can4linux — The Linux CAN device driver

can4linux is a Linux CAN device driver for different CAN interface boards or embedded Linux devices having controllers with integrated CAN controllers.

Supported CAN controllers

Stand alone CAN controllers

- Intel 82527
- Microchip MCP2515 via SPI
- NXP SJA1000/82C200
- IFI CAN FD IP (FPGA)

Integrated CAN controllers

- Freescale FlexCAN (i.MX, ColdFire)
- Analog Devices BlackFin
- ATMEL
- Allwinner A20
- Xilinx XCAN

Since version 4.0, released July 2012, can4linux can be used in a **virtual CAN** mode without the need of a installed CAN hardware and it is prepared for the new CAN FD standard, presented by BOSCH in March 2012.

The virtual CAN mode is enabled by setting the module parameter
`insmod can4linux.ko virtual=1`
or
`make simload`
To support CAN FD compile with `-DCANFD`.

1. Installation

Take care that the kernel sources and configuration files are available. It is useful to have loadable module support enabled. The *sysctl* interface must be enabled as well. *.config* must contain `CONFIG_PROC_SYSCTL=y`.

To get the latest files stored on SourceForge, check out using **svn**:
`svn co https://can4linux.svn.sourceforge.net/svnroot/can4linux can4linux`

Alternatively download can4linux, create a can4linux directory, e.g. *can4linux* and unpack the archive:
```
$ mkdir can4linux
$ cd can4linux
$ tar zxvf ../can4linux*tgz
```
The driver has to be compiled for each supported hardware. A list of supported boards can be found in the Makefile. It is displayed by typing
`$ make help`
Compile the driver with specifying the hardware target:
`$ make TARGET=ATCANMINI_PELICAN`[1]
The driver object file *can4linux.ko* is created in the actual working directory.

Now create the necessary device entries by using make again:
`$ make inodes`
The default value for a CAN drivers major number is 91. Please check before using it if the major number is already used. (major 91 is a registered number for CAN drivers.)
`$ cat /proc/devices`
lists all devices major numbers in use.

[1] `make help` will give a list of available TARGET values

Using *.conf files

Older ISA boards or PC104 like boards are equipped with jumpers to set there CAN controllers base address and IRQ lines. The values set by jumpers have to used by the driver. This can be done by writing to the **Base** and **IRQ** and **IOModel** entries in /proc/sys/dev/Can. Alternatively there is already a mechanism using a shell script in the etc/ directory.

Using this, next go into *etc*. Look for a configuration that fits to your hardware.

In the most cases you can use one of the available: *2-at_can_mini.conf* or *1-cpcpci.conf* .

If you don´t see any pre-configured settings that match your hardware read your hardware manuals carefully and add a new entry.

Create a configuration file named according to the name of your computer. Your computers name is returned:

```
$ uname -n
uschi
$ cp 1-cpcpci.conf uschi.conf
```

Some entries are hardware dependant. Be careful when using your own hardware.

The content in the configuration file is used to overwrite the appropriate entries in the */proc* file system.

`/proc/sys/dev/Can/*`

Now you can do a

```
$ make load
```

The driver **can4linux.ko** is loaded using `insmod`(1) and entries in */proc/sys/dev/Can/** are overwritten with the config file contents.

ATTENTION! When using PCI boards, the driver is using values obtained from the BIOS for addresses, access type and IRQ numbers. In this case these values are read-only and can not be overwritten. Ignore the warnings given while loading a configuration.

1.1. Installing a second can4linux driver

Sometimes it might be necessary to have two different CAN controllers supported. This will mostly happen on PC like systems, equipped with two different PCI boards for example. In this case select one TARGET as your default one. This one then is using the default device major number 91 and the default module name *can4linux.ko*.

To compile a second (or more) can4linux driver, add the following lines in the target specific section of the Makefile

```
ifeq "$(TARGET)" "CC_CANPCI"

CAN_MODULE_POSTFIX = _cc
CAN_MODULE := $(CAN_MODULE)$(CAN_MODULE_POSTFIX)
CAN_MAJOR  = 92

# Contemporary Controls
# CC PCI PeliCAN  PCI (only with SJA1000) ----------------------------------
# CANPCI-CO and CANPCI-DN
DEFS =  -D$(TARGET) -D$(DEBUG) -DDEFAULT_DEBUG -DCAN_MAJOR=$(CAN_MAJOR)\
    -DCAN4LINUX_PCI \
    -DCAN_PORT_IO   \
    -DCAN_SYSCLK=8   \
    -DMAX_CHANNELS=1 -DCAN_MAX_OPEN=2\
    -DCAN_MODULE_POSTFIX=\"$(CAN_MODULE_POSTFIX)\"

TARGET_MATCHED = true
endif
```

2. Test

Change into the directory *examples.* and compile the applications there

```
$ cd examples
$ make
gcc -Wall -I../src -DUSE_RT_SCHEDULING   -c -o ctest.o ctest.c
gcc   ctest.o   -o ctest
gcc -Wall -I../src -DUSE_RT_SCHEDULING   -c -o baud.o baud.c
gcc   baud.o   -o baud
gcc -Wall -I../src -DUSE_RT_SCHEDULING   -c -o can_send.o can_send.c
gcc   can_send.o   -o can_send
gcc -Wall -I../src -DUSE_RT_SCHEDULING   -c -o acceptance.o acceptance.c
gcc   acceptance.o   -o acceptance
gcc -Wall -I../src -DUSE_RT_SCHEDULING   -c -o noiser.o noiser.c
gcc   noiser.o   -o noiser
gcc -Wall -I../src -DUSE_RT_SCHEDULING   -c -o receive.o receive.c
gcc   receive.o   -o receive
gcc -Wall -I../src -DUSE_RT_SCHEDULING   -c -o transmit.o transmit.c
gcc   transmit.o   -o transmit
gcc -Wall -I../src -DUSE_RT_SCHEDULING   -c -o can_verify.o can_verify.c
gcc   can_verify.o   -o can_verify
```

Before calling `ctest` watch the message log of the driver in the file */var/log/messages.* Open a separate window **xterm**

```
$ tail -f /var/log/messages
```

(super user access rights are needed on most systems)

In order to see more messages increase the debug level by writing to the /proc filesystem

```
echo 7 > /proc/sys/dev/Can/dbgMask
```

(super user access rights are needed on most systems)

or use the shell script

```
./debug 7
```

After the start of `ctest` you will see the following in the message log

```
Sep 17 13:13:31 uschi kernel: Can: - :in  can_open
Sep 17 13:13:31 uschi kernel: Can: - :in  CAN_VendorInit
Sep 17 13:13:31 uschi kernel: Can: - :in  Can_RequestIrq
Sep 17 13:13:31 uschi kernel: Can: - :Requested IRQ: 5 @ 0xce2e28c0
Sep 17 13:13:31 uschi kernel: Can: - :out
Sep 17 13:13:31 uschi kernel: Can: - :out
Sep 17 13:13:31 uschi kernel: Can: - :in  Can_WaitInit
Sep 17 13:13:31 uschi kernel: Can: - :out
Sep 17 13:13:31 uschi kernel: Can: - :in  Can_FifoInit
Sep 17 13:13:31 uschi kernel: Can: - :out
Sep 17 13:13:31 uschi kernel: Can: - :in  CAN_ChipReset
Sep 17 13:13:31 uschi kernel: Can: - : INT 0x0
Sep 17 13:13:31 uschi kernel:
Sep 17 13:13:31 uschi kernel: Can: - :status=0x3c mode=0x1
Sep 17 13:13:31 uschi kernel: Can: - :[0] CAN_mode 0x1
Sep 17 13:13:31 uschi kernel:
Sep 17 13:13:31 uschi kernel: Can: - :[0] CAN_mode 0x9
Sep 17 13:13:31 uschi kernel:
Sep 17 13:13:31 uschi kernel: Can: - :[0] CAN_mode 0x9
Sep 17 13:13:31 uschi kernel:
Sep 17 13:13:31 uschi kernel: Can: - :in  CAN_SetTiming
Sep 17 13:13:31 uschi kernel: Can: - :baud[0]=125
Sep 17 13:13:31 uschi kernel: Can: - :tim0=0x3 tim1=0x1c
Sep 17 13:13:31 uschi kernel: Can: - :out
```

```
Sep 17 13:13:31 uschi kernel: Can: - :[0] CAN_mode 0x9
Sep 17 13:13:31 uschi kernel:
Sep 17 13:13:31 uschi kernel: Can: - :in  CAN_SetMask
Sep 17 13:13:31 uschi kernel: Can: - :[0] acc=0xffffffff mask=0xffffffff
Sep 17 13:13:31 uschi kernel: Can: - :out
Sep 17 13:13:31 uschi kernel: Can: - :[0] CAN_mode 0x9
Sep 17 13:13:31 uschi kernel:
Sep 17 13:13:31 uschi kernel: Can: - :out
Sep 17 13:13:31 uschi kernel: Can: - :in  CAN_StartChip
Sep 17 13:13:31 uschi kernel: Can: - :[0] CAN_mode 0x9
Sep 17 13:13:31 uschi kernel:
Sep 17 13:13:31 uschi kernel: Can: - :start mode=0x8
Sep 17 13:13:31 uschi kernel: Can: - :out
Sep 17 13:13:31 uschi kernel:  MODE 0x8, STAT 0x3c, IRQE 0xf,
Sep 17 13:13:31 uschi kernel: Can: - :out
```

Other messages denote a corrupt or wrong *.conf configuration or using a not supported hardware.

Please start `can_send` to send a CAN message with 8 data byte with the contents of 0x55:

```
$ can_send -D can0 0x555 0x55 0x55 0x55 0x55 0x55 0x55 0x55 0x55
```

After execution of `can_send` there should be a CAN message on the bus. When the bus is not connected with a receiving device the CAN controller will continue to send the message because it doesn't receive an acknowledge. This behaviour is easy to trace with an oscilloscope. The chosen CAN identifier and the pattern of the data is quite easy to recognise and to measure. With a baud rate of 125 kBit/s the measurement of the shortest signal change should be 8 µs.

Reset the debug level of the driver:

```
./debug 0
```

3. Entries in /proc/sys/dev/Can

Older versions used to have these entries at */proc/sys/Can*. Please see also the example in the configuration files *etc/*.conf*.

Entry	per channel	CAN FD	meaning
AccCode	*		CAN Controller Acceptance Code Register
AccMask	*		CAN Controller Acceptance Maske
Base	*		CAN Controller Address
CAN clock			Clock frequency used for the CAN controller
Chipset			CAN controller supported by the driver
dbgMask			global debug level
framelength		*	8 for classic CAn, or 64 for CAN FD extended frame length
IOModel			One letter per channel for the IO-Model
IRQ	*		IRQ Number
Outc	*		Output Control Register
Overrun	*		Overrun Flag of the channel
RxErr	*		number of Rx errors
RxErrCounter	*		CAN controllers RX error counter
Speedfactor	*	*	Speedfactor for data phase bit rate
Transmitterdelay	*	*	Transmitter delay compensation TDC
Timeout	*		Timeout value
TxErr	*		Number of TX errors
TxErrCounter	*		CAN controllers TX error counter
version			versions string

The values for the bit timing registers of the CAN controller for the bit-rates 10,20,40,50,125,250,500,800 and 1000 kBit/s are taken from internal tables. These tables are only valid if the clock cycle of the CAN controller is 8 MHz (external 16 MHz quartz). When using a different clock cycle the bit timing registers

are calculated as follows:

```
BTR0 = (Baud >> 8) && 0xFF
BTR1 =  Baud       && 0xFF
```

Example for setting the bit rate to 125 kBit/s for a SJA1000 with a clock cycle of 10 MHz (20MHz), 16 time quanta and the sampling point at 87.5%

```
Calculated values for the bit timing register:
                                                BTR0 = 0x04
                                                BTR1 = 0x1c
From this it follows that:
                  Baud = (0x04 << 8) + 0x1c = 0x041c = 1052
```

The site http://www.port.de/deutsch/canprod/sv_req_form.html provides a input form for bit timing calculation of the bit timing register.

dbgMask
: global debug level
default 0 - no debug messages
every bit of this mask has a specified meaning.

Bit	meaning
0	Flag for setting all options=on
1	log function entries
2	log function exits
3	log branches
4	log data given to functions
5	log interrupts
6	log register info
7	reserved

Outc The output control register of the Philips 82C200/SJA1000

4. DIL-NET-PC TRM/816 Hardware by SSV embedded Systems

The code was provided by Sven Geggus ⟨geggus@iitb.fraunhofer.de⟩. Please read his *README.trm816* and the *trm816/README*.

5. PC104 board PC104-200 from ESD

modified**can4linux**was and extended to support the esd ⟨http://www.esd-electronics.com/⟩ elctronic system design GmbH PC/104-CAN Card by Jean-Jacques Tchouto ⟨tchouto@fokus.fraunhofer.de⟩.

6. Embedded controllers with internal CAN

These need cross compilation. Since 3.5.3 the Makefile concept has changed.

There is a global Makefile, called Makefile-standard and some specific target Makefiles called Makefile-target. A shell script can be used to switch between.
Calling ./target without arguments will tell you more. Settings for the cross compilation and location of kernel source files are handled via shell environment variables defined in files like cross-target. After setting your specific values, simply source these files into your shell environment and call make.

6.1. ATMEL AT91SAM9263

6.2. ATMEL AT919260 with external SJA1000

6.3. ATMEL AT91 wit MCP2515 via SPI

6.4. Freescale MCF5282

6.5. Xilinx XCAN (on Zynq)

6.6. BananaPi with Allwinner A20

6.7. RaspberryPi with external MCP2515 on SPI

7. In the case of ...

In the case of any malfunction of the driver, e.g. **open**() returns EINVAL - invalid argument -, set the debugmask **dbgMask** to a higher level, and watch the system log at */var/log/messages*.

8. Configuring the driver

Some configuration parameters of the driver, like bit rate, are accesible via the /proc/sys/Can/* entries. Since version 3.5.3 the location is /proc/sys/dev/Can/*.

To view the entries one can do

```
$ cat /proc/sys/dev/Can/Baud
125     125     125     250
```

or to see all values

```
/sbin/sysctl dev.Can
```

and set values

```
$ echo '250 250 500 125' > /proc/sys/dev/Can/Baud
$ cat /proc/sys/dev/Can/Baud
250     250     500     125
```

Linux has special commands to handle these

```
$ /sbin/sysctl dev.Can.version
dev.Can.version = 3.4_CPC_PCI
```

And values can be set using

```
$ sudo /sbin/sysctl -w dev.Can.dbgMask=7
dev.Can.dbgMask = 7
```

The changes you make do not apply across a reboot. In order for them to persist, you need to either:

- Create a new startup script in an appropriate /etc/rc#.d directory which runs the cat or sysctl commands, or
- Put "variable=value" lines into /etc/sysctl.conf. These lines are just like sysctl commands, without the 'sysctl' keyword or '-w' flag. For example:
  `dev.Can.Baud = '125 125 125'`