Low Level CAN Framework

Application Programmers Interface

Version 1.0.0

2006-02-20

Inhaltsverzeichnis

1 Einleitung

Im Rahmen verschiedener Projekte wurde in der Konzernforschung der Volkswagen AG ein so genanntes *Low Level CAN Framework* (**LLCF**) entwickelt, das den einfachen Zugriff auf die Kommunikationsschichten des *Controller Area Network* (CAN) für Anwendungen erlaubt. Das *Low Level CAN Framework* sollte dabei möglichst modular konzipiert werden, um eine möglichst große Wiederverwendbarkeit in weiteren Projekten zu erreichen.

Wesentliche Komponenten des **LLCF** sind die Netzwerk(!)-Treiber für die verschiedenen CAN-Controller und die darüberliegenden Protokolle wie TP1.6, TP2.0, MCNet, ISO-TP, etc. Diese Komponenten sind im Linux-Kernel implementiert und werden über die standardisierte Socket-Schnittstelle angesprochen. Das Ziel dieses Konzeptes liegt darin, die Kommunikation über den CAN-Bus soweit wie möglich an die Benutzung gewöhnlicher TCP/IP-Sockets anzupassen. Dies gelingt jedoch nur zum Teil, da der CAN-Bus eine Reihe von Unterschieden zur Kommunikation mit TCP/IP und Ethernet aufweist:

- CAN kennt keine Geräte-Adressen wie die MAC-Adressen beim Ethernet. Das CAN-Frame enthält eine CAN-ID, die durch die übliche Zuordnung von zu sendenen CAN-IDs zu realen Endgeräten am ehesten einer Absender-Adresse entspricht. Weil alle Nachrichten Broadcast-Nachrichten sind, ist es auch nicht möglich, eine CAN-Nachricht nur an ein Gerät zu senden. Geräte am CAN-Bus können empfangene Nachrichten also nicht nach Zieladressen, sondern nur nach der CAN-ID 'Absenderadresse' filtern. CAN-Frames können daher nicht - wie beim Ethernet - explizit an ein bestimmtes Zielgerät gerichtet werden.
- Es gibt keinen Network Layer und damit auch keine Network-Layer-Adressen wie IP-Adressen. Folglich gibt es auch kein Routing (z.B. über verschiedene Netzwerk-Interfaces), wie es mit IP-Adressen möglich ist.

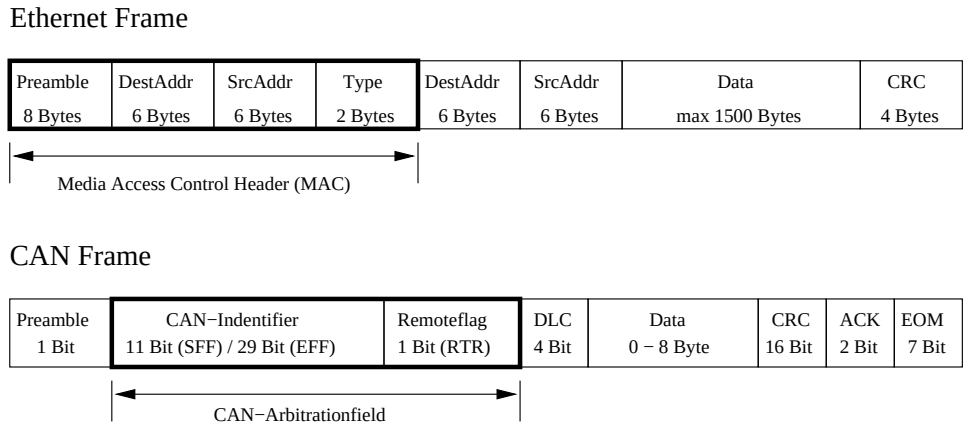

Abbildung 1: Unterschiedliche Adressierungen bei Ethernet / CAN

Diese Unterschiede führen dazu, dass die Struktur `struct sockaddr_can` sich nicht ganz analog zu der bekannten `struct sockaddr_in` für die TCP/IP-Protokollfamilie verhält. Der Ablauf eines Verbindungsaufbaus und die Benutzung geöffneter Sockets zum Datenaustausch sind jedoch stark an TCP/IP angelehnt.
Neben diesem Dokument sind daher auch die Manual Pages socket(2), bind(2), listen(2), accept(2), connect(2), read(2), write(2), recv(2), recvfrom(2), recvmsg(2), send(2), sendto(2), sendmsg(2), socket(7), packet(7) eines aktuellen Linux-Systems für das **LLCF** relevant. Außerdem bieten die Manual Pages ip(7), udp(7) und tcp(7) einen Einblick in Grundlagen, auf denen auch das **LLCF** basiert.

Das *Low Level CAN Framework* ist neben den bekannten Protokollen, wie z.B. den Protokollen der Internetprotokollfamilie PF_INET im Linux-Kernel integriert. Dazu wurde eine neue Protokollfamilie PF_CAN eingeführt. Durch die Realisierung der verschiedenen CAN-Protokolle als Kernelmodule, können zeitliche Randbedingungen im Kernel-Kontext eingehalten werden, die auf der Anwenderschicht in dieser Form nicht realisierbar wären. Für verschiedene Anwendungen (was zu mehreren Socket-Instanzen führt) kommt immer derselbe Code zur Ausführung.

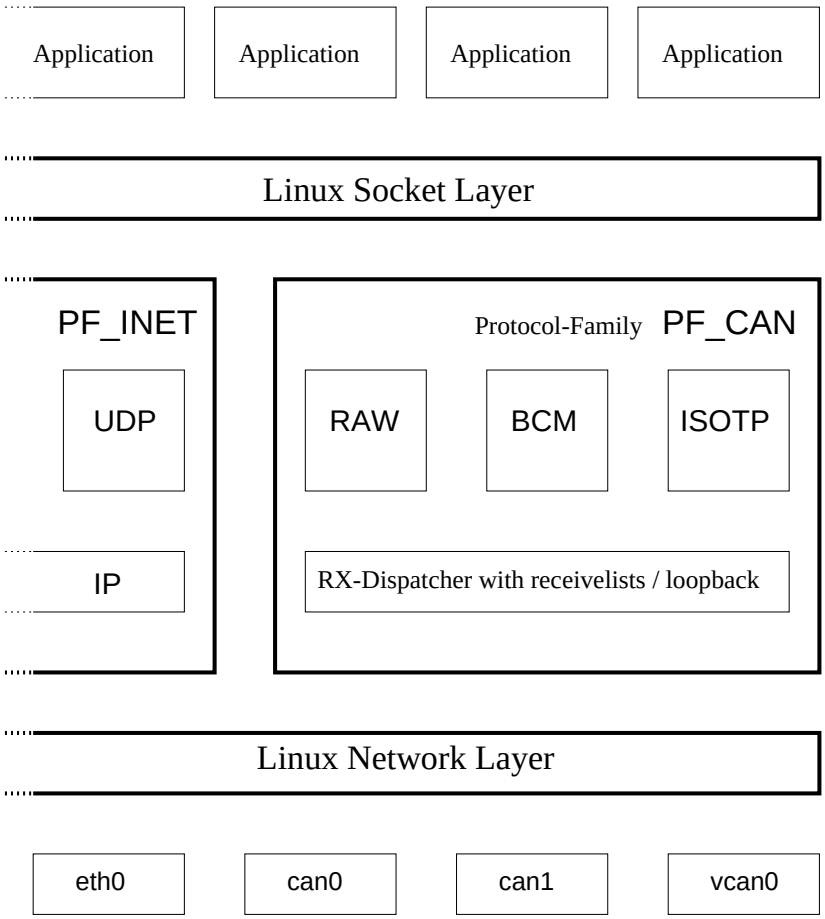

Abbildung 2: Das **LLCF** im Linux-Kernel

Das **LLCF** stellt für die verschiedenen Transportprotokolle und einen so genannten *Broadcast-Manager* (**BCM**) eine Reihe verschiedener Socket-Typen zur Verfügung. Außerdem ist ein RAW-Socket vorgesehen, der den direkten Zugriff auf den CAN-Bus ohne dazwischenliegende Protokollschichten erlaubt.

Eine Besonderheit stellt der so genannte RX-Dispatcher des **LLCF** dar. Durch die Art der Adressierung der CAN-Frames kann es mehrere 'Interessenten' an einer empfangenen CAN-ID geben. Durch die **LLCF**-Funktionen rx_register() und rx_unregister() können sich die Protokollmodule beim **LLCF**-Kernmodul für ein oder mehrere CAN-IDs von definierten CAN-Netzwerkgeräten registrieren, die ihnen beim Empfang automatisch zugestellt werden. Das **LLCF**-Kernmodul sorgt beim Senden auf den CAN-Bus auch für ein lokales Echo ('local loopback') der zu versendenden CAN-Frames, damit für alle Applikationen auf einem System die gleichen Informationen verfügbar sind.

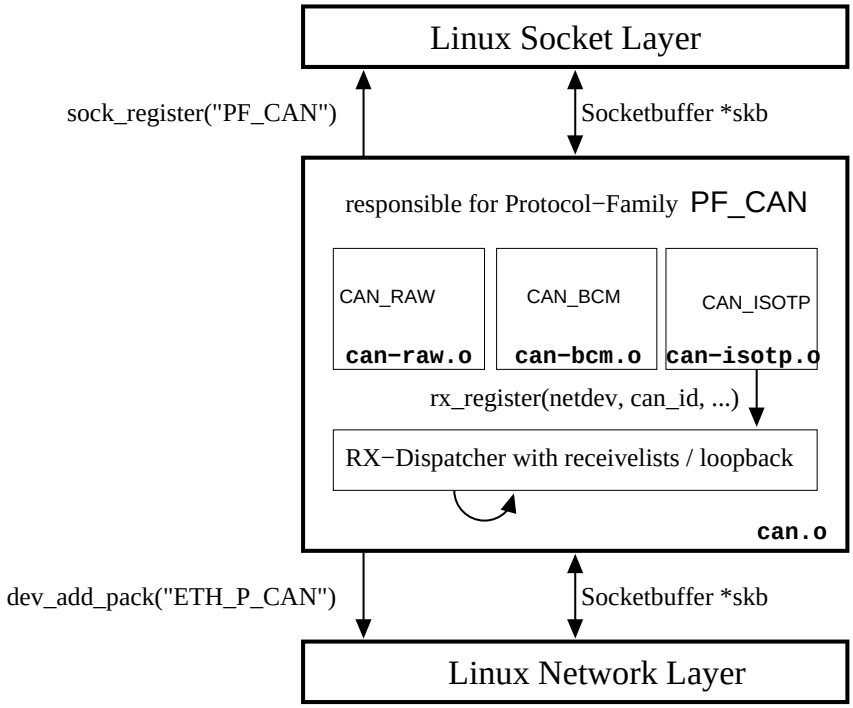

Abbildung 3: Das **LLCF**-Kernmodul im Linux-Kernel

Für die Anbindung der CAN-Netzwerktreiber wurde ein neuer 'Ethernet-Protokoll-Typ' ETH_P_CAN eingeführt, der die Durchleitung der empfangenen CAN-Frames durch die Linux-Netzwerkschicht sicherstellt. Das **LLCF**-Kernmodul meldet sich dazu als Empfänger von ETH_P_CAN-'Ethernetframes' beim Kernel an.

Durch die konsequente Realisierung der Anbindung des CAN-Busses mit Schnittstellen aus der etablierten Standard-Informationstechnologie eröffnen sich für den Anwender (Programmierer) alle Möglichkeiten, die sich auch sonst bei der Verwendung von Sockets zur Kommunikation ergeben. D.h. es können beliebig viele Sockets (auch verschiedener Socket-Typen auf verschiedenen CAN-Bussen) von einer oder mehreren Applikationen gleichzeitig geöffnet werden. Bei der Kommunikation auf verschiedenen Sockets kann beispielsweise mit select(2) auf Daten aus den einzelnen asynchronen Kommunikationskanälen ressourcenschonend gewartet werden.

2 Rechtliche Hinweise

Volkswagen geht davon aus, dass diese rechtlichen Hinweise vom Anwender gelesen, verstanden und akzeptiert worden sind.

Im Quellcode des *Low Level CAN Framework* findet man folgenden Hinweis:

```
/*
 * Copyright (c) 2002-2005 Volkswagen Group Electronic Research
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions, the following disclaimer and
 *    the referenced file 'COPYING'.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. Neither the name of Volkswagen nor the names of its contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * Alternatively, provided that this notice is retained in full, this
 * software may be distributed under the terms of the GNU General
 * Public License ("GPL") version 2 as distributed in the 'COPYING'
 * file from the main directory of the linux kernel source.
 *
 * The provided data structures and external interfaces from this code
 * are not restricted to be used by modules with a GPL compatible license.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
 * OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 * DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 * THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
 * DAMAGE.
 *
 * Send feedback to <llcf@volkswagen.de>
 *
 */
```

2.1 Erweiterter Haftungsausschluss

Im Geltungsbereich der deutschen Rechtsprechung besteht auch bei der kostenlosen Überlassung bei grob fahrlässigen oder vorsätzlich verschwiegenen Mängeln die Möglichkeit, den Urheber für entstandene (Folge-)Schäden haftbar machen zu können.

Wenngleich die Autoren bemüht sind, eine fehlerfreie Software zur Verfügung zu stellen, lassen sich Fehler nicht generell ausschließen. Aus diesem Grunde erklärt sich der Anwender mit dem Einsatz des *Low Level CAN Framework* damit einverstanden, den Haftungsausschluss gegenüber den Autoren und der Volkswagen AG **uneingeschränkt** anzuerkennen und auf jedwede rechtlichen Möglichkeiten/Forderungen beim Auftreten von Mängeln/Schäden/Folgeschäden durch den Einsatz des **LLCF** zu verzichten.

2.2 Logo

Das Logo des *Low Level CAN Framework* ('/dev/< beetle >') ist als zusammengesetzte Bildmarke Eigentum der Volkswagen AG. Es symbolisiert die Integration des Fahrzeugs in eine Umgebung der Standard-Informationstechnologie.

2.3 Linux Module License

Teile der vollständigen **LLCF**-Distribution unterliegen nicht der GNU Public License sondern sind proprietärer Code der Volkswagen AG. Richtigerweise ist dieses auch im Quellcode der einzelnen Kernelmodule mit dem Makro **MODULE_LICENSE("Volkswagen Group closed source")** markiert. Beim Laden der **LLCF**-Module in den Kernel kann daher beispielsweise folgende Fehlermeldung auftreten:

```
Warning: loading can-tp20.o will taint the kernel: Volkswagen Group closed source
See http://www.tux.org/lkml/#export-tainted for information about tainted modules
Module tainted loaded, with warnings
```

Dieses ist eine Warnung, dass das geladene Modul nicht unter der GNU Public License steht, was die Funktionsfähigkeit des *Low Level CAN Framework* jedoch nicht beeinflusst. Typischerweise werden jedoch Anfragen zu Fehlermeldungen, die von 'verschmutzten' tainted-Kernels erzeugt wurden, im Allgemeinen von der Linux-Community nicht kommentiert / beantwortet.

Module, die unter der GNU Public License stehen, enthalten das Makro **MODULE_LICENSE("GPL")**.

2.4 Transportprotokolle

Die in der vollständigen **LLCF**-Distribution enthaltenen Protokoll-Module für die Transport-Protokolle (derzeit VAG TP1.6, VAG TP2.0, Bosch MCNet) sind nur in Verbindung mit einem Non Disclosure Agreement (NDA) für Zulieferer der Volkswagen AG erhältlich. Diese Protokoll-Module sind keine Referenz-Treiber. Nach Auslauf des NDA gelten die im NDA vereinbarten Bedingungen zum Vernichten von Arbeitsergebnissen/Unterlagen/Quellcode.

Der Quelltext für die bezeichneten CAN-Transport-Protokolle ist in klar separierten Modulen und existierte bereits vor einer Integration in den Linux Kernel:

```
This TP code is in clearly seperate modules and had a life outside Linux
from the beginning and does something self-contained that doesn't
really have any impact on the rest of the kernel. The transport
protocol drivers have been originally written for something else and
do not need any but the standard UNIX read/write kind of interfaces.
See <http://www.atnf.csiro.au/people/rgooch/linux/docs/licensing.txt>
```

MCNet ist ein CAN-Transport-Protokoll der Robert Bosch GmbH. Die Implementierung ist nach Maßgabe des MCNet-Disclaimers *ANFORDERUNG DER MCNet-SPEZIFIKATION* vom 13.07.1998 erfolgt. Weitergehende Informationen zu MCNet sind erhältlich bei:

Robert Bosch GmbH
Abteilung K7/EFT62
Postfach 77 77 77
D-31132 Hildesheim

Ansprechpartner:
Dr. Uwe Zurmühl <uwe.zurmuehl@de.bosch.com>
Detlef Rode <detlef.rode@de.bosch.com>

3 Installation

3.1 Übersicht über den Quellcode

Am Beispiel der Verzeichnisstruktur der **LLCF**-Version v1.0.0-rc1 soll der Inhalt des tar-Files vorgestellt werden:

```
llcf-v1/
llcf-v1/doc/
llcf-v1/install/
llcf-v1/src/
llcf-v1/src/drivers/
llcf-v1/src/drivers/sja1000/
llcf-v1/src/drivers/tricore/
llcf-v1/src/test/
```

Die Verzeichnisse beinhalten im Einzelnen:

doc Diese Dokumentation (als LaTeX Quelltext oder nur als PDF).

install Scripts und Dateien zum automatischen Laden der Module.

src Den Quellcode der **LLCF**-Module.

src/drivers Unterverzeichnis für CAN-Netzwerktreiber.

src/drivers/sja1000 Philips SJA1000 Netzwerktreiber.

src/drivers/tricore Infineon TwinCAN TC1920 Netzwerktreiber.

src/test Verschiedene Testrahmen und Beispielcode.

3.2 Kompilieren des Quellcodes

3.2.1 Systemvoraussetzungen

Die **LLCF**-Module werden zur Laufzeit in den Linux-Kernel geladen. Dazu muss sichergestellt sein, dass die Module unter den selben Randbedingungen erzeugt wurden, wie der Kernel und die dazugehörigen Module, die bereits erstellt wurden.

Insbesondere müssen übereinstimmen:

- Die Version des Linux Kernel (z.B. 2.4.31)
- Die Version des verwendeten Compilers (z.B. gcc 3.3)

Die Information, welcher Kernel auf dem System gerade läuft, lässt sich folgendermaßen ermitteln:

```
hartko@pplinux1:~> cat /proc/version
Linux version 2.4.26 (root@vwagwockfe40) (gcc version 2.95.4) #2 Mi Mär 30 14:13:59 CEST 2005
```

Beim Übersetzen des Kernel müssen die folgenden Randbedingungen gegeben sein:

- Linux Kernel Version 2.4 (Eine Anpassung für Kernel 2.6 ist in Arbeit)
- Der Kernel Module Loader muss konfiguriert sein (CONFIG_KMOD)
- Das Dateisystem procfs muss konfiguriert sein (CONFIG_PROC_FS)

Bei einer Linux-Installation (z.B. Knoppix) sollten zumindest die `include`-Dateien des laufenden Kernel (typischerweise unter `/usr/src/linux`) vorhanden sein. Ist dieses nicht der Fall, muss ein aktueller Kernel 2.4 heruntergeladen (www.kernel.org), ausgepackt, compiliert und installiert werden.

3.2.2 Kompilieren der LLCF-Module

Das Kompilieren der **LLCF**-Kernelmodule geschieht im Verzeichnis `src` mit Aufruf des Befehles `make`. Dabei wird als Compiler 'cc' und als Verzeichnis für den Kernel-Quellcode `/usr/src/linux` angenommen.
Soll ein anderer Compiler verwendet werden oder befinden sich die zu verwendenden Kernel-Quellen an einer anderen Stelle, so können diese Standard-Einstellungen geändert werden. Z.B. mit

```
make CC=gcc-2.95 KERNELDIR=/usr/src/linux-2.4.26
```

Zusätzlich besteht die Möglichkeit, die **LLCF**-Kernelmodule mit einer DEBUG-Option zu übersetzen. Mit dem Module-Parameter `debug=1` kann zum Lade-Zeitpunkt des Moduls eine erweiterte Informationsausgabe im Kernel-Log (beispielweise in `/var/log/kern.log`) erreicht werden. Siehe dazu auch der Hinweis in Kapitel 3.3.4. An einem realen Fahrzeug mit mehreren hundert CAN-Frames pro Sekunde sollte man auf diese Möglichkeit allerdings verzichten! Der Parameter, um die DEBUG-Funktionalität mit einzukompilieren lautet `DEBUG=-DDEBUG` - also wird `make` z.B. so aufgerufen:

```
make CC=gcc-2.95 KERNELDIR=/usr/src/linux-2.4.26 DEBUG=-DDEBUG
```

Nach dem Aufruf von `make` sollten verschiedene Kernel-Module (Dateiendung `.o`) im `src`-Verzeichnis erzeugt worden sein. Darunter z.B. die Datei `can.o`.

3.2.3 Kompilieren der CAN-Netzwerk-Module

Analog zu den **LLCF**-Modulen wird beispielsweise im Verzeichnis `src/drivers/sja1000` der Befehl `make` ausgeführt. Das Kernel-Modul für die PC104/ISA-Anbindung des SJA1000-Controllers heißt `sja1000-isa.o`. Der Treiber für das iGate (Jaybrain GW2) heißt `sja1000-gw2.o`.

3.2.4 Kompilieren der Testanwendungen

Die Testanwendungen benötigen keinen konkreten Bezug zum Kernel und können einfach mit `make` im Verzeichnis `src/test` übersetzt werden.

3.3 Installation der Module

Derzeit existiert noch keine Make-Umgebung, die durch ein einfaches `make install` die Installation der Module ausführen kann. Daher diese etwas ausführlichere Anleitung.

3.3.1 Kopieren der Module

Um die Module zu installieren muss, man sich als Benutzer 'root' im System anmelden. Die ladbaren Module werden unter Linux in einer Verzeichnisstruktur in `/lib/modules/<kernelversion>` abgelegt. Für dieses Beispiel gehen wir von einer Kernelversion 2.4.31 aus.

- Neues Verzeichnis für die **LLCF**-Module anlegen: `mkdir /lib/modules/2.4.31/llcf`
- **LLCF**-Module kopieren (in `src`): `cp *.o /lib/modules/2.4.31/llcf`
- ggf. Treiber-Module kopieren z.B.: `cp sja1000-isa.o /lib/modules/2.4.31/llcf`
- Modulabhängigkeiten aktualisieren: `depmod -a`

Dieses Verfahren ist auch für CAN-Netzwerk-Treibermodule anzuwenden, die nicht im **LLCF**-tar-File enthalten sind. Siehe Kapitel 9.

3.3.2 Automatisches Laden und Starten der Module

Der Module-Loader unter Linux kann - bei entsprechender Konfiguration - Kernelmodule automatisch laden. D.h. beim Öffnen eines **LLCF**-Sockets können die entsprechenden Kernelmodule geladen werden, ohne die **LLCF**-Module fest in den Kernel einbinden zu müssen.

Zudem können Modulen beim Ladevorgang Parameter übergeben werden. Diese Funktionalitäten werden durch die Konfigurationseinträge in der Datei `/etc/modules.conf` realisiert.

Für das **LLCF** wurde als Ausgangsbasis im Verzeichnis `install` eine Datei `llcf` angelegt, die ..

- ... an die Datei `/etc/modules.conf` anzuhängen ist ODER
- ... z.B. bei Debian-System nach `/etc/modutils` zu kopieren ist

Bei Debian-Systemen muss nach dem Kopiervorgang oder bei Änderungen der Datei `/etc/modutils/llcf` das Script `update-modules.modutils` aufgerufen werden.

Ein Ausschnitt aus der Datei `llcf` ohne die Modul-Parameter für die unterstütze CAN Hardware:

```
# Low Level CAN Framework
# Copyright (c) 2005 Volkswagen Group Electronic Research
#
# uncomment and edit lines for your specific hardware!
#
# On debian systems copy this file to the directory
# /etc/modutils and say 'update-modules.modutils'.
# Other systems: Add this content to /etc/modules.conf

# protocol family PF_CAN
alias net-pf-30 can

# protocols in PF_CAN
alias can-proto-1 can-tp16
alias can-proto-2 can-tp20
alias can-proto-3 can-raw
alias can-proto-4 can-bcm
alias can-proto-5 can-mcnet
alias can-proto-6 can-isotp
alias can-proto-7 can-bap

# protocol module options
#option tp_gen printstats=1

# virtual CAN devices
alias vcan0 vcan
alias vcan1 vcan
alias vcan2 vcan
alias vcan3 vcan

(..)
```

Für die verwendete CAN-Hardware sind in der Datei `llcf` auch Modul-Parameter vorhanden. Dabei sind besonders die Einstellungen der Bit-Timing-Register (btr) zum Zeitpunkt des Modul-Ladens zu beachten. Entsprechend der verwendeten Hardware sind hier Änderungen durchzuführen.

```
# CAN hardware (uncomment the currently used)

##> Trajet GW2
#alias can0 sja1000-gw2
#alias can1 sja1000-gw2
#alias can2 sja1000-gw2
#alias can3 sja1000-gw2
#options sja1000-gw2 speed=500,100,500,100
```

```
#options sja1000-gw2 btr=0xC03D,0xC4F9,0xC03D,0xC4F9

##> Peak System hardware (ISA/PCI/Parallelport Dongle)
##> to set BTR-values to PCI-devices see Peak System documentation
##> e.g. echo "i 0x4914 e" > /dev/pcan0
#alias can0 pcan
#alias can1 pcan
#alias can2 pcan
#options pcan type=isa,isa io=0x2C0,0x320 irq=10,5 btr=0x4914,0x4914
#options pcan type=epp btr=0x4914
#options parport_pc io=0x378 irq=7

##> EMS Wuensche CPC-Card
#options cpc-card btr=0x4914,0x4914
#
##> add the following lines to /etc/pcmcia/config.opts (!)
## EMS Wuensche CPC-Card CAN Interface
#device "cpc-card_cs"
#  module "cpc-card", "cpc-card_cs"
#  card "EMS Dr. Thomas Wuensche CPC-Card CAN Interface"
#  version "EMS_T_W", "CPC-Card", "*", "*"
#  bind "cpc-card_cs"
```

Für die häufig verwendeten Philips SJA1000 CAN-Controller ergeben sich bei einem Controller-Takt von 16 MHz beispielsweise folgende Werte für die Bit-Timing-Register (`btr=<xx>`):

0x4914 100 kBit

0x4114 500 kBit

0x4014 1000 kBit

3.3.3 Automatisches Hochfahren der CAN-Netzwerk-Interfaces

Die CAN-Netzwerk-Interfaces können wie jedes andere Netzwerk-Interface auch mit dem Befehl `ifconfig` hoch- und heruntergefahren werden.

```
hartko@pplinux1:~> ifconfig can0 up
hartko@pplinux1:~> ifconfig can0 down
```

Für das automatische Hoch- und Herunterfahren der CAN-Netzwerk-Interfaces wurde im Verzeichnis `install` eine Datei `can_if` angelegt, die in das Verzeichnis `/etc/init.d` zu kopieren ist. In dieser Datei sind drei Variable angelegt, die die zu bearbeitenden Interfaces bestimmen.

```
CAN_IF="can0 can1"
VCAN_IF="vcan0 vcan1"
PROBE=""
```

In diesem Beispiel werden beim Starten des Systems die Interfaces 'can0', 'can1', 'vcan0' und 'vcan1' hochgefahren.
Mit `/etc/init.d/can_if start` kann man als Benutzer 'root' die Interfaces starten und mit `/etc/init.d/can_if stop` wieder herunterfahren (beispielsweise, wenn neue Kernel-Module installiert werden sollen und die alten mit `rmmod <modulname>` entfernt werden sollen).

Die Variable `PROBE` ermöglicht es dem Anwender, zum Startzeitpunkt der CAN-Interfaces mit `modprobe` Kernelmodule zu laden, noch bevor die automatische Ladefunktionalität durch das Öffnen eines Socket ausgeführt wird. Hintergrund: Auf sehr langsamen Systemen kann ein zeitnahes Laden nicht immer in der Art gewährleistet werden, wie es manche Protokoll-Spezifikation verlangen. Durch das Setzen der Variable `PROBE="can-tp20 can-tp16"` werden beispielsweise diese Protokollmodule im Speicher vorgehalten.

Soll das Script `/etc/init.d/can_if` beim Systemstart automatisch gestartet werden, müssen gemäß den Runleveln im SystemV Init symbolische Links gesetzt werden. Beispielsweise:

```
root@pplinux1:/# ln -sf /etc/init.d/can_if /etc/rc0.d/S35can_if
root@pplinux1:/# ln -sf /etc/init.d/can_if /etc/rc6.d/S35can_if
root@pplinux1:/# ln -sf /etc/init.d/can_if /etc/rcS.d/S40can_if
```

3.3.4 Modul Parameter der LLCF-Module

Bezugnehmend auf die in Kapitel 3.3.2 vorgestellte Datei `llcf` wird hier auf die Modul-Parameter der **LLCF**-Module eingegangen. Wurde beim Kompilieren (siehe Kapitel 3.2.2) die Option `DEBUG=-DDEBUG` mit angegeben, kann beim Laden des Modules ein Parameter `debug=<x>` angegeben werden. Der Wert `<x>` ist dabei binär kodiert und bedeutet:

Bit 0 gesetzt Debug-Ausgabe des Moduls eingeschaltet

Bit 1 gesetzt Ausgabe der Socket-Buffer-Daten eingeschaltet

Es ist möglich (wenn auch nicht empfehlenswert) in der Datei `can` zum Beispiel die Zeile

```
option can debug=1
```

einzutragen. Besser ist es, mit `insmod can debug=1` das Modul einmalig zum Testen 'von Hand' zu laden. Ggf. muss es vorher allerdings mit `rmmod can` entfernt werden (siehe dazu Kapitel 3.3.5).

Der generische Transportprotokoll-Treiber `can-tpgen` bietet die Option, sich die Statistiken einer Transportprotokollverbindung (Anzahl der Pakete, Anzahl der Bytes, Anzahl der Retries) im Kernel-Log ausgeben zu lassen. Dazu muss die Option `printstats=1` gesetzt sein. Da hier nicht viele Daten anfallen, kann dieses bei Bedarf auch in der Datei `llcf` eingetragen werden.

```
option can-tpgen printstats=1
```

3.3.5 Entfernen von geladenen LLCF-Modulen

ACHTUNG! Zum Entfernen von geladenen **LLCF**-Modulen müssen zunächst immer alle Applikationen, die auf das **LLCF** aufsetzen, beendet und alle CAN-Interfaces heruntergefahren werden (z.B. mit `/etc/init.d/can_if stop`).

In diesem Beispiel sind die CAN-Interfaces 'vcan0' und 'vcan1' noch hochgefahren, was beim Module 'vcan' zu einem Usage-Count von zwei führt.

```
root@pplinux1:/# lsmod
Module                  Size  Used by    Tainted: P
pcan                   29424   1 (autoclean)
vcan                    2560   2 (autoclean)
can-tp20                6692   0 (unused)
can-tpgen               5368   0 [can-tp20]
can-bcm                 7940   0 (unused)
can-raw                 2564   0 (unused)
can                    10432   0 [can-tp20 can-tpgen can-bcm can-raw]
(..)
```

Nach dem Herunterfahren der CAN-Interfaces kann man entsprechend den Abhängigkeiten die Module entfernen. Im Beispiel hängt `can-tp20` von `can-tpgen` ab und `can-tp20 can-tpgen can-raw vcan` hängen von `can` ab. D.h. die Reihenfolge zum Entladen der Module ist:

```
rmmod can-tp20 can-tpgen can-raw vcan can
```

Das kann man auch mit einzelnen `rmmod`-Aufrufen realisieren. Wenn ein Modul nicht entladen werden kann, gibt es eine Fehlermeldung.

4 Allgemeine Hinweise zur Socket-API beim LLCF

Für die Kommunikation auf dem CAN-Bus wird eine neue Protokoll-Familie `PF_CAN` im Socket-Layer implementiert. Aus Anwender- bzw. Programmierersicht wird mit den **LLCF**-Sockets analog zu Internet-Protokoll-Sockets (Protokoll-Familie `PF_INET`) mit den üblichen Systemaufrufen socket(2), bind(2), listen(2), accept(2), connect(2), read(2), write(2) und close(2) genutzt. Siehe dazu auch die Einleitung in Kapitel 1.

Im Gegensatz zur Adressstruktur der Internet-Adressen (`sockaddr_in`) benötigt die Adressierung der PF_CAN-Sockets andere Inhalte. Die Adressstruktur `sockaddr_can` ist in der Include-Datei `af_can.h` definiert:

```
struct sockaddr_can {
    sa_family_t   can_family;
    int           can_ifindex;
    union {
        struct { canid_t rx_id, tx_id; } tp16;
        struct { canid_t rx_id, tx_id; } tp20;
        struct { canid_t rx_id, tx_id; } mcnet;
    } can_addr;
};
```

Neben dem Interface-Index des CAN-Interfaces sind hierbei besonders für die jeweiligen Transport-Protokolle die CAN-IDs `tx_id` und `rx_id` relevant. Transport-Protokolle bilden auf dem CAN-Bus auf zwei CAN-IDs eine virtuelle Punkt-zu-Punkt-Verbindung ab.

Eine weitere wichtige Struktur stellt das CAN-Frame dar, dass in der Include-Datei `can.h` definiert ist:

```
typedef __u32 canid_t;

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

/* special address description flags for the CAN_ID */
#define CAN_EFF_FLAG 0x80000000U /* EFF/SFF is set in the MSB */
#define CAN_RTR_FLAG 0x40000000U /* remote transmission request */

/* valid bits in CAN ID for frame formats */
#define CAN_SFF_MASK 0x000007FFU /* standard frame format (SFF) */
#define CAN_EFF_MASK 0x1FFFFFFFU /* extended frame format (EFF) */
```

Die definierte Struktur `can_frame` enthält die Elemente eines CAN-Frame, wie es auf dem CAN-Bus definiert ist. Die Anordnung der Nutzdaten (Byte-Order, Word-Order, Little/Big Endian) ist auf dem CAN-Bus generell nicht definiert, weshalb die Datenelemente `data[]` als Array von 8 Byte ausgeführt sind. Da die Datenelemente allerdings auf einer 8 Byte Speichergrenze ausgerichtet, ist hier auch ein Zugriff bis zu einer Breite von 64 Bit möglich:

```
#define U64_DATA(p) (*(unsigned long long*)(p)->data)

U64_DATA(&myframe) = 0xFFFFFFFFFFFFFFFFULL;
U64_DATA(&myframe) = 0;
```

Durch die Trennung der Informationen in die Include-Dateien `af_can.h` und `can.h` ist zum Erstellen eines CAN-Netzwerk-Treibers nur die Include-Datei `can.h` nötig.

4.1 Zeitstempel

Für die Anwendungen im CAN-Umfeld ist häufig ein genauer Zeitstempel von Interesse, der den Empfangszeitpunkt einer Nachricht vom CAN-Bus wiedergibt. Ein solcher zugehöriger Zeitstempel kann über ein ioctl(2) nach dem Lesen einer Nachricht vom Socket ausgelesen werden. Dieses gilt auch für die Sockets von Transportprotokollen, wobei hier der Zeitstempel der letzten zugehörigen TPDU ausgegeben wird. Der Aufruf - z.B. `ioctl(s, SIOCGSTAMP, &tv)` - wird in den jeweiligen Testprogrammen zur Veranschaulichung verwendet.

Die Zeitstempel haben unter Linux eine Auflösung von einer Mikrosekunde und werden beim Empfang eines CAN-Frame im CAN-Networkdevice automatisch gesetzt.

5 RAW-Sockets

RAW-Sockets erlauben, Nachrichten direkt auf einem CAN-Bus zu senden und alle Nachrichten, die auf einem CAN-Bus übertragen werden, zu lesen. Geöffnet wird ein RAW-Socket durch

```
s = socket(PF_CAN, SOCK_RAW, 0);
```

Der geöffnete Socket muss zunächst mittels bind(2) an einen CAN-Bus gebunden werden. Dabei spielen die für Transportprotokolle benötigten Adress-Elemente `tx_id` und `rx_id` in der Struktur `struct sockaddr_can` keine Rolle. Der folgende Code bindet den geöffneten Socket s an das CAN-Interface `can1`:

```
struct sockaddr_can addr;
struct ifreq ifr;

addr.can_family = AF_CAN;
strcpy(ifr.ifr_name, "can1");
ioctl(s, SIOCGIFINDEX, &ifr);
addr.can_ifindex = ifr.ifr_ifindex;
bind(s, (struct sockaddr *)&addr, sizeof(addr));
```

Es können nun mit read(2) alle auf dem Bus empfangenen CAN-Frames gelesen und mit write(2) beliebige CAN-Frames gesendet werden.

Die mit read(2) und write(2) übertragenen Daten haben die Struktur `struct can_frame`. Jeder zu sendende CAN-Frame muss mit *einem* Aufruf von write(2) übergeben werden und empfangene CAN-Frame müssen mit *einem* Aufruf von read(2) gelesen werden.

Zum gleichzeitigen Empfang von Nachrichten *aller* CAN-Netzwerk-Interfaces (z.B. mit recvfrom(2)) ist als Interface-Index Null (im Beispiel: `addr.can_ifindex = 0;`) einzutragen. Das Senden von CAN-Frames über einen solchen RAW-Socket muss dann über sendmsg(2) erfolgen.

Der RAW-Socket bietet eine einfache Filterfunktion mit der Bereiche von CAN-IDs aus dem Datenstrom ausgefiltert werden können. Dazu kann noch vor dem Aufruf von bind(2) mit dem Systemaufruf setsockopt(2) ein Array einfacher Filter gesetzt werden. In diesem Beispiel sollen alle CAN-IDs von 0x200 - 0x2FF durchgelassen werden:

```
struct can_filter rfilter;

rfilter.can_id   = 0x200; /* SFF frame */
rfilter.can_mask = 0xF00;

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

Der Filter läßt sich mit `rfilter.can_id |= CAN_INV_FILTER;` auch invertieren, wodurch in diesem Fall die CAN-IDs von 0x200 - 0x2FF nicht durchgelassen werden würden.

```
struct can_filter rfilter[4];

rfilter[0].can_id   = 0x80001234;   /* Exactly this EFF frame */
rfilter[0].can_mask = CAN_EFF_MASK; /* 0x1FFFFFFFU all bits valid */
rfilter[1].can_id   = 0x123;        /* Exactly this SFF frame */
rfilter[1].can_mask = CAN_SFF_MASK; /* 0x7FFU all bits valid */
rfilter[2].can_id   = 0x200 | CAN_INV_FILTER; /* all, but 0x200-0x2FF */
rfilter[2].can_mask = 0xF00;        /* (CAN_INV_FILTER set in can_id) */
rfilter[3].can_id   = 0;            /* don't care */
rfilter[3].can_mask = 0;            /* ALL frames will pass this filter */

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

Für die Anwendung der Filterfunktion muss die Include-Datei `raw.h` eingebunden werden. Eine Veränderung des Filters zur Laufzeit ist über weitere Aufrufe von setsockopt(2) möglich.

5.1 Testprogramme

candump ist ein Programm, das über den RAW-Socket CAN-Frames von einem oder mehreren CAN-Device(s) einliest und in lesbarer Form - auf Wunsch mit Zeitstempeln - ausgibt. Die beschriebene Filterfunktion der RAW-Sockets (s.o.) ist über Komandozeilenparameter einstellbar. Ebenso wie eine Ausgabe im bekannten ASC-Format.

Wird `candump` ohne Parameter aufgerufen, erscheint ein Hilfetext.

```
hartko@pplinux1:~/llcf/src/test > ./candump
Usage: candump [can-interfaces]
Options: -m <mask>   (default 0x00000000)
         -v <value>  (default 0x00000000)
         -i <0|1>    (inv_filter)
         -t <type>   (timestamp: Absolute/Delta/Zero)
         -c          (color mode)
         -s <level>  (silent mode - 1: animation 2: nothing)
         -b <can>    (bridge mode - send received frames to <can>)
         -a          (create ASC compatible output)
         -1          (increment interface numbering in ASC mode)
         -A          (enable ASCII output)

When using more than one CAN interface the options
m/v/i have comma seperated values e.g. '-m 0,7FF,0'
```

Beispiele für die Benutzung von `candump`:

Einfaches Anzeigen von zwei CAN-Bussen. Timestamps beginnen bei Null.

```
hartko@pplinux1:~/llcf/src/test > candump can0 can1 -t z
(0.000000) can0  3FC  [1] 05
(0.001185) can0   64  [8] 20 14 3F 16 B8 0B 98 3A
(0.002396) can0   66  [8] 39 00 A1 45 00 00 00 00
(0.015395) can0   C9  [6] 13 01 00 00 10 27
(0.028665) can1  110  [3] 87 00 00
(0.049990) can0  3FC  [1] 05
(0.051176) can0   64  [8] 20 14 3F 16 B8 0B 98 3A
(0.052386) can0   66  [8] 39 00 A1 45 00 00 00 00
(0.065397) can0   C9  [6] 13 01 00 00 10 27
(0.099974) can0  3FC  [1] 05
(0.101159) can0   64  [8] 20 14 3F 16 B8 0B 98 3A
```

Einfaches Anzeigen von zwei CAN-Bussen. Timestamps beginnen bei Null. Ausgabe im ASC-Format. Die Kanalnummern werden um 1 erhöht (`can0` $\Rightarrow$ 1, `can1` $\Rightarrow$ 2).

```
hartko@pplinux1:~/llcf/src/test > candump can0 can1 -t z -a -1
date Tue Oct 18 09:39:57 2005
base hex  timestamps absolute
no internal events logged
   0.0000 1  3FC             Rx   d 1 05
   0.0011 1  64              Rx   d 8 20 14 3F 14 B8 0B 98 3A
   0.0023 1  66              Rx   d 8 49 00 A1 45 00 00 00 00
   0.0154 1  C9              Rx   d 6 13 01 00 00 10 27
   0.0286 2  110             Rx   d 3 87 00 00
   0.0500 1  3FC             Rx   d 1 05
   0.0512 1  64              Rx   d 8 20 14 3F 14 B8 0B 98 3A
   0.0524 1  66              Rx   d 8 49 00 A1 45 00 00 00 00
   0.0654 1  C9              Rx   d 6 13 01 00 00 10 27
   0.0999 1  3FC             Rx   d 1 05
   0.1011 1  64              Rx   d 8 20 14 3F 14 B8 0B 98 3A
```

Filtern von Nachrichten auf `can0` mit Bit-Maske 0x7FC und Vergleichswert 0x66. Zeitstempel mit Differenzzeit.

```
hartko@pplinux1:~/llcf/src/test > candump can0 -m 0x7FC -v 0x66 -t d
CAN ID filter[0] for can0 set to mask = 000007FC, value = 00000066
(0.000000) can0   64  [8] 1B 14 3F 21 B8 0B 98 3A
(0.001202) can0   66  [8] B9 DA A0 45 00 00 00 00
(0.048833) can0   64  [8] 1B 14 3F 21 B8 0B 98 3A
(0.001192) can0   66  [8] EB DA A0 45 00 00 00 00
(0.048790) can0   64  [8] 1B 14 3F 21 B8 0B 98 3A
```

Filtern von Nachrichten auf `can0` mit Bit-Maske 0x7FC und Vergleichswert 0x66. Der CAN-Bus `can1` wird ohne Filterung angegeigt.

```
hartko@pplinux1:~/llcf/src/test > candump can0 can1 -m 0x7FC,0 -v 0x66,0 -t d
CAN ID filter[0] for can0 set to mask = 000007FC, value = 00000066
(0.000000) can0   64  [8] 20 14 3F 14 B8 0B 98 3A
(0.001202) can0   66  [8] 48 00 A1 45 00 00 00 00
(0.048794) can0   64  [8] 20 14 3F 14 B8 0B 98 3A
(0.001201) can0   66  [8] 48 00 A1 45 00 00 00 00
(0.026214) can1  110  [3] 87 00 00
(0.003006) can1  41B  [4] 1C 12 02 FF
(0.019612) can0   64  [8] 20 14 3F 14 B8 0B 98 3A
(0.001201) can0   66  [8] 48 00 A1 45 00 00 00 00
(0.048770) can0   64  [8] 20 14 3F 14 B8 0B 98 3A
```

Invertiertes Filtern von Nachrichten auf `can0` mit Bit-Maske 0x7FC und Vergleichswert 0x66. Der CAN-Bus `can1` wird ohne Filterung angegeigt. Der Timestamp wird absolut ausgegeben, d.h. in Sekunden seit 01.01.1970. Analog `'date +%s'` - siehe date(1).

```
hartko@pplinux1:~/llcf/src/test > candump can0 can1 -m 0x7FC,0 -v 0x66,0 -i 1,0 -t a
CAN ID filter[0] for can0 set to mask = 000007FC, value = 00000066 (inv_filter)
(1129625880.726372) can0   C9  [6] 13 01 00 00 10 27
(1129625880.739543) can1  110  [3] 87 00 00
(1129625880.760949) can0  3FC  [1] 05
(1129625880.776377) can0   C9  [6] 13 01 00 00 10 27
(1129625880.810983) can0  3FC  [1] 05
(1129625880.811580) can1  41C  [4] 1D 12 02 FF
(1129625880.826379) can0   C9  [6] 13 01 00 00 10 27
(1129625880.839544) can1  110  [3] 87 00 00
(1129625880.860955) can0  3FC  [1] 05
(1129625880.876380) can0   C9  [6] 13 01 00 00 10 27
(1129625880.910986) can0  3FC  [1] 05
```

tst-raw ist ein Programm zum Testen des RAW-Sockets. Es nutzt den read(2) Systemcall.

tst-raw-filter ist ein Programm zum Testen der Filterfunktion des RAW-Sockets. Es nutzt den recvfrom(2) Systemcall.

tst-raw-sendto ist ein Programm zum Senden eines CAN-Frame auf einem nicht an ein besonders Interface gebundenen RAW-Socket mit dem sendto(2) Systemcall.

canecho ist ein Programm, dass nach dem Empfang eines CAN-Frames dieses mit einer um 1 erhöhten CAN-ID wieder aussendet. Es war für die ersten Versuche mit einem realen CAN-Bus implementiert worden. Seit dem **LLCF** V0.6 ist jedoch die lokale Echofunktionalität realisiert, so dass `canecho` nur noch dazu geeignet ist, einen Volllast-Test auszuführen ...

6 Sockets für Transport-Protokolle

Die betrachteten CAN-Transport-Protokolle bilden auf dem CAN-Bus auf zwei CAN-IDs eine virtuelle Punkt-zu-Punkt-Verbindung ab. Dazu wird im Ersten der Acht in einem CAN-Frame vorhandenen Nutzbytes die protokollspezifische Information übertragen, die das korrekte Segmentieren von Nutzdaten gewährleistet. Die restlichen (maximal) sieben Nutzbytes des CAN-Frames enthalten die segmentierten Nutzdaten.

Für die Transport-Protokolle TP1.6, TP2.0, etc. wird ein Socket vom Typ SEQPACKET geöffnet unter Angabe des zu verwendenden Protokolls:

```
s = socket(PF_CAN, SOCK_SEQPACKET, CAN_TP16);
s = socket(PF_CAN, SOCK_SEQPACKET, CAN_TP20);
s = socket(PF_CAN, SOCK_SEQPACKET, CAN_MCNET);
```

Protokollspezifische Parameter können nach dem Öffnen eines Sockets mit setsockopt(2) und getsockopt(2) gesetzt bzw. gelesen werden. Siehe dazu auch die protokollspezifischen Hinweise am Ende dieses Kapitels ab Seite 22.
Der Verbindungsaufbau erfolgt ähnlich wie mit TCP/IP-Sockets. Der wesentliche Unterschied besteht darin, dass ein Prozess, der auf einen Verbindungsaufbau wartet, also die Rolle eines Servers spielt, angeben muss, von welchem Client er Verbindungen annehmen möchte, d.h. er muss die CAN-ID von CAN-Frames angeben, die er auf diesem Socket empfangen möchte. Zusätzlich muss er dem Socket-Layer gegenüber angeben, welche CAN-ID in den von ihm gesendeten CAN-Frames zu verwenden ist.
Analog muss der Client beim Verbindungsaufbau nicht nur die CAN-ID seines Kommunikationspartners, sondern auch seine eigene angeben. Die bei bind(2) und connect(2) verwendeten Strukturen vom Typ `struct sockaddr_can` enthalten daher im Gegensatz zu TCP/IP nicht nur eine Adresse, sondern immer die „Adressen“ beider Kommunikationspartner. Weil die CAN-Architektur kein Routing anhand von netzweiten Adressen kennt, muss außerdem zusätzlich auch immer das CAN-Interface angegeben werden, auf dem die Kommunikation stattfinden soll. Die Struktur ist daher folgendermaßen definiert:

```
struct sockaddr_can {
    sa_family_t   can_family;
    int           can_ifindex;
    union {
        struct { canid_t rx_id, tx_id; } tp16;
        struct { canid_t rx_id, tx_id; } tp20;
        struct { canid_t rx_id, tx_id; } mcnet;
    } can_addr;
};
```

Im Folgenden werden zwei kurze Code-Beispiele angegeben, die die Verwendung von Sockets auf der Server- und der Client-Seite verdeutlichen sollen. Im Beispiel soll eine TP2.0-Verbindung aufgebaut werden, wobei der Client die CAN-ID 0x740 und der Server die CAN-ID 0x760 verwendet. Dieses Beispiel ist dahingehend vereinfacht, dass auf eine Fehlerbehandlung verzichtet wird. Eine mögliche Fehlerbehandlung ist aber in den Beispielprogrammen in Kapitel 6.6 realisiert.

```
/* This is the server code */

int s, n, nbytes, sizeofpeer=sizeof(struct sockaddr_can);
struct sockaddr_can addr, peer;
struct ifreq ifr;

s = socket(PF_CAN, SOCK_SEQPACKET, CAN_TP20);

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

```
addr.can_addr.tp20.tx_id  = 0x760;
addr.can_addr.tp20.rx_id  = 0x440;

bind(s, (struct sockaddr *)&addr, sizeof(addr));
listen(s, 1);

n = accept(s, (struct sockaddr *)&peer, sizeof(peer));

read(n, data, nbytes);
write(n, data, nbytes);

close(n);
close(s);



/* This is the client code */

int s, nbytes;
struct sockaddr_can addr;
struct ifreq ifr;

s = socket(PF_CAN, SOCK_SEQPACKET, CAN_TP20);

addr.can_family = AF_CAN;
strcpy(ifr.ifr_name, "can0");
ioctl(s, SIOCGIFINDEX, &ifr);
addr.can_ifindex = ifr.ifr_ifindex;
addr.can_addr.tp20.tx_id  = 0x440;
addr.can_addr.tp20.rx_id  = 0x760;

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

write(s, data, nbytes);
read(s, data, nbytes);

close(s);
```

6.1 Tracemode

Wie schon beim RAW-Socket (Kapitel 5) besteht auch bei den Transport-Protokoll-Sockets (TP-Sockets) die Möglichkeit über setsockopt(2) die Eigenschaften des Sockets zu beeinflussen. Diese sind zumeist spezifisch für das jeweilige Protokoll. Beim **LLCF** besteht die bisher in allen Transportprotokollen realisierte Möglichkeit, die TP-Sockets mit der Socketoption `TRACE_MODE` in einen Nur-Lese-Modus zu schalten, bei dem der empfangene, segmentierte Datenstrom zusammengesetzt wird, ohne dem Sender Bestätigungen zu senden. Für das Mitschneiden einer bi-direktionalen Verbindung müssen daher zwei Sockets mit 'verdrehten' CAN-IDs `tx_id` und `rx_id` geöffnet werden.

Vereinfachtes Beispiel (ohne Fehlerbehandlung) aus einer älteren Version vom Testprogramm `mcnet-sniffer.c`:

```
int s, t;
struct sockaddr_can addr1, addr2;
struct can_mcnet_options opts;

s = socket(PF_CAN, SOCK_SEQPACKET, CAN_MCNET);
t = socket(PF_CAN, SOCK_SEQPACKET, CAN_MCNET);

opts.blocksize = 15;
opts.config = TRACE_MODE;
setsockopt(s, SOL_CAN_MCNET, CAN_MCNET_OPT, &opts, sizeof(opts));
setsockopt(t, SOL_CAN_MCNET, CAN_MCNET_OPT, &opts, sizeof(opts));
```

```
addr1.can_family = AF_CAN;
strcpy(ifr.ifr_name, "can0");
ioctl(s, SIOCGIFINDEX, &ifr);
addr1.can_ifindex = ifr.ifr_ifindex;
addr1.can_tx_id  = 0x248;
addr1.can_rx_id  = 0x448;

addr2.can_family = AF_CAN;
addr2.can_ifindex = ifr.ifr_ifindex; /* also can0 */
addr2.can_tx_id  = 0x448;
addr2.can_rx_id  = 0x248;

connect(s, (struct sockaddr *)&addr1, sizeof(addr1));
connect(t, (struct sockaddr *)&addr2, sizeof(addr2));

(..)
```

Mit select(2) kann nun auf beiden Sockets auf eintreffende Daten ressourcenschonend gewartet werden.

6.2 Besonderheiten des VAG TP1.6

Das VAG Transportprotokoll TP1.6 besitzt 6 konfigurierbare Parameter, die mit setsockopt(2) gesetzt werden können. Dazu gehören die Timer T1 bis T4, die Blocksize und ein Konfigurationswert, der z.B. angibt, ob ein Kommunikationskanal nach einer bestimmten Zeit automatisch geschlossen werden soll oder nicht. Diese Parameter können beispielsweise wie folgt gesetzt werden:

```
struct can_tp16_options opts;

opts.t1.tv_sec       = 0; /* ACK timeout 100ms */
opts.t1.tv_usec      = 100000;
opts.t2.tv_sec       = 0; /* unused */
opts.t3.tv_sec       = 0; /* transmit delay 10ms */
opts.t3.tv_usec      = 10000;
opts.t4.tv_sec       = TP16_T4_DISABLED; /* disabled */
opts.blocksize       = 11;

opts.config = USE_DISCONNECT | HALF_DUPLEX | ENABLE_BREAK;

setsockopt(s, SOL_CAN_TP16, CAN_TP16_OPT, &opts, sizeof(opts));
```

Die für das Transportprotokoll TP1.6 relevanten Optionen finden sich in den Dateien `tp16.h` und `tp_conf.h`.

Die Struktur `can_tp16_options` ist definiert als

```
struct can_tp16_options {

  struct timeval t1;       /* ACK timeout for DT TPDU. VAG: T1 */
  struct timeval t2;       /* max. time between two DT TPDU's. VAG: T2 (NOT IMPLEMENTED!) */
  struct timeval t3;       /* transmit delay for DT TPDU's. VAG: T3 */
  struct timeval t4;       /* receive timeout for automatic disconnect. VAG: T4 */

  unsigned char blocksize; /* max number of unacknowledged DT TPDU's (1 ..15) */
  unsigned int  config;    /* analogue tp_user_data.conf see tp_gen.h */

};
```

Die bei setsockopt(2) für VAG TP1.6 gesetzten Werte werden dem Kommunikationspartner im Rahmen des Channel-Setup (CS/CA) mitgeteilt und beeinflussen somit ausschließlich die Kommunikationsparameter des Kommunikationspartners.
Eine weitere Besonderheit beim VAG TP1.6 ist der 'dynamische Kanalaufbau', bei dem vor der eigentlichen Kommunikation die CAN-Identifier für den Transportkanal ermittelt werden. Dabei

existieren auch zeitliche Anforderungen, die eine maximale Zeitspanne zwischen dem Aushandeln der Identifier und der Eröffnung des Transportkanals festlegen. Siehe dazu auch die Hinweise zur Variablen `PROBE` in Kapitel 3.3.3.

Entgegen bisherigen Implementierungen unterstützt diese Realisierung für das *Low Level CAN Framework* die dynamische Identifiervergabe nicht im Rahmen der TP2.0-Implementierung. Übertragen auf die IT-Welt entspräche eine solche Implementierung der Integration des Domain-Name-Service in das IP-Protokoll. Das o.g. Verfahren wird im **LLCF** über Broadcastnachrichten auf der Benutzerebene realisiert. Siehe dazu die protokollspezifischen Testprogramme in Kapitel 6.6.

6.3 Besonderheiten des VAG TP2.0

Das VAG Transportprotokoll TP2.0 besitzt 6 konfigurierbare Parameter, die mit setsockopt(2) gesetzt werden können. Dazu gehören die Timer T1 bis T4, die Blocksize und ein Konfigurationswert, der z.B. angibt, ob ein regelmäßiger Connection Test durchgeführt werden soll oder nicht. Diese Parameter können beispielsweise wie folgt gesetzt werden:

```
struct can_tp20_options opts;

opts.t1.tv_sec       = 0; /* ACK timeout 100ms */
opts.t1.tv_usec      = 100000;
opts.t2.tv_sec       = 0; /* unused */
opts.t3.tv_sec       = 0; /* transmit delay 10ms */
opts.t3.tv_usec      = 10000;
opts.t4.tv_sec       = 0; /* unused */
opts.blocksize       = 11;
opts.config = USE_CONNECTIONTEST | USE_DISCONNECT | ENABLE_BREAK;

setsockopt(s, SOL_CAN_TP20, CAN_TP20_OPT, &opts, sizeof(opts));
```

Die für das Transportprotokoll TP2.0 relevanten Optionen finden sich in den Dateien `tp20.h` und `tp_conf.h`.

Die Struktur `can_tp20_options` ist definiert als

```
struct can_tp20_options {

  struct timeval t1;       /* ACK timeout for DT TPDU. VAG: T1 */
  struct timeval t2;       /* unused */
  struct timeval t3;       /* transmit delay for DT TPDU's. VAG: T3 */
  struct timeval t4;       /* unused */

  unsigned char blocksize; /* max number of unacknowledged DT TPDU's (1 ..15) */
  unsigned int  config;    /* analogue tp_user_data.conf see tp_gen.h */

};
```

Die bei setsockopt(2) für VAG TP2.0 gesetzten Werte werden dem Kommunikationspartner im Rahmen des Channel-Setup (CS/CA) mitgeteilt und beeinflussen somit ausschließlich die Kommunikationsparameter des Kommunikationspartners.
Eine weitere Besonderheit beim VAG TP2.0 ist der 'dynamische Kanalaufbau', bei dem vor der eigentlichen Kommunikation die CAN-Identifier für den Transportkanal ermittelt werden. Dabei existieren auch zeitliche Anforderungen, die eine maximale Zeitspanne zwischen dem Aushandeln der Identifier und der Eröffnung des Transportkanals festlegen. Siehe dazu auch die Hinweise zur Variablen `PROBE` in Kapitel 3.3.3.

Entgegen bisherigen Implementierungen unterstützt diese Realisierung für das *Low Level CAN Framework* die dynamische Identifiervergabe nicht im Rahmen der TP2.0-Implementierung. Übertragen auf die IT-Welt entspräche eine solche Implementierung der

Integration des Domain-Name-Service in das IP-Protokoll. Das o.g. Verfahren wird im **LLCF** über Broadcastnachrichten auf der Benutzerebene realisiert. Siehe dazu die protokollspezifischen Testprogramme in Kapitel 6.6.

6.4 Besonderheiten des Bosch MCNet

Das Transportprotokoll MCNet besitzt 3 konfigurierbare Parameter, die mit setsockopt(2) gesetzt werden können. Dazu gehören die Blocksize und ein Konfigurationswert, der z.B. angibt, ob ein regelmäßiger Connection Test durchgeführt werden soll oder nicht. Diese Parameter können beispielsweise wie folgt gesetzt werden:

```
struct can_mcnet_options opts;

opts.blocksize  = 11;
opts.td.tv_sec  = 0; /* no transmit delay */
opts.td.tv_usec = 0;
opts.config     = USE_CONNECTIONTEST;

setsockopt(s, SOL_CAN_MCNET, CAN_MCNET_OPT, &opts, sizeof(opts));
```

Die für das Transportprotokoll MCNet relevanten Optionen finden sich in den Dateien `mcnet.h` und `tp_conf.h`.

Die Struktur `can_mcnet_options` ist definiert als

```
struct can_mcnet_options {

  unsigned char blocksize; /* max number of unacknowledged DT TPDU's (1 ..15) */
  struct timeval td;       /* transmit delay for DT TPDU's */
  unsigned int  config;    /* analogue tp_user_data.conf see tp_gen.h */

};
```

Die bei setsockopt(2) für MCNet gesetzten Werte beeinflussen die lokalen Kommunikationsparameter.

6.5 ISO-Transportprotokoll

Eine Implementierung des CAN-Transportprotokolls nach ISO/DIS 15765 ist in Arbeit und wird unter BSD/GPL-Lizenz von Volkswagen zur Verfügung gestellt werden.

6.6 Testprogramme

tp16-client ist ein Programm, dass aktiv eine TP1.6 Verbindung eröffnet und dann Daten sendet und empfängt. Es wird auch gezeigt wie man mit Hilfe des *Broadcast-Manager* die dynamische Kanaleröffnung des TP1.6-Clients mit dem **LLCF** realisiert.

tp16-server ist ein Programm, dass passiv auf eine von einem `tp16-client` zu initiierende TP1.6 Verbindung wartet und dann Daten empfängt und zurücksendet. Es wird auch gezeigt wie man mit Hilfe des RAW-Sockets die dynamische Kanaleröffnung eines TP1.6-Servers mit dem **LLCF** realisiert.

tp20-client ist ein Programm, dass aktiv eine TP2.0 Verbindung eröffnet und dann Daten sendet und empfängt. Es wird auch gezeigt wie man mit Hilfe des *Broadcast-Manager* die dynamische Kanaleröffnung des TP2.0-Clients mit dem **LLCF** realisiert.

tp20-server ist ein Programm, dass passiv auf eine von einem `tp20-client` zu initiierende TP2.0 Verbindung wartet und dann Daten empfängt und zurücksendet. Es wird auch gezeigt wie man mit Hilfe des RAW-Sockets die dynamische Kanaleröffnung eines TP2.0-Servers mit dem **LLCF** realisiert.

tp20-sniffer ist ein Programm, mit dem man eine TP2.0-Verbindung mitlesen kann (siehe Tracemode-Beschreibung in Kapitel 6.1). Anhand der Kommandozeilen-Parameter kann man dabei konkrete CAN-IDs oder auch logische Gerätenummern zur Bestimmung der mitzulesenden TP-Kanals angeben.

mcnet-sniffer ist ein Programm, mit dem man eine MCNet-Verbindung mitlesen kann (siehe Tracemode-Beschreibung in Kapitel 6.1).

mcnet-vit-emu ist ein Programm, mit dem ein MCNet-TV-Tuner emuliert werden kann. Angeschlossen an den CAN-Bus eines Volkswagen RNS MFD (altes 2DIN-Gerät) wird hier die Kommunikation nachgebildet, die ein TV-Tuner mit dem Bedienteil durchführt, wodurch die Tasteninformationen zur Bedienung des TV-Tuners ausgelesen werden können.

7 Sockets für den Broadcast-Manager

Der *Broadcast-Manager* stellt Funktionen zur Verfügung, um Nachrichten auf dem CAN-Bus einmalig oder periodisch zu senden, sowie um (inhaltliche) Änderungen von (zyklisch) empfangenen CAN-Frames mit einer bestimmten CAN-ID zu erkennen.

Dabei muss der *Broadcast-Manager* folgende Anforderungen erfüllen:

Sendeseitig:

- Zyklisches Senden einer CAN-Botschaft mit einem gegebenen Intervall
- Verändern von Botschaftsinhalten und Intervallen zur Laufzeit (z.B. Umschalten auf neues Intervall mit/ohne sofortigen Neustart des Timers)
- Zählen von Intervallen und automatisches Umschalten auf ein zweites Intervall
- Sofortige Ausgabe von veränderten Botschaften, ohne den Intervallzyklus zu beeinflussen ('Bei Änderung sofort')
- Einmalige Aussendung von CAN-Botschaften

Empfangsseitig:

- Empfangsfilter für die Veränderung relevanter Botschaftsinhalte
- Empfangsfilter ohne Betrachtung des Botschaftsinhalts (CAN-ID-Filter)
- Empfangsfilter für Multiplexbotschaften (z.B. mit Paketzählern im Botschaftsinhalt)
- Empfangsfilter für die Veränderung vom Botschaftslängen
- Beantworten von RTR-Botschaften
- Timeoutüberwachung von Botschaften
- Reduzierung der Häufigkeit von Änderungsnachrichten (Throttle-Funktion)

7.1 Kommunikation mit dem Broadcast-Manager

Im Gegensatz zum RAW-Socket (Kapitel 5) und den Transportprotokoll-Sockets (Kapitel 6) werden über den Socket des *Broadcast-Manager* weder einzelne CAN-Frames noch längere - zu segmentierende - Nutzdaten übertragen.

Der *Broadcast-Manager* ist vielmehr ein programmierbares Werkzeug, dass über besondere Nachrichten vom Anwender gesteuert wird und auch Nachrichten an den Anwender über die Socket-Schnittstelle schicken kann.

Für die Anwendung des *Broadcast-Manager* muss die Include-Datei `bcm.h` eingebunden werden.

Ein Socket zum *Broadcast-Manager* wird durch

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

geöffnet.

Mit dem connect() wird dem Socket das CAN-Interface eindeutig zugewiesen. Möchte ein Prozess auf mehreren CAN-Bussen agieren, muss er folglich mehrere Sockets öffnen. Es ist allerdings auch möglich, dass ein Prozess mehrere Instanzen (Sockets) des *Broadcast-Manager* auf einem CAN-Bus öffnet, wenn dieses für den Anwendungsprogrammierer zur Strukturierung verschiedener Datenströme sinnvoll ist. Jede einzelne Instanz des *Broadcast-Manager* ist in der Lage beliebig viele Filter- und/oder Sendeaufträge zu realisieren.

```
addr.can_family = AF_CAN;
strcpy(ifr.ifr_name, "can0");
ioctl(s, SIOCGIFINDEX, &ifr);
addr1.can_ifindex = ifr.ifr_ifindex;

connect(s, (struct sockaddr *)&addr, sizeof(addr));
```

Alle Nachrichten zwischen dem (Anwender-)Prozess und dem *Broadcast-Manager* besitzen die selbe Struktur. Sie besteht aus einem Nachrichtenkopf mit dem Steuerungskommando und der für diesen Socket/CAN-Bus eindeutigen CAN-ID:

```
struct bcm_msg_head {
    int opcode;                  /* command */
    int flags;                   /* special flags */
    int count;                   /* run 'count' times ival1 then ival2 */
    struct timeval ival1, ival2; /* intervals */
    canid_t can_id;              /* 32 Bit SFF/EFF. MSB set at EFF */
    int nframes;                 /* number of following can_frame's */
    struct can_frame frames[0];
};
```

Der Wert `nframes` gibt an, wie viele Nutzdaten-Frames dem Nachrichtenkopf folgen. Die Nutzdaten-Frames beschreiben den eigentlichen Nachrichteninhalt einer CAN-Botschaft:

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

Der `opcode` definiert, um was für eine Nachricht es sich handelt. Nachrichten vom Anwender an den *Broadcast-Manager* steuern die Operationen des *Broadcast-Manager*, Nachrichten vom *Broadcast-Manager* an den Anwender signalisieren bestimmte Änderungen, Timeouts, etc.

Der Sende- und Empfangszweig des *Broadcast-Manager* sind dabei zwei eigenständige Funktionsblöcke.

Für den Sendezweig existieren die Opcodes

TX_SETUP zum Einrichten und Ändern von Sendeaufträgen

TX_DELETE zum Löschen von Sendeaufträgen

TX_READ zum Auslesen des aktuellen Sendeauftrags (zu Debug-Zwecken)

TX_SEND zum einmaligen Senden einer CAN-Botschaft

Für den Empfangszweig existieren die Opcodes

RX_SETUP zum Einrichten und Ändern von Empfangsfiltern

RX_DELETE zum Löschen von Empfangsfiltern

RX_READ zum Auslesen des aktuellen Empfangsfilters (zu Debug-Zwecken)

Als Antwort schickt der *Broadcast-Manager* Nachrichten in der gleichen Form, wie er selbst die Anforderungen erhält. Dabei sendet der *Broadcast-Manager* die Opcodes

TX_STATUS als Antwort auf TX_READ

TX_EXPIRED wenn der Zähler `count` für `ival1` abgelaufen ist (nur bei gesetztem Flag `TX_COUNTEVT`, s.u.)

RX_STATUS als Antwort auf RX_READ

RX_TIMEOUT wenn der zeitlich überwachte Empfang einer Botschaft ausgeblieben ist

RX_CHANGED wenn die erste bzw. eine geänderte CAN-Nachricht empfangen wurde

Jede dieser durch einen `opcode` bestimmten Funktionen wird eindeutig mit Hilfe der `can_id` referenziert.

Zusätzlich existieren noch optionale `flags`, mit denen der *Broadcast-Manager* in seinem Verhalten beeinflusst werden kann:

SETTIMER : Die Werte `ival1`, `ival2` und `count` werden übernommen

STARTTIMER : Der Timer wird mit den aktuellen Werten von `ival1`, `ival2` und `count` gestartet. Das Starten des Timers führt gleichzeitig zur Aussendung eines `can_frame`'s.

TX_COUNTEVT : Erzeuge die Nachricht TX_EXPIRED, wenn `count` abgelaufen ist

TX_ANNOUNCE : Eine Änderung der Daten durch den Prozess wird zusätzlich unmittelbar ausgesendet. (Anforderung aus 'Bei Änderung Sofort' - BÄS)

TX_CP_CAN_ID : Kopiert die `can_id` aus dem Nachrichtenkopf in jede der angehängten `can_frame`'s. Dieses ist lediglich als Vereinfachung der Benutzung gedacht.

TX_RESET_MULTI_IDX : Erzwingt das Rücksetzen des Index-Zählers beim Update von zu sendenden von Multiplex-Nachrichten auch wenn dieses aufgrund der gleichen Länge nicht nötig wäre. Siehe Seite 29.

RX_FILTER_ID : Es wird keine Filterung der Nutzdaten ausgeführt. Eine Übereinstimmung mit der empfangenen `can_id` führt automatisch zu einer Nachricht RX_CHANGED. **Vorsicht also bei zyklischen Nachrichten!** Bei gesetztem RX_FILTER_ID-Flag *kann* auf das CAN-Frame beim RX_SETUP verzichtet werden (also `nframes=0`).

RX_RTR_FRAME : Die im Filter übergebene CAN-Nachricht wird beim Empfang eines RTR-Frames ausgesendet. Siehe Seite 32.

RX_CHECK_DLC : Eine Änderung des DLC führt zu einem RX_CHANGED.

RX_NO_AUTOTIMER : Ist der Timer ival1 beim RX_SETUP ungleich Null gesetzt worden, wird beim Empfang der CAN-Nachricht automatisch der Timer für die Timeout-Überwachung gestartet. Das Setzen dieses Flags unterbindet das automatische Starten des Timers.

RX_ANNOUNCE_RESUME : Bezieht sich ebenfalls auf die Timeout-Überwachung der Funktion RX_SETUP. Ist der Fall des RX-Timeouts eingetreten, kann durch Setzen dieses Flags ein RX_CHANGED erzwungen werden, wenn der (zyklische) Empfang wieder einsetzt. Dieses gilt besonders auch dann, wenn sich die Nutzdaten nicht geändert haben.

7.2 TX_SETUP

Mit TX_SETUP wird für eine bestimmte CAN-ID ein (zyklischer) Sendeauftrag eingerichtet oder geändert.

Typischerweise wird dabei eine Variable angelegt, bei der die Komponenten `can_id`, `flags` (`SETTIMER`,`STARTTIMER`), `count`=0, `ival2`=100ms, `nframes`=1 gesetzt werden und die Nutzdaten in der Struktur `can_frame` entsprechend eingetragen werden. Diese Variable wird dann im Stück(!) mit einem write()-Systemcall auf dem Socket an den *Broadcast-Manager* übertragen. Beispiel:

```
struct {
  struct bcm_msg_head msg_head;
  struct can_frame frame[4]; /* just an example */
} msg;
```

```
msg.msg_head.opcode  = TX_SETUP;
msg.msg_head.can_id  = 0x42;
msg.msg_head.flags   = SETTIMER|STARTTIMER|TX_CP_CAN_ID;
msg.msg_head.nframes = 1;
msg.msg_head.count = 0;
msg.msg_head.ival1.tv_sec = 0;
msg.msg_head.ival1.tv_usec = 0;
msg.msg_head.ival2.tv_sec = 0;
msg.msg_head.ival2.tv_usec = 100000;
msg.frame[0].can_id    = 0x42; /* obsolete when using TX_CP_CAN_ID */
msg.frame[0].can_dlc   = 3;
msg.frame[0].data[0]   = 0x123;
msg.frame[0].data[1]   = 0x312;
msg.frame[0].data[2]   = 0x231;

write(s, &msg, sizeof(msg));
```

Die Nachrichtenlänge für den Befehl TX_SETUP ist also {`[bcm_msg_head] [can_frame]+`} d.h. ein Nachrichtenkopf und mindestens ein CAN-Frame.

7.2.1 Besonderheiten des Timers

Der Timer kann durch Setzen des Intervalls auf 0 ms (`ival1` und `ival2`) gestoppt werden. Dabei wird die o.g. Variable wieder mit dem gesetzten Flag SETTIMER an den *Broadcast-Manager* übertragen. Um eine zyklische Aussendung mit den übergebenen Timerwerten zu starten, müssen also die Flags `SETTIMER` und `STARTTIMER` im Element `flags` gesetzt sein.

Als Ergänzung zum obigen Beispiel kann auch mit zwei Intervallen für die zyklische Aussendung der CAN-Botschaft gearbeitet werden. Dabei wird die CAN-Botschaft zunächst `count` mal im Intervall `ival1` gesendet und danach bis zur expliziten Löschung durch TX_DELETE oder durch Stoppen des Timers im Intervall `ival2`. Das Intervall `ival2` darf auch Null sein, in welchem Fall die Aussendung nach den ersten `count` Aussendungen stoppt. Falls `count` Null ist, spielt der Wert von `ival1` keine Rolle und muss nicht angegeben zu werden.

Ist das Flag `STARTTIMER` gesetzt, wird unmittelbar die erste CAN-Botschaft ausgesendet.

Ist es für den Anwender wichtig zu erfahren, wann der *Broadcast-Manager* vom Intervall `ival1` auf `ival2` umschaltet (und somit u.U. die Aussendung einstellt), kann dieses dem *Broadcast-Manager* durch das Flag `TX_COUNTEVT` angezeigt werden. Ist der Wert von `count` auf Null heruntergezählt und das Flag `TX_COUNTEVT` gesetzt worden, erzeugt der *Broadcast-Manager* eine Nachricht mit dem Opcode TX_EXPIRED an den Prozess. Diese Nachricht besteht nur aus einem Nachrichtenkopf (`nframes` = 0).

7.2.2 Veränderung von Daten zur Laufzeit

Zur Laufzeit können auch die Daten in der CAN-Botschaft geändert werden. Dazu werden die Daten in der Variable geändert und mit dem Opcode TX_SETUP an den *Broadcast-Manager* übertragen. Dabei kann es folgende Sonderfälle geben:

1. Der Zyklus soll neu gestartet werden: Flag `STARTTIMER` setzen
2. Der Zyklus soll beibehalten werden aber die geänderten/beigefügten Daten sollen sofort einmal gesendet werden: Flag `TX_ANNOUNCE` setzen
3. Der Zyklus soll beibehalten werden und die geänderten Daten erst mit dem nächsten Mal gesendet werden: default Verhalten

Hinweis: Beim Neustarten des Zyklus werden die zuletzt gesetzten Timerwerte (`ival1`, `ival2`) zugrunde gelegt, die vom *Broadcast-Manager* nicht modifiziert werden. Sollte aber mit zwei Timern gearbeitet werden, wird der Wert `count` zur Laufzeit vom *Broadcast-Manager* dekrementiert.

7.2.3 Aussenden verschiedener Nutzdaten (Multiplex-Nachrichen)

Mit dem *Broadcast-Manager* können auch Multiplex-Nachrichten versendet werden. Dieses wird benötigt, wenn z.B. im ersten Byte der Nutzdaten ein Wert definiert, welche Informationen in den folgenden 7 Bytes zu finden sind. Ein anderer Anwendungsfall ist das Umschalten / Toggeln von Dateninhalten. Dazu wird im Prozess eine Variable erzeugt, bei der hinter dem Nachrichtenkopf mehr als ein Nutzdaten-Frame vorhanden ist. Folglich werden an den *Broadcast-Manager* für eine CAN-ID nicht ein sondern mehrere `can_frame`'s übermittelt. Die verschiedenen Nutzdaten werden nacheinander im Zyklus der Aussendung ausgegeben. D.h. bei zwei `can_frame`'s werden diese abwechselnd im gewünschten Intervall gesendet. Bei einer Änderung der Daten zur Laufzeit, wird mit der Aussendung des ersten `can_frame` neu begonnen, wenn sich die Anzahl der zu sendenden `can_frame`'s beim Update verändert (also $\text{nframes}_{neu} \neq \text{nframes}_{alt}$). Bei einer gleichbleibenden Anzahl zu sendender `can_frame`'s kann dieses Rücksetzen des ansonsten normal weiterlaufenden Index-Zählers durch Setzen des Flags `TX_RESET_MULTI_IDX` erzwungen werden.

7.3 TX_DELETE

Diese Nachricht löscht den Eintrag zur Aussendung der CAN-Nachricht mit dem in `can_id` angegebenen CAN-Identifier. Die Nachrichtenlänge für den Befehl TX_DELETE ist {`[bcm_msg_head]`} d.h. ein Nachrichtenkopf.

7.4 TX_READ

Mit dieser Nachricht kann der aktuelle Zustand, also die zu sendende CAN-Nachricht, Zähler, Timer-Werte, etc. zu dem in `can_id` angegebenen CAN-Identifier ausgelesen werden. Der *Broadcast-Manager* antwortet mit einer Nachricht mit dem `opcode` TX_STATUS, die das entsprechende Element enthält. Diese Antwort kann je nach Länge der Daten beim zugehörigen TX_SETUP unterschiedlich lang sein. Die Nachrichtenlänge für den Befehl TX_READ ist {`[bcm_msg_head]`} d.h. ein Nachrichtenkopf.

7.5 TX_SEND

Zum einmaligen Senden einer CAN-Nachricht, ohne eine besondere Funktionalität des *Broadcast-Manager* zu nutzen, kann der `opcode` TX_SEND genutzt werden. Dabei wird eine Variable erzeugt, in der die Komponenten `can_id`, `can_dlc`, `data[]` mit den entsprechenden Werten gefüllt werden. Der *Broadcast-Manager* sendet diese CAN-Botschaft unmittelbar auf dem durch den Socket definierten CAN-Bus. Die Nachrichtenlänge für den Befehl TX_SEND ist {`[bcm_msg_head] [can_frame]`} d.h. ein Nachrichtenkopf und genau ein CAN-Frame.

Anmerkung: Selbstverständlich können einzelne CAN-Botschaften auch mit dem RAW-Socket versendet werden. Allerdings muss man dazu einen RAW-Socket öffnen, was für eine einzelne CAN-Botschaft bei einem bereits geöffneten **BCM**-Socket ein unverhältnismäßig großer Programmieraufwand wäre.

7.6 RX_SETUP

Mit RX_SETUP wird für eine bestimmte CAN-ID ein Empfangsauftrag eingerichtet oder geändert. Der *Broadcast-Manager* kann bei der Filterung von CAN-Nachrichten dieser CAN-ID nach verschiedenen Kriterien arbeiten und bei Änderungen und/oder Timeouts eine entsprechende Nachricht an den Prozess senden.

Analog zum `opcode` TX_SETUP (siehe Seite 27) wird auch hier typischerweise eine Variable angelegt die der Nachrichtenstruktur des *Broadcast-Manager* entspricht. Die Nachrichtenlänge für den Befehl RX_SETUP ist {`[bcm_msg_head] [can_frame]+`} d.h. ein Nachrichtenkopf und

mindestens ein CAN-Frame.

Im Unterschied zu TX_SETUP haben die Komponenten der Struktur im Rahmen der Empfangsfunktionalität zum Teil andere Bedeutungen, wenn sie vom Prozess an den *Broadcast-Manager* geschickt werden:

count keine Funktion

ival1 Timeout für CAN-Nachrichtenempfang

ival2 Drosselung von RX_CHANGED Nachrichten

can_data enthält eine Maske zum Filtern von Nutzdaten

7.6.1 Timeoutüberwachung

Wird vom *Broadcast-Manager* eine CAN-Nachricht für einen längeren Zeitraum als `ival1` nicht vom CAN-Bus empfangen, wird eine Nachricht mit dem `opcode` RX_TIMEOUT an den Prozess gesendet. Diese Nachricht besteht nur aus einem Nachrichtenkopf (`nframes` = Null). Eine Timeoutüberwachung wird in diesem Fall nicht neu gestartet.

Typischerweise wird die Timeroutüberwachung mit dem Empfang einer CAN-Botschaft gestartet. Mit Setzen des Flags `STARTTIMER` kann aber auch sofort beim RX_SETUP mit dem Timeout begonnen werden. Das Setzen des Flags `RX_NO_AUTOTIMER` unterbindet das automatische Starten der Timeoutüberwachung beim Empfang einer CAN-Nachricht.

Hintergrund: Das automatische Starten der Timeoutüberwachung beim Empfang einer Nachricht macht jeden auftretenden zyklischen Ausfall einer CAN-Nachricht deutlich, ohne dass der Anwender aktiv werden muss.

Um ein Wiedereinsetzen des Zyklus' bei gleich bleibenden Nutzdaten sicher zu erkennen kann das Flag `RX_ANNOUNCE_RESUME` gesetzt werden.

7.6.2 Drosselung von RX_CHANGED Nachrichten

Auch bei einer aktivierten Filterung von Nutzdaten kann die Benutzerapplikation bei der Bearbeitung von RX_CHANGED Nachrichten überfordert sein, wenn sich die Daten schnell ändern (z.B. Drehzahl).

Dazu kann der Timer `ival2` gesetzt werden, der den minimalen Zeitraum beschreibt, in der aufeinanderfolgende RX_CHANGED Nachrichten für die jeweilige `can_id` vom *Broadcast-Manager* gesendet werden dürfen.

Hinweis: Werden innerhalb der gesperrten Zeit weitere geänderte CAN-Nachrichten empfangen, wird die letzt gültige nach Ablauf der Sperrzeit mit einem RX_CHANGED übertragen. Dabei können zwischenzeitliche (z.B. alternierende) Zustandsübergänge verloren gehen.

Hinweis zu MUX-Nachrichten: Nach Ablauf der Sperrzeit werden alle aufgetretenen RX_CHANGED Nachrichten hintereinander an den Prozess gesendet. D.h. für jeden MUX-Eintrag wird eine evtl. eingetretene Änderung angezeigt.

7.6.3 Nachrichtenfilterung (Nutzdaten - simple)

Analog der Übertragung der Nutzdaten bei TX_SETUP (siehe Seite 27) wird bei RX_SETUP eine Maske zur Filterung der eintreffenden Nutzdaten an den *Broadcast-Manager* übergeben. Dabei wird vom *Broadcast-Manager* zur Nachrichtenfilterung zunächst nur der Nutzdatenteil (`data[]`) der

Struktur `can_frame` ausgewertet.

Ein gesetztes Bit in der Maske bedeutet dabei, das dieses entsprechende Bit in der CAN-Nachricht auf eine Veränderung hin überwacht wird.

Wenn in einer empfangenen CAN-Nachrichten eine Änderungen gegenüber der letzten empfangenen Nachricht in einem der durch die Maske spezifizierten Bits eintritt, wird die Nachricht RX_CHANGED mit dem empfangenen CAN-Frame an den Prozess gesendet.
Beim ersten Empfang einer Nachricht, wird das empfangene CAN-Frame grundsätzlich an den Prozess gesendet - erst danach kann schließlich auf eine *Änderung* geprüft werden. Tipp: Das Setzen der Filtermaske auf Null bewirkt somit das einmalige Empfangen einer sonst z.B. zyklischen Nachricht.

7.6.4 Nachrichtenfilterung (Nutzdaten - Multiplex)

Werden auf einer CAN-ID verschiedene, sich zyklisch wiederholende Inhalte übertragen, spricht man von einer Multiplex-Nachricht. Dazu wird beispielsweise im ersten Byte der Nutzdaten des CAN-Frames ein MUX-Identifier eingetragen, der dann die folgenden Bytes in ihrer Bedeutung definiert. Bsp.: Das erste Byte (Byte 0) hat den Wert `0x02` ⇒ in den Bytes 1-7 ist die Zahl der zurückgelegten Kilometer eingetragen. Das erste Byte (Byte 0) hat den Wert `0x04` ⇒ in den Bytes 1-7 ist die Zahl der geleisteten Betriebsstunden eingetragen. Usw.

Solche Multiplex-Nachrichten können mit dem *Broadcast-Manager* gesendet werden, wenn für das Aussenden über eine CAN-ID mehr als ein Nutzdatenframe `can_frame` an den *Broadcast-Manager* gesendet werden (siehe Seite 29).

Zur Filterung von Multiplex-Nachrichten werden mindestens zwei (`nframes` ≥ 2) `can_frame`'s an den *Broadcast-Manager* gesendet, wobei im ersten `can_frame` die MUX-Maske enthalten ist und in den folgenden `can_frame`('s) die Nutzdaten-Maske(n), wie oben beschrieben. In die Nutzdaten-Masken sind an den Stellen, die die MUX-Maske definiert hat, die MUX-Identifier eingetragen, anhand derer die Nutzdaten unterschieden werden.

Für das obige Beispiel würde also gelten:

Das erste Byte im ersten `can_frame` (der MUX-Maske) wäre `0xFF` - die folgenden 7 Bytes wären `0x00` - damit ist die MUX-Maske definiert. Die beiden folgenden `can_frame`'s enthalten wenigstens in den jeweils ersten Bytes die `0x02` bzw. `0x04` wodurch die MUX-Identifier der Multiplex-Nachrichten definiert sind. Zusätzlich können (sinnvollerweise) in den Nutzdatenmasken noch weitere Bits gesetzt sein, mit denen z.B. eine Änderung der Betriebsstundenzahl überwacht wird.

Eine Änderung einer Multiplex-Nachricht mit einem bestimmten MUX-Identifier führt zu einer Nachricht RX_CHANGED mit genau dem einen empfangenen CAN-Frame an den Prozess. D.h. der Prozess muss anhand des MUX-Identifiers die vom *Broadcast-Manager* empfangene Nachricht bewerten.

Im gezeigten Beispiel (Abbildung 4) ist die MUX-Maske im Byte 0 auf `0x5F` gesetzt. Beim Empfang von RX-Frame 1 wird keine Nachricht an den Anwender geschickt (MUX-Identifier ist nicht bekannt). Bei RX-Frame 2 gibt es eine Nachricht (MUX-Identifier bekannt und relevante Daten haben sich - beim ersten Empfangsvorgang - geändert). Beim Empfang von RX-Frame 3 (Änderungen in den gelb markierten Bits) wird keine Nachricht an den Anwender geschickt, weil sich keine relevanten Daten für den eingetragenen MUX-Identifier geändert haben.

7.6.5 Nachrichtenfilterung (Länge der Nutzdaten - DLC)

Auf Anforderung kann der *Broadcast-Manager* auch zusätzlich eine Veränderung der in den CAN-Nachrichten angegebenen Nutzdatenlänge überwachen. Dazu wird der empfangene Data Length

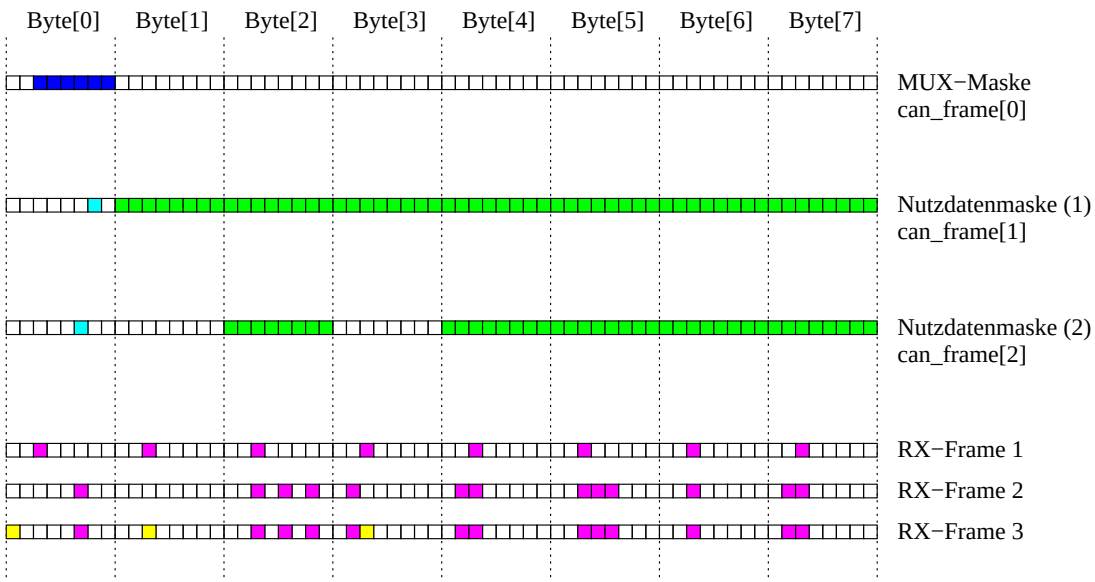

Abbildung 4: Beispiel für die Anwendung des Multiplexfilters

Code (DLC) mit dem zu diesem CAN-Frame passenden, bereits empfangenen DLC verglichen. Ein Unterschied führt wie bei der Filterung der Nutzdaten zu einer Nachricht RX_CHANGED an den Prozess. Zum Aktivieren dieser Funktionalität muss in der Komponente `flags` der Wert `RX_CHECK_DLC` gesetzt sein.

7.6.6 Filterung nach CAN-ID

Im Gegensatz zu den oben beschriebenen Nachrichtenfiltern besteht auch die Möglichkeit nur nach der angegebenen CAN-ID zu filtern. Dazu wird in der Komponente `flags` der Wert `RX_FILTER_ID` gesetzt. Die Komponente `nframes` kann dabei Null sein und so werden folglich auch keine Nutzdaten (`can_frame`'s) an den *Broadcast-Manager* geschickt. Angehängte Nutzdaten (d.h. `nframes` > 0 und entsprechende `can_frame`'s) werden ignoriert. Werden beim RX_SETUP keine `can_frames` übertrags, ist also `nframes` $= 0$, wird im *Broadcast-Manager* automatisch das Flag `RX_FILTER_ID` gesetzt.

Hinweis: Die Filterung nach CAN-IDs sollte nur bei nicht zyklischen CAN-Nachrichten genutzt werden.

7.6.7 Automatisches Beantworten von RTR-Frames

Grundsätzlich können Remote-Transmission-Requests (RTR) mit dem *Broadcast-Manager* ODER in einer Applikation im Userspace beantwortet werden. Im Userspace würde eine Anwendung über den *Broadcast-Manager*-Socket oder einen RAW-Socket eine CAN-Nachricht empfangen, auf das gesetzte RTR-Bit prüfen und entsprechend eine Antwort senden. Das RX_SETUP könnte in diesem Fall beispielsweise so aussehen:

```
/* normal receiption of RTR-frames in Userspace */
txmsg.msg_head.opcode  = RX_SETUP;
txmsg.msg_head.can_id  = 0x123 | CAN_RTR_FLAG;
txmsg.msg_head.flags   = RX_FILTER_ID;
txmsg.msg_head.ival1.tv_sec = 0;
txmsg.msg_head.ival1.tv_usec = 0;
txmsg.msg_head.ival2.tv_sec = 0;
txmsg.msg_head.ival2.tv_usec = 0;
txmsg.msg_head.nframes = 0;
```

```
if (write(s, &txmsg, sizeof(txmsg)) < 0)
  perror("write");
```

Diese Aufgabe kann auch der *Broadcast-Manager* übernehmen, indem man beim RX_SETUP statt eines Filters die auszusendende Nachricht angibt und das Flag `RX_RTR_FRAME` setzt:

```
/* specify CAN-Frame to send as reply to a RTR-request */
txmsg.msg_head.opcode  = RX_SETUP;
txmsg.msg_head.can_id  = 0x123 | CAN_RTR_FLAG;
txmsg.msg_head.flags   = RX_RTR_FRAME; /* | TX_CP_CAN_ID */;
txmsg.msg_head.ival1.tv_sec  = 0; /* no timers in RTR-mode */
txmsg.msg_head.ival1.tv_usec = 0;
txmsg.msg_head.ival2.tv_sec  = 0;
txmsg.msg_head.ival2.tv_usec = 0;
txmsg.msg_head.nframes       = 1; /* exact 1 */

/* the frame to send as reply ... */
txmsg.frame.can_id    = 0x123; /* 'should' be the same */
txmsg.frame.can_dlc   = 4;
txmsg.frame.data[0]   = 0x12;
txmsg.frame.data[1]   = 0x34;
txmsg.frame.data[2]   = 0x56;
txmsg.frame.data[3]   = 0x78;

if (write(s, &txmsg, sizeof(txmsg)) < 0)
  perror("write");
```

Beim Empfang einer CAN-Nachricht mit der CAN-ID 0x123 und gesetztem RTR-Bit wird das `can_frame txmsg.frame` ausgesendet. Bei gesetztem Flag `TX_CP_CAN_ID` wird die Zeile mit `txmsg.frame.can_id` obsolet. Der Wert `txmsg.frame.can_id` ist nicht beschränkt, d.h. der *Broadcast-Manager* könnte auf ein RTR-Frame mit der CAN-ID 0x123 auch mit einer CAN-Nachricht mit einer anderen CAN-ID (z.B. 0x42) antworten. Achtung Denksportaufgabe: Bei gleicher CAN-ID und einem gesetzten RTR-Flag im `can_frame txmsg.frame` erfolgt ein Vollast-Test. Aus diesem Grunde wird bei Gleichheit von `txmsg.msg_head.can_id` und `txmsg.frame.can_id` (z.B. bei Anwendung der Option `TX_CP_CAN_ID`) das RTR-Flag in `txmsg.frame.can_id` beim RX_SETUP automatisch gelöscht.
Die bei einem RTR-Frame auszusendende Nachricht kann durch ein erneutes RX_SETUP mit der identischen CAN-ID (mit gesetztem Flag `RX_RTR_FRAME`) jederzeit aktualisiert werden. Die Nachrichtenlänge für den Befehl RX_SETUP mit gesetztem Flag `RX_RTR_FRAME` ist {`[bcm_msg_head] [can_frame]`} d.h. ein Nachrichtenkopf und genau ein CAN-Frame.

7.7 RX_DELETE

Mit RX_DELETE wird für eine bestimmte CAN-ID ein Empfangsauftrag gelöscht. Die angegebene CAN-ID wird vom *Broadcast-Manager* nicht mehr vom CAN-Bus empfangen. Die Nachrichtenlänge für den Befehl RX_DELETE ist {`[bcm_msg_head]`} d.h. ein Nachrichtenkopf.

7.8 RX_READ

Mit RX_READ kann der aktuelle Zustand des Filters für CAN-Frames mit der angegebenen CAN-ID ausgelesen werden. Der Broadcast-Manager antwortet mit der Nachricht RX_STATUS an den Prozess. Diese Antwort kann je nach Länge der Daten beim zugehörigen RX_SETUP unterschiedlich lang sein. Die Nachrichtenlänge für den Befehl RX_READ ist {`[bcm_msg_head]`} d.h. ein Nachrichtenkopf.

7.9 Weitere Anmerkungen zum Broadcast-Manager

- Die Nachrichten TX_EXPIRED, RX_TIMEOUT vom *Broadcast-Manager* an den Prozess enthalten keine Nutzdaten (`nframes` = 0)

- Die Nachrichten TX_STATUS, RX_STATUS vom *Broadcast-Manager* an den Prozess enthalten genau so viele Nutzdaten, wie vom Prozess bei der Einrichtung des Sende-/Empfangsauftrags mit TX_SETUP bzw. RX_SETUP an den *Broadcast-Manager* geschickt wurden.
- Die Nachricht RX_CHANGED vom *Broadcast-Manager* an den Prozess enthält genau das vom CAN empfangene, geänderte Nutzdaten-Frame (`nframes` = 1)
- Beim Ändern von zu sendenden Multiplex-Nachrichten (TX_SETUP) müssen immer alle Nutzdaten-Frames übertragen werden. Es wird generell mit der Aussendung der ersten MUX-Nachricht begonnen.
- Die Komponente `can_id` in der Struktur `bcm_msg_head` kann *sendeseitig* auch als 'Handle' betrachtet werden, weil bei der Aussendung von CAN-Nachrichten die beim TX_SETUP mit übertragenen `can_frame`'s gesendet werden. Das Setzen jeder einzelnen `can_id` in den `can_frame`'s kann durch das Flag `TX_CP_CAN_ID` vereinfacht werden.
- Beim Auslesen der Sende-/Empfangsaufträge mit TX_READ bzw. RX_READ können folgende Werte in den Antworten TX_STATUS bzw. RX_STATUS von der ursprünglich gesendeten Nachricht abweichen:

  count Entspricht dem aktuellen Wert

  SETTIMER Wurde ausgeführt und damit konsumiert

  STARTTIMER Wurde ausgeführt und damit konsumiert

  TX_ANNOUNCE Wurde ausgeführt und damit konsumiert
- Das Schließen des **BCM**-Sockets mit close(2) bzw. das Terminieren des Anwenderprozesses löscht alle Konfigurationseinträge der zugehörigen **BCM**-Instanz. Zyklische Aussendungen dieser **BCM**-Instanz werden folglich sofort beendet.

7.10 Testprogramme

tst-bcm-single führt eine einzelne TX_SEND-Operation aus.

tst-bcm-cycle Zyklisches Aussenden einer CAN-Botschaft mit TX_SETUP und beenden der zyklischen Aussendung mit TX_SETUP (ohne TX_DELETE).

tst-bcm-tx_read Funktionsprüfung der Debug-Möglichkeit mit TX_READ.

tst-bcm-rtr Beispiel für die Anwendung des Flags `RX_RTR_FRAME`.

tst-bcm-filter diverse Filtertests inklusive Multiplex-Filter.

tst-bcm-throttle Funktionsprüfung der Throttle-Funktionalität (Update-Bremse).

can-sniffer ist ein Programm zur Beobachtung dynamischer Dateninhalte in zyklischen CAN-Nachrichten. Änderungen können in hexadezimaler, binärer oder in ASCII-Darstellung farblich hervorgehoben werden. Filter können zur Laufzeit verändert und gespeichert bzw. geladen werden.Wird `can-sniffer` ohne Parameter aufgerufen, erscheint ein Hilfetext.

8 LLCF-Status im /proc-Filesystem

Das *Low Level CAN Framework* unterstützt das /proc-Filesystem und stellt darüber Statistiken und Informationen über interne Strukturen und Stati in lesbarer Form zur Verfügung. Die Informationen können vom Benutzer beispielsweise mit

```
cat /proc/sys/net/can/stats
```

abgefragt werden. Im Folgenden werden die einzelnen Einträge erläutert.

8.1 Versionsinformation /proc/sys/net/can/version

Die **LLCF**-Versionsinformationen können für eine Anwendung z.B. durch das Öffnen der Datei `/proc/sys/net/can/version` ausgelesen werden. Dazu werden die ersten 6 Zeichen in einen Puffer kopiert und mit `llcf_version_code = strtoul(mybuffer, (char **)NULL, 16);` in den LLCF_VERSION_CODE überführt. Der LLCF_VERSION_CODE wird nach der Regel

```
LLCF_VERSION_CODE = (((MAJORVERSION) << 16) + ((MINORVERSION) << 8) + (PATCHLEVEL))
```

berechnet.

```
hartko@pplinux1:~> cat /proc/sys/net/can/version
010000 [ Volkswagen AG - Low Level CAN Framework (LLCF) v1.0.0-rc1 ]
```

8.2 Statistiken /proc/sys/net/can/stats

Über die angebotenen Statistiken kann man sich über das aktuelle Datenaufkommen informieren und wie beispielsweise der Anteil der von Applikationen benötigten (matched) CAN-Frames im Verhältnis aller vom CAN-Bus empfangener CAN-Frames ist.

Die Informationen werden mit dem Start des **LLCF** jede Sekunde aktualisiert.

```
hartko@pplinux1:~> cat /proc/sys/net/can/stats

      811 transmitted frames (TXF)
   319427 received frames (RXF)
    69504 matched frames (RXMF)

       21 % total match ratio (RXMR)
        0 frames/s total tx rate (TXR)
        0 frames/s total rx rate (RXR)

      100 % current match ratio (CRXMR)
        2 frames/s current tx rate (CTXR)
      166 frames/s current rx rate (CRXR)

      100 % max match ratio (MRXMR)
        2 frames/s max tx rate (MTXR)
      167 frames/s max rx rate (MRXR)

        6 current receive list entries (CRCV)
        6 maximum receive list entries (MRCV)
```

8.3 Zurücksetzen von Statistiken /proc/sys/net/can/reset_stats

Das Zurücksetzen der statistischen Informationen kann durch interne Überläufe von Zählern oder vom Benutzer selbst initiiert werden. Über das Zurücksetzen der statistischen Informationen informiert eine zusätzliche Zeile (STR). An diesem Beispiel sind die Auswirkungen in einem laufenden System bezüglich der obigen Ausgabe der Statistiken gut zu erkennen.

```
hartko@pplinux1:~> cat /proc/sys/net/can/reset_stats
LLCF statistic reset #1 done.
hartko@pplinux1:~> cat /proc/sys/net/can/stats

       31 transmitted frames (TXF)
     2585 received frames (RXF)
     2585 matched frames (RXMF)

      100 % total match ratio (RXMR)
        1 frames/s total tx rate (TXR)
      165 frames/s total rx rate (RXR)

      100 % current match ratio (CRXMR)
        2 frames/s current tx rate (CTXR)
      165 frames/s current rx rate (CRXR)

      100 % max match ratio (MRXMR)
        2 frames/s max tx rate (MTXR)
      167 frames/s max rx rate (MRXR)

        6 current receive list entries (CRCV)
        6 maximum receive list entries (MRCV)

        1 statistic resets (STR)
```

8.4 Interne Empfangslisten des RX-Dispatchers

LLCF-Module können sich beim **LLCF**-RX-Dispatcher für den Empfang von einzelnen CAN-IDs (oder Bereichen von CAN-IDs) von bestimmten CAN-Netzwerk-Interfaces registrieren. Diese Registrierung führt zu einem Eintrag in einer zugehörigen Empfangsliste, bei der zu jeder registrierten CAN-ID eine Funktion mit einem Parameter (z.B. eine modulspezifische Referenz wie 'userdata' oder 'sk') aufgerufen wird, wenn das entsprechende CAN-Frame empfangen wurde. In der Spalte 'ident' trägt sich das registrierende Protokoll-Modul namentlich ein. Zusammen mit der Debug-Funktionalität (siehe Kapitel 3.3.4) kann man anhand dieser Informationen die Funktionsweise des **LLCF** einfach nachvollziehen.

Zur schnellen Verarbeitung empfangener CAN-Frames sind im RX-Dispatcher des **LLCF** verschiedenartige Empfangslisten (für jedes CAN-Netzwerk-Interface) realisiert:

rcvlist_all In dieser Liste sind Registrierungen eingetragen, die von einem CAN-Bus alle empfangenen CAN-Frames benötigen. Typischerweise sind dieses die so genannten RAW-Sockets ohne aktiven Filter (siehe Kapitel 5).

rcvlist_fil In dieser Liste sind Registrierungen eingetragen, die nur einen über Bitmasken definierten Bereich von CAN-Frames benötigen (z.B. 0x200 - 0x2FF).

rcvlist_inv In dieser Liste sind Registrierungen eingetragen, die einen über Bitmasken definierten Bereich von CAN-Frames ausblenden wollen - also die Umkehrung von 'rcvlist_fil'.

rcvlist_eff In dieser Liste sind Registrierungen für einzelne CAN-Frames im Extended Frame Format (29 Bit Identifier) eingetragen.

rcvlist_sff In dieser Liste sind Registrierungen für einzelne CAN-Frames im Standard Frame Format (11 Bit Identifier) eingetragen.

Durch das Aufteilen der Empfangslisten, wird der Aufwand zum Suchen und Vergleichen des empfangenen CAN-Frames mit den registrierten Empfangsfiltern minimiert.

So wurde z.B. die Liste 'rcvlist_sff' als Array mit 2048 Einträgen (11 Bit CAN-ID) mit einer einfach verketteten Liste für die jeweiligen CAN-IDs realisiert. Auf eine Überprüfung von Filtern und Inhalten kann hier beim Empfang einer passenden Nachricht verzichtet werden.

Derzeit wird die Funktionalität der Extended CAN-Frames in aktuellen Projekten nicht genutzt, weshalb die Registrierungen für einzelne EFF-Frames in eine einfach verkettete Liste eingetragen werden. Bei intensiverer Nutzung der Extended CAN-Frames sollte man als 'rcvlist_eff' eine Hash-Tabelle (analog zur 'rcvlist_sff') im **LLCF** realisieren, um eine effiziente Verarbeitung zu gewährleisten.

Möchte man alle Empfangslisten auf einmal ansehen, kann man zur Vereinfachung folgendes eingeben:

```
hartko@pplinux1:~> cat /proc/sys/net/can/rcvlist_*

receive list 'rx_all':
  device   can_id   can_mask  function  userdata   matches  ident
   can1      000    00000000  f8c995ac  f0e59280     42726  raw
  device   can_id   can_mask  function  userdata   matches  ident
   can0      000    00000000  f8c995ac  f0e59800     55240  raw


receive list 'rx_eff':
  (can1: no entry)
  (can0: no entry)


receive list 'rx_fil':
  device   can_id   can_mask  function  userdata   matches  ident
   can1      200    00000700  f8c995ac  f0e5b380         0  raw
  (can0: no entry)


receive list 'rx_inv':
  (can1: no entry)
  (can0: no entry)


receive list 'rx_sff':
  (can1: no entry)
  device   can_id   can_mask  function  userdata   matches  ident
   can0      123    000007ff  f8c86bec  e2e14380        29  bcm
   can0      456    000007ff  f8c86bec  ea954880         0  bcm
   can0      789    000007ff  f8c86bec  e30e6200       130  bcm
   can0      3FF    000007ff  f8c86bec  deaf2580        14  bcm
   can0      740    000007ff  f8c93680  e48322c4       178  tp20
```

Es geht natürlich auch so:

```
hartko@pplinux1:~> cat /proc/sys/net/can/rcvlist_sff

receive list 'rx_sff':
  (can1: no entry)
  device   can_id   can_mask  function  userdata   matches  ident
   can0      123    000007ff  f8c86bec  e2e14380        29  bcm
   can0      456    000007ff  f8c86bec  ea954880         0  bcm
   can0      789    000007ff  f8c86bec  e30e6200       130  bcm
   can0      3FF    000007ff  f8c86bec  deaf2580        14  bcm
   can0      740    000007ff  f8c93680  e48322c4       178  tp20
```

8.5 CAN Network-Devices im Verzeichnis /proc/net/drivers

In dieses Verzeichnis sollen sich CAN-Netzwerk-Treiber mit ihren Proc-Filesystem-Einträgen registrieren. Derzeit ist dieses nur für den mitgelieferten SJA1000-Treiber auf `src/drivers/sja1000` realisiert, dessen Ausgaben kurz beschrieben werden.

Im Beispiel wird der SJA1000-Netzwerk-Treiber auf dem iGate (Jaybrain GW2) gezeigt. Beim ISA-Treiber (sja1000-isa) sind die ausgelesenen Informationen analog.

8.5.1 Treiberstatus /proc/net/drivers/sja1000-xxx

Hier wird der Zustand der CAN-Controller und des jeweils zugehörigen CAN-Busses angezeigt (siehe dazu die Dokumentation zum Philips SJA1000).

```
hartko@pplinux1:~> cat /proc/net/drivers/sja1000-gw2
CAN bus device statistics:
       errwarn  overrun   wakeup   buserr   errpass  arbitr   restarts clock        baud
can0:        0        0        0        0        0        0        0   20000000      500
can0: bus status: OK, RXERR: 0, TXERR: 0
can1:        0        0        0        0        0        0        0   20000000      100
can1: bus status: OK, RXERR: 0, TXERR: 0
can2:        0        0        0        0        0        0        0   20000000      100
can2: bus status: OK, RXERR: 0, TXERR: 0
can3:        0        0        0        0        0        0        0   20000000      500
can3: bus status: OK, RXERR: 0, TXERR: 0
```

8.5.2 Registeranzeige /proc/net/drivers/sja1000-xxx_regs

Hier werden die 32 Register der SJA1000-CAN-Controller angezeigt (siehe dazu die Dokumentation zum Philips SJA1000).

```
hartko@pplinux1:~> cat /proc/net/drivers/sja1000-gw2_regs
SJA1000 registers:
can0 SJA1000 registers:
00: 02 00 0c 00 05 00 40 4d 1a 1a 00 00 00 60 00 00
10: 65 ef d3 5f a2 08 01 05 fa ff 0e 7f 0c 00 00 cf
can1 SJA1000 registers:
00: 02 00 0c 00 05 00 43 ff 1a 1a 00 00 00 60 00 00
10: 61 ff de 3d 80 00 10 45 d7 ef fb 4a 06 00 00 cf
can2 SJA1000 registers:
00: 02 00 0c 00 05 00 43 ff 1a 1a 00 00 00 60 00 00
10: 61 fb ee 87 a0 4a 80 10 76 ff da bd 00 00 00 cf
can3 SJA1000 registers:
00: 02 00 0c 00 05 00 40 4d 1a 1a 00 00 00 60 00 00
10: 61 ef 7f ff 21 1c 42 08 32 df 57 6f a1 00 00 cf
```

8.5.3 Zurücksetzen des Treibers /proc/net/drivers/sja1000-xxx_reset

Das Lesen dieses Eintrages führt einen Reset der SJA1000-CAN-Controller durch. Wie man im Beispiel sieht, ist die Anzahl der 'restarts' danach um 1 erhöht.

```
hartko@pplinux1:~> cat /proc/net/drivers/sja1000-gw2_reset
resetting can0 can1 can2 can3 done
hartko@pplinux1:~> cat /proc/net/drivers/sja1000-gw2
CAN bus device statistics:
       errwarn  overrun   wakeup   buserr   errpass  arbitr   restarts clock        baud
can0:        0        0        0        0        0        0        1   20000000      500
can0: bus status: OK, RXERR: 0, TXERR: 0
can1:        0        0        0        0        0        0        1   20000000      100
can1: bus status: OK, RXERR: 0, TXERR: 0
can2:        0        0        0        0        0        0        1   20000000      100
can2: bus status: OK, RXERR: 0, TXERR: 0
can3:        0        0        0        0        0        0        1   20000000      500
can3: bus status: OK, RXERR: 0, TXERR: 0
```

8.5.4 Testprogramme

tst-proc Öffnet bis zu 800 RAW-Sockets, um einen Überlauf bei der Ausgabe von `/proc/net/can/rcvlist_all` zu provozieren.

9 Unterstützte CAN-Hardware

Bisherige Realisierungen von CAN-Treibern unter Linux und auch anderen Betriebssystemen sind nach dem Zeichen-Treibermodell (dem so genannten Character-Device) ausgeführt.

Im Unterschied dazu setzt das *Low Level CAN Framework* auf CAN-Treiber nach dem Netzwerk-Treibermodell auf (so genannte Network-Devices), die es ermöglichen, dass mehrere Anwendungen gleichzeitig auf einem CAN-Bus arbeiten können.

Wenngleich ein Treiber nach dem Netzwerk-Treibermodell einfacher zu realisieren ist, sind die bei einer kommerziellen CAN-Hardware beigelegten Treiber für das **LLCF** so nicht einsetzbar. Ist der Quellcode des beigelegten Treibers verfügbar, kann man diesen allerdings so modifizieren, dass er sich nicht als Character-Device sondern als Network-Device im Linux-Kernel registriert und entsprechend andere Schnittstellen des Kernel bedient.

Der unter `src/drivers/sja1000` realisierte Philips-SJA1000-Treiber ist eine komplette Neuentwicklung und kann als Beispiel für einen CAN-Netzwerktreiber genommen werden. Derzeit unterstützt das **LLCF** ausschließlich passive CAN-Karten, weil dieses für den Linux-Kernel im Gegensatz zu anderen Betriebssystemen problemlos möglich ist. Ein modifizierter Treiber für eine aktive PCMCIA-Karte ist in Arbeit.

Derzeit werden folgende CAN-Hardware-Komponenten unterstützt:

9.1 PC104 / ISA / plain access

In diesen Karten liegen die SJA1000-Controller linear im Adressraum.
Z.B. http://www.peak-system.com/db/de/pcanpc104.html

Mögliche Treiber:

- **LLCF**-SJA1000-Treiber in `src/drivers/sja1000` (empfohlen)
- Modifizierter Linux-Treiber v2.15 von PEAK-System (auf Anfrage)

9.2 PCI

In diesen Karten liegen die SJA1000-Controller linear im PCI-Adressraum.
Z.B. http://www.peak-system.com/db/de/pcanpci.html

Mögliche Treiber:

- Modifizierter Linux-Treiber v2.15 von PEAK-System (auf Anfrage)

9.3 Parallelport

Benötigt Linux-Parport-Unterstützung.
Z.B. http://www.peak-system.com/db/de/pcandongle.html

Mögliche Treiber:

- Modifizierter Linux-Treiber v2.15 von PEAK-System (auf Anfrage)

9.4 USB

USB-CAN-Adapter.
http://www.peak-system.com/db/de/pcanusb.html

Mögliche Treiber:

- Modifizierter Linux-Treiber v2.15 von PEAK-System

9.5 PCMCIA

Passive PCMCIA-Karte mit zwei SJA1000-Controllern.
http://www.ems-wuensche.com/catalog/english/datasheet/htm/cpccard_e.htm

Mögliche Treiber:

- Modifizierter Linux-Treiber cdkl-1.12 von EMS Wünsche (auf Anfrage)

Aktive PCMCIA-Karte mit zwei SJA1000-Controllern.
http://www.kvaser.com/prod/hardware/lapcan_i.htm
http://www.kvaser.com/prod/hardware/lapcan_ii.htm

Mögliche Treiber (in Arbeit!):

- Modifizierter Linux-Treiber v4.1beta von Kvaser

9.6 Virtual CAN Bus (vcan)

Der virtuelle CAN-Bus-Treiber realisiert ein logisches CAN-Network-Device, über das Anwendungen auf einem System ohne real vorhandene CAN-Hardware kommunizieren können. Die Idee entspricht einem Loopback-Device, wobei die Loopback-Funktionalität (siehe Kapitel 1) bereits im **LLCF**-Rahmen realisiert ist. Der vcan-Treiber ist Bestandteil des *Low Level CAN Framework*.

10 Ansprechpartner

Dieses Dokument sollte zusammen mit den Test-Programmen in `src/test` einen umfassenden Einstieg in das *Low Level CAN Framework* ermöglichen.

Für Rückfragen, Fehlermeldungen und Anregungen sind die Autoren dennoch immer offen und dankbar. Deshalb hat die Konzernforschung der Volkswagen AG eine besondere EMail-Adresse eingerichtet, auf der die Autoren erreichbar sind:

`contact email: llcf@volkswagen.de`

Postadresse:
Volkswagen AG
Oliver Hartkopp
Brieffach 1776
D-38426 Wolfsburg

Oliver Hartkopp <oliver.hartkopp@volkswagen.de>
Dr. Urs Thürmann <urs.thuermann@volkswagen.de>