==========

can.txt

Readme file for the Controller Area Network Protocol Family (aka Socket CAN)

This file contains

==========

1. Overview / What is Socket CAN

The socketcan package is an implementation of CAN protocols
(Controller Area Network) for Linux. CAN is a networking technology
which has widespread use in automation, embedded devices, and

automotive fields. While there have been other CAN implementations for Linux based on character devices, Socket CAN uses the Berkeley socket API, the Linux network stack and implements the CAN device drivers as network interfaces. The CAN socket API has been designed as similar as possible to the TCP/IP protocols to allow programmers, familiar with network programming, to easily learn how to use CAN sockets.

2. Motivation / Why using the socket API

There have been CAN implementations for Linux before Socket CAN so the question arises, why we have started another project. Most existing implementations come as a device driver for some CAN hardware, they are based on character devices and provide comparatively little functionality. Usually, there is only a hardware-specific device driver which provides a character device interface to send and receive raw CAN frames, directly to/from the controller hardware. Queueing of frames and higher-level transport protocols like ISO-TP have to be implemented in user space applications. Also, most character-device implementations support only one single process to open the device at a time, similar to a serial interface. Exchanging the CAN controller requires employment of another device driver and often the need for adaption of large parts of the application to the new driver's API.

Socket CAN was designed to overcome all of these limitations. A new protocol family has been implemented which provides a socket interface to user space applications and which builds upon the Linux network layer, so to use all of the provided queueing functionality. A device driver for CAN controller hardware registers itself with the Linux network layer as a network device, so that CAN frames from the controller can be passed up to the network layer and on to the CAN protocol family module and also vice-versa. Also, the protocol family module provides an API for transport protocol modules to register, so that any number of transport protocols can be loaded or unloaded dynamically. In fact, the can core module alone does not provide any protocol and cannot be used without loading at least one additional protocol module. Multiple sockets can be opened at the same time, on different or the same protocol module and they can listen/send frames on different or the same CAN IDs. Several sockets listening on the same interface for frames with the same CAN ID are all passed the same received matching CAN frames. An application wishing to communicate using a specific transport protocol, e.g. ISO-TP, just selects that protocol when opening the socket, and then can read and write application data byte streams, without having to deal with CAN-IDs, frames, etc.

Similar functionality visible from user-space could be provided by a character device, too, but this would lead to a technically inelegant solution for a couple of reasons:

* Intricate usage. Instead of passing a protocol argument to socket(2) and using bind(2) to select a CAN interface and CAN ID, an application would have to do all these operations using ioctl(2)s.

* Code duplication. A character device cannot make use of the Linux
  network queueing code, so all that code would have to be duplicated
  for CAN networking.

* Abstraction. In most existing character-device implementations, the
  hardware-specific device driver for a CAN controller directly
  provides the character device for the application to work with.
  This is at least very unusual in Unix systems for both, char and
  block devices. For example you don't have a character device for a
  certain UART of a serial interface, a certain sound chip in your
  computer, a SCSI or IDE controller providing access to your hard
  disk or tape streamer device. Instead, you have abstraction layers
  which provide a unified character or block device interface to the
  application on the one hand, and a interface for hardware-specific
  device drivers on the other hand. These abstractions are provided
  by subsystems like the tty layer, the audio subsystem or the SCSI
  and IDE subsystems for the devices mentioned above.

  The easiest way to implement a CAN device driver is as a character
  device without such a (complete) abstraction layer, as is done by most
  existing drivers. The right way, however, would be to add such a
  layer with all the functionality like registering for certain CAN
  IDs, supporting several open file descriptors and (de)multiplexing
  CAN frames between them, (sophisticated) queueing of CAN frames, and
  providing an API for device drivers to register with. However, then
  it would be no more difficult, or may be even easier, to use the
  networking framework provided by the Linux kernel, and this is what
  Socket CAN does.

  The use of the networking framework of the Linux kernel is just the
  natural and most appropriate way to implement CAN for Linux.

3. Socket CAN concept

  As described in chapter 2 it is the main goal of Socket CAN to
  provide a socket interface to user space applications which builds
  upon the Linux network layer. In contrast to the commonly known
  TCP/IP and ethernet networking, the CAN bus is a broadcast-only(!)
  medium that has no MAC-layer addressing like ethernet. The CAN-identifier
  (can_id) is used for arbitration on the CAN-bus. Therefore the CAN-IDs
  have to be chosen uniquely on the bus. When designing a CAN-ECU
  network the CAN-IDs are mapped to be sent by a specific ECU.
  For this reason a CAN-ID can be treated best as a kind of source address.

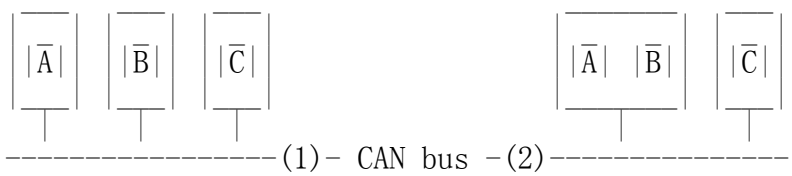
3.1 receive lists

  The network transparent access of multiple applications leads to the
  problem that different applications may be interested in the same
  CAN-IDs from the same CAN network interface. The Socket CAN core
  module - which implements the protocol family CAN - provides several
  high efficient receive lists for this reason. If e.g. a user space
  application opens a CAN RAW socket, the raw protocol module itself
  requests the (range of) CAN-IDs from the Socket CAN core that are
  requested by the user. The subscription and unsubscription of
  CAN-IDs can be done for specific CAN interfaces or for all(!) known

CAN interfaces with the can_rx_(un)register() functions provided to
CAN protocol modules by the SocketCAN core (see chapter 5).
To optimize the CPU usage at runtime the receive lists are split up
into several specific lists per device that match the requested
filter complexity for a given use-case.

3.2 local loopback of sent frames

As known from other networking concepts the data exchanging
applications may run on the same or different nodes without any
change (except for the according addressing information):



To ensure that application A receives the same information in the
example (2) as it would receive in example (1) there is need for
some kind of local loopback of the sent CAN frames on the appropriate
node.

The Linux network devices (by default) just can handle the
transmission and reception of media dependent frames. Due to the
arbitration on the CAN bus the transmission of a low prio CAN-ID
may be delayed by the reception of a high prio CAN frame. To
reflect the correct* traffic on the node the loopback of the sent
data has to be performed right after a successful transmission. If
the CAN network interface is not capable of performing the loopback for
some reason the SocketCAN core can do this task as a fallback solution.
See chapter 6.2 for details (recommended).

The loopback functionality is enabled by default to reflect standard
networking behaviour for CAN applications. Due to some requests from
the RT-SocketCAN group the loopback optionally may be disabled for each
separate socket. See sockopts from the CAN RAW sockets in chapter 4.1.

* = you really like to have this when you're running analyser tools
    like 'candump' or 'cansniffer' on the (same) node.

3.3 network security issues (capabilities)

The Controller Area Network is a local field bus transmitting only
broadcast messages without any routing and security concepts.
In the majority of cases the user application has to deal with
raw CAN frames. Therefore it might be reasonable NOT to restrict
the CAN access only to the user root, as known from other networks.
Since the currently implemented CAN_RAW and CAN_BCM sockets can only
send and receive frames to/from CAN interfaces it does not affect
security of others networks to allow all users to access the CAN.
To enable non-root users to access CAN_RAW and CAN_BCM protocol
sockets the Kconfig options CAN_RAW_USER and/or CAN_BCM_USER may be
selected at kernel compile time.

3.4 network problem notifications

The use of the CAN bus may lead to several problems on the physical and media access control layer. Detecting and logging of these lower layer problems is a vital requirement for CAN users to identify hardware issues on the physical transceiver layer as well as arbitration problems and error frames caused by the different ECUs. The occurrence of detected errors are important for diagnosis and have to be logged together with the exact timestamp. For this reason the CAN interface driver can generate so called Error Frames that can optionally be passed to the user application in the same way as other CAN frames. Whenever an error on the physical layer or the MAC layer is detected (e.g. by the CAN controller) the driver creates an appropriate error frame. Error frames can be requested by the user application using the common CAN filter mechanisms. Inside this filter definition the (interested) type of errors may be selected. The reception of error frames is disabled by default. The format of the CAN error frame is briefly decribed in the Linux header file "include/linux/can/error.h".

4. How to use Socket CAN

Like TCP/IP, you first need to open a socket for communicating over a CAN network. Since Socket CAN implements a new protocol family, you need to pass PF_CAN as the first argument to the socket(2) system call. Currently, there are two CAN protocols to choose from, the raw socket protocol and the broadcast manager (BCM). So to open a socket, you would write

```
s = socket(PF_CAN, SOCK_RAW, CAN_RAW);
```

and

```
s = socket(PF_CAN, SOCK_DGRAM, CAN_BCM);
```

respectively. After the successful creation of the socket, you would normally use the bind(2) system call to bind the socket to a CAN interface (which is different from TCP/IP due to different addressing - see chapter 3). After binding (CAN_RAW) or connecting (CAN_BCM) the socket, you can read(2) and write(2) from/to the socket or use send(2), sendto(2), sendmsg(2) and the recv* counterpart operations on the socket as usual. There are also CAN specific socket options described below.

The basic CAN frame structure and the sockaddr structure are defined in include/linux/can.h:

```
struct can_frame {
        canid_t can_id;  /* 32 bit CAN_ID + EFF/RTR/ERR flags */
        __u8    can_dlc; /* data length code: 0 .. 8 */
        __u8    data[8] __attribute__((aligned(8)));
};
```

The alignment of the (linear) payload data[] to a 64bit boundary allows the user to define own structs and unions to easily access the

CAN payload. There is no given byteorder on the CAN bus by default. A read(2) system call on a CAN_RAW socket transfers a struct can_frame to the user space.

The sockaddr_can structure has an interface index like the PF_PACKET socket, that also binds to a specific interface:

```
struct sockaddr_can {
        sa_family_t can_family;
        int         can_ifindex;
        union {
                /* transport protocol class address info (e.g. ISOTP) */
                struct { canid_t rx_id, tx_id; } tp;

                /* reserved for future CAN protocols address information */
        } can_addr;
};
```

To determine the interface index an appropriate ioctl() has to be used (example for CAN_RAW sockets without error checking):

```
int s;
struct sockaddr_can addr;
struct ifreq ifr;

s = socket(PF_CAN, SOCK_RAW, CAN_RAW);

strcpy(ifr.ifr_name, "can0" );
ioctl(s, SIOCGIFINDEX, &ifr);

addr.can_family = AF_CAN;
addr.can_ifindex = ifr.ifr_ifindex;

bind(s, (struct sockaddr *)&addr, sizeof(addr));

(..)
```

To bind a socket to all(!) CAN interfaces the interface index must be 0 (zero). In this case the socket receives CAN frames from every enabled CAN interface. To determine the originating CAN interface the system call recvfrom(2) may be used instead of read(2). To send on a socket that is bound to 'any' interface sendto(2) is needed to specify the outgoing interface.

Reading CAN frames from a bound CAN_RAW socket (see above) consists of reading a struct can_frame:

```
struct can_frame frame;

nbytes = read(s, &frame, sizeof(struct can_frame));

if (nbytes < 0) {
        perror("can raw socket read");
        return 1;
}
```

```
  /* paranoid check ... */
  if (nbytes < sizeof(struct can_frame)) {
          fprintf(stderr, "read: incomplete CAN frame\n");
          return 1;
  }

  /* do something with the received CAN frame */
```

Writing CAN frames can be done similarly, with the write(2) system call:

```
  nbytes = write(s, &frame, sizeof(struct can_frame));
```

When the CAN interface is bound to 'any' existing CAN interface (addr.can_ifindex = 0) it is recommended to use recvfrom(2) if the information about the originating CAN interface is needed:

```
  struct sockaddr_can addr;
  struct ifreq ifr;
  socklen_t len = sizeof(addr);
  struct can_frame frame;

  nbytes = recvfrom(s, &frame, sizeof(struct can_frame),
                    0, (struct sockaddr*)&addr, &len);

  /* get interface name of the received CAN frame */
  ifr.ifr_ifindex = addr.can_ifindex;
  ioctl(s, SIOCGIFNAME, &ifr);
  printf("Received a CAN frame from interface %s", ifr.ifr_name);
```

To write CAN frames on sockets bound to 'any' CAN interface the outgoing interface has to be defined certainly.

```
  strcpy(ifr.ifr_name, "can0");
  ioctl(s, SIOCGIFINDEX, &ifr);
  addr.can_ifindex = ifr.ifr_ifindex;
  addr.can_family  = AF_CAN;

  nbytes = sendto(s, &frame, sizeof(struct can_frame),
                  0, (struct sockaddr*)&addr, sizeof(addr));
```

4.1 RAW protocol sockets with can_filters (SOCK_RAW)

Using CAN_RAW sockets is extensively comparable to the commonly known access to CAN character devices. To meet the new possibilities provided by the multi user SocketCAN approach, some reasonable defaults are set at RAW socket binding time:

- The filters are set to exactly one filter receiving everything
- The socket only receives valid data frames (=> no error frames)
- The loopback of sent CAN frames is enabled (see chapter 3.2)
- The socket does not receive its own sent frames (in loopback mode)

These default settings may be changed before or after binding the socket. To use the referenced definitions of the socket options for CAN_RAW sockets, include <linux/can/raw.h>.

4.1.1 RAW socket option CAN_RAW_FILTER

The reception of CAN frames using CAN_RAW sockets can be controlled by defining 0 .. n filters with the CAN_RAW_FILTER socket option.

The CAN filter structure is defined in include/linux/can.h:

```
struct can_filter {
        canid_t can_id;
        canid_t can_mask;
};
```

A filter matches, when

```
<received_can_id> & mask == can_id & mask
```

which is analogous to known CAN controllers hardware filter semantics. The filter can be inverted in this semantic, when the CAN_INV_FILTER bit is set in can_id element of the can_filter structure. In contrast to CAN controller hardware filters the user may set 0 .. n receive filters for each open socket separately:

```
struct can_filter rfilter[2];

rfilter[0].can_id   = 0x123;
rfilter[0].can_mask = CAN_SFF_MASK;
rfilter[1].can_id   = 0x200;
rfilter[1].can_mask = 0x700;

setsockopt(s, SOL_CAN_RAW, CAN_RAW_FILTER, &rfilter, sizeof(rfilter));
```

To disable the reception of CAN frames on the selected CAN_RAW socket:

```
setsockopt(s, SOL_CAN_RAW, CAN_RAW_FILTER, NULL, 0);
```

To set the filters to zero filters is quite obsolete as not read data causes the raw socket to discard the received CAN frames. But having this 'send only' use-case we may remove the receive list in the Kernel to save a little (really a very little!) CPU usage.

4.1.2 RAW socket option CAN_RAW_ERR_FILTER

As described in chapter 3.4 the CAN interface driver can generate so called Error Frames that can optionally be passed to the user application in the same way as other CAN frames. The possible errors are divided into different error classes that may be filtered using the appropriate error mask. To register for every possible error condition CAN_ERR_MASK can be used as value for the error mask. The values for the error mask are defined in linux/can/error.h .

```
can_err_mask_t err_mask = ( CAN_ERR_TX_TIMEOUT | CAN_ERR_BUSOFF );

setsockopt(s, SOL_CAN_RAW, CAN_RAW_ERR_FILTER,
           &err_mask, sizeof(err_mask));
```

4.1.3 RAW socket option CAN_RAW_LOOPBACK

To meet multi user needs the local loopback is enabled by default
(see chapter 3.2 for details). But in some embedded use-cases
(e.g. when only one application uses the CAN bus) this loopback
functionality can be disabled (separately for each socket):

```
int loopback = 0; /* 0 = disabled, 1 = enabled (default) */

setsockopt(s, SOL_CAN_RAW, CAN_RAW_LOOPBACK, &loopback, sizeof(loopback));
```

4.1.4 RAW socket option CAN_RAW_RECV_OWN_MSGS

When the local loopback is enabled, all the sent CAN frames are
looped back to the open CAN sockets that registered for the CAN
frames' CAN-ID on this given interface to meet the multi user
needs. The reception of the CAN frames on the same socket that was
sending the CAN frame is assumed to be unwanted and therefore
disabled by default. This default behaviour may be changed on
demand:

```
int recv_own_msgs = 1; /* 0 = disabled (default), 1 = enabled */

setsockopt(s, SOL_CAN_RAW, CAN_RAW_RECV_OWN_MSGS,
           &recv_own_msgs, sizeof(recv_own_msgs));
```

4.2 Broadcast Manager protocol sockets (SOCK_DGRAM)
4.3 connected transport protocols (SOCK_SEQPACKET)
4.4 unconnected transport protocols (SOCK_DGRAM)

5. Socket CAN core module

The Socket CAN core module implements the protocol family
PF_CAN. CAN protocol modules are loaded by the core module at
runtime. The core module provides an interface for CAN protocol
modules to subscribe needed CAN IDs (see chapter 3.1).

5.1 can.ko module params

- stats_timer: To calculate the Socket CAN core statistics
  (e.g. current/maximum frames per second) this 1 second timer is
  invoked at can.ko module start time by default. This timer can be
  disabled by using stattimer=0 on the module commandline.

- debug: (removed since SocketCAN SVN r546)

5.2 procfs content

As described in chapter 3.1 the Socket CAN core uses several filter
lists to deliver received CAN frames to CAN protocol modules. These
receive lists, their filters and the count of filter matches can be
checked in the appropriate receive list. All entries contain the
device and a protocol module identifier:

```
foo@bar:~$ cat /proc/net/can/rcvlist_all
```

```
  receive list 'rx_all':
    (vcan3: no entry)
    (vcan2: no entry)
    (vcan1: no entry)
    device   can_id   can_mask  function  userdata   matches  ident
     vcan0     000    00000000  f88e6370  f6c6f400         0  raw
    (any: no entry)
```

In this example an application requests any CAN traffic from vcan0.

```
  rcvlist_all - list for unfiltered entries (no filter operations)
  rcvlist_eff - list for single extended frame (EFF) entries
  rcvlist_err - list for error frames masks
  rcvlist_fil - list for mask/value filters
  rcvlist_inv - list for mask/value filters (inverse semantic)
  rcvlist_sff - list for single standard frame (SFF) entries
```

Additional procfs files in /proc/net/can

```
  stats       - Socket CAN core statistics (rx/tx frames, match ratios, ...)
  reset_stats - manual statistic reset
  version     - prints the Socket CAN core version and the ABI version
```

5.3 writing own CAN protocol modules

To implement a new protocol in the protocol family PF_CAN a new
protocol has to be defined in include/linux/can.h .
The prototypes and definitions to use the Socket CAN core can be
accessed by including include/linux/can/core.h .
In addition to functions that register the CAN protocol and the
CAN device notifier chain there are functions to subscribe CAN
frames received by CAN interfaces and to send CAN frames:

```
  can_rx_register   - subscribe CAN frames from a specific interface
  can_rx_unregister - unsubscribe CAN frames from a specific interface
  can_send          - transmit a CAN frame (optional with local loopback)
```

For details see the kerneldoc documentation in net/can/af_can.c or
the source code of net/can/raw.c or net/can/bcm.c .

6. CAN network drivers

Writing a CAN network device driver is much easier than writing a
CAN character device driver. Similar to other known network device
drivers you mainly have to deal with:

- TX: Put the CAN frame from the socket buffer to the CAN controller.
- RX: Put the CAN frame from the CAN controller to the socket buffer.

See e.g. at Documentation/networking/netdevices.txt . The differences
for writing CAN network device driver are described below:

6.1 general settings

```
  dev->type  = ARPHRD_CAN; /* the netdevice hardware type */
  dev->flags = IFF_NOARP;  /* CAN has no arp */

  dev->mtu   = sizeof(struct can_frame);
```

The struct can_frame is the payload of each socket buffer in the protocol family PF_CAN.

6.2 local loopback of sent frames

As described in chapter 3.2 the CAN network device driver should support a local loopback functionality similar to the local echo e.g. of tty devices. In this case the driver flag IFF_ECHO has to be set to prevent the PF_CAN core from locally echoing sent frames (aka loopback) as fallback solution:

```
  dev->flags = (IFF_NOARP | IFF_ECHO);
```

6.3 CAN controller hardware filters

To reduce the interrupt load on deep embedded systems some CAN controllers support the filtering of CAN IDs or ranges of CAN IDs. These hardware filter capabilities vary from controller to controller and have to be identified as not feasible in a multi-user networking approach. The use of the very controller specific hardware filters could make sense in a very dedicated use-case, as a filter on driver level would affect all users in the multi-user system. The high efficient filter sets inside the PF_CAN core allow to set different multiple filters for each socket separately. Therefore the use of hardware filters goes to the category 'handmade tuning on deep embedded systems'. The author is running a MPC603e @133MHz with four SJA1000 CAN controllers from 2002 under heavy bus load without any problems ...

6.4 The virtual CAN driver (vcan)

Similar to the network loopback devices, vcan offers a virtual local CAN interface. A full qualified address on CAN consists of

- a unique CAN Identifier (CAN ID)
- the CAN bus this CAN ID is transmitted on (e.g. can0)

so in common use cases more than one virtual CAN interface is needed.

The virtual CAN interfaces allow the transmission and reception of CAN frames without real CAN controller hardware. Virtual CAN network devices are usually named 'vcanX', like vcan0 vcan1 vcan2 ...
When compiled as a module the virtual CAN driver module is called vcan.ko

Since Linux Kernel version 2.6.24 the vcan driver supports the Kernel netlink interface to create vcan network devices. The creation and removal of vcan network devices can be managed with the ip(8) tool:

- Create a virtual CAN network interface:
```
    $ ip link add type vcan
```

- Create a virtual CAN network interface with a specific name 'vcan42':
    $ ip link add dev vcan42 type vcan

- Remove a (virtual CAN) network interface 'vcan42':
    $ ip link del vcan42

6.5 The CAN network device driver interface

The CAN network device driver interface provides a generic interface to setup, configure and monitor CAN network devices. The user can then configure the CAN device, like setting the bit-timing parameters, via the netlink interface using the program "ip" from the "IPROUTE2" utility suite. The following chapter describes briefly how to use it. Furthermore, the interface uses a common data structure and exports a set of common functions, which all real CAN network device drivers should use. Please have a look to the SJA1000 or MSCAN driver to understand how to use them. The name of the module is can-dev.ko.

6.5.1 Netlink interface to set/get devices properties

The CAN device must be configured via netlink interface. The supported netlink message types are defined and briefly described in "include/linux/can/netlink.h". CAN link support for the program "ip" of the IPROUTE2 utility suite is avaiable and it can be used as shown below:

- Setting CAN device properties:

```
  $ 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 } ]

      [ 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 }
```

- Display CAN device details and statistics:

```
  $ ip -details -statistics link show can0
  2: can0: <NOARP,UP,LOWER_UP,ECHO> mtu 16 qdisc pfifo_fast state UP qlen 10
    link/can
    can <TRIPLE-SAMPLING> state ERROR-ACTIVE restart-ms 100
```

```
  bitrate 125000 sample_point 0.875
  tq 125 prop-seg 6 phase-seg1 7 phase-seg2 2 sjw 1
  sja1000: tseg1 1..16 tseg2 1..8 sjw 1..4 brp 1..64 brp-inc 1
  clock 8000000
  re-started bus-errors arbit-lost error-warn error-pass bus-off
  41         17457      0          41         42         41
  RX: bytes  packets  errors  dropped overrun mcast
  140859     17608    17457   0       0       0
  TX: bytes  packets  errors  dropped carrier collsns
  861        112      0       41      0       0
```

More info to the above output:

"<TRIPLE-SAMPLING>"
    Shows the list of selected CAN controller modes: LOOPBACK, LISTEN-ONLY, or TRIPLE-SAMPLING.

"state ERROR-ACTIVE"
    The current state of the CAN controller: "ERROR-ACTIVE", "ERROR-WARNING", "ERROR-PASSIVE", "BUS-OFF" or "STOPPED"

"restart-ms 100"
    Automatic restart delay time. If set to a non-zero value, a restart of the CAN controller will be triggered automatically in case of a bus-off condition after the specified delay time in milliseconds. By default it's off.

"bitrate 125000 sample_point 0.875"
    Shows the real bit-rate in bits/sec and the sample-point in the range 0.000..0.999. If the calculation of bit-timing parameters is enabled in the kernel (CONFIG_CAN_CALC_BITTIMING=y), the bit-timing can be defined by setting the "bitrate" argument. Optionally the "sample-point" can be specified. By default it's 0.000 assuming CIA-recommended sample-points.

"tq 125 prop-seg 6 phase-seg1 7 phase-seg2 2 sjw 1"
    Shows the time quanta in ns, propagation segment, phase buffer segment 1 and 2 and the synchronisation jump width in units of tq. They allow to define the CAN bit-timing in a hardware independent format as proposed by the Bosch CAN 2.0 spec (see chapter 8 of http://www.semiconductors.bosch.de/pdf/can2spec.pdf).

"sja1000: tseg1 1..16 tseg2 1..8 sjw 1..4 brp 1..64 brp-inc 1
 clock 8000000"
    Shows the bit-timing constants of the CAN controller, here the "sja1000". The minimum and maximum values of the time segment 1 and 2, the synchronisation jump width in units of tq, the bitrate pre-scaler and the CAN system clock frequency in Hz. These constants could be used for user-defined (non-standard) bit-timing calculation algorithms in user-space.

"re-started bus-errors arbit-lost error-warn error-pass bus-off"
    Shows the number of restarts, bus and arbitration lost errors, and the state changes to the error-warning, error-passive and bus-off state. RX overrun errors are listed in the "overrun" field of the standard network statistics.

6.5.2 Setting the CAN bit-timing

The CAN bit-timing parameters can always be defined in a hardware independent format as proposed in the Bosch CAN 2.0 specification specifying the arguments "tq", "prop_seg", "phase_seg1", "phase_seg2" and "sjw":

```
  $ ip link set canX type can tq 125 prop-seg 6 \
                              phase-seg1 7 phase-seg2 2 sjw 1
```

If the kernel option CONFIG_CAN_CALC_BITTIMING is enabled, CIA recommended CAN bit-timing parameters will be calculated if the bit-rate is specified with the argument "bitrate":

```
  $ ip link set canX type can bitrate 125000
```

Note that this works fine for the most common CAN controllers with standard bit-rates but may *fail* for exotic bit-rates or CAN system clock frequencies. Disabling CONFIG_CAN_CALC_BITTIMING saves some space and allows user-space tools to solely determine and set the bit-timing parameters. The CAN controller specific bit-timing constants can be used for that purpose. They are listed by the following command:

```
  $ ip -details link show can0
  ...
    sja1000: clock 8000000 tseg1 1..16 tseg2 1..8 sjw 1..4 brp 1..64 brp-inc 1
```

6.5.3 Starting and stopping the CAN network device

A CAN network device is started or stopped as usual with the command "ifconfig canX up/down" or "ip link set canX up/down". Be aware that you *must* define proper bit-timing parameters for real CAN devices before you can start it to avoid error-prone default settings:

```
  $ ip link set canX up type can bitrate 125000
```

A device may enter the "bus-off" state if too much errors occurred on the CAN bus. Then no more messages are received or sent. An automatic bus-off recovery can be enabled by setting the "restart-ms" to a non-zero value, e.g.:

```
  $ ip link set canX type can restart-ms 100
```

Alternatively, the application may realize the "bus-off" condition by monitoring CAN error frames and do a restart when appropriate with the command:

```
  $ ip link set canX type can restart
```

Note that a restart will also create a CAN error frame (see also chapter 3.4).

6.6 Supported CAN hardware

Please check the "Kconfig" file in "drivers/net/can" to get an actual
list of the support CAN hardware. On the Socket CAN project website
(see chapter 7) there might be further drivers available, also for
older kernel versions.

7. Socket CAN resources

You can find further resources for Socket CAN like user space tools,
support for old kernel versions, more drivers, mailing lists, etc.
at the BerliOS OSS project website for Socket CAN:

http://developer.berlios.de/projects/socketcan

If you have questions, bug fixes, etc., don't hesitate to post them to
the Socketcan-Users mailing list. But please search the archives first.

8. Credits

Oliver Hartkopp (PF_CAN core, filters, drivers, bcm, SJA1000 driver)
Urs Thuermann (PF_CAN core, kernel integration, socket interfaces, raw, vcan)
Jan Kizka (RT-SocketCAN core, Socket-API reconciliation)
Wolfgang Grandegger (RT-SocketCAN core & drivers, Raw Socket-API reviews,
CAN device driver interface, MSCAN driver)
Robert Schwebel (design reviews, PTXdist integration)
Marc Kleine-Budde (design reviews, Kernel 2.6 cleanups, drivers)
Benedikt Spranger (reviews)
Thomas Gleixner (LKML reviews, coding style, posting hints)
Andrey Volkov (kernel subtree structure, ioctls, MSCAN driver)
Matthias Brukner (first SJA1000 CAN netdevice implementation Q2/2003)
Klaus Hitschler (PEAK driver integration)
Uwe Koppe (CAN netdevices with PF_PACKET approach)
Michael Schulze (driver layer loopback requirement, RT CAN drivers review)
Pavel Pisa (Bit-timing calculation)
Sascha Hauer (SJA1000 platform driver)
Sebastian Haas (SJA1000 EMS PCI driver)
Markus Plessing (SJA1000 EMS PCI driver)
Per Dalen (SJA1000 Kvaser PCI driver)
Sam Ravnborg (reviews, coding style, kbuild help)