CAN1  39   80MHz     CLOSED   500k+2M
|_ pcanpcifd1  CAN2  39   80MHz     CLOSED   500k+2M
|_ pcanpcifd2  CAN3  39   80MHz     CLOSED   500k+2M
|_ pcanpcifd3  CAN4  39   80MHz     CLOSED   500k+2M
[PCAN-PCI Express FD 1]
|_ pcanpcifd4  CAN1  40   80MHz     CLOSED   500k+2M
|_ pcanpcifd5  CAN2  40   80MHz     CLOSED   500k+2M
[PCAN-USB 0]
|_ pcanusb34   CAN1  -    8MHz      CLOSED   500k
[PCAN-USB Pro FD 0]
|_ pcanusbfd32 CAN1  -    80MHz     CLOSED   500k+2M
|_ pcanusbfd33 CAN2  -    80MHz     CLOSED   500k+2M
[PCAN-USB FD 0]
|_ pcanusbfd35 CAN1  -    80MHz     CLOSED   500k+2M
[PCAN-USB XL 0]
|_ pcanusbx136 CAN1  -    160MHz    CLOSED   500k+2M+5M
[PCAN-USB XL 1]
|_ pcanusbx137 CAN1  -    160MHz    CLOSED   500k+2M+5M
```

On the other hand, running `lspcan` with `-T -t -s -f` refreshes the screen every second with a detailed view of statistics collected from all the PC CAN interfaces present on the Linux host:

```
dev name      port  bus      %bus  rx %fifo tx %fifo  err
[PCAN-PCI Express 0]
|_ pcanpci6    CAN1  CLOSED   -     0 0.00 0 0.00  0
|_ pcanpci7    CAN2  CLOSED   -     0 0.00 0 0.00  0
[PCAN-PCI Express FD 0]
|_ pcanpcifd0  CAN1  CLOSED   0.00  0 0.00 0 0.00  0
|_ pcanpcifd1  CAN2  CLOSED   0.00  0 0.00 0 0.00  0
|_ pcanpcifd2  CAN3  CLOSED   0.00  0 0.00 0 0.00  0
|_ pcanpcifd3  CAN4  CLOSED   0.00  0 0.00 0 0.00  0
[PCAN-PCI Express FD 1]
|_ pcanpcifd4  CAN1  CLOSED   0.00  0 0.00 0 0.00  0
|_ pcanpcifd5  CAN2  CLOSED   0.00  0 0.00 0 0.00  0
[PCAN-USB 0]
|_ pcanusb34   CAN1  CLOSED   -     0 0.00 0 0.00  0
[PCAN-USB Pro FD 0]
|_ pcanusbfd32 CAN1  CLOSED   0.00  0 0.00 0 0.00  0
|_ pcanusbfd33 CAN2  CLOSED   0.00  0 0.00 0 0.00  0
[PCAN-USB FD 0]
|_ pcanusbfd35 CAN1  CLOSED   0.00  0 0.00 0 0.00  0
[PCAN-USB XL 0]
|_ pcanusbx136 CAN1  CLOSED   -     0 0.00 0 0.00  0
[PCAN-USB XL 1]
|_ pcanusbx137 CAN1  CLOSED   -     0 0.00 0 0.00  0
```



Note: The content of the above screen copies may change, depending on the version of the driver.

4.6 pcanosdiag.sh Tool

Starting from v8.14, the pcan driver package includes and installs another tool named pcanosdiag.sh. When launched with root rights, this Shell script produces a log file that takes a snapshot of the running Linux host.

```
$ sudo pcanosdiag.sh
[sudo] password for xxx:
/usr/local/bin/pcanosdiag.sh v1.1.0
Done.
Please send /tmp/pcanosdiag-1.1.0-20250911_155543.log to <support.peak@hms-networks.com>
```

The output log file can be useful to assist in the diagnosis in certain situations.

4.7 read/write Interface

As described, when reading `/proc/pcan`, once loaded, the driver is ready to operate on the CAN channels it has detected. For each of them, a default bitrates configuration is defined that enables user to read/write from/to the channel. In *chardev* mode, the read/write entries of the driver's *chardev* interface can:

- └ initialize a CAN channel
- └ write CAN 2.0/CAN FD frames
- └ read CAN 2.0/CAN FD frames

This (very) simple interface makes it possible to quickly check if the driver correctly works. This interface uses a syntax made of:

1. a letter that indicates the command
2. a list of parameters for the command



Note: This interface does not handle CAN XL channels!

Command and parameters must be separated by blank characters.

Command	Parameter	Description
i	XXXX	If XXXX is a number ≤ 65535 , then it is interpreted as a BTR0BTR1 SJA1000 register value. The CAN channel is then initialized with the corresponding bitrate value in CAN 2.0 mode only.

Command	Parameter	Description																		
	param1=value1[,param2=value2...]	If the parameter is not a number, then it is parsed as a characters string made of a list of param=value couples. Each couple is separated from the next one by a “,” (comma). The parameters list is: <table><tr><th>Parameter</th><th>Description</th></tr><tr><td>f_clock</td><td>The clock to select</td></tr><tr><td>nom_bitrate</td><td>The nominal bitrate in bit/s.</td></tr><tr><td>nom_brp</td><td rowspan="4">The bit timing specifications for the nominal bitrate, as defined by ISO 11898.</td></tr><tr><td>nom_tseg1</td></tr><tr><td>nom_tseg2</td></tr><tr><td>nom_sjw</td></tr><tr><td>fd_bitrate</td><td>The data bitrate in bit/s. if the CAN channel is to be initialized in CAN FD mode.</td></tr><tr><td>fd_brp</td><td rowspan="4">The bit timing specifications for the data bitrate, as defined by ISO 11898, when the channel is to be initialized in CAN FD mode.</td></tr><tr><td>fd_tseg1</td></tr><tr><td>fd_tseg2</td></tr><tr><td>fd_sjw</td></tr></table> Each value is a numeric value. Unit symbols like k or M can be used as shortcuts. Example: \$ echo "i nom_bitrate=1M" > /dev/pcanusb0 The above command initializes the pcanusb0 CAN channel to connect to a 1 Mbit/s CAN 2.0 channel.	Parameter	Description	f_clock	The clock to select	nom_bitrate	The nominal bitrate in bit/s.	nom_brp	The bit timing specifications for the nominal bitrate, as defined by ISO 11898.	nom_tseg1	nom_tseg2	nom_sjw	fd_bitrate	The data bitrate in bit/s. if the CAN channel is to be initialized in CAN FD mode.	fd_brp	The bit timing specifications for the data bitrate, as defined by ISO 11898, when the channel is to be initialized in CAN FD mode.	fd_tseg1	fd_tseg2	fd_sjw
Parameter	Description																			
f_clock	The clock to select																			
nom_bitrate	The nominal bitrate in bit/s.																			
nom_brp	The bit timing specifications for the nominal bitrate, as defined by ISO 11898.																			
nom_tseg1																				
nom_tseg2																				
nom_sjw																				
fd_bitrate	The data bitrate in bit/s. if the CAN channel is to be initialized in CAN FD mode.																			
fd_brp	The bit timing specifications for the data bitrate, as defined by ISO 11898, when the channel is to be initialized in CAN FD mode.																			
fd_tseg1																				
fd_tseg2																				
fd_sjw																				
m	s id len [xx [xx ...]]	Write CAN standard message id (numeric value <= 0x7ff) with len data bytes valued by xx [xx]. Example: \$ echo "m s 0x123 3 01 02 03" > /dev/pcanusb0 The above command writes CAN message ID 0x123 with 3 the data bytes “01 02 03” on the CAN bus connected to the 1st CAN port of the USB CAN interface.																		
	e id len [xx [xx ...]]	Write CAN extended message id (numeric value <= 0x3ffffff) with len data bytes valued by xx [xx]. Example: \$ echo "m e 0x123 3 01 02 03" > /dev/pcanusb0 The above command writes CAN message ID 0x00000123 with 3 the data bytes “01 02 03” on the CAN bus connected to the 1st CAN port of the USB CAN interface.																		
r	s id	Write the CAN RTR (Remote Transmission Request) of standard id (numeric value <= 0x7ff).																		
	e id	Write the CAN RTR (Remote Transmission Request) of extended id (numeric value <= 0x7ff).																		
M	Same as m but asking the driver to activate the self-receive feature (if the CAN controller of the given channel can copy an outgoing CAN frame to its own rx queue).																			
R	Same as r but asking the driver to activate the self-receive feature (if the CAN controller of the given channel can copy an outgoing CAN frame to its own rx queue).																			
b	Same as m but asking the driver to activate the BRS feature (if the given channel is equipped with a CAN FD controller).																			
B	Same as b but asking the driver to activate the self-receive feature (if the CAN FD controller of the given channel can copy an outgoing CAN FD frame to its own rx queue).																			

Table 7: read/write interface syntax

If reading from this interface, the user can receive any of the above messages, plus status (x) messages:

Message	Parameter	Description	
x	b id len [xx [xx ...]]	Bus status message indicating CAN bus state:	
		id	Bus State
		1	ACTIVE
		2	WARNING
		3	PASSIVE
	4	BUSOFF	
	c id len [xx [xx ...]]	Controller error/status:	
		id	Error
		5	Controller Rx queue empty
		6	Controller Rx queue overflow
7		Controller Tx queue empty	
8	Controller Tx queue overflow		
i id len [xx [xx ...]]	Internal (driver) error/status.		
	id	Error	
	5	Driver Rx queue empty	
	6	Driver Rx queue overflow	
	7	Driver Tx queue empty	
8	Driver Tx queue overflow		

Table 8: Status (x) message

4.8 CAN 2.0/CAN FD test Directory

The PCAN Driver for Linux package includes a `test` directory that contains the C/C++ sources and Makefile enabling user to quickly build and run some simple test binary applications, to check if the *chardev* installation (driver and libraries) is fully operational for CAN 2.0 and CAN FD applications. These test programs also are example programs that demonstrate the usage of the driver library in a non-RT as well as in an RT environment.

Note: Even if these programs can run without any problem with PC CAN-XL interface devices, they provide access to a CAN 2.0 or a CAN FD bus only! The `utils-xl` directory of that package contains examples of specific programs that demonstrate how to access the features offered by the CAN XL standard (see 4.9 CAN XL `utils-xl` Directory).

The `test` directory applications should be built after the libraries (under `lib` directory) have been built.

The global package installation described in 3.1 *Build Binaries* on page 7 has built and installed those binaries in the system. To (re-)build them:

```
$ cd peak-linux-driver-x.y.z
$ make -C test
```

32-bit version:

Since driver version 8.3, a 64-bit version of the PCAN driver can operate with any 32-bit application. To build the 32-bit version of the applications stored in this `test` directory, you need to do:

```
$ cd peak-linux-driver-x.y.z
$ make -C test a1132
```

Note: 32-bit versions of `libpcan/libpcanfd/libpcanxl` libraries must have been built first (see *Making the 32-bit version of the library* on page 8). Moreover, to build any 32-bit application while

running a 64-bit Kernel, you first need to install the `gcc-multilib` package. Finally, the specific `libpopt` 32-bit package must be installed to:

```
$ sudo apt-get install gcc-multilib
$ sudo apt-get install libpopt-dev:i386
```

Real-time versions:

A user who wants to rebuild the RT version of these binaries will have to:

```
$ cd peak-linux-driver-x.y.z
$ make -C test RT=XENOMAI # Or "make xeno" since pcan 8.2
```

if running a Xenomai RT extended kernel, or

```
$ cd peak-linux-driver-x.y.z
$ make -C test RT=RTAI # Or "make rtai" since pcan 8.2
```

if running a RTAI extended kernel.

Note: Users (as well as developers) of CAN-FD-specific applications can directly have a look at the new `pcanfdtst` application described in 4.8.5 on page 34.

4.8.1 receivetest

This application writes all frames it receives from a given CAN 2.0 channel (only!) to `stdout`. This application also demonstrates the usage of the old `libpcan` CAN 2.0 API in both RT and non-RT environments.

Usage:

```
$ receivetest --help

receivetest Version "Release_20150611_n" (www.peak-system.com)
----- Copyright (C) 2004-2009 PEAK System-Technik GmbH -----
receivetest comes with ABSOLUTELY NO WARRANTY. This is free
software and you are welcome to redistribute it under certain
conditions. For details see attached COPYING file.

receivetest - a small test program which receives and prints CAN messages.
usage: receivetest [-b=BTR0BTR1] [-e] [-?]
                {[-f=device node] | {[-t=type] [-p=port [-i=irq]]}}

options:
-f=device node  path to PCAN device node (default=/dev/pcan0)
-t=type         type of interface (pci, sp, epp, isa, pccard, usb (default=pci)
-p=port         port number if applicable (default=1st port of type)
-i=irq          irq number if applicable (default=irq of 1st port)
-b=BTR0BTR1     bitrate code in hex (default=see /proc/pcan)
-e             accept extended frames (default=standard frames only)
-d=no           donot display received messages (default=yes)
-n=mloop        number of loops to run before exit (default=infinite)
-? or --help    displays this help

receivetest: finished (0): 0 message(s) received
```

Example:

Display up to 100 (extended and standard) messages received from the 1st CAN port of a USB interface connected to a CAN bus at 1 Mbit/s:

```
$ receivetest -f=/dev/pcanusb32 -b=0x14 -e -n=100
```

Note: The bitrate set by this program to this CAN interface is exported by the driver:

```
$ cat /proc/pcan | grep -e "^32"
32 usb      -NA-          3 030 0x0014 00000001 00000000 00000000 00000001 0x0000
$ cat /sys/class/pcan/pcanusb32/nom_bitrate
1000000
$ cat /sys/class/pcan/pcanusb32/btr0btr1
0x0014
```

Note: The RT device doesn't appear under "/dev" while running an RT Linux like Xenomai or RTAI, so RT version of CAN_Open(libpcan) removes the "/dev" prefix from the device name characters string, while pcanfd_open(lipcanfd) DOES NOT. This workaround ONLY works with "/dev/pcanX" device names. Modern versions of Xenomai do export the real-time device model nodes under /dev/rtdm.

4.8.2 transmittest

This application writes all the frames it finds in the given text file to the given CAN 2.0 channel (only!). This application also demonstrates the use of the old libpcan CAN 2.0 API in both RT and non-RT environments.

Usage:

```
$ transmittest --help

transmittest Version "Release_20150610_n" (www.peak-system.com)
----- Copyright (C) 2004-2009 PEAK System-Technik GmbH -----
transmittest comes with ABSOLUTELY NO WARRANTY. This is free
software and you are welcome to redistribute it under certain
conditions. For details see attached COPYING file.

transmittest - a small test program which transmits CAN messages.
usage: transmittest filename
               [-b=BTR0BTR1] [-e] [-r=msec] [-n=max] [-?]
               {[-f=device] | {[-t=type] [-p=port [-i=irq]]}}

filename      mandatory name of message description file.
options:
-f=device     path to PCAN device node (default=/dev/pcan0)
-t=type       type of interface (pci, sp, epp, isa, pccard, usb (default=pci)
-p=port       port number if applicable (default=1st port of type)
-i=irq        irq number if applicable (default=irq of 1st port)
-b=BTR0BTR1   bitrate code in hex (default=see /proc/pcan)
-e            accept extended frames (default=standard frames only)
-r=msec       max time to sleep before transm. next msg (default=no sleep)
-n=loop       number of loops to run before exit (default=infinite)
-? or --help  displays this help

transmittest: finished (0).
```

The file `transmit.txt` is given as an example in the `test` directory. The syntax of this file is quite simple and follows the syntax of the write interface of the driver. The test loops into the transmission of the frames found in the input text file. The number of loops is infinite unless the `-n` option is specified on command line.

Example:

Transmit 100 times all the CAN 2.0 frames described in `transmit.txt` to the 1st CAN port of a USB interface connected to a CAN bus at 1 Mbit/s:

```
$ transmitest transmit.txt -f=/dev/pcanusb32 -b=0x14 -e -n=100
```

Note: The bitrate set by this program to this CAN interface is exported by the driver:

```
$ cat /proc/pcan | grep -e "^32"
32 usb      -NA-          3 030 0x0014 00000001 00000000 00000000 00000001 0x0000
$ cat /sys/class/pcan/pcanusb32/nom_bitrate
1000000
$ cat /sys/class/pcan/pcanusb32/btr0btr1
0x0014
```

Note: The RT device doesn't appear under `/dev` while running an RT Linux like Xenomai or RTAI, so RT version of `CAN_Open(libpcan)` removes the `/dev` prefix from the device name characters string, while `pcanfd_open(libpcanfd)` DOES NOT. This workaround ONLY works with `/dev/pcanX` device names. Modern versions of Xenomai do export the real-time device model nodes under `/dev/rtdm`.

4.8.3 pcan-settings

This application enables user to read/write some specific values from/to the non-volatile memory of some PC CAN interfaces. This feature is useful to the user who wants his hot-pluggable CAN interfaces to always have the same device node name, whatever socket it is plugged on (operating systems devices enumeration rules don't give the same number to the same device, if this device is not plugged to the same socket/bus/port...).

Since driver version 8.8, `pcan-settings` allows any super user to switch specific PC CAN interfaces to "USB Mass Storage Device" mode. This mode is used to easily upgrade these PC CAN interfaces with a new firmware (see also 4.11 *USB Mass Storage Device Mode* on page 58).

Usage:

```
$ pcan-settings --help
Usage: pcan-settings [OPTION...]
  -f, --deviceNode='device file path'    Set path to PCAN device (default:
                                         "/dev/pcan32")
  -s, --SerialNo                          Get serial number
  -d, --DeviceNo[='device number']       Get or set device number
  -v, --verbose                           Make it verbose
  -q, --quiet                             No display at all
  -M, --MSD                              Switch device in Mass Storage Device
                                         mode (root privileges needed)

Help options:
  -?, --help                             Show this help message
  --usage                                Display brief usage message
```


Example:

- Get the serial number of a USB CAN interface:

```
$ pcan-settings -f=/dev/pcanusb32 -s
0x00000003
```

- Set device numbers 30 and 31 for CAN1 and CAN2 of a USB 2xCAN channels interface:

```
$ pcan-settings -f=/dev/pcanusb32 -d 30
$ pcan-settings -f=/dev/pcanusb33 -d 31
```

- Read the device numbers of CAN1 and CAN2 of a USB 2xCAN channels interface:

```
$ pcan-settings -f=/dev/pcanusb32 -d
30
$ pcan-settings -f=/dev/pcanusb33 -d
31
```

When the driver is reloaded, it reads these numbers and exports them to `/sysfs`:

```
$ cat /sys/class/pcan/pcanusb32/devid
30
$ cat /sys/class/pcan/pcanusb33/devid
31
```

Thus, Udev is notified and reads the driver's rules. These default rules say that, if `devid` is not -1, then it should be used to create a symbolic link to the true device node under a directory which name is the adapter name. In this example, if the USB CAN interface is a PCAN-USB Pro, then two symbolic links are created under `/dev/pcan-usb_pro`:

```
$ ls -l /dev/pcan-usb_pro
total 0
drwxr-xr-x 2 root root 11 nov. 8 11:00 0
lrwxrwxrwx 1 root root 11 nov. 8 11:00 devid=30 -> ../pcanusb32
lrwxrwxrwx 1 root root 11 nov. 8 11:00 devid=31 -> ../pcanusb33
```

Note: device numbers can also be defined using the `sysfs` interface (see `/sysfs` Interface on page 25).

Note: since v8.10, a device number can also be set to each CAN channel of the PCAN PCI Express / PCAN-PCIe FD cards.

4.8.4 bitratetest

Note: This application is kept for historical reasons only but, since bitrate values and clock selection are now proposed by the new API to the user space, it is considered as deprecated.

This application displays the BTR0BTR1 values for some well-known bitrate values. The BTR0BTR1 16-bits codification is 8 MHz SJA1000-controller-specific.

Usage:

```
$ bitratetest --help

bitratetest Version "Release_20150617_a" (www.peak-system.com)
----- Copyright (C) 2004-2009 PEAK System-Technik GmbH -----
bitratetest comes with ABSOLUTELY NO WARRANTY. This is free
software and you are welcome to redistribute it under certain
conditions. For details see attached COPYING file.

bitratetest - a small test the calculation of BTR0BTR1 data from PCAN.
usage: bitratetest [-f=device node] [-?]
        -f=device node - path to device file, default=/dev/pcan0
        -? or --help - this help

bitratetest: finished (0).
```

4.8.5 pcanfdtst

This application enables testing the driver, since it can receive/transmit CAN 2.0/CAN FD messages from/to all the device nodes handled by the driver. It works in several modes:

- └ When running in RX mode, the application writes everything received from all the opened device nodes on the screen. It is also able to check the content of the incoming frames
- └ When running in TX mode, the application transmits CAN 2.0/CAN FD frames on all the opened devices and displays any event received from them
- └ When running in record mode, the application records the CAN 2.0/CAN FD frames in a local file instead of transmitting them. Recording frames allows users to play the same scenario several times (see `-play` option below)

Moreover, this application demonstrates the usage of the `libpcanfd` API of the driver in both RT and non-RT Linux. Among all the novelties, the application allows to:

- └ specify nominal and data bitrates for CAN FD usage
- └ select the device clock
- └ select ISO and non-ISO CAN FD modes
- └ demonstrate the usage of the new entry points of the new API that enable applications to transmit and receive several messages at once
- └ demonstrate the new event-based API
- └ get and set some device specific options value
- └ play (i.e. transmit) frames from a recorded file

Usage:

```

$ pcanfdtst --help
Setup CAN[FD] tests between CAN channels over the pcan driver (>= v8.x)

WARNING
This application comes with ABSOLUTELY NO WARRANTY. This is free
software and you are welcome to redistribute it under certain
conditions. For details, see attached COPYING file.

USAGE
$ pcanfdtst MODE [OPTIONS] FILE [FILE...]

MODE
tx      generate CAN traffic on the specified CAN interfaces
rx      check CAN traffic received on the specified CAN interfaces
getopt  get a specific option value from the given CAN interface(s)
setopt  set an option value to the given CAN interface(s)
rec     same as 'tx' but frames are recorded into the given file

FILE
For all modes except 'rec' mode:

/dev/pcanx    indicate which CAN interface is used in the test.
               Several CAN interfaces can be specified. In that case,
               each one is opened in non-blocking mode.

'rec' mode only:

file_name     file path in which frames have to be recorded.

OPTIONS
-l | --one-shot          select one-shot mode
-a | --accept f-t        add message filter [f...t]
-b | --bitrate v         set [nominal] bitrate to "v" bps
                        --btr0btr1    bitrates with BTR0BTR1 format
-B | --brs               set BRS bit in outgoing CAN FD frames
-c | --clock v           select clock frequency "v" Hz
-D | --debug              (maybe too) lot of display
-d | --dbitrate v        set data bitrate to "v" bps
                        --dsample-pt v define the data bitrate sample point ratio x 10000
-E | --esi               set ESI bit in outgoing CANFD msgs
                        --echo         tx frame is echoed by the hw into the rx path
-f | --fd                select CAN-FD ISO mode
                        --fd-non-iso   select CAN-FD non-ISO mode
-F | --filler v|r|i|c    select how data are filled:
                        fixed, randomly, incrementally or by the clock
                        --file file    transmit data from/receive data to file
-h | --help              display this help
-i | --id v|r|i          set fixed CAN Id. to "v", randomly or incr.
-is v|r|i                set fixed standard CAN Id "v", randomly or incr.
-ie v|r|i                set fixed extended CAN Id "v", randomly or incr.
-I | --incr v            "v"=nb of data bytes to use for increment counter
-l | --len v|r|i         set fixed CAN dlc "v", randomly or incr.
-m | --mul v             tx/rx "v" msgs at once
-M | --max-duration v    define max duration the test should run in s.
-n v                     send/read "v" CAN msgs then stop
-o | --listen-only       set pcan device in listen-only mode
                        --opt-name v   specify the option name (getopt/setopt modes)
                        --opt-value v  specify the option value (getopt/setopt modes)
                        --opt-size v   specify the option size (getopt/setopt modes)

```

```

-p | --pause-us v      "v" us. pause between sys calls (rx/tx def=0/1000)
  --play file          play recorded frames from "file" according to MODE
  --play-forever       file same as --play but loop forever on "file"
-P | --tx-pause-us v   force a pause of "v" us. between each Tx frame
                        (if hw supports it)
-q | --quiet           nothing is displayed
-r | --rtr             set the RTR flag to msgs sent
  --no-rtr            clear the RTR flag from msgs sent
-s | --stdmsg-only     don't handle extended msgs
  --sample-pt v        define the bitrate sample point ratio x 10000
-T | --check-ts        check host vs. driver timestamps, stop if wrong
  --ts-base v          set timestamp base [0..2]
  --ts-mode v          set hw timestamp mode to v (hw dependant)
-u | --bus-load        get bus load notifications from the driver
-v | --verbose         things are (very much) explained
-w | --with-ts         logs are prefixed with time of day (s.us)
+FORMAT               output line format:
                        %t timestamp (s.us format)
                        %d direction (< or >)
                        %n device node name
                        %i CAN Id. (hex format)
                        %f flags
                        %l data length
                        %D data bytes
                        (default format is: "%t %n %d %i %f %l - %D")

```

- └─ `opt-name`, `opt-value` and `opt-size` parameters are only used when in `getopt` or `setopt` mode only. These options enable user to get or set devices global or specific options value (see also `int pcanfd_get_option()`)
- └─ Bitrates and clock values can be expressed with ending `k` or `M` as shortcuts for factor 1,000 or factor 1,000,000. Note that if the option `--btr0btr1` is used, then `bitrate` and `dbitrate` options value is read as a `BTR0BTR1` format coded value.
- └─ The unit of the pause delay between each write or read system call is the microsecond. Here, using an `m` appended to a value (e.g. 5m) changes to milliseconds and an appended `s` to full seconds (e.g. 7s).
- └─ The unit of the `timeout-ms` parameter is millisecond. Appending an `s` to the value switches to seconds (e.g. 7s).
- └─ If only one PC CAN interface is given on the command line, the application runs in "blocking" mode, that is, the application task blocks into the driver while the receive queue of the driver is empty, or while the transmission queue of the driver is full.
- └─ If more than one PC CAN interface is given on the command line, the application does the following:
 - It runs in non-blocking mode and uses the `select()` system call in non-RT environment, to be able to wait for several events at once.
 - It creates as many real-time tasks as given device nodes, to be able to wait for several events at the same time.
- └─ The application's default behavior is to read/write messages from/to the driver one by one. When the `--mul x` option is used (with `x > 1`), then the application reads/writes `x` messages at once.
- └─ The `+` option is a character's string that runs like the Linux Shell command "date": it enables to specify his own format of the output lines.
- └─ `--ts-base` option allows user to set the base of the timestamps of the frames the driver received:

--ts-base	Description
0	Timestamps are based on the host time (default)
1	Timestamps are based on the time when the device has been opened.
2	Timestamps are based on the time the driver has been loaded.

Some options (like `id`, `len`, `incr`, `filler...`) can be used either in `tx` (or `rec`) mode or in `rx` mode:

- When used in `tx` mode, they control how the transmitted CAN frames are generated
- When used in `rx` mode, they control how the received CAN frames must be.

Examples:

1. Write 10 CAN 2.0 frames (with random ID and data length) each second on a bus with a bitrate of 250 kbit/s using the 2nd USB CAN interface:

```
$ pcanfdtst tx -n 10 -b 250k -p 1s /dev/pcanusb33
0.429301518 /dev/pcanusb33 > BUS_STATE=ACTIVE [Rx:0 Tx:0]
0.4293989212 /dev/pcanusb33 < 567 ..... 7 - 00 00 00 00 00 00 00
1.4293989342 /dev/pcanusb33 < 069 ..... 7 - 00 00 00 00 00 00 00
2.4293989614 /dev/pcanusb33 < 451 ..... 7 - 00 00 00 00 00 00 00
3.4293989798 /dev/pcanusb33 < 44a ..... 3 - 00 00 00
4.4293989995 /dev/pcanusb33 < 729 ..... 1 - 00
5.4293990176 /dev/pcanusb33 < 0ba ..... 4 - 00 00 00 00
6.4293990468 /dev/pcanusb33 < 1f2 ..... 7 - 00 00 00 00 00 00 00
7.4293990660 /dev/pcanusb33 < 1e3 ..... 4 - 00 00 00 00
8.4293990845 /dev/pcanusb33 < 07c ..... 0 -
9.4293991023 /dev/pcanusb33 < 054 ..... 1 - 00
/dev/pcanusb33 < [packets=10 calls=10 bytes=41 eagain=0]
sent frames: 10
```

2. Write CAN FD (non-ISO) frames with extended ID 0x123 and 24 data bytes at a nominal bitrate of 1 Mbit/s and data bitrate of 2 Mbit/s, using the 60 MHz clock of the 2nd USB interface and the 1st PCI interface of the host:

```
$ pcanfdtst tx --fd-non-iso -n 10 -ie 0x123 -l 24 -b 1M -d 2M -c 60M /dev/pcanusbfd33
/dev/pcanpcifd0
0.001871 /dev/pcanusbfd33 > BUS_STATE=ACTIVE [Rx:0 Tx:0]
0.022460 /dev/pcanusbfd33 < 00000123 .e... 24 - 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0.000000 /dev/pcanpcifd0 > BUS STATE=ACTIVE [Rx:0 Tx:0]
0.023558 /dev/pcanpcifd0 < 00000123 .e... 24 - 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0.024662 /dev/pcanusbfd33 < 00000123 .e... 24 - 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0.025754 /dev/pcanpcifd0 < 00000123 .e... 24 - 00 00 00 00 00 00 00 00 00 00 00 00 00 00
...
```

3. Same as above but record (instead of writing) the frames into a file named "test.rec":

```
$ pcanfdtst rec --fd-non-iso -n 10 -ie 0x123 -l 24 test.rec
0.022460 test.rec < 00000123 .e... 24 - 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0.023558 test.rec < 00000123 .e... 24 - 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0.024662 test.rec < 00000123 .e... 24 - 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0.025754 test.rec < 00000123 .e... 24 - 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
...
```

4. Play file "test.rec" writing its frames every 1 s through the 1st channel of a PCAN-PCle FD on a 1Mbps CAN bus:

```
$ pcanfdtst tx --fd-non-iso --play test.rec -b 1M -p 1s /dev/pcanpcifd0
0.022460 /dev/pcanpcifd0 < 00000123 .e... 24 - 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
1.022460 /dev/pcanpcifd0 < 00000123 .e... 24 - 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
2.022460 /dev/pcanpcifd0 < 00000123 .e... 24 - 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
3.022460 /dev/pcanpcifd0 < 00000123 .e... 24 - 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
...
```

5. Read the same bus, but from the 1st USB interface:

```
$ pcanfdtst rx --fd-non-iso -b 1M -d 2M -c 60M /dev/pcanusbfd32
0.001848 /dev/pcanusb32 > BUS_STATE=ACTIVE [Rx:0 Tx:0]
14.761845 /dev/pcanusb32 > 00000123 .e... 24 - 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
14.764041 /dev/pcanusb32 > 00000123 .e... 24 - 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
14.766249 /dev/pcanusb32 > 00000123 .e... 24 - 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
...
```

6. Transmit frames but use the new entry point of the multi-messages write API. Here, the application transmits 3 copies of the same frame:

```
$ /pcanfdtst tx --fd-non-iso -n 10 --mul 3 -ie 0x123 -I 4 -b 1M -d 2M -c 60M /dev/pcanpcifd0
0.000000 /dev/pcanpcifd0 > BUS_STATE=ACTIVE [Rx:0 Tx:0]
0.000283 /dev/pcanpcifd0 < 00000123 .e... 4 - 00 00 00 00
0.001426 /dev/pcanpcifd0 < 00000123 .e... 4 - 01 00 00 00
0.002528 /dev/pcanpcifd0 < 00000123 .e... 4 - 02 00 00 00
0.003675 /dev/pcanpcifd0 < 00000123 .e... 4 - 03 00 00 00
0.005042 /dev/pcanpcifd0 < 00000123 .e... 4 - 04 00 00 00
0.006147 /dev/pcanpcifd0 < 00000123 .e... 4 - 05 00 00 00
0.007252 /dev/pcanpcifd0 < 00000123 .e... 4 - 06 00 00 00
0.008349 /dev/pcanpcifd0 < 00000123 .e... 4 - 07 00 00 00
0.009457 /dev/pcanpcifd0 < 00000123 .e... 4 - 08 00 00 00
0.010564 /dev/pcanpcifd0 < 00000123 .e... 4 - 09 00 00 00
/dev/pcanpcifd0 < [packets=30 calls=10 bytes=120 eagain=0]
sent frames: 30
```

When reading on the same bus, you can see that the driver has written each frame 3 times:

```
$ pcanfdtst rx --fd-non-iso -b 1M -d 2M -c 60M /dev/pcanusbfd32
0.001802 /dev/pcanusbfd32 > BUS STATE=ACTIVE [Rx:0 Tx:0]
8.714190 /dev/pcanusbfd32 > 00000123 .e... 4 - 00 00 00 00
8.714307 /dev/pcanusbfd32 > 00000123 .e... 4 - 00 00 00 00
8.714424 /dev/pcanusbfd32 > 00000123 .e... 4 - 00 00 00 00
8.714540 /dev/pcanusbfd32 > 00000123 .e... 4 - 01 00 00 00
8.714656 /dev/pcanusbfd32 > 00000123 .e... 4 - 01 00 00 00
8.714772 /dev/pcanusbfd32 > 00000123 .e... 4 - 01 00 00 00
8.715402 /dev/pcanusbfd32 > 00000123 .e... 4 - 02 00 00 00
8.715518 /dev/pcanusbfd32 > 00000123 .e... 4 - 02 00 00 00
8.715634 /dev/pcanusbfd32 > 00000123 .e... 4 - 02 00 00 00
8.716552 /dev/pcanusbfd32 > 00000123 .e... 4 - 03 00 00 00
8.716668 /dev/pcanusbfd32 > 00000123 .e... 4 - 03 00 00 00
...
```



Note: The RT device doesn't appear under `/dev` while running an RT Linux like Xenomai or RTAI, so the RT version of `pcanfdtst` **MUST** use the real name of the CAN device, that is `pcanX` or `/dev/rtdm/pcanX`. There are neither aliases nor links that Udev can make when a RT device is created.

7. Changing the device id. of a PCAN-USB FD device using the `getopt/setopt` modes:

```
$ pcanfdtst getopt --opt-name 1 /dev/pcanusbfd32
/dev/pcanusbfd32 > [option=1 size=4 value=[ff ff ff ff ]
$ pcanfdtst setopt --opt-name 1 --opt-value 0xEFBEADDE --opt-size 4
/dev/pcanusbfd32
/dev/pcanusbfd32 > [option=1 size=4 value=[de ad be ef ]
$ pcanfdtst getopt --opt-name 1 /dev/pcanusbfd32
/dev/pcanusbfd32 > [option=1 size=4 value=[de ad be ef ]
```

8. Getting the version of the firmware running on a PCAN-USB FD adapter:

```
$ pcanfdtst getopt --opt-name 11 /dev/pcanusbfd32
/dev/pcanusbfd32 > [option=11 size=4 value=[00 00 02 03 ]
```

4.9 CAN XL `utils-xl` Directory

The PCAN Driver v9 package for Linux includes the `utils-xl` directory, which contains the C source files and the Makefile needed to quickly compile and run a few simple binary application examples to verify that the complete *chardev* mode installation (driver and libraries) is fully operational for CAN CC/FD/XL. These example programs illustrate the use of the (new) `libpcanxl` library from the package, which allows access to the CAN bus in CAN CC, CAN FD, or CAN XL mode.

By default, these applications are built to operate in full CAN XL mode. However, all of them can be re-compiled to operate in CAN CC mode only:

```
$ cd peak-linux-driver-x.y.z
$ make -C utils-xl clean can_cc_only
```

Or, in CAN FD mode only:

```
$ cd peak-linux-driver-x.y.z
$ make -C utils-xl clean can_fd_only
```

4.9.1 pcanxl-loop

This program allows you to easily evaluate the complete installation of the PC CAN interface whether it is CAN CC, CAN FD, or CAN XL.

`pcanxl-loop` configures the CAN interface in loopback mode so that it returns the frame(s) sent to it by the program. By using the “`--slfack`” option, you can even be alone on the bus, with no need for other equipment to acknowledge your frames.

When `pcanxl-loop` is built to configure the PC CAN interface in CAN-XL:

```
$ pcanxl-loop -h
Writes CAN frames and waits for their echo.

USAGE:
pcanxl-loop [OPTIONS] /dev/pcanX

OPTIONS:
-h | --help          display this help
-d | --nolog         set discreet mode
-c | --count v       Number of CAN frames to process (0: nolimit, default: 1)
-l | --datalen v     CAN frame data length specification (default: 2048 Bytes)
-q | --clock v       Clock specification in MHz (default: 0 MHz)
-s | --slfack        set self-ack mode (default: off)
-n | --nombtr v      nominal bitrate btr[:sp] (default: 1000K:8000)
-f | --fdbtr v       CANFD data bitrate btr[:sp] (default: 4M:0000)
-x | --xlbtr v       CANXL data bitrate btr[:sp] (default: 10M:5620)
-t | --tms v         CANXL Transceiver Mode Switch on|off (default: on)
-e | --es v          CANXL Error Signaling mode on|off (default: off)
-p | --pwm v         CANXL PWM specification pwms[:pwml[:pwmo]] (default 0:0:0)
```

Then, the PC CAN-XL interface can then be tested when it is connected, and only it, to a correctly terminated CAN network by asking `pcanxl-loop` to write one self-acknowledged CAN-XL frame of 100 bytes of data (for example):

```
$ pcanxl-loop -s -l 100 /dev/pcan36
/dev/pcan36: ALLOWED_MSGS option set
/dev/pcan36: init 0MHz core in CAN_XL with flg=00070000h bitrates=[nominal=10000
00:8000 xl=10000000:5620] ES=0 TMS=1 pwm[short=0 long=0 offset=0]
/dev/pcan36: device initialized
/dev/pcan36: init settings=[clock=160MHz flags=80070000h bitrate[nominal=1000000
:8000 brp=1 tseg1=127 tseg2=32 sjw=16 xl=10000000:5000 brp=1 tseg1=7 tseg2=8
sjw=4 ssp_offset=4294967295] ES=0 TMS=1 pwm[short=4 long=12 offset=0]]
/dev/pcan36: 1758025924.959265117 > PCANXL_TYPE_STATUS(3): [PCANFD_ERROR_ACTIVE
(1) flg=130000080h (bus,ts,hw,ec)]
/dev/pcan36: 1758025924.972678083 < PCANXL_TYPE_CANXL(5): [PID=0 VCID=0 flg=0100
0004h (slf,ts) len=100]

00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E
0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D
1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C
2D 2E 2F 30 31 32 33 34 35 36 37 38 39 3A 3B
3C 3D 3E 3F 40 41 42 43 44 45 46 47 48 49 4A
```



```

4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59
5A 5B 5C 5D 5E 5F 60 61 62 63
/dev/pcan36: 1758025924.973909251 > PCANXL_TYPE_CANXL(5): [PID=0 VCID=0 flg=1100
0004h (slf,ts,ec) len=100]
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E
0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D
1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C
2D 2E 2F 30 31 32 33 34 35 36 37 38 39 3A 3B
3C 3D 3E 3F 40 41 42 43 44 45 46 47 48 49 4A
4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59
5A 5B 5C 5D 5E 5F 60 61 62 63
/dev/pcan36: contents are ok!
/dev/pcan36: closing device

```

Like all `utils-xl` applications, `pcanxl-loop` can be compiled in “`can_cc_only`” mode (or “`can_fd_only`” mode) to also operate in CAN CC (or CAN FD) mode with any compatible PC CAN interface. For example:

```

$ cd peak-linux-driver-x.y.z/utils-xl
$ make clean can_fd_only
...
$ ./pcanxl-loop -s -l 32 /dev/pcanusbfd32
/dev/pcanusbfd32: ALLOWED_MSGS option set
/dev/pcanusbfd32: init 0MHz core in CAN_FD with flg=00000004h bitrates=[nominal=
1000000:8000 fd=4000000:0000
/dev/pcanusbfd32: device initialized
/dev/pcanusbfd32: init settings=[clock=80MHz flags=80000004h bitrate[nominal=100
0000:8000 brp=1 tseg1=63 tseg2=16 sjw=8 fd=4000000:7500 brp=1 tseg1=14 tseg2=5 s
jw=2 ssp_offset=0]
/dev/pcanusbfd32: 1758027206.077947039 > PCANXL_TYPE_STATUS(3): [PCANFD_ERROR_AC
TIVE (1) flg=33000080h (bus,ts,hw,ec,bl) bus=0.0%]
/dev/pcanusbfd32: 1758027206.092600480 < PCANXL_TYPE_CANFD(2): [00000000h(0) flg
=01100006h (ext,slf,brs,ts) len=32]
00 01 02 03 04 05 06 07 08 09 0A 0B 0C
0D 0E 0F 10 11 12 13 14 15 16 17 18 19
1A 1B 1C 1D 1E 1F
/dev/pcanusbfd32: 1758027206.092925230 > PCANXL_TYPE_CANFD(2): [00000000h(0) flg
=33100d06h (ext,slf,brs,ts,hw,ec,bl) len=32]
00 01 02 03 04 05 06 07 08 09 0A 0B 0C
0D 0E 0F 10 11 12 13 14 15 16 17 18 19
1A 1B 1C 1D 1E 1F
/dev/pcanusbfd32: contents are ok!
/dev/pcanusbfd32: closing device

```

4.9.2 pcanxl-ping / pcanxl-pong

These programs allow you to easily evaluate the complete installation of two PC CAN interfaces whether they are CAN CC, CAN FD, or CAN XL.

`pcanxl-ping` writes a certain number of CAN frames (CC/FD/XL) to the CAN bus and waits for them to be read back before exiting. In contrast, `pcanxl-pong` responds to a certain number of frames that it receives. Using these two tools, it is easy to check that the CAN bus is operating correctly.

When `pcanxl-ping/pcanxl-pong` are built to configure the PC CAN interface in CAN-XL:

```

$ pcanxl-ping -h
Writes CAN frames on the wire and waits for them.

```

```

USAGE:
pcanxl-ping [OPTIONS] /dev/pcanX

OPTIONS:
-h | --help          display this help
-d | --nolog         set discreet mode
-c | --count v       Number of CAN frames to process (0: nolimit, default: 1)
-l | --datalen v     CAN frame data length specification (default: 2048 Bytes)
-q | --clock v       Clock specification in MHz (default: 0 MHz)
-n | --nombtr v      nominal bitrate btr[:sp] (default: 1000K:8000)
-f | --fdbtr v       CANFD data bitrate btr[:sp] (default: 4M:0000)
-x | --xlbtr v       CANXL data bitrate btr[:sp] (default: 10M:5620)
-t | --tms v         CANXL Transceiver Mode Switch on|off (default: on)
-e | --es v          CANXL Error Signaling mode on|off (default: off)
-p | --pwm v         CANXL PWM specification pwms[:pwml[:pwmo]] (default 0:0:0)

$ pcanxl-pong -h
Waits for CAN frames and writes them back on the wire.

USAGE:
pcanxl-pong [OPTIONS] /dev/pcanX

OPTIONS:
-h | --help          display this help
-d | --nolog         set discreet mode
-c | --count v       Number of CAN frames to process (0: nolimit, default: 1)
-q | --clock v       Clock specification in MHz (default: 0 MHz)
-n | --nombtr v      nominal bitrate btr[:sp] (default: 1000K:8000)
-f | --fdbtr v       CANFD data bitrate btr[:sp] (default: 4M:0000)
-x | --xlbtr v       CANXL data bitrate btr[:sp] (default: 10M:5620)
-t | --tms v         CANXL Transceiver Mode Switch on|off (default: on)
-e | --es v          CANXL Error Signaling mode on|off (default: off)
-p | --pwm v         CANXL PWM specification pwms[:pwml[:pwmo]] (default 0:0:0)

```

Thus, a CAN bus can be tested in CAN XL mode through two PCAN-USB XL interfaces (/dev/pcanusbxl36 and /dev/pcanusbxl37) by exchanging (for example) 3 CAN XL frames of 80 data bytes at 20 Mbps.

From a first Terminal window, launch `pcanxl-pong` (in discreet mode to remove log messages) which connects through a PCAN-USB XL device to the CAN bus:

```
$ pcanxl-pong -d -c 3 -x 20M /dev/pcanusbxl36
```

Then, from the 2nd Terminal window, launch `pcanxl-ping` which connects to the CAN bus through another PCAN-USB XL device:

```

$ pcanxl-ping -l 80 -c 3 -x 20M /dev/pcanusbxl37
/dev/pcanusbxl37: init 0MHz core in CAN_XL with flg=00070000h bitrates=[nominal=1000000:8000 xl=20000000:5620] ES=0 TMS=1 pwm[short=0 long=0 offset=0]
/dev/pcanusbxl37: device initialized
/dev/pcanusbxl37: init settings=[clock=160MHz flags=80070000h bitrate[nominal=1000000:8000 brp=1 tseg1=127 tseg2=32 sjw=16 xl=20000000:5000 brp=1 tseg1=3 tseg2=4 sjw=2 ssp_offset=4294967295] ES=0 TMS=1 pwm[short=2 long=6 offset=0]]
/dev/pcanusbxl37: 1758028377.897555445 > PCANXL_TYPE_STATUS(3):
[PCANFD_ERROR_ACTIVE (1) flg=13000080h (bus,ts,hw,ec)]
/dev/pcanusbxl37: 1758028377.924525826 < PCANXL_TYPE_CANXL(5): [PID=0 VCID=0 flg=01000010h (ech,ts) echo=17 len=80]

```

```

00 01 02 03 04 05 06 07 08 09 0A 0B 0C
0D 0E 0F 10 11 12 13 14 15 16 17 18 19
1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26
27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33
34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40
41 42 43 44 45 46 47 48 49 4A 4B 4C 4D
4E 4F
/dev/pcanusbxl37: 1758028377.925682725 > PCANXL_TYPE_CANXL(5): [PID=0 VCID=0 flg
=01000014h (slf,ech,ts) echo=17 len=80]
00 01 02 03 04 05 06 07 08 09 0A 0B 0C
0D 0E 0F 10 11 12 13 14 15 16 17 18 19
1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26
27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33
34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40
41 42 43 44 45 46 47 48 49 4A 4B 4C 4D
4E 4F
/dev/pcanusbxl37: 1758028377.926838150 > PCANXL_TYPE_CANXL(5): [PID=0 VCID=0 flg
=11000000h (ts,ec) len=80]
00 01 02 03 04 05 06 07 08 09 0A 0B 0C
0D 0E 0F 10 11 12 13 14 15 16 17 18 19
1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26
27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33
34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40
41 42 43 44 45 46 47 48 49 4A 4B 4C 4D
4E 4F
/dev/pcanusbxl37: rtt=0.001155425 s
/dev/pcanusbxl37: contents are ok!
/dev/pcanusbxl37: 1758028377.926878983 < PCANXL_TYPE_CANXL(5): [PID=0 VCID=0 flg
=01000010h (ech,ts) echo=17 len=80]
00 01 02 03 04 05 06 07 08 09 0A 0B 0C
0D 0E 0F 10 11 12 13 14 15 16 17 18 19
1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26
27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33
34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40
41 42 43 44 45 46 47 48 49 4A 4B 4C 4D
4E 4F
/dev/pcanusbxl37: 1758028377.928123404 > PCANXL_TYPE_CANXL(5): [PID=0 VCID=0 flg
=01000014h (slf,ech,ts) echo=17 len=80]
00 01 02 03 04 05 06 07 08 09 0A 0B 0C
0D 0E 0F 10 11 12 13 14 15 16 17 18 19
1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26
27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33
34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40
41 42 43 44 45 46 47 48 49 4A 4B 4C 4D
4E 4F
/dev/pcanusbxl37: 1758028377.929303550 > PCANXL_TYPE_CANXL(5): [PID=0 VCID=0 flg
=11000000h (ts,ec) len=80]
00 01 02 03 04 05 06 07 08 09 0A 0B 0C
0D 0E 0F 10 11 12 13 14 15 16 17 18 19
1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26
27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33
34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40
41 42 43 44 45 46 47 48 49 4A 4B 4C 4D
4E 4F
/dev/pcanusbxl37: rtt=0.001180146 s
/dev/pcanusbxl37: contents are ok!
/dev/pcanusbxl37: 1758028377.929349444 < PCANXL_TYPE_CANXL(5): [PID=0 VCID=0
flg=01000010h (ech,ts) echo=17 len=80]
00 01 02 03 04 05 06 07 08 09 0A 0B 0C
0D 0E 0F 10 11 12 13 14 15 16 17 18 19
1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26

```

```

27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33
34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40
41 42 43 44 45 46 47 48 49 4A 4B 4C 4D
4E 4F
/dev/pcanusbxl37: 1758028377.930538403 > PCANXL_TYPE_CANXL(5): [PID=0 VCID=0
flg=01000014h (slf,ech,ts) echo=17 len=80]
00 01 02 03 04 05 06 07 08 09 0A 0B 0C
0D 0E 0F 10 11 12 13 14 15 16 17 18 19
1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26
27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33
34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40
41 42 43 44 45 46 47 48 49 4A 4B 4C 4D
4E 4F
/dev/pcanusbxl37: 1758028377.931678542 > PCANXL_TYPE_CANXL(5): [PID=0 VCID=0
flg=11000000h (ts,ec) len=80]
00 01 02 03 04 05 06 07 08 09 0A 0B 0C
0D 0E 0F 10 11 12 13 14 15 16 17 18 19
1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26
27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33
34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40
41 42 43 44 45 46 47 48 49 4A 4B 4C 4D
4E 4F
/dev/pcanusbxl37: rtt=0.001140139 s
/dev/pcanusbxl37: contents are ok!
/dev/pcanusbxl37: closing device

```

Note that the `pcanxl-ping` application writes each CAN frame by requesting an echo of that frame from the CAN controller, so that the application will be able to better compute the round-trip time of the frame (see “ech” in `flg` field above) when the response will be read from the bus.

4.9.3 pcanxl-gen / pcanxl-dump

These programs allow you to easily evaluate the complete installation of two PC CAN interfaces whether they are CAN CC, CAN FD, or XL.

`pcanxl-gen` writes a certain number of CAN frames (CC/FD/XL) to the CAN bus while `pcanxl-dump` writes the content of the CAN frames it reads from the bus. Using these two tools, it is easy to check that the CAN bus is operating correctly.

When `pcanxl-gen/pcanxl-dump` are built to configure the PC CAN interface in CAN-XL:

```

$ pcanxl-gen -h
Writes CAN frames on a bus.

USAGE:
pcanxl-gen [OPTIONS] /dev/pcanX

OPTIONS:
-h | --help          display this help
-d | --nolog         set discreet mode
-c | --count v       Number of CAN frames to process (0: nolimit, default: 1)
-l | --datalen v     CAN frame data length specification (default: 2048 Bytes)
-s | --slfack        set self-ack mode (default: off)
-r | --every v       Writing period in us. or 'echo' (default: 1000 us.)
-q | --clock v       Clock specification in MHz (default: 0 MHz)
-n | --nombtr v      nominal bitrate btr[:sp] (default: 1000K:8000)
-f | --fdbtr v       CANFD data bitrate btr[:sp] (default: 4M:0000)
-x | --xlbtr v       CANXL data bitrate btr[:sp] (default: 10M:5620)

```

```
-t | --tms v      CANXL Transceiver Mode Switch on|off (default: on)
-e | --es v      CANXL Error Signaling mode on|off (default: off)
-p | --pwm v      CANXL PWM specification pwms[:pwml[:pwmo]] (default 0:0:0)

$ pcanxl-dump -h
Dumps a CAN bus.

USAGE:
pcanxl-dump [OPTIONS] /dev/pcanX

OPTIONS:
-h | --help          display this help
-c | --count v       Number of CAN frames to process (0: nolimit, default: 0)
-l | --listen-only   Set listen-only initialisation mode (default: off)
-q | --clock v       Clock specification in MHz (default: 0 MHz)
-n | --nombtr v      nominal bitrate btr[:sp] (default: 1000K:8000)
-f | --fdbtr v       CANFD data bitrate btr[:sp] (default: 4M:0000)
-x | --xlbtr v       CANXL data bitrate btr[:sp] (default: 10M:5620)
-t | --tms v        CANXL Transceiver Mode Switch on|off (default: on)
-e | --es v         CANXL Error Signaling mode on|off (default: off)
-p | --pwm v        CANXL PWM specification pwms[:pwml[:pwmo]] (default 0:0:0)
```

Thus, to dump CAN-XL traffic from a bus configured with a data rate of 20 Mbps, type:

```
$ pcanxl-dump -x 20M /dev/pcanusb136
```

And, to generate infinite CAN-XL frame traffic of 1KB of data written every 100 µs on a bus configured with a data rate of 20 Mbps, in discreet mode, type:

```
$ pcanxl-gen -d -r 100 -x 20M -l 1k -c 0 /dev/pcanusb137
```

4.9.4 pcanxl-write / pcanxl-read

These programs allow you to easily evaluate the complete installation of two PC CAN interfaces whether they are CAN CC, CAN FD, or CAN XL.

`pcanxl-write` forwards data bytes read from `stdin` to a CAN (CC/FD/XL) bus it is connected to, through a PC CAN interface, while `pcanxl-read` forwards the CAN (CC/FD/XL) frames it reads to `stdout`. Using these two tools, it is easy to check that the CAN bus is operating correctly.

When `pcanxl-write/pcanxl-read` are built to configure the PC CAN interface in CAN-XL:

```
$ pcanxl-write -h
Forward stdin to a CAN channel.

USAGE:
pcanxl-write [OPTIONS] /dev/pcanX

OPTIONS:
-h | --help          display this help
-d | --nolog         set discreet mode
-l | --datalen v     CAN frame data length specification (default: 2048 Bytes)
-r | --every v       Writing period in us. or 'echo' (default: 0 us.)
-q | --clock v       Clock specification in MHz (default: 0 MHz)
-n | --nombtr v      nominal bitrate btr[:sp] (default: 1000K:8000)
-f | --fdbtr v       CANFD data bitrate btr[:sp] (default: 4M:0000)
```

```
-x | --xlbtr v    CANXL data bitrate btr[:sp] (default: 10M:5620)
-t | --tms v     CANXL Transceiver Mode Switch on|off (default: on)
-e | --es v      CANXL Error Signaling mode on|off (default: off)
-p | --pwm v     CANXL PWM specification pwms[:pwml[:pwmo]] (default 0:0:0)
```

```
$ pcanxl-read -h
Reads CAN frames and writes them to stdout.
```

USAGE:

```
pcanxl-read [OPTIONS] /dev/pcanX
```

OPTIONS:

```
-h | --help          display this help
-d | --nolog         set discreet mode
-l | --listen-only   Set listen-only initialisation mode (default: off)
-q | --clock v       Clock specification in MHz (default: 0 MHz)
-n | --nombtr v      nominal bitrate btr[:sp] (default: 1000K:8000)
-f | --fdbtr v       CANFD data bitrate btr[:sp] (default: 4M:0000)
-x | --xlbtr v       CANXL data bitrate btr[:sp] (default: 10M:5620)
-t | --tms v         CANXL Transceiver Mode Switch on|off (default: on)
-e | --es v          CANXL Error Signaling mode on|off (default: off)
-p | --pwm v         CANXL PWM specification pwms[:pwml[:pwmo]] (default 0:0:0)
```

Thus, to transfer a file from one Linux host to another one, using the CAN bus in CAN XL mode configured with a data bitrate of 20 Mbps, type from the receiving Linux host to which a 1st PCAN-USB XL device is connected (using discreet mode):

```
$ pcanxl-read -d -x 20M /dev/pcanusbxl36 > /tmp/pcan.ko
```

Then, from the sending Linux host, start sending the file content (cut into 2 KB data frames) through the 2nd PCAN-USB XL interface with:

```
$ cat driver/pcan.ko | pcanxl-write -d -x 20M /dev/pcanusbxl37
```

At the end of the transfer, both tools display some statistics, particularly the number of bytes received, as well as the time taken for the transfer (and the measured throughput). In the above example, `pcanxl-read` outputs:

```
$ pcanxl-read -d -x 20M /dev/pcan36 > /tmp/pcan.ko
Got 13168272 bytes in 6.064568266s (17372000 bps)
```

4.9.5 CAN CC/FD/XL Compatibility

The three different CAN CC, FD, and XL protocols used for communication on a CAN bus have been designed to be backward compatible, meaning that communication in CAN CC will not interfere with communication in CAN FD/XL (downward compatibility).

Thanks to the PCAN driver package for Linux, this compatibility can be easily verified, provided you have the appropriate PC CAN interfaces connected to the same CAN bus, such as:

- └ 2 x PCAN-USB for communication in CAN CC
- └ 1 x PCAN-USB Pro FD offering 2 channels capable of operating in CAN FD
- └ 2 x PCAN-USB XL for communication in CAN XL

After installation, the PCAN v9 driver sees the different interfaces as follows:

```
$ cat /proc/pcan
*----- PEAK-System CAN interfaces (www.peak-system.com) -----
*----- Release_YYYYMMDD_n (9.0.0) Sep 17 2025 09:12:19 -----
*----- [mod] [isa] [pci] [pec] [usb] -----
*----- 6 interfaces @ major 506 found -----
*n -type- -ndev- --base-- irq --btr- --read-- --write- --irqs-- -errors- status
32  usb  -NA- ffffffff 001 0x001c 00000000 00000000 00000000 00000000 0x0000
33  usb  -NA- ffffffff 001 0x001c 00000000 00000000 00000000 00000000 0x0000
34  usbfd -NA-      3 002 0x001c 00000000 00000000 00000000 00000000 0x0000
35  usbfd -NA-      3 123 0x001c 00000000 00000000 00000000 00000000 0x0000
36  usbxl -NA- ffffffff 071 0x001c 00000000 00000000 00000000 00000000 0x0000
37  usbxl -NA- ffffffff 072 0x001c 00000000 00000000 00000000 00000000 0x0000
$
$ lspcan -t -T -i
dev name          port  irq   clock      bus      btrs
[PCAN-USB 0]
|_ pcanusb32      CAN1  -    8MHz      ACTIVE    500k
[PCAN-USB 1]
|_ pcanusb33      CAN1  -    8MHz      CLOSED    500k
[PCAN-USB Pro FD 0]
|_ pcanusbfd34    CAN1  -   80MHz     ACTIVE    500k+2M
|_ pcanusbfd35    CAN2  -   80MHz     CLOSED    500k+2M
[PCAN-USB XL 0]
|_ pcanusbxl36    CAN1  -  160MHz    ACTIVE    500k+2M+8M
[PCAN-USB XL 1]
|_ pcanusbxl37    CAN1  -  160MHz    CLOSED    500k+2M+8M
```

Writing CAN CC frames

From 3 different Terminal window; start 3 applications that configure their own CAN controller in the 3 different CAN standards, that is:

1. CAN CC for PCAN-USB #0, with nominal bitrate = 500 Kbps:

```
$ pcanfdtst rx -b 500k /dev/pcanusb32
```

2. CAN FD for PCAN-USB Pro FD CAN 1, with same nominal bitrate and the CAN FD data bitrate = 2 Mbps:

```
$ pcanfdtst rx --fd -b 500k -d 2M /dev/pcanusbfd34
```

3. CAN XL for PCAN-USB XL #0, with same nominal and FD data bitrates, plus the CAN XL data bitrate = 8 Mbps:

```
$ pcanxl-dump -e on -n 500k -f 2M -x 8M /dev/pcanusbxl36
```



Note: For CAN XL to be backward compatible, the error signaling mode must be enabled.

Then, write (say) 10x CAN 2.0 frames with 8 data bytes from a 4th Terminal window through PCAN-USB #1:

```
$ pcanfdtst tx -b 500k -n 10 -i 1 -I 8 /dev/pcanusb33
1758097176~401938 /dev/pcanusb33 > BUS_STATE=ACTIVE [Rx:0 Tx:0] -
1758097176~412089 /dev/pcanusb33 < 001 ..... 8 - 00 00 00 00 00 00 00 00
1758097176~413186 /dev/pcanusb33 < 001 ..... 8 - 01 00 00 00 00 00 00 00
1758097176~414263 /dev/pcanusb33 < 001 ..... 8 - 02 00 00 00 00 00 00 00
1758097176~415349 /dev/pcanusb33 < 001 ..... 8 - 03 00 00 00 00 00 00 00
1758097176~416433 /dev/pcanusb33 < 001 ..... 8 - 04 00 00 00 00 00 00 00
1758097176~417522 /dev/pcanusb33 < 001 ..... 8 - 05 00 00 00 00 00 00 00
1758097176~418605 /dev/pcanusb33 < 001 ..... 8 - 06 00 00 00 00 00 00 00
1758097176~419689 /dev/pcanusb33 < 001 ..... 8 - 07 00 00 00 00 00 00 00
1758097176~420773 /dev/pcanusb33 < 001 ..... 8 - 08 00 00 00 00 00 00 00
1758097176~421850 /dev/pcanusb33 < 001 ..... 8 - 09 00 00 00 00 00 00 00
/dev/pcanusb33 < [packets=10 calls=10 bytes=80 eagain=0 drv_ovr=0 ctr_ovr=0]
Test duration: 0.022087 s. sent frames: 10 (28976 bps)
```

And see how the 3 other applications get correctly these 10x CAN 2.0 frames:

```
$ pcanfdtst rx -b 500k /dev/pcanusb32
1758097160~677506 /dev/pcanusb32 > BUS_STATE=ACTIVE [Rx:0 Tx:0] -
1758097176.412765 /dev/pcanusb32 > 001 ..... 8 - 00 00 00 00 00 00 00 00
1758097176.413875 /dev/pcanusb32 > 001 ..... 8 - 01 00 00 00 00 00 00 00
1758097176.414941 /dev/pcanusb32 > 001 ..... 8 - 02 00 00 00 00 00 00 00
1758097176.416008 /dev/pcanusb32 > 001 ..... 8 - 03 00 00 00 00 00 00 00
1758097176.417075 /dev/pcanusb32 > 001 ..... 8 - 04 00 00 00 00 00 00 00
1758097176.418141 /dev/pcanusb32 > 001 ..... 8 - 05 00 00 00 00 00 00 00
1758097176.419293 /dev/pcanusb32 > 001 ..... 8 - 06 00 00 00 00 00 00 00
1758097176.420360 /dev/pcanusb32 > 001 ..... 8 - 07 00 00 00 00 00 00 00
1758097176.421427 /dev/pcanusb32 > 001 ..... 8 - 08 00 00 00 00 00 00 00
1758097176.422536 /dev/pcanusb32 > 001 ..... 8 - 09 00 00 00 00 00 00 00
```

```
$ pcanfdtst rx --fd -b 500k -d 2M /dev/pcanusbfd34
1758097154.096575 /dev/pcanusbfd34 > BUS_STATE=ACTIVE [Rx:0 Tx:0] -
1758097176.412605 /dev/pcanusbfd34 > 001 ..... 8 - 00 00 00 00 00 00 00 00
1758097176.413699 /dev/pcanusbfd34 > 001 ..... 8 - 01 00 00 00 00 00 00 00
1758097176.414777 /dev/pcanusbfd34 > 001 ..... 8 - 02 00 00 00 00 00 00 00
1758097176.415851 /dev/pcanusbfd34 > 001 ..... 8 - 03 00 00 00 00 00 00 00
1758097176.416929 /dev/pcanusbfd34 > 001 ..... 8 - 04 00 00 00 00 00 00 00
1758097176.417989 /dev/pcanusbfd34 > 001 ..... 8 - 05 00 00 00 00 00 00 00
1758097176.419115 /dev/pcanusbfd34 > 001 ..... 8 - 06 00 00 00 00 00 00 00
1758097176.420185 /dev/pcanusbfd34 > 001 ..... 8 - 07 00 00 00 00 00 00 00
1758097176.421277 /dev/pcanusbfd34 > 001 ..... 8 - 08 00 00 00 00 00 00 00
1758097176.422379 /dev/pcanusbfd34 > 001 ..... 8 - 09 00 00 00 00 00 00 00
```



```
$ pcanxl-dump -e on -n 500k -f 2M -x 8M /dev/pcanusbxl36
/dev/pcanusbxl36: init 0MHz core in CAN_XL with flg=00030004h bitrates=[nominal=500000:8000 fd=20000
00:0000 xl=80000000:5620] ES=1 TMS=1 pwm[short=0 long=0 offset=0]
/dev/pcanusbxl36: device initialized
/dev/pcanusbxl36: init settings=[clock=160MHz flags=80010004h bitrate[nominal=500000:8000 brp=1 tseg
1=255 tseg2=64 sjw=32 fd=2000000:7500 brp=1 tseg1=59 tseg2=20 sjw=10 ssp_offset=0 xl=80000000:5500 br
p=1 tseg1=10 tseg2=9 sjw=4 ssp_offset=0] ES=1 TMS=0]
/dev/pcanusbxl36: 1758097173.530316666 > PCANXL_TYPE_STATUS(3): [PCANFD_ERROR_ACTIVE (1) flg=1300008
0h (bus,ts,hw,ec)]
/dev/pcanusbxl36: 1758097176.413637867 > PCANXL_TYPE_CANCC(1): [001h(1) flg=13000000h (ts,hw,ec) len
=8] 00 00 00 00 00 00 00 00
/dev/pcanusbxl36: 1758097176.413771433 > PCANXL_TYPE_CANCC(1): [001h(1) flg=11000000h (ts,hw,ec) len
=8] 01 00 00 00 00 00 00 00
/dev/pcanusbxl36: 1758097176.414692400 > PCANXL_TYPE_CANCC(1): [001h(1) flg=13000000h (ts,hw,ec) len
=8] 02 00 00 00 00 00 00 00
/dev/pcanusbxl36: 1758097176.415609960 > PCANXL_TYPE_CANCC(1): [001h(1) flg=13000000h (ts,hw,ec) len
=8] 03 00 00 00 00 00 00 00
/dev/pcanusbxl36: 1758097176.416530937 > PCANXL_TYPE_CANCC(1): [001h(1) flg=13000000h (ts,hw,ec) len
=8] 04 00 00 00 00 00 00 00
/dev/pcanusbxl36: 1758097176.417438245 > PCANXL_TYPE_CANCC(1): [001h(1) flg=13000000h (ts,hw,ec) len
=8] 05 00 00 00 00 00 00 00
/dev/pcanusbxl36: 1758097176.418400229 > PCANXL_TYPE_CANCC(1): [001h(1) flg=13000000h (ts,hw,ec) len
=8] 06 00 00 00 00 00 00 00
/dev/pcanusbxl36: 1758097176.419314372 > PCANXL_TYPE_CANCC(1): [001h(1) flg=13000000h (ts,hw,ec) len
=8] 07 00 00 00 00 00 00 00
/dev/pcanusbxl36: 1758097176.420247310 > PCANXL_TYPE_CANCC(1): [001h(1) flg=13000000h (ts,hw,ec) len
=8] 08 00 00 00 00 00 00 00
/dev/pcanusbxl36: 1758097176.421188791 > PCANXL_TYPE_CANCC(1): [001h(1) flg=13000000h (ts,hw,ec) len
=8] 09 00 00 00 00 00 00 00
```

Writing CAN FD frames

Using the same Terminal window; start again the 3 applications that configure their own CAN controller in the 3 different CAN standard, except that the CAN controller configured as Classic CAN will obviously not understand the format of the CAN FD frames it will read on the bus and will therefore generate errors that will also be propagated on the bus and thus read by the other controllers. So, either it is removed from the test platform, or it is placed in “listen-only” mode to prevent it from disrupting the CAN bus:

1. CAN CC for PCAN-USB #0, with nominal bitrate = 500 Kbps, in **listen-only mode**:

```
$ pcanfdtst rx --listen-only -b 500k /dev/pcanusb32
```

2. CAN FD for PCAN-USB Pro FD CAN1, with same nominal bitrate and the CAN FD data bitrate = 2 Mbps:

```
$ pcanfdtst rx --fd -b 500k -d 2M /dev/pcanusbfd34
```

3. CAN XL for PCAN-USB XL #0, with same nominal and FD data bitrates, plus the CAN XL data bitrate = 8 Mbps:

```
$ pcanxl-dump -e on -n 500k -f 2M -x 8M /dev/pcanusbxl36
```



Note: For CAN XL to be backward compatible, the error signaling mode must be enabled (which automatically disables transceiver mode switch).

Then, write (say) 5x CAN FD frames with 64 data bytes and BRS set from a 4th Terminal window through PCAN-USB Pro FD CAN2:

```
$ pcanfdtst tx --fd -b 500k -d 2M --brs -n 5 -i 1 -I 8 -l 64 /dev/pcanusbfd35
1758097818.741282 /dev/pcanusbfd35 > BUS_STATE=ACTIVE [Rx:0 Tx:0] -
```

```

1758097818.752996 /dev/pcanusbfd35 < 001 ...b. 64 - 00 00 00 00 00 00 00 00
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
1758097818.754117 /dev/pcanusbfd35 < 001 ...b. 64 - 01 00 00 00 00 00 00 00
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
1758097818.755234 /dev/pcanusbfd35 < 001 ...b. 64 - 02 00 00 00 00 00 00 00
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
1758097818.756352 /dev/pcanusbfd35 < 001 ...b. 64 - 03 00 00 00 00 00 00 00
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
1758097818.757469 /dev/pcanusbfd35 < 001 ...b. 64 - 04 00 00 00 00 00 00 00
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
/dev/pcanusbfd35 < [packets=5 calls=5 bytes=320 eagain=0 drv_ovr=0 ctr_ovr=0]
Test duration: 0.011441 s. sent frames: 5 (223746 bps)

```

And let's see now how the other three applications react:

- └ The CAN CC application obviously receives bus error notifications:

```
$ pcanfdtst rx --listen-only -b 500k /dev/pcanusb32
1758097799~949827 /dev/pcanusb32 > BUS_STATE=ACTIVE [Rx:0 Tx:0] -
1758097818.753578 /dev/pcanusb32 > BUS_ERROR=STF_ERR [type=2 dir=Rx code=10] -
1758097818.753578 /dev/pcanusb32 > BUS_ERROR=OTH_ERR [type=3 dir=Rx code=19] -
1758097818.753820 /dev/pcanusb32 > BUS_ERROR=STF_ERR [type=2 dir=Rx code=7] -
1758097818.755740 /dev/pcanusb32 > BUS_ERROR=OTH_ERR [type=3 dir=Rx code=19] -
1758097818.756038 /dev/pcanusb32 > BUS_ERROR=STF_ERR [type=2 dir=Rx code=7] -
1758097818.760220 /dev/pcanusb32 > BUS_ERROR=OTH_ERR [type=3 dir=Rx code=19] -
1758097818.760476 /dev/pcanusb32 > BUS_ERROR=STF_ERR [type=2 dir=Rx code=2] -
1758097818.763590 /dev/pcanusb32 > BUS_ERROR=OTH_ERR [type=3 dir=Rx code=19] -
1758097818.763889 /dev/pcanusb32 > BUS_ERROR=STF_ERR [type=2 dir=Rx code=5] -
```

— The CAN FD application obviously reads the CAN FD frames:

```
$ pcanfdtst rx --fd -b 500k -d 2M /dev/pcanusbfd34
1758097154.096575 /dev/pcanusbfd34 > BUS_STATE=ACTIVE [Rx:0 Tx:0] -
...
1758097818.753639 /dev/pcanusbfd34 > 001 ...b. 64 - 00 00 00 00 00 00 00 00
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
1758097818.754761 /dev/pcanusbfd34 > 001 ...b. 64 - 01 00 00 00 00 00 00 00
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
1758097818.755880 /dev/pcanusbfd34 > 001 ...b. 64 - 02 00 00 00 00 00 00 00
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
1758097818.756972 /dev/pcanusbfd34 > 001 ...b. 64 - 03 00 00 00 00 00 00 00
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
1758097818.758126 /dev/pcanusbfd34 > 001 ...b. 64 - 04 00 00 00 00 00 00 00
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
```

- The CAN XL application obviously reads the CAN FD frames:

```
$ pcanxl-dump -e on -n 500k -f 2M -x 8M /dev/pcanusbxl36
/dev/pcanusbxl36: init 0MHz core in CAN_XL with flg=00030004h bitrates=[nominal=500000:80
00 fd=2000000:0000 xl=8000000:5620] ES=1 TMS=1 pwm[short=0 long=0 offset=0]
/dev/pcanusbxl36: device initialized
/dev/pcanusbxl36: init settings=[clock=160MHz flags=80010004h bitrate[nominal=500000:8000
brp=1 tseg1=255 tseg2=64 sjw=32 fd=2000000:7500 brp=1 tseg1=59 tseg2=20 sjw=10 ssp_offset
=0 xl=8000000:5500 brp=1 tseg1=10 tseg2=9 sjw=4 ssp_offset=0] ES=1 TMS=0]
/dev/pcanusbxl36: 1758097173.530316666 > PCANXL_TYPE_STATUS(3): [PCANFD_ERROR_ACTIVE (1)
flg=13000080h (bus,ts,hw,ec)]
...
/dev/pcanusbxl36: 1758097818.754701285 > PCANXL_TYPE_CANFD(2): [001h(1) flg=13100000h (br
s,ts,hw,ec) len=64]
00 00 00 00 00 00 00 00 00 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
/dev/pcanusbxl36: 1758097818.754802197 > PCANXL_TYPE_CANFD(2): [001h(1) flg=11100000h (br
s,ts,ec) len=64]
01 00 00 00 00 00 00 00 00 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
/dev/pcanusbxl36: 1758097818.756084412 > PCANXL_TYPE_CANFD(2): [001h(1) flg=13100000h (br
s,ts,hw,ec) len=64]
02 00 00 00 00 00 00 00 00 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
/dev/pcanusbxl36: 1758097818.757009914 > PCANXL_TYPE_CANFD(2): [001h(1) flg=11100000h (br
s,ts,ec) len=64]
03 00 00 00 00 00 00 00 00 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
/dev/pcanusbxl36: 1758097818.758331066 > PCANXL_TYPE_CANFD(2): [001h(1) flg=13100000h (br
s,ts,hw,ec) len=64]
04 00 00 00 00 00 00 00 00 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
```

Writing CAN XL frames

Using the same Terminal window; start again the 3 applications that configure their own CAN controller in the 3 different CAN standard, except that the CAN controller configured as Classic CAN will obviously not understand the format of the CAN XL frames it will read on the bus and will therefore generate errors that will also be propagated on the bus and thus read by the other controllers. So, either it is removed from the test platform, or it is placed in “listen-only” mode to prevent it from disrupting the CAN bus:

1. CAN CC for PCAN-USB #0, with nominal bitrate = 500 Kbps, in **listen-only mode**:

```
$ pcanfdtst rx --listen-only -b 500k /dev/pcanusb32
```

2. CAN FD for PCAN-USB Pro FD CAN1, with same nominal bitrate and the CAN FD data bitrate = 2 Mbps:

```
$ pcanfdtst rx --fd -b 500k -d 2M /dev/pcanusbfd34
```

3. CAN XL for PCAN-USB XL #0, with same nominal and FD data bitrates, plus the CAN XL data bitrate = 8 Mbps:

```
$ pcanxl-dump -e on -n 500k -f 2M -x 8M /dev/pcanusbxl36
```

Note: For CAN XL to be backward compatible, the error signaling mode must be enabled (which automatically disables transceiver mode switch).

Then, write (say) 3x CAN XL frames with 128 data bytes from a 4th Terminal window through PCAN-USB XL #1:

```
$ pcanxl-gen -n 500k -f 2M -x 8M -e on -c 3 -l 128 /dev/pcanusbxl37
/dev/pcanusbxl37: init 0MHz core in CAN_XL with flg=00030004h bitrates=[nominal=500000:8000 fd=2000000:0000 xl=8000000:5620] ES=1 TMS=1 pwm[short=0 long=0 off set=0]
/dev/pcanusbxl37: device initialized
/dev/pcanusbxl37: init settings=[clock=160MHz flags=80010004h bitrate[nominal=500000:8000 brp=1 tseg1=255 tseg2=64 sjw=32 fd=2000000:7500 brp=1 tseg1=59 tseg2=20 sjw=10 ssp_offset=0 xl=8000000:5500 brp=1 tseg1=10 tseg2=9 sjw=4 ssp_offset=0] ES=1 TMS=0]
/dev/pcanusbxl37: 1758101636.882195696 > PCANXL_TYPE_STATUS(3): [PCANFD_ERROR_ACTIVE (1) flg=13000080h (bus,ts,hw,ec)]
/dev/pcanusbxl37: 1758101636.892069069 < PCANXL_TYPE_CANXL(5): [PID=0 VCID=0 flg=01000000h (ts) len=128]
  01 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17
  18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
  30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40 41 42 43 44 45 46 47
  48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
  60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77
  78 79 7A 7B 7C 7D 7E 7F
/dev/pcanusbxl37: 1758101636.893182822 < PCANXL_TYPE_CANXL(5): [PID=0 VCID=0 flg=01000000h (ts) len=128]
  02 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17
  18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
  30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40 41 42 43 44 45 46 47
  48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
  60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77
  78 79 7A 7B 7C 7D 7E 7F
/dev/pcanusbxl37: 1758101636.894280247 < PCANXL_TYPE_CANXL(5): [PID=0 VCID=0 flg=01000000h (ts) len=128]
  03 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17
  18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
  30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40 41 42 43 44 45 46 47
  48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
  60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77
  78 79 7A 7B 7C 7D 7E 7F
/dev/pcanusbxl37: closing device
```

And let's see now how the other three applications react:

— The CAN CC application obviously receives bus error notifications:

```
$ pcanfdtst rx --listen-only -b 500k /dev/pcanusb32
...
1758101636.892639 /dev/pcanusb32 > BUS_ERROR=STF_ERR [type=2 dir=Rx code=8] -
1758101636.893747 /dev/pcanusb32 > BUS_ERROR=FRM_ERR [type=1 dir=Rx code=27] -
```

— The CAN FD application is not disturbed at all by the CAN XL traffic:

```
$ pcanfdtst rx --fd -b 500k -d 2M /dev/pcanusbfd34
1758097154.096575 /dev/pcanusbfd34 > BUS_STATE=ACTIVE [Rx:0 Tx:0] -
...
```

- The other CAN XL application receives the written CAN XL frames correctly.

```
$ pcanxl-dump -e on -n 500k -f 2M -x 8M /dev/pcanusbxl36
/dev/pcanusbxl36: init 0MHz core in CAN_XL with flg=00030004h
bitrates=[nominal=500000:8000 fd=2000000:0000 xl=8000000:5620] ES=1 TMS=1 pwm[short=0
long=0 offset=0]
/dev/pcanusbxl36: device initialized
/dev/pcanusbxl36: init settings=[clock=160MHz flags=80010004h bitrate[nominal=500000:8000
brp=1 tseg1=255 tseg2=64 sjw=32 fd=2000000:7500 brp=1 tseg1=59 tseg2=20 sjw=10
ssp_offset=0 xl=8000000:5500 brp=1 tseg1=10 tseg2=9 sjw=4 ssp_offset=0] ES=1 TMS=0]
/dev/pcanusbxl36: 1758097173.530316666 > PCANXL_TYPE_STATUS(3): [PCANFD_ERROR_ACTIVE (1)
flg=13000080h (bus,ts,hw,ec)]
...
/dev/pcanusbxl36: 1758101636.893504576 > PCANXL_TYPE_CANXL(5): [PID=0 VCID=0
flg=11000000h (ts,ec) len=128]
  01 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17
  18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
  30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40 41 42 43 44 45 46 47
  48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
  60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77
  78 79 7A 7B 7C 7D 7E 7F
/dev/pcanusbxl36: 1758101636.893505265 > PCANXL_TYPE_CANXL(5): [PID=0 VCID=0
flg=11000000h (ts,ec) len=128]
  02 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17
  18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
  30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40 41 42 43 44 45 46 47
  48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
  60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77
  78 79 7A 7B 7C 7D 7E 7F
/dev/pcanusbxl36: 1758101636.895630548 > PCANXL_TYPE_CANXL(5): [PID=0 VCID=0
flg=11000000h (ts,ec) len=128]
  03 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17
  18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
  30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40 41 42 43 44 45 46 47
  48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
  60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77
  78 79 7A 7B 7C 7D 7E 7F
```

4.10 netdev Mode

If the PCAN driver for Linux has been built for SocketCAN⁵ usage (a.k.a., in *netdev* mode), it is compatible for running with some network tools as well as the CAN utilities proposed by the SocketCAN community.

Note: Since kernel version 3.6, the *netdev* interface with all of the PEAK-System PC CAN interfaces is natively included in the mainline kernel. So, there is no need to install the PCAN driver for Linux when planning to use the SocketCAN interface in CAN CC/CAN FD applications.

Note: CAN XL can also be used through the *netdev* interface starting with Linux kernel v6.19.

In this mode, the driver registers a “CAN network interface” for each PC CAN interface it enumerates. Each network interface is given a name made of the prefix `can`, followed by a number starting from 0.

⁵ Background information: en.wikipedia.org/wiki/SocketCAN

4.10.1 assign Parameter

The `assign` parameter of the driver (described in Table 2: Driver module parameters on page 12) allows to break the default ascending number assignment model.

assign=peak

When loading the driver with the parameter `assign=peak`, the CAN network CAN interface number is fixed to the PCAN device minor number. In this mode, `canX` interface defines the same PC CAN interface as `/dev/pcanX`.

assign=pcanX:canY[,pcanX:canY]

Loading the driver with the parameter `assign=pcanX:canY` sets the name `canY` to the device which name is `pcanX`. When selecting this mode, the `assign` parameter value can be a list of several assignments, each separated by a `“,”` (comma).

assign=devid[,peak]

When loading the driver with the parameter `assign=devid`, then the name of the network CAN interface is made by using the `devid` value of the corresponding PC CAN interface. If the PC CAN interface does not define any `devid`, then the usual (ascending) order enumeration scheme is used (as if `assign=` was not used) unless `assign=devid,peak` is used. In that case, the CAN network number will be the same as the PCAN device number (as if `assign=peak` was used).

 **Note:** The value of the `devid` property can be changed using `test/pcan-settings` utility (see 4.8.3 *pcan-settings* on page 32).

4.10.2 defclk Parameter

The `defclk` parameter of the driver (described in Table 2: Driver module parameters on page 12) allows to change the default clock value of a CAN interface. Some PEAK-System PC-CAN interfaces can be programmed to switch from one clock to another, to get more accurate bit-timing.

defclk=value

When loading the driver with the parameter `defclk=value`, all the PC-CAN interfaces will try to switch from their default clock value to the given one. `value` is expressed in Hz. Ending letter `“M”` or `“k”` can be used as a shortcut to `“x1000000”` or `“x1000”`. For example:

```
defclk=12M
```

selects the 12 MHz clock of each PC CAN interface that can run with such a clock. If a PC-CAN interface can't select the given clock value, then it silently ignores it. If `value` is 0, the default clock is unchanged.

defclk=pcanX:valueA[,pcanY:valueB]

Loading the driver with the parameter `defclk=pcanX:valueA[,pcanY:valueB]` defines a specific clock value to each PC CAN interface which name is given in the characters string. For example:

```
defclk=pcan0:12M,pcan1:60M,pcan2:0,pcan3:24M
```

tells the first four PC CAN interfaces to respectively switch to their 12 MHz, 60 MHz, default and 24 MHz clock. If any of these interfaces can't select the given clock, then it silently ignores it.

4.10.3 ifconfig/iproute2

Both utilities configure a `canX` interface. While `ifconfig` is somewhere too old to support all the CAN/CAN-FD-specific features, the last versions of the `iproute2` package (especially the `ip` tool) include

options to set up a canX interface. Since v8, the canX interfaces exported by the PCAN driver can be configured using the `ip link` command.

Note: Configuring the canX interfaces needs root privileges.

The `ip` tool has been modified to handle protocol-specific features of CAN and CAN FD. This simplifies the bitrate setup of a CAN interface. The help of the tool describes its usage:

```
$ ip link set can0 type can help
Usage: ip link set DEVICE type can
      [ bitrate BITRATE [ sample-point SAMPLE-POINT] ] |
      [ tq TQ prop-seg PROP_SEG phase-seg1 PHASE-SEG1
        phase-seg2 PHASE-SEG2 [ sjw SJW ] ]

      [ loopback { on | off } ]
      [ listen-only { on | off } ]
      [ triple-sampling { on | off } ]
      [ one-shot { on | off } ]
      [ berr-reporting { on | off } ]

      [ restart-ms TIME-MS ]
      [ restart ]

Where: BITRATE      := { 1..1000000 }
       SAMPLE-POINT := { 0.000..0.999 }
       TQ           := { NUMBER }
       PROP-SEG     := { 1..8 }
       PHASE-SEG1   := { 1..8 }
       PHASE-SEG2   := { 1..8 }
       SJW          := { 1..4 }
       RESTART-MS   := { 0 | NUMBER }
```

Thus, setting the bitrate to a CAN interface is now possible using one of the following options:

- └─ bitrate bit-timing parameters set (a.k.a. sample-point, tq, prop-seg, phase-seg1, phase-seg2, sjw)
- └─ bitrate option followed by numeric value (if the kernel configuration option `CONFIG_CAN_CALC_BITTIMING` was set)

The `restart-ms` option defines a timer in milliseconds. After this period the CAN interface is automatically restarted on BUS-OFF condition. If the given numeric value is 0, then the automatic restart mechanism is disabled, thus user will have to manually do:

```
$ sudo ip link set can0 type can restart
```

The last and complete version of how to use the `ip link` tool with CAN networks is available online at: www.kernel.org/doc/Documentation/networking/can.txt

Examples:

- └─ Set up a PCAN *netdev* interface with 500 kbit/s:

```
$ ip link set canX up type can bitrate 500000
```


- Set up a PCAN *netdev* CAN FD interface with 1 Mbit/s and 2 Mbit/s of data bitrate (if supported):

```
$ ip link set canX up type can bitrate 1000000 dbitrate 2000000 fd on
```

- Set up a PCAN *netdev* CAN FD interface with 1 Mbit/s nominal bitrate and 2 Mbit/s data bitrate, running in non-ISO mode (if supported by the device and the kernel):

```
$ ip link set canX up type can bitrate 1000000 dbitrate 2000000 fd-non-iso on
```

- Set up a PCAN *netdev* CAN XL interface in CAN XL mixed mode with 1 Mbit/s arbitration bitrate, 4 Mbit/s CAN FD data bitrate, and 8 Mbit/s CAN XL data bitrate:

```
$ ip link set canX up type can bitrate 1000000 xl on dbitrate 4000000 xbitrate 8000000
```

- Set up a PCAN *netdev* CAN XL interface in pure CAN XL with 1 Mbit/s arbitration bitrate and 20 Mbit/s CAN XL data bitrate (TMS on):

```
$ ip link set canX up type can bitrate 1000000 xl on tms on xbitrate 20000000
```

- See also “ip link help can” for more details and explanations about the parameters added to the NETLINK interface to configure protocols more precisely.

 **Note:** The latest version of *iproute2* package can be downloaded from:
www.kernel.org/pub/linux/utils/net/iproute2/
(knowing that *iproute2-ss141224 v3.18* is ok)

You might use *ifconfig* for setting the interface UP or DOWN only:

```
$ ifconfig canX down
# canX can't be used no more
$ ifconfig canX up
# canX can be used by any application
```

 **Note:** loopback mode is supported since v8.10 of the driver. The below example shows how to configure *can0* to receive the echo of each frame sent as well as the frame looped back by the controller:

```
$ sudo ip link set can0 up type can loopback on bitrate 500000
```

```
$ candump -x can0
```

```
$ cansend can0 123#0011223344556677
```

```
$ candump -x can0
Can0 TX - - 123 [8] 00 11 22 33 44 55 66 77
Can0 RX - - 123 [8] 00 11 22 33 44 55 66 77
```

4.10.4 can-utils

The `can-utils` package⁶ contains some tools and utilities that allow transmitting and receiving CAN as well as CAN FD messages over the PCAN *netdev* interfaces.

Note: Transmitting and receiving to/from the CAN bus through the SocketCAN network interfaces needs these interfaces to be configured (see 4.10.3 *ifconfig/iproute2* on page 55).

Examples:

- Dump CAN/CAN FD messages received from the `canX` interface, display timestamps:

```
$ candump -t a canX
```

- Transmit a CAN message with ID 0x123 on `canX` with 4 data bytes 00 11 22 33:

```
$ cansend canX 123#00112233
```

- Transmit the same message with CAN FD (`##`) on `canX`, select the data bitrate for the data bytes (BRS flags = 1):

```
$ cansend can1 123##_100112233
```

4.11 USB Mass Storage Device Mode

Since driver version 8.8, it is possible to switch specific PC CAN interfaces to Mass Storage Device (MSD) mode. In this mode, the PC CAN interface appears as an external disk drive to the system. The purpose of this mode is to facilitate the upgrade of the firmware of the PC CAN interface. Once turned into that mode, the PC CAN interface must be off to restart in normal mode afterwards.

A PC CAN interface can be switched into MSD mode if its device nodes export the `mass_storage_mode` file under the `/sysfs` tree. In the example below, the PCAN-USB adapter cannot switch into MSD mode, while the PCAN-USB FD is able to:

```
$ cat /sys/class/pcan/pcanusb33/mass_storage_mode
cat: /sys/class/pcan/pcanusb33/mass_storage_mode: No such file or directory
$ cat /sys/class/pcan/pcanusbfd38/mass_storage_mode
0
```

Note: Reading the `mass_storage_mode` file (if it exists) always returns the character string "0".

⁶ Website can-utils: github.com/linux-can/can-utils/

Switching to MSD mode is only useful if the PC CAN interface firmware should be upgraded. In that case, super user should first get compatible firmware from the support pages of the PEAK-System website.

The user can switch a device to MSD mode in two ways:

1. With root privileges, writing 1 to the `mass_storage_mode` file of (for example) the directory entry that corresponds to the first device node of the PC CAN interface:

```
# echo 1 > /sys/class/pcan/pcanusbfd38/mass_storage_mode
```

Users of `sudo` will have to enter the command below instead:

```
$ sudo sh -c "echo 1 > /sys/class/pcan/pcanusbfd38/mass_storage_mode"
```

2. With root privileges, running the `pcan-settings` test application:

```
$ sudo pcan-settings -M -f /dev/pcanusbfd38
pcan-settings: Mass Storage mode successfully set
Please wait for the LED(s) of the USB device to flash, then, if not
automatically done by the system, mount a VFAT filesystem on the newly
detected USB Mass Storage Device "/dev/sdX".
```

 **Hint:** the “verbose” mode gives more details that may help for the next steps:

```
$ sudo pcan-settings -v -M -f /dev/pcanusbfd38
pcan-settings: Mass Storage mode successfully set

The device node "/dev/pcanusbfd38" doesn't exist anymore.
Please wait for the LED(s) of the USB device to flash, then, if not
automatically done by the system, mount a VFAT filesystem on the newly
detected USB Mass Storage Device "/dev/sdX".

For example:
# mkdir -p /mnt/pcan-usb
# mount -t vfat /dev/sdX /mnt/pcan-usb
# ls -al /mnt/pcan-usb
```

After a few seconds, the LED(s) of the PC CAN interface should blink, and the Kernel should detect a new USB mass storage device:

```
$ dmesg | tail -15
[27207.291209] usb 2-1.3.1: USB disconnect, device number 42
[27211.354058] usb 2-1.3.1: new high-speed USB device number 45 using ehci-pci
[27211.462592] usb 2-1.3.1: New USB device found, idVendor=0c72, idProduct=0101
[27211.462596] usb 2-1.3.1: New USB device strings: Mfr=0, Product=0, SerialNumber=0
[27211.462977] usb-storage 2-1.3.1:1.0: USB Mass Storage device detected
[27211.463223] scsi host11: usb-storage 2-1.3.1:1.0
[27212.482743] scsi 11:0:0:0: Direct-Access                USB to CAN            1.0  PQ: 0 ANSI: 0 CCS
[27212.483167] sd 11:0:0:0: Attached scsi generic sg4 type 0
[27212.483718] sd 11:0:0:0: [sde] 2048 512-byte logical blocks: (1.05 MB/1.00 MiB)
[27212.484335] sd 11:0:0:0: [sde] Write Protect is off
[27212.484339] sd 11:0:0:0: [sde] Mode Sense: 03 00 00 00
[27212.484944] sd 11:0:0:0: [sde] No Caching mode page found
[27212.484951] sd 11:0:0:0: [sde] Assuming drive cache: write through
[27212.490199]  sde:
[27212.492690] sd 11:0:0:0: [sde] Attached SCSI removable disk
```

In the example above, the Kernel attached the storage device `sde` to the newly detected USB mass storage device with `idVendor0c72` (that is PEAK-System). If the running Linux system doesn't automatically mount any file system onto that storage device, then super user must do it manually:

1. Create a mount point (`/mnt/pcan-usb-fd`, for example):

```
$ sudo mkdir -p /mnt/pcan-usb-fd
```

2. Mount the whole storage device on that mount point:

```
$ sudo mount -t vfat /dev/sde /mnt/pcan-usb-fd
```

3. Check the content of the mounted device (for example):

```
$ ls -al /mnt/pcan-usb-fd
total 830
drwxr-xr-x  2 root root    512 janv.  1  1970 .
drwxr-xr-x 10 root root   4096 déc.   13 11:23 ..
-rwxr-xr-x  1 root root 844800 avril  1  2015 firmware.bin
```

4. Remove the existing firmware file:

```
$ sudo rm -f /mnt/pcan-usb-fd/*.bin
```

Note: The remove operation is purely virtual but is mandatory to let the system think that the storage device is large enough to store the new firmware file. At that point, if the PC CAN interface is unplugged, then it will normally restart as usual once plugged in again.

5. Copy the new firmware file:

```
$ sudo cp PCAN-New_firmware_file.bin /mnt/pcan-usb-fd
```

6. Unmount all mount points for the storage device:

```
$ sudo umount -A /dev/sdX
```

After a few seconds, the PC CAN interface must be power cycled to run the new firmware. Either unplug it or switch it off, in case the PC CAN interface is powered by another source than the USB cable (like the PCAN-USB X6, for example).

Note: The PCAN-USB X6 adapter is equipped with 3 modules, each managing 2 CAN ports. Also, it is necessary to perform the previous manipulation 3 times in total, using each time the first device node of each module (CAN1, CAN3, and CAN5).

For example, if the connected PCAN-USB X6 adapter is exported by the system like that:

```
$ lspcan -t -T -i
dev name      port    irq    clock  btrs    bus
[PCAN-USB X6 0]
|_ pcanusbfd38 CAN1    -      80MHz  500k+2M CLOSED
|_ pcanusbfd39 CAN2    -      80MHz  500k+2M CLOSED
```

```
|_ pcanusbfd40  CAN3  -      80MHz  500k+2M  CLOSED
|_ pcanusbfd41  CAN4  -      80MHz  500k+2M  CLOSED
|_ pcanusbfd42  CAN5  -      80MHz  500k+2M  CLOSED
|_ pcanusbfd43  CAN6  -      80MHz  500k+2M  CLOSED
```

Then `pcanusbfd38`, `pcanusbfd40`, and `pcanusbfd42` should all be switched to MSD mode.

Once restarted, the PC CAN interface runs the new firmware. The version of the firmware that is embedded into a PC CAN interface (if any) can be read in the `/sysfs` tree. For example:

```
$ cat /sys/class/pcan/pcanusbfd38/adapter_version
3.2.0
```

5 Developer Guide

As explained in 3.1 *Build Binaries* on page 7, the PCAN Driver for Linux can be configured to run in two exclusive modes:

1. If built for *chardev* mode, the driver exports a classic open/read/write/ioctl/close character device interface to the user space applications, while
2. If built in *netdev* mode, the driver exports a socket interface.

 **Note:** The *netdev* mode is not available when building the driver for real-time environment.

Building and installing the driver as described in 3.1 *Build Binaries* on page 7 and in 3.2 Install Package on page 9 also builds and installs some user API libraries that encapsulate the system calls to the driver:

- `libpcan` is the good and old API which is still offering access to CAN 2.0 channels (see 5.1.1 CAN 2.0 libpcan API on page 64)
- `libpcanfd` is an API included in the package since version 8 of the driver. This API offers access to CAN 2.0 and CAN FD channels, as well as multi-messages services and status events messaging. Since this library also includes all the entry points of `libpcan` described in 5.1.1 CAN 2.0 libpcan API on page 64, this library can also be linked with CAN 2.0 API applications instead of using `libpcan`.
- `libpcanxl` is the new API included in the package since version 9 of the driver. This API offers access to CAN 2.0, CAN FD as well as CAN XL channels. Since this library also includes all the entry points of `libpcanfd` described in 5.1.2 CAN FD libpcanfd API on page 66, this library can also be linked with CAN 2.0 API (or CANFD API) applications instead of using `libpcanfd`.

The three libraries can be built to be used by real-time applications. Two RT environments can be selected when building these libraries, knowing that both make usage of RTDM.

 To build real-time libraries for running Xenomai real-time tasks:

```
$ make -C lib RT=XENOMAI # Or "make xeno" since pcan 8.2
```

 To build real-time libraries, for running RTAI real-time tasks:

```
$ make -C lib RT=RTAI # Or "make rtai" since pcan 8.2
```

5.1 chardev Mode

In this mode, the PCAN Driver for Linux creates one device node per CAN CC/FD/XL channel it discovers and attaches a minor number to it (unique for the driver). Like every character mode driver, the PCAN Driver for Linux is being attached to a major number by the system.

Each device node can be opened, closed, read, and written (see 4.7 *read/write Interface* on page 27). The main functions are implemented through the `ioctl()` entry point. The architecture of the several software components of the driver package since v8 is summarized in Figure 1: software components architecture.

The *chardev* mode is especially needed when one wants to take benefits from the PCAN-Basic API PEAK-System has developed, for writing applications that can run on both Windows and Linux systems.

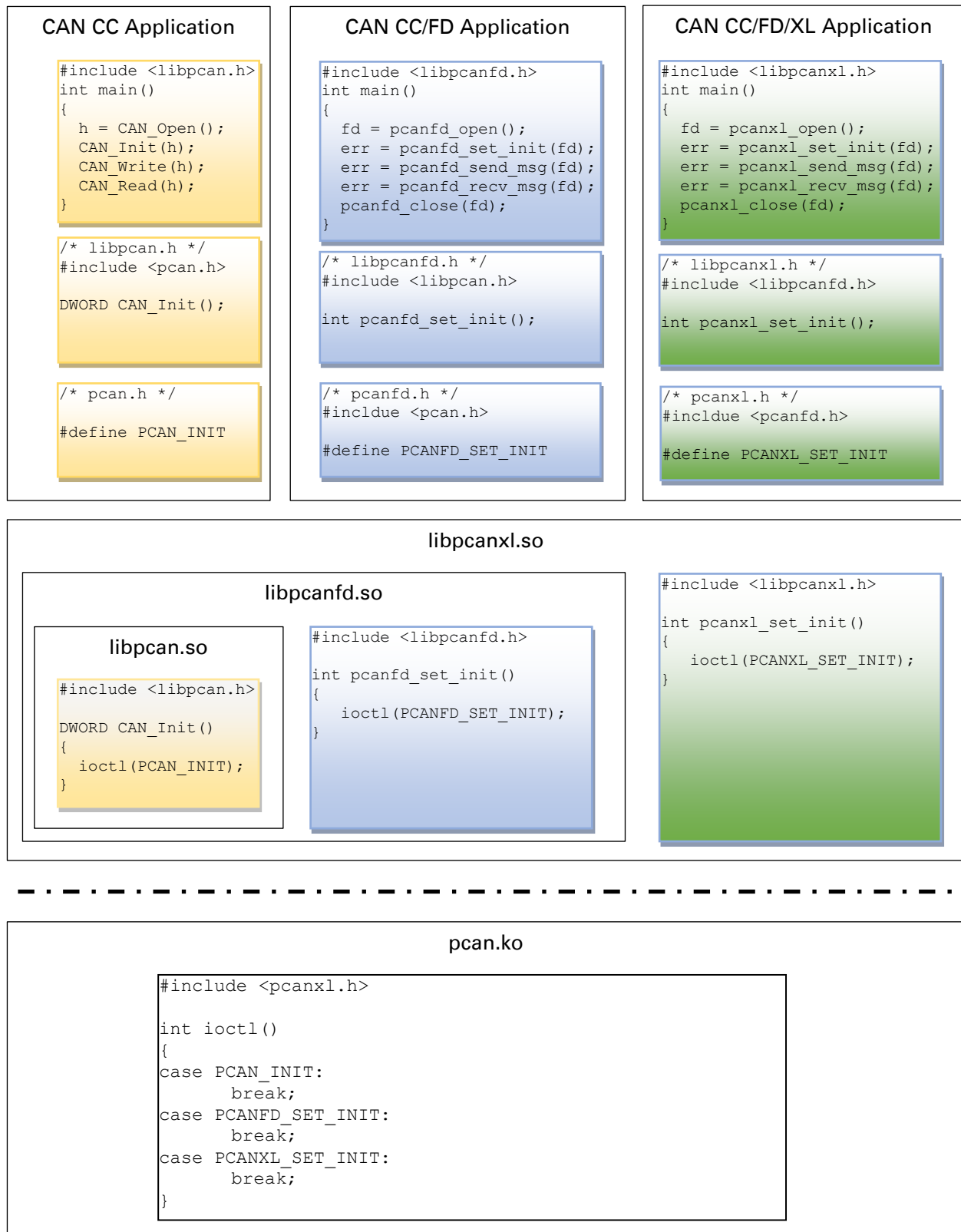


Figure 1: software components architecture

5.1.1 CAN 2.0 libpcan API

Note: This API is kept for backward compatibility reasons, thus these entry points are also proposed by the `libpcanfd` and `libpcanxl` libraries. But this API is considered deprecated. Use the new `libpcanfd` or `libpcanxl` API instead.

The (old) CAN 2.0 API *ioctl*/codes are defined by `pcan.h`:

```
#define PCAN_INIT          _IOWR(PCAN_MAGIC_NUMBER, MYSEQ_START,      TPCANInit)
#define PCAN_WRITE_MSG    _IOW (PCAN_MAGIC_NUMBER, MYSEQ_START + 1, TPCANMsg)
#define PCAN_READ_MSG     _IOR (PCAN_MAGIC_NUMBER, MYSEQ_START + 2, TPCANRdMsg)
#define PCAN_GET_STATUS   _IOR (PCAN_MAGIC_NUMBER, MYSEQ_START + 3, TPSTATUS)
#define PCAN_DIAG         _IOR (PCAN_MAGIC_NUMBER, MYSEQ_START + 4, TPDIAG)
#define PCAN_BTR0BTR1     _IOWR(PCAN_MAGIC_NUMBER, MYSEQ_START + 5, TPBTR0BTR1)
#define PCAN_GET_EXT_STATUS _IOR (PCAN_MAGIC_NUMBER, MYSEQ_START + 6, TPEXTENDEDSTATUS)
#define PCAN_MSG_FILTER    _IOW (PCAN_MAGIC_NUMBER, MYSEQ_START + 7, TPMSGFILTER)
#define PCAN_EXTRA_PARAMS _IOWR(PCAN_MAGIC_NUMBER, MYSEQ_START + 8, TPEXTRAPARAMS)
```

This API enables the user application to read and write CAN 2.0 messages (only) from/through any PC CAN interface of PEAK-System. This API is encapsulated by the `libpcan` library (C/C++ programs like `transmittest`, `receivetest`, `bitratetest`, and `pcan-settings` stored in the `test` directory use this API). Since this API is always supported for CAN 2.0 access, to use this API, the application must link with `-lpcan` or `-lpcanfd` or `-lpcanxl`.

The principle of this API is to implement a CAN 2.0 channel with something like an object `HANDLE` used during the whole life of the connection to the CAN bus. This API is greatly inspired by the PCAN-Light version for Windows®.

The library defines the following entry points:

HANDLE CAN_Open(WORD wHardwareType, ...);

This function opens a CAN 2.0 channel according to its type (PCI, USB, ISA ...) and its channel number (or some other arguments depending on the chosen type). See the list of `HW_XXX` symbols defined in `pcan.h` to get the list of supported values for `wHardwareType`.

For example:

```
#include <libpcan.h>

/* open the 2nd CAN 2.0 PCI channel in the system (first is 0) */
HANDLE h = CAN_Open(HW_PCI, 1);
```

DWORD CAN_Init(HANDLE hHandle, WORD wBTR0BTR1, int nCANMsgType);

This function initializes an opened CAN 2.0 channel with a bitrate (expressed in BTR0BTR1 SJA1000 format) and a filter set (or not set) to the extended Id of the CAN messages.

See the list of `CAN_BAUD_XXX` and `CAN_INIT_TYPE_XX` symbols defined in `libpcan.h` to get the list of supported values for `wBTR0BTR1` values and `nCANMsgType`.

For example:

```
#include <libpcan.h>

/* CAN 2.0 channel handle */
HANDLE h;
DWORD status;

...

/* initialize the CAN 2.0 channel with 500 kbps BTR0BTR1, accepting extended ID. */
status = CAN_Init(h, CAN_BAUD_500K, CAN_INIT_TYPE_EX);
```


DWORD CAN_Write(HANDLE hHandle, TPCANMsg* pMsgBuff);

This function writes a CAN 2.0 message to a CAN bus through an opened CAN 2.0 channel.

For example:

```
#include <libpcan.h>

/* CAN 2.0 channel handle */
HANDLE h;
DWORD status;
TPCANMsg msg;

...
msg.ID = 0x123
msg.MSGTYPE = MSGTYPE_STANDARD;
msg.LEN = 3;
msg.DATA[0] = 0x01;
msg.DATA[1] = 0x02;
msg.DATA[2] = 0x03;

/* write standard msg ID = 0x123. with 3 data bytes 0x01 0x02 0x03
 * (the function may block)
 */
status = CAN_Write(h, &msg);
```

DWORD CAN_Read(HANDLE hHandle, TPCANMsg* pMsgBuff);

This function reads a CAN 2.0 message received from a CAN bus through an opened CAN 2.0 channel. If no message has been received, the calling task is blocked.

For example:

```
#include <libpcan.h>

/* CAN 2.0 channel handle */
HANDLE h;
DWORD status;
TPCANMsg msg;

...
/* wait for a CAN 2.0 msg received from the CAN channel
 * (the function may block)
 */
status = CAN_Read(h, &msg);
```

DWORD CAN_Status(HANDLE hHandle);

This function returns the status of an opened CAN 2.0 channel (corresponding to the last column displayed with `cat /proc/pcan`). The returned value is a bitmask (see the list of `CAN_ERR_XXX` symbols defined in `pcan.h` to get the meaning of each bit).



Note: Reading the status of a channel with this function clears up it!

For example:

```
#include <libpcan.h>

/* CAN 2.0 channel handle */
HANDLE h;
DWORD status;

...
/* get the status of a CAN 2.0 channel */
status = CAN_Status(h);
```

DWORD CAN_Close(HANDLE hHandle);

This function closes an opened CAN 2.0 channel. The given handle should not be used next.

For example:

```
#include <libpcan.h>

/* CAN 2.0 channel handle */
HANDLE h;

...

/* wait for a CAN 2.0 msg received from the CAN channel
 * (the function may block)
 */
CAN_Close(h);
```

To get profit from the multi-tasking environment of Linux, the library has been extended with the following `LINUX_XXX()` functions:

int LINUX_CAN_FileHandle(HANDLE hHandle);

This function returns the file descriptor corresponding to the device node opened by the driver. This is useful when an application must wait for more than one read/write event.

HANDLE LINUX_CAN_Open(const char *szDeviceName, int nFlag);

This function opens a CAN 2.0 channel, but with the Linux system device node name instead.

DWORD LINUX_CAN_Read(HANDLE hHandle, TPCANRdMsg* pMsgBuff);

This functions acts like "DWORD CAN_Read(HANDLE hHandle, TPCANMsg* pMsgBuff);", but returns extra timestamp information.

DWORD LINUX_CAN_Read_Timeout(HANDLE hHandle, TPCANRdMsg* pMsgBuff, int nMicroSeconds);

This function acts like "DWORD LINUX_CAN_Read(HANDLE hHandle, TPCANRdMsg* pMsgBuff);", but, in case there is no message to read from the CAN, it blocks the calling task for `nMicroSeconds` at maximum.

DWORD LINUX_CAN_Write_Timeout(HANDLE hHandle, TPCANMsg* pMsgBuff, int nMicroSeconds);

This function acts like "DWORD CAN_Write(HANDLE hHandle, TPCANMsg* pMsgBuff);", but in case there is no more room in the transmit queue of the CAN channel, it blocks the calling task for `nMicroSeconds` at maximum.

DWORD LINUX_CAN_Extended_Status(HANDLE hHandle, int *nPendingReads, int *nPendingWrites);

This function acts like "DWORD CAN_Status(HANDLE hHandle);", but also returns the count of messages waiting to be read from the receive queue of the channel in `*nPendingReads`, and the count of messages waiting to be sent from the transmit queue of the channel in `* nPendingWrites`.

DWORD LINUX_CAN_Statistics(HANDLE hHandle, TPDIAG *diag);

This function gives some statistics about a CAN 2.0 channel but without clearing the status of this channel (like "DWORD CAN_Status(HANDLE hHandle);" does).

WORD LINUX_CAN_BTR0BTR1(HANDLE hHandle, DWORD dwBitRate);

This function returns the BTR0BTR1 8 MHz SJA1000 code corresponding to the given bitrate.

5.1.2 CAN FD libpcanfd API

This API is new since version 8 of the driver. It always proposes the entry points and data structures defined in the old one (see 5.1.1 CAN 2.0 libpcan API on page 64) but adds definition of some new data

structures and *ioctl*/codes (see `pcanfd.h`). The old entry points always allow connections to the CAN 2.0 bus as usual, while the new ones enable user to connect to CAN 2.0 and/or CAN FD buses. In other words, the new API is a newer, more modern and universal way of accessing the CAN bus. The old entry points are only kept for ensuring backward compatibility with existing application code.

```
#define PCANFD_SET_INIT      _IOW(PCAN_MAGIC_NUMBER, PCANFD_SEQ_SET_INIT, \
                                sizeof(struct pcanfd_init))
#define PCANFD_GET_INIT     _IOR(PCAN_MAGIC_NUMBER, PCANFD_SEQ_GET_INIT, \
                                sizeof(struct pcanfd_init))
#define PCANFD_GET_STATE    _IOR(PCAN_MAGIC_NUMBER, PCANFD_SEQ_GET_STATE, \
                                sizeof(struct pcanfd_state))
#define PCANFD_ADD_FILTERS  _IOW(PCAN_MAGIC_NUMBER, PCANFD_SEQ_ADD_FILTERS, \
                                sizeof(struct pcanfd_msg_filters))
#define PCANFD_GET_FILTERS  _IOW(PCAN_MAGIC_NUMBER, PCANFD_SEQ_GET_FILTERS, \
                                sizeof(struct pcanfd_msg_filters))
#define PCANFD_SEND_MSG     _IOW(PCAN_MAGIC_NUMBER, PCANFD_SEQ_SEND_MSG, \
                                sizeof(struct pcanfd_msg))
#define PCANFD_RECV_MSG     _IOR(PCAN_MAGIC_NUMBER, PCANFD_SEQ_RECV_MSG, \
                                sizeof(struct pcanfd_msg))
#define PCANFD_SEND_MSGS    _IOWR(PCAN_MAGIC_NUMBER, PCANFD_SEQ_SEND_MSGS, \
                                sizeof(struct pcanfd_msgs))
#define PCANFD_RECV_MSGS    _IOWR(PCAN_MAGIC_NUMBER, PCANFD_SEQ_RECV_MSGS, \
                                sizeof(struct pcanfd_msgs))
#define PCANFD_GET_AVAILABLE_CLOCKS _IOWR(PCAN_MAGIC_NUMBER, \
                                PCANFD_SEQ_GET_AVAILABLE_CLOCKS, \
                                sizeof(struct pcanfd_available_clocks))

#define PCANFD_GET_BITTIMING_RANGES _IOWR(PCAN_MAGIC_NUMBER, \
                                PCANFD_SEQ_GET_BITTIMING_RANGES, \
                                sizeof(struct pcanfd_bittiming_ranges))
#define PCANFD_GET_OPTION   _IOWR(PCAN_MAGIC_NUMBER, PCANFD_SEQ_GET_OPTION, \
                                sizeof(struct pcanfd_option))
#define PCANFD_SET_OPTION   _IOW(PCAN_MAGIC_NUMBER, PCANFD_SEQ_SET_OPTION, \
                                sizeof(struct pcanfd_option))
```

These new *ioctl*/codes are also encapsulated by some new entry points of the new `libpcanfd` library. These new entry points are defined in `libpcanfd.h`.

Note: The test application `pcanfdtst` uses these new entry points.

This library does not anymore encapsulate CAN channels into any `HANDLE` objects but directly deals with file descriptors returned by the `open()` system call, made on the corresponding device node.

Note: The `libpcan` and `libpcanfd` APIs are not compatible! Once a CAN channel is opened through one API, it cannot be used with the other one. In other words, opening a CAN channel selects the API that is used for the connection.

The `libpcanfd` API offers several levels of usage. While level 1 encapsulates the above *ioctl*/codes, level 2 API offers a more friendly way of opening and closing a device node.

Finally, all the entry points of the `libpcanfd` API return an integer value. If it is negative, it must be interpreted as an error code that equals to `-errno`.

int pcanfd_set_init(int fd, struct pcanfd_init *pfdi);

This function initializes an opened device node with some new settings that enable you to select CAN 2.0 as well as CAN FD properties (if the corresponding hardware is compatible). These properties are defined by `struct pcanfd_init` object (see also `pcanfd.h`):

```

struct pcanfd_init {
    __u32 flags;
    __u32 clock_Hz;
    struct pcan_bittiming    nominal;
    struct pcan_bittiming    data;
};

```

Field	Values	Description
flags	PCANFD_INIT_LISTEN_ONLY	The device is open in listen-only mode.
	PCANFD_INIT_STD_MSG_ONLY	Only standard CAN message IDs are transmitted and received. If not set, all kinds of messages IDs are used for that device.
	PCANFD_INIT_FD	Open the device for CAN FD ISO access if the device is CAN-FD-capable.
	PCANFD_INIT_FD_NON_ISO	Open the device for CAN FD non-ISO access if the device is CAN-FD-capable.
	PCANFD_INIT_TS_DEV_REL	Timestamps set by the driver to the messages received from the CAN bus are relative to the moment the device is initialized.
	PCANFD_INIT_TS_HOST_REL	Timestamps set by the driver to the messages received from the CAN bus are relative to the moment the host has started.
	PCANFD_INIT_TS_DRV_REL	Timestamps set by the driver to the messages received from the CAN bus will be relative to the moment the driver has started (default).
	PCANFD_INIT_BUS_LOAD_INFO	If the CAN bus load is information the corresponding hardware can provide, then the driver will periodically put STATUS messages in the rx fifo queue of this channel to inform the application of the current bus load the channel is connected to.
clock	0	The default clock of the CAN device is selected by the driver (default).
	any other value	The clock frequency (expressed in Hz) to select in the CAN interface hardware device to select the right bit timing specifications.
nominal	struct pcan_bittiming	Defines the nominal bitrate to use to connect to the CAN bus (default value is defined in Table 2 on page 12).
data	struct pcan_bittiming	Defines the data bitrate to select when the device is a CAN FD one, and the written message flag PCANFD_MSG_BRS is set (default value is defined in Table 2 on page 12).

Table 9: struct pcanfd_init description

```
int pcanfd_get_init(int fd, struct pcanfd_init *pfdi);
```

This function enables the user application to get the initialization settings that are set to an opened device.

```
int pcanfd_get_state(int fd, struct pcanfd_state *pfds);
```

This function gets the current state of an opened device. The state of a CAN channel is summarized in the new struct pcanfd_state object (see also pcanfd.h):

```

struct pcanfd_state {
    __u16    ver_major, ver_minor, ver_subminor;

    struct timeval  tv_init;          /* time the device was initialized */

    enum pcanfd_status  bus_state;    /* CAN bus state */

    __u32    device_id;              /* device ID, ffffffff is unused */

    __u32    open_counter;            /* open() counter */
    __u32    filters_counter;         /* count of message filters */

    __u16    hw_type;                 /* PCAN device type */
    __u16    channel_number;          /* channel number for the device */

    __u16    can_status;              /* same as wCANStatus but NOT CLEARED */
    __u16    bus_load;               /* bus load value, ffff if not given */

    __u32    tx_max_msgs;             /* Tx fifo size in count of msgs */
    __u32    tx_pending_msgs;         /* msgs waiting to be sent */
    __u32    rx_max_msgs;             /* Rx fifo size in count of msgs */
    __u32    rx_pending_msgs;         /* msgs waiting to be read */
    __u32    tx_frames_counter;        /* Tx frames written on device */
    __u32    rx_frames_counter;        /* Rx frames read on device */
    __u32    tx_error_counter;         /* CAN Tx errors counter */
    __u32    rx_error_counter;         /* CAN Rx errors counter */

    __u64    host_time_ns;            /* host time in nanoseconds as it was */
    __u64    hw_time_ns;              /* when hw_time_ns has been received */
};

```

int pcanfd_add_filter(int fd, struct pcanfd_msg_filter *pf);

This function adds a message filter to the device's filters list. When a device is opened, no filters exist for the device, that is, the application receives all message IDs read from the CAN bus. Adding a message filter enables filtering CAN messages which are to pass to the application, and which are to discard. The message filter is described by the new `struct pcanfd_msg_filter` object (see also `pcanfd.h`):

```

struct pcanfd_msg_filter {
    __u32    id_from;                /* msgs ID in range [id_from..id_to] */
    __u32    id_to;                  /* and flags == msg_flags */
    __u32    msg_flags;              /* will be passed to applications */
};

```

int pcanfd_add_filters(int fd, struct pcanfd_msg_filters *pfl);

This function adds several message filters to the device's filters list at once. The list of messages is saved into the following `struct pcanfd_msg_filters`:

```

struct pcanfd_msg_filters {
    __u32    count
    struct pcanfd_msg_filter list[0];
};

```



Note: The `count` field should contain the number of message filters saved in the `list[]` array field.

```
int pcanfd_add_filters_list(int fd, int count, struct
pcanfd_msg_filter *pf);
```

This function adds several message filters to the device's filters list at once. This is a shortcut easier to use than "int pcanfd_add_filters(int fd, struct pcanfd_msg_filters *pfl);".

```
int pcanfd_del_filters(int fd);
```

This function deletes all the filters linked to device's filters list. No filters do exist anymore for the device, so the application will receive all message IDs read from the CAN bus. This is the default behavior of a CAN device when it has been opened.

```
int pcanfd_send_msg(int fd, struct pcanfd_msg *pfdm);
```

This function writes a message to the CAN bus through an opened device. The message is defined by the new struct pcanfd_msg object (see also pcanfd.h):

```
struct pcanfd_msg {
    __u16      type;           /* PCANFD_TYPE_XXX */
    __u16      data_len;      /* true length (not the DLC) */
    __u32      id;
    __u32      flags;         /* PCANFD_XXX definitions */
    struct timeval timestamp;
    __u8       ctrlr_data[PCANFD_CTRLR_MAXDATALEN];
    __u8       data[PCANFD_MAXDATALEN] __attribute__((aligned(8)));
};
```

This C structure object can carry a CAN 2.0 as well as a CAN FD message. It also can contain some out-of-band message types (like status messages) that can be pushed by the driver to the application.

Note: Writing a message to the CAN bus might block the calling task, unless the device node has been opened in non-blocking mode. In that case, `-EWOULDBLOCK` is returned by this function if the task had not enough room to store the outgoing message.

Field	Values	Description
type	PCANFD_TYPE_CAN20_MSG	This message is a CAN 2.0 message (the data_len field cannot be larger than 8).
	PCANFD_TYPE_CANFD_MSG	This message is a CAN FD message. Bits like PCANFD_MSG_BRS are handled by the flags field. The data_len field cannot be larger than 64.
data_len	<= 8	Number of data bytes to copy from the data field to transmit on the CAN bus.
	<= 64	In case of CAN FD message, this value is the true count of bytes to write. The driver is in charge to adapt this to the corresponding DLC code.
flags	PCANFD_MSG_RTR	Remote Transmission Request message.
	PCANFD_MSG_EXT	The message ID is to be coded using 29 bits (the standard message format uses 11 bits only).
	PCANFD_MSG_SLF	If supported, this message is looped back by the CAN interface hardware to its internal receive queue. Note: Depending on the hardware, this bit may or may not be set in the flags of the looped back frame.
	PCANFD_MSG_SNG	If supported, this message is transmitted in Single-Shot mode, that is, if the CAN frame is not transmitted successfully, no further transmissions are attempted.
	PCANFD_MSG_BRS	In case of CAN FD, this bit enables the alternate data bitrate for the transport of the data bytes, instead of the nominal bitrate.
	PCANFD_MSG_ECHO	If supported, this message is written on the bus and echoed by the hardware to its internal receive queue. Moreover, that bit is set in the echoed frame, as well as ctrlr_data[PCANFD_ECHOID] that contains a copy of ctrlr_data[PCANFD_ECHOID] of the sent message.
id		The ID of the CAN message.
data		The data bytes of the CAN message. Only the count of bytes given by data_len field is copied onto the bus.

Field	Values	Description
ctrlr_data		When PCANFD_MSG_ECHO is set, the 7 low order bit of ctrlr_data[PCANFD_ECHOID] will be copied into ctrlr_data[PCANFD_ECHOID] of the echoed frame.

Table 10: Usage of struct pcanfd_msg on the transmit side

```
int pcanfd_send_msgs(int fd, struct pcanfd_msgs *pfdm1);
```

This function writes a list of messages to the CAN bus through an opened device. The message list is defined by the new struct pcanfd_msgs object (see also pcanfd.h):

```
struct pcanfd_msgs {
    __u32 count;
    struct pcanfd_msg list[0];
};
```

This C structure object can carry several CAN 2.0 and CAN FD messages. The number of messages to write is given by the `count` field. This field is also used to indicate how many messages have really been written in the transmit queue of the device.

Note: Writing several messages to the CAN bus might block the calling task, unless the device node has been opened in non-blocking mode. In that case, `-EWOULDBLOCK` is returned by this function if the task had not enough room to store the outgoing messages.

If at least one message has been successfully written in the transmit queue, then the function returns 0. Otherwise, it returns a negative error code.

Using this function saves memory copies and constant round trips between kernel and user spaces.

Example:

```
#include <malloc.h>
#include <libpcanfd.h>

int fill_msg(struct pcanfd_msg *pm);

struct pcanfd_msgs *pml;

/* allocate enough room to store 5 CAN messages */
pml = malloc(sizeof(*pml) + 5 * sizeof(struct pcanfd_msg));
pml->count = 5;

for (pml->count = 0; pml->count < 5; pml->count++) {
    fill_msg(pml->list + pml->count);
}

/* put all of the messages at once in the transmit queue of the device... */
err = pcanfd_send_msgs(fd, pml);
if (err)
    printf("Only %u/5 msgs have been sent because of errno=%d\n",
           pml->count, err);

free(pml);
...
```

```
int pcanfd_send_msgs_list(int fd, int count, struct pcanfd_msg
*pfdm);
```

This function writes a list of messages to the CAN bus through an opened device. This is a shortcut easier to use than `"int pcanfd_send_msgs(int fd, struct pcanfd_msgs *pfdm1);"`.

int pcanfd_recv_msg(int fd, struct pcanfd_msg *pfdm);

This function reads any pending message the driver might have pushed in the corresponding device receive queue. This message can be an in-band message if it contains a CAN 2.0 or a CAN FD message received from the CAN bus, or an out-of-band message if it contains a status message.

Note: Reading a message from the driver might block the calling task, unless the device node has been opened in non-blocking mode. In that case, `-EWOULDBLOCK` is returned by this function if the task didn't find any message to read.

Field	Values	Description
type	PCANFD_TYPE_CAN20_MSG	This message is a CAN 2.0 message.
	PCANFD_TYPE_CANFD_MSG	This message is a CAN FD message. Bits like <code>PCANFD_MSG_BRS</code> or <code>PCANFD_MSG_ESI</code> can also be set in the flags field.
	PCANFD_TYPE_STATUS	This message is a status message, giving some more information about the state of the CAN device.
	PCANFD_TYPE_ERROR_MSG	This message is an error message read from the CAN bus. This kind of messages ISNOT received by default (see also option <code>PCANFD_ALLOWED_MSG_ERROR</code> in <code>int pcanfd_set_option(int fd, int name, void *value, int size);</code>).
data_len		Number of data bytes in the message received from the CAN device. Note that in case of CAN FD, this value might not be the same as the one given on the transmission side.
id		The ID of the CAN message.
flags	PCANFD_MSG_RTR	Remote Transmission Request message.
	PCANFD_MSG_EXT	The message ID format is an extended one.
	PCANFD_MSG_SLF	This message has been looped-back by the hardware to its internal receive queue.
	PCANFD_MSG_SNG	This message has been transmitted in Single-Shot mode.
	PCANFD_MSG_BRS	In case of CAN FD, this bit indicates that data bitrate has been selected for transmitting the data bytes of the received message.
	PCANFD_MSG_ECHO	This message has been echoed by the hardware to its internal receive queue as well as written on the bus and <code>ctrlr_data[PCANFD_ECHOID]</code> contains a copy of <code>ctrlr_data[PCANFD_ECHOID]</code> of the sent message.
	PCANFD_MSG_ESI	CAN FD error indicator: errors detected on the CAN bus.
	PCANFD_TIMESTAMP	The timestamp field is valued with the timestamp the message has been received.
	PCANFD_HWTIMESTAMP	When <code>PCANFD_TIMESTAMP</code> is set, this flag indicates that the given timestamp is made from the timestamp given by the hardware. If not set, the timestamp has been built by the driver from the host time.
	PCANFD_ERRCNT	<code>ctrlr_data[PCANFD_RXERRCNT]</code> and <code>ctrlr_data[PCANFD_TXERRCNT]</code> contain Rx and Tx error counters read from the CAN controller.
	PCANFD_BUSLOAD	<code>ctrlr_data[PCANFD_BUSLOAD_UNIT]</code> contains the percentage of the bus load computed by the hardware controller, while <code>ctrlr_data[PCANFD_BUSLOAD_DEC]</code> contains the decimal part.
	PCANFD_OVRCNT	<code>ctrlr_data[PCANFD_RXERRCNT]</code> contains the number of messages lost because the Rx queue of the driver was full.
timestamp	struct timeval	If <code>PCANFD_TIMESTAMP</code> is set in the flag field, then this one indicates the moment the message has been received. If <code>PCANFD_HWTIMESTAMP</code> is also set, the given moment is a time made from the hardware clock. If <code>PCANFD_HWTIMESTAMP</code> is not set, this moment is made by the driver, from the host current time (see also option in <code>int pcanfd_set_option(int fd, int name, void *value, int size);</code>).
ctrlr_data		CAN-controller-specific data (see <code>PCANFD_MSG_ECHO</code> , <code>PCANFD_ERRCNT</code> and <code>PCANFD_BUSLOAD</code> flags above).
data		The data bytes of the CAN message. The count of data bytes received is given by the <code>data_len</code> field.

Table 11: Usage of struct pcanfd_msg on the receive side

int pcanfd_recv_msgs(int fd, struct pcanfd_msgs *pfdm1);

This function can read a list of messages at once from the driver device receive queue. The messages list is defined by the new `struct pcanfd_msgs` object (see also `pcanfd.h`):

```
struct pcanfd_msgs {
    __u32          count;
    struct pcanfd_msg list[0];
};
```

This C structure object can carry several CAN 2.0 and CAN FD messages. The maximum number of messages the list can contain must be set in the `count` field. When returning from this function, the `count` field is set by the driver to the real number of copied messages.

Note: Reading several messages from the driver might block the calling task, unless the device node has been opened in non-blocking mode. In that case, `-EWOULDBLOCK` is returned by this function if the task didn't find any message to read.

If at least one message has been successfully read, then the function returns 0. Otherwise, it returns a negative error code.

Using this function saves memory copies and constant round trips between kernel and user spaces.

Example:

```
#include <malloc.h>
#include <libpcanfd.h>
#include <errno.h>

int process_msg(struct pcanfd_msg *pm)
{
    switch (pm->type) {
    case PCANFD_TYPE_CAN20_MSG:
        return process_CAN_2_0_msg(pm);
    case PCANFD_TYPE_CANFD_MSG:
        return process_CAN_FD_msg(pm);
    case PCANFD_TYPE_STATUS:
        return process_status_msg(pm);
    case PCANFD_TYPE_ERROR_MSG:
        /* if enabled, see PCANFD_OPT_ALLOWED_MSGS[PCANFD_ALLOWED_MSG_ERROR] */
        return process_error_msg(pm);
    }
}

return -EINVAL
}

struct pcanfd_msgs *pml;
int i, err;

/* allocate enough room to store at least 5 CAN messages */
pml = malloc(sizeof(*pml) + 5 * sizeof(struct pcanfd_msg));
pml->count = 5;

/* waiting for these messages... */
err = pcanfd_recv_msgs(fd, pml);
if (err)
    exit(1);

/* process the received messages... */
for (i = 0; i < pml->count; i++) {
    process_msg(pml->list + i);
}

free(pml);
...
```

```
int pcanfd_recv_msgs_list(int fd, int count, struct pcanfd_msg *pm);
```

This function is able to read a list of messages at once from the driver device receive queue. This is a shortcut easier to use than "int pcanfd_recv_msgs(int fd, struct pcanfd_msgs *pfdml);".

If the return value is positive, then it indicates the real count of messages read from the device input queue. Otherwise, it's an error code.

```
int pcanfd_get_available_clocks(int fd, struct
pcanfd_available_clocks *pac);
```

This function returns the list of all the available clocks the underlying CAN/CAN FD device can run with. The clock is selected at the time the device is initialized (see int pcanfd_set_init(int fd, struct pcanfd_init *pfdi);).

```
/* Device available clocks value */
struct pcanfd_available_clock {
    __u32    clock_Hz;
    __u32    clock_src;
};

struct pcanfd_available_clocks {
    __u32    count;
    struct pcanfd_available_clock list[0];
};
```

User is responsible to setup the "count" field with the count of items it has allocated in the "list[]" array.

Example:

```
struct pcanfd_available_clocks *pac;
int i, err;

/* allocate enough room to store at least 8 clock values */
pac = malloc(sizeof(*pac) + 6 * sizeof(struct pcanfd_available_clock));
pac->count = 6;

/* reading the available clocks list */
err = pcanfd_get_available_clocks(fd, pac);
if (err)
    exit(1);

/* display all available clocks */
for (i = 0; i < pac->count; i++) {
    printf("clock #%u/%u: %u Hz\n", i, pac->count, pac->list[i]);
}

free(pac);
```

 **Note:** list[0] always contains default clock value. Only CAN FD devices define more than one clock.

```
int pcanfd_get_bittiming_ranges(int fd, struct
pcanfd_bittiming_ranges *pbtr)
```

This function returns the list of all the available bit timing ranges the underlying CAN/CAN FD device can run with. The bit timings are selected at the time the device is initialized (see int pcanfd_set_init(int fd, struct pcanfd_init *pfdi);).

```

/* CAN FD bittiming capabilities */
struct pcanfd_bittiming_range {
    __u32    brp_min;
    __u32    brp_max;
    __u32    brp_inc;
    __u32    tseg1_min;
    __u32    tseg1_max;
    __u32    tseg2_min;
    __u32    tseg2_max;
    __u32    sjw_min;
    __u32    sjw_max;
};

struct pcanfd_bittiming_ranges {
    __u32    count;
    struct pcanfd_bittiming_range list[0];
};

```

User is responsible for setting up the "count" field with the count of items it has allocated in the "list[]" array.

Version 8.2 of the PCAN driver always sets 1 in the "count" field for any CAN 2.0 device, while it sets 2 for any CAN FD device.

Example:

```

struct pcanfd_bittiming_ranges *pbr;
int err;

/* allocate enough room to store 2 ranges */
pbr = malloc(sizeof(*pbr) + 2 * sizeof(struct pcanfd_bittiming_range));
pbr->count = 2;

/* reading the bit timings ranges list */
err = pcanfd_get_bittiming_ranges(fd, pbr);
if (err)
    exit(1);

if (pbr->count == 1)
    printf("CAN 2.0 device\n");
else
    printf("CAN FD device\n");

free(pbr);

```

int pcanfd_get_option(int fd, int name, void *value, int size);

This function enables application to read the current value of an option attached to a channel device. Each channel handles the same set of options which values are initialized once it is opened. The list of these options is given below and may evolve over time (see also `pcanfd.h`).

Getting the value of an option that doesn't exist returns `-EINVAL`, while getting an unsupported option (for the device) return `-EOPNOTSUPP`. Reading the value of an option with a too small *value* buffer return `-ENOSPC`.

Successfully reading the value of an option returns the number of bytes that have been copied into *value*.

Option	Size (bytes)	Description												
PCANFD_OPT_CHANNEL_FEATURES	4	The value of this option is a bitmask that gives the features of an opened channel: <table><tr><td>PCANFD_FEATURE_FD</td><td>Channel is CAN FD capable</td></tr><tr><td>PCANFD_FEATURE_XL</td><td>Channel is CAN XL capable</td></tr><tr><td>PCANFD_FEATURE_IFRAME_DELAYUS</td><td>Delay can be inserted between frames</td></tr><tr><td>PCANFD_FEATURE_BUSLOAD</td><td>Channel is able to compute bus load</td></tr><tr><td>PCANFD_FEATURE_HWTIMESTAMP</td><td>timestamp are read from the device</td></tr><tr><td>PCANFD_FEATURE_DEVICEID</td><td>Channel can be labeled with a user device id.</td></tr></table>	PCANFD_FEATURE_FD	Channel is CAN FD capable	PCANFD_FEATURE_XL	Channel is CAN XL capable	PCANFD_FEATURE_IFRAME_DELAYUS	Delay can be inserted between frames	PCANFD_FEATURE_BUSLOAD	Channel is able to compute bus load	PCANFD_FEATURE_HWTIMESTAMP	timestamp are read from the device	PCANFD_FEATURE_DEVICEID	Channel can be labeled with a user device id.
PCANFD_FEATURE_FD	Channel is CAN FD capable													
PCANFD_FEATURE_XL	Channel is CAN XL capable													
PCANFD_FEATURE_IFRAME_DELAYUS	Delay can be inserted between frames													
PCANFD_FEATURE_BUSLOAD	Channel is able to compute bus load													
PCANFD_FEATURE_HWTIMESTAMP	timestamp are read from the device													
PCANFD_FEATURE_DEVICEID	Channel can be labeled with a user device id.													
PCANFD_OPT_DEVICE_ID	4	Get the user id attached to the channel device (if supported by the channel)												
PCANFD_OPT_AVAILABLE_CLOCKS		Return the list of clocks available in the channel device. The value returned is of type <code>pcanfd_available_clocks</code> (see <code>pcanfd.h</code> and <code>int pcanfd_get_available_clocks(int fd, struct pcanfd_available_clocks *pac);</code>). Getting this option is equivalent to calling <code>pcanfd_get_available_clocks()</code> .												
PCANFD_OPT_BITTIMING_RANGES		Give the bit timings ranges available for the channel, to specify the nominal bitrate. These ranges depend on which CAN controller the channel is equipped with (see also <code>int pcanfd_get_bittiming_ranges(int fd, struct pcanfd_bittiming_ranges *pbtr)</code>).												
PCANFD_OPT_DBITTIMING_RANGES		Give the bit timings ranges available for the channel, to specify the CAN FD data bitrate. These ranges depend on which CAN FD controller the channel is equipped with (see also <code>int pcanfd_get_bittiming_ranges(int fd, struct pcanfd_bittiming_ranges *pbtr)</code>).												
PCANFD_OPT_XL_BITTIMING_RANGES		Give the bit timings ranges available for the channel, to specify the CAN XL data bitrate. These ranges depend on which CAN XL controller the channel is equipped with (see also <code>int pcanfd_get_bittiming_ranges(int fd, struct pcanfd_bittiming_ranges *pbtr)</code>).												
PCANFD_OPT_XL_PWM_RANGES		Give the PWM settings range available for the CAN XL channel.												
PCANFD_OPT_ALLOWED_MSGS	4	The value of this option is a bitmask that describes which kind of message an application is able to receive: <table><tr><td>PCANFD_ALLOWED_MSG_CAN</td><td>CAN CC/CAN FD/CAN XL frames</td></tr><tr><td>PCANFD_ALLOWED_MSG_RTR</td><td>RTR frames</td></tr><tr><td>PCANFD_ALLOWED_MSG_EXT</td><td>Extended Id.</td></tr><tr><td>PCANFD_ALLOWED_MSG_STATUS</td><td>STATUS messages</td></tr><tr><td>PCANFD_ALLOWED_MSG_ERROR</td><td>Error from the CAN bus</td></tr></table>	PCANFD_ALLOWED_MSG_CAN	CAN CC/CAN FD/CAN XL frames	PCANFD_ALLOWED_MSG_RTR	RTR frames	PCANFD_ALLOWED_MSG_EXT	Extended Id.	PCANFD_ALLOWED_MSG_STATUS	STATUS messages	PCANFD_ALLOWED_MSG_ERROR	Error from the CAN bus		
PCANFD_ALLOWED_MSG_CAN	CAN CC/CAN FD/CAN XL frames													
PCANFD_ALLOWED_MSG_RTR	RTR frames													
PCANFD_ALLOWED_MSG_EXT	Extended Id.													
PCANFD_ALLOWED_MSG_STATUS	STATUS messages													
PCANFD_ALLOWED_MSG_ERROR	Error from the CAN bus													
PCANFD_OPT_ACC_FILTER_11B	8	Get the current acceptance filter code and mask for the standard messages received on the channel. The high-order 32-bits contain the acceptance code while the low-order ones contain the acceptance mask.												
PCANFD_OPT_ACC_FILTER_29B	8	Get the current acceptance filter code and mask for the extended messages received on the channel. The high-order 32-bits contain the acceptance code while the low-order ones contain the acceptance mask.												
PCANFD_OPT_IFRAME_DELAYUS	4	Get the value of the delay in μ s that is currently inserted by the CAN controller between each frame it sends.												

Option	Size (bytes)	Description
PCANFD_OPT_HWTIMESTAMP_MODE	4	Get the current mode the driver currently uses to compute the timestamps saved into each <code>struct pcanfd_msg</code>. PCANFD_OPT_HWTIMESTAMP_OFF Host time based only (even if hw is capable). PCANFD_OPT_HWTIMESTAMP_ON Host time base + raw hw time offset. PCANFD_OPT_HWTIMESTAMP_COOKED Host time base + cooked hw time offset. PCANFD_OPT_HWTIMESTAMP_RAW Raw hardware timestamps. Same as above but hw timestamp is fired at Start Of Frame instead of End Of Frame:. PCANFD_OPT_HWTIMESTAMP_SOF_ON PCANFD_OPT_HWTIMESTAMP_SOF_COOKED PCANFD_OPT_HWTIMESTAMP_SOF_RAW Cooked timestamps handle clocks drift between the different clocks systems (PC, board, USB controller...) Raw hardware timestamps are 64-bits μs timestamps given by the controller converted into s. + μs. These timestamps ARE NOT host time related.
PCANFD_OPT_DRV_VERSION	4	Get the driver version.
PCANFD_OPT_FW_VERSION	4	Get the device firmware version.
PCANFD_IO_DIGITAL_CFG	4	Get/set the configuration of the digital I/O pins of the PCAN-Chip USB (firmware $\geq$ 3.3.0): 1 the I/O pin is setup in output mode 0 the I/O pin is set up in input mode.
PCANFD_IO_DIGITAL_VAL	4	Get/set the digital I/O pins value.
PCANFD_IO_DIGITAL_SET	4	Set the digital I/O pins to high.
PCANFD_IO_DIGITAL_CLR	4	Clear the digital I/O pins to low.
PCANFD_IO_ANALOG_VAL	4	Get the analog I/O value from the PCAN-Chip USB.
PCANFD_OPT_MASS_STORAGE_MODE	4	The value of this option is always 0 if the device is able to switch in Mass Storage Device mode. If the device is not able to switch in MSD mode, reading this option fails and <code>errno</code> is set to <code>EOPNOTSUPP</code>.
PCANFD_OPT_DRV_CLK_REF	4	Get the clock reference used by the driver (see Table 6 on page 21).
PCANFD_OPT_LINGER	4	Get the maximum waiting time in ms the driver will wait before closing the device, while there are frames to write pending in the driver device Tx queue.
PCANFD_OPT_SELF_ACK	4	Check whether controller sends ACK by itself to the frames it writes on the bus.
PCANFD_OPT_BRS_IGNORE	4	Check whether the controller ignores received frames with BRS flag set.

Table 12

```
int pcanfd_set_option(int fd, int name, void *value, int size);
```

This function enables application to set a value to an option attached to a channel device. Each channel handles the same set of options which values are initialized once it is opened. The list of the options that can be changed is given below and may evolve over time (see also `pcanfd.h`).

Setting the value of an option that does not exist or setting an invalid value to an existing option return `-EINVAL`, while setting a value to an unsupported option (for the device) return `-EOPNOTSUPP`.

Correctly setting a value to an option returns 0.

Option	Size (in bytes)	Description
PCANFD_OPT_DEVICE_ID	4	Set a user numeric value to the channel device (if supported by the channel)

Option	Size (in bytes)	Description
PCANFD_OPT_ALLOWED_MSGS	4	Set what kinds of message the application wants to be notified of. Once opened, each channel is able to receive: PCANFD_ALLOWED_MSG_CAN CAN/CAN-FD frames PCANFD_ALLOWED_MSG_RTR RTR frames PCANFD_ALLOWED_MSG_EXT Extended Id. PCANFD_ALLOWED_MSG_STATUS STATUS messages
PCANFD_OPT_ACC_FILTER_11B	8	Set the current acceptance filter code and mask for the standard messages received on the channel. The high-order 32-bits should contain the acceptance code while the low-order ones should contain the acceptance mask.
PCANFD_OPT_ACC_FILTER_29B	8	Set the current acceptance filter code and mask for the extended messages received on the channel. The high-order 32-bits should contain the acceptance code while the low-order ones should contain the acceptance mask.
PCANFD_OPT_IFRAME_DELAYUS	4	Set the value of the delay in μ s that should be inserted by the CAN controller between each frame it sends, if this controller is able to.
PCANFD_OPT_HWTIMESTAMP_MODE	4	Set the current mode the driver should use to compute the timestamps saved into each <code>struct pcanfd_msg</code> . PCANFD_OPT_HWTIMESTAMP_OFF Host time based only (even if hw is capable). PCANFD_OPT_HWTIMESTAMP_ON Host time base + raw hw time offset. PCANFD_OPT_HWTIMESTAMP_COOKED Host time base + cooked hw time offset. PCANFD_OPT_HWTIMESTAMP_RAW Raw hardware timestamps. Same as above but hw timestamp is fired at Start Of Frame instead of End Of Frame:. PCANFD_OPT_HWTIMESTAMP_SOF_ON PCANFD_OPT_HWTIMESTAMP_SOF_COOKED PCANFD_OPT_HWTIMESTAMP_SOF_RAW Cooked timestamps handle clocks drift between the different clocks systems (PC, board, USB controller...) Raw hardware timestamps are 64-bits μ s timestamps given by the controller converted into s. + μ s. These timestamps ARE NOT host time related.
PCANFD_OPT_MASS_STORAGE_MODE	4	If the device is compatible, setting a value different from 0 to this option switches the PC CAN interface in Mass Storage Device mode. If the device cannot switch in MSD mode, setting this option fails and <code>errno</code> is set to <code>EOPNOTSUPP</code> . If the user hasn't got root privileges, setting this option fails and <code>errno</code> is set to <code>EPERM</code> .
PCANFD_OPT_FLASH_LED	4	Make the LED of the device blink to identify it (if it is equipped with one). Value is the number of milliseconds the LED should blink.
PCANFD_OPT_DRV_CLK_REF	4	Set the clock reference used by the driver (see Table 6 on page 21).
PCANFD_OPT_LINGER	4	PCANFD_OPT_LINGER_NOWAIT Task doesn't wait for the Tx queue to be empty before closing the device. PCANFD_OPT_LINGER_AUTO Driver automatically computes time to wait for the TX queue to be empty before closing the device. Any other positive value defines the maximum waiting time in ms before closing the device. The default value is 1000.
PCANFD_OPT_SELF_ACK	4	If 1 then requests the controller to send ACK by itself to the frames it writes on the bus. 0 disables this option.
PCANFD_OPT_BRS_IGNORE	4	If 1 then requests the controller to ignore received frames with BRS flag set. 0 disables this option.

Table 13

```
int pcanfd_open(char *dev_pcan, __u32 flags, ...);
```

This function is a shortcut used to open and initialize any PC CAN interface. First parameter is the name of the device node known by the system. Second argument is a bitmask which indicates what the next

parameters of the function are, and their sequence order, as well as the `PCANFD_INIT_XXX` flags used to initialize the CAN controller (see also `libpcanfd.h` and `pcanfd.h`).

Table 14 describes the order and how each bit of the `flags` argument is interpreted:

Bit	Description
<code>OFD_BITRATE</code>	The specification of the nominal bitrate starts with the third parameter: If <code>OFD_BTR0BTR1</code> is set too, then the third parameter is interpreted as a 16-bit value respecting the BTR0BTR1 SJA1000 format. If <code>OFD_BRPTSEGSJW</code> is specified, then the 3rd, 4th, 5th, and 6th parameters are interpreted as BRP, TSEG1, TSEG2, and SJW values. If none of the above bits is set, then the third argument is interpreted as a bits-per-second value.
<code>OFD_SAMPLEPT</code>	Argument next to the nominal bitrate is the minimal sample point rate requested. If not specified, the driver uses its own default value. If specified, this value must be expressed in 1/10000th (that is, 8750 stands for 87,5 %)
<code>OFD_DBITRATE</code>	The data bitrate is given in the next arguments: If <code>OFD_BTR0BTR1</code> is set too, then the next parameter is interpreted as a 16-bit value respecting the BTR0BTR1 SJA1000 format. If <code>OFD_BRPTSEGSJW</code> is specified, then the 4 next parameters are interpreted as BRP, TSEG1, TSEG2 and SJW values. If none of the above bits is set, then the next argument is interpreted as a bits-per-second value.
<code>OFD_DSAMPLEPT</code>	Argument next to the data bitrate is the minimal sample point rate requested. If not specified, the driver uses its own default value. If specified, this value must be expressed in 1/10000th (that is, 8750 stands for 87,5 %)
<code>OFD_CLOCKHZ</code>	The clock frequency (in Hz) to select in the CAN controller is given in the next argument.
<code>OFD_NONBLOCKING</code>	The device node is opened in non-blocking mode.
<code>PCANFD_INIT_XXX</code>	All of these flags are used to initialize the CAN device, as if it was initialized using "int pcanfd_set_init(int fd, struct pcanfd_init *pfdi);".

Table 14: Usage of the flags argument of `pcanfd_open()`

Example:

```
#include <libpcanfd.h>

int fd;

/* open the 1st CANFD channel of the PCAN-USB Pro FD and set 1Mbps+2Mbps bitrate */
fd = pcanfd_open("/dev/pcanusbprofd0",
                OFD_BITRATE|OFD_DBITRATE,
                1000000,
                2000000);
...
```

int pcanfd_is_canfd_capable(int fd);

This function allows application to know if an open device can work in CAN-FD or not. It returns 0 if the device is not a CAN-FD device.

int pcan_set_extra_params(int fd, struct pcan_extra_params *pe);

This function encapsulates an ioctl code that exists since CAN 2.0 API. It enables to set/get some extra parameters to/from the device through the driver, using the following C structure (see `pcan.h`):

```
#define PCAN_SF_DATA_MAXLEN    64          /* New since 8.14 */

typedef struct pcan_extra_params {
    int    nSubFunction;
```

```

        union {
            DWORD    dwSerialNumber;
            BYTE     ucHCDeviceNo;
            BYTE     ucDevData[PCAN_SF_DATA_MAXLEN];    /* New since 8.14 */
        } func;
    } TPEXTRAPARAMS;
    ...

```

CAN 2.0 API defines these two sub functions to set/get specific parameters:

nSubFunction	func	Description
SF_GET_SERIALNUMBER	dwSerialNumber	Get the serial number from the device (is possible).
SF_GET_HCDEVICENO	ucHCDeviceNo	Get the user defined device number from the device flash memory.
SF_SET_HCDEVICENO		Set a user defined device number value into the device flash memory.

With pcan v8.14, CAN-FD API has extended this usage by including the "ucDevData" 64 bytes array field to be able to exchange more than simple 32-bit data objects:

nSubFunction	func	Description
SF_GET_FWVERSION	dwSerialNumber	Get firmware version of the device.
SF_GET_ADAPTERNAME	ucDevData	Get the commercial name of the PC CAN interface that controls the device. The driver returns a null terminated string.
SF_GET_PARTNUM		Get the PC CAN interface part number. The driver returns a null terminated string.

```

int pcan_init(int fd, const struct pcan_init *pi);
int pcan_read_msg(int fd, struct pcan_rd_msg *prdm);
int pcan_write_msg(int fd, const struct pcan_msg *pm);
int pcan_get_status(int fd, struct pcan_status *ps);
int pcan_get_ext_status(int fd, struct pcan_ext_status *ps);
int pcan_get_diag(int fd, struct pcan_diag *pd);
int pcan_get_btr0btr1(int fd, struct pcan_btr0btr1 *pb);
int pcan_set_msg_filter(int fd, const struct pcan_msg_filter *pf);

```

These functions are simple clones of the existing functions of the original API (see 5.1.1 CAN 2.0 libpcan API on page 64).

5.1.3 CAN XL libpcanxl API

This API is new since version 9 of the driver. It always proposes the entry points and data structures defined in the old ones (see 5.1.2 *CAN FD libpcanfd API* on page 66), but adds definition of some new data structures and *ioctl*/codes (see `pcanxl.h`). The old entry points always allow connections to the CAN 2.0/CAN FD bus as usual, while the new ones enable user applications to connect to CAN CC/FD/XL buses. The old entry points are only kept for ensuring backward compatibility with existing application code.


```
#define PCANXL_SET_INIT      _IOW(PCAN_MAGIC_NUMBER, PCANFD_SEQ_SET_INIT, \
                                sizeof(struct pcanxl_init))
#define PCANXL_GET_INIT      _IOR(PCAN_MAGIC_NUMBER, PCANFD_SEQ_GET_INIT, \
                                sizeof(struct pcanxl_init))
#define PCANXL_SEND_MSG(d)   _IOW(PCAN_MAGIC_NUMBER, PCANXL_SEQ_SEND_MSG, d)
#define PCANXL_RECV_MSG(d)   _IOR(PCAN_MAGIC_NUMBER, PCANXL_SEQ_RECV_MSG, d)
```

These new *ioctl* codes are also encapsulated by some new entry points of the new `libpcanxl` library. These new entry points are defined in `libpcanxl.h`.

 **Note:** The `utils-xl` applications use these new entry points.

All the entry points of the `libpcanxl` API return an integer value. If it is negative, it must be interpreted as an error code that equals to `-errno`.

`libpcanxl` does not need to redefine any other existing `libpcanfd` entry point while this entry point does not deal with CAN XL new data structures. However, to simplify the writing and reading of code for applications using the `libpcanxl` API, it provides aliases for these entry points that are independent of the CAN protocol used. For example:

```
static inline int pcanxl_set_device_id(int fd, __u32 devid)
{
    return pcanfd_set_device_id(fd, devid);
}
```

`int pcanxl_set_init(int fd, struct pcanxl_init *pxli);`

This function initializes an opened device node with some new settings that enable you to configure CAN CC, CAN FD, or CAN XL protocol (if the corresponding hardware is compatible). The protocol properties are defined by `struct pcanxl_init` object (see also `pcanxl.h`):

```
struct pcanxl_init {
    __u32 flags;
    __u32 clock_Hz;
    struct pcan_bittiming nominal;
    struct pcan_bittiming fd_data;
    struct pcan_bittiming xl_data;
    struct pcanxl_pwm xl_pwm;
    __u8 rxm_limit[PCANXL_CAN_MAX]
};
```

Field	Type/Values	Description
flags	PCANXL_INIT_FD	Open the device for CAN FD ISO access if the device is CAN FD-capable.
	PCANXL_INIT_XL	Open the device for CAN XL access if the device is CAN XL-capable.
	PCANXL_INIT_TRX_MODE_SWITCH_ON	Activates CAN XL Transceiver Mode Switch which allows data transfer up to 20 Mbps.
	PCANXL_INIT_ERR_SIGNALING_OFF	Activate pure CAN XL mode.
	PCANFD_INIT_XXX	See Table 9: <code>struct pcanfd_init</code> description
clock	0	The default clock of the CAN device is selected by the driver (default).
	any other value	The clock frequency (expressed in Hz) to select in the CAN device hardware to select the right bit timing specifications.
nominal	<code>struct pcan_bittiming</code>	Defines the nominal bitrate to use to connect to the CAN bus (default value is defined in Table 2 on page 12).
fd_data	<code>struct pcan_bittiming</code>	Defines the CAN FD data bitrate to switch to when the selected CAN protocol is CAN FD and if the written frame flag <code>PCANFD_MSG_BRS</code> is set (default value is defined in Table 2 on page 12).

Field	Type/Values	Description
xl_data	struct pcan_bittiming	Defines the CAN XL data bitrate to switch to when CAN XL protocol is selected.
xl_pwm	struct pcanxl_pwm	Defines the short, long and offset values needed to configure the PWM signal when TMS is selected. If short and long values are 0, then the driver selects the preferred values according to the CAN XL standard.
rxmt_limit	__u8 [PCANXL_CAN_MAX]	Array of bytes. Now, only rxm_limit[PCANXL_CAN_XL] is used by the driver to define the maximal number of retransmissions of a CAN frame.

Table 15: struct pcanxl_init description

```
int pcanxl_send_msg(int fd, const struct pcanxl_msg *pxlm);
int pcanxl_send_msg_local(int fd, const struct pcanxl_msg *pxlm);
```

These functions both write a frame message to the CAN bus through an opened device. The message frame is defined by the new struct pcanxl_msg object (see also pcanxl.h):

```
struct pcanxl_msg {
    __u16 type;          /* PCANXL_TYPE_xxx */
    __u16 data_len;      /* Length of the data */
    __u32 id;            /* CAN CC/FD frame Id */
    __u32 flags;         /* PCANXL_MSG_xxx / PCANFD_MSG_xxx */
    __u8 ctrlr_data[PCANXL_MAXCTRLRDATALEN];
    __u8 sdt;           /* CAN XL SDU Type (see PCANXL_SDT_xxx) */
    __u32 af;           /* */
    struct __kernel_timespec timestamp; /* Message timestamp */
    __u8 data[] __attribute__((aligned(8)));
};
```

This C structure object can carry CAN CC, CAN FD, and CAN XL message frame.

When used to transport a CAN frame with (or without) data bytes, "data_len" is used to indicate the length in bytes of the "data" flexible array field. Therefore, "struct pcanxl_msg" can't obviously be used "as is" to instantiate a static variable:

- When the device is opened for CAN_CC only, then user application can instantiate a "struct pcanxl_msg_cc" message instead, which is a shortcut to:

```
struct pcanxl_msg {
    __u16 type;          /* PCANXL_TYPE_xxx */
    __u16 data_len;      /* Length of the data */
    __u32 id;            /* CAN CC/FD frame Id */
    __u32 flags;         /* PCANXL_MSG_xxx / PCANFD_MSG_xxx */
    __u8 ctrlr_data[PCANXL_MAXCTRLRDATALEN];
    __u8 sdt;           /* CAN XL SDU Type (see PCANXL_SDT_xxx) */
    __u32 af;           /* */
    struct __kernel_timespec timestamp; /* Message timestamp */
    __u8 data[PCANFD_CAN20_MAXDATALEN] __attribute__((aligned(8)));
};
```

- When the device is opened for CAN CC/FD, then user application can instantiate a "struct pcanxl_msg_fd" message instead, which is a shortcut to:

```

struct pcanxl_msg {
    __u16 type;          /* PCANXL_TYPE_xxx */
    __u16 data_len;      /* Length of the data */
    __u32 id;            /* CAN CC/FD frame Id */
    __u32 flags;         /* PCANXL_MSG_xxx / PCANFD_MSG_xxx */
    __u8 ctrlr_data[PCANXL_MAXCTRLRDATALEN];
    __u8 sdt;           /* CAN XL SDU Type (see PCANXL_SDT_xxx) */
    __u32 af;           /* */
    struct __kernel_timespec timestamp; /* Message timestamp */
    __u8 data[PCANFD_CANFD_MAXDATALEN] __attribute__((aligned(8)));
};

```

- When the device is opened for CAN CC/FD/XL, then user application can instantiate a "struct pcanxl_msg_xl" message instead, which is a shortcut to:

```

struct pcanxl_msg {
    __u16 type;          /* PCANXL_TYPE_xxx */
    __u16 data_len;      /* Length of the data */
    __u32 id;            /* CAN CC/FD frame Id */
    __u32 flags;         /* PCANXL_MSG_xxx / PCANFD_MSG_xxx */
    __u8 ctrlr_data[PCANXL_MAXCTRLRDATALEN];
    __u8 sdt;           /* CAN XL SDU Type (see PCANXL_SDT_xxx) */
    __u32 af;           /* */
    struct __kernel_timespec timestamp; /* Message timestamp */
    __u8 data[PCANXL_CANXL_MAXDATALEN] __attribute__((aligned(8)));
}

```

IMPORTANT: For performance reasons, "struct pcanxl_msg_xxx" CAN'T be instantiated on the stack (a.k.a. local variable) but must be instantiated in static or dynamic memory (a.k.a. global variable). If, for any reason, a local variable must be used, then user must do a copy of it in global memory or use the helper function `pcanxl_send_msg_local()` instead.

Note: Writing a message to the CAN bus might block the calling task, unless the device node has been opened in non-blocking mode. In that case, `-EWOULDBLOCK` is returned by this function if the task had not enough room to store the outgoing message.

Field	Type/Values	Description
type	PCANFD_TYPE_CAN20_MSG/ PCANXL_TYPE_CANCC	This message is a CAN 2.0 message (the <code>data_len</code> field cannot be larger than 8).
	PCANFD_TYPE_CANFD_MSG/ PCANXL_TYPE_CANFD	This message is a CAN FD message. Bits like <code>PCANFD_MSG_BRS</code> are handled by the <code>flags</code> field. The <code>data_len</code> field cannot be larger than 64.
	PCANXL_TYPE_CANXL	This message is a CAN XL message. The <code>data_len</code> field cannot be larger than 2048. <code>sdt</code> , <code>id</code> and <code>af</code> fields should be populated accordingly.
data_len	<= 8	Number of data bytes to copy from the data field to transmit on the CAN bus.
	<= 64	In case of CAN FD message, this value is the true count of bytes to write. The driver is in charge to adapt this to the corresponding DLC code.
	<= 2048	Number of data bytes of the CAN XL frame to write on the CAN bus.
id		The ID of the CAN CC/FD message, VCID/PID of the CAN XL message otherwise (see <code>PCANXL_ID()</code> macro in <code>pcanxl.h</code>)
flags	PCANXL_MSG_SEC	CAN XL Simple Extended Content message flag.
	PCANXL_MSG_RRS	CAN XL Remote Request Substitute message flag.
	PCANFD_MSG_xxx	See Table 10: Usage of struct <code>pcanfd_msg</code> on the transmit side
ctrlr_data		When <code>PCANFD_MSG_ECHO</code> is set, the 7 low order bit of <code>ctrlr_data[PCANFD_ECHOID]</code> will be copied into <code>ctrlr_data[PCANFD_ECHOID]</code> of the echoed frame.

Field	Type/Values	Description
sdt	PCANXL_SDT_xxx	CAN XL CiA611-1 SDU Type (see PCANXL_SDT_xxx in pcanxl.h)
af		CAN XL Acceptance Field This field is included in the 64-bit hardware acceptance filter of the CAN XL controller. It may contain node address or content-indicating information.
data	Flexible array of bytes	CAN message data bytes

Table 16: Usage of struct pcanxl_msg on the transmit side

```
int pcanxl_recv_msg(int fd, struct pcanxl_msg *pxlm);
int pcanxl_recv_msg_local(int fd, struct pcanxl_msg *pxlm);
```

These functions read any pending message the driver might have pushed in the corresponding device receive queue. This message can be an in-band message if it contains a CAN CC/FD/XL message received from the CAN bus, or an out-of-band message if it contains a status message.

```
struct pcanxl_msg {
    __u16 type;          /* PCANXL_TYPE_xxx */
    __u16 data_len;      /* Length of the data */
    __u32 id;            /* CAN CC/FD frame Id */
    __u32 flags;         /* PCANXL_MSG_xxx / PCANFD_MSG_xxx */
    __u8 ctrlr_data[PCANXL_MAXCTRLRDATALEN];
    __u8 sdt;           /* CAN XL SDU Type (see PCANXL_SDT_xxx) */
    __u32 af;           /* */
    struct __kernel_timespec timestamp; /* Message timestamp */
    __u8 data[] __attribute__((aligned(8)));
};
```

IMPORTANT: For performance reasons, “struct pcanxl_msg_xxx” CAN’T be instantiated on the stack (a.k.a. local variable) but must be instantiated in static or dynamic memory (a.k.a. global variable). If, for any reason, a local variable must be used, then user must do a copy of it in global memory or use the helper function `pcanxl_recv_msg_local()` instead.

Note: Reading a message from the driver might block the calling task, unless the device node has been opened in non-blocking mode. In that case, `-EWOULDBLOCK` is returned by this function if the task didn’t find any message to read.

Field	Values	Description
type	PCANFD_TYPE_CAN20_MSG/ PCANXL_TYPE_CANCC	This message is a CAN 2.0 message.
	PCANFD_TYPE_CANFD_MSG/ PCANXL_TYPE_CANFD	This message is a CAN FD message. Bits like <code>PCANFD_MSG_BRS</code> or <code>PCANFD_MSG_ESI</code> can also be set in the flags field.
	PCANXL_TYPE_CANXL	This message is a CAN XL message. <code>sdt</code> , <code>id</code> and <code>af</code> fields are populated accordingly.
	PCANXL_TYPE_STATUS	This message is a status message, giving some more information about the state of the CAN device.
	PCANXL_TYPE_ERROR	This message is an error message read from the CAN bus. This kind of messages ISNOT received by default (see also option <code>PCANFD_ALLOWED_MSG_ERROR</code> in <code>int pcanfd_set_option(int fd, int name, void *value, int size);</code>)
data_len	0..2048	Number of data bytes in the message received from the CAN device. Note that in case of CAN FD, this value might not be the same as the one given on the transmission side.
id		The ID of the CAN CC/FD message. In case of CAN XL message, this field contains the PID and the VCID (see <code>PCANXL_PID()</code> and <code>PCANXL_VCID()</code> macros in <code>pcanxl.h</code>)

Field	Values	Description
flags	PCANXL_MSG_SEC	CAN XL Simple Extended Content message flag.
	PCANXL_MSG_RRS	CAN XL Remote Request Substitute message flag.
	PCANFD_MSG_XXX	See Table 11: Usage of struct pcanfd_msg on the receive side
ctrlr_data		CAN-controller-specific data (see PCANFD_MSG_ECHO, PCANFD_ERRCNT and PCANFD_BUSLOAD flags above).
sdt	PCANXL_SDT_XXX	CAN XL CiA611-1 SDU Type (see PCANXL_SDT_XXX in pcanxl.h)
af		CAN XL Acceptance Field This field is included in the 64-bit hardware acceptance filter of the CAN XL controller. It may contain node address or content-indicating information.
timestamp	nanoseconds	Timestamp of the received message. Unlike libpcan and libpcanfd APIs, this timestamp is given in nanoseconds.
data		The data bytes of the CAN message. The count of data bytes received is given by the data_len field.

Table 17: Usage of struct pcanxl_msg on the receive side

5.2 netdev Mode

The PCAN Driver for Linux is built in *netdev* mode, that is, with:

```
$ make -C driver NET=NETDEV_SUPPORT
```

or:

```
$ make -C driver netdev
```

In this case the user application can neither use `libpcan` nor `libpcanfd` nor `libpcanxl` libraries but must be built over the socket API instead. The programmer can access the online documentation, starting, for example, at these links:

- ↳ en.wikipedia.org/wiki/SocketCAN
- ↳ www.kernel.org/doc/Documentation/networking/can.txt