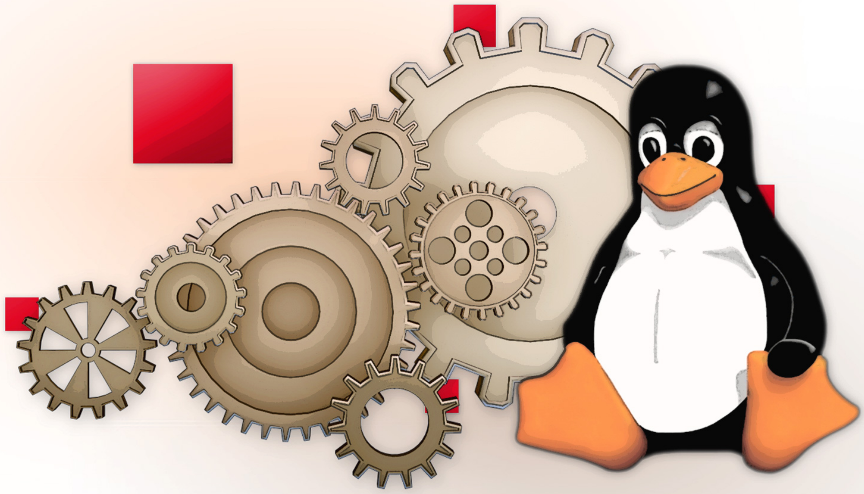
PCAN Driver for Linux v8

CAN Driver and Library API for Linux

User Manual



Document version 3.2.0 (2017-01-09)

Relevant products

Product Name	Version	Part number
PCAN Driver for Linux	8.x.x	not applicable

PEAK-System Technik GmbH
Otto-Roehm-Strasse 69
64293 Darmstadt
Germany

Phone: +49 (0)6151 8173-20
Fax: +49 (0)6151 8173-29

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

Document version 3.2.0 (2017-01-09)

Contents

1 Disclaimer

The provided files 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 of 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 https://www.gnu.org/licenses/.

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

2 Introduction

With the PCAN Driver for Linux, you can use CAN 2.0 and, since v8, CAN FD hardware products from PEAK-System under Linux-based systems. Even if the use of Linux 2.4 kernels is declining, the canonical age of the driver ensures compatibility with some versions of this kernel line and with older PEAK-System hardware products.

The driver is also compatible with the latest versions of well-known real-time (RT) extensions like Xenomai[1] and RTAI[2] , 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 drivers 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 of installing this PCAN Driver for Linux package anymore. The so-called *netdev* interface is however kept for backward compatibility.

Version 8 of the PCAN Driver for Linux is a major evolution since it mainly includes the support of the new CAN FD specification. Because of the new features CAN FD 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 one is obviously always supported.

The package is always evolving, because of the constant support of some new 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:

http://www.peak-system.com/fileadmin/media/linux/index.htm

2.1 Features

- Support of all CAN 2.0 a/b and CAN FD hardware products made by PEAK-System
- Support of all 2.6.x, 3.x, and 4.x Linux Kernels in 32 and 64-bit environments
- `DESTDIR` and cross-compilation supported
- Udev system support
- Enhanced *sysfs* integration
- Optimized character mode device driver interface (*chardev*) supporting CAN 2.0 as well as CAN FD standard and multiple messages transfers between applications and the driver
- SocketCAN device driver interface (*netdev*) supporting CAN 2.0 as well as CAN FD new features, with enhanced NETLINK integration (*ip link* support)

[1] Website Xenomai: https://xenomai.org/
[2] Website RTAI: https://www.rtai.org/

- Real-time Linux extensions like Xenomai 2.6 and RTAI 4.x 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 2.0 *chardev* applications that run over older versions of the driver (7.x and older)

2.2 System Requirements

- Linux-based system running a 32 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
- 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 of that change is that a great number of PEAK-System CAN hardware products are now natively supported by the mainline kernel as SocketCAN interfaces[3]. 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 driver (only) with:

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

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

```
$ make RT=XENOMAI
```

Note: Since driver version 8.2, you can also build the Xenomai binaries with:

```
$ make xeno
```

[3] Kernel code: http://lxr.free-electrons.com/source/drivers/net/can/usb/peak_usb/pcan_usb_core.c?v=3.4

To build real-time binaries running in a RTAI kernel:

```
$ make RT=RTAI
```

Note: Since driver version 8.2, you can also build the RTAI binaries with:

```
$ make rtai
```

Note: Selecting one of the above real-time compilations also removes the support of some of the non-RT PC CAN interfaces (like the 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:

1. the `driver` directory,
2. the `lib` directory, and
3. the `test` directory.

It is equivalent to the following 3 commands:

```
$ make -C driver
$ make -C lib
$ make -C test
```

Making the 32-bit version of the library:

Since driver version 8.3, it is possible to build a 32-bit version of the `libpcan` library while running a 64-bit pcan driver on a 64-bit Kernel:

```
$ make -C lib all32
```

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, in order 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 (default)
	NO_DONGLE_SUPPORT	Remove the support of the parallel port CAN interfaces from the driver
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

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

Table 1: Supported PC CAN interfaces switches

For example, to build the driver without including the support of neither the PCAN-Dongle nor the PCAN-PC Card CAN interfaces:

```
$ make -C driver DNG=NO_DONGLE_SUPPORT PCC=NO_PCCARD_SUPPORT
```

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.

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:

<table>
<tr><th>Parameter</th><th>Type</th><th>Description</th></tr>
<tr><td><code>type</code></td><td>List of characters strings, separated by “,” (comma).</td><td>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><code>isa</code></td><td>ISA and PC/104</td></tr>
<tr><td><code>sp</code></td><td>Standard parallel port</td></tr>
<tr><td><code>epp</code></td><td>Enhanced parallel port</td></tr>
</table></td></tr>
<tr><td><code>io</code></td><td>List of hexadecimal values, separated by”,” (comma).</td><td>Gives the list of I/O ports to use to dialog with the corresponding PC CAN interface (see <code>type</code>).</td></tr>
<tr><td><code>irq</code></td><td>List of decimal values, separated by “,” (comma).</td><td>Gives the list of IRQ levels to connect to dialog with the corresponding PC CAN interface (see <code>type</code>).</td></tr>
<tr><td><code>btr0btr1</code></td><td>Hexadecimal value.</td><td>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).</td></tr>
<tr><td><code>bitrate</code></td><td>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.</td><td>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.</td></tr>
<tr><td><code>dbitrate</code></td><td>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.</td><td>Change the default 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).</td></tr>
<tr><td><code>assign</code></td><td>Characters string.</td><td>Change the default name assignment between PCAN and SocketCAN layer (see 4.8.1 assign Parameter on page 33). This parameter is only used when the netdev interface is selected.</td></tr>
</table>

Table 2: Driver module parameters

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

In order 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 is `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 has to 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/PCIe and USB PC CAN interfaces families is entirely handled by the system.

When using some non-plug-and-play PC CAN interfaces, the driver has to 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 has to 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 Load Driver

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 of these modules manually:

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

$ sudo modprobe pcmcia parport i2C-alog-bits
$ 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
[24612.510888] pcan: Release_YYYYMMMDD_n (le)
[24612.510894] pcan: driver config [mod] [isa] [pci] [pec] [dng] [par] [usb] [pcc]
[24612.511057] pcan: uCAN PCI device sub-system ID 14h (4 channels)
[24612.511125] pcan 0000:01:00.0: irq 48 for MSI/MSI-X
[24612.511140] pcan: uCAN PCB v4h FPGA v1.0.5 (design 3)
[24612.511146] pcan: pci uCAN device minor 0 found
[24612.511148] pcan: pci uCAN device minor 1 found
[24612.511150] pcan: pci uCAN device minor 2 found
[24612.511153] pcan: pci uCAN device minor 3 found
[24612.516206] pcan: pci device minor 4 found
[24612.516230] pcan: pci device minor 5 found
[24612.516258] pcan: pci device minor 6 found
[24612.516280] pcan: pci device minor 7 found
[24612.516335] pcan: isa SJA1000 device minor 8 expected (io=0x0300,irq=10)
[24612.516369] pcan: isa SJA1000 device minor 9 expected (io=0x0320,irq=5)
[24612.516999] pcan: new high speed usb adapter with 2 CAN controller(s) detected
[24612.517237] pcan: PCAN-USB Pro FD (01h PCB01h) fw v2.1.0
[24612.517244] pcan: usb hardware revision = 1
[24612.517605] pcan: PCAN-USB Pro FD channel 1 device number=30
[24612.517729] pcan: usb device minor 0 found
[24612.517732] pcan: usb hardware revision = 1
[24612.518231] pcan: PCAN-USB Pro FD channel 2 device number=31
[24612.518354] pcan: usb device minor 1 found
[24612.522469] pcan: new usb adapter with 1 CAN controller(s) detected
[24612.522491] pcan: usb hardware revision = 28
[24612.579450] pcan: PCAN-USB channel device number=161
[24612.579453] pcan: usb device minor 2 found
[24612.579487] usbcore: registered new interface driver pcan
[24612.586265] pcan: major 249.
```

The driver enumerates each PC CAN interface according to its type. By default, each type has the following range of device minor numbers:

Hardware type	Minor number range
PCI/PCIe	[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

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 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
- unique minor number

For example:

```
$ ls -l /dev/pcan* | grep "^c"
crw-rw-rw- 1 root root 246,  8 févr.  3 14:59 /dev/pcanisa8
crw-rw-rw- 1 root root 246,  9 févr.  3 14:59 /dev/pcanisa9
crw-rw-rw- 1 root root 246,  4 févr.  3 14:59 /dev/pcanpci4
crw-rw-rw- 1 root root 246,  5 févr.  3 14:59 /dev/pcanpci5
crw-rw-rw- 1 root root 246,  0 févr.  3 14:59 /dev/pcanpcifd0
crw-rw-rw- 1 root root 246,  1 févr.  3 14:59 /dev/pcanpcifd1
crw-rw-rw- 1 root root 246,  2 févr.  3 14:59 /dev/pcanpcifd2
crw-rw-rw- 1 root root 246,  3 févr.  3 14:59 /dev/pcanpcifd3
crw-rw-rw- 1 root root 246, 35 févr.  3 15:24 /dev/pcanusb35
crw-rw-rw- 1 root root 246, 36 févr.  3 15:24 /dev/pcanusb36
crw-rw-rw- 1 root root 246, 32 févr.  3 14:59 /dev/pcanusbfd32
crw-rw-rw- 1 root root 246, 33 févr.  3 14:59 /dev/pcanusbfd33
crw-rw-rw- 1 root root 246, 34 févr.  3 14:59 /dev/pcanusbfd34
```

The Udev rules that the driver installs enable 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 are able to).

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 févr.  3 14:59 /dev/pcan0 -> pcanpcifd0
lrwxrwxrwx 1 root root      10 févr.  3 14:59 /dev/pcan1 -> pcanpcifd1
lrwxrwxrwx 1 root root      10 févr.  3 14:59 /dev/pcan2 -> pcanpcifd2
lrwxrwxrwx 1 root root      10 févr.  3 14:59 /dev/pcan3 -> pcanpcifd3
lrwxrwxrwx 1 root root      11 févr.  3 14:59 /dev/pcan32 -> pcanusbfd32
lrwxrwxrwx 1 root root      11 févr.  3 14:59 /dev/pcan33 -> pcanusbfd33
lrwxrwxrwx 1 root root      11 févr.  3 14:59 /dev/pcan34 -> pcanusbfd34
lrwxrwxrwx 1 root root       9 févr.  3 15:24 /dev/pcan35 -> pcanusb35
lrwxrwxrwx 1 root root       9 févr.  3 15:24 /dev/pcan36 -> pcanusb36
lrwxrwxrwx 1 root root       8 févr.  3 14:59 /dev/pcan4 -> pcanpci4
lrwxrwxrwx 1 root root       8 févr.  3 14:59 /dev/pcan5 -> pcanpci5
lrwxrwxrwx 1 root root       8 févr.  3 14:59 /dev/pcan8 -> pcanisa8
lrwxrwxrwx 1 root root       8 févr.  3 14:59 /dev/pcan9 -> pcanisa9
crw-rw-rw- 1 root root 246,  8 févr.  3 14:59 /dev/pcanisa8
crw-rw-rw- 1 root root 246,  9 févr.  3 14:59 /dev/pcanisa9
crw-rw-rw- 1 root root 246,  4 févr.  3 14:59 /dev/pcanpci4
crw-rw-rw- 1 root root 246,  5 févr.  3 14:59 /dev/pcanpci5
crw-rw-rw- 1 root root 246,  0 févr.  3 14:59 /dev/pcanpcifd0
crw-rw-rw- 1 root root 246,  1 févr.  3 14:59 /dev/pcanpcifd1
crw-rw-rw- 1 root root 246,  2 févr.  3 14:59 /dev/pcanpcifd2
crw-rw-rw- 1 root root 246,  3 févr.  3 14:59 /dev/pcanpcifd3
crw-rw-rw- 1 root root 246, 35 févr.  3 15:24 /dev/pcanusb35
crw-rw-rw- 1 root root 246, 36 févr.  3 15:24 /dev/pcanusb36
crw-rw-rw- 1 root root 246, 32 févr.  3 14:59 /dev/pcanusbfd32
crw-rw-rw- 1 root root 246, 33 févr.  3 14:59 /dev/pcanusbfd33
crw-rw-rw- 1 root root 246, 34 févr.  3 14:59 /dev/pcanusbfd34
lrwxrwxrwx 1 root root      11 févr.  3 14:59 /dev/pcanusbpfd32 -> pcanusbfd32
lrwxrwxrwx 1 root root      11 févr.  3 14:59 /dev/pcanusbpfd33 -> pcanusbfd33

/dev/pcan-pci:
total 0
drwxr-xr-x 2 root root 80 févr.  3 14:59 0

/dev/pcan-pcie_fd:
total 0
drwxr-xr-x 2 root root 80 févr.  3 14:59 0
drwxr-xr-x 2 root root 80 févr.  3 14:59 1

/dev/pcan-usb:
total 0
drwxr-xr-x 2 root root 60 févr.  3 15:24 0
drwxr-xr-x 2 root root 60 févr.  3 15:24 1
lrwxrwxrwx 1 root root 12 févr.  3 15:24 devid=161 -> ../pcanusb35

/dev/pcan-usb_fd:
total 0
drwxr-xr-x 2 root root 60 févr.  3 14:59 0
lrwxrwxrwx 1 root root 14 févr.  3 14:59 devid=12345678 -> ../pcanusbfd34

/dev/pcan-usb_pro_fd:
total 0
drwxr-xr-x 2 root root 80 févr.  3 14:59 0
lrwxrwxrwx 1 root root 14 févr.  3 14:59 devid=2 -> ../pcanusbfd32
lrwxrwxrwx 1 root root 14 févr.  3 14:59 devid=31 -> ../pcanusbfd33
```

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 -> ../../pcanpci4
    └── can1 -> ../../pcanpci5

1 directory, 2 files
```

```
$ tree /dev/pcan-pcie_fd
/dev/pcan-pcie_fd
├── 0
│   ├── can0 -> ../../pcanpcifd0
│   └── can1 -> ../../pcanpcifd1
└── 1
    ├── can0 -> ../../pcanpcifd2
    └── can1 -> ../../pcanpcifd3

2 directories, 4 files
```

```
$ tree /dev/pcan-usb
/dev/pcan-usb
├── 0
│   └── can0 -> ../../pcanusb35
├── 1
│   └── can0 -> ../../pcanusb36
└── devid=161 -> ../pcanusb35

2 directories, 3 files
```

```
$ tree /dev/pcan-usb_fd
/dev/pcan-usb_fd
├── 0
│   └── can0 -> ../../pcanusbfd34
└── devid=12345678 -> ../pcanusbfd34

1 directory, 2 files
```

```
$ tree /dev/pcan-usb_pro_fd
/dev/pcan-usb_pro_fd
├── 0
│   ├── can0 -> ../../pcanusbfd32
│   └── can1 -> ../../pcanusbfd33
├── devid=2 -> ../pcanusbfd32
└── devid=31 -> ../pcanusbfd33

1 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=31`
- `/dev/pcan-usb_pro_fd/0/can1`

Note: With a properly configured and running Udev system, all of 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 so, read the `/proc/pcan` pseudo file.

Example:

```
$ cat /proc/pcan

*------------- PEAK-System CAN interfaces (www.peak-system.com) -------------
*------------- Release_YYYYMMDD_n (X.Y.Z) MMM DD YYYY HH:MN:SS --------------
*------------- [mod] [isa] [pci] [pec] [dng] [par] [usb] [pcc] --------------
*--------------------- XX interfaces @ major 249 found -----------------------
*n -type- -ndev- --base-- irq --btr- --read-- --write- --irqs-- -errors- status
 0  pcifd   -NA- f8c21000 048 0x001c 00000000 00000000 00000000 00000000 0x0000
 1  pcifd   -NA- f8c22000 048 0x001c 00000000 00000000 00000000 00000000 0x0000
 2  pcifd   -NA- f8c23000 048 0x001c 00000000 00000000 00000000 00000000 0x0000
 3  pcifd   -NA- f8c24000 048 0x001c 00000000 00000000 00000000 00000000 0x0000
 4    pci   -NA- fdee0000 016 0x001c 00000000 00000000 00000000 00000000 0x0000
 5    pci   -NA- fdee0400 016 0x001c 00000000 00000000 00000000 00000000 0x0000
 6    pci   -NA- fdee0800 016 0x001c 00000000 00000000 00000000 00000000 0x0000
 7    pci   -NA- fdee0c00 016 0x001c 00000000 00000000 00000000 00000000 0x0000
 8    isa   -NA-      300 010 0x001c 00000000 00000000 00000000 00000000 0x0000
 9    isa   -NA-      320 005 0x001c 00000000 00000000 00000000 00000000 0x0000
32  usbfd   -NA-        3 030 0x001c 00000000 00000000 00000000 00000000 0x0000
33  usbfd   -NA-        3 031 0x001c 00000000 00000000 00000000 00000000 0x0000
34    usb   -NA- ffffffff 161 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)
	usbfd	USB interface equipped with a CAN FD FPGA (PCAN-USB FD)
	pcifd	PCI/PCIe based interface equipped with a CAN FD FPGA
ndev	canx	If the *netdev* 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 *chardev* 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 `devid` set to the PC CAN interface, if the PC CAN interface is an USB interface
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 an USB CAN interface
errors	hexadecimal value	Number of errors encountered by the driver for this interface. This counter handles all kind of errors (controller error as well as driver internal errors). Some more information about errors is given in the `status` column
status	bit mask	The signification of each error bit is defined by the CAN_ERR_xxx constants defined in /usr/include/pcan.h.

Table 4: `/proc/pcan` columns

4.4 /sysfs Interface

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

For historical reasons, v8.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 of 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
8.0.0
```

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

```
$ tree -a /sys/class/pcan
/sys/class/pcan
├── pcanisa8 -> ../../devices/virtual/pcan/pcanisa8
├── pcanisa9 -> ../../devices/virtual/pcan/pcanisa9
├── pcanpci4 -> ../../devices/virtual/pcan/pcanpci4
├── pcanpci5 -> ../../devices/virtual/pcan/pcanpci5
├── pcanpcifd0 -> ../../devices/virtual/pcan/pcanpcifd0
├── pcanpcifd1 -> ../../devices/virtual/pcan/pcanpcifd1
├── pcanpcifd2 -> ../../devices/virtual/pcan/pcanpcifd2
├── pcanpcifd3 -> ../../devices/virtual/pcan/pcanpcifd3
├── pcanusb35 -> ../../devices/virtual/pcan/pcanusb35
├── pcanusb36 -> ../../devices/virtual/pcan/pcanusb36
├── pcanusbfd32 -> ../../devices/virtual/pcan/pcanusbfd32
├── pcanusbfd33 -> ../../devices/virtual/pcan/pcanusbfd33
├── pcanusbfd34 -> ../../devices/virtual/pcan/pcanusbfd34
└── version
```

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/pcanpci4/
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_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 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
-r--r--r-- 1 root root 4096 nov.   6 12:34 devid
-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 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_fifo_ratio
-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 tx_fifo_ratio
-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
```

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

```
$ for f in /sys/class/pcan/pcanpci4/*; do [ -f $f ] && echo -n "`basename $f` =
" && cat $f; done
adapter_name = PCAN-PCI
adapter_number = 0
adapter_version =
base = 0xfdee0000
btr0btr1 = 0x001c
bus_state = 0
clock = 8000000
ctrlr_number = 0
dev = 249:4
devid = 4294967295
errors = 0
hwtype = 10
init_flags = 0x00000000
irq = 16
irqs = 0
minor = 4
nom_bitrate = 500000
nom_brp = 1
nom_sjw = 1
nom_tq = 125
nom_tseg1 = 13
nom_tseg2 = 2
read = 0
rx_fifo_ratio = 0.00
status = 0x0000
tx_fifo_ratio = 0.00
type = pci
uevent = MAJOR=249
MINOR=4
DEVNAME=pcanpci4
write = 0
```

Note: Depending on the CAN hardware, the device node might export some more properties. For example, a CAN FD PCIe device will export the following properties (new 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_version = 2.1.3
base = 0xf8ba1000
btr0btr1 = 0x001c
bus_load = 0
bus_state = 0
clock = 80000000
ctrlr_number = 1
data_bitrate = 2000000
data_brp = 2
data_sample_point = 7500
data_sjw = 1
data_tq = 25
data_tseg1 = 14
data_tseg2 = 5
dev = 249:1
devid = 4294967295
```

```
errors = 0
hwtype = 19
irq = 48
irqs = 0
minor = 0
nom_bitrate = 500000
nom_brp = 4
nom_sample_point = 8750
nom_sjw = 1
nom_tq = 50
nom_tseg1 = 34
nom_tseg2 = 5
read = 0
rx_error_counter = 0
rx_fifo_ratio = 0.00
status = 0x0000
tx_error_counter = 0
tx_fifo_ratio = 0.00
type = pcifd
uevent = MAJOR=249
MINOR=1
DEVNAME=pcanpcifd1
write = 0
```

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/CANFD devices found by 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
     --version     display driver version
```

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

```
$ ./lspcan -T -t -i
dev name        port    irq     clock   btrs    bus
[PCAN-ISA 0]
 |_ pcanisa8    CAN1    10      8MHz    500k    CLOSED
 |_ pcanisa9    CAN2    5       8MHz    500k    CLOSED
[PCAN-PCI 0]
 |_ pcanpci4    CAN1    19      8MHz    500k    CLOSED
 |_ pcanpci5    CAN2    19      8MHz    500k    CLOSED
[PCAN-PCIe FD 0]
 |_ pcanpcifd0  CAN1    32      80MHz   500k+2M CLOSED
 |_ pcanpcifd1  CAN2    32      80MHz   500k+2M CLOSED
[PCAN-PCIe FD 1]
 |_ pcanpcifd2  CAN1    33      80MHz   500k+2M CLOSED
 |_ pcanpcifd3  CAN2    33      80MHz   500k+2M CLOSED
[PCAN-USB 0]
 |_ pcanusb32   CAN1    -       8MHz    500k    CLOSED
[PCAN-USB 1]
 |_ pcanusb33   CAN1    -       8MHz    500k    CLOSED
[PCAN-USB Pro FD 0]
 |_ pcanusbfd34 CAN1    -       80MHz   500k+2M CLOSED
 |_ pcanusbfd35 CAN2    -       80MHz   500k+2M CLOSED
```

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:

```
PCAN driver version: 8.x.y
dev name        port    irq     clock   btrs    bus     %bus    rx      tx
err
[PCAN-ISA 0]
 |_ pcanisa8    CAN1    10      8MHz    500k    CLOSED  -       0       0       0
 |_ pcanisa9    CAN2    5       8MHz    500k    CLOSED  -       0       0       0
[PCAN-PCI 0]
 |_ pcanpci4    CAN1    19      8MHz    500k    CLOSED  -       0       0       0
 |_ pcanpci5    CAN2    19      8MHz    500k    CLOSED  -       0       0       0
[PCAN-PCIe FD 0]
 |_ pcanpcifd0  CAN1    30      80MHz   500k+2M CLOSED  0.00    0       0       0
 |_ pcanpcifd1  CAN2    30      80MHz   500k+2M CLOSED  0.00    0       0       0
[PCAN-PCIe FD 1]
 |_ pcanpcifd2  CAN1    31      80MHz   500k+2M CLOSED  0.00    0       0       0
 |_ pcanpcifd3  CAN2    31      80MHz   500k+2M CLOSED  0.00    0       0       0
[PCAN-USB 0]
 |_ pcanusb35   CAN1    -       8MHz    500k    CLOSED  -       0       0       0
[PCAN-USB 1]
 |_ pcanusb36   CAN1    -       8MHz    1M      PASSIVE -       535608  0
585
[PCAN-USB Pro FD 0]
 |_ pcanusbfd32 CAN1    -       80MHz   500k+2M CLOSED  0.00    0       0       0
 |_ pcanusbfd33 CAN2    -       80MHz   1M      ACTIVE  10.01   1       535634  0
[PCAN-USB FD 0]
 |_ pcanusbfd34 CAN1    -       80MHz   500k+2M CLOSED  0.00    0       0       0
```

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

4.6 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 to read/write from/to the channel. In *chardev* mode, the read/write entries of the driver's *chardev* interface are able to:

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

Command and parameters must be separated by blank characters.

<table>
<tr><th>Command</th><th>Parameter</th><th>Description</th></tr>
<tr><td>i</td><td>xxxx</td><td>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.</td></tr>
<tr><td></td><td>param1=value1[,param2=value2…]</td><td>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
nom_tseg1
nom_tseg2
nom_sjw</td><td>The bit timing specifications for the nominal bitrate, as defined by ISO 11898.</td></tr>
<tr><td>data_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>data_brp
data_tseg1
data_tseg2
data_sjw</td><td>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>
</table>
Each value is a numeric value. Unit symbol like k or M can be used as shortcut.

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.</td></tr>
</table>

Command	Parameter	Description
`m`	`s id len [xx [xx …]]`	Write CAN **s**tandard **m**essage `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 **e**xtended **m**essage `id` (numeric value <= 0x3fffffff) 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 **R**TR (Remote Transmission Request) of **s**tandard `id` (numeric value <= 0x7ff).
	`e id`	Write the CAN **R**TR (Remote Transmission Request) of **e**xtended `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 has the ability to 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 has the ability to 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 has the ability to copy an outgoing CAN FD frame to its own rx queue).	

Table 5: read/write interface syntax

If reading from this interface, the user is able to 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 6: Status (`x`) message

4.7 test Directory

The PCAN Driver for Linux package includes a `test` directory that contains the C/C++ sources and Makefile enabling to quickly build and run some simple test binary applications, in order to check if the entire *chardev* installation (driver and libraries) is fully operational. 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.

The `test` directory applications should be built after the libraries under `lib` directory have been built and installed. Like the driver, these libraries and applications accept non-RT and RT compilation.

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 (without using any RT system calls):

```
$ 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 all32
```

Note: A 32-bit version of `libpcan` must have been built and installed first (see *Making the 32-bit version of the* library on page 7). Moreover, in order 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 4.7.5 on page 29.

4.7.1 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 `lipcan` 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=devicenode] | {[-t=type] [-p=port [-i=irq]]}}
options:
-f=devicenode   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, 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.

4.7.2 transmitest

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

Usage:

```
$ transmitest --help

transmitest Version "Release_20150610_n"  (www.peak-system.com)
------- Copyright (C) 2004-2009 PEAK System-Technik GmbH ------
transmitest 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.

transmitest - a small test program which transmits CAN messages.
usage: transmitest filename
                   [-b=BTR0BTR1] [-e] [-r=msec] [-n=max] [-?]
                   {[-f=devicenode] | {[-t=type] [-p=port [-i=irq]]}}
filename        mandatory name of message description file.
options:
-f=devicenode   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

transmitest: 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 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, 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.

4.7.3 pcan-settings

This application enables 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...).

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 No
  -d, --DeviceNo[='non-volatile device number']   get or set device No
  -v, --verbose                                   make it verbose

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 `/sys`:

```
$ 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
```

4.7.4 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=devicenode] [-?]
         -f=devicenode - path to devicefile, default=/dev/pcan0
         -? or --help  - this help

bitratetest: finished (0).
```

4.7.5 pcanfdtst

This application enables to test the driver, since it is able to receive/transmit CAN 2.0/CAN FD messages from/to all of the device nodes handled by the driver. It works in two modes:

- when running in RX mode, the application only writes everything received from all the opened CAN FD device nodes on the screen
- when running in TX mode, the application transmits CAN FD frames on all the opened devices and also displays any event received from them

Moreover, this application demonstrates the usage of the new CAN FD 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 multi-messages transmit/receive
- demonstrate the new event-based API

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] CAN [CAN...]

MODE
        tx    generate CAN traffic on the specified CAN interfaces
        rx    check CAN traffic received on the specified CAN interfaces

CAN
        /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.

OPTIONS
        -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           data bitrate used for transmitting CANFD msgs
        -c | --clock v       select clock frequency "v" Hz
        -D | --debug         (maybe too) lot of display
        -d | --dbitrate v    set data bitrate to "v" bps
        -f | --fd            select CAN FD ISO mode
        -F | --fd-non-iso    select CAN FD non-ISO mode
        -h | --help          display this help
        -i | --id v|r        set fixed CAN ID "v" or randomly
        -is v|r              set fixed standard CAN ID "v" or randomly
        -ie v|r              set fixed extented CAN ID "v" or randomly
        -I | --incr v        "v"=nb of data bytes to use for increment counter
        -l | --len v         set fixed CAN dlc "v" for tests
        -m | --mul v         tx/rx "v" msgs at once
        -n v                 do "v" test loops and stop
        -o | --listen-only   set PCAN device in listen-only mode
        -p | --pause-us v    set "v" us. pause between tests
        -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
        -t | --timeout-ms v  wait "v" ms. for events
        -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)
```

- 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.

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
     0.001518 /dev/pcanusb1 > BUS STATE=ACTIVE  [Rx:0 Tx:0]
     0.4293989212 /dev/pcanusb1 <      567 ..... [00 00 00 00 00 00 00]
     1.4293989342 /dev/pcanusb1 <      069 ..... [00 00 00 00 00 00 00]
     2.4293989614 /dev/pcanusb1 <      451 ..... [00 00 00 00 00 00 00]
     3.4293989798 /dev/pcanusb1 <      44a ..... [00 00 00]
     4.4293989995 /dev/pcanusb1 <      729 ..... [00]
     5.4293990176 /dev/pcanusb1 <      0ba ..... [00 00 00 00]
     6.4293990468 /dev/pcanusb1 <      1f2 ..... [00 00 00 00 00 00 00]
     7.4293990660 /dev/pcanusb1 <      1e3 ..... [00 00 00 00]
     8.4293990845 /dev/pcanusb1 <      07c .....
     9.4293991023 /dev/pcanusb1 <      054 ..... [00]
/dev/pcanusb1 < [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... [00 00 00 00 00 00 00 00 00 00 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... [00 00 00 00 00 00 00 00 00 00 00 00]
                                                [00 00 00 00 00 00 00 00 00 00 00 00]
     0.024662 /dev/pcanusbfd33 < 00000123 .e... [00 00 00 00 00 00 00 00 00 00 00 00]
                                                [00 00 00 00 00 00 00 00 00 00 00 00]
     0.025754 /dev/pcanpcifd0  < 00000123 .e... [00 00 00 00 00 00 00 00 00 00 00 00]
                                                [00 00 00 00 00 00 00 00 00 00 00 00]
…
```

3. 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... [00 00 00 00 00 00 00 00 00 00 00 00]
                                              [00 00 00 00 00 00 00 00 00 00 00 00]
    14.764041 /dev/pcanusb32 > 00000123 .e... [00 00 00 00 00 00 00 00 00 00 00 00]
                                              [00 00 00 00 00 00 00 00 00 00 00 00]
    14.766249 /dev/pcanusb32 > 00000123 .e... [00 00 00 00 00 00 00 00 00 00 00 00]
                                              [00 00 00 00 00 00 00 00 00 00 00 00]
…
```

4. 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
     0.000283 /dev/pcanpcifd0 < 00000123 .e... [00 00 00 00]
     0.000000 /dev/pcanpcifd0 > BUS STATE=ACTIVE  [Rx:0 Tx:0]
     0.001426 /dev/pcanpcifd0 < 00000123 .e... [01 00 00 00]
     0.002528 /dev/pcanpcifd0 < 00000123 .e... [02 00 00 00]
     0.003675 /dev/pcanpcifd0 < 00000123 .e... [03 00 00 00]
     0.005042 /dev/pcanpcifd0 < 00000123 .e... [04 00 00 00]
     0.006147 /dev/pcanpcifd0 < 00000123 .e... [05 00 00 00]
     0.007252 /dev/pcanpcifd0 < 00000123 .e... [06 00 00 00]
     0.008349 /dev/pcanpcifd0 < 00000123 .e... [07 00 00 00]
     0.009457 /dev/pcanpcifd0 < 00000123 .e... [08 00 00 00]
     0.010564 /dev/pcanpcifd0 < 00000123 .e... [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... [00 00 00 00]
     8.714307 /dev/pcanusbfd32 > 00000123 .e... [00 00 00 00]
     8.714424 /dev/pcanusbfd32 > 00000123 .e... [00 00 00 00]
     8.714540 /dev/pcanusbfd32 > 00000123 .e... [01 00 00 00]
     8.714656 /dev/pcanusbfd32 > 00000123 .e... [01 00 00 00]
     8.714772 /dev/pcanusbfd32 > 00000123 .e... [01 00 00 00]
     8.715402 /dev/pcanusbfd32 > 00000123 .e... [02 00 00 00]
     8.715518 /dev/pcanusbfd32 > 00000123 .e... [02 00 00 00]
     8.715634 /dev/pcanusbfd32 > 00000123 .e... [02 00 00 00]
     8.716552 /dev/pcanusbfd32 > 00000123 .e... [03 00 00 00]
     8.716668 /dev/pcanusbfd32 > 00000123 .e... [03 00 00 00]
…
```

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

4.8 netdev Mode

If the PCAN driver for Linux has been built for SocketCAN[4] usage (that is, 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 applications.

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.

4.8.1 assign Parameter

The `assign` parameter of the driver (described in Table 2: Driver module parameters on page 10) 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[,pcanX=canY]` defines the name `canY` to the device which name is `pcanX`.

assign=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 4.7.3 *pcan-settings* on page 28).

[4] Background information: https://en.wikipedia.org/wiki/SocketCAN

4.8.2 ifconfig/iproute2

Both of these utilities configure a `canX` interface. While `ifconfig` is somewhere too old to support all of the CAN/CAN-FD-specific features, the last versions of the `iproute2` package (especially the `ip` tool) include options to setup a `canX` interface. Since v8, the `canX` interfaces exported by the 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 (aka `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: https://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 te 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
```

Note: The latest version of `iproute2` package can be downloaded from:
https://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
```

4.8.3 can-utils

The `can-utils` package[5] 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 4.8.2 *ifconfig/iproute2* on page 34).

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
```

[5] Website can-utils: https://github.com/linux-can/can-utils/

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:

- `lipcan` is the good and old API which is always offering access to CAN 2.0 channels (see 5.1.1 *CAN 2.0 API* on page 37)
- `libpcanfd` is the new API included in the package since version 8 of the driver. This new 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 API* on page 37, this library can also be linked with CAN 2.0 API applications instead of using `libpcan`.

Both of these libraries can be built for being used by real-time applications. Two RT environments can be selected when building these libraries:

➡ 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/CAN FD 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 a major number by the system.

Each device node can be opened, closed, read, and written (see 4.6 *read/write Interface* on page 23). 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 below.

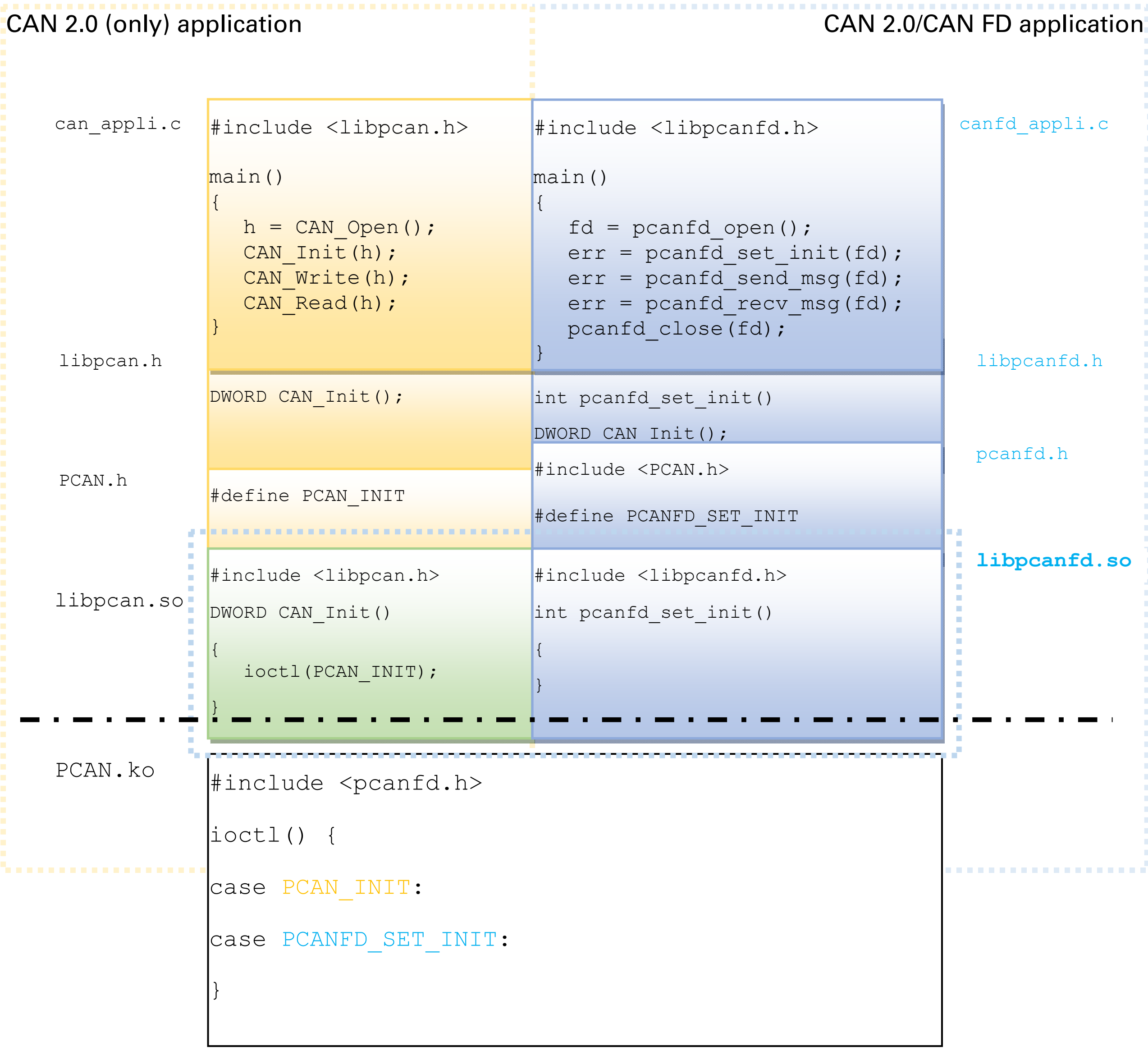


Figure 1: software components architecture

5.1.1 CAN 2.0 API

Note: This API is kept for backward compatibility reasons, thus these entry points are also proposed by the new `libpcanfd` library. But, this API is considered as deprecated. Use the new CAN FD 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 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 `transmitest`, `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`.

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 of 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 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 has to 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 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 API* on page 37), but adds definition of some new data structures and ioctl codes (see `pcanfd.h`). The old entry points always allow connecting to the CAN 2.0 bus as usual, while the new ones enable to connect to CAN 2.0 and/or CAN FD busses. In other words, the new API is a new, 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, \
                                        struct pcanfd_init)
#define PCANFD_GET_INIT          _IOR(PCAN_MAGIC_NUMBER, PCANFD_SEQ_GET_INIT, \
                                        struct pcanfd_init)
#define PCANFD_GET_STATE         _IOR(PCAN_MAGIC_NUMBER, PCANFD_SEQ_GET_STATE, \
                                        struct pcanfd_state)
#define PCANFD_ADD_FILTERS        _IOW(PCAN_MAGIC_NUMBER, PCANFD_SEQ_ADD_FILTERS, \
                                        struct pcanfd_msg_filters)
#define PCANFD_GET_FILTERS        _IOW(PCAN_MAGIC_NUMBER, PCANFD_SEQ_GET_FILTERS, \
                                        struct pcanfd_msg_filters)
#define PCANFD_SEND_MSG          _IOW(PCAN_MAGIC_NUMBER, PCANFD_SEQ_SEND_MSG, \
                                        struct pcanfd_msg)
#define PCANFD_RECV_MSG          _IOR(PCAN_MAGIC_NUMBER, PCANFD_SEQ_RECV_MSG, \
                                        struct pcanfd_msg)
#define PCANFD_SEND_MSGS         _IOWR(PCAN_MAGIC_NUMBER, PCANFD_SEQ_SEND_MSGS, \
                                        struct pcanfd_msgs)
#define PCANFD_RECV_MSGS         _IOWR(PCAN_MAGIC_NUMBER, PCANFD_SEQ_RECV_MSGS, \
                                        struct pcanfd_msgs)
#define PCANFD_GET_AVAILABLE_CLOCKS      _IOWR(PCAN_MAGIC_NUMBER,\
                                              PCANFD_SEQ_GET_AVAILABLE_CLOCKS,\
                                              struct pcanfd_available_clocks)

#define PCANFD_GET_BITTIMING_RANGES      _IOWR(PCAN_MAGIC_NUMBER,\
                                              PCANFD_SEQ_GET_BITTIMING_RANGES,\
                                              struct pcanfd_bittiming_ranges)
```

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 new 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 old and new 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 new 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 of the entry points of this new 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 to select CAN 2.0 as well as CAN FD properties (if the corresponding hardware is compatible). These properties are defined by the new `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 opened 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 is able to 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 device hardware 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 above).
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 above).

Table 7: 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 to filter among incoming 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 is able to 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_CAN_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 hardware to its internal receive queue.
	`PCANFD_MSG_SNG`	If supported, this message is transmit 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.
`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.

Table 8: Usage of struct pcanfd_msg on the transmit side

int pcanfd_send_msgs(int fd, struct pcanfd_msgs *pfdml);

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 is able to 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 *pfdml);”.

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 `PCANFD_MSG_BRS` or `PCANFD_MSG_ESI` 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.
`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_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 `PCANFD_TIMESTAMP` 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`	ctrlr_data[PCANFD_RXERRCNT] and ctrlr_data[PCANFD_TXERRCNT] contain Rx and Tx error counters read from the CAN controller.
	`PCANFD_BUSLOAD`	ctrlr_data[PCANFD_BUSLOAD_UNIT] contains the percentage of the bus load computed by the hardware controller, while ctrlr_data[PCANFD_BUSLOAD_DEC] contains the decimal part.
`timestamp`	`struct timeval`	If `PCANFD_TIMESTAMP` is set in the flag field, then this one indicates the moment the message has been received. If `PCANFD_HWTIMESTAMP` is also set, the given moment is a time made from the hardware clock. If `PCANFD_HWTIMESTAMP` is not set, this moment is made by the driver, from the host current time.
`ctrlr_data`		CAN-controller-specific data (see `PCANFD_ERRCNT` and `PCANFD_BUSLOAD` flags above).
`data`		The data bytes of the CAN message. The count of data bytes received is given by the `data_len` field.

Table 9: Usage of struct pcanfd_msg on the receive side

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

This function is able to 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 is able to carry several CAN 2.0 and CAN FD messages. The maximum number of messages the list is able to 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);
	}

	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 to setup 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_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 10 describes the order and how each bit of the `flags` argument is interpreted:

Bit	Description
OFD_BITRATE	The specification of the nominal bitrate starts with the third parameter: If OFD_BTR0BTR1 is set too, then the third parameter is interpreted as a 16-bit value respecting the BTR0BTR1 SJA1000 format. If OFD_BRPTSEGSJW 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.
OFD_SAMPLEPT	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 %)
OFD_DBITRATE	The data bitrate is given in the next arguments: If OFD_BTR0BTR1 is set too, then the next parameter is interpreted as a 16-bit value respecting the BTR0BTR1 SJA1000 format. If OFD_BRPTSEGSJW 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.
OFD_DSAMPLEPT	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 %)
OFD_CLOCKHZ	The clock frequency (in Hz) to select in the CAN controller is given in the next argument.
OFD_NONBLOCKING	The device node is opened in non-blocking mode.
PCANFD_INIT_xxx	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 10: 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);
…
```

5.2 netdev Mode

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

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

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

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