# Programmierschnittstellen für eingebettete Netzwerke in Mehrbenutzerbetriebssystemen am Beispiel des Controller Area Network

**Dissertation**

zur Erlangung des akademischen Grades

**Doktoringenieur (Dr.-Ing.)**

angenommen durch die Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von Dipl. Inform. Oliver Hartkopp
geboren am 17. Juni 1967 in Peine

Gutachter:
Prof. Dr. Jörg Kaiser
Prof. Dr. Jürgen Leohold
Prof. Dr. Jörg Nolte

Promotionskolloquium:
Magdeburg, den 14. Januar 2011

# Danksagung

Die vorliegende Arbeit ist neben meiner beruflichen Tätigkeit als Mitarbeiter der Volkswagen AG an der Fakultät für Informatik der Otto-von-Guericke-Universität Magdeburg entstanden. Stellvertretend für meine direkten Vorgesetzten möchte ich mich bei Dr. Bernd Rech bedanken, der mich seit 2006 im Hinblick auf meine Dissertation aktiv unterstützte und besonders dazu beigetragen hat, dass ich meine angesparte Mehrarbeit zur Anfertigung dieser Arbeit nutzen konnte. Das Zusammenschreiben der Arbeit am heimischen Schreibtisch in jeweils längeren Zeitabschnitten hat mir sehr geholfen.

Mein besonderer Dank gilt meinem Doktorvater Prof. Dr. Jörg Kaiser für die persönliche und fachliche Unterstützung sowie für den letztendlichen Anschub zur zeitnahen Abgabe dieser Arbeit. Herrn Prof. Dr. Kaiser und Herrn Prof. Dr. Leohold möchte ich an dieser Stelle auch für die wertvollen Hinweise und die qualifizierten Rückmeldungen zu den Arbeitsständen dieser Dissertation danken, die mir die Reduktion auf die zwingend darzustellenden Kernpunkte sehr erleichtert haben.

Stellvertretend für die vielfach erfahrene Unterstützung in meinem Arbeitsumfeld in der Volkswagen Konzernforschung und am Institut für Verteilte Systeme (Arbeitsgruppe Eingebettete Systeme und Betriebssysteme) möchte ich Dr. Urs Thürmann und (cand. Dr.) Michael Schulze für ihre technische Expertise und die in Diskussionen gemeinsam erarbeiteten Konzepte danken, die dieser Arbeit jeweils entscheidende Impulse gaben.

Besonders dankbar bin ich meiner Frau Andrea für ihr entgegengebrachtes Verständnis, ihre stets zutreffenden Review-Anmerkungen ('Das sollte besser in den Anhang', 'Das versteht niemand', 'Schreib es neu') und für die störungsfreie Arbeitsmöglichkeit zu Hause.

Die zahllosen abendlichen Stunden der Recherche im Internet und der technischen Diskussionen auf Mailing-Listen gaben mir häufig neue Ideen für diese Arbeit. Neben vielen motivierenden Rückmeldungen zu Anwendungen in Prototypen und Konzeptfahrzeugen verschiedener Fahrzeughersteller hat mich der Einsatz im industriellen Umfeld zur Steuerung eines Teilchenbeschleunigers im Kernforschungszentrum in Novosibirsk [2] besonders begeistert.

# Zusammenfassung

Etablierte Mehrbenutzerbetriebssysteme wie Microsoft Windows, QNX und Linux dringen aufgrund stetig steigender Prozessorleistungen zunehmend in die Domäne der eingebetteten Systeme vor, in der sie mit neuen Anforderungen an die Software- und Treiberunterstützung für gängige eingebettete Hardwarekomponenten konfrontiert werden.

Die Vorteile beim Einsatz von Standard Betriebssystemen sind die bekannten Betriebssystemschnittstellen und die weitreichende Unterstützung von Hardwarekomponenten, wie beispielsweise Massenspeicher oder Netzwerkkarten. Die in den Mehrbenutzerbetriebssystemen vorhandene Hardware-Abstraktion ermöglicht es dem Programmierer, über eine stabile Programmierschnittstelle die unterstützte Hardware unterschiedlicher Hersteller anzusprechen, ohne die eigene Anwendung darauf jeweils anpassen zu müssen.

Eine entsprechende Abstraktion eines Feldbusses, wie dem Controller Area Network, ist für die betrachten Mehrbenutzerbetriebssysteme nach dem Stand der Technik nicht vorhanden. Stattdessen realisieren die verschiedenen CAN Anwendungsentwickler individuelle, domänenspezifische Lösungen auf Basis der beim Controller Area Network definierten ISO/OSI Sicherungsschicht (Media Access Layer). Die unterschiedliche CAN Hardware und die fehlende Standardisierung für CAN Programmierschnittstellen führt in der Folge zu technischen Insellösungen und einer daraus resultierenden Abhängigkeit des Anwenders vom CAN Hardwarehersteller bzw. CAN Treiberhersteller (Vendor-Lock-In).

Zielsetzung dieser Dissertation ist die Erarbeitung eines allgemeingültigen Konzeptes zur Abstraktion von Feldbussen zur Integration in etablierte Mehrbenutzerbetriebssysteme.

Am Beispiel der Integration des Controller Area Networks in das Betriebsystem Linux soll das allgemeingültige Konzept erarbeitet und evaluiert werden. In diesem Rahmen wird untersucht, ob ein Top-Down-Ansatz mit etablierten Programmierschnittstellen zur Netzwerkkommunikation geeignet ist, die Anforderungen der verschiedenen CAN Nutzergruppen zu erfüllen. Dabei steht eine Abstraktion der unterschiedlichen CAN Hardware ebenso im Fokus, wie die vollumfängliche Abbildung vorhandener und zu erwartender Anwendungsfälle und Performanzvorgaben aus dem eingebetteten Umfeld.

Die Anwendung der erarbeiteten, allgemeingültigen Konzepte auf weitere Betriebssysteme (wie beispielsweise Microsoft Windows XP) und eingebettete Systeme schließt die Arbeit ab.

# Abstract

Due to ever increasing processor performance, established multi-user operating systems such as Microsoft Windows, QNX and Linux gain importance in the domain of embedded systems from where they are confronted with new demands on software and driver support for popular embedded hardware.

The advantages of using standard operating systems are the well known programming interfaces and the extensive support of hardware components such as mass storage devices or network cards. The existing hardware abstraction in multi-user operating systems allows the programmer to access supported hardware of different manufacturers without having to adapt the own application to each kind of existing hardware.

According to the state of the art a similar abstraction of a field bus, such as the Controller Area Network, is not available for the considered multi-user operating systems. Instead, the various CAN application developers implement individual domain specific solutions based on the ISO/OSI data link layer (Media Access Layer) of the Controller Area Network. The various CAN hardware and the lack of standards for CAN programming interfaces subsequently led to individual solutions for CAN applications, resulting in a dependence of the end-user from single CAN hardware manufacturers or single CAN driver manufacturers (vendor lock-in).

The aim of this thesis is to develop a general concept for an abstraction of field busses for the integration into established multi-user operating systems.

Using the example of the integration of the Controller Area Networks into the Linux operating system the general concept will be developed and evaluated. In this context it is investigated whether a top-down approach with established programming interfaces for the network communication suits the requirements of the various CAN user groups. Focus is the abstraction of the different CAN hardware as well as the full range of existing and expected use cases and performance requirements in the embedded environment.

A porting of the developed generalized concept to other operating systems (such as Microsoft Windows XP) and other embedded systems concludes this thesis.

# Inhaltsverzeichnis


**1 Einleitung** **1**

1.1 Zielsetzung der Arbeit . . . 6

1.2 Gliederung der Arbeit . . . 7

**2 Grundlagen und Anwendungen des Controller Area Network** **9**

2.1 Historie . . . 9

2.2 Kommunikationsverfahren . . . 10

2.2.1 Physikalische Bitübertragung . . . 10

2.2.2 Arbitrierung . . . 11

2.2.3 CAN Botschaftsformate . . . 13

2.2.4 Fehlererkennung . . . 14

2.3 Anwendungsgebiete für das Controller Area Network . . . 15

2.3.1 CAN im Kraftfahrzeug . . . 17

2.3.1.1 Spontane und zyklische Botschaften . . . 17

2.3.1.2 Multiplex Botschaften . . . 18

2.3.1.3 Netzwerkmanagement im Kraftfahrzeug / Wake-On-CAN 18

2.3.2 CAN basierte Transportprotokolle . . . 21

2.3.3 CAN basierte Kommunikationsprotokoll Standards . . . 28

2.3.3.1 Flotten-Management-Schnittstelle / J1939 . . . 28

2.3.3.2 CANopen . . . 30

2.3.3.3 CAN-Protokolle für Echtzeitanwendungen . . . 31

2.3.4 Definitionsbedarf bei der Verwendung des Controller Area Networks 32

2.4 Werkzeuge zur Analyse des Controller Area Networks . . . 34

2.5 Zusammenfassung . . . 35

**3 Stand der Technik für den Zugriff auf das Controller Area Network** **37**

3.1 CAN Controller . . . 38

3.2 Aktive und passive CAN PC Schnittstellenkarten . . . 42

3.3 Gerätetreiber für CAN PC Schnittstellenkarten . . . 44

3.4 CAN Programmierschnittstellen unter Linux . . . 46

3.5 CAN Programmierschnittstellen unter Windows . . . 50

3.6 Programmierschnittstellen für CAN Protokolle . . . 52

3.7 Zusammenfassung und Problemexposition . . . 54

**4 Die CAN Programmierschnittstelle in Mehrbenutzersystemen** **57**
4.1 Anforderungsanalyse . . . . . 58
4.1.1 Realisierung von vorhandenen Anwendungen . . . . . 59
4.1.2 Wiederverwendung von vorhandener CAN Software . . . . . 61
4.1.3 Wiederverwendung von vorhandener CAN Hardware . . . . . 62
4.1.4 Wiederverwendung von Konzepten aus der Informationstechnologie 63
4.1.5 Zusammenfassung der Anforderungen . . . . . 64
4.2 Bewertung möglicher Lösungsansätze . . . . . 65
4.2.1 Zeichenorientierte Betriebssystemschnittstellen . . . . . 65
4.2.2 Netzwerk Betriebssystemschnittstellen . . . . . 67
4.3 Konzeptentwicklung des CAN Zugriffes über Netzwerkschnittstellen . . . 72
4.3.1 Netzwerktreibermodell für CAN Hardware Treiber . . . . . 72
4.3.2 Behandlung von CAN Nachrichten im Betriebssystem . . . . . 74
4.3.2.1 Die Datenstruktur des CAN Frames . . . . . 76
4.3.3 Die Grenzen des PACKET Sockets . . . . . 78
4.3.4 Protokollfamilie PF_CAN . . . . . 80
4.3.5 Filtern und Verteilen empfangener CAN Nachrichten . . . . . 82
4.3.6 Lokales Echo gesendeter CAN Nachrichten . . . . . 84
4.3.7 CAN Protokolle in der Protokollfamilie PF_CAN . . . . . 85
4.3.7.1 CAN RAW Protokoll . . . . . 86
4.3.7.2 CAN Broadcast Manager Protokoll . . . . . 88
4.3.7.3 CAN Transport Protokolle . . . . . 90
4.4 Zusammenfassung . . . . . 92

**5 Bewertung** **95**
5.1 Performanzvergleich zum Stand der Technik . . . . . 95
5.1.1 Ungefilterte CAN Nachrichten . . . . . 96
5.1.2 CAN Inhaltsfilter . . . . . 102
5.1.3 Deadline Monitoring . . . . . 104
5.1.4 CAN Transportprotokoll ISO 15765-2 . . . . . 106
5.1.5 Auswirkungen der CAN Treiberperformanz . . . . . 110
5.2 Transfer der erarbeiteten Konzepte . . . . . 111
5.2.1 Realisierung unter Windows XP . . . . . 111
5.2.2 RT-SocketCAN . . . . . 112
5.2.3 Der Broadcast Manager auf einem 8-Bit Microcontroller . . . . . 114
5.2.4 Zusammenfassung . . . . . 116

**6 Zusammenfassung und Ausblick** **117**
6.1 Zusammenfassung der Arbeit . . . . . 117
6.2 Ausblick . . . . . 120

**Abkürzungsverzeichnis / Glossar** 121

**Literaturverzeichnis** 125

**Abbildungsverzeichnis** 131

**Tabellenverzeichnis** 133

**A Details der Linux Implementierung** 135
A.1 Busfehlersignalisierung durch Error Messages . . . . . . . . . . . . . . . . . 136
A.2 Filterkonzept im Software-Interrupt . . . . . . . . . . . . . . . . . . . . . . . 137
A.3 Zeitstempel für CAN Nachrichten . . . . . . . . . . . . . . . . . . . . . . . . 142
A.4 Echo Funktionalität für CAN Netzwerktreiber . . . . . . . . . . . . . . . . . 143
A.5 Der virtuelle CAN Netzwerktreiber (vcan) . . . . . . . . . . . . . . . . . . . 145
A.6 Hot(un)plugging von CAN Geräten . . . . . . . . . . . . . . . . . . . . . . . 146
A.7 Anzeige der internen Datenstrukturen zur Laufzeit . . . . . . . . . . . . . . 148
A.8 Migration existierender CAN Anwendungen . . . . . . . . . . . . . . . . . . 156
A.8.1 Anpassung existierender CAN Anwendungen . . . . . . . . . . . . . 156
A.8.2 Weiterverwendung existierender closed-source CAN Anwendungen 158
A.9 Konzepte für zukünftige Entwicklungen . . . . . . . . . . . . . . . . . . . . . 161
A.9.1 Nutzung aktiver CAN Hardware zur Nachrichtenfilterung . . . . . 161
A.9.2 Hash Verfahren für EFF Nachrichtenfilter . . . . . . . . . . . . . . 162
A.9.3 Identifierbasierte Priorisierung von CAN Sendewarteschlangen . . 164

**B Die Programmierschnittstelle für das CAN Subsystem in Linux** 169
B.1 Generelle Hinweise . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 173
B.1.1 CAN Protokolle . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 173
B.1.2 CAN Netzwerktreiber . . . . . . . . . . . . . . . . . . . . . . . . . . 173
B.1.3 virtuelle CAN Netzwerktreiber . . . . . . . . . . . . . . . . . . . . . 175
B.1.4 Konfigurieren und Hochfahren von CAN Netzwerktreibern . . . . 176
B.1.5 CAN Datenstrukturen . . . . . . . . . . . . . . . . . . . . . . . . . 177
B.1.6 Zeitstempel . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 179
B.2 CAN Raw Protokoll Sockets . . . . . . . . . . . . . . . . . . . . . . . . . . . 180
B.2.1 CAN Identifier Filter . . . . . . . . . . . . . . . . . . . . . . . . . . 180
B.3 CAN Broadcast Manager Protokoll Sockets . . . . . . . . . . . . . . . . . . . 183
B.3.1 Kommunikation mit dem Broadcast-Manager . . . . . . . . . . . . 183
B.3.2 TX_SETUP . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 187
B.3.2.1 Besonderheiten des Timers . . . . . . . . . . . . . . . . . 187
B.3.2.2 Veränderung von Daten zur Laufzeit . . . . . . . . . . . 188
B.3.2.3 Aussenden verschiedener Nutzdaten (Multiplex) . . . . . 188
B.3.3 TX_DELETE . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 189
B.3.4 TX_READ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 189
B.3.5 TX_SEND . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 189
B.3.6 RX_SETUP . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 190
B.3.6.1 Timeoutüberwachung . . . . . . . . . . . . . . . . . . . . 190
B.3.6.2 Drosselung von RX_CHANGED Nachrichten . . . . . . . 191

B.3.6.3 Nachrichtenfilterung (Nutzdaten - simple) . . . . . . . . 191
B.3.6.4 Nachrichtenfilterung (Nutzdaten - Multiplex) . . . . . . 191
B.3.6.5 Nachrichtenfilterung (Länge der Nutzdaten - DLC) . . . 193
B.3.6.6 Filterung nach CAN-ID . . . . . . . . . . . . . . . . . 193
B.3.6.7 Automatisches Beantworten von RTR-Frames . . . . . . 193
B.3.7 RX_DELETE . . . . . . . . . . . . . . . . . . . . . . . . . . 195
B.3.8 RX_READ . . . . . . . . . . . . . . . . . . . . . . . . . . . 195
B.3.9 Weitere Anmerkungen zum Broadcast-Manager . . . . . . . . . . 195
B.4 CAN Transportprotokoll Sockets (Stream) . . . . . . . . . . . . . . 197
B.4.1 Tracemode . . . . . . . . . . . . . . . . . . . . . . . . . . . 199
B.4.2 Besonderheiten des VAG TP1.6 . . . . . . . . . . . . . . . . . 200
B.4.3 Besonderheiten des VAG TP2.0 . . . . . . . . . . . . . . . . . 201
B.4.4 Besonderheiten des Bosch MCNet . . . . . . . . . . . . . . . . 203
B.5 CAN Transportprotokoll Sockets (Datagram) . . . . . . . . . . . . . 204
B.5.1 ISO-Transportprotokoll (ISO 15765-2) . . . . . . . . . . . . . 204

**C Werkzeuge** **209**
C.1 Anzeige und Erzeugung von CAN Daten . . . . . . . . . . . . . . . 210
C.1.1 candump . . . . . . . . . . . . . . . . . . . . . . . . . . . . 210
C.1.2 cansniffer . . . . . . . . . . . . . . . . . . . . . . . . . . . 213
C.1.3 cansend . . . . . . . . . . . . . . . . . . . . . . . . . . . . 215
C.1.4 canplayer . . . . . . . . . . . . . . . . . . . . . . . . . . . 216
C.1.5 cangen . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 217
C.1.6 canbusload . . . . . . . . . . . . . . . . . . . . . . . . . . . 218
C.2 Konvertierung von Log-Dateien . . . . . . . . . . . . . . . . . . . 219
C.2.1 log2long . . . . . . . . . . . . . . . . . . . . . . . . . . . . 219
C.2.2 asc2log . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 219
C.2.3 log2asc . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 220
C.3 CAN ISO-TP Transportprotokoll . . . . . . . . . . . . . . . . . . . 221
C.3.1 isotpsend . . . . . . . . . . . . . . . . . . . . . . . . . . . . 222
C.3.2 isotprecv . . . . . . . . . . . . . . . . . . . . . . . . . . . . 222
C.3.3 isotpdump . . . . . . . . . . . . . . . . . . . . . . . . . . . . 223
C.3.4 isotpsniffer . . . . . . . . . . . . . . . . . . . . . . . . . . . 224
C.3.5 isotptun (Internet Protocol over CAN) . . . . . . . . . . . . . 225

# 1 Einleitung

Die Leistungssteigerung von eingebetteten Prozessoren führt zunehmend zum Einsatz von etablierten Mehrbenutzerbetriebssystemen im Umfeld der eingebetteten Systeme.

Die aktuell verfügbaren und eingesetzten 32-Bit Prozessoren eignen sich mit ihrer für Multitasking und virtueller Speicherverwaltung benötigten Memory Management Unit hervorragend für den Einsatz von Standard Betriebssystemen wie GNU Linux, Microsoft Windows XP oder dem Echtzeitbetriebssystem QNX.

Diese Eignung führt zu einer Veränderung bei der Betriebssystemauswahl in klassischen Anwendungsfeldern von eingebetteten Systemen, wie beispielsweise Mobiltelefonen, Navigationssystemen, MP3-Abspielgeräten oder Empfangsgeräten für das digitale Fernsehen. Die zur Steuerung dieser Systeme verwendete Rechnereinheit wird vom Benutzer in der Regel nicht als eigenständiger Computer wahrgenommen, weshalb eine solche Recheneinheit in ihrem technischen Kontext als 'eingebettet' bezeichnet wird.

Durch den Einsatz in Konsumprodukten, wird das durch die hohen Stückzahlen beeinflusste Preis-Leistungs-Verhältnis für die eingebetteten 32-Bit Prozessoren für eine steigende Anzahl von Entwicklungsprojekten immer attraktiver. Dieses gilt besonders vor dem Hintergrund, dass sich die Softwareentwicklung mittlerweile zum Hauptrisiko für eine kosten- und termingerechte Projektabwicklung entwickelt hat. Aus diesem Grund kann sich die Auswahl einer etwas kostenintensiveren 32-Bit Hardware lohnen, wenn dadurch der Einsatz vorhandener und getesteter Softwarekomponenten ermöglicht wird, was wiederum das Entwicklungsrisiko erheblich reduzieren kann.

Neben der genannten Reduzierung des Entwicklungsrisikos bieten die etablierten Mehrbenutzerbetriebssysteme dem Entwickler beispielsweise die folgenden Vorteile:

- bekannte Programmierschnittstellen
- kurze Einarbeitungs- und Entwicklungszeiten
- ausgereifte Entwicklungsumgebungen
- ausgereifte Ressourcenverwaltung, Netzwerkunterstützung
- Unterstützung von Standard Hardware-Komponenten (bspw. Netzwerkkarten)
- grafische Oberflächen

Der Einsatz von Standard Betriebssystemen auf aktuellen eingebetteten Systemen stellt jedoch auch neue Anforderungen an die üblicherweise durch ein solches Standard Betriebssystem zu unterstützende Hardware. Neben den aus dem Umfeld der Personal Computer (PC) bekannten Hardware-Komponenten wie Festplatten, Grafikkarten und Tastaturen sind neue Schnittstellen für Kommunikationsprotokolle zu unterstützen, die bislang primär in eingebetteten Systemen Verwendung finden.

In einem modernen PC wird beispielsweise für die Abfrage und die Konfiguration von verbauten Hardware-Komponenten (z.B. der Dimmung von Notebook Displays oder der Abfrage von Temperatursensoren) der so genannte System Management Bus (SMBus) eingesetzt. Der SMBus verwendet zur Datenübertragung das von der Firma Philips entwickelte **I**nter-**I**ntegrated **C**ircuit Protokoll ($I^2C$)[45], welches zur Kommunikation zwischen integrierten Schaltungen entworfen wurde.

Die Verbreitung des $I^2C$ Protokolls zur Anbindung von integrierten Schaltungen in Konsumprodukten hat dazu geführt, dass diese ursprünglich *eingebettete* $I^2C$ Hardware zu den standardmäßig unterstützten PC-Komponenten zählt. In der Folge unterstützen verschiedene Standard Betriebssysteme das $I^2C$ Protokoll mit entsprechenden Hardwaretreibern, die unter einer einheitlichen Programmierschnittstelle eine Vielzahl verschiedener Sensoren über diverse Kommunikationshardware einbinden können. Die realisierte Abstraktion für den Zugriff auf eine beliebige $I^2C$ Hardware ermöglicht dem Anwender des Linux Betriebssystems beispielsweise eine über $I^2C$ angebundene Echtzeituhr transparent zu nutzen, ohne sich mit dem dabei verwendeten $I^2C$ Kommunikationsprotokoll im Detail auseinander setzen zu müssen.

Eine solche, die Hardware abstrahierende, einheitliche Anwenderschnittstelle existiert jedoch nicht für das im Rahmen dieser Arbeit betrachtete Controller Area Network (CAN), das als Hardware-Komponente ebenfalls auf vielen eingebetteten Systemen vorhanden ist. Für die fehlende Unterstützung des Controller Area Networks in Standard Betriebssystemen können verschiedene potenzielle Gründe gefunden werden, die eine allgemeingültige Abstraktion analog zum $I^2C$ Protokoll erschweren:

- Die Spezifikation des Controller Area Networks umfasst lediglich den unteren Teil (Media Access Control) des Data Link Layers (OSI Schicht 2) und lässt dem CAN Anwendungsprogrammierer die auf diesem Layer basierende Kommunikation offen.
- Durch die fehlende verbindliche Spezifikation von höheren Protokollschichten wird der CAN Bus in verschiedenen Anwendungsgebieten mit unterschiedlichsten Kommunikationsverfahren und Protokollen genutzt.
- Die Vielzahl bereits existierender, proprietärer Lösungen für die Realisierung von CAN basierten Protokollen und von CAN Hardwaretreibern machen den Umstieg auf eine neue Lösung unattraktiv.
- Die historisch gewachsenen CAN Programmierschnittstellen und CAN Zugriffskonzepte sind nicht mit dem Stand der Technik in Mehrbenutzerbetriebssystemen kompatibel.

Bei der Betrachtung der historisch gewachsenen CAN Programmierschnittstellen und CAN Zugriffskonzepte sind die Software Entwicklungsprozesse für eingebettete Systeme zum Zeitpunkt der Einführung des Controller Area Networks zu bewerten. Als 1982 an der Universität von Kalifornien die Internet Kommunikationsprotokolle und die dazugehörigen Programmierschnittstellen in das dort entwickelte BSD Unix integriert wurden, geschah dieses bereits auf Basis eines vorhandenen Mehrbenutzerbetriebssystems mit Ressourcenmanagement und Unix-Dateisystem.

Für den Programmierer eines eingebetteten Systems ergaben sich zur damaligen Zeit jedoch typischerweise folgende Entwicklungsbedingungen:

- Kein Betriebssystem auf dem Prozessor
- Eine Entwicklungsumgebung des Prozessorherstellers (Assembler)
- Ein Schaltplan für die zu programmierende Hardware
- Eine Registerbeschreibung des Prozessors
- Eine Registerbeschreibung der in dem Prozessor enthaltenen Hardware

Der Programmierer war der einzige Nutzer des Prozessors und musste sich beispielsweise keine Gedanken über konkurrierende Prozesse (weiterer Systembenutzer) machen. Die Kommunikation zu anderen eingebetteten Systemen wurde über das Füllen von Registerinhalten mit abschließendem Sendebefehl realisiert und als 'Betriebssystem' wurde häufig eine einfache Schleife über alle zu realisierenden Aufgaben gewählt.

Zu dieser Zeit wurden bereits Personal Computer (PC) als Messmittel zur Entwicklung eingebetteter Systeme eingesetzt. Dazu wurden handelsübliche Rechner um Hardware-Bausteine für eingebettete Kommunikationsprotokolle erweitert, um beispielsweise von einem Personal Computer CAN Daten an ein zu testendes Steuergerät senden zu können.

Mit dem Einzug der ersten Mehrbenutzerbetriebssysteme wie OS/2 und Microsoft Windows NT in die Labore der Entwickler von eingebetteten Systemen ist die Exklusivität für den Zugriff auf Hardware-Ressourcen für die erstellten CAN Anwendungsprogramme jedoch nicht mehr gewährleistet. Die nicht deterministischen Systemantwortzeiten und die Latenzen für Unterbrechungsanforderungen machen einen *aktive* Hardware für den Zugriff auf eingebettete Kommunikationsprotokolle nötig, wo bisher der Protokoll-Hardware-Baustein unmittelbar ausgelesen und dessen Daten direkt vom Testprogramm verarbeitet werden konnten. Eine *aktive* Hardware nutzt zur zeitgrechten Verarbeitung von CAN Protokollen einen separaten (eingebetteten) Prozessor, der über eigenen Speicher und eine spezialisierte Firmware verfügt. Die *aktive* CAN Hardware verfügt über eine herstellerspezifische Schnittstelle zum Personal Computer. Üblicherweise kann trotz der Umsetzung der CAN Schnittstelle in einem Mehrbenutzerbetriebssystem jeweils nur ein CAN Anwendungsprogramm zu einer Zeit auf die CAN Hardware zugreifen.

Die hohen Kosten bei der kommerziellen Entwicklung proprietärer CAN Anwendungen führen zumeist zu Implementierungen, welche zur direkten Erfüllung der projektspezifischen Anforderungen erforderlich sind. Die Erarbeitung allgemeingültiger Lösungsansätze wird zusätzlich dadurch erschwert, dass bei kommerziellen Produkten üblicherweise ein geringer Austausch von Entwicklererfahrungen stattfindet, um dem potenziellen Wettbewerber keinen Einblick in die eigene Technologie zu ermöglichen.

Aufgrund der bereits geleisteten Investitionen in proprietäre Entwicklungen und den mangelnden Austausch zur Definition gemeinsamer Anforderungen an eine standardisierte CAN Infrastruktur in Standard Betriebssystemen, ist von der Seite einzelner kommerzieller CAN Anbieter keine allgemeingültige Lösung zu erwarten. Bei verschiedenen kommerziellen Anbietern kann eine CAN Anwendung nur im Zusammenhang mit einer CAN Hardware des selben Herstellers erworben werden (engl. Bundle). Dieses Geschäftsmodell zwingt den Kunden in eine Abhängigkeit zum Anbieter (engl. Vendor-Lock-In), welches im Rahmen der Problemexposition detaillierter dargestellt wird.

Als Folge der genannten Wettbewerbssituation sind öffentlich zugängliche Informationen von Schnittstellenspezifikationen kommerzieller CAN Hardware unüblich. Verschiedene, nicht-kommerzielle OpenSource Projekte konnten daher nur Hardware nutzen, bei denen die Spezifikation des CAN Hardwarebausteins öffentlich verfügbar war oder wenn der Treiber des Herstellers im Quelltext öffentlich vorlag. Diese Einschränkungen können mit dazu beigetragen haben, dass bis zu dieser Arbeit keine hardwareabstrahierende, einheitliche CAN Treiberschnittstelle für das OpenSource Betriebssystem Linux existierte.

Nahezu jede CAN Hardware wird nach dem Stand der Technik mit einem herstellerspezifischen Gerätetreiber und mit einer herstellerspezifischen Programmierschnittstelle ausgeliefert. Eine Änderung der Prozessorplattform oder der Wechsel des CAN Hardware Herstellers in einem Entwicklungsprojekt führt daher zu einer riskanten und kostenintensiven Überarbeitung der Anwendung in ihren Kommunikationsschnittstellen zum CAN Bus. Übertragen auf ein Beispiel aus der Fahrzeugtechnik entspräche das in etwa der Situation, dass bei einem Wechsel auf eine andere Reifenmarke die zusätzlich mitgelieferten Achsen und Antriebswellen an das eigene Fahrzeug angepasst werden müssten. Diese aus Sicht der einzelnen CAN Hardware Hersteller zunächst *positive* Kundenbindungsfunktion bringt allerdings den Nachteil mit sich, die hauseigenen Gerätetreiber an die sich ständig weiterentwickelnden Betriebssysteme anpassen zu müssen.

Eine Lösung für wiederverwendbare CAN Anwendungen bestünde in der gleichzeitigen Unterstützung verschiedener CAN Treiberschnittstellen. Ein solcher Ansatz stellt für den Programmierer einer CAN Anwendung nach dem Stand der Technik die einzige Möglichkeit dar, dass erstellte Programm wiederverwenden zu können. Die Unterstützung verschiedener Treiberschnittstellen kann es beispielsweise ermöglichen, ein kommerzielles CAN Anwendungsprogramm einem größeren Kundenkreis zugänglich zu machen.

So wirbt die Firma Port GmbH für das CAN Anwendungsprogramm "CANopen Device Monitor" mit einer vergleichsweise umfangreichen CAN Treiberunterstützung, wenn man die zuvor beschriebenen Probleme des CAN Marktes betrachtet:

```
The CANopen Device Monitor works best with the USB-Interface CPC-USB
or the PCI-Card CPC-PCI from EMS Wünsche.

Supported hardware interfaces are:

    * CPC - EMS Wünsche
    * LevelX - I+ME Actia
    * Kvaser CAN interfaces
    * IGW 900 by SSV,
    * CAN232 by Lawicel,
    * and can4linux.
```

(Quelle: http://www.port.de/engl/download/download.html)

Der CAN Anwendungsprogrammierer muss sich bei diesem Ansatz auf eine aufwändige Pflege und eine ständige Weiterentwicklung seiner verschiedenen Treiberadapter einstellen. Zusätzlich ist er davon abhängig, dass der Anbieter des CAN Treibers seine proprietäre Schnittstelle nicht verändert. Möchte der CAN Hardware Anbieter den Verkauf seiner eigenen CANopen Anwendung fördern, macht eine Veränderung der Treiberschnittstelle bei einer entsprechenden Marktpräsenz für ihn sogar Sinn.

Für den Anbieter einer CAN Anwendung, der nicht auf eine Verknüpfung seiner Software mit einer einzelnen Hardware abzielt, ist der Aufwand zur Erstellung einer wiederverwendbaren CAN Anwendung nach dem Stand der Technik erheblich. Eine Unterstützung *aller* marktüblichen CAN Treiber ist schon aus ökonomischer Sicht nicht sinnvoll zu realisieren.

## 1.1 Zielsetzung der Arbeit

Seit der Einführung des Controller Area Networks im Jahr 1986 haben sich nach dem Stand der Technik unterschiedliche, zueinander inkompatible Hardware- und Softwarelösungen zum Zugriff auf den CAN Bus in Mehrbenutzerbetriebssystemen entwickelt. Diese zueinander inkompatiblen Lösungen erschweren und behindern die Wiederverwendung realisierter CAN Anwendungen auf Mehrbenutzerbetriebssystemen, weil die Anwendung jeweils auf eine herstellerspezifische CAN Programmierschnittstelle angepasst werden muss.

Mit der zunehmenden Verwendung etablierter Mehrbenutzerbetriebssysteme in eingebetteten Systemen mit bereits integrierter CAN Hardware werden die bisherigen, spezialisierten Lösungen für Mehrbenutzerbetriebssysteme in Frage gestellt. Die Nutzung von Standard Betriebssystemen auf eingebetteten Prozessoren begründet die Erwartung der Anwender, die im eingebetteten Bereich gängigen Hardwarekomponenten analog zu Standard Hardwarekomponenten wie Massenspeichern und Netzwerkkarten zu abstrahieren. Eine solche Abstraktion ermöglicht die Verwendung von identischen Anwendungsprogrammen bei der Nutzung von unterschiedlichen Hardwarekomponenten der gleichen Art - beispielsweise von Netzwerkkarten verschiedener Hersteller.

Das Ziel dieser Arbeit ist die Definition einer allgemeingültigen, abstrakten und mehrbenutzerfähigen Schnittstelle für den Zugriff auf das Controller Area Network in etablierten Mehrbenutzerbetriebssystemen.

Dabei stehen einfache, möglichst etablierte Standard Programmierschnittstellen und eine Abstraktion der unterschiedlichen CAN Hardware ebenso im Fokus, wie die vollumfängliche Abbildung vorhandener und zu erwartender Anwendungsfälle und Performanzvorgaben aus dem eingebetteten Umfeld.

## 1.2 Gliederung der Arbeit

**Kapitel 1 Einleitung**
Führt in das Thema ein und beschreibt die Zielsetzung dieser Arbeit.

**Kapitel 2 Grundlagen und Anwendungen des Controller Area Network**
Beschreibt das Controller Area Network und dessen Kommunikationsverfahren auf dem Kommunikationsmedium. Zusätzlich gibt dieses Kapitel einen Überblick zu Anwendungsprotokollen und Einsatzgebieten des Controller Area Networks. Der Schwerpunkt liegt dabei auf CAN Anwendungen im Kraftfahrzeug, welche hinsichtlich der Anforderungsanalyse in Kapitel 4 einen Großteil der bekannten Anwendungsszenarien abdeckt.

**Kapitel 3 Stand der Technik für den Zugriff auf das Controller Area Network**
Gibt einen Überblick zu Hardwarekonzepten, Treibermodellen und verwendeten Programmierschnittstellen für das Controller Area Network. Dabei wird insbesondere auf die Situationen in Mehrbenutzerbetriebssystemen wie GNU Linux und Microsoft Windows eingegangen. Am Ende des Kapitels wird eine Zusammenstellung von wesentlichen Anforderungen an eine CAN Programmierschnittstelle für Mehrbenutzerbetriebssysteme gegeben und eine Problemexposition für den Stand der Technik durchgeführt.

**Kapitel 4 Die CAN Programmierschnittstelle in Mehrbenutzersystemen**
Die in Kapitel 3 beschriebenen, wesentlichen Anforderungen werden um zusätzliche Anforderungen ergänzt, die aus drei realen Anwendungsszenarien abgeleitet werden können. Die in Kapitel 3 beschriebenen Realisierungsansätze nach dem Stand der Technik erweisen sich als nicht geeignet, die vorhandenen Anforderungen zu erfüllen. Es wird die versuchsweise Umsetzung der identifizierten Anforderungen in ein netzwerkbasiertes Konzept in mehreren Iterationen erläutert und diskutiert.

**Kapitel 5 Bewertung**
Bewertet das erarbeitete Konzept im Hinblick auf seine Performanz und seine Transferierbarkeit in weitere Betriebssysteme und Projekte. Dazu werden anhand verschiedener, in der Arbeit betrachteter Anwendungsszenarien Performanzvergleiche mit dem Stand der Technik durchgeführt. Zusätzlich wird die Anwendung des erarbeiteten, allgemeingültigen Konzeptes zur Integration von Feldbussen in Mehrbenutzerbetriebssysteme auf weitere Betriebssysteme (wie beispielsweise Microsoft Windows XP) und auf ein eingebettetes 8-Bit Mikrocontrollersystem dargestellt.

**Kapitel 6 Zusammenfassung und Ausblick**
Fasst die vorliegende Arbeit und ihren Beitrag zum Stand der Technik zusammen.

## Anhang A Details der Linux Implementierung

Die im Rahmen von Kapitel 4 beschriebenen Verfahren und Konzepte werden in anhand einer realen Implementierung für das Betriebssystem GNU Linux erläutert und vertieft. Dabei wird die Akzeptanzfilterung von CAN Botschaften im Software-Interrupt als kritischer Pfad mit einer anwendungsfallabhängigen Komplexität identifiziert. Es werden Anforderungen für einen produktiven Einsatz des gewählten Konzeptes gelöst, sowie auf die Möglichkeiten der Migration oder Weiternutzung bereits bestehender CAN Anwendungen eingegangen. Abschließend werden Lösungsmöglichkeiten für mögliche funktionale Erweiterungen des erarbeiteten CAN Subsystems aufgezeigt, wie beispielsweise Konzepte für CAN spezifische Warteschlangen oder Hash-Verfahren zur Filterung von CAN Identifiern.

## Anhang B Die Programmierschnittstelle für das CAN Subsystem in Linux

Referenz der SocketCAN Programmierschnittstelle, die im Rahmen dieser Arbeit entstanden ist.

## Anhang C Werkzeuge

Gibt einen Überblick zu Werkzeugen und deren Aufrufsyntax, die im Rahmen dieser Arbeit verwendet werden.

# 2 Grundlagen und Anwendungen des Controller Area Network

In diesem Kapitel wird in Abschnitt 2.2 das Kommunikationsverfahren des Controller Area Networks erläutert, soweit es für das Verständnis dieser Arbeit erforderlich ist. In Abschnitt 2.3 wird eine Auswahl verschiedener Anwendungsgebiete für das Controller Area Network dargestellt. Für detaillierte technische Spezifikationen zum Controller Area Network sei auf entsprechende weiterführende Literatur, wie [31] oder die entsprechenden ISO Normen 11898-1, 11898-2 und 11898-3 [23] verwiesen.

## 2.1 Historie

Für eine sichere Kommunikation von mehreren Steuergeräten in einem Kraftfahrzeug wurden im Jahr 1981 beim Automobilzulieferer Robert Bosch GmbH verschiedene serielle Übertragungsverfahren untersucht und bewertet. Da keine der existierenden Lösungen die Anforderungen der Ingenieure erfüllen konnten, wurde 1983 mit der Entwicklung des Controller Area Networks begonnen, welches im Februar 1986 auf einem Kongress der Society of Automotive Engineers (SAE) der Öffentlichkeit vorgestellt wurde [6].

Bereits 1987 waren erste so genannte CAN Controller von der Firma Philips [46] und von der Firma Intel [21] verfügbar, die das bitserielle Kommunikationsprotokoll nach der Bosch Spezifikation [52] realisieren. Das von Bosch entwickelte Controller Area Network wurde 1993 als ISO 11898-1 Norm [23] verabschiedet.

Der CAN Bus wurde erstmals ab 1991 in der Mercedes-Benz S-Klasse (Baureihe W140) zur Vernetzung von bis zu fünf Steuergeräten serienmäßig eingesetzt und ist heutzutage der Industriestandard für Kommunikationsnetzwerke in Kraftfahrzeugen. Aufgrund seines robusten Kommunikationsverfahrens (siehe Abschnitt 2.2.4) wird das Controller Area Network auch in industriellen Steuerungen, im Marine-Umfeld sowie für die Vernetzung von redundanten Steuerungskomponenten und wissenschaftlichen Experimenten in Raumsonden [28] eingesetzt. Eine Übersicht über die in den verschiedenen Anwendungsgebieten genutzten Kommunikationsprotokolle für das Controller Area Network wird in Kapitel 2.3 gegeben.

## 2.2 Kommunikationsverfahren

In diesem Abschnitt wird das Kommunikationsverfahren des Controller Area Networks beschrieben. Die für das Verständnis dieser Arbeit relevanten Grundlagen werden in den Abschnitten 2.2.2 (Arbitrierung) und 2.2.3 (CAN Botschaftsformate) erläutert.

### 2.2.1 Physikalische Bitübertragung

Die serielle Bitübertragung beim Controller Area Network wird im Allgemeinen über zwei ungeschirmte, miteinander verdrillte Leitungen realisiert. Die Endpunkte der verdrillten Leitungen sind mit ohmschen Abschlusswiderständen versehen, welche eine Reflektion des Bitstromes an den Leitungsenden verhindern (Terminierung). Es besteht auch die Möglichkeit, den CAN Bitstrom über eine einzelne Leitung zu führen bzw. andere Formen der Bus-Terminierung zu wählen. Diese Details sind für das Verständnis dieser Arbeit nicht relevant und werden daher nicht weiter dargestellt.

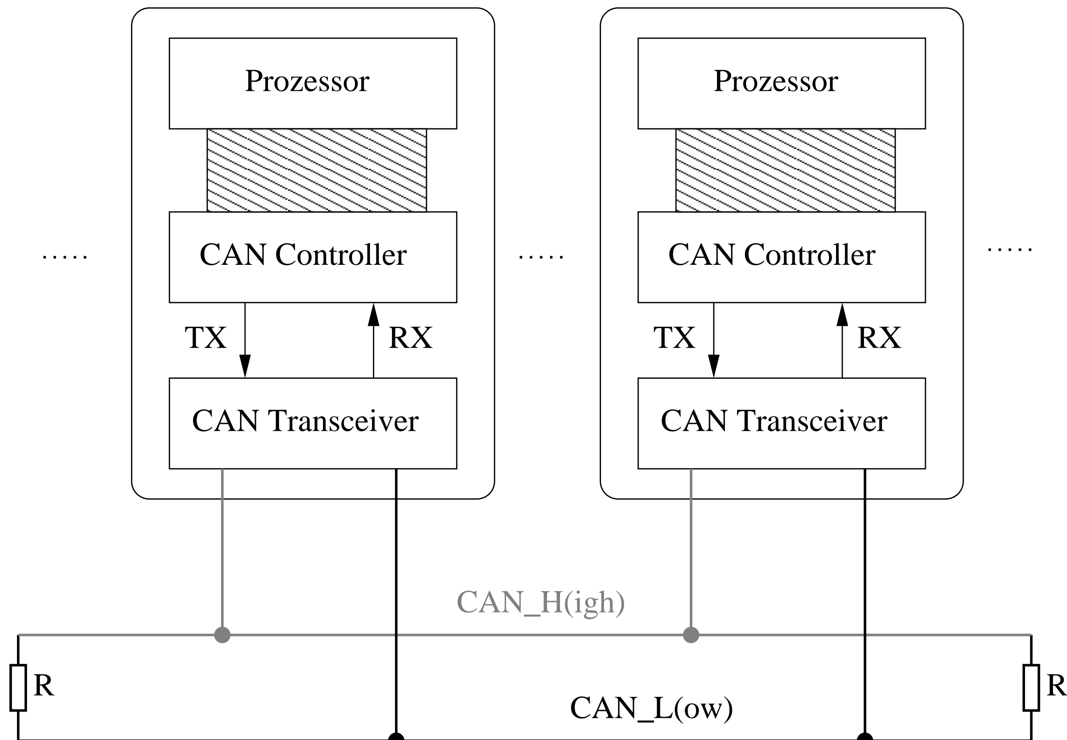



Abbildung 2.1: Das Prinzip eines CAN Netzwerkes

Die CAN Transceiver setzen die vom CAN Controller vorgegebenen Signale in differenzielle Spannungen um, was zusammen mit der verdrillten Leitung ein hohes Maß an Störunempfindlichkeit bietet. Eine externe Beeinflussung wirkt sich grundsätzlich

auf beide verdrillte Leitungen aus, wodurch sich die Differenzspannung zwischen den Leitungen nicht verändert.

Eine Besonderheit des Buszugriffes besteht in der Tatsache, dass der CAN Controller über den CAN Transceiver auch während der Aussendung seiner(!) CAN Botschaft eine Rückmeldung über den aktuellen Buszustand erhält. Vergleichbar mit der Aussendung einer seriellen Folge von Tonsignalen zur Übertragung von Daten kann der Controller erkennen, ob ein anderer CAN-Teilnehmer einen Ton sendet, wenn er selbst keinen Ton sendet. Diese Möglichkeit des parallelen 'Horchens' während der Übertragung einer CAN Botschaft bildet die Basis für das kollisionsauflösende Zugriffsverfahren (CSMA/CR) durch die bitweise Arbitrierung.

### 2.2.2 Arbitrierung

Beim asynchronen Zugriff auf ein gemeinsames Übertragungsmedium wird zunächst gewartet, dass eine Datenübertragung abgeschlossen ist bevor der Sender seine zu sendende Nachricht schickt. Das Verfahren Carrier Sense Multiple Access (CSMA) (Mehrfachzugriff mit Trägerprüfung) kann dabei in verschiedenen Ausführungen genutzt werden:

**CSMA/CD** Collision Detection (Erkennung einer Kollision mehrerer Sender)

**CSMA/CA** Collision Avoidance (Vermeidung einer Kollision mehrerer Sender)

**CSMA/CR** Collision Resolution (Auflösung einer Kollision mehrerer Sender)

Während das CSMA/CD Verfahren beispielsweise beim Ethernet eingesetzt wird und bei (Amateur-)Funknetzen mit dem CSMA/CA Verfahren nach dem Empfang einer Nachricht eine zufällige Zeit gewartet wird, setzt man beim Controller Area Network auf das CSMA/CR Verfahren.

Die Auflösung einer Kollision erfolgt durch die oben beschriebene Möglichkeit, während einer Aussendung einer CAN Botschaft gleichzeitig auf dem Medium mithören zu können. Die eigene Übertragung einer '1' kann jederzeit durch die Übertragung einer '0' von einem weiteren CAN Knoten 'überschrieben' werden[1]. Diese Tatsache kann von dem '1'-sendenden CAN Knoten erkannt werden, wodurch dieser seine eigene Übertragung abbricht und zu einem späteren Zeitpunkt versucht, seine Botschaft erneut zu senden.

Wie in Abbildung 2.2 dargestellt, beenden die CAN Knoten Node1 und Node2 jeweils ihre eigene Aussendung, wenn erkennbar ist, dass die von ihnen gesendete '1' von einer '0' eines anderen Knotens überschrieben wird. Durch die Tatsache, dass die Adressinformation - der so genannte CAN Identifier - zu Beginn einer CAN Botschaft gesendet wird, wird die Vergabe des Senderechts praktisch durch diesen CAN Identifier festgelegt.

[1]Die '0' wird dabei als *dominanter* Bitwert bezeichnet, die '1' wird als *rezessiv* bezeichnet.

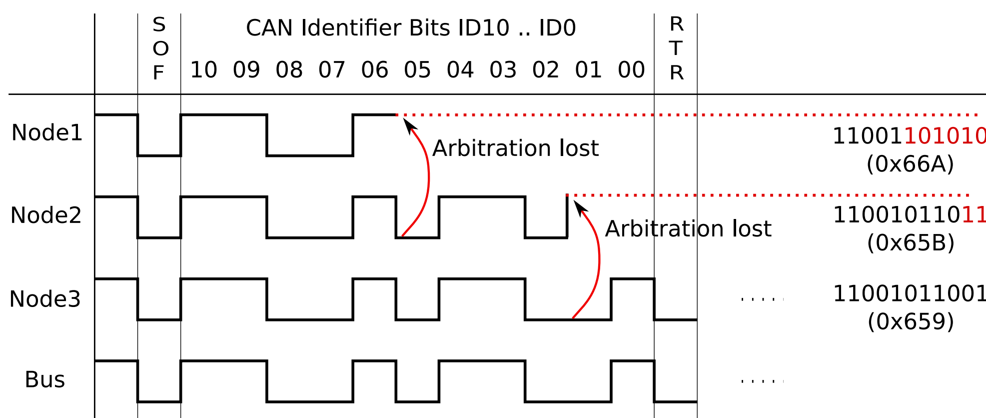



Abbildung 2.2: Kollisionsauflösung durch bitweise Arbitrierung [28]

Daraus ergeben sich implizit zwei Regeln für die Vergabe von CAN Identifiern:

1. Der CAN Identifier mit dem niedrigeren Wert erhält das Senderecht aufgrund der bitweisen Arbitrierung.
2. Es dürfen niemals zwei CAN Knoten mit dem selben CAN Identifier senden.

Abhängig vom jeweiligen Anwendungsfall kann der CAN Identifier beispielsweise als eine Absenderadresse verwendet werden, weil in einem korrekt definierten Netzwerk ein bestimmter CAN Identifier (zu einer Zeit) nur von einem CAN Knoten versendet werden darf. In Kapitel 2.3 werden neben der Identifikation des jeweiligen sendenden CAN Knotens verschiedene Möglichkeiten zur Verwendung des CAN Identifiers dargestellt.

#### Arbitrierung und Echtzeitverhalten

Die mögliche Verdrängung einer niedriger priorisierten CAN Botschaft (mit größerem CAN Identifier Wert) erlaubt die effiziente Auflösung von Spitzen im Kommunikationsaufkommen (traffic peaks). Die in [29] beschriebene Unterscheidung in CAN Nachrichten mit garantierter Latenzzeit und CAN Nachrichten mit nicht-kritischer Latenzzeit erlaubt eine erhebliche Reduzierung der benötigten Übertragungsrate auf dem prioritätsbasierten CAN Bus gegenüber einer Auslegung ohne prioritätsbasierten Medienzugriff.

Durch die Verdrängung und die mitunter mehrfachen Anläufe, eine niedriger priorisierte CAN Botschaft auf das Kommunikationsmedium zu senden, ist allerdings eine Echtzeitigkeit der Kommunikation nicht zwingend gegeben. Zur Sicherstellung von Echtzeitanforderungen sind daher verschiedene Verfahren zur Steuerung der Aussendung von CAN Botschaften entwickelt worden. Das zeitgesteuerte Protokoll TTCAN [20] [26] (Time Triggered CAN - siehe Kapitel 2.3.3.3) ermöglicht je nach Konfiguration mit einem

TDMA Verfahren eine kollisionsfreie Aussendung von CAN Botschaften, wobei die bei CAN vorhandene Möglichkeit zur Arbitrierung in diesen Fällen nicht angewendet wird.

Ein effizienteres echtzeitfähiges Kommunikationsverfahren wird in [34] und [33] dargestellt, welches unter Benutzung der Arbitrierung einen vergleichsweise höheren Datendurchsatz und eine größere Flexibilität in der Übertragung verschiedener Daten erlaubt.

Dazu werden die zu übertragenden Inhalte in

- Hard-Realtime Messages
- Soft-Realtime Messages
- Non-Realtime Messages

unterschieden und abhängig von den jeweiligen Anforderungen zeitlich und zusätzlich durch unterschiedliche CAN Identifier differenziert gesendet. Im Gegensatz zu einem einfachen TDMA Verfahren wird hierbei die (zum Teil arbitrierende) CAN Kommunikation mit einem EDF (Earliest Deadline First) basierten Scheduling zeitlich gesteuert.

Die mögliche Verdrängung einer niedriger priorisierten CAN Botschaft ist in besonderer Weise relevant für die korrekte Darstellung des CAN Datenverkehrs in Mehrbenutzersystemem, welches in Kapitel 4.3.6 eingehend erläutert wird.

### 2.2.3 CAN Botschaftsformate

Die auf den CAN Bus als serielle Bitströme gesendeten CAN Botschaften (auch als CAN Nachrichten oder CAN Frames bezeichnet) können in zwei Formate aufgeteilt werden:

**Standard Frame Format** mit einem 11 Bit CAN Identifier ($2^{11} = 2048$ Adressen)

**Extended Frame Format** mit einem 29 Bit CAN Identifier ($2^{29} = 536870912$ Adressen)

Beide Botschaftsformate verfügen im direkten Anschluss an die durch den CAN Identifier gegebene Adressinformation über ein so genanntes Remote Transmission Request Bit (RTR). Dieses RTR Bit kann zur Abfrage von Informationen genutzt werden, die auf einem bestimmten CAN Identifier zur Verfügung gestellt werden. Ein Beispiel:

1. CAN Knoten A sendet auf dem CAN Identifier 123 alle 30 Sek. einen Sensorwert
2. CAN Knoten B ist neu gestartet und benötigt den Sensorwert umgehend
3. CAN Knoten B sendet den CAN Identifier 123 mit gesetztem RTR Bit
4. CAN Knoten A sendet auf dem CAN Identifier 123 den Sensorwert

Das bei Punkt 3 der Knoten B mit dem CAN Identifier 123 sendet, widerspricht an sich der Vorgabe, dass niemals zwei CAN Knoten mit dem selben CAN Identifier senden dürfen. Durch die Tatsache, dass das RTR Bit allerdings rezessiv (also '1') ist, kann mit dem Knoten A keine Kollision auftreten, da Daten Frames an der Stelle des RTR Bit eine '0' setzen, wodurch der Remote Transmission Request von Knoten B verworfen wird. Das RTR Bit ist somit ein Adressierungsbit mit der geringsten Priorität.

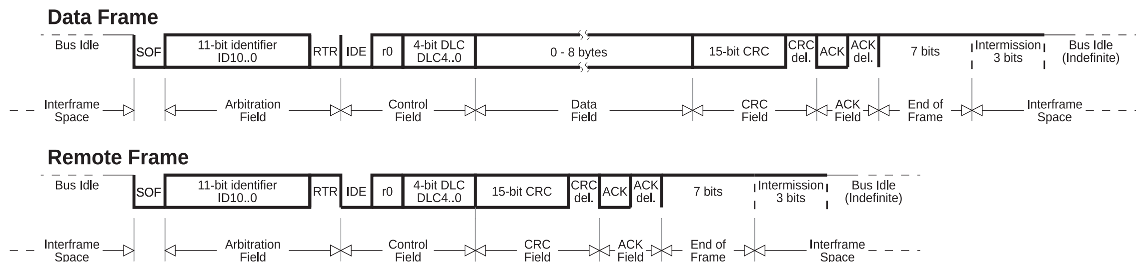



Abbildung 2.3: Standard Frame Format (SFF) [1]

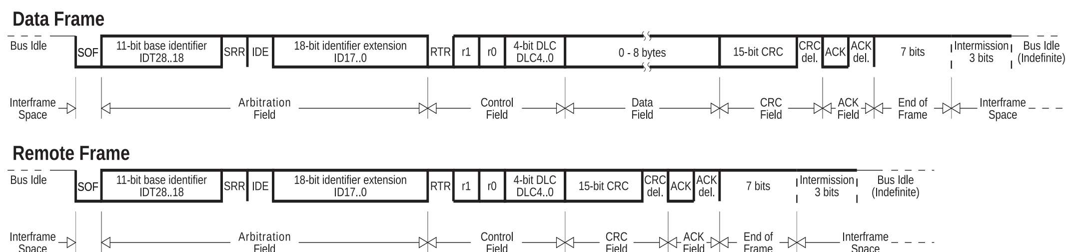



Abbildung 2.4: Extended Frame Format (EFF) [1]

Die Abbildungen 2.3 und 2.4 zeigen den detaillierten Aufbau einer CAN Botschaft ohne die zusätzlichen Maßnahmen zur Vermeidung von mehr als fünf gleichpoligen Bits (Bit-Stuffing). Auf eine weitergehende Detaillierung wird an dieser Stelle verzichtet und auf die Dokumentation [52] verwiesen.

Für die Nutzung von CAN Botschaften in dieser Arbeit sind folgende Fakten relevant:

- Es gibt CAN Identifier mit 11 Bit (SFF) oder 29 Bit (EFF) Länge
- Es gibt die Möglichkeit der Anforderung von CAN Botschaften (RTR Frames)
- Die Länge der Nutzdaten beträgt 0 .. 8 Byte (bei RTR Frames immer 0)

### 2.2.4 Fehlererkennung

Der CAN Controller muss fehlerbehaftete Übertragungen erkennen können und das aktuell übertragene, fehlerhafte CAN Frame verwerfen. Nach verschiedenen Untersuchungen

kann die Restfehlerwahrscheinlichkeit, d.h. die Wahrscheinlichkeit, dass ein vorhandener Fehler nicht erkannt wird, beim CAN mit deutlich unter $10^{-11}$ angegeben werden [63].

Bei einer CAN Datenübertragung können durch ein entsprechendes *Bit-Monitoring* im empfangenden CAN Controller folgende Fehler erkannt werden:

- Fehler im Bit-Stuffing: Nach fünf gleichpoligen Bits muss ein Bit in der jeweils anderen Polarität gesendet werden.
- Fehler im CRC-Feld: Die Prüfsumme der zyklische Redundanzprüfung (Cyclic Redundancy Check) muss korrekt sein.
- Fehler im Botschaftsformat: Die Nachricht (z.B. die Bitwerte reservierter Bits) muss der CAN Spezifikation entsprechen.
- Fehlendes ACK: Beim Senden einer CAN Botschaft muss von mindestens einer Gegenstelle ein dominantes Acknowledge-Bit gesetzt werden.

Im Fall eines erkannten Fehlers sendet der CAN Controller eine Folge von sechs dominanten Bits (eine klare Verletzung des Bit-Stuffing), was allen Busteilnehmern den erkannten Fehler anzeigt, wodurch das aktuelle CAN Frame verworfen wird. Der oben beschriebenen Fehlersignalisierung folgt eine Anzahl von acht rezessiven Bits.

## 2.3 Anwendungsgebiete für das Controller Area Network

Wie in Kapitel 2.2.3 beschrieben, stellt die CAN Botschaft (auch *CAN Nachricht* oder *CAN Frame*) die Basis der Kommunikation über den CAN Bus dar.

Aus der Sicht eines CAN Anwenders hat ein CAN Frame folgende Eigenschaften:

- Es gibt CAN Identifier mit 11 Bit (SFF) oder 29 Bit (EFF) Länge.
- Je geringer der Wert des CAN Identifiers ist, desto höher ist seine Priorität.
- Es gibt die Möglichkeit der Anforderung von CAN Botschaften (RTR Frames).
- Die Länge der Nutzdaten beträgt 0 .. 8 Byte (bei RTR Frames immer 0).

Je nach Anwendungsfall können verschiedene Inhalte innerhalb der veränderbaren Datenfelder des CAN Frames durch den Anwender definiert werden. Beispielsweise kann der CAN Identifier für die Adressierung einer Komponente genutzt werden oder er definiert die Art des Messwertes, der sich in den bis zu 8 Byte Nutzdaten befindet.

Beispiel zur Verwendung der CAN Botschaftinhalte:

| | |
|---|---|
| CAN Indentifier | 123 |
| Datalength Code | 2 (Byte) |
| Nutzdaten Registerinhalte | 45 67 89 99 00 11 22 33 |

Könnte wie folgt interpretiert werden:

Die Drehzahl beträgt 4567 Umdrehungen pro Minute
(oder)
Gerät2 an Gerät3: "Storniere den Auftrag 6745"

Im ersten Fall bedeutet die CAN-ID 123 einen inhaltsadressierten Wert mit der Bedeutung 'Drehzahl' - im zweiten Fall bedeutet die erste Stelle ('1') der CAN-ID das Kommando 'Storniere' und die folgenden Stellen den Sender ('2') und Empfänger ('3') des Kommandos.

Dieses führt unmittelbar zu der Fragestellung der Darstellung von Werten, die über die Grenze eines Nutzdaten Bytes hinausgehen: Auch hier besteht für die sogenannte *Byte-Order* keine Vorgabe seitens der CAN Spezifikation. Ob die Anordnung der Anwendungsinformationen in den Nutzdaten des CAN Frames in der Intel-Notation (*Little Endian* - das geringstwertigste Byte zuerst) oder in der Motorola-Notation (*Big Endian* - das höchstwertigste Byte zuerst) erfolgt, liegt im Ermessen des CAN Anwenders.

Die Nutzdaten Bytes sind (je nach CAN Treiber) von 0 .. 7 bzw. 1 .. 8 aufsteigend nummeriert und die jeweilige Gültigkeit der Werte wird durch den Datalength Code (im Beispiel '2') angezeigt. Im obigen Beispiel wurden auch die Zahlen '89 99 00 11 22 33' aus den Registern des CAN Controllers ausgelesen, die jedoch nicht gültig sind und die auch nicht über den CAN Bus übertragen wurden.

Weiterhin definiert der CAN Anwender, ob die Verwendung von RTR Frames in seinem spezifischen Netzwerk unterstützt wird oder ob es eine Timeout-Behandlung gibt, wenn beispielsweise ein bestimmter CAN Identifier länger als 500 ms nicht empfangen wurde. Letzteres wird beispielsweise im Umfeld von Kraftfahrzeugen genutzt, um den Ausfall von CAN Knoten oder von Sensoren zu erkennen, die im Normalfall ihre Informationen in einem festen zeitlichen Zyklus auf den Bus senden (auch wenn sich der Inhalt der Daten nicht verändert hat).

Zusammenfassend kann mit dem Controller Area Network der Kommunikationsbedarf jeder Anwendung erfüllt werden, der sich in die Datenstruktur eines CAN Frames (CAN Identifier, RTR Bit, bis zu 8 Byte Nutzdaten) abbilden lässt. Die Informationen des gesamten Frames sind je nach Anwendungsfall beliebig interpretierbar.

Im Folgenden sollen verschiedene Beispiele für solche unterschiedlichen Anwendungsfälle gegeben werden, um einen Eindruck über verschiedene Anwendungsgebiete zu gewinnen.

### 2.3.1 CAN im Kraftfahrzeug

Die Verwendung im Kraftfahrzeug war die ursprüngliche Motivation zur Entwicklung des Controller Area Networks. In diesem Abschnitt werden Anwendungen auf der Basis des CAN Busses dargestellt, wie sie im Umfeld von Kraftfahrzeugen zum Einsatz kommen.

#### 2.3.1.1 Spontane und zyklische Botschaften

Grundsätzlich werden auf dem Broadcast Medium CAN im Fahrzeug so genannte 'Signale' versendet. Unter dem Begriff 'Signal' wird dabei ein (Sensor-)Wert verstanden, der in den Nutzdatenbereich eines CAN Frames kodiert wird. Diese Kodierung bezieht sich auf die Auflösung und den Wertebereich des Signals.

Wenn beispielsweise die Außentemperatur im Bereich von -50°C bis +77°C mit einer Auflösung von 0,5°C kodiert werden soll, könnte die Abbildung in ein einzelnes Byte in den Nutzdaten des CAN Frames so aussehen:

$$\text{Außentemperatur} = \text{Bytewert}/2 - 50$$

Durch eine solche Signaldefinition kann die Außentemperatur gemäß den Anforderungen auch auf einen einzelnen Byte-Wert (0 .. 255) abgebildet werden. Der Bytewert 0 entspräche -50°C und der Bytewert 255 entspräche +77,5°C.

Mehrere derart definierte Signale werden häufig in einer CAN Botschaft zusammengefasst, wenn sie thematisch zueinander gehören und von einem CAN Teilnehmer gesendet werden. Beispielsweise wären die Motordrehzahl, die Öltemperatur und der Öldruck thematisch in eine 'MotorStatus' Botschaft einzuordnen, oder der Türöffnungsstatus, der Türverriegelungsstatus und der elektrische Fensterheber in eine 'Fahrertür' Botschaft.

Zur Erkennung des Ausfalls einer Komponente werden Signalbotschaften üblicherweise in einem festen zeitliche Raster von 5 ms bis zu 2000 ms von der Datenquelle gesendet. Die Aktualität des in einer CAN Botschaft enthaltenen Signals ist folglich abhängig von diesem zeitlichen Raster, der so genannten Zykluszeit der CAN Botschaft. Neben der Zusammenfassung von thematisch ähnlichen Signalen in eine CAN Botschaft werden daher auch Signale nach ihrer benötigten Aktualität zusammen gruppiert. Für weitergehende Informationen sei auf [39] verwiesen.

Die Zykluszeit stellt für die Ausfallerkennung eine zeitliche Obergrenze dar, womit generell die Zykluszeit kleiner sein darf, als spezifiziert. Dieses wird genutzt, um beispielsweise Änderungen der Signalwerte sofort mit der definierten CAN Botschaft übertragen zu können. Nach diesem Aussenden dieser asynchronen Änderung wird der definierte Zyklus dann wieder aufgenommen.

Die Verwendung zyklischer Botschaften führt zu einer kontinuierlichen Buslast im Fahrzeug, wie eigene Messungen mit dem Werkzeug **canbusload** (siehe Kapitel C.1.6) bei einem Volkswagen T5 California belegen. Die unter 'Zündung an' angegebenen Werte bleiben auch mit den CAN Daten eines sich bewegenden Fahrzeugs während einer Fahrt weitgehend gleich.

| Fahrzeugstatus | Antriebs-CAN | | Komfort-CAN | |
|---|---|---|---|---|
| | Frames/s | Buslast % | Frames/s | Buslast % |
| Zündung aus | 0 | 0 | 188 | 15 |
| Zündung an | 945 | 20 | 230 | 19 |

Tabelle 2.1: Buslast bei einem Volkswagen Nutzfahrzeug

#### 2.3.1.2 Multiplex Botschaften

Für einen effizienten Umgang bei der Vergabe von CAN Identifiern wurden so genannte Multiplex Botschaften eingeführt, bei denen ein Multiplex Identifier innerhalb der Nutzdaten des CAN Frames die Bedeutung der anderen Nutzdaten in diesem CAN Frame bestimmt. Dieses Verfahren wird vorzugsweise eingesetzt, wenn sich die zu übertragenden Daten wenig oder, wie im Beispiel einer Fahrgestellnummer, gar nicht ändern.

Soll beispielsweise ein Steuergerät für die Reifendruckkontrolle den Status für vier Reifen melden und dafür nur einen CAN Identifier nutzen, wird im ersten Byte der CAN Nutzdaten die Reifennummer übertragen und in den folgenden 7 Bytes die reifenspezifischen Werte. Das erste Byte der CAN Nutzdaten wäre dabei der beschriebene Multiplex Identifier. Dieser Multiplex Identifier würde sich bei jeder (zyklischen) Aussendung der CAN Botschaft nach dem Schema 0, 1, 2, 3, 0, 1, ... erhöhen und so die reifenspezifischen Werte nach vier Aussendungen vollständig übertragen haben.

#### 2.3.1.3 Netzwerkmanagement im Kraftfahrzeug / Wake-On-CAN

Das Netzwerkmanagement in Kraftfahrzeugen soll das sichere Starten und Herunterfahren von Steuergeräten auf dem CAN Bus realisieren. Dieses betrifft üblicherweise nur CAN Busse, die nicht durch eine separate Zündungsleitung ('Klemme 15') bei eingestecktem Zündschlüssel aktiviert werden. Ein solcher CAN Bus ohne Zündungsleitung ist beispielsweise der Komfort-CAN (oder auch 'Body-CAN'), der im Fahrzeug für die Vernetzung von Türen, Klimaanlagen oder Radios zuständig ist. Steuergeräte auf einem solchen Komfort-CAN werden mit weckfähigen, fehlertoleranten CAN Transceivern wie dem TJA1054 [47] oder dem TJA1041A [48] der Firma NXP realisiert.

Im so genannten Schlafmodus verbrauchen die referenzierten CAN Transceiver einen Ruhestrom von typischerweise 20µA, was bei korrekter Realisierung der Hardware des Steuergerätes, dessen Ruhestromaufnahme darstellt. Werden im Schlafmodus alternierende Spannungen an den CAN Leitungen (siehe Kapitel 2.2.1) erkannt, schaltet der CAN Transceiver das Netzteil des Steuergerätes ein und es startet. Sendet also beim Öffnen einer Fahrzeugtür das entsprechende Türsteuergerät die Information "Tür offen" auf den CAN Bus, weckt es damit alle Wake-On-CAN-fähigen Steuergeräte auf.

Die verschiedenen Steuergeräte in einem Kraftfahrzeug werden von unterschiedlichen Zulieferfirmen nach der Spezifikation des Fahrzeugherstellers entwickelt. Das Aufwecken über den CAN führt zu einem entsprechenden Stromverbrauch im Fahrzeug, der beispielsweise bei jedem Öffnen der Fahrzeugtür die Fahrzeugbatterie belastet. Zur Vermeidung von Fahrzeugen mit entladener Batterie ('Liegenbleiber') wird ein sicheres Verfahren benötigt, um die geweckten Steuergeräte jedes Zulieferers nach einer möglichst kurzen Zeit wieder in den energiesparenden Schlafmodus versetzen zu können.

### OSEK Netzwerkmanagement

Als ein solches sicheres Verfahren kann das CAN Netzwerkmanagement (NM) des Industriekonsortiums OSEK/VDX ("Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug") betrachtet werden [42]. Das OSEK Netzwerkmanagement realisiert mit CAN Botschaften einen so genannten 'logischen Ring' aller Steuergeräte eines CAN Busses. In diesen logischen Ring können sich die eindeutig nummerierten CAN Steuergeräte auch nachträglich einhängen. Der Ausfall eines Steuergerätes wird ebenfalls erkannt und das ausgefallene Steuergerät wird daraufhin übersprungen.

Das folgende Beispiel zeigt einen stabilen OSEK Netzwerkmanagement Ring mit den eindeutigen Netzwerkmanagement Adressen 0x00, 0x01, 0x08, 0x0C, 0x0F, 0x16 und 0x1E. Die CAN Nachrichten wurden mit dem Werkzeug **candump** (siehe C.1.1) angezeigt:

```
user@host:~> candump can1,400:7C0 -td
 (0.000000)  can1  40C  [6] 0F 01 00 00 00 00
 (0.042147)  can1  40F  [6] 16 01 00 00 00 00
 (0.048373)  can1  416  [6] 1E 01 00 00 00 00
 (0.042079)  can1  41E  [6] 00 01 00 00 00 00
 (0.043581)  can1  400  [6] 01 01 00 00 00 00
 (0.049055)  can1  401  [6] 08 01 00 00 00 00
 (0.044438)  can1  408  [6] 0C 01 00 00 00 00
 (0.051533)  can1  40C  [6] 0F 01 00 00 00 00
 (0.051608)  can1  40F  [6] 16 01 00 00 00 00
 (0.042302)  can1  416  [6] 1E 01 00 00 00 00
 (0.048596)  can1  41E  [6] 00 01 00 00 00 00
 ...
```

Wie anhand der farblichen Markierungen erkennbar ist, enthält das erste Nutzdaten Byte einer Netzwerkmanagement Botschaft die NM Adresse des nachfolgenden Teilnehmers. Durch Abweichungen in diesem Schema wird die Rekonfiguration des logischen Ringes gemäß der genannten Spezifikation ausgelöst. Als Basis-Wert für den CAN Identifier (OSEK NM 'IdBase', siehe [42] Kapitel 2.2.1.2) wird im Beispiel der Wert 0x400 verwendet. Die dargestellten CAN Identifier errechnen sich aus 'IdBase' + 'NM Adresse'.

Neben dem Aufbau eines logischen Ringes wird über das zweite Nutzdatenbyte der Ringbotschaft der Status des Netzwerkes kommuniziert und abgeglichen. Ein Steuergerät, das aufgrund seiner Programmierung erkennt, dass es sich abschalten könnte, sendet diese Information im zweiten Byte in seiner Ringbotschaft. Wenn alle beteiligten Steuergeräte diesen Zustand eingenommen haben, wird das Netzwerkmanagement und damit die gesamte CAN Kommunikation gemeinsam beendet.

Die weiteren vier Nutzdatenbytes sind reserviert für fahrzeugherstellerspezifische Daten. Hier könnte beispielsweise eine Information über die Weck-Ursache abgelegt werden oder eine Information, ob das Steuergerät über einen Eintrag im Fehlerspeicher verfügt. Für eine Fehleranalyse am Fahrzeug ist die Weckursache (z.B. 'interne Weckursache', 'Tür wurde geöffnet' oder 'durch CAN Datenverkehr') eine wertvolle Informationsquelle.

Als Nachteil des OSEK Netzwerkmanagements muss allerdings die benötigte Bandbreite auf dem CAN Bus gesehen werden. Es wird mindestens alle 50ms eine Ringbotschaft von einem der beteiligten Steuergeräte gesendet - im Beispiel variiert der zeitliche Abstand zwischen zwei Ringbotschaften gemäß dem Delta-Zeitstempel von **candump** zwischen 42ms und 52ms. Diese mindestens 20 Nachrichten pro Sekunde belasten den in Tabelle 2.1 betrachteten CAN Bus mit ca. 2% - also einem Zehntel der gemessenen Buslast.

### Bosch MCNet Netzwerkmanagement

Häufig wird CAN Netzwerkmanagement im Kraftfahrzeug mit dem oben beschriebenen OSEK CAN Netzwerkmanagement gleichgesetzt. Eine erheblich einfachere Lösung stellt das Netzwerkmanagement des CAN-basierten Mobile Communication Network (MCNet) der Robert Bosch GmbH dar (siehe MCNet System Concept [54]).

Bei MCNet sendet ein Master-CAN-Knoten jede Sekunde eine bestimmte Nachricht auf dem hoch prioren CAN Identifier '1' (siehe "Arbitrierung“ ab Seite 11). Die benötigte Bandbreite ist bei diesem Verfahren aufgrund der Zykluszeit von einer Sekunde vernachlässigbar. Bleibt diese zyklische Nachricht aus, schalten sich die CAN Teilnehmer auf dem CAN Bus nach einem definierten Verfahren nach spätestens fünf Sekunden ab.

### 2.3.2 CAN basierte Transportprotokolle

Die vorgegebene Anzahl von maximal acht Bytes in den Nutzdaten einer CAN Botschaft ist für verschiedene Anwendungsfälle nicht ausreichend. Es werden beispielsweise im Rahmen einer Fahrzeugdiagnose Daten wie die 17-stellige Fahrzeug-Identnummer (VIN) aus Steuergeräten ausgelesen. Ein weiterer Anwendungsfall aus dem Umfeld der Fahrzeugdiagnose stellt die Aktualisierung der Betriebssoftware eines Steuergerätes über den CAN Bus dar - das so genannte 'Flashen'. Dabei werden die Inhalte des Programmspeichers des Steuergerätes üblicherweise in Blöcken von mehr als 200 Byte übertragen.

Für die Übertragung von Anwendungsdaten, die eine Größe von acht Byte überschreiten, werden im Fahrzeugumfeld so genannte CAN Transportprotokolle eingesetzt. CAN Transportprotokolle ermöglichen aus der Sicht des Anwenders eine Punkt-zu-Punkt-Kommunikation zwischen zwei Kommunikationsteilnehmern. Dieses wird realisiert durch die Verwendung zweier exklusiv für den Transportkanal vereinbarter CAN Identifier. Zum Beispiel:

| | | |
|---|---|---|
| CAN Identifier 740 | : | Daten von Teilnehmer A zu Teilnehmer B |
| CAN Identifier 760 | : | Daten von Teilnehmer B zu Teilnehmer A |

Innerhalb der acht Byte Nutzdaten wird für die Segmentierung der Anwendungsdaten ein Verfahren definiert, das vorzugsweise im ersten Nutzdatenbyte protokollspezifische Zustände und Segmentzähler realisiert. Dieses erste Byte entspricht der Protocol Control Information (PCI) der Protocol Data Unit (PDU) im OSI-Modell und kann - abhängig vom gewählten Protokoll - folgende Anforderungen realisieren:

- Aufbau einer Verbindung *
- Abbau einer Verbindung *
- Testen einer Verbindung *
- Start einer neuen Nachricht
- Ende einer Nachricht
- Segmentzähler für segmentierte Nachrichten
- Flusskontrolle für den Datenverkehr
  - Bestätigungen (für korrekt empfangene Segmente)
  - Wiederholungsanforderungen (bei inkorrekten Segmentzählern)
  - Statusinformationen der Kommunikationpartner (z.B. 'Receiver not ready')
  - Vereinbarung von zeitlichen Kommunikationsparametern (z.B. minimaler zeitlicher Abstand aufeinanderfolgender CAN Botschaften)

*nur bei verbindungsorientierten CAN Transportprotokollen

CAN Transportprotokolle können als in verbindungsorientierte oder verbindungslose Protokolle ausgeführt sein. Bei verbindungsorientierten Protokollen wird ein Transportkanal geöffnet und nach Beendigung der Kommunikation geschlossen - vergleichbar mit TCP/IP beim Internet Protokoll. Dabei wird protokollseitig auf die korrekte Reihenfolge und Fehlerfreiheit der übertragenen Daten geachtet. Bei einem verbindungslosen Protokoll wird analog zu UDP/IP eine *nicht-zuverlässige* Übertragung von Anwendungsdaten realisiert. Dabei kann es zu einem Verlust von Datenpaketen, einem mehrfachen Empfang von Datenpaketen oder einem Vertauschen der Reihenfolge von Datenpaketen kommen, auf die die kommunizierende Anwendung eingerichtet sein muss.

Als eine minimale Ausprägung eines CAN Transportprotokolls kann ein Multiplex Verfahren bezeichnet werden, bei dem im Protocol Controll Information Byte durch einen Zähler (den Multiplex Identifier) der segmentierte Teil einer PDU zugewiesen wird (siehe Broadcast Announce Message in Kapitel 2.3.3.1 auf Seite 29). Diese segmentierten Mehrpaket Nachrichten (siehe Kapitel B.3.6.4) werden vom Empfänger der Nachrichten nicht im Rahmen des Protokolls bestätigt und sind folglich als verbindungsloses Protokoll ohne Datenflusskontrolle einzuordnen.

Weitere Unterschiede bei CAN Transportprotokollen ergeben sich aus dem Kommunikationsverfahren, dass als Halbduplex[2]- oder Vollduplex[3]-Verfahren implementiert sein kann. Auch besteht optional die Möglichkeit, Kommunikationskanäle verbindungsorientierter Protokolle aufgrund ausbleibender Kommunikationsnutzung zu schließen.

Die zwei für einen Transportkanal vorgesehenen CAN Identifier können für einen Anwendungsfall fest definiert werden oder sie werden anhand von Steuergerätenummern oder anderen eindeutigen Adressen berechnet. Ein solches Berechungsverfahren wird beispielsweise beim so genannten Channel-Setup der Volkswagen Transportprotokolle TP1.6 [61] und TP2.0 [62] eingesetzt. Die Vergabe von CAN Identifiern für Transportkanäle wird üblicherweise in den Spezifikationen für CAN Transportprotokolle spezifiziert, obwohl die Definition der CAN Identifier im Allgemeinen von der Definition des Datentransportverfahrens getrennt werden kann.

Zur Verdeutlichung soll eine Datenübertragung durch ein verbindungsorientiertes CAN Transportprotokoll am Beispiel des Bosch MCNet [53] und des Volkswagen TP2.0 im Vergleich gezeigt werden. Bei gegebenen CAN Identifiern (0x740 und 0x760) für den Transportkanal wird eine lesbare Nachricht in Kleinbuchstaben versendet, die an den ursprünglichen Sender in Großbuchstaben zurückgesendet wird.

[2]halbduplex: wechselseitiger Sendebetrieb
[3]vollduplex: gleichzeitiger Sendebetrieb

Anzeige der beschriebenen MCNet Kommunikation mit dem Werkzeug **candump**. Für eine verbesserte Darstellung wurde die Ausgabe von **candump** nachträglich koloriert.

```
user@host:~> candump any -ta
(1238775477.097285)  vcan2  740  [3] A0 0F 00
(1238775477.099627)  vcan2  760  [2] A1 0F
(1238775478.124455)  vcan2  740  [8] 20 69 63 68 20 77 6F 6C
(1238775478.135631)  vcan2  740  [8] 21 6C 74 20 69 63 68 20
(1238775478.143600)  vcan2  740  [8] 22 77 61 65 72 20 65 69
(1238775478.151600)  vcan2  740  [7] 13 6E 20 68 75 68 6E
(1238775478.151618)  vcan2  760  [1] B4
(1238775483.118314)  vcan2  760  [8] 20 49 43 48 20 57 4F 4C
(1238775483.126616)  vcan2  760  [8] 21 4C 54 20 49 43 48 20
(1238775483.134600)  vcan2  760  [8] 22 57 41 45 52 20 45 49
(1238775483.142600)  vcan2  760  [7] 13 4E 20 48 55 48 4E
(1238775483.142623)  vcan2  740  [1] B4
```

Die zugehörige Ausgabe des **tpdump** Werkzeuges, welches das PCI Byte dekodiert:

```
user@host:~> tpdump -s 740 -d 760 -i vcan2 -c -a -ta
(1238775477.097285)  vcan2  740  [CS] 0F 00 - '..'
(1238775477.099627)  vcan2  760  [CA] 0F - '.'
(1238775478.124455)  vcan2  740  [DT] 69 63 68 20 77 6F 6C - 'ich wol'
(1238775478.135631)  vcan2  740  [DT] 6C 74 20 69 63 68 20 - 'lt ich '
(1238775478.143600)  vcan2  740  [DT] 77 61 65 72 20 65 69 - 'waer ei'
(1238775478.151600)  vcan2  740  [DT,AR,EOM] 6E 20 68 75 68 6E - 'n huhn'
(1238775478.151618)  vcan2  760  [ACK,RR]
(1238775483.118314)  vcan2  760  [DT] 49 43 48 20 57 4F 4C - 'ICH WOL'
(1238775483.126616)  vcan2  760  [DT] 4C 54 20 49 43 48 20 - 'LT ICH '
(1238775483.134600)  vcan2  760  [DT] 57 41 45 52 20 45 49 - 'WAER EI'
(1238775483.142600)  vcan2  760  [DT,AR,EOM] 4E 20 48 55 48 4E - 'N HUHN'
(1238775483.142623)  vcan2  740  [ACK,RR]
```

Im Detail wird anhand dieses Beipiels sichtbar:

- der Aufbau des Transportprotokoll Kanals zwischen 0x740 und 0x760:
  0x740 → 0x760: Connection Setup
  0x760 → 0x740: Connection Acknowledge

  ```
  (1238775477.097285)  vcan2  740  [CS] 0F 00 - '..'
  (1238775477.099627)  vcan2  760  [CA] 0F - '.'
  ```

- die Kommunikation von 0x740 nach 0x760 mit Bestätigung von 0x760:
  0x740 → 0x760: Datentelegramme.
  0x740 → 0x760: Das letzte Datentelegramm mit End Of Message Markierung.
  0x760 → 0x740: Bestätigung des korrekten Empfangs der Datentelegramme.

```
(1238775478.124455)  vcan2  740  [DT] 69 63 68 20 77 6F 6C - 'ich wol'
(1238775478.135631)  vcan2  740  [DT] 6C 74 20 69 63 68 20 - 'lt ich '
(1238775478.143600)  vcan2  740  [DT] 77 61 65 72 20 65 69 - 'waer ei'
(1238775478.151600)  vcan2  740  [DT,AR,EOM] 6E 20 68 75 68 6E - 'n huhn'
(1238775478.151618)  vcan2  760  [ACK,RR]
```

- die Kommunikation von 0x760 nach 0x740 mit Bestätigung von 0x740:
  0x760 → 0x740: Datentelegramme.
  0x760 → 0x740: Das letzte Datentelegramm mit End Of Message Markierung.
  0x740 → 0x760: Bestätigung des korrekten Empfangs der Datentelegramme.

```
(1238775483.118314)  vcan2  760  [DT] 49 43 48 20 57 4F 4C - 'ICH WOL'
(1238775483.126616)  vcan2  760  [DT] 4C 54 20 49 43 48 20 - 'LT ICH '
(1238775483.134600)  vcan2  760  [DT] 57 41 45 52 20 45 49 - 'WAER EI'
(1238775483.142600)  vcan2  760  [DT,AR,EOM] 4E 20 48 55 48 4E - 'N HUHN'
(1238775483.142623)  vcan2  740  [ACK,RR]
```

Zum Vergleich dazu die Kommunikation des Volkswagen Transportprotokolls TP2.0:

```
user@host:~> tpdump -s 740 -d 760 -i vcan2 -c -a -ta
(1238774638.674965)  vcan2  740  [CS] 0F CA FF 00 FF - '.....'
(1238774638.675024)  vcan2  760  [CA] 0F CA FF 00 FF - '.....'
(1238774639.481545)  vcan2  740  [DT] 69 63 68 20 77 6F 6C - 'ich wol'
(1238774639.483616)  vcan2  740  [DT] 6C 74 20 69 63 68 20 - 'lt ich '
(1238774639.483622)  vcan2  740  [DT] 77 61 65 72 20 65 69 - 'waer ei'
(1238774639.483627)  vcan2  740  [DT,AR,EOM] 6E 20 68 75 68 6E - 'n huhn'
(1238774639.483672)  vcan2  760  [ACK,RR]
(1238774642.617825)  vcan2  760  [DT] 49 43 48 20 57 4F 4C - 'ICH WOL'
(1238774642.618611)  vcan2  760  [DT] 4C 54 20 49 43 48 20 - 'LT ICH '
(1238774642.618617)  vcan2  760  [DT] 57 41 45 52 20 45 49 - 'WAER EI'
(1238774642.618622)  vcan2  760  [DT,AR,EOM] 4E 20 48 55 48 4E - 'N HUHN'
(1238774642.618666)  vcan2  740  [ACK,RR]
(1238774643.618051)  vcan2  760  [DC]
(1238774643.618626)  vcan2  740  [DC]
```

Die Unterschiede ergeben sich aus den zusätzlichen (Timing-) Parametern, die beim Connection-Setup und dem Connection-Acknowledge beim Aufbau der Kommunikation übergeben werden. Außerdem wird der TP2.0 Kanal am Ende mit einem Disconnect (DC) geschlossen. Das Schließen eines Transportkanals ist bei MCNet nicht vorgesehen. Grundsätzlich unterscheiden sich Art der Segmentierung und die Bestätigung der Datentelegramme bei den Transportprotokollen MCNet, TP2.0 und TP1.6 nicht.

Die im Anhang in Kapitel C.3 beschriebenen Open Source Werkzeuge erlauben die Kommunikation mit dem verbindungslosen CAN Transportprotokoll ISO 15765-2 [27]. Unter der Voraussetzung, dass ein Empfänger einer segmentierten Nachricht vorhanden ist, z.B. mit

```
isotprecv -s 740 -d 760 vcan2
```

ergibt das Versenden der in vorherigen Beispiel gezeigten Daten mit

```
echo "69 63 68 20 77 6F 6C ... 75 68 6E" | isotpsend -s 760 -d 740 vcan2
```

folgende Anzeige der ISO-TP Kommunikation mit dem Werkzeug **candump** (koloriert):

```
user@host:~> candump any -ta
(1238780505.466769)  vcan2  760  [8] 10 1B 69 63 68 20 77 6F
(1238780505.466927)  vcan2  740  [3] 30 00 00
(1238780505.466970)  vcan2  760  [8] 21 6C 6C 74 20 69 63 68
(1238780505.467007)  vcan2  760  [8] 22 20 77 61 65 72 20 65
(1238780505.467039)  vcan2  760  [8] 23 69 6E 20 68 75 68 6E
```

Die zugehörige Ausgabe des **isotpdump** Werkzeuges (hier ohne Zeitstempel):

```
user@host:~> isotpdump -s 740 -d 760 vcan2 -c -a
  vcan2  760  [8]  [FF] ln: 27    data: 69 63 68 20 77 6F     -  'ich wo'
  vcan2  740  [3]  [FC] FC: 0 = CTS # BS: 0 = off # STmin: 0x00 = 0 ms
  vcan2  760  [8]  [CF] sn: 1    data: 6C 6C 74 20 69 63 68  -  'llt ich'
  vcan2  760  [8]  [CF] sn: 2    data: 20 77 61 65 72 20 65  -  ' waer e'
  vcan2  760  [8]  [CF] sn: 3    data: 69 6E 20 68 75 68 6E  -  'in huhn'
```

Die Kommunikation erfolgt gemäß dem in Abbildung 2.5 gezeigten Kommunikationsprinzip. Wie im Beispiel erkennbar, ergeben sich folgende Unterschiede zu verbindungsorientierten Transportprotokollen wie Bosch MCNet und Volkswagen TP1.6 / TP2.0:

- Es wird keine Verbindung aufgebaut, d.h. keine 'CS' oder 'CA' Botschaften.
- Das erste CAN Frame (First Frame 'FF') enthält die Länge der zu versendenden Daten (hier 27 bzw. hexadezimal 0x1B) und die ersten 6 Bytes der Nutzdaten.
- Der Empfänger teilt seine Anforderungen für zeitliche Abstände von CAN Nachrichten und die maximale Anzahl unbestätigter Datentelegramme (Blocksize) an den Sender der Daten in einem (ggf. wiederholten) Flow Control Frame 'FC' mit.
- Es wird keine Bestätigung des Empfängers der Daten für den korrekten Empfang der vollständigen Nachricht gesendet.
- Es wird keine Verbindung abgebaut, d.h. keine 'DC' Botschaften.

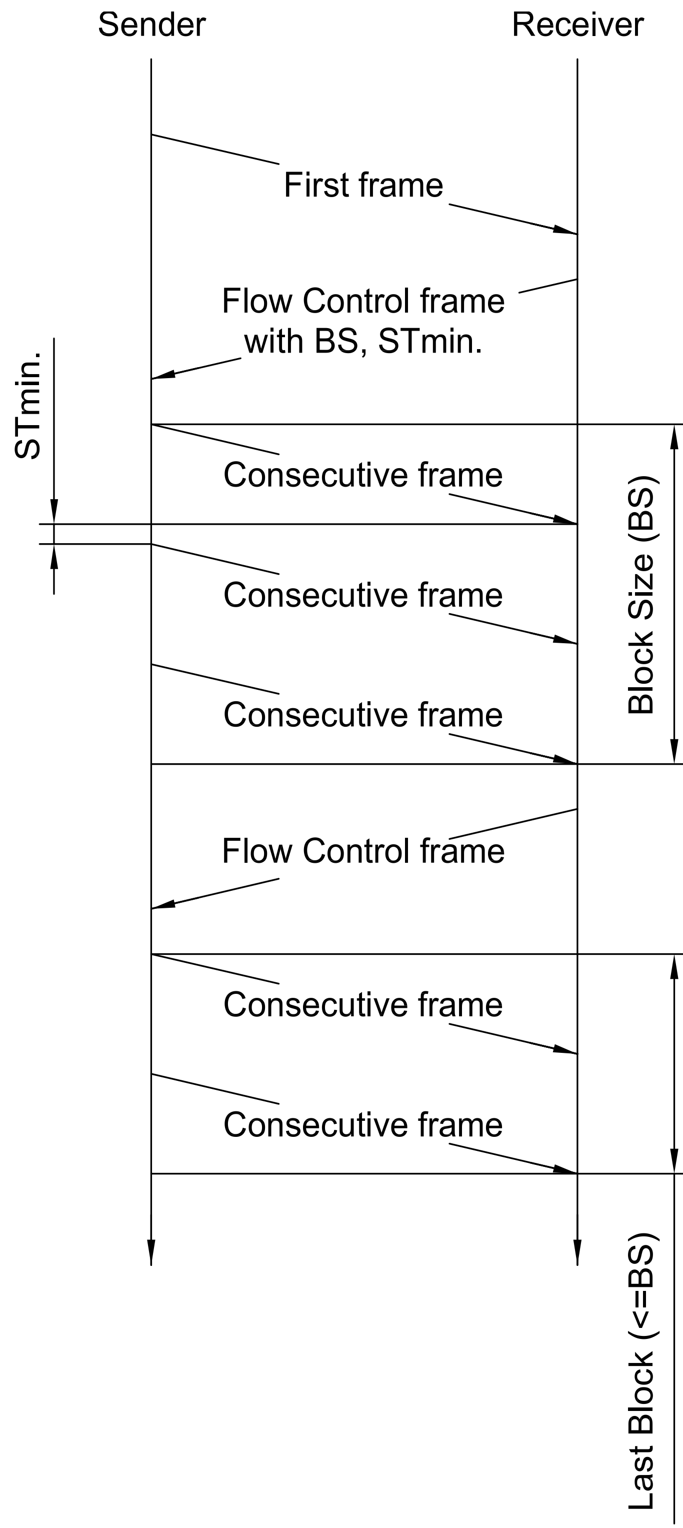



Abbildung 2.5: Prinzip der Kommunikation bei ISO 15765-2 [27]

Im Unterschied zu den gezeigten CAN Transportprotokollen wird beim SAE Standard J1939-21 [56] das Etablieren der Verbindung und die Übertragung der Datentelegramme auf verschiedenen CAN Identifiern realisiert (siehe Beispiel auf Seite 29). Dabei werden nach einer im Rahmen von J1939 vorgegebenen Definition die 29 Bit CAN Identifier gemäß den miteinander kommunizierenden Steuergeräten berechnet.
Durch die Sequenznummer im jeweils ersten Byte eines Datentelegramms ist die Länge der segmentierten PDU bei SAE J1939-21 auf 255 * 7 Byte = 1785 Byte beschränkt.

| Transport-protokoll | verbindungs orientiert | disconnect möglich | halb-duplex | maximale PDU Länge | zeitlicher Frameabstand |
|---|---|---|---|---|---|
| Bosch MCNet | Ja | Nein | Nein | $\infty$[1] | 0 - 100 ms |
| VAG TP1.6 | Ja | Ja | Ja | $\infty$[1] | 0 - 6300 ms |
| VAG TP2.0 | Ja | Ja | Nein | $\infty$[1] | 0 - 6300 ms |
| ISO 15765-2 | Nein | Nein | Nein | 4095 | 0 - 127 ms |
| J1939 (CON[2]) | Ja | Ja | Nein | 1785 | 0 - 1250 ms |
| J1939 (BAM[3]) | Nein | Nein | Nein | 1785 | 50 - 200 ms |

Tabelle 2.2: Betrachtete CAN Transportprotokolle im Vergleich

Zusammenfassend bieten alle fünf betrachteten CAN Transportprotokolle die Möglichkeit, über den CAN Bus Informationen zu übertragen, deren Länge über die in einem CAN Frame vorhandenen 8 Bytes hinausgeht. Die Unterschiede in der Implementierung der Transportprotokolle ergeben sich aus den Anforderungen an ...

- die Größe der zur Verfügung stehenden Kommunikationspuffer
- die simultane, bidirektionale Übertragung von Informationen (Voll-Duplex)
- den zeitlichen Frameabstand aufeinanderfolgender Datentelegramme
- die Fehlererkennung bei einer Verletzung von zeitlichen Restriktionen

Als besonders kritisch sind die beiden letzten Punkte für etablierte Mehrbenutzerbetriebssysteme zu bewerten. Zeitliche Anforderungen im Bereich von 100 µs (siehe ISO 15765-2 Transportprotokoll [27]) können nach dem Stand der Technik mit der Auflösung der Betriebssystem-Zeitgeber im Anwendungskontext nicht erfüllt werden. Aus diesem Grund können beim Stand der Technik CAN Transportprotokolle in Mehrbenutzerbetriebssystemen nur mit Echtzeitbetriebssystemen realisiert werden. Dabei wird das CAN Transportprotokoll in der Form eines speziellen Echtzeitanwendungsprogramms oder einer Echtzeit-Task realisiert, welche die zeitlichen Restriktionen des Transportprotokolls erfüllen kann.

[1] Beschränkung der maximalen PDU Größe durch die Anwendung
[2] Connection Oriented
[3] Broadcast Announce Message

### 2.3.3 CAN basierte Kommunikationsprotokoll Standards

#### 2.3.3.1 Flotten-Management-Schnittstelle / J1939

Der SAE Standard J1939 [56] beschreibt die Nutzung der CAN Botschaft für fahrzeugspezifische Informationen wie beispielsweise der Fahrzeuggeschwindigkeit, des Betriebsstundenzählers oder der Fahzeug-Identnummer.

Dazu wird als Teil des 29 Bit CAN Identifiers eine so genannte Parameter Group Number (PGN) definiert, die als eindeutiger Bezeichner eines Fahrzeugsignales verwendet wird.

Der 29 Bit CAN Identifier wird bei J1939 gebildet aus:

| | |
|---|---|
| Steuerinformationen | Priorität (3 Bit), Reserve (1 Bit), Datapage (1 Bit) |
| Parameter Group Number | (16 Bit), bestehend aus PDU Format und PDU Specific |
| Quelladresse | Steuergeräteadresse (8 Bit) |

Die Flotten-Management-Schnittstelle (FMS-Standard) basiert auf dem SAE J1939 Standard und bietet verschiedene in J1939 definierte Dateninhalte auf einer CAN Schnittstelle im Fahrzeug an. An diese mit 250kBit/s konfigurierte CAN Schnittstelle werden digitale Tachografen, On-Board-Units und andere Telematikgeräte angeschlossen.

| CAN Identifier | PGN | J1939 | FMS | Beschreibung |
|---|---|---|---|---|
| 0CF00203 | F002 | x | | Electronic Transmission Controller #1 - ETC1 |
| 0CF00300 | F003 | x | x | Electronic Engine Controller #2 - EEC2 |
| 0CF00400 | F004 | x | x | Electronic Engine Controller #1 - EEC1 |
| 0CFE6CEE | FE6C | x | x | Tachograph - TCO1 |
| 18FEC027 | FEC0 | x | x | Service Information - SERV |
| 18FEC1EE | FEC1 | x | x | High Resolution Vehicle Distance - VDHR |
| 18FEE527 | FEE5 | x | x | Engine Hours, Revolutions - HOURS |
| 18FEE6EE | FEE6 | x | | Time/Date - TD |
| 18FEE927 | FEE9 | x | x | Fuel Consumption (Liquid) - LFC |
| 18FEED27 | FEED | x | | Cruise Control/Vehicle Speed Setup - CCSS |
| 18FEEE00 | FEEE | x | x | Engine Temperature #1 - ET1 |
| 18FEEF00 | FEEF | x | | Engine Fluid Level/Pressure #1 - EFL/P1 |
| 18FEF100 | FEF1 | x | x | Cruise Control/Vehicle Speed - CCVS |
| 18FEF227 | FEF2 | x | | Fuel Economy (Liquid) - LFE |
| 18FEF500 | FEF5 | x | | Ambient Conditions - AMB |

Tabelle 2.3: Ausschnitt der CAN Nachrichten an der FMS Schnittstelle eines MAN Nutzfahrzeugs Baujahr 2004

Die Informationen des Tachografen (Quelladresse 0xEE) mit der Bezeichnung TCO1 enthalten für die Erkennung des Fahrzeugzustandes (Nutzungszeit, Geschwindigkeit, etc.) die entsprechenden Informationen:

| 00FE6C | | | | | | | | PGN Hex |
|---|---|---|---|---|---|---|---|---|
| 65,132 | | | | | | | | PGN |
| 50 ms | | | | | | | | Rep. Rate |
| Data Byte 1 | Data Byte 2 | Data Byte 3 | Data Byte 4 | Data Byte 5 | Data Byte 6 | Data Byte 7 | Data Byte 8 | Byte No |
| 8 7 6 5 4 3 2 1 | 8 7 6 5 4 3 2 1 | 8 7 6 5 4 3 2 1 | 8 7 6 5 4 3 2 1 | | | 8 7 6 5 4 3 2 1 | 8 7 6 5 4 3 2 1 | Bit No. |
| Drive recognize<br>00 = Vehicle motion not detected<br>01 = vehicle motion detected<br>5.2.6.78<br>SPN 1611 | Overspeed<br>00 = No overspeed<br>01 = Overspeed<br>5.2.6.81<br>SPN 1614 | Not used for FMS-Standard could be sent as “not available” or “don´t care” | Direction indicator<br>00 = Forward<br>01 = Reverse<br>5.2.6.85<br>SPN 1619 | Not used for FMS-Standard | Not used for FMS-Standard | Tachogr. vehicle speed<br>1/256 km/h Bit gain<br>0 km/h offset<br>5.2.5.283<br>SPN 1624 | Tachogr. Vehicle speed<br>1/256 km/h Bit gain<br>0 km/h offset<br>5.2.5.283<br>SPN 1624 | Name<br>values<br>values<br>values<br>values<br>SAE ref<br>SPN |
| Driv. 2 working state<br>000 = Rest<br>001 = Driver available<br>010 = Work<br>011 = Drive<br>110 = Error<br>111 = not available<br>5.2.6.77<br>SPN 1613 | Driver 1 card<br>00 = Card not present<br>01= Card present<br>5.2.6.80<br>SPN 1615 | Driver 2 card<br>00 = Card not present<br>01= Card present<br>5.2.6.80<br>SPN 1616 | Tachgraph performance<br>00 = Normal performance<br>01 = Performance analysis<br>5.2.6.84<br>SPN 1620 | | | | | Name<br>values<br>values<br>values<br>values<br>values<br>values<br>values<br>SAE ref<br>SPN |
| Driv. 1 working state<br>000 = Rest<br>001 = Driver available<br>010 = Work<br>011 = Drive<br>110 = Error<br>111 = not available<br>5.2.6.77<br>SPN 1612 | Driv. 1 time rel states<br>0000 = normal<br>0001 = 15 min bef. 4 ½ h<br>0010 = 4 ½ h reached<br>0011 = 15 min bef. 9 h<br>0100 = 9 h reached<br>0101 = 15 min bef. 16 h<br>0110 = 16h reached<br>1110 = Error<br>1111 = not available<br>5.2.6.79<br>SPN 1617 | Driv 2 time rel. states<br>0000 = normal<br>0001 = 15 min bef. 4 ½ h<br>0010 = 4 ½ h reached<br>0011 = 15 min before 9 h<br>0100 = 9 h reached<br>0101 = 15 min bef. 16 h<br>0110 = 16h reached<br>1110 = Error<br>1111 = not available<br>5.2.6.79<br>SPN 1618 | Handling information<br>00 = no handling information<br>01 = handling information<br>5.2.6.83<br>SPN 1621 | | | | | Name<br>values<br>values<br>values<br>values<br>values<br>values<br>values<br>values<br>values<br>SAE ref<br>SPN |
| | | | System event<br>00 = no tachogr. Event<br>01 = tachogr. Event<br>5.2.6.82<br>SPN 1622 | | | | | Name<br>values<br>values<br>SAE ref<br>SPN |

Abbildung 2.6: Tachograf Botschaftsdefinition 0xFE6C (FMS) [8]

In SAE J1939 besteht auch die Möglichkeit Informationen zu übertragen, die die Nutzdatenlänge des CAN Frames überschreiten. Dazu können Nachrichten bis zu einer Länge von 1785 Bytes durch ein Transportprotokollverfahren segmentiert werden. Im Unterschied zu den in Kapitel 2.3.2 beschriebenen Protokolle werden für eine Übertragung in einer Datenrichtung jeweils zwei CAN Identifier verwendet - eine zum Anzeigen einer Datenübertragung und eine weitere für die segmentierte Übertragung der Nutzdaten.

Die Übertragung einer 17 Zeichen umfassenden Fahrzeugidentnummer (VIN) wird als Broadcast Announce Message (BAM) in J1939 wie folgt realisiert:

- Eröffung der Broadcast Announce Message (BAM) mit der Angabe der Datenlänge von 17 (0x11) und der Parameter Group Number EC00 (Eröffnung der VIN)
- Nach 50ms - 200ms Datentelegramm #01 mit der Parameter Group Number EB00. Enthält die Datenbytes 1-7.
- Nach 50ms - 200ms Datentelegramm #02 mit der Parameter Group Number EB00. Enthält die Datenbytes 8-14.
- Nach 50ms - 200ms Datentelegramm #03 mit der Parameter Group Number EB00. Enthält die Datenbytes 15-17. Der Rest ist mit 0xFF aufgefüllt.

```
user@host:~> candump -td -a vcan2
(0.000000)  vcan2  1CECFF00  [8] 20 11 00 03 FF EC FE 00  -  ' .......'
(0.050000)  vcan2  1CEBFF00  [8] 01 57 56 57 5A 5A 5A 31  -  '.WVWZZZ1'
(0.100000)  vcan2  1CEBFF00  [8] 02 4A 5A 33 57 35 32 35  -  '.JZ3W525'
(0.150000)  vcan2  1CEBFF00  [8] 03 39 30 31 FF FF FF FF  -  '.901....'
```

#### 2.3.3.2 CANopen

Das CAN basierte Kommunikationsprotokoll CANopen realisiert auf der Basis von CAN Nachrichten ein objekt-orientiertes Kommunikationsverfahren, mit dem beispielsweise Sensoren und Aktoren über das CAN Netzwerk angesprochen werden können. Im Einzelnen unterscheidet man bei CANopen folgende Kommunikationsobjekttypen:

- Process Data Objects (PDO)
- Service Data Object (SDO)
- Special Function Objects:
  - Synchronization Object (SYNC)
  - Time Stamp Object
  - Emergency Object (EMCY)
- Network Management Objects:
  - NMT Message
  - Boot-Up Object
  - Error Control Object

Dabei werden Process Data Objects (PDOs) für die Übertragung von Objektinformationen zur Laufzeit genutzt. Service Data Objects (SDOs) erlauben den lesenden oder schreibenden Zugriff auf Werte im so genannten Dictionary eines Gerätes. Special Function Objects und Network Management Objects ermöglichen zusätzliche CANopen spezifische Dienste zur Synchronisation, zur Initialisierung oder beispielsweise auch zur Abfrage des Gerätestatus. Als Modi zur Übertragung von Informationen kommen dabei folgende Verfahren in Betracht:

**Producer/Consumer Modell** Ein Produzent erzeugt Information, die von einem oder mehreren Empfängern konsumiert werden. Dabei besteht auch die Möglichkeit, das ein oder mehrere Consumer diese Informationen anfragen.

**Client/Server Modell** Im Unterschied zum Producer/Consumer Modell besteht hierbei eine 1:1 Beziehung zwischen Produzenten und dem Konsumenten der Informationen.

**Master/Slave Modell** Es besteht eine feste Master/Slave Beziehung bei der der Slave nur aufgrund von Anforderungen des Masters kommunizieren darf.

Durch das bei CAN Bus vorhandene Broadcast Verfahren ergeben sich in vorteilhafter Weise bei CANopen die entsprechenden Kommunikationsbeziehungen:

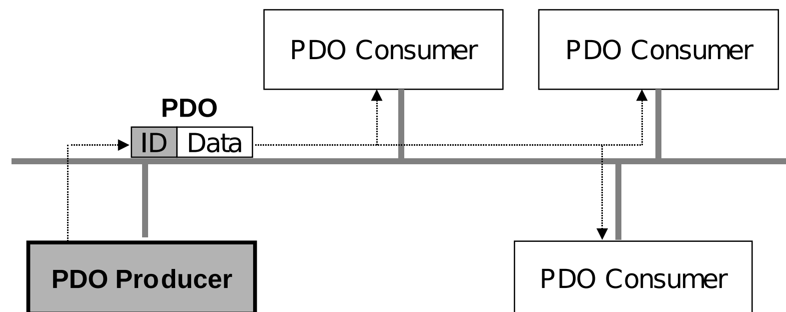



Abbildung 2.7: Beispiel einer PDO Kommunikation bei CANopen [8]

#### 2.3.3.3 CAN-Protokolle für Echtzeitanwendungen

Für industrielle Anwendungsfälle in der Mess- und Regelungstechnik ist ein streng deterministisches Übertragungsverhalten von CAN Nachrichten zu garantieren, was durch die Arbitrierung (siehe Kapitel 2.2.2) nicht ohne weiteres sichergestellt werden kann.

Für die Durchführung einer deterministischen, echtzeitfähigen Kommunikation auf Basis des Controller Area Networks existieren daher verschiedene Protokolle, wie das zum ISO 11898-4 Standard [26] erklärte Time Triggered CAN Verfahren (TTCAN) [20] oder das EDF-basierte Scheduling in [33] und [34], welche eine echtzeitfähige Kommunikation auf dem CAN Bus ermöglichen. Zur Realisierung einer zeitrichtigen Aussendung von CAN Botschaften kann beispielsweise eine Softwarelösung mit Hilfe eines Echtzeitbetriebssystems genutzt werden oder es werden besondere TTCAN Controller eingesetzt, die die Synchronisierung und zeitrichtige Aussendung von CAN Nachrichten im entsprechenden Zeitschlitz durch eine Hardwarelösung sicherstellen. Auch bei einer Unterstützung der Datenübertragung durch einen TTCAN Controller ist die datenverarbeitende Anwendung zur korrekten Bereitstellung von Dateninhalten, bzw. zur Verarbeitung empfangener Inhalte, als eine Echtzeitanwendung mit zugesicherter Rechenzeit zu realisieren.

Ein CAN-basiertes, deterministische Scheduling nach dem TDMA Verfahren kann grundsätzlich auf die CAN-Eigenschaft der Arbitierung verzichten, weil immer nur ein Sender zu einer Zeit das Medium belegen darf. Erlaubt das Protokoll allerdings zeitliche Bereiche, in denen ein arbitrierender Zugriff auf den CAN Bus möglich ist, ist sicherzustellen, dass das Zeitfenster dafür eingehalten wird. Zur Vermeidung von Konflikten mit dem nachfolgenden Zeitfenster kann bei TTCAN-tauglichen CAN Controllern die übliche Wiederholung von CAN Nachrichten bei einem Verlust der Arbitrierung oder sonstigen Kommunikationsfehlern ausgeschaltet werden ('single-shot mode').

### 2.3.4 Definitionsbedarf bei der Verwendung des Controller Area Networks

Im Abschnitt 2.3.1 und 2.3.2 dieses Kapitels wurden Verfahren gezeigt, die sich im Schwerpunkt mit der Verwendung der Nutzdaten einer CAN Nachricht befasst haben. Dabei wurden die Möglichkeiten der Kodierung von Informationen, Verfahren zur Multiplexübertragung und die Segmentierung von 'langen' Nachrichten in die 8 Byte Nutzdaten aufgezeigt. Die in Abschnitt 2.3.3 dargestellten CAN Kommunikationsprotokoll Standards definieren neben der Beschreibung der Nutzdateninhalte beispielsweise auch die Verwendung und ggf. Berechnung des CAN Identifiers zur Datenübertragung, der gleichzeitig zur Priorisierung von Informationen genutzt werden kann (siehe 2.2.2).

Die folgende Aufstellung stellt eine Zusammenfassung des Definitionsbedarfs bei der Verwendung des Controller Area Networks von der Bitübertragungsschicht (OSI-Layer 1) bis zur Anwendungsschicht (OSI-Layer 7) dar. Die Zusammenstellung erhebt keinen Anspruch auf Vollständigkeit.

- Definition der Nutzdateninhalte
  - Signalbeschreibungen für Nutzdaten
  - Segmentierungsverfahren für lange Nachrichten
  - Multiplexverfahren
- Definition der verwendeten Nutzdatenlängen (DLC)
- Definition der verwendeten CAN Identifier
  - Verwendung von 11 Bit oder 29 Bit CAN Identifiern
  - Statische Berechnungsvorschriften für CAN Identifier zur Compilezeit
  - Dynamische Berechnungsvorschriften für CAN Identifier zur Laufzeit
- Definition zeitlicher Anforderungen
  - Sendeseitig: Reduzierung der benötigten Bandbreite
  - Sendeseitig: Einhalten von empfangsseitigen Anforderungen
  - Sendeseitig: Vermeidung von Arbitrierungskonflikten (z.B. bei TTCAN [26])
  - Empfangsseitig: Überwachung und Ausfallerkennung
- Verwendung von Remote Transmission Request Botschaften
- Verwendung bestimmter CAN Controller mit spezifischen Eigenschaften
  - Listen-Mode: Unterdrückung der Aussendung des Acknowledge Bits
  - Single-Shot-Mode: Nur einmalige Übertragung des CAN Frames bei Arbitrierungsverlust (beispielsweise für TTCAN)
  - Time-Triggered-Mode: Mit zeitgesteuerter TTCAN Unterstützung
  - Remote Transmission Request Unterstützung im CAN Controller

- Verfahren für das Netzwerkmanagement
  - zentrales / dezentrales Netzwerkmanagement
  - aktives / passives Netzwerkmanagement
- Physikalische Bitübertragung
  - ISO 11898-2 High Speed Medium Access Unit [24]
  - ISO 11898-3 Low Speed Fault Tolerant Medium Access Unit [25]
  - SAE J2411 Single Wire
  - Eigenentwicklungen für die Bitübertragungsschicht
- Definition bzw. Reduzierung auf bestimmte zu verwendende Bitraten
  - Definition der Bitrate z.B. 83,333 kBit/s, 100 kBit/s, 500 kBit/s
  - Definition der präzisen Bittiming Parameter (Sampling Point, etc.)
- Definition der logischen Bustopologie (Vernetzung der CAN Teilnehmer)
- Definition der physikalischen Bustopologie
  - Bus mit verteilten Abschlußwiderständen
  - Stern mit zentralen Abschlußwiderstand
  - Vorgesehene Leitungslängen
- Definition von Abschlußwiderständen, Hochfrequenzdrosseln und Impedanzen
- Absicherung gegen elektrostatische Entladungen
- Optoelektronische Isolation der CAN Transceiver vom CAN Controller
- Weckfähigkeit der verwendeten CAN Transceiver
  - Definition der maximalen Ruhestromaufnahme
  - Definition der maximalen Zeit bis zur Aufnahme der CAN Kommunikation
- Zu verwendende Steckverbinder und deren elektrische Belegung

Diese aufgezeigten Möglichkeiten zur Definition von CAN basierten Kommunikationsanwendungen vermitteln einen Eindruck über die verschiedenen Einsatzmöglichkeiten des CAN Busses.

Die internationale Anwender- und Herstellervereinigung “CAN in Automation” (CiA, siehe `http://www.can-cia.org`) befasst sich mit der Verbreitung und Standardisierung des Controller Area Networks. Aufgrund der Aktivitäten dieser Organisation existieren beispielsweise für die Berechnung von Bittimings und die Verwendung von Steckverbindern und deren Belegung eindeutige Vorgaben, die von den meisten CAN Hardware Herstellern unterstützt werden. Eine kommerzielle CAN Hardware bietet daher standardmäßig einen High Speed CAN Physical Layer nach ISO 11898-2 und einen 9 poligen SUB-D Stecker mit einer gemäß CiA empfohlenen Beschaltung.

## 2.4 Werkzeuge zur Analyse des Controller Area Networks

In Kapitel 2.3 wurden verschiedene Möglichkeiten zur Nutzung des Controller Area Networks für Anwendungen in eingebetteten Systemen aufgezeigt. Zur Entwicklung der CAN Anwendungen und zur Fehleranalyse im laufenden Betrieb werden Analysewerkzeuge für das Controller Area Network benötigt, die eine besondere Klasse von CAN Anwendungen darstellen. Im Gegensatz zur Benutzung des CAN zur Übertragung von Anwendungsinformationen, dienen CAN Werkzeuge dem Aufzeichnen, Wiedergeben und Analysieren der CAN Kommunikation.

Die CAN Kommunikation kann auf verschiedenen ISO/OSI Schichten analysiert werden:

1. Analyse der Bitübertragungsschicht (ISO/OSI Layer 1)
2. Analyse der Sicherungsschicht (ISO/OSI Layer 2)
4. Analyse der Transportprotokolle (ISO/OSI Layer 4)
7. Analyse der Anwendungsdaten (ISO/OSI Layer 7)

Für die Analyse der Bitübertragungsschicht, z.B. für die Messung der Elektromagnetischen Verträglichkeit (EMV) oder zur Entwicklung von CAN Controllern werden beispielsweise Oszilloskope oder Logik-Analysatoren eingesetzt. Die Analyse der physikalischen Bitübertragung hat keinen Einfluss auf die Programmierschnittstelle des CAN Anwenders und wird daher im Rahmen dieser Arbeit nicht weiter betrachtet.

Bei den Informationen der Sicherungsschicht (Layer 2) wird die CAN Kommunikation auf der Basis der CAN Botschaften analysiert. Zu diesen Analysen zählt beispielsweise:

- Die zeitliche Abfolge von CAN Frames auf dem Übertragungsmedium
- Die Verletzung der Spezifikation zum Aufbau von CAN Frames
- Die Erkennung von Protokollverletzungen durch (andere) CAN Controller
- Der Verlust der Arbitriering des eigenen zu sendenden CAN Frames

Ein Werkzeug zum Aufzeichnen des CAN Datenverkehrs zeichnet mindestens[4] für jede empfangene CAN Nachricht dessen Inhalte, dessen Empfangszeitstempel und die Bezeichnung des jeweiligen CAN Busses auf, auf dem die Nachricht empfangen wurde. Ein Beispiel für ein solches CAN Werkzeug ist das Programm **candump**, das im Anhang in Kapitel C.1.1 auf Seite 210 beschrieben wird.

[4] Abhängig von der verwendeten CAN Hardware und dem verwendeten CAN Werkzeug können bspw. auch Protokollverletzungen und interne Zustände des CAN Controllers mit aufgezeichnet werden.

Eine Aufzeichnung der empfangenen Daten in eine Aufzeichungsdatei (engl. Logfile) ermöglicht eine spätere Analyse der CAN Kommunikation. Aufgrund der gespeicherten Zeitstempel können die Inhalte einer Aufzeichungsdatei zeitrichtig (also mit den gleichen Zeitabständen wie bei der Aufzeichnung) wiedergegeben werden. Ein Programm mit den Grundfunktionalitäten zur Aufzeichnung der CAN Kommunikation wird vom Hersteller einer CAN Hardware üblicherweise zusammen mit dem Hardwaretreiber ausgeliefert.

CAN Werkzeuge zur Analyse von Inhalten aus CAN Nachrichten (Schicht 7) oder CAN Transportprotokollen (Schicht 4) sind stark abhängig von den jeweiligen CAN Protokollen und CAN Anwendungen, wie sie in Kapitel 2.3 beschrieben wurden. Beispiele:

- Ein Werkzeug zur Analyse einer CANopen Kommunikation dekodiert die CANopen spezifischen Informationen der empfangenen CAN Nachrichten.
- Ein Werkzeug zur Analyse von Fahrzeuginformationen zeigt gemäß der vorhandenen Signaldefinition die kodierten CAN Werte im Klartext an, beispielsweise 'Geschwindigkeit = 123 km/h' oder 'Aussentemperatur = -23°C'.
- Die Analyse von Abläufen in der Fahrzeugdiagnose erfordert die Dekodierung der Diagnosebefehle sowie die Kenntnis des zugrundeliegenden CAN Transportprotokolls, über welches die Diagnosebefehle übertragen werden.

## 2.5 Zusammenfassung

In diesem Kapitel wurden verschiedene Anwendungs- und Verwendungsmöglichkeiten für das Controller Area Network aufgezeigt, um einen Teil der Anforderungen an Programmierschnittstellen für das Controller Area Network in Mehrbenutzerbetriebssystemen transparent zu machen.

Dabei wurden neben der unterschiedlichen Verwendung der CAN Botschaft als Kommunikationselement (Fahrzeug/J1939/CANopen) auch Verfahren des Netzwerkmanagements in Kraftfahrzeugen mit Wake-On-CAN Funktionalitäten beschrieben. Die Realisierung von CAN Transportprotokollen und die zeitgesteuerten, CAN-basierten Kommunikationsververfahren begründen zeitliche Anforderungen an Anwortzeiten und das zeitrichtige Aussenden von CAN Nachrichten. Zeitgesteuerte Kommunikationsverfahren wie TTCAN erfordern zusätzlich geeignete CAN Controller und ein echtzeitfähiges Betriebssystem für eine zeitgerechte Verarbeitung der Anwendungsdaten.

# 3 Stand der Technik für den Zugriff auf das Controller Area Network

Der Zugriff auf den CAN Bus erfolgt aus der Sicht eines Anwendungsprogrammierers über eine so genannte Programmierschnittstelle (engl. Application Programmers Interface, kurz API). Diese Programmierschnittstelle definiert die Möglichkeiten des Anwendungsprogrammierers, welche bei unterschiedlichen CAN Hardware - und Softwareanbietern sowohl in der Darstellung der verwendeten Datenstrukturen als auch im Umfang der unterstützten Funktionalitäten variieren. Ein Wechsel zu einem anderen CAN (Hardware/Software-) Anbieter erfordert die Anpassung einer vorhandenen CAN Anwendung an die jeweilige herstellerspezifische Programmierschnittstelle, deren langfristige Stabilität nicht sichergestellt ist.

Dieses Kapitel gibt einen Überblick zu CAN Hardware, CAN Treibermodellen und verwendeten Programmierschnittstellen für das Controller Area Network in Mehrbenutzerbetriebssystemen. In diesem Überblick werden die grundsätzlichen Techniken für den Zugriff auf den CAN Bus aus der Sicht des Anwendungsprogrammierers dargestellt. Unter anderem werden proprietäre Erweiterungen beschrieben, die beispielsweise eine Verwendung virtueller CAN Hardware ermöglichen oder auch den parallelen Zugriff verschiedener CAN Anwendungen auf den CAN Bus realisieren.

Der Zugriff auf den CAN Bus erfordert zwingend eine entsprechende CAN Hardware, die an einen Prozessor angebunden ist auf dem die CAN Anwendung ausgeführt wird. Ein integraler Bestandteil dieser CAN Hardware ist der so genannte CAN Controller, welcher im folgenden Abschnitt 3.1 in seiner Funktionalität dargestellt werden soll.

## 3.1 CAN Controller

Ein CAN Controller stellt den Inhalt eines empfangenen CAN Bitstroms in Nachrichtenpuffern zur Verfügung, so dass diese von einem Prozessor ausgelesen und verarbeitet werden können. Der CAN Controller übernimmt dabei die Bitsynchronisierung, die Fehlererkennung und Fehlerbehandlung sowie weitere nach der CAN Spezifikation erforderliche Aufgaben. Analog zu einem Controller für einen seriellen Schnittstellenbaustein wird nach dem Empfang einer gültigen CAN Botschaft dessen Dateninhalt in Empfangsregistern bereitgestellt und eine Unterbrechungsanforderung (engl. Interrupt) beim Prozessor ausgelöst. Der Prozessor liest daraufhin die entsprechenden Empfangsregister aus und setzt die Unterbrechungsanforderung für den Empfang von weiteren CAN Botschaften zurück.

Soll eine CAN Botschaft gesendet werden, schreibt der Prozessor die Inhalte der zu sendenden CAN Botschaft in die Senderegister des CAN Controllers und startet die Übertragung. Nach einer erfolgreichen Übertragung (oder auch im Fall eines Übertragungsfehlers) löst der CAN Controller eine Unterbrechungsanforderung beim Prozessor aus, worauf dieser weitere Daten in den Controller schreiben oder eine Fehlerbehandlung durchführen kann. Siehe beispielsweise [46].

CAN Controller sind als so genannte diskrete Bauelemente in einem separaten Gehäuse oder auch in Form von System-On-Chip[1] Komponenten verfügbar, bei der die CAN Controller Logik direkt auf den Prozessor oder in einem FPGA[2] integriert wird. Aus Sicht eines Software Entwicklers ist dieser Unterschied in der Hardware-Ausführung nicht notwendigerweise erkennbar.

Zur Verdeutlichung der Unterschiede von CAN Controllern werden auf den folgenden Seiten zwei CAN Controller vorgestellt, die als diskrete (engl. Stand-Alone) Bauelemente in verschiedenen CAN Adaptern eine weite Verbreitung gefunden haben.

[1] Ein integrierter Schaltkreis, der zusätzlich zum Prozessor-Kern mit verschiedenen Schaltkreisen zum direkten Anschluss externer Peripherie (bspw. Speichergeräten) ausgestattet ist.

[2] Field Programmable Gate Array. Ein integrierter Schaltkreis, in den eine logische Schaltung - beispielsweise ein CAN Controller - programmiert werden kann.

Abbildung 3.1 zeigt das Blockschaltbild des verbreiteten SJA1000 CAN Controllers der Firma NXP (vormals Philips Semiconductors). Über das in der Zeichnung oben links beschriebene Host-Interface, können die Register des SJA1000 vom Prozessor angesteuert werden.

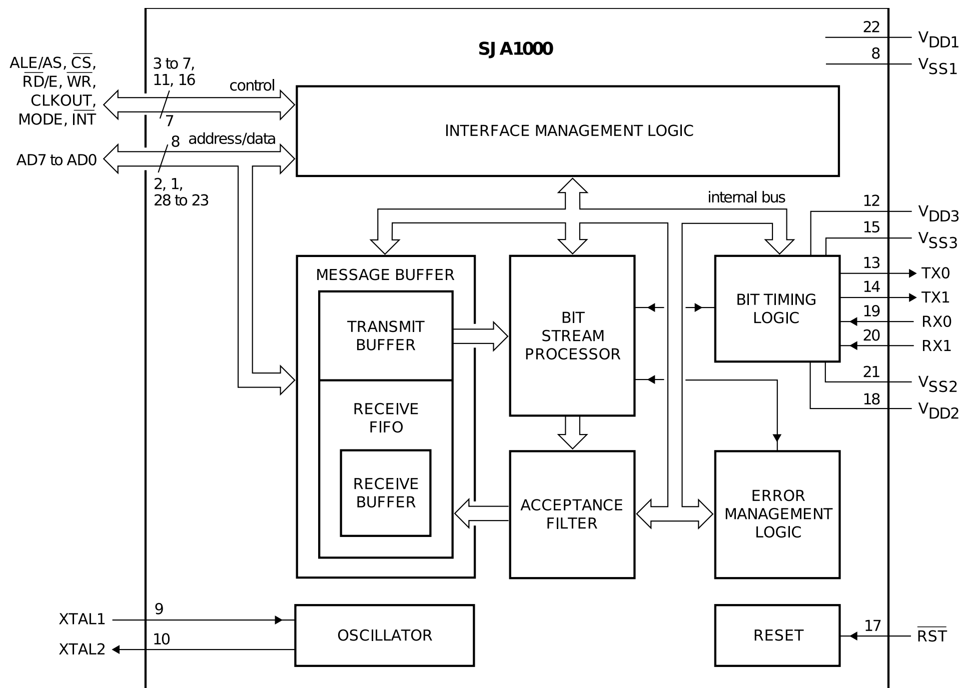



Abbildung 3.1: Blockschaltbild des SJA1000 CAN Controllers [46]

Eine weitere Möglichkeit der Anbindung des CAN Controllers an den Prozessor besteht in der Verwendung von Dual-Ported-RAM (DPRAM). Dabei handelt es sich aus der Sicht des Anwenders um einen gemeinsamen Speicherbereich, der sowohl vom CAN Controller als auch vom Prozessor gelesen und beschrieben werden kann. Bei einem DPRAM muss allerdings der konkurrierende Zugriff auf die gemeinsamen Speicherbereiche gelöst werden, um eine Inkonsistenz der Daten zu vermeiden.

Stehen an einem Prozessor keine ausreichenden Anschlussleitungen für die oben genannten Verfahren (Prozessorbus/DPRAM) zur Verfügung, besteht auch die Möglichkeit, einen CAN Controller über das Vier-Draht Master-Slave Kommunikationsprotokoll SPI [38] anzubinden, wie es beispielsweise beim Microchip MCP2515 [36] realisiert ist.

Neben den unterschiedlichen Möglichkeiten, CAN Controller an einen Prozessor anzubinden, unterscheiden sich CAN Controller auch im Funktionsumfang und im Layout der zur Verfügung gestellten Register (siehe Abbildung 3.2).

Der Intel i82527 [21] realisiert in einem Adressraum von 256 Registern 15 so genannte 'Message Objects', die zum Empfang *oder* zum Versenden von CAN Botschaften verwendet werden können. Dabei unterstützt der i82527 auch das direkte Beantworten von RTR Frames auf Controller Ebene - dafür sind ein Empfang und eine Verarbeitung von RTR Frames durch den Prozessor nicht möglich.

Der NXP SJA1000 [46] realisiert in einem Adressraum von 32 Registern nur jeweils einen Registersatz zum Senden oder Empfangen von CAN Botschaften und überlässt die Verarbeitung von RTR Frames generell dem angeschlossenen Prozessor.

| CAN ADDRESS | SEGMENT | OPERATING MODE | | RESET MODE | |
|---|---|---|---|---|---|
| | | READ | WRITE | READ | WRITE |
| 0 | control | control | control | control | control |
| 1 | | (FFH) | command | (FFH) | command |
| 2 | | status | – | status | – |
| 3 | | interrupt | – | interrupt | – |
| 4 | | (FFH) | – | acceptance code | acceptance code |
| 5 | | (FFH) | – | acceptance mask | acceptance mask |
| 6 | | (FFH) | – | bus timing 0 | bus timing 0 |
| 7 | | (FFH) | – | bus timing 1 | bus timing 1 |
| 8 | | (FFH) | – | output control | output control |
| 9 | | test | test; note 2 | test | test; note 2 |
| 10 | transmit buffer | identifier (10 to 3) | identifier (10 to 3) | (FFH) | – |
| 11 | | identifier (2 to 0), RTR and DLC | identifier (2 to 0), RTR and DLC | (FFH) | – |
| 12 | | data byte 1 | data byte 1 | (FFH) | – |
| 13 | | data byte 2 | data byte 2 | (FFH) | – |
| 14 | | data byte 3 | data byte 3 | (FFH) | – |
| 15 | | data byte 4 | data byte 4 | (FFH) | – |
| 16 | | data byte 5 | data byte 5 | (FFH) | – |
| 17 | | data byte 6 | data byte 6 | (FFH) | – |
| 18 | | data byte 7 | data byte 7 | (FFH) | – |
| 19 | | data byte 8 | data byte 8 | (FFH) | – |
| 20 | receive buffer | identifier (10 to 3) | identifier (10 to 3) | identifier (10 to 3) | identifier (10 to 3) |
| 21 | | identifier (2 to 0), RTR and DLC | identifier (2 to 0), RTR and DLC | identifier (2 to 0), RTR and DLC | identifier (2 to 0), RTR and DLC |
| 22 | | data byte 1 | data byte 1 | data byte 1 | data byte 1 |
| 23 | | data byte 2 | data byte 2 | data byte 2 | data byte 2 |
| 24 | | data byte 3 | data byte 3 | data byte 3 | data byte 3 |
| 25 | | data byte 4 | data byte 4 | data byte 4 | data byte 4 |
| 26 | | data byte 5 | data byte 5 | data byte 5 | data byte 5 |
| 27 | | data byte 6 | data byte 6 | data byte 6 | data byte 6 |
| 28 | | data byte 7 | data byte 7 | data byte 7 | data byte 7 |
| 29 | | data byte 8 | data byte 8 | data byte 8 | data byte 8 |
| 30 | | (FFH) | – | (FFH) | – |
| 31 | | clock divider | clock divider; note 3 | clock divider | clock divider |

Registerbelegung SJA1000 (BasicCAN)

82527 Address Map

| Address | Register |
|---|---|
| 00H | Control Register |
| 01H | Status Register |
| 02H | CPU Interface Reg. |
| 03H | Reserved |
| 04–05H | High Speed Read |
| 06–07H | Global Mask - Standard |
| 08–0BH | Global Mask - Extended |
| 0C–0FH | Message 15 Mask |
| 10–1EH | Message 1 |
| 1FH | CLKOUT Register |
| 20–2EH | Message 2 |
| 2FH | Bus Config. Reg. |
| 30–3EH | Message 3 |
| 3FH | Bit Timing Reg. 0 |
| 40–4EH | Message 4 |
| 4FH | Bit Timing Reg. 1 |
| 50–5EH | Message 5 |
| 5FH | Interrupt Register |
| 60–6EH | Message 6 |
| 6FH | Reserved |
| 70H–7EH | Message 7 |
| 7FH | Reserved |
| 80–8EH | Message 8 |
| 8FH | Reserved |
| 90–9EH | Message 9 |
| 9FH | P1CONF |
| A0–AEH | Message 10 |
| AFH | P2CONF |
| B0–BEH | Message 11 |
| BFH | P1IN |
| C0–CEH | Message 12 |
| CFH | P2IN |
| D0–DEH | Message 13 |
| DFH | P1OUT |
| E0–EEH | Message 14 |
| EFH | P2OUT |
| F0–FEH | Message 15 |
| FFH | Serial Reset Address |

Message Object Structure

| Base Address | |
|---|---|
| a 0 | Control 0 |
| a 1 | Control 1 |
| a 2 | Arbitration 0 |
| a 3 | Arbitration 1 |
| a 4 | Arbitration 2 |
| a 5 | Arbitration 3 |
| a 6 | Mess. Conf. |
| a 7 | Data 0 |
| a 8 | Data 1 |
| a 9 | Data 2 |
| a 10 | Data 3 |
| a 11 | Data 4 |
| a 12 | Data 5 |
| a 13 | Data 6 |
| a 14 | Data 7 |

Registerbelegung i82527

Abbildung 3.2: Register Layout bei CAN Controllern (Beispiele)

Schon diese einfache Gegenüberstellung zweier CAN Controller zeigt die zu erwartenden Probleme bei der Realisierung einer wiederverwendbaren CAN Anwendung. Durch eine ggf. vorhandene 'lokale Intelligenz' des CAN Controllers muss im obigen Beispiel für die Behandlung von Remote Transmission Requests (RTR) ein unterschiedliches Verfahren in der Software realisiert werden. Eigenschaften eines CAN Controllers, die für den Programmierer einer eingebetteten Anwendung sinnvoll und praktisch erscheinen, können eine Abstraktion in der Form eines generischen CAN Treibers erheblich erschweren.

Zu diesen möglichen Unterschieden im Verhalten von CAN Controllern zählt auch die Behandlung von Kommunikationsfehlern auf dem CAN Bus. Üblicherweise initialisiert der Prozessor den CAN Controller nach dem Übergang des Controllers in den so genannten Bus-Off Status (bspw. beim SJA1000 / i82527). Andere CAN Controller - wie beispielsweise der MCP2515 von Microchip oder MSCAN von Freescale - schalten nach einem erkannten Bus-Off Status selbstständig in den Modus, der eine Wiederaufnahme der CAN Kommunikation ermöglicht.

## Zusammenfassung der Funktionalitäten von CAN Controllern

- CAN Controller setzen den seriellen Bitstrom des CAN Busses in Registerinhalte um, die von einem Prozessor verarbeitet werden können.
- CAN Controller führen die Fehlererkennung und Fehlerbehandlung auf dem CAN Bus selbstständig durch.
- Die Zeitlogik für das Verarbeiten von Bitströmen und die Einstellung der Bitrate zur Kommunikation wird im CAN Controller durchgeführt.
- Es existieren so genannte Stand-Alone CAN Controller, die über verschiedene Schnittstellen an den Prozessor angebunden werden können, beispielsweise
  - linear im Adressraum des Prozessors
  - multiplex im Adressraum des Prozessors (spart Adressraum)
  - Dual Ported RAM im Adressraum des Prozessors
  - Serielle Kommunikationsprotokolle (SPI, UART)
- Das Register Layout der CAN Controller ist herstellerabhängig und die Nutzung der Register zur Ausführung von Sende- und Empfangsoperationen unterschiedlich.
- Es existieren CAN Controller, die in eingebetteten Prozessoren als Hardwarekomponenten (wie serielle Schnittstellen) integriert sind. Solche CAN Controller sind üblicherweise linear im Adressraum des Prozessors sichtbar.
- Die vom CAN Controller zur Verfügung gestellte Funktionalität ist herstellerabhängig und unterschiedlich, beispielsweise
  - Verhalten bei der Verarbeitung von Remote Transmission Requests
  - Verhalten bei der Behandlung von Kommunikationsfehlern (Bus-Off)
  - Möglichkeit der Aussendung von CAN Frames ohne Wiederholung im Fehlerfall bzw. bei Verlust des Senderechts bei der bitweisen Arbitrierung (Single Shot) - dieses ist eine Voraussetzung für Time-Triggered-CAN [26].
  - Möglichkeit zum Empfang der eigenen Nachricht innerhalb des CAN Controllers zu Testzwecken (Loopback).
  - Möglichkeit zum Lesen vom CAN Bus ohne eigene Aussendung des dominanten Acknowledge Bits (Listen Mode).
- CAN Controller bieten unterschiedliche Möglichkeiten zum Filtern einzelner CAN Identifier oder Gruppen von CAN Identifiern, die ein Verwerfen nicht benötigter CAN Botschaften bereits beim Empfang durch den CAN Controller realisieren.

## 3.2 Aktive und passive CAN PC Schnittstellenkarten

Im Abschnitt 3.1 wurde auf verschiedene Ausführungen von CAN Controllern und ihre Anbindung an den jeweiligen Prozessor eingegangen. Dabei wurde jedoch nicht betrachtet, unter welchen Randbedingungen eine Verarbeitung der CAN Nachrichten auf dem Prozessor stattfindet. Wenn auf dem CAN Controller nur ein Registersatz zum Auslesen einer einzelnen CAN Botschaft vorhanden ist, muss die Unterbrechungsanforderung (engl. Interrupt) so zeitnah vom Prozessor bearbeitet werden, dass auch bei maximaler Buslast keine CAN Botschaften verloren gehen.

Mit der Einführung von Mehrbenutzerbetriebssystemen (siehe Seite 3) sind Fragestellungen zur Interrupt Latenzzeit dieser Systeme und die Verarbeitung von CAN Nachrichten in Personal Computern im Allgemeinen in den Fokus gekommen. So wird in dem Artikel "Eine Frage der Treiber-Software" von Frank Förster (Fa. IXXAT) [10] die Motivation für die Einführung einer *aktiven* CAN Schnittstellenkarte für Personal Computer sehr deutlich:

| | Nutzdaten/Nachricht 1Byte | | Nutzdaten/Nachricht 4 Bytes | | Nutzdaten/Nachricht 8 Bytes | |
|---|---|---|---|---|---|---|
| Max Nachrichten /s | ~18.000 | | ~12.000 | | ~8.400 | |
| Zeitabstand zwischen zwei Messages | 55 µs | | 83 µs | | 119 µs | |
| | Passiv SJA 1000 CAN Controller | Aktiv iP CI-165/PCI | Passiv SJA 1000 CAN Controller | Aktiv iPCI-165/PCI | Passiv SJA 1000 CAN Controller | Aktiv iPCI-165/PCI |
| Anzahl der speicherbaren empfangener Messages | 16 | 250 | 9 | 250 | 5 | 250 |
| Dauer bis Datenbuffer überläuft | 880 µs | 13,7 ms | 747 µs | 20,7 ms | 595 µs | 29,7 ms |

Abbildung 3.3: Mögliche Latenzzeiten bei CAN Schnittstellenkarten [10]

Wie aus der Abbildung 3.3 ersichtlich wird, ist die Anforderung an den Anwendungsrechner abhängig von der Ausführung der verwendeten CAN Hardware. Bei der Verwendung einer *passiven* CAN Hardware müssen der Anwendungsrechner und dessen Betriebssystem eine zeitnahe Verarbeitung der eingehenden CAN Nachrichten sicherstellen, um den Verlust von empfangenen CAN Nachrichten zu vermeiden.

Die Anforderungen an die zeitnahe Verarbeitung eingehender CAN Nachrichten sind bei der Verwendung einer *aktiven* CAN Hardware des genannten Herstellers aufgrund eines Empfangspuffers von 250 CAN Nachrichten erheblich geringer: Im schlechtesten Fall kann der Anwendungsrechner 15 mal langsamer auf eingehende Nachrichten reagieren, als mit einer *passiven* CAN Hardware - ohne dabei einen Datenverlust zu riskieren.

*Aktive* CAN Schnittstellenkarten zeichnen sich durch einen separaten, eingebetteten Prozessor aus, der einen oder mehrere CAN Bus Controller zeitnah bedient. Die CAN Nachrichten werden zusammen mit einem Empfangszeitstempel zumeist in einem Dual-Ported-RAM (DPRAM) abgelegt, von wo sie durch den Treiber im Personal Computer ausgelesen und einer weiteren Verarbeitung zugeführt werden. Der separate, eingebettete Prozessor verfügt dabei über einen eigenen Speicher und eine eigene Betriebssoftware (Firmware), die in speziellen Anwendungsfällen auch höhere CAN Anwendungsprotokolle unterstützen kann, wie beispielsweise CANopen [5] (siehe Kapitel 2.3.3.2).

Die nicht standardisierte Programmierschnittstelle für den Anwendungsprogrammierer ändert sich dadurch allerdings massiv, weil nun auf OSI Schicht 7 (Anwendungsschicht) mit der CAN Karte interagiert werden muss.

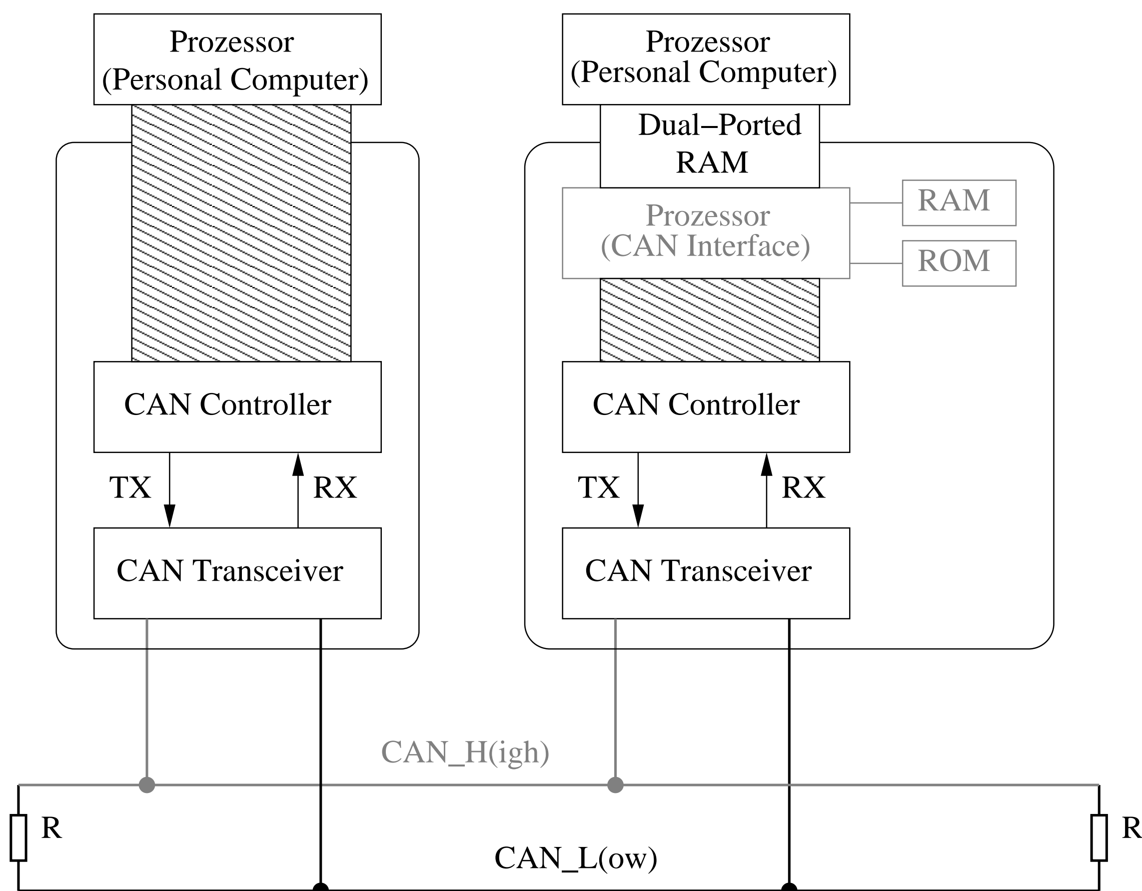



Abbildung 3.4: *Passive* (links) und *aktive* CAN Hardware

Aufgrund des erheblich höheren Entwicklungsaufwandes für *aktive* CAN Schnittstellenkarten sind diese im Allgemeinen in einem gehobenen Preissegment angesiedelt.

Der Entwicklungsaufwand umfasst die Definition der herstellerspezifischen Inhalte des Dual-Ported-RAM und die Entwicklung eines sicheren Kommunikationsverfahrens über das DPRAM mit entsprechenden Locking-Mechanismen zur Vermeidung eines Datenverlustes. Zusätzlich muss ein entsprechender CAN Treiber auf der PC Seite ggf. für verschiedene Betriebssysteme realisiert werden, der mit dieser *aktiven*, herstellerspezifischen CAN Hardware funktioniert.

| Eigenschaft | *Aktive* CAN Hardware | *Passive* CAN Hardware |
|---|---|---|
| Antwortverhalten der Anwendung | weniger kritisch | sehr kritisch |
| Extra Firmware nötig | ja | nein |
| Performanz Hostsystem abhängig | nein | ja |
| Zeitstempel lastabhängig | nein | ja |
| Höhere CAN Protokolle | möglich | nein |
| Spezifikation zur Treibererstellung | herstellerabhängig | offen (Controller) |

Tabelle 3.1: Konzeptvergleich: *Aktive* und *passive* CAN Hardware

## 3.3 Gerätetreiber für CAN PC Schnittstellenkarten

Analog zu Gerätetreibern für Festplatten, Grafikadaptern oder serielle Schnittstellen wird für die Anbindung von CAN Hardware eine Treibersoftware benötigt. Diese Treibersoftware wird beispielsweise bei Windows oder Linux im so genannten Betriebssystemkontext ausgeführt, was den effizienten Zugriff auf Schnittstellen und Ressourcen innerhalb des Betriebssystems ermöglicht. Fehlerhafte Treiber im Betriebssystemkontext bergen allerdings die Gefahr, dass das Betriebssystem instabil wird oder terminiert.

Der Zugriff auf die jeweilige CAN Hardware aus dem Anwendungskontext wird über eine Betriebssystemschnittstelle realisiert, die der CAN Treiber nach seiner erfolgreichen Initialisierung bereitstellt. Die verschiedenen Ausprägungen von Betriebssystemschnittstellen für den Zugriff auf CAN Hardware werden ab Kapitel 3.4 dargestellt.

In diesem Abschnitt sollen die grundsätzlichen Probleme bei der Verwendung herstellerspezifischer Treiber für unterschiedliche, kommerzielle CAN Hardware aufgezeigt werden. Für die auf den ersten Blick sehr ähnlichen Anforderungen an eine CAN Schnittstellenkarte - dem sicheren Senden und Empfangen von CAN Botschaften - existieren verschiedene, zueinander inkompatible Lösungen.

Als *passive* PCI CAN Schnittstellenkarten auf Basis zweier SJA1000 CAN Controller bieten sich beispielsweise folgende Produkte an:

| Hersteller | Bezeichnung | Preisangabe[1] |
|---|---|---|
| PEAK-System Technik GmbH | PCAN PCI | ca. 245 € |
| Kvaser AB | PCIcanx HS/HS | ca. 500 € |
| IXXAT Automation GmbH | PC-I 04/PCI | ca. 250 € |
| EMS Dr. Thomas Wünsche e.K. | CPC-PCI | (auf Anfrage) |
| ESD Electronic System Design GmbH | CAN-PCI/266 | ca. 250 € |

Diese fünf CAN Karten besitzen verschiedene eindeutige PCI Hersteller Identifier, werden geringfügig anders initialisiert und blenden die Register des SJA1000 Controllers linear in den Adressraum des Personal Computers ein. Einen erkennbaren funktionalen Unterschied gibt es weder aus technischer Sicht noch aus der Sicht des Anwenders.

Jeder der genannten Hersteller bietet zu seiner CAN Hardware einen CAN Treiber (jeweils für Windows und Linux) an. Dieser Treiber bietet dem CAN Anwender zwar nur *eine* herstellerspezifische, proprietäre Programmierschnittstelle - unterstützt dafür aber *alle* CAN Schnittstellenkarten des jeweiligen Herstellers. Ein Wechsel zu einer CAN Hardware eines anderen Herstellers macht aufgrund der unterschiedlichen Treiberschnittstelle Änderungen an der CAN Anwendung erforderlich. Wie bereits in Kapitel 1 beschrieben, bindet sich der CAN Anwender bereits mit einen solchen, technisch einfach zu realisierenden CAN Produkt fest an einen CAN Hardware Hersteller.

Für die jeweiligen Hersteller der genannten CAN Produkte erzwingt eine Änderung der Schnittstellen innerhalb des Betriebssystems die Anpassung des zugehörigen CAN Treibers. Die Änderung solcher Schnittstellen erfolgt beispielsweise bei der Auslieferung einer neuen Version des Betriebssystems (bspw. Windows XP → Windows Vista) und erfordert eine ständige Wartung und Weiterentwicklung der Treibersoftware.

Die Situation bei *aktiver* CAN Hardware stellt sich zum Zeitpunkt der Markterhebung im März 2009 ähnlich dar. Die *aktiven* CAN Schnittstellenkarten werden mit unterschiedlichen Prozessoren realisiert, was für den CAN Anwender zunächst keinen funktionalen Unterschied darstellt. Verschiedene *aktive* Schnittstellenkarten bieten allerdings die Möglichkeit, auf dem integrierten (eingebetteten) Prozessor CAN Anwendungsprotokolle zu unterstützen. Durch eine Realisierung eines CAN Transportprotokolls (siehe Kapitel 2.3.2) auf dem integrierten Prozessor kann der Anwendungsentwickler in seiner Anwendung auf die Einhaltung zeitlicher Restriktionen für die CAN Transportprotokoll-Kommunikation weitgehend verzichten. Wie in Kapitel 3.2 bereits erläutert, sind die Programmierschnittstellen für die OSI Schichten 4 und 7 herstellerabhängig. Dadurch ist eine CAN Anwendung, die die 'integrierte Intelligenz' einer *aktiven* CAN Hardware nutzt, wieder fest an den entsprechenden CAN Hardware Hersteller gebunden.

[1]Angaben zu Produkten und Preisen ermittelt auf der Fachmesse "Embedded World", März 2009

## 3.4 CAN Programmierschnittstellen unter Linux

Unix-ähnliche Betriebssysteme wie BSD Unix, Solaris oder Linux ermöglichen den Zugriff auf die Hardwareressourcen über besondere Gerätedateien (engl. device files), die im Dateisystem mit speziellen Systemaufrufen erzeugt werden.

Die Gerätedateien werden üblicherweise in dem Verzeichnis `/dev` erzeugt und ermöglichen durch ein systemweit bekanntes Namensschema den Zugriff auf die hinterlegten Hardwareressourcen.

```
user@host:/dev> ls -l
crw-rw-rw- 1 root root      1,   3 23. Mai 2009  null
crw-rw---- 1 root root     10, 135 23. Mai 2009  rtc
brw-rw---- 1 root disk      8,   0 23. Mai 2009  sda
brw-rw---- 1 root disk      8,   1 23. Mai 2009  sda1
brw-rw---- 1 root disk      8,   2 23. Mai 2009  sda2
crw-rw---- 1 root root      4,   0 23. Mai 2009  tty0
```

| Type | User | Group | All | Links | Owner | Group | Major | Minor | Date | Name |
|---|---|---|---|---|---|---|---|---|---|---|
| c | rw- | rw- | rw- | 1 | root | root | 1 | 3 | 23. Mai 2009 | null |
| c | rw- | rw- | --- | 1 | root | root | 10 | 135 | 23. Mai 2009 | rtc |
| b | rw- | rw- | --- | 1 | root | disk | 8 | 0 | 23. Mai 2009 | sda |
| b | rw- | rw- | --- | 1 | root | disk | 8 | 1 | 23. Mai 2009 | sda1 |
| b | rw- | rw- | --- | 1 | root | disk | 8 | 2 | 23. Mai 2009 | sda2 |
| c | rw- | rw- | --- | 1 | root | root | 4 | 0 | 23. Mai 2009 | tty0 |

Tabelle 3.2: Beispiel für Gerätedateien in Unix Dateisystemen

Zusätzlich zu den Einträgen, die die Eigentums- und Zugriffsrechte sowie das Zugriffsdatum für die Datei beschreiben, sind für Gerätedateien der Gerätetyp sowie die so genannten Major und Minor Nummern relevant. Die Major und Minor Nummern sind betriebssystemintern auf einen 8-Bit Wert beschränkt und können jeweils Werte von 0 bis 255 annehmen.

Der Gerätetyp gibt im Beispiel an, ob es sich um ein zeichenorientiertes Gerät (`character`) oder ein blockorientiertes Gerät (`block`) handelt. Beispiele für blockorientierte Geräte sind Festplatten, CD-Laufwerke, USB-Sticks oder Bandlaufwerke, deren kleinste Dateneinheit ein Datenblock einer bestimmten Größe (z.B. 512 Byte) darstellt. Die kleinste Dateneinheit von zeichenorientierten Geräten ist ein einzelnes Zeichen (1 Byte). Zeichenorientierte Geräte werden beispielsweise für serielle Schnittstellen und verschiedene CAN Treiber nach dem Stand der Technik eingesetzt.

Die Major und Minor Nummern stellen innerhalb des Betriebssystems die Verbindung der Gerätedatei zum Gerätetreiber her. In diesem Beispiel hat sich der im Folgenden betrachtete CAN Treiber der Fa. PEAK-System Technik GmbH die Major Nummer 251 registriert und nutzt für die Unterscheidung der verschiedenen CAN Hardwarevarianten (USB:32+x, PCMCIA:40+x, ISA:0+x, ...) die angegebenen Minor Nummern. Es können so maximal *acht* CAN Schnittstellen einer Hardwarevariante angesprochen werden.

```
user@host:/dev> ls -l /dev/pcan*
crw-rw-rw- 1 root root 251,  0 23. Mai 09:31 /dev/pcan0
crw-rw-rw- 1 root root 251,  1 23. Mai 09:31 /dev/pcan1
crw-rw-rw- 1 root root 251, 16 23. Mai 09:31 /dev/pcan16
crw-rw-rw- 1 root root 251, 17 23. Mai 09:31 /dev/pcan17
crw-rw-rw- 1 root root 251, 24 23. Mai 09:31 /dev/pcan24
crw-rw-rw- 1 root root 251, 25 23. Mai 09:31 /dev/pcan25
crw-rw-rw- 1 root root 251, 32 23. Mai 09:31 /dev/pcan32
crw-rw-rw- 1 root root 251, 33 23. Mai 09:31 /dev/pcan33
crw-rw-rw- 1 root root 251, 40 23. Mai 09:31 /dev/pcan40
crw-rw-rw- 1 root root 251, 41 23. Mai 09:31 /dev/pcan41
crw-rw-rw- 1 root root 251,  8 23. Mai 09:31 /dev/pcan8
crw-rw-rw- 1 root root 251,  9 23. Mai 09:31 /dev/pcan9
```

Wie beschrieben stellt die Gerätedatei mit Hilfe der Major und Minor Nummern eine Verbindung zum CAN Treiber dar. Diese Verbindung wird hergestellt, wenn eine Anwendung mit dem Systemaufruf open(2) die Gerätedatei (z.B. `/dev/pcan32`) öffnet. Der von open(2) gelieferte Dateideskriptor kann mit read(2) gelesen und mit write(2) geschrieben werden.

Der Zugriff des Anwendungsprogrammierers auf die CAN Schnittstelle kann auf der Basis einer zeichenorientierten Gerätedatei mit unterschiedlichen Verfahren durchgeführt werden:

1. Direkte Lese- und Schreibzugriffe auf die Gerätedatei (z.B. `/dev/pcan32`)
   a) in der Form von 'lesbaren' Zeichenketten
   b) in der Form von binären Strukturen
2. Unter Verwendung des I/O Control Interface (ioctl)
3. Indirekter Zugriff auf die Gerätedatei über Bibliotheksfunktionen (Libraries)

Jedes dieser Verfahren kann - abhängig vom Anbieter der Treibersoftware - in unterschiedlichen Ausprägungen implementiert sein.

Zum Versenden einer CAN Nachricht mit dem 11-Bit CAN Identifier 0x123, der Datenlänge 2 und den Nutzdaten 0x11 und 0x22 stellt sich die Anwendungsschnittstelle unterschiedlich dar:

**Beispiele für das Zugriffsverfahren 1a:**
(direkter Zugriff auf die Gerätedatei mit 'lesbaren' Zeichenketten)

*PEAK:*[3]
```
echo "m s 0x123 2 0x11 0x22" > /dev/pcan32
```

*CAN232:*[4]
```
echo "t12321122" > /dev/ttyS0
```

**Beispiel für das Zugriffsverfahren 1b:**
(direkter Zugriff auf die Gerätedatei mit binären Strukturen)

*CAN4LINUX:*[5]
```
canmsg_t tx;

tx.id = 0x123;
tx.length = 2;
tx.flags = 0;
tx.data[0] = 0x11;
tx.data[1] = 0x22;

write(fd, tx, sizeof(tx));
```

**Beispiel für das Zugriffsverfahren 2:**
(Zugriff auf die Gerätedatei mit I/O Control Interface)

*PEAK:*[6]
```
TPCANMsg MsgBuff;

MsgBuff.ID = 0x123;
MsgBuff.MSGTYPE = MSGTYPE_STANDARD;
MsgBuff.LEN = 2;
MsgBuff.DATA[0] = 0x11;
MsgBuff.DATA[1] = 0x22;

__ioctl(desc->nFileNo, PCAN_WRITE_MSG, &MsgBuff);
```

[3] http://www.peak-system.com/fileadmin/media/linux/files/Installation-en_ab6.9.pdf, Seite 18
[4] http://www.can232.com/can232.pdf, Seite 7
[5] http://www.port.de/software/can4linux/structcanmsg__t.html
[6] ioctl()-Aufruf aus peak-linux-driver/lib/src/libpcan.c, CAN_write(), Zeile 392

**Beispiel für das Zugriffsverfahren 3:**

Im Gegensatz zu den gezeigten Zugriffsverfahren, die trotz unterschiedlicher Ausprägung direkt auf die vom Betriebssystem zur Verfügung gestellte Treiberschnittstelle zugreifen, besteht die Möglichkeit des CAN Zugriffs über Programmbibliotheken.

*CANpie:* [7]

```
_TsCpCanMsg tsMyCanMsgT;
_U08 aubDataT[8];

CpMsgSetStdId(&tsMyCanMsgT, 0x123);
CpMsgSetDlc(&tsMyCanMsgT, 2);
CpCoreBufferInit(ptsCanPortV, &tsCanMsgT, CP_BUFFER_1, CP_BUFFER_DIR_TX);

aubDataT[0] = 0x11;
aubDataT[1] = 0x22;
CpCoreBufferSetData(ptsCanPortV, CP_BUFFER_1, &aubDataT);

CpCoreBufferSend(ptsCanPortV, CP_BUFFER_1);
```

Die herstellerspezifische CAN Bibliothek greift dabei als einzige Instanz direkt auf den CAN Treiber zu und bietet daher eine gewisse Abstraktion vom eigentlichen CAN Treiber. Die direkte Programmierschnittstelle des CAN Treibers wird durch die Bibliothek vor der CAN Anwendung verborgen.

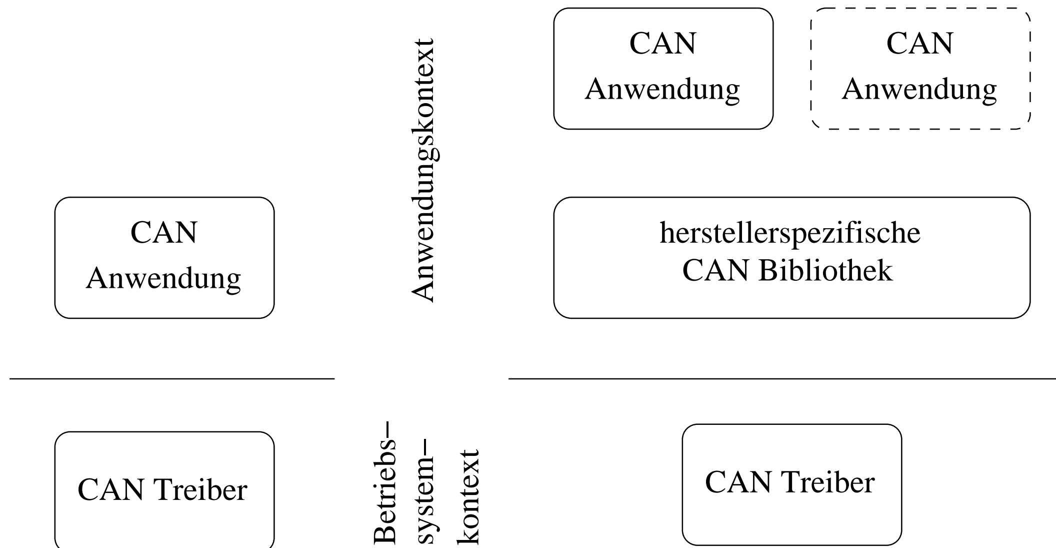



Abbildung 3.5: CAN Zugriff über Bibliotheksfunktionen (rechts)

[7] http://canpie.svn.sourceforge.net/viewvc/canpie/trunk/docs/pdf/canpie_user.pdf, Kapitel 4.6, 4.8

Die herstellerspezifischen CAN Bibliotheken bieten die zusätzliche Möglichkeit der CAN Anwendung Funktionalitäten anzubieten, die durch eine einfache Treiber Schnittstelle nicht realisierbar ist. Beispielsweise die Nutzung einer CAN Schnittstelle durch mehr als eine Anwendung (siehe gestrichelte CAN Anwendung in Abbildung 3.5). Allerdings ist eine solche Lösung proprietär und ein Fehler in der Bibliothek kann das Projektziel gefährden, wenn diese beispielsweise nur in binärer Form und nicht im Quelltext vorliegt.

## 3.5 CAN Programmierschnittstellen unter Windows

Die Programmierschnittstelle für CAN Treiber unter Microsoft Windows ist dem für Linux beschriebenen Zugriffsverfahren sehr ähnlich. Im Unterschied zu der in Unix üblichen Gerätedatei, die über das Dateisystem geöffnet werden kann, registriert ein Microsoft Windows Treiber im Windows Driver Model (WDM) einen eindeutigen Namen über welchen ein Anwendungsprogramm auf den Treiber zugreifen kann:

```
int _cdecl main(void)
{
    HANDLE hFile;
    DWORD dwReturn;

    hFile = CreateFile("\\\\.\\MyExampleCANdriver",
            GENERIC_READ | GENERIC_WRITE, 0, NULL,
            OPEN_EXISTING, 0, NULL);

    if(hFile)
    {
        WriteFile(hFile, "Hello from user mode!",
                  sizeof("Hello from user mode!"), &dwReturn, NULL);
        CloseHandle(hFile);
    }

    return 0;
}
```

Über den gültigen Dateideskriptor `hFile` kann der Treiber `"MyExampleCANdriver"` mit den Systemaufrufen `WriteFile()`, `ReadFile()`, `DeviceIoControl()` und `CloseHandle()` von der Anwendungsseite aus angesprochen werden[8].

Die Situation der unterschiedlichen Dateninhalte, die beispielsweise beim Versenden einer einzelnen CAN Nachricht in den Treiber übertragen werden, entspricht der Situation im Linux Umfeld (siehe Kapitel 3.4). Im Unterschied zu Linux wurden bei den verschiedenen untersuchten Windows CAN Treibern die detaillierten Schnittstellenspezifikationen

[8] Quelltextbeispiel von http://www.codeproject.com/KB/system/driverdev.aspx

der Treiber nicht in der Dokumentation angegeben. Ein quelloffener CAN Treiber für Microsoft Windows wurde trotz intensiver Recherche nicht gefunden.

Der Anwender ist daher auf die von Hersteller mitgelieferten Bibliotheksfunktionen für einen Zugriff auf das Controller Area Network angewiesen (siehe Abbildung 3.5). Diese dynamisch gelinkten Bibliotheken (engl. dynamic linked libraries - DLL) ermöglichen es dem Treiberanbieter zusätzliche Funktionalitäten anzubieten (siehe Seite 50). Allerdings ist eine solche Lösung wie auch im Linux Umfeld proprietär. Entsprechend der Situation bei Windows CAN Treibern liegen die dynamisch gelinkten CAN Bibliotheken seitens der CAN Hardware Hersteller nur in binärer Form vor wodurch ein Fehler in der herstellerspezifischen DLL potenziell das Projektziel gefährden kann.

Für die Abstraktion von CAN Anwendungsschnittellen unter Microsoft Windows setzt sich ein OpenSource Projekt CANAL[9] ein. Die Zielsetzung ist eine Abstration für die existierenden, binären CAN Bibliotheken verschiedener CAN Hardwarehersteller.

| CAN Anwendung | (OpenSource) |
|---|---|
| CANAL Anwendungsschnittstelle | (OpenSource) |
| CANAL Adapter für herstellerspezifische CAN Bibliothek | (OpenSource) |
| herstellerspezifische CAN Bibliothek | (binär) |
| Betriebssystemschnittstelle MS Windows | - |
| herstellerspezifischer CAN Treiber | (binär) |
| herstellerspezifische CAN Hardware | (Hardware) |

Tabelle 3.3: Schichtenmodell des CAN Abstraction Layer für Windows

Das im Umfeld von CANAL entwickelte Very Simple Control Protocol (VSCP) ermöglicht auch die Übertragung von CAN Informationen zu einem VSCP Client über ein Internet Protokoll Netzwerk. Ein VSCP Server stellt dazu beispielsweise auf einem entfernten Rechner seine CAN Schnittstellen zur Verfügung.

Eine detaillierte Bewertung der Funktionalitäten und insbesondere der Performanz des CANAL Konzeptes wurde im Rahmen dieser Arbeit nicht durchgeführt, weil die Zielsetzung einer allgemeingültigen Betriebssystemschnittstelle für das Controller Area Network beim CANAL Ansatz nicht betrachtet wird. CANAL erzeugt eine weitere Abstraktionsschicht im Anwendungskontext (mit den daraus resultierenden Restriktionen), um durch zwei Treiberbibliotheken (Hersteller/CANAL) die beschriebenen Probleme der Wiederverwendung von CAN Anwendungen zu reduzieren.

[9] **CAN** **A**bstraction **L**ayer http://www.vscp.org

## 3.6 Programmierschnittstellen für CAN Protokolle

Wie in Kapitel 2.3 ff. vorgestellt, erfordern verschiedene CAN basierte Kommunikationsprotokolle Antworten von CAN Teilnehmern oder Aussendungen von CAN Botschaften innerhalb gewisser zeitlicher Randbedingungen. Eine fehlerhafte Kommunikation wird im Protokoll beispielsweise durch Zeitüberschreitungen (engl. Timeouts) erkannt.

Für eine Realisierung von CAN Transportprotokollen, zyklischen Sendeaufträgen oder CANopen Teilnehmern bieten sich Protokollbibliotheken an, die den Anwendungsprogrammierer von den Details des verwendeten Protokolls (beispielsweise der Timeout-Überwachung) entlasten sollen. Diese vom Anwendungsprogramm verwendeten Protokollbibliotheken nutzen ihrerseits für den Zugriff auf den CAN Bus eine CAN Hardware. Wie in Kapitel 3.2 bereits dargestellt wurde, können für die Realisierung von zeitkritischen Anwortzeiten *aktive* CAN Karten eingesetzt werden, welche die benötigten CAN Protokolle auf einer eingebetten Hardware umsetzen.

Bei der Verwendung von *passiven* CAN Karten sind die CAN Protokollbibliotheken nach dem Stand der Technik im PC-Umfeld im Anwendungskontext realisiert, wie es in Abbildung 3.6 dargestellt ist. In der Abbildung ist auch die Umsetzung eines CAN Protokolls auf Basis einer herstellerspezifischen CAN Bibliothek (vergleiche Abbildung 3.5 auf Seite 49) dargestellt.

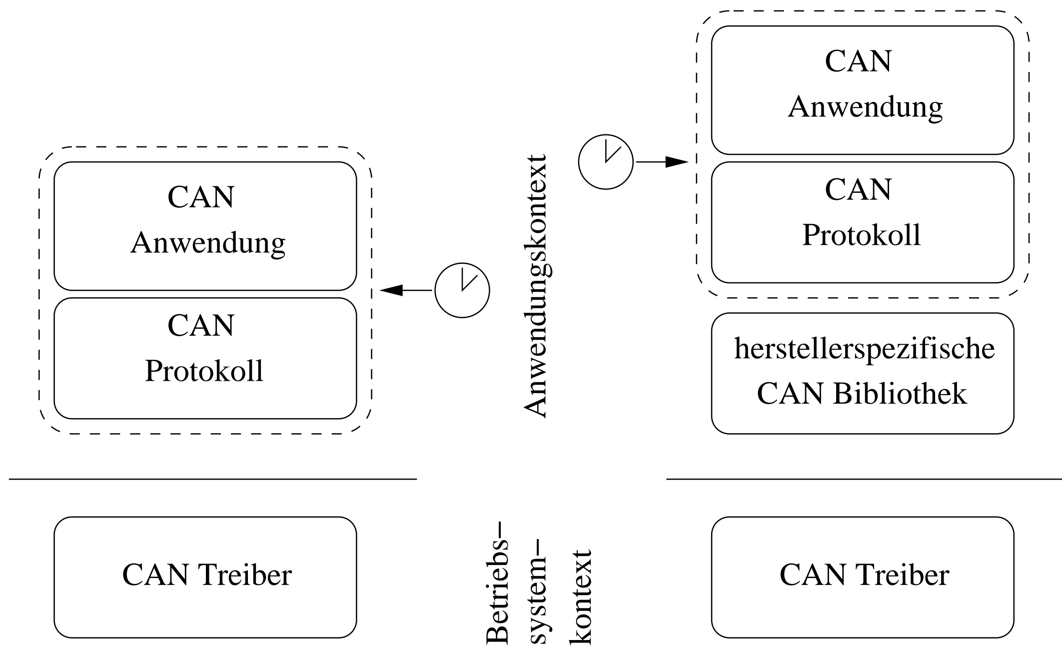



Abbildung 3.6: CAN Protokolle im Anwendungskontext

Die Implementierung von CAN Protokollen im Anwendungskontext von Mehrbenutzerbetriebssystemen ist in Bezug auf die Einhaltung der zeitlichen Anforderungen verschiedenen Restriktionen unterworfen, wie beispielsweise

- einer zumeist geringen Auflösung des Zeitgebers (bspw. 10ms)
- einer nicht deterministischen Ablaufsteuerung (engl. Scheduling)

Die geringe Auflösung des Zeitgebers lässt sich in vielen Fällen durch einen entsprechenden Programmieraufwand (Aufrunden von Zeitwerten) kompensieren, wobei sich allerdings die Performanz der Lösung zwangsweise reduziert, wenn sich beispielsweise der minimal mögliche Nachrichtenabstand von 3ms auf 10ms erhöht.

Als Beispiel für eine zeitgeberabhängige Bibliotheksimplementierung zum Senden zyklischer CAN Botschaften kann das 'C-API Programmierhandbuch' für das so genannte 'VCI - Virtual CAN Interface' der Firma IXXAT in Version 3.2[10] betrachtet werden. Aus der Beschreibung der zyklischen Senderliste auf den Seiten 24 bis 26 ergibt sich die Einschränkung auf die Aussendung jeweils einer CAN Nachricht 'pro Tick', wobei die Zeitspanne für einen 'Tick' aus systemspezifischen Konstanten ermittelt werden kann. Eine Genauigkeit für das Verfahren ist auf +/- 1 Tick zugesichert.

Ein weiteres umfassenderes Beispiel für eine Spezifikation für den Umgang mit zyklischen CAN Nachrichten existiert in der OSEK Communication Specification [43]. In dieser Spezifikation können unter anderem Filterregeln für Signalwerte definiert werden, die in CAN Botschaften kodiert wurden. Zusätzlich können Regeln für die Ausfallerkennung beim Empfang von zyklischen CAN Botschaften definiert werden. Die zeitliche Auflösung dieser Funktionalitäten ist gebunden an die gegebenen Möglichkeiten im Anwendungskontext (s.o.). Daher wird die OSEK Communication Specification ausschließlich im eingebetteten Umfeld - beispielsweise im Zusammenhang mit dem OSEK Betriebssystem - eingesetzt.

Die aus der Anwendungssicht nicht deterministische Verfügbarkeit des Prozessors zur Einhaltung von zeitlichen Restriktionen sowie die geringe Auflösung von Zeitgebern im Anwendungskontext stellen schlechte Voraussetzungen für die Implementierung von CAN Protokollen im Anwendungskontext von Mehrbenutzerbetriebssystemen dar. Vor diesem Hintergrund ist der Einsatz von *aktiver* CAN Hardware in Mehrbenutzerbetriebssystemen wie Microsoft Windows beispielsweise für die zuverlässige Durchführung eines Software-Updates von CAN Steuergeräten gerechtfertigt.

[10] http://www.ixxat.de/download/vci_32_4.02.0250.10012_[c-api]_1.4.pdf

## 3.7 Zusammenfassung und Problemexposition

In diesem Kapitel wurde der Stand der Technik zum Zugriff auf das Controller Area Network durch den Anwendungs- und Treiberprogrammierer dargestellt. In den gezeigten Beispielen wurden die unterschiedlichen Verfahren zur Konfiguration des CAN Controllers und zur Ermittlung der korrekten CAN Schnittstelle zur Vereinfachung weggelassen. Dennoch zeigt sich anhand des beispielhaften Versendens einer einzelnen CAN Nachricht, dass der Aufbau der Datenstrukturen für eine CAN Botschaft und die Programmierschnittstelle zum letztendlichen Versenden einer CAN Botschaft sehr unterschiedlich realisiert werden können.

Schon die verschiedenartige Umsetzung dieses einfachen und üblichen Anwendungsfalles "Versenden einer CAN Nachricht" verdeutlicht die erhebliche Bindung der CAN Anwendung mit dem darunterliegenden CAN Treiber bzw. der CAN Bibliothek nach dem Stand der Technik. Der Anwendungsprogrammierer wird auf der Schnittstellenebene beispielsweise mit hardwarespezifischen Details belastet, wie den '16 Mailboxen' im Intel i82527 CAN Controller (siehe Abbildung 3.2).

In der Folge muss sich der CAN Anwendungsprogrammierer konkret mit der vorliegenden CAN Hardware, der Schnittstelle zum CAN Treiber bzw. der Schnittstelle zur CAN Bibliothek auseinandersetzen. Das Verhalten im Fehlerfall und die Art der Konfiguration der CAN Schnittstellen (beispielsweise der Bitrate) unterscheidet sich zusätzlich.

Das Anwendungsprogramm wird dadurch für eine konkrete CAN Hardware beziehungsweise für eine konkrete CAN Treiberschnittstelle realisiert, was einen Austausch der CAN Hardware zu einem schwer kalkulierbaren Entwicklungsaufwand werden lässt. Für den CAN Anwender ergibt sich daraus eine erhebliche Abhängigkeit vom Hersteller seiner CAN Hardware. Dieser im wirtschaftswissenschaftlichen Umfeld beschriebene Lock-In-Effekt, beschreibt den Umstand, dass eine Änderung einer aktuellen Situation beispielsweise durch zu erwartende Wechselkosten unwirtschaftlich ist und daher ggf. ein verlustbehaftetes Verhalten weitergeführt wird, um noch größere Verluste abzuwenden (siehe [30] Kapitel 2.2.2.3 und 2.2.2.4).

Die beschriebenen technischen und wirtschaftlichen Probleme und Abhängigkeiten bei der Anbindung von Feldbussen werden bei der Verwendung von etablierten Mehrbenutzerbetriebssystemen auf leistungsfähigen, eingebetteten Prozessoren nicht gelöst. Würde man beispielsweise ein aus dem PC-Umfeld bekanntes Konzept wie *aktive* CAN Hardware in das einbettete Umfeld übertragen, würde dieses eine zusätzliche eingebettete CPU für den CAN Bus erforderlich machen, selbst wenn der leistungsfähige Anwendungsprozessor bereits über eine integrierte CAN Schnittstelle verfügt - eine aus Komplexitäts- und Kostengründen inakzeptable Lösung.

Nach dem Stand der Technik ist eine einheitliche Lösung für eine sinnvolle Abstraktion von Feldbussen in Mehrbenutzerbetriebssystemen nicht vorhanden. In der Folge liefern die Chip-Hersteller von neuen, leistungsfähigen eingebetteten Prozessoren mit integriertem CAN Controller (bspw. Freescale MPC5200 oder Intel EP80579) jeweils einen weiteren CAN Treiber mit, der auf die spezifischen Eigenschaften des eigenen Produktes angepasst ist. Durch jede Neuentwicklung einer leistungsfähigen, eingebetteten Hardware mit integriertem CAN Controller wird auf diese Weise ein weiterer CAN Treiber mit herstellerspezifischen Programmierschnittstellen in den Markt gegeben.

Die bisher verfolgten und in diesem Kapitel dargestellen Konzepte sind aus technischer und wirtschaftlicher Sicht für eine Anbindung des Controller Area Networks in ein Mehrbenutzerbetriebssystem auf eingebetteten Prozessoren nicht sinnvoll anwendbar. Der sich ökonomisch verbietende Einsatz einer zusätzlichen *aktiven* CAN Hardware führt zu einer technisch mangelhaften Realisierung von CAN Transportprotokollen im Anwendungskontext unter Verwendung des integrierten, *passiven* CAN Controllers.

Die Motivation des Anwenders, ein Mehrbenutzerbetriebssystem auf eingebetteten Prozessoren einzusetzen, um bekannte Programmierschnittstellen und sinnvoll abstrahierte (CAN-) Hardware nutzen zu können, wird durch den Stand der Technik für die Anbindung des CAN Busses in diesen Betriebssystemen in keinster Weise erfüllt.

# 4 Die CAN Programmierschnittstelle in Mehrbenutzersystemen

In diesem Kapitel wird ein Konzept für eine CAN Programmierschnittstelle in Mehrbenutzerbetriebssystemen erarbeitet.

Dazu werden im Abschnitt 4.1 (Anforderungsanalyse) die Anforderungen zusammengeführt, die sich aus der Analyse der Anforderungen aus Kapitel 2 (Grundlagen und Anwendungen des Controller Area Network) und 3 (Stand der Technik für den Zugriff auf das Controller Area Network) ergeben.

Im Abschnitt 4.2 (Bewertung möglicher Lösungsansätze) wird versucht, die erarbeiteten Anforderungen auf unterschiedliche Betriebssystemschnittstellen abzubilden. Im Rahmen dieser Abbildung wird sichtbar, dass die nach dem Stand der Technik verwendeten Betriebssystemschnittstellen für die gegebenen Anforderungen nicht geeignet sind. Die alternativ betrachtete Abbildung der Anforderungen auf eine Betriebssystemschnittstelle zur Netzwerkprogrammierung erfüllt nach dem Stand der Technik nur einen Teil der Anforderungen aus Abschnitt 4.1.

Die Betriebssystemschnittstellen zur Netzwerkprogrammierung stellen in Abschnitt 4.2 einen möglichen Lösungsansatz für eine zu konzipierende CAN Programmierschnittstelle in Mehrbenutzerbetriebssystemen dar. Im Abschnitt 4.3 werden die im Rahmen der Anforderungsanalyse erarbeiteten und bei der versuchsweisen Adaption in Abschnitt 4.2 nicht erfüllten Anforderungen in das Netzwerkkonzept aufgenommen. Als Ergebnis des Abschnittes 4.3 wird damit eine CAN Programmierschnittstelle für Mehrbenutzerbetriebssysteme entwickelt, die die gegebenen Anforderungen bestmöglich erfüllt.

Im Abschnitt 4.4 wird die am Ende der Anforderungsanalyse formulierte These mit den Ergebnissen aus der in Abschnitt 4.3 entwickelten Lösung belegt. Es wird unter anderem die mögliche Substitution von *aktiver* CAN Hardware dargestellt, sowie auf weiterführende Informationen in Kapitel 5 und im Anhang A (Details der Linux Implementierung) verwiesen.

## 4.1 Anforderungsanalyse

Für eine Betrachtung der Anforderungen aus Kapitel 2 und 3 werden vier Teilbereiche untersucht, die unterschiedliche Aspekte der Problemstellung adressieren:

**Realisierung von vorhandenen Anwendungen** Zur Ermittlung von nicht-trivialen Anforderungen aus Anwendungsszenarien werden drei exemplarische Beispiele zur Verwendung des Controller Area Networks aus Kapitel 2 für eine Konzeptentwicklung herangezogen. Zur Unterstützung einer Migration vom bisherigen Stand der Technik ist für eine Benutzerakzeptanz grundsätzlich eine einfach umsetzbare Realisierung der bekannten Anwendungsszenarien zu ermöglichen.

**Wiederverwendung von vorhandener CAN Software** Der allgemein akzeptierte Stand der Technik zum Zugriff auf den CAN Bus wird in der Form einer zeichenbasierten Betriebssystemschnittstelle (siehe Kapitel 3.4 und 3.5) realisiert. Für eine Migration vom bisherigen Stand der Technik ist eine möglichst ähnliche und einfach adaptierbare, abstrakte Programmierschnittstelle für den CAN Anwendungsprogrammierer vorzusehen.

**Wiederverwendung von vorhandener CAN Hardware** In Kapitel 3 wurden unterschiedliche Konzepte für die Anbindung von *aktiver* und *passiver* CAN Hardware an den Anwendungsprozessor beschrieben. Für eine Migration vom bisherigen Stand der Technik ist die Unterstützung jeglicher (neuer oder bereits vorhandener) CAN Hardware durch das zu entwickelnde Konzept sicher zu stellen.

**Wiederverwendung von Konzepten aus der Informationstechnologie** Eine neu zu entwickelnde Programmierschnittstelle für Mehrbenutzerbetriebssysteme erfordert die Anwendung und Adaption von Konzepten und Designvorgaben nach dem Stand der Technik. Für eine Akzeptanz von Benutzern, die sich aus der Perspektive der Mehrbenutzerbetriebssysteme mit dem Controller Area Network auseinandersetzen, sind etablierte und in ihrer Funktionalität bekannte Programmierschnittstellen zu realisieren.

Die in diesem Abschnitt ermittelten Anforderungen entstammen unterschiedlichen Abstraktionsebenen, woraus sich ein unterschiedlicher Detaillierungsgrad der einzelnen Anforderungen ergeben kann. Die im Folgenden mit '$A < Nummer >$' bezeichneten Anforderungen sind sämtlich für die Konzeptfindung und die Bewertung relevant.

### 4.1.1 Realisierung von vorhandenen Anwendungen

#### Zugriff auf zyklische Fahrzeugdaten

Wie im Kapitel 2.3.1 eingehend beschrieben, werden im Kraftfahrzeug Sensorinformationen in der Form von zyklisch gesendeten CAN Botschaften übertragen. Diese zyklische Übertragung mit Zykluszeiten von 5ms - 1000ms wird zur Erkennung eines Signalausfalles und entsprechender Fehlerbehandlung genutzt.

Zur Filterung von Fahrzeugsignalen (beispielsweise dem Blinkersignal oder der Motordrehzahl) wurde im Rahmen der OSEK Communication Specification [43] eine Programmierschnittstelle vorgesehen, die eine so genannte signalbasierte Filterung ermöglicht. Diese Filter werden abhängig vom Anwendungsfall zur Kompilierungszeit definiert und sprechen zur Laufzeit eine bestimmte Callback-Funktion an, wenn sich das überwachte Signal gemäß der Spezifikation geändert hat.

Durch ein solches Verfahren wird die Anwendung davon entlastet, jede empfangene CAN Botschaft hinsichtlich des darin enthaltenen Signals zu bewerten. Vor dem Hintergrund der Tatsache, dass die angenommene Zykluszeit der CAN Botschaft mit dem Blinkersignal bei 100ms liegt und nur in 10% der Reisezeit mit dem Fahrzeug der Blinker betätigt wird, welcher sich dann nur jeweils einmal pro Sekunde ändert, ergibt sich folgende Reduzierung in der Verarbeitung von Informationen im Anwendungsprogramm:

$$\begin{aligned} \text{Information}_{\,ungefiltert} &= 100 \text{ Botschaften} \\ \text{Information}_{\,gefiltert} &= 1 \text{ Botschaft} \end{aligned}$$

(10% der Reisezeit von 100 Botschaften = 10 Botschaften in denen der Blinker betätigt ist, sowie eine Änderung des Signals alle 1000ms, d.h. 9 von den übrig gebliebenen 10 Botschaften werden beim aktivierten Blinker unterdrückt)

Ein inhaltsbasierter Filter für zyklische Botschaften im Kraftfahrzeug kann eine CAN Anwendung zur Laufzeit und in Ihrer Komplexität massiv entlasten und wird daher als Anforderung betrachtet.

**(A1)** MÖGLICHKEIT ZUR INHALTSBASIERTEN FILTERUNG VON CAN BOTSCHAFTEN

#### Deadline Monitoring

In der OSEK Communication Specification [43] ist zusätzlich zu der beschriebenen Programmierschnittstelle für die signalbasierte Filterung von CAN Nachrichten in Kapitel 2.5.1 ein Verfahren zum so genannten 'Reception Deadline Monitoring' beschrieben. Mit Deadline Monitoring wird in diesem Zusammenhang die Erkennung des Ausbleibens einer zyklischen CAN Botschaft bezeichnet.

Für eine Realisierung des Deadline Monitorings durch die Anwendung muss nach dem Stand der Technik die zu überwachende CAN Botschaft ungefiltert an die CAN Anwendung weitergeleitet werden, um den letzten Empfangszeitpunkt zu bestimmen. Der Programmierer der CAN Anwendung muss sich dabei mit der Verwendung von Zeitgebern im Anwendungskontext und den Randbedingungen des betriebssystemspezifischen Schedulings auseinandersetzen. Werden die Dateninhalte der zu zeitlich zu überwachenden CAN Botschaft nicht benötigt, wird die Botschaft nach der Übertragung in den Anwendungskontext zu 100% verworfen.

**(A2)** Effizientes Deadline Monitoring von CAN Botschaften

### Multiple Instanzen von Kommunikationsprotokollen

Aus Kapitel 2.3.2 sind verschiedene CAN Transportprotokolle bekannt, die es ermöglichen eine Punkt-zu-Punkt Kommunikationsverbindung auf dem CAN Bus zu realisieren und dabei Informationen zu segmentieren, deren Länge über die in CAN Botschaften möglichen 8 Byte hinausgeht.

Für eine performante Datenübertragung und zur 'schnellen' Erkennung von Kommunikationsfehlern erfordern CAN Transportprotokolle zur Bestätigung empfangener Datensegmente Antwortzeiten im Bereich von wenigen Millisekunden. Diese Anforderungen werden beim Stand der Technik durch *aktive* CAN Hardware (siehe Kapitel 3.2) erfüllt oder es wird ein Echtzeitbetriebssystem verwendet, welches die Antwortzeit für das CAN Protokoll durch eine garantierte zeitliche Verfügbarkeit des Prozessors sicherstellt. Mit Ausnahme des Betriebssystems QNX erfüllt keines der in dieser Arbeit betrachteten Mehrbenutzerbetriebssysteme die gestellten Anforderungen an Echtzeitbetriebssysteme.

**(A3)** Realisierung von Anwortzeiten von Null bis wenigen Millisekunden

**(A4)** Keine Einschränkung auf Echtzeitbetriebssysteme

Da nur jeweils eine begrenzte Anzahl von zwei bis vier CAN Identifiern für einen CAN Transportkanal benötigt werden, können zu einer Zeit mehrere hundert Transportkanäle auf einem CAN Bus realisiert werden.

**(A5)** Multiple Instanzen von Kommunikationsprotokollen

Als Konsequenz soll die konzeptionell mögliche, parallele Datenübertragung durch verschiedene CAN Transportprotokolle ausschließlich durch die auf dem CAN Bus zur Verfügung stehende Bandbreite begrenzt sein.

**(A6)** Simultane Unterstützung verschiedener Kommunikationsprotokolle

### 4.1.2 Wiederverwendung von vorhandener CAN Software

Die Integration einer CAN Hardware in ein Mehrbenutzerbetriebssystem erfordert eine Abstraktion von einem konkreten CAN Controller. Diese Abstraktion ist vergleichbar mit den Anforderungen an die Anbindung einer Festplatte, die für das Betriebssystem letztendlich nur noch ein Speichergerät mit einer bestimmten Anzahl von Datenblöcken darstellt.

Die Verwendung des Controller Area Networks durch ein CAN Anwendungsprogramm ist nach dem Stand der Technik geprägt von dem direkten Zugriff auf einen spezifischen CAN Controller (siehe Kapitel 3.1) oder alternativ von dem Zugriff über eine spezifische CAN Bibliothek (siehe Abbildung 3.5). Der dem CAN Anwendungsprogramm zur Verfügung gestellte Funktionsumfang geht dabei üblicherweise von einer *exklusiven* Nutzung der CAN Schnittstelle durch die CAN Anwendung aus ("Single-User-Betrieb").

Zur Erlangung einer entsprechenden Benutzerakzeptanz ist eine Programmierschnittstelle vorzusehen, die der Programmierung eines *abstrakten* CAN Controllers nachempfunden ist. Daraus ergeben sich die folgenden Anforderungen an die zu entwickelnde CAN Programmierschnittstelle für Mehrbenutzerbetriebssysteme:

**(A7)** Öffnen (Initialisierung) und Schliessen der CAN Schnittstelle

**(A8)** Konfiguration der Bitrate des CAN Controllers

**(A9)** Konfiguration der Empfangsfilter des CAN Controllers

**(A10)** Senden und Empfangen von CAN Nachrichten

Die dargestellten Anforderungen beschreiben ein allgemeines und übliches Benutzungsszenario bei der Verwendung des Controller Area Networks durch CAN Anwender. Das Benutzungsszenario ist unabhängig davon, ob dabei eine *aktive* oder *passive* CAN Hardware Verwendung findet. *Aktive* CAN Hardware bietet unter Umständen zusätzlich zu dem beschriebenen Funktionsumfang weitergehende Funktionalitäten (siehe Kapitel 3.2).

### 4.1.3 Wiederverwendung von vorhandener CAN Hardware

In Kapitel 3 wurden unterschiedliche Konzepte für die Realisierung von CAN Hardware dargestellt, welche im Rahmen einer Realisierung von CAN Programmierschnittstellen für Mehrbenutzerbetriebssysteme betrachtet und angebunden werden müssen.

- *passive* CAN Hardware mit direktem Zugriff auf den CAN Controller (Kapitel 3.1)
- *aktive* CAN Hardware mit zusätzlichem eingebettetem Prozessor (Kapitel 3.2)

Bei der Verwendung existierender CAN Controller in der Form einer *passiven* CAN Hardware wird die Abstraktion der Hardware durch herstellerspezifsche Funktionalitäten erschwert, wie beispielsweise die '16 Mailboxen' im Intel i82527 CAN Controller (siehe Abbildung 3.2 auf Seite 40).

Wie in Kapitel 3.2 dargestellt, sind *aktive* CAN Karten zusätzlich in der Lage, CAN Kommunikationsprotokolle auf dem Prozessor der *aktiven* CAN Hardware zu realisieren, um die Probleme mit unterschiedlichen Latenzen bei der Verarbeitung von Unterbrechungsanforderungen in Windows-PCs zu lösen. Wenngleich die zusätzlichen Funktionalitäten verschiedener *aktiver* und *passiver* CAN Schnittstellenkarten weit über das Senden und Empfangen von CAN Botschaften hinausgehen können, sind diese proprietären Eigenschaften der CAN Hardware für eine Abstraktion nicht adaptierbar.

Grundsätzlich besteht der kleinste gemeinsame Funktionsumfang von CAN Hardware standardmäßig aus dem Senden von CAN Botschaften und dem Empfang ungefilterter CAN Botschaften. Für eine allgemeingültige Abstraktion einer CAN Hardware muss folglich dieser kleinste gemeinsame Funktionsumfang aller CAN Controller betrachtet und realisiert werden, welche den CAN Standard ISO 11898-1 [23] implementieren.

**(A11)** VERZICHT AUF PROPRIETÄRE EIGENSCHAFTEN VON CAN HARDWARE

Eine potenziell vorhandene Integration von *passiven* CAN Controllern auf den betrachteten, eingebetteten 32-Bit Prozessoren verhindert bereits aus ökonomischen Gründen die Installation einer *aktiven* CAN Hardware mit einem zusätzlichen eingebetten Prozessor, siehe Kapitel 3.2 (Aktive und passive CAN PC Schnittstellenkarten) und Kapitel 3.7 (Zusammenfassung und Problemexposition). Für die Anforderungen, die bisher eine *aktive* CAN Hardware erforderlich gemacht haben, sind alternative Lösungen zu erarbeiten.

### 4.1.4 Wiederverwendung von Konzepten aus der Informationstechnologie

Der aktuelle Stand der Technik erlaubt teilweise die gleichzeitige Nutzung des Controller Area Networks durch verschiedene Anwendungen. Diese Anforderung ist daher mit dem für das Mehrbenutzerbetriebssystem übliche Zugriffs- und Rechtemanagement aufzunehmen. Die verschiedenen CAN Anwendungen müssen nach dem Stand der Technik in Mehrbenutzersystemen voneinander unabhängig realisierbar und ausführbar sein.

**(A12)** Mehrbenutzerfähigkeit

Die Motivation zum Einsatz eines Mehrbenutzerbetriebssystem auf eingebetteten Prozessoren ergibt sich insbesondere durch die geringe Einarbeitungszeit für Programmierer aufgrund bekannter Programmierschnittstellen.

**(A13)** Verwendung von bekannten Programmierschnittstellen

Grundsätzlich sollen die zu erarbeitenden CAN Programmierschnittstellen den Anforderungen nach dem Stand der Technik in der Informationstechnologie gerecht werden. Dazu müssen die anerkannten Regeln zum Entwurf von Programmierschnittstellen beachtet werden[3]:

- Easy to learn
- Easy to use, even without documentation
- Hard to misuse
- Easy to read and maintain code that uses it
- Sufficiently powerful to satisfy requirements
- Easy to extend
- Appropriate to audience

Diese nicht-funktionalen Anforderungen müssen beim Entwurf einer Betriebssystemschnittstelle zusätzlich in das vorhandene Entwurfsschema des jeweiligen Betriebssystems eingepasst werden, um eine konsistente Erweiterung der Betriebssystemschnittstellen für den Programmierer zu erreichen. Es ermöglicht dem Programmierer, sein Wissen über bekannte Programmierschnittstellen und deren Verhalten auf die neue Schnittstelle anwenden zu können.

### 4.1.5 Zusammenfassung der Anforderungen

Im Folgenden werden die verschiedenen, in diesem Abschnitt zusammengetragenen Anforderungen aus den vier betrachteten Teilbereichen in einer Übersicht dargestellt.

Realisierung von vorhandenen Anwendungen

**(A1)** Möglichkeit zur inhaltsbasierten Filterung von CAN Botschaften

**(A2)** Effizientes Deadline Monitoring von CAN Botschaften

**(A3)** Realisierung von Anwortzeiten von Null bis wenigen Millisekunden

**(A4)** Keine Einschränkung auf Echtzeitbetriebssysteme

**(A5)** Multiple Instanzen von Kommunikationsprotokollen

**(A6)** Simultane Unterstützung verschiedener Kommunikationsprotokolle

Wiederverwendung von vorhandener CAN Software

**(A7)** Öffnen (Initialisierung) und Schliessen der CAN Schnittstelle

**(A8)** Konfiguration der Bitrate des CAN Controllers

**(A9)** Konfiguration der Empfangsfilter des CAN Controllers

**(A10)** Senden und Empfangen von CAN Nachrichten

Wiederverwendung von vorhandener CAN Hardware

**(A11)** Verzicht auf proprietäre Eigenschaften von CAN Hardware

Wiederverwendung von Konzepten aus der Informationstechnologie

**(A12)** Mehrbenutzerfähigkeit

**(A13)** Verwendung von bekannten Programmierschnittstellen

Der Erfüllungsgrad dieser Anforderungen ist das Maß für die Bewertung möglicher Lösungsansätze zur Integration des Controller Area Networks in Mehrbenutzersysteme im folgenden Abschnitt 4.2.

## 4.2 Bewertung möglicher Lösungsansätze

In diesem Abschnitt werden verschiedene Betriebssystemschnittstellen als Lösungsansätze zur Integration des Controller Area Networks in Mehrbenutzersysteme bewertet.

### 4.2.1 Zeichenorientierte Betriebssystemschnittstellen

Die vom Stand der Technik (siehe Kapitel 3) aus verschiedenen Betriebssystemen bekannten Konzepte zum Zugriff auf den CAN Bus

- Direkter Zugriff auf eine zeichenorientierte Schnittstelle
- Programmierschnittstelle für eine Bibliothek im Anwendungskontext

setzen beide auf eine zeichenorientierte Schnittstelle zum Betriebssystem auf. Dieses ist unabhängig davon, ob eine *aktive* oder eine *passive* CAN Hardware verwendet wird.

#### Bewertung von anwendungsbestimmten Anforderungen

Bei Betrachtung der aus dem Anwendungsumfeld generierten Anforderungen

**(A1)** Möglichkeit zur inhaltsbasierten Filterung von CAN Botschaften

**(A2)** Effizientes Deadline Monitoring von CAN Botschaften

**(A3)** Realisierung von Anwortzeiten von Null bis wenigen Millisekunden

**(A6)** Simultane Unterstützung verschiedener Kommunikationsprotokolle

ist offensichtlich, das sie aufgrund der zeitlichen Anforderungen (Punkte A3 und A6) und zur Vermeidung redundanter CAN Nachrichten an die Anwendung (Punkte A1 und A2) grundsätzlich *nicht im Anwendungskontext* realisiert werden können. Weil bei *passiven* CAN Schnittstellenkarten die CAN Anwendung die dargestellten Anforderungen im Anwendungskontext realisieren müsste, ist eine *passive* CAN Hardware nach dem Stand der Technik für diese Anforderungen ungeeignet. Eine Möglichkeit zur Realisierung dieser Anforderungen ist nach dem Stand der Technik ausschließlich durch eine *aktive* CAN Hardware gegeben, was der Anforderung Verzicht auf proprietäre Eigenschaften von CAN Hardware **(A11)** widerspricht.

Zusätzlich ist der eingebettete Prozessor auf den *aktiven* CAN Karten bezüglich seiner Leistungsfähigkeit und dem zur Verfügung stehenden Speicher zumeist auf einen bestimmten Anwendungsfall zugeschnitten. Die in Kapitel 5.1.4 verwendete EDICcard2 verfügt beispielsweise über einen Infineon C165 16-Bit Prozessor mit 512kByte RAM,

welches für die Firmware der Karte genutzt wird. Die Anforderungen an eine Mehrbenutzerfähigkeit **(A12)** sowie die Realisierung der Anforderung **(A5)** (Multiple Instanzen von Kommunikationsprotokollen) können mit den im Vergleich zu Mehrbenutzerbetriebssystemen sehr begrenzten Ressourcen einer *aktiven* CAN Hardware nicht umgesetzt werden.

### Bewertung der Betriebssystemschnittstelle

Der LinCAN[1] Treiber des OCERA Projekts ermöglicht den parallelen Zugriff verschiedener CAN Anwendungen auf einen einzelnen CAN Controller. Dazu realisiert der CAN Treiber im Betriebssystem für jedes Anwendungsprogramm eine separate Instanz für den Eingabe- / Ausgabedatenstrom. Der Zugriff erfolgt über eine einzelne Gerätedatei (beispielsweise `/dev/can0`, siehe Tabelle 3.2 auf Seite 46), welche mehrfach geöffnet werden kann. Die Anzahl der mit diesem Verfahren erzeugbaren Gerätedateien ist aufgrund des möglichen Wertebereiches von Major/Minor-Nummern grundsätzlich beschränkt.

Abgesehen von der beschränkten Anzahl der CAN Schnittstellen kann das LinCAN-Konzept mit der zeichenorientierten Betriebssystemschnittstelle die Anforderungen an die Mehrbenutzerfähigkeit **(A12)** erfüllen. Die Reduzierung auf die Übertragung von CAN Botschaften erfüllt jedoch nicht die Anforderungen an die Simultane Unterstützung verschiedener Kommunikationsprotokolle **(A6)** und Multiple Instanzen von Kommunikationsprotokollen **(A5)**.

### Fazit zur Verwendung zeichenorientierter Betriebssystemschnittstellen

Jede einzelne in Kapitel 4.1 (Anforderungsanalyse) definierte Anforderung lässt sich unter Vernachlässigung der Anforderungen aus Kapitel 4.1.4 (Wiederverwendung von Konzepten aus der Informationstechnologie) für sich genommen mit dem Stand der Technik umsetzen. Die Probleme ergeben sich erst bei der Realisierung der Anforderung Multiple Instanzen von Kommunikationsprotokollen **(A5)**, wenn die etablierten Verfahren nach dem Stand der Technik beispielsweise aufgrund der Anforderungen **(A11)** Verzicht auf proprietäre Eigenschaften von CAN Hardware (hier besonders der Verzicht auf *aktive* CAN Hardware) und **(A4)** Keine Einschränkung auf Echtzeitbetriebssysteme nicht angewandt werden können.

Die Gesamtheit der Anforderungen erfordert eine mehrbenutzerfähige Lösung im Betriebssystemkontext, die sich mit dem Stand der Technik unter Verwendung von zeichenorientierten Betriebssystemschnittstellen nicht realisieren lässt. Die Konzepte nach dem Stand der Technik sind daher für die vorhandenen Anforderungen für Mehrbenutzerbetriebssysteme auf eingebetteten Prozessoren nicht geeignet.

[1] http://www.ocera.org/download/components/WP7/lincan-0.3.3.html

### 4.2.2 Netzwerk Betriebssystemschnittstellen

Die in Kapitel 4.2.1 betrachteten Konzepte zum Zugriff auf die CAN Hardware beschränken sich auf die Nutzung einer bestimmten Klasse der vorhandenen Betriebssystemschnittstellen: Dem so genannten I/O[2]-Subsystem, welches im Allgemeinen für die Anbindung von seriellen und parallelen Schnittstellen, Festplatten und Grafikkarten genutzt wird [32]. Das I/O-Subsystem wird über die in Tabelle 3.2 beschriebenen Device-Files von Anwendungsprogramm genutzt und ermöglicht einen zeichenorientierten bzw. blockweisen Zugriff auf die im System vorhandenen Hardwarekomponenten.

Die Anforderungen aus Kapitel 4.1.1, ein verbindungsorientiertes CAN Transportprotokoll gemäß Kapitel 2.3.2 zu realisieren, findet allerdings im PC-Umfeld seine Entsprechung vergleichsweise in dem bekannten Verfahren der Netzwerkkommunikation über das ebenfalls verbindungsorientierte TCP/IP Protokoll.

Das Transmission Control Protocol (TCP) [50] ist ein auf dem Internetprotokoll (IP) [49] basierendes, verbindungsorientiertes Übertragungprotokoll. Vereinfacht werden bei diesem Verfahren im Betriebssystemkontext Anwendungsdaten segmentiert, um diese unter Verwendung einer Netzwerk-(Ethernet-)Karte übertragen zu können. Gemäß IEEE 802.3 ist die Länge von Ethernetframes auf 1500 Zeichen beschränkt.

Der Zugriff der Anwendung auf den bei TCP/IP-Verbindungen zur Verfügung gestellten, bidirektionalen Datenkanal erfolgt über die so genannte Socket-Schnittstelle. Für die Benutzung der Socket-Schnittstelle existieren zusätzliche Systemaufrufe im jeweiligen Betriebssystem. Zum Öffnen eines I/O-Gerätes wird der Systemaufruf `open()` genutzt, zum Erzeugen eines Datenkanals im Netzwerk wird der Systemaufruf `socket()` verwendet. Der englische Begriff *Socket* bezeichnet dabei den Endpunkt einer Kommunikationsverbindung und entspricht dem deutschen Begriff der *Steckdose*.

Die Anforderung **(A5)** Multiple Instanzen von Kommunikationsprotokollen zu realisieren, führt demnach als Analogie zu anderen verbindungsorientierten Kommunikationsprotokollen zum dem Ansatz, eine Socket-Schnittstelle zu verwenden.

#### Versuchsweise Verwendung von Netzwerk Betriebssystemschnittstellen

Die Realisierung von CAN Transportprotokollen im Betriebssystemkontext könnte sich als eine potenzielle Lösung zur Erfüllung der Anforderungen aus Kapitel 4.1.1 (Realisierung von vorhandenen Anwendungen) darstellen. Eine Implementierung im Betriebssystemkontext könnte in der Folge auch das Potenzial bergen, die Verwendung *aktiver* CAN Hardware sowie die alternative Nutzung von Echtzeitbetriebssystemen zu erübrigen.

[2]Input/Output - zu Deutsch: Eingabe/Ausgabe

Der aus der Sicht eines Anwendungsprogrammierers argumentierbare Top-Down Ansatz für einen einzelnen betrachteten Anwendungsfall führt zur Fragestellung, ob und ggf. wie die übrigen formulierten Anforderungen über eine Socket-Schnittstelle - also mit einem netzwerkorientierten Ansatz - realisierbar sind. Zu der Beantwortung dieser Fragestellung sollen im Folgenden die aus Kapitel 4.1 (Anforderungsanalyse) entstandenen Anforderungen an eine CAN Programmierschnittstelle *versuchsweise* mit den existierenden Verfahren und Konzepten aus dem Umfeld des Internetprotokolls abgebildet werden.

Aufgrund der Ergebnisse der versuchsweisen Abbildung der genannten Anforderungen auf Netzwerkbetriebssystemschnittstellen wird eine These formuliert, welche einer deduktiven Prüfung anhand der ursprünglich formulierten Anforderungen unterzogen wird. Hierzu werden die in der versuchsweisen Abbildung aufgetretenen offenen Punkte zu einer Falsifikation der formulierten These herangezogen und bewertet.

Basierend auf dem bekannten Internet Protokollstapel (IP, UDP, TCP) , der verwendeten Netzwerk-Hardware (Ethernetkarten) und der Betriebssystemschnittstellen zur Verwendung des Internet Protokolls werden die formulierten Anforderungen bewertet:

- Verzicht auf proprietäre Eigenschaften von CAN Hardware

  Ethernetkarten haben grundsätzlich die Aufgabe, ein Ethernetframe als Bitstrom auf einem physikalischen Medium zu senden bzw. von ihm zu empfangen. Im Unterschied zu CAN Frames beinhalten Ethernetframes bis zu 1500 Byte Nutzdaten und werden nach dem CSMA/CD[3] Verfahren auf dem Übertragungsmedium gesendet. Die zu sendenden Ethnernetframes entnimmt der Netzwerktreiber der Ethernetkarte einer vom Betriebssystem zur Verfügung gestellten Warteschlange.

  Ein Ethernetframe enthält eine Sender-MAC[4]-Adresse und eine Empfänger-MAC-Adresse. Sie ermöglicht der Ethernetkarte, nur solche Ethernetframes zu empfangen, die an ihre eigene spezifische MAC-Adresse gesendet wurde. Im Unterschied zu Ethnernetframes enthält ein CAN Frame nur *einen* CAN Identifier, der beim Controller Area Network - abhängig vom Anwendungsfall - üblicherweise als eine Absenderadresse interpretiert wird.

  Die Schnittstelle zum Netzwerktreiber für eine bidirektionale Übertragung von Ethernetframes ist sehr einfach gestaltet, um die Anbindung kostengünstiger Ethernethardware sicherstellen zu können. Höherwertige Ethernetkarten mit besserem Datendurchsatz und direktem Zugriff auf den Hauptspeicher (DMA[5]) können beispielsweise über mehrere Sendewarteschlangen parallel befüllt werden. Diese zusätzliche Funktionalität wird allerdings vor dem Betriebssystem durch die bekannte, einfache Schnittstelle zum Netzwerktreiber verborgen.

[3]Carrier Sense Multiple Access / Collision Detection
[4]Media Access Control (Address), bei Ethernetkarten eine weltweit eindeutige 48 Bit Adresse
[5]Direct Memory Access

Zusammengefasst erfüllt die einfach gestaltete Schnittstelle zur Übertragung von Ethernetframes durch einen Netzwerktreiber die betrachteten Anforderungen. Die Umsetzung der Anforderungen 'Senden von Ethernetframes' und 'Empfangen von Ethernetframes' wird durch den Netzwerktreiber zur Verfügung gestellt und unter Verwendung der konkreten, herstellerspezifischen Ethernethardware durchgeführt. Dabei wird eine ggf. vorhandene lokale Intelligenz der Ethernetkarte vor dem Betriebssystem verborgen. Diese Reduzierung der Schnittstelle auf die Übertragung von Ethernetframes kann als Konzeptvorlage für die Übertragung von CAN Frames für einen CAN-Netzwerktreiber verwendet werden.

> *Offene Punkte aus dieser Anforderung:*
> *Ethernet und CAN nutzen unterschiedliche Adressierungskonzepte auf dem jeweiligen Übertragungsmedium.*

- Verwendung von bekannten Programmierschnittstellen

  Die Socket-Schnittstelle ist eine bekannte und etablierte Programmierschnittstelle.

- Mehrbenutzerfähigkeit

  Die Socket-Schnittstelle kann durch die Erzeugung eines Kommunikationsendpunktes durch den `socket()` Systemaufruf beliebig viele Instanzen von Kommunikationsverbindungen erzeugen. Dieses ermöglicht beispielsweise die parallele Ausführung von WWW-Browser und E-Mailprogramm auf einem PC. Eine Mehrbenutzerfähigkeit ist somit gegeben.

- Möglichkeit zur inhaltsbasierten Filterung von CAN Botschaften

  > *Offene Punkte aus dieser Anforderung:*
  > *Die Möglichkeit der inhaltsbasierten Filterung von CAN Botschaften im Betriebssystemkontext ist zu prüfen, da eine Übertragung von Informationen vom Betriebssystemkontext in den Anwendungskontext auf diese Weise reduziert werden könnte. Eine solche Funktionalität ist aus dem Umfeld des Internetprotokolls nicht bekannt.*

- Öffnen (Initialisierung) und Schliessen der CAN Schnittstelle

  Ist durch die Systemaufrufe `socket()` und `close()` gegeben.

- Konfiguration der Bitrate des CAN Controllers

  Eine CAN Schnittstelle hat analog zu einer Ethernetkarte zu einem Zeitpunkt eine festgelegte Bitrate, die der Bitrate der weiteren Kommunikationsteilnehmer entspricht. In den Anwendungsszenarien nach dem Stand der Technik ist die CAN Anwendung für die Konfiguration der CAN-spezifischen Kommunikationsparameter verantwortlich.

  > *Offene Punkte aus dieser Anforderung:*
  > *In einem Mehrbenutzersystem ist die Konfiguration der Bitrate durch jedes einzelne Anwendungsprogramm nicht sinnvoll. Es ist eine Konfigurationsmöglichkeit vorzusehen, die es dem Systemadministrator ermöglicht,*

*die Bitrate eines CAN Controllers gemäß der Spezifikation des Anwendungsfalls und der weiteren CAN Kommunikationsteilnehmer einzustellen.*

- KONFIGURATION DER EMPFANGSFILTER DES CAN CONTROLLERS

  Die Reduzierung der empfangenen CAN Nachrichten durch die Filterung nach verschiedenen CAN Identifiern ist ein üblicher Anwendungsfall und für eine Reduzierung der Datenübertragung an die Anwendung grundsätzlich sinnvoll. Die von einer Anwendung nach dem Stand der Technik konfigurierten Empfangsfilter sind abhängig von der jeweiligen CAN Hardware und reduzieren die Datenübertragung beispielsweise direkt über hardwarespezifische Filter im CAN Controller.

  *Offene Punkte aus dieser Anforderung:*
  *Aufgrund der Anforderungen an die* MEHRBENUTZERFÄHIGKEIT *ist eine Möglichkeit vorzusehen, den verschiedenen Anwendungen im Mehrbenutzersystem eine Filterung von CAN Nachrichten auf der Basis von CAN Identifiern zu erlauben. Dabei dürfen keine Abhängigkeiten zwischen CAN Anwendungen oder deren jeweils konfigurierten Nachrichtenfiltern bestehen.*

- SENDEN UND EMPFANGEN VON CAN NACHRICHTEN

  Ist beispielsweise durch die Systemaufrufe `read()` und `write()` gegeben.

- REALISIERUNG VON ANWORTZEITEN VON NULL BIS WENIGEN MILLISEKUNDEN

  *Offene Punkte aus dieser Anforderung:*
  *Zeitliche Anforderungen, die über das Antwortverhalten des Internetprotokolls und dessen Zeitüberschreitungen im Fehlerfall hinausgehen sind zu prüfen.*

- MULTIPLE INSTANZEN VON KOMMUNIKATIONSPROTOKOLLEN

  und

- SIMULTANE UNTERSTÜTZUNG VERSCHIEDENER KOMMUNIKATIONSPROTOKOLLE

  Diese Funktionalitäten werden durch einen Internetprotokollstapel in einem Mehrbenutzersystem geleistet. Das Transmission Control Protocol (TCP) und das User Datagram Protocol (UDP) stellen dabei beispielhaft zwei Protokolle der Internetprotokollfamilie dar.

- KEINE EINSCHRÄNKUNG AUF ECHTZEITBETRIEBSSYSTEME

  Sockets und Internetprotokolle sind Bestandteile etablierter Mehrbenutzerbetriebssysteme ohne Echtzeitfähigkeiten. In diesen Betriebssystemen sind Kommunikationsprotokolle nach dem Stand der Technik bekannt, welche zeitliche Restriktionen wie Zeitüberschreitungen (Timeoutüberwachung) im Betriebssystemkontext realisieren. Die konkrete zeitliche Auflösung dieser Verfahren wird als offener Punkt im Rahmer der Bewertung der Anforderung **(A3)** REALISIERUNG VON ANWORTZEITEN VON NULL BIS WENIGEN MILLISEKUNDEN diskutiert (s.o.).

#### Fazit zur Verwendung von Netzwerkbetriebssystemschnittstellen

Zusammenfassend ergeben sich aus der versuchweisen Abbildung der formulierten Anforderungen an CAN Programmierschnittstellen aus Kapitel 4.1 auf die Netzwerk Programmierschnittstellen des Internet Protokollstapels folgende Ergebnisse:

- Mit der Verwendung des Netzwerkkonzepts werden zwei Programmierschnittstellen in unterschiedlichen Schichten des Betriebssystems sichtbar:
  - Die Schnittstelle zur CAN Hardware in der Form eines Netzwerktreibers analog zu Ethernetkarten (ISO/OSI Schicht 2)
  - Die Schnittstelle zum Anwendungskontext in der Form von Netzwerk-Sockets analog der Schnittstelle zur IP-Kommunikation (ISO/OSI Schicht 4)
- Die Schnittstelle zur Hardware in der Form eines Netzwerktreibers zur Übertragung von Daten für die Media Access Control Schicht ist für die Anforderungen des Controller Area Networks voraussichtlich adaptierbar.
- Die Socket Schnittstelle erlaubt die Realisierung verschiedener Protokolle (bspw. CAN Transportprotokolle) in beliebig vielen voneinander unabhängigen Instanzen.

Im Gegensatz zur Bewertung der Konzepte aus dem Stand der Technik sind bei der Verwendung der Netzwerkschnittstellen keine Ausschlusskriterien erfüllt worden, worauf sich folgende These formulieren lässt:

**Die etablierten Betriebssystemschnittstellen zur Nutzung von Netzwerkkommunikation in Mehrbenutzerbetriebssystemen sind geeignet, die aus der Anforderungsanalyse abgeleiteten Anforderungen an eine Integration des Controller Area Networks in Mehrbenutzerbetriebssysteme zu erfüllen.**

Zur Falsifikation dieser These sind folgende Fragestellungen aus den *Offenen Punkten* der versuchsweisen Abbildung auf Netzwerk Programmierschnittstellen zu bewerten:

Q1 Ist das Netzwerktreiber Modell für die CAN Hardware adaptierbar?

Q2 Kann eine systemweite Schnittstelle zur Konfiguration der CAN Hardware durch den Systemadministrator (bspw. der Bitrate) realisiert werden?

Q3 Kann eine Möglichkeit zur inhaltsbasierten Filterung von CAN Botschaften innerhalb des Betriebssystems realisiert werden?

Q4 Kann eine Möglichkeit zur CAN Identifier basierten Filterung von CAN Botschaften innerhalb des Betriebssystems realisiert werden?

Q5 Können zeitliche Anforderungen von CAN (Transport-)Protokollen innerhalb des Betriebssystems realisiert werden?

## 4.3 Konzeptentwicklung des CAN Zugriffes über Netzwerkschnittstellen

In den folgenden Abschnitten wird die formulierte These auf der Basis der technischen Randbedingungen des Controller Area Networks sowie der Realisierung der Netzwerkschnittstellen unter Linux belegt. Dabei wird nicht auf jedes Detail einer möglichen Implementierung unter Linux eingegangen, um die erarbeiteten Konzepte klar herausstellen zu können. Zusätzlich soll die potenzielle Anwendung des Konzeptes für weitere Mehrbenutzerbetriebssysteme erkennbar bleiben.

Die Darstellung des vorgestellten Konzeptes geht auch nicht im Detail auf die Programmierschnittstelle ein, die dem Anwendungsprogrammierer zur Verfügung gestellt wird. Eine Referenz für die Anwenderprogrammierschnittstelle der Implementierung unter Linux wird im Anhang dieser Arbeit in Kapitel B ausführlich gegeben.

### 4.3.1 Netzwerktreibermodell für CAN Hardware Treiber

Netzwerkkarten für lokale Netzwerke ("Ethernetkarten") übertragen so genannte Ethernetframes ("frame" = Datenrahmen) als Bitstrom auf ein physikalisches Übertragungsmedium nach dem CSMA/CD[6] Zugriffsverfahren.

Die Netzwerkkarte wird vom Betriebssystem dabei als vergleichsweise "dumme" Hardwarekomponente betrachtet, die ausschließlich Ethernetframes senden und empfangen kann, ohne die Dateninhalte des Ethernetframes zu bewerten oder zu verarbeiten. Zwar kann die Ethernetkarte in verschiedenen Modi betrieben werden, welche ihr Verhalten beispielsweise beim Empfang von Ethernetframes beeinflusst. Allerdings sind diese Eigenschaften für die Darstellung des Netzwerktreibermodells nicht relevant.

Die einfach gehaltene Schnittstelle zum Betriebssystem wird über eine Sammlung von Funktionen realisiert, die jeder Netzwerktreiber implementieren muss:

**open()** Zur Aktivierung der bereits initialisierten Hardware

**stop()** Zur Deaktivierung der aktivierten Hardware

**hard_start_xmit()** Zum Senden (Transmit, Abk. xmit) eines Ethernetframes

Die Hardware wird bei der Initialisierung des Treibers bereits initialisiert, so dass durch `open()` beispielsweise die Interrupt-Leitung angefordert wird und die Hardware in den Betriebszustand gesetzt wird.

[6] Carrier Sense Multiple Access / Collision Detection

Der Netzwerktreiber selbst nutzt die vom Betriebssystem zur Verfügung gestellten Funktionen:

**register_netdev()** Zur Registrierung einer zuvor angelegten Treiberstruktur, die das Netzwerkgerät im Betriebssystem bekannt macht. Die Treiberstruktur enthält beispielsweise den Namen des Netzwerkgerätes (“eth0”), den Typ des Netzwerkgerätes (bspw. Ethernet, Appletalk, Tokenring, etc) und die Verweise auf die oben genannten Funktionen `open()`, `stop()` und `hard_start_xmit()`.

**netif_start_queue()** Initialisiert die Warteschlange für zu sendende Ethernetframes.

**netif_stop_queue()** Stoppt die Warteschlange wenn vom Netzwerktreiber aktuell keine neuen Ethernetframes für eine Aussendung verarbeitet werden können.

**netif_wake_queue()** ’Weckt’ die (selbst) gestoppte Warteschlange.

**netif_rx()** Übergibt ein empfangenes Ethernetframe an das Betriebssystem zur weiteren Bearbeitung. Diese Funktion kann direkt aus einem Interrupt-Kontext heraus aufgerufen werden.

Die Warteschlangen werden vom Betriebssystem bereitgestellt (siehe Abbildung 4.1). Zudem existiert für Netzwerkgeräte eine Konfigurationsschnittstelle, die für die Konfiguration CAN spezifischer Einstellungen erweitert werden kann.

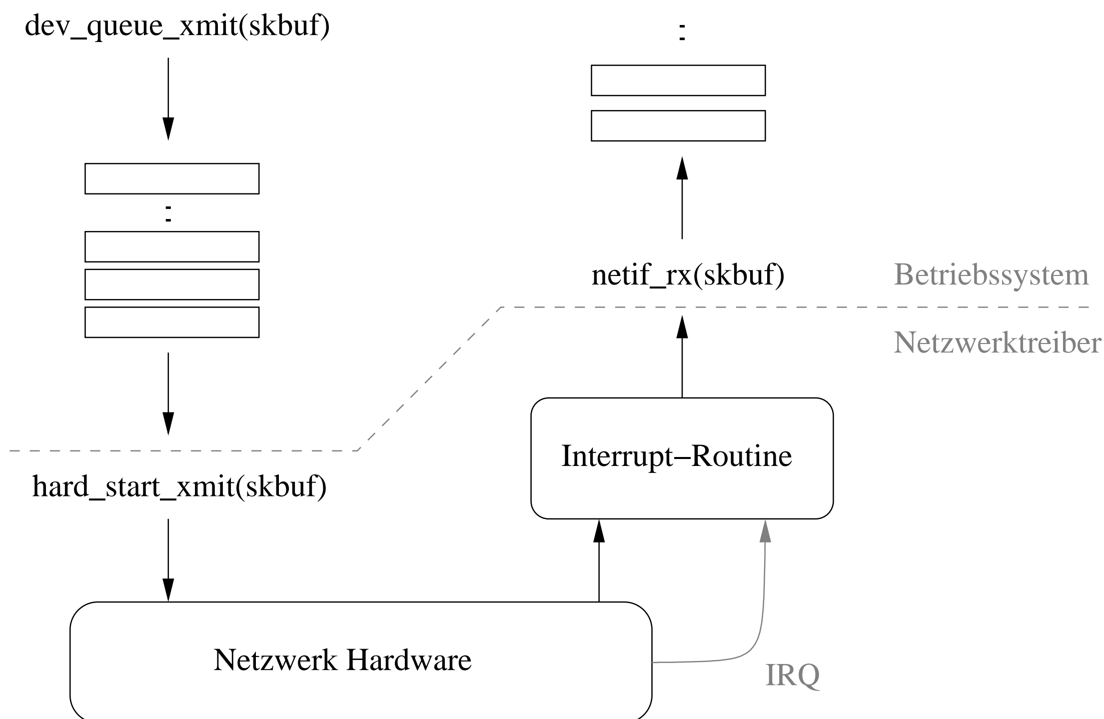



Abbildung 4.1: Konzept der Anbindung von Netzwerkgeräten

Die Übertragung von Netzwerkdateninhalten innerhalb des Linux Kernels wird durch so genannte "Socket-Buffer" realisiert. Die Erzeugung eines Socket-Buffers wird (vereinfacht) beim Senden durch den Anwender initiiert und beim Empfang durch das Netzwerkgerät. Als besondere Eigenschaft des Socket-Buffers nimmt diese Datenstruktur beim Empfang eines Ethernetframes dessen Inhalt vollständig auf. Die im Ethernetframe enthaltenen Informationen wie beispielsweise die Internetprotokoll Adresse wird über Zeiger und Offset-Berechnungen zugreifbar, ohne das dabei Informationen kopiert werden müssen.

Generell verfolgt das Konzept der Socket-Buffer einen "Zero-Copy" Ansatz. Der Socket-Buffer mit dem empfangenen Ethernetframe wird im Normalfall bis zur Übertragung der Anwendungsdaten in den Anwendungskontext nicht im Speicher kopiert. Neben der Tatsache, dass auf die verschiedenen Daten und Adressinhalte über berechnete Offsets innerhalb des Ethernetframes zugegriffen werden kann, wird für eine Weiterleitung des Socket-Buffers grundsätzlich nur ein Zeiger auf diesen weitergegeben. Die Inhalte und die Struktur des Socket-Buffers verbleiben dabei an der Stelle, die beispielsweise bei seiner Erzeugung durch den Netzwerktreiber beim Empfang bestimmt wurde.

### 4.3.2 Behandlung von CAN Nachrichten im Betriebssystem

Die Informationen innerhalb des Ethernetframes, auf welche mit den Offset-Berechnungen innerhalb eines Socket-Buffers zugegriffen werden kann, sind spezifisch für das Ethernetprotokoll. So kann beispielsweise auf die zwei MAC-Adressen, IP-Adressen und weitere Netzwerkinformationen zugegriffen werden. Diese spezifischen Informationen sind in dem Datenrahmen der MAC-Schicht für das Controller Area Network nicht vorhanden.

Wie in Kapitel 2.2.3 beschrieben, besteht ein CAN Frame aus der Sicht des Anwenders aus den Komponenten

- CAN Identifier mit 11 Bit (SFF) oder 29 Bit (EFF) Länge
- Kenzeichnung für die Anforderung von CAN Botschaften (RTR Frames)
- Nutzdaten mit einer Länge von 0 .. 8 Byte

Wenngleich der CAN Identifier in seiner Verwendung als Absenderadresse eine funktionale Ähnlichkeit zu der MAC-Adresse des Ethernetframes aufweist, bleibt die Verwendung des CAN Identifiers im Gegensatz zur MAC-Adresse beim Internetprotokoll aus Anwendungssicht relevant. Protokolle zur Auflösung von Teilnehmernamen im IP-Netzwerk (ARP[7]) oder das Routing von IP-Informationen existieren für das Controller Area Network nicht. Hingegen nutzen verschiedene CAN Protokolle die CAN Identifier als Teil der Protokollinformation, was die direkte Verfügbarkeit des CAN Identifiers für die Anwendung erforderlich macht.

[7] Address Resolution Protocol

Zur korrekten Einbettung eines CAN Frames in einen Socket-Buffer werden daher die Ethernet-spezifischen Informationen und Offsets nicht genutzt und entsprechend initialisiert. Das CAN Frame wird in den Nutzdatenbereich des Socket-Buffers eingetragen, der für die Übertragung des CAN Frames direkt mit einer geringeren Nutzdatenlänge erzeugt wurde. Der Socket-Buffer mit dem darin enthaltenen CAN Frame stellt die Datenstruktur für den Umgang mit CAN Frames in der Linux Netzwerk Software dar.

Zur Unterscheidung verschiedener Typen von Netzwerkhardware und Netzwerkprotokollen sind in der Struktur des Socket-Buffers verschiedene Informationen abgelegt:

- Eine Referenz auf das Netzwerkgerät, über welche beispielsweise der Typ der Netzwerkhardware ermittelbar ist. Beispiele für definierte Netzwerkhardware Typen:

| | | |
|---|---|---|
| ARPHRD_ETHER | 1 | *Ethernet* |
| ARPHRD_APPLETLK | 8 | *Appletalk* |
| ARPHRD_SLIP | 256 | *Serial Line IP* |
| ARPHRD_X25 | 271 | *CCITT X.25* |

- Eine Protokollinformation nach IETF RFC1700 [51], die den Inhalt der Nutzdaten im Socket-Buffer beschreibt. Beispiele für "Ethernet-Types" nach RFC1700:

| | | |
|---|---|---|
| ETH_P_IP | 0x0800 | *Internet IP (IPv4)* |
| ETH_P_X25 | 0x0805 | *CCITT X.25* |
| ETH_P_ARP | 0x0806 | *Address Resolution packet* |
| ETH_P_ATALK | 0x809B | *Appletalk* |

Diese Definitionen sind für eine Unterscheidung der Socket-Buffer Inhalte essentiell, weshalb für die Integration des Controller Area Networks entsprechende Definitionen im Betriebssystem ergänzt werden müssen, um eine korrekte Behandlung von Nutzdaten und Netzwerkgeräten für den CAN Bus im Folgenden zu gewährleisten.

Für das Betriebssystem Linux wurden anhand der üblichen Vergabemodalitäten folgende Hardwaretyp und Ethernettyp Definitionen gewählt:

| | | |
|---|---|---|
| ARPHRD_CAN | 280 | *Controller Area Network* |
| ETH_P_CAN | 0x000C | *Controller Area Network* |

Mit Hilfe dieser Definitionen ist es nun möglich einen Netzwerktreiber für das Controller Area Network zu realisieren, der Socket-Buffer mit darin enthaltenen CAN Frames verarbeiten kann.

Die Definition des zusätzlichen Ethernettyps ETH_P_CAN für Socket-Buffer mit CAN Inhalten sorgt gleichzeitig dafür, dass beispielsweise der Internet-Protokollstapel die über einen CAN Netzwerktreiber empfangenen Informationen nicht erhält. Der Internet-Protokollstapel erhält durch eine entsprechende Registrierung beim Betriebssystem beispielsweise ausschließlich Socket-Buffer mit dem Ethernettyp ETH_P_IP.

#### 4.3.2.1 Die Datenstruktur des CAN Frames

Die für den Anwender sichtbaren Datenstrukturen verlangen besondere Aufmerksamkeit, weil sie die Komplexität der binären Anwendungsschnittstelle prägen, die mit der Aufnahme in ein Betriebssystem "in Stein gemeißelt" werden (D. Miller, Linux Network Maintainer, http://marc.info/?l=linux-netdev&m=119880294531605, 28.12.2007).

Die Definition des CAN Frames ist die zentrale Datenstruktur einer CAN Schnittstelle und sollte daher genau das enthalten, was ein CAN Frame ausmacht:

- CAN Identifier mit 11 Bit (SFF) oder 29 Bit (EFF) Länge
- Kenzeichnung für die Anforderung von CAN Botschaften (RTR Frames)
- Nutzdaten mit einer Länge von 0 .. 8 Byte

Die Datenstrukturen für CAN Nachrichten enthalten bei den betrachteten CAN Treibern nach dem Stand der Technik beispielsweise zusätzlich:

1. Die Mailbox-ID des CAN Controllers über die das CAN Frame empfangen wurde.
2. Einen Zeitstempel, der den Empfangszeitpunkt des CAN Frames markiert.
3. Informationen über den Zustand des CAN Controllers und des CAN Netzwerkes.

Während Punkt 1 aufgrund der Anforderung Verzicht auf spezifische Eigenschaften von CAN Controllern definitiv nicht in das gewählte Konzept passt, sind Punkte 2 und 3 als zusätzliche Anforderung an eine CAN Schnittstelle zu betrachten.

Bei der Analyse der Netzwerk-Kommunikation mit entsprechenden Werkzeugen, werden die empfangenen Ethernetframes mit einem Zeitstempel dargestellt. In der Tat enthält die Datenstruktur des Socket-Buffers bereits die Möglichkeit, einen Zeitstempel in einer Auflösung von einer Nanosekunde abzulegen. Von der Schaffung einer zweiten Infrastruktur zur Übertragung eines Zeitstempels an das Anwendungsprogramm ist daher abzusehen, zumal diese Information unter Umständen nicht für jede Anwendung relevant ist. Die Übertragung potenziell ungenutzter Information ist aus Gründen der Effizienz zu vermeiden.

Bezüglich der Informationen über den Zustand des CAN Controllers und des CAN Netzwerks (Punkt 3) bestehen nach den neuesten Entwicklungen im Linux Betriebssystem verschiedene Möglichkeiten zur Signalisierung solcher Informationen. Für eine Darstellung der Zustandsinformationen in einem Analysewerkzeug analog marktüblicher Anwendungen soll diese Information auf dem selben Kanal an die Anwendung übertragen werden, wie gewöhnliche CAN Frames. Dieser Ansatz der Kodierung von so genannten "Error Messages" in der Form eines CAN Frames erlaubt dabei auch die Übertragung des Konzepts auf weitere Betriebssysteme.

Die Struktur des CAN Frames wird wie folgt definiert:

```
/*
 * CAN Identifier structure
 *
 * bit 0-28     : CAN identifier (11/29 bit)
 * bit 29       : error frame flag (0 = data frame, 1 = error frame)
 * bit 30       : remote transmission request flag (1 = rtr frame)
 * bit 31       : frame format flag (0 = standard 11 bit, 1 = extended 29 bit)
 */
typedef __u32 canid_t;

/**
 * struct can_frame - basic CAN frame structure
 * @can_id:  the CAN ID of the frame and CAN_*_FLAG flags, see above.
 * @can_dlc: the data length field of the CAN frame
 * @data:    the CAN frame payload.
 */
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)));
};
```

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[]` *unabhängig von der Prozessorarchitektur* als ein lineares Array von 8 Byte ausgeführt sind. Da die `data[]` Datenelemente allerdings auf einer 8 Byte Speichergrenze ausgerichtet sind, ist ein 64 Bit Datenzugriff durch den Prozessor möglich:

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

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

Der 64 Bit Zugriff erlaubt beispielsweise in der Programmiersprache C eine 'Struktur' oder eine 'Union' über das 8 Byte Array zu definieren ('casten'). Im Beispiel werden zur Initialisierung der CAN Datenelemente zwei vorzeichenlose 64 Bit Werte zugewiesen.

Die C Typdefinition für den CAN Identifier `canid_t` enthält auch Bitwerte für besondere Kennzeichnungen, wie der Kennzeichnung für die 29 Bit Identifier des Extended Frame Format (EFF), dem Remote Transmission Request (RTR) und der Kennzeichnung für "Error Messages", die zuätzliche Informationen in den Nutzdaten enthalten.

### 4.3.3 Die Grenzen des PACKET Sockets

Mit der Realisierung eines Netzwerktreibers für CAN Hardware im vorherigen Abschnitt besteht die Möglichkeit über einen so genannten PACKET Socket auf das Controller Area Network zuzugreifen. Ein Socket der Protokollfamilie PF_PACKET ist im Gegensatz zur Protokollfamilie PF_INET nicht an die Kommunikationsprotokolle des Internet Protokolls gebunden.

Der Anwender kann einen PACKET Socket mit einem Netzwerkgerät beispielsweise 'eth0' oder 'wlan0' 'binden' und steht dann mit diesem in direkter Verbindung. Auf diese Weise kann der Anwender beispielsweise die Ethernetframes einer Ethnernetkarte empfangen und die darin enthaltenen Informationen selbst auswerten. PACKET Sockets werden beispielsweise von Werkzeugen zur Netzwerkanalyse[8] genutzt. Um die Manipulation von Netzwerkinhalten zu verhindern, sind PACKET Sockets nur mit Administratorrechten nutzbar.

Das folgende Beispiel zeigt im Quelltext[9] das Senden und Empfangen eines CAN Frames auf dem CAN Netzwerkgerät 'can0' mit einem PACKET Socket. Auf die Angabe der einzubindenden Bibliotheksfunktionen und auf die Prüfung von Rückgabewerten im Fehlerfall wurde zur Vereinfachung verzichtet.

```
int s;
struct can_frame frame;
static struct ifreq ifr;
static struct sockaddr_ll sll;

s = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_CAN));

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

sll.sll_family   = AF_PACKET;
sll.sll_ifindex  = ifr.ifr_ifindex;
sll.sll_protocol = htons(ETH_P_CAN);

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

frame.can_id  = 0x123;
frame.can_dlc = 2;
frame.data[0] = 0x11;
frame.data[1] = 0x22;

write(s, &frame, sizeof(struct can_frame));

read(s, &frame, sizeof(struct can_frame));
```

[8]bspw. WireShark/Ethereal http://www.wireshark.com
[9]Quelle: http://svn.berlios.de/svnroot/repos/socketcan/trunk/test/tst-packet.c

Durch die nun vorhandene Möglichkeit eine beliebige Anzahl von Sockets öffnen zu können, wurden die folgenden Anforderungen erfüllt:

**(A7)** Öffnen (Initialisierung) und Schliessen der CAN Schnittstelle

**(A10)** Senden und Empfangen von CAN Nachrichten

**(A12)** Mehrbenutzerfähigkeit

**(A13)** Verwendung von bekannten Programmierschnittstellen

An jedem einzelnen Socket kann der von der CAN Netzwerkkarte empfangene Datenverkehr unabhängig von den anderen Sockets gelesen werden. Dazu wird für jeden geöffneten PACKET Socket von jedem empfangenen Socket-Buffer ein 'Clon' erzeugt, der in die Empfangswarteschlange der Anwendung eingetragen wird. Das Schreiben eines CAN Frames in den Socket führt über die Warteschlange zur Aussendung des CAN Frames auf dem CAN Bus.

Aus der Sicht des Anwendungsprogrammierers stellt sich die Socket-Schnittstelle ähnlich dar wie die Schnittstelle zu einem zeichenorientierten CAN Treiber. Allerdings fehlt eine entscheidende Eigenschaft der bisherigen Treiberschnittstelle:

**(A9)** Konfiguration der Empfangsfilter des CAN Controllers

Wenngleich der PACKET Socket unter Linux die Möglichkeit bietet, durch einen Berkley Packet Filter (BPF)[35] Socket-Buffer Inhalte bereits im Betriebssystemkontext zu filtern, ist dieses Konzept nicht notwendiger Weise auf den anderen betrachteten Mehrbenutzersystemen vorhanden. Im Vergleich zu den Filtermöglichkeiten marktüblicher CAN Hardware ist die Formulierung einer entsprechenden Filterregel für CAN Socket-Buffer für den CAN Anwender außerdem als sehr aufwändig einzustufen.

### Zusammenfassung

Mit einem PACKET Socket besteht die Möglichkeit mit mehreren Anwendungen auf ein CAN Netzwerkgerät zuzugreifen. Allerdings ist mit den Sockets der Protokollfamilie PF_PACKET die Filterung von CAN Frames (Konfiguration der Empfangsfilter des CAN Controllers **(A9)**) nicht betriebssystemunabhängig und anwenderfreundlich realisierbar. Die Implementierung von CAN Kommunikationsprotokollen im Betriebssystemkontext (Multiple Instanzen von Kommunikationsprotokollen **(A5)**) analog zu Verfahren der Protokollfamilie des Internet Protokolls (PF_INET) ist für einen PACKET Socket nicht vorgesehen und damit nicht realisierbar. Als Konsequenz stellt der PACKET Socket keine vollumfängliche Lösung zur Integration des Controller Area Networks in Mehrbenutzerbetriebssysteme dar.

### 4.3.4 Protokollfamilie PF_CAN

Wie in den vorherigen Kapiteln 4.3.2 und 4.3.3 beschrieben, wurde durch die Realisierung eines neu definierten CAN Netzwerktreibers der gleichzeitige Zugriff auf den CAN Bus durch einen PF_PACKET Socket möglich. Der PACKET Socket ist jedoch nicht geeignet, die formulierten Anforderungen an eine Integration des Controller Area Networks in ein Mehrbenutzersystem zu erfüllen.

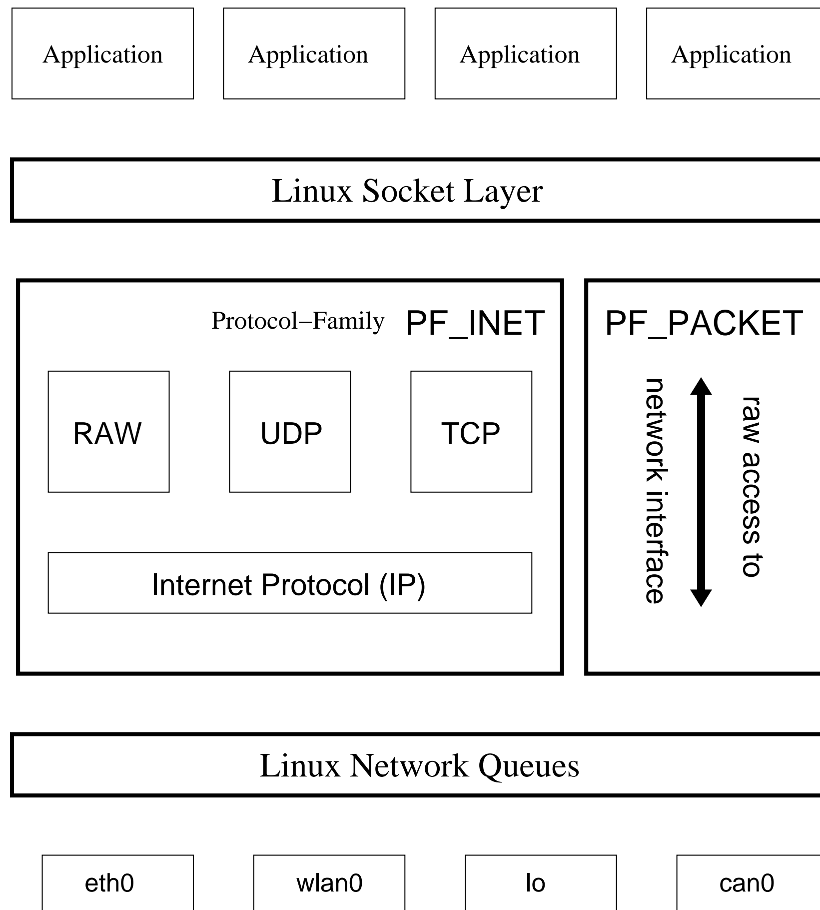



Abbildung 4.2: Protokollfamilien im Linux-Kernel

Tatsächlich erfordert die Simultane Unterstützung verschiedener Kommunikationsprotokolle **(A6)** im Betriebssystemkontext ein Konzept, wie es bei der Protokollfamilie des Internetprotokolls (IP) realisiert ist. Dabei existieren verschiedene Transportprotokolle wie beispielsweise das TCP[10], das UDP[11] oder das SCTP[12], welche

[10]Transmission Control Protocol
[11]User Datagram Protocol
[12]Stream Control Transmission Protocol

dem Anwendungsprogramm als Transportschicht (ISO/OSI Schicht 4) an der Socket-Schnittstelle zur Verfügung stehen.

Die Umsetzung verschiedener verbindungsorientierter (analog TCP/IP) oder verbindungsloser (analog UDP/IP) CAN Transportprotokolle ist daher nur durch ein ähnliches Konzept realisierbar, wie es beim Protokollstapel für das Internetprotokoll Verwendung findet.

Der Internet Protokollstapel registriert sich beim Betriebssystem mit der Angabe eines "Ethernettyps" (siehe Kapitel 4.3.2) für den Empfang von Ethnernetframes, die IP-Pakete enthalten (ETH_P_IP). Die IP-Pakete enthalten wiederum eine Protokollnummer, die die unterschiedlichen Protokolle (TCP, UDP, ...) identifizieren. Mit Hilfe dieser Protokollnummern werden die empfangenen IP-Pakete zur weiteren Verarbeitung an die entsprechenden Protokolle der Protokollfamilie PF_INET weitergeleitet.

Die Internet Protokollfamilie ermöglicht auch den direkten Zugriff auf IP-Pakete über so genannte RAW-Sockets. Die 'rohen' RAW-IP-Sockets ermöglichen das Lesen und Schreiben von IP-Paketen unabhängig von einer IP-Protokollnummer. Wegen der dadurch gegebenen Möglichkeit 'unspezifizierte' oder 'falsche' IP-Pakete erzeugen zu können, kann aus Sicherheitsgründen nur mit entsprechenden Zugriffsrechten ("Netzwerkadministrator") auf diese Art der IP-Sockets zugegriffen werden. Im Gegensatz zu PACKET-Sockets kann bei RAW-IP-Sockets nicht auf das vollständige Ethernetframe inklusive der Ethernetadresse zugegriffen werden sondern nur auf das IP-Datagramm innerhalb des Ethernetframes.

Standardmäßig ist der Zugriff auf den CAN Bus bei der Protokollfamilie PF_CAN ohne besondere Administrator-Zugriffsrechte möglich und ist damit analog dem Zugriff von nicht-privilegierten Anwendungen zur Kommunikation über das TCP/IP Protokoll ausgeführt.

### 4.3.5 Filtern und Verteilen empfangener CAN Nachrichten

Im Gegensatz zur Identifizierung von Internetprotokollen wie TCP oder UDP anhand einer Protokollnummer im IP-Paket, werden CAN Transportprotokolle anhand zuvor definierter CAN Identifier 'erkannt'.

Konkret muss ein CAN-TP Teilnehmer beispielsweise bei der Kommunikation über das ISO15765-2 [27] Protokoll nur wissen, auf welchem (einzelnen) CAN Identifier der Kommunikationspartner seine segmentierten Daten und Protokollinformationen sendet. Der Teilnehmer antwortet dann über einen zweiten CAN Identifier, den der Kommunikationspartner auf der anderen Seite auswerten muss. Zusätzlich ist für die eindeutige Beschreibung von CAN Botschaften noch das verwendete CAN Netzwerkgerät relevant. Eine Botschaft mit dem CAN Identifier 0x123 kann beispielsweise auf dem CAN Bus 'can0' eine andere Bedeutung haben als auf dem CAN Bus 'can1'.

**Dieser signifikante Unterschied zur Identifizierung von Kommunikationsprotokollen beim Internetprotokoll erfordert für die Protokollfamilie PF_CAN eine neue Infrastruktur, bei der sich die verschiedenen CAN Protokolle für den Empfang von bestimmten CAN Nachrichten registrieren können.**

Die neue Infrastruktur zum Verteilen empfangener CAN Nachrichten ist ein integraler Bestandteil der Protokollfamilie PF_CAN. Sie registriert sich beim Betriebssystem für den Empfang von CAN Nachrichten, genauer von Socket-Buffern mit dem Ethernettyp 'ETH_P_CAN', welche ausschließlich von CAN Netzwerkgeräten erzeugt werden.

Beim Empfang eines CAN Frames wird die vom jeweiligen CAN Protokoll im Rahmen der Registrierung angegebene Callback-Funktion[13] in einem Software-Interrupt Kontext (`NET_RX_SOFTIRQ`) aufgerufen. Durch dieses Verfahren wird eine zeitnahe Auslieferung und Verarbeitung der empfangenen Socket-Buffer sichergestellt, ohne dass diese im Interrupt Kontext unmittelbar nach dem Auslesen der Hardware-Register erfolgen muss.

Die Verteilung von CAN Nachrichten durch eine Realisierung im Software-Interrupt wirft die Frage nach einer Benutzung der in CAN Controllern vorhandenen Nachrichtenfilter (Hardware-Akzeptanzfilter) auf.

Die unterschiedlichen Möglichkeiten für die Realisierung von Akzeptanzfiltern in der CAN Hardware steht der Forderung "VERZICHT AUF SPEZIFISCHE EIGENSCHAFTEN VON CAN CONTROLLERN" entgegen. Die in Abbildung 3.2 verglichenen CAN Controller Intel i82527 [21] und NXP SJA1000 [46] weisen in den Möglichkeiten der Hardwarefilterung von CAN Identifiern erhebliche Unterschiede auf. Während der SJA1000 eine Filterung nach der Maskierung empfangener CAN Identifier realisiert (*CAN-ID (logisch UND) Maske: Wenn ungleich Null => Daten weiterleiten*), wird beim i82527 zunächst

[13]Protokollspezifische Funktion, die beim Empfang der registrierten CAN Botschaft aufgerufen wird.

eine für SFF- und EFF-Frames unterschiedliche 'globale' Maske angewendet, wonach zusätzlich auf einzelne CAN Identifier gefiltert werden kann.

Unter der Annahme, dass zumindest ein 'einfacher' CAN-ID/Masken Filter wie beim SJA1000 grundsätzlich bei allen CAN Controllern vorhanden ist, wurde eine Bewertung für eine CAN-ID Hardwarefilterung anhand einer potenziellen Anwendungsanforderung durchgeführt. Sollen mit einem 'einfachen' CAN-ID/Masken Filter die CAN Identifier 0x123, 0x042, 0x624, 0x5A7 und 0x318 an eine CAN Anwendung weitergeleitet werden, reduziert sich die Filterwirkung (siehe Spalte 'verwendbare Filtermaske' in Tabelle 4.1) mit jeder neuen Anforderung signifikant:

| Anforderung | (zus.) benötigte CAN-ID | CAN-ID binär | verwendbare Filtermaske |
|---|---|---|---|
| 1 | 0x123 | 00100100011 | 11111111111 |
| 2 | 0x042 | 00001000010 | 11010011110 |
| 3 | 0x624 | 11000100100 | 00010011000 |
| 4 | 0x5A7 | 10110100111 | 00000011000 |
| 5 | 0x318 | 01100011000 | 00000000000 |

Tabelle 4.1: Verwendbarkeit 'einfacher' CAN-ID/Masken Filter

In dem Beispiel wird sichtbar, das aufgrund einer Auswahl von fünf einzelnen CAN Identifiern die resulierende Filtermaske zu '00000000000' wird, also faktisch keine CAN-ID Hardwareakzeptanzfilterung auf dieser Ebene mehr durchgeführt wird.

Als Konsequenz aus diesem Beispiel und weiteren Betrachtungen, die eine Erweiterung der bislang einfachen Schnittstelle zum CAN Netzwerktreiber erforderlich machen würden, werden CAN Netzwerkgeräte ohne Filterung auf der Hardware- bzw. Treiberebene realisiert. Ein CAN Netzwerkgerät leitet jeglichen empfangenen CAN Datenverkehr ungefiltert an die höheren Protokollschichten (PF_CAN, PF_PACKET) zur Verarbeitung und Filterung im Software-Interrupt weiter.

Die Übertragung des kürzest möglichen CAN Frames (47 Bit) auf einem CAN Bus mit der maximalen Bitrate von 1MBit/s benötigt 47µs. Daraus resultiert eine maximale Rate von 21.276 CAN Frames, die pro Sekunde von einem CAN Controller empfangen werden können. Diese theoretische Anzahl möglicher Unterbrechungsanforderungen (Hardware-Interrupts) wird in realen Anwendungen, beispielsweise im Fahrzeug, nicht auftreten:

- maximale Bitrate im Fahrzeug 500 kBit/s
- Buslast nur zu maximal 50% ausgenutzt
- bis zu 8 Byte Nutzdaten incl. Bit-Stuffing vergrößern die Nachrichtenlänge

Die abschlägig geschätzten 2.500 (ungefilterten) CAN Nachrichten pro Sekunde und CAN Netzwerkgerät stellen dennoch eine Anforderungen an eine effiziente Realisierung für die Filterung im Software-Interrupt dar. Das zu diesem Zweck realisierte Konzept wird im Anhang in Kapitel A.2 detailliert dargestellt.

### 4.3.6 Lokales Echo gesendeter CAN Nachrichten

Für die Anforderung der Mehrbenutzerfähigkeit **(A12)** ist es erforderlich, dass jedes auf dem Mehrbenutzersystem ausgeführte Anwendungsprogramm eine korrekte und identische Darstellung des Datenverkehrs auf dem CAN Bus erhält.

Wie in Kapitel 4.3.1 erläutert, ist das Treibermodell für Netzwerkhardware und dessen Schnittstelle zum Betriebssystem sehr einfach gehalten. Ein Netzwerktreiber sendet Datenrahmen (Frames) aus einer Sendewarteschlange des Betriebssystems auf das Kommunikationsmedium und trägt empfangene Datenrahmen in eine Empfangswarteschlange des Betriebssystems ein.

Sendet beispielsweise eine Anwendung eine CAN Botschaft, ist diese CAN Botschaft für alle CAN Knoten auf dem CAN Bus sichtbar. Da die CAN Botschaft von dem lokalen CAN Controller aber nur *gesendet* und nicht *empfangen* wurde, ist die (gesendete) CAN Botschaft nicht an die CAN Anwendungen auf dem lokalen System weitergeleitet worden. Die CAN Anwendungen auf dem Mehrbenutzerbetriebssystem, welches die CAN Botschaft versendet hat, verfügen in der Folge nicht über eine vollständige Darstellung des vorhandenen CAN Datenverkehrs.

Aus diesem Grunde muss über eine 'lokale Echo' Funktionalität im CAN Netzwerktreiber sichergestellt werden, dass die verschiedenen CAN Anwendungen auf dem System das korrekte Abbild des CAN Datenverkehrs erhalten, auch wenn das System selbst als Sender der CAN Nachricht fungiert. Die Funktionalität im CAN Netzwerktreiber zu realisieren ist erforderlich, um das lokale Echo einer ausgesendeten CAN Botschaft *erst dann* zu erzeugen, wenn es auch fehlerfrei auf dem Medium gesendet werden konnte. Dieses ist besonders vor dem Hintergrund relevant, dass die Aussendung eines CAN Frames auf dem CAN Bus aufgrund der Arbitrierung (siehe Kapitel 2.2.2) durch den Empfang von höher prioren CAN Botschaften (mehrfach) verschoben werden kann.

CAN Adapter, welche aufgrund ihrer Hardwareschnittstelle nicht über die Möglichkeit zur Rückmeldung einer fehlerfreien Aussendung von CAN Frames verfügen, erlauben grundsätzlich keine korrekte Abbildung der *Reihenfolge* der auf dem CAN Bus vorhandenen CAN Botschaften. In diesen Fällen kann die Echo Funktionalität ersatzweise durch die Netzwerkschicht des CAN Subsystems bereitgestellt werden, um die gesendete CAN Botschaft zumindest im lokalen System sichtbar zu machen. Für Anwendungsfälle, bei denen die Darstellung der korrekten Reihenfolge der CAN Botschaften relevant ist, ist eine solche (limitierte) CAN Hardware ungeeignet.

Für eine Unterscheidung der Verfügbarkeit der Echo Funktionalität auf der Treiberebene wird ein neuer Bitwert 'IFF_ECHO' (= Adapter unterstützt die Echo Funktionalität) in den so genannten Interface-Flags von CAN Netzwerkgeräten definiert.

## 4.3.7 CAN Protokolle in der Protokollfamilie PF_CAN

Mit der Definition der Protokollfamilie PF_CAN analog zu Protokollfamilien wie PF_INET wird die Möglichkeit der Realisierung verschiedener CAN Protokolle innerhalb der Protokollfamilie PF_CAN geschaffen.

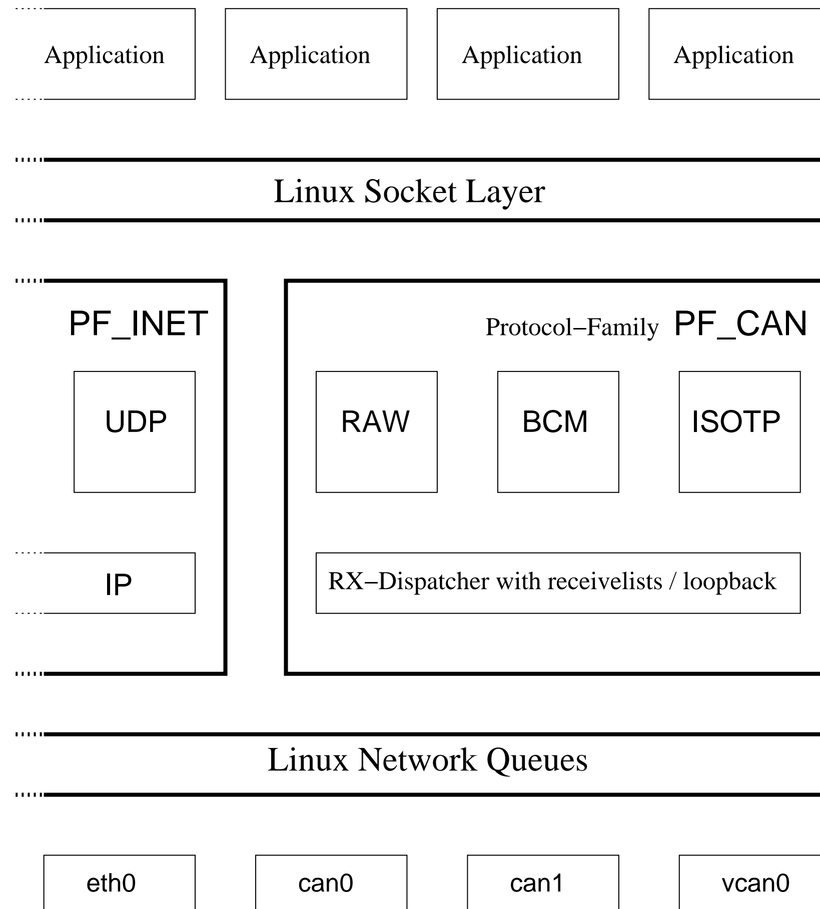



Abbildung 4.3: CAN Protokolle der Protokollfamilie PF_CAN

Wie in den Kapiteln 4.3.5 und 4.3.6 beschrieben, stellt die Protokollfamilie PF_CAN dabei eine CAN spezifische Infrastruktur für CAN Protokolle zur Verfügung.

Zu den besonderen Eigenschaften der Protokollfamilie PF_CAN zählen:

- Filterung und Verteilung empfangener CAN Botschaften im Software-Interrupt
- Sicherstellung einer Echo Funktionalität für gesendete CAN Botschaften
- Einbettung der CAN Protokolle in das Betriebssystem
- CAN Netzwerkgerätespezifische Strukturen mit Hot-(Un-)Plug Unterstützung

Die in Kapitel 4.3.3 (Die Grenzen des PACKET Sockets) dargestellten Probleme bei der Verwendung von PF_PACKET Sockets zum Zugriff auf CAN Netzwerkgeräte werden mit der Einführung der Protokollfamilie PF_CAN gelöst.

#### 4.3.7.1 CAN RAW Protokoll

Im Rahmen der Anforderungsanalyse zur Wiederverwendung von vorhandener CAN Software in Kapitel 4.1.2 ist der direkte Zugriff auf CAN Frames zusammen mit der Konfiguration von Empfangsfiltern als ein übliches Benutzungsszenario herausgestellt worden. Es ist daher analog zur Packet-Schnittstelle des Internetprotokolls (RAW/IP) ein RAW-Socket für die Protokollfamilie PF_CAN zu entwerfen, welcher den direkten Zugriff auf CAN Frames und eine entsprechende Konfiguration von Empfangsfiltern unter Berücksichtigung der Anforderungen an die MEHRBENUTZERFÄHIGKEIT **(A12)** ermöglicht.

Für eine weitgehende Wiederverwendung existierender CAN Anwendungsprogramme ist dabei eine Programmierschnittstelle für die Konfiguration von Empfangsfiltern vorzusehen, die - analog zu bekannten Akzeptanzfiltern von CAN Controllern - eine Empfangsfilterung für CAN Identifier ermöglicht. Die Akzeptanzfilter in CAN Controllern werden beispielsweise beim NXP SJA1000 über maskierte Bitvergleichsoperationen realisiert (siehe [46], Seite 44ff). Die Funktionsweise dieser Akzeptanzfilterung soll im Folgenden erläutert werden:

**recv_id** Empfangener CAN Identifier, für den die Akzeptanzfilterung ausgeführt wird

**can_mask** Binäre Maske zur Definition relevanter Bitwerte in der `can_id` / `recv_id`

**can_id** CAN-ID (Wert), der nach der Maskierung mit `can_mask` vorhanden sein soll

Eine CAN Botschaft kann den gegebenen Akzeptanzfilter passieren, wenn gilt:

$$\texttt{recv_id} \wedge \texttt{can_mask} = \texttt{can_id} \wedge \texttt{can_mask}$$

Beispiele für 11-Bit CAN Identifier mit möglichen Werten von 0x000 bis 0x7FF:

| recv_id | can_mask | can_id | | Akzeptanz des CAN Identifiers recv_id |
|---|---|---|---|---|
| 0x432 | 0x700 | 0x400 | Ja | Es werden IDs von 0x400 - 0x4FF akzeptiert. |
| 0x123 | 0x7FF | 0x123 | Ja | Alle Bits in 'mask' gesetzt: Nur 0x123 akzep. |
| 0x124 | 0x7FF | 0x123 | Nein | Alle Bits in 'mask' gesetzt: Nur 0x123 akzep. |
| 0x320 | 0x7F8 | 0x320 | Ja | Es werden IDs von 0x320 - 0x327 akzeptiert. |
| 0x321 | 0x7F8 | 0x320 | Ja | Es werden IDs von 0x320 - 0x327 akzeptiert. |
| 0x327 | 0x7F8 | 0x320 | Ja | Es werden IDs von 0x320 - 0x327 akzeptiert. |
| 0x328 | 0x7F8 | 0x320 | Nein | Es werden IDs von 0x320 - 0x327 akzeptiert. |
| 0x330 | 0x7F8 | 0x320 | Nein | Es werden IDs von 0x320 - 0x327 akzeptiert. |
| 0x321 | 0x0FF | 0x021 | Ja | Akzeptiert alle IDs, die mit 0x21 enden. |
| 0x321 | 0x000 | 0x000 | Ja | Kein Bit in 'mask' gesetzt: Alle IDs akzeptiert. |

Grundsätzlich erlaubt diese Art der Akzeptanzfilterung bestimmte Bereiche von CAN Identifiern zu filtern, wobei der Bereich so klein gewählt werden kann, dass nur genau *ein* CAN Identifier akzeptiert wird.

Durch den fehlenden Bezug auf eine konkrete CAN Hardware kann diese Art des Akzeptanzfilters für eine Verwendung mit einem CAN Netzwerksocket problemlos erweitert werden:

- Es kann mehr als einen Akzeptanzfilter zu einer Zeit geben
- Es kann ein gegebener Akzeptanzfilter 'invertiert' werden

Die Möglichkeit auf jedem einzelnen RAW-Socket - also unabhängig von anderen CAN Anwendungen - beliebig viele, ggf. invertierte Akzeptanzfilter aktivieren zu können, stellt eine erhebliche funktionale Erweiterung für den Anwender gegenüber bisherigen hardwareabhängigen Filterkonzepten dar. Beim Öffnen eines RAW-Sockets ist standardmäßig ein Filter gesetzt, der alle CAN Identifier akzeptiert.

Zusätzlich zu der Filterung von CAN Botschaften können über eine Socket-spezifische Konfigurationsschnittstelle für jeden RAW-Socket folgende Eigenschaften aktiviert bzw. deaktiviert werden:

- Empfang von Error-Messages (siehe Kapitel 4.3.2)
- Erzeugen eines lokalen Echos für gesendete CAN Botschaften (siehe Kapitel 4.3.6)
- Empfang von lokalen Echos für die von diesem Socket gesendeten CAN Botschaften

Das Erzeugen eines lokalen Echos für gesendete CAN Botschaften ist standardmäßig aktiviert, um die Mehrbenutzerfähigkeit **(A12)** (siehe Kapitel 4.3.6) zu gewährleisten.

Im Unterschied zu dem in Kapitel 4.3.3 beschriebenen PACKET-Socket besteht beim RAW-Socket der Protokollfamilie PF_CAN die Möglichkeit, den Socket an alle vorhandenen CAN Interfaces gleichzeitig zu binden. Dieses bedeutet, dass ein Socket nicht über den so genannten Interface-Index nur CAN Botschaften dieses einen Interfaces empfängt, sondern jegliche CAN Kommunikation im System mitlesen kann. Beim Senden von CAN Botschaften über einen solchen Socket mit unbestimmtem Interface-Index ist daher über einen `sendto()` Systemaufruf der Interface-Index zusätzlich mit anzugeben.

Eine detailierte Beschreibung der Funktionalitäten des RAW-Sockets finden sich im Anhang in Kapitel B.2 (CAN Raw Protokoll Sockets).

#### 4.3.7.2 CAN Broadcast Manager Protokoll

Im Anwendungsfall "Zugriff auf zyklische Fahrzeugdaten" ab Seite 59 werden die Vorteile einer inhaltsbasierten Filterung von CAN Botschaften dargestellt. Bei einer inhaltsbasierten Filterung werden Informationen im bis zu acht Byte umfassenden Nutzdatenbereich des CAN Frames (`data[]`) verarbeitet und beispielsweise mit zuvor empfangenen Inhalten verglichen.

Die im Rahmen der OSEK Communication Specification [43] definierte Programmierschnittstelle ermöglicht eine so genannte 'signalbasierte' Filterung. Als ein 'Signal' wird beispielsweise eine Aussentemperatur oder eine Motordrehzahl verstanden. OSEK COM ermöglicht Filterbedingungen zu definieren, die beispielsweise erfüllt werden, wenn sich die Motortemperatur ausserhalb eines definierten Wertebereiches befindet.

Die OSEK COM Schnittstelle erfordert an dieser Stelle eine genaue Kenntnis über die Definition und Berechnung der ggf. über mehrere Bytes verteilten Signalwerte. Während eine solche Implementierung zur signalbasierten Auswertung im Rahmen einer Quelltextgenerierung für Fahrzeugsteuergeräte ein effizientes Konzept darstellen kann, erfordert eine entsprechende Konfiguration von Fahrzeugsignalen zur Laufzeit eine komplexe Schnittstelle mit einem Interpreter für die definierten Bewertungsregeln.

Zur Reduzierung der Komplexität der Betriebssystemschnittstelle zur inhaltsbasierten Filterung von CAN Botschaften wurde daher auf die bei OSEK COM definierte, abstrakte Signaldarstellung verzichtet. Stattdessen besteht die Möglichkeit eine binäre Filtermaske für den Nutzdatenbereich des CAN Frames zu definieren. Nur bei einer Änderung der empfangenen Daten in dem relevanten (bit-maskierten) Bereich wird die CAN Botschaft an die Anwendung weitergeleitet. Ausschließlich die Anwendung wandelt die empfangenen Informationen in den entsprechenden Signalwert um und führt ggf. weitere Operationen darauf aus.

Das Konzept einer inhaltsbasierten Filterung im Betriebssystemkontext, genauer im Software-Interrupt, ermöglicht weitere Funktionalitäten zum Umgang mit zyklischen CAN Botschaften, welche für eine OSEK COM konforme Anwendungsschnittstelle erforderlich sind.

Für eine vollständige Umsetzung der Communication Conformance Class 1 (CCC1) müssen beispielsweise das so genannte 'Deadline Monitoring' oder verschiedene Modi des 'Periodic Transmission Mode' realisiert werden (siehe [43] Seite 59ff.). Das 'Deadline Monitoring' (siehe auch Seite 59) erfasst beispielsweise das Ausbleiben einer zyklischen CAN Botschaft und der 'Periodic Transmission Mode' ermöglicht, eine CAN Botschaft zyklisch zu senden.

Durch die Verlagerung dieser Funktionalitäten in den Betriebssystemkontext kann zu ihrer Realisierung auf Zeitgeber mit einer Auflösung von einer Nanosekunde[14] und einem geringeren Jitter[15] zurückgegriffen werden, als dieses im Anwendungskontext mit einer Auflösung von (bestenfalls) einer Millisekunde möglich wäre.

Der für die beschriebenen Anwendungsfälle als CAN Protokoll im Betriebssystemkontext implementierte *Broadcast-Manager* realisiert folgende Funktionalitäten:

**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 Identifier Filter)
- Empfangsfilter für Multiplex Botschaften (siehe Kapitel 2.3.1.2)
- Empfangsfilter für die Veränderung vom Botschaftslängen
- Beantworten von RTR-Botschaften
- Timeoutüberwachung von Botschaften
- Reduzierung der Häufigkeit von Änderungsnachrichten (Throttle-Funktion)

Die Konfiguration des *Broadcast-Managers* durch die CAN Anwendung erfolgt über das Schreiben und Lesen von strukturieren Daten auf einem BCM-Socket der Protokollfamilie PF_CAN. Die zyklische Aussendung von CAN Botschaften oder die Überwachung des Empfangs zyklischer Botschaften können durch eine einzelne Konfigurationsnachricht eingerichtet werden. Statusänderungen wie beispielsweise der Ausfall einer überwachten zyklischen CAN Botschaft, werden spontan an die Anwendung gesendet.

Eine detailierte Beschreibung der Funktionalitäten des BCM-Sockets findet sich im Anhang in Kapitel B.3 (CAN Broadcast Manager Protokoll Sockets).

[14]Nutzung von Hardware-Zeitgebern im Prozessorkern, bspw. in x86 und PowerPC Architekturen
[15]dt. etwa 'Phasenrauschen'. Bezeichnet die Genauigkeitsschwankung bei einem gegebenen Takt.

#### 4.3.7.3 CAN Transport Protokolle

Die Realisierung von CAN Transportprotokollen im Rahmen einer PF_CAN Protokollfamilie bietet sich aus folgenden Gründen an:

- Die durch ein CAN Transportprotokoll vorgegebenen zeitlichen Anforderungen können im Betriebssystemkontext erfüllt werden (Nachweis siehe Kapitel 5.1.4). Im Gegensatz dazu ist eine Implementierung im Anwendungskontext durch das Scheduling der Anwendungsprozesse funktional beschränkt (siehe Kapitel 3.6).
- Die Socket-Schnittstelle ist eine bekannte Netzwerk-Programmierschnittstelle für die Übertragung von Anwendungsdaten, beispielsweise für TCP/IP oder UDP/IP.
- Die Funktion zum Erzeugen eines Sockets erfordert drei Parameter:

  **domain** für die Protokollfamilie, beispielsweise PF_INET oder PF_CAN.

  **type** für den Protokolltyp, beispielsweise SOCK_STREAM, SOCK_DGRAM oder SOCK_SEQPACKET

  **protocol** für eine genaue Bezeichnung des Transportprotokolls

  Diese Unterteilung ermöglicht eine umfassende Abbildung der verschiedenen CAN Transportprotokolle zur Punkt-zu-Punkt Kommunikation in CAN Netzwerken.

Aus der Sicht eines Anwendungsprogrammierers sollte bei der Verwendung der Socket-Schnittstelle das verwendete Medium für eine Kommunikation über ein Netzwerk transparent sein. Dieses bedeutet, dass die Programmierschnittstelle für einen Socket verbirgt, ob eine Punkt-zu-Punkt-Kommunikationsverbindung zu einem WWW-Server in Japan besteht oder eine Diagnose-Verbindung zu einem Motorsteuergerät hergestellt wurde.

Die Problemstellung besteht bei der Realisierung von CAN Transportprotokollen als Protokoll innerhalb der Protokollfamilie PF_CAN darin, eine Analogie von CAN Protokollen zu Internetprotokollen herzustellen, die in ihrer Anwendung wohldefiniert, bekannt und etabliert sind.

In Kapitel 2.3.2 (CAN basierte Transportprotokolle) ist ausführlich auf verschiedene CAN Transportprotokolle eingegangen worden, wobei die Unterschiede von verbindungsorientierten Stream-Sockets (siehe Anhang, Kapitel B.4) und verbindungslosen Datagramm-Sockets (siehe Anhang, Kapitel B.5) erläutert wurden.

| Sockettyp | Internetprotokolle (PF_INET) | CAN Protokolle (PF_CAN) |
|---|---|---|
| Stream-Socket | TCP/IP | VAG TP1.6, VAG TP2.0, Bosch MCNet |
| Datagramm-Socket | UDP/IP | ISO 15765-2 |

Tabelle 4.2: Analogie zwischen Internet- und CAN-Sockettypen

Bei der Benutzung der verschiedenen Sockettypen kommen bei der Initiierung einer Kommunikationsverbindung verschiedene Betriebssystemaufrufe zum Einsatz:

| Sockettyp | Betriebssystemaufrufe |
|---|---|
| Stream-Socket (Server) | bind(2), listen(2), accept(2) |
| Stream-Socket (Client) | connect(2) |
| Datagramm-Socket | bind(2) |

Tabelle 4.3: Betriebssystemaufrufe für verschiedene Sockettypen

Das aufwändige Verfahren für den Stream-Socketserver ergibt sich aus der Tatsache, dass sich beim Internetprotokoll ein Socketserver mit bind(2) auf eine Internetprotokoll Portnummer registriert und dann mit listen(2) auf n ($n \in \mathbb{N}$) eingehende Client-Verbindungen wartet. Mit der Ausführung des Systemaufrufes accept(2) wartet der Server blockierend auf eine eingehende, durch einen Client initiierte Verbindung. Der Rückgabewert von accept(2) enthält den Dateideskriptor für die etablierte Kommunikationsverbindung.

Bei einer Adaptierung dieses Verfahrens an CAN Transportprotokolle ergeben sich folgende Änderungen, die durch die unterschiedliche Adressierungsform (mit CAN Identifiern) begründet sind:

- Die Adressierung auf dem CAN Bus kennt keine 'Portnummern'
- Ein CAN Transportkanal verwendet genau zwei definierte CAN Identifier.

  Als Folge daraus kann ein CAN Transportkanal zu einer Zeit nur ein Mal auf dem CAN Bus etabliert sein und der Systemaufruf listen(2) für CAN Transportkanäle nur auf *genau eine* eingehende Client-Verbindung warten ($n = 1$).

Diese Unterschiede in der Adressierung erfordern eine CAN spezifische Adress-Struktur vom Typ `struct sockaddr_can` , die im Gegensatz zu TCP/IP nicht eine Adresse und eine Portnummer enthält, sondern immer die 'Adressen' (CAN Identifier) beider Kommunikationspartner.

Beim Aufruf der Funktionen bind(2) oder connect(2) wird für CAN Transportprotokolle eine Adress-Struktur `struct sockaddr_can` übergeben, die die CAN Identifier der Kommunikationspartner enthält. Die bekannte Adress-Struktur vom Internetprotokoll (`struct sockaddr_in`) findet bei der Protokollfamilie PF_CAN keine Verwendung.

Details und Beispiele zur Verwendung von CAN Sockets zur Realisierung von CAN Transportprotokollen finden sich im Anhang in den Kapiteln B.4 (CAN Transportprotokoll Sockets (Stream)) und B.5 (CAN Transportprotokoll Sockets (Datagram)).

## 4.4 Zusammenfassung

In diesem Kapitel wurde dargestellt, dass der CAN Zugriff über Zeichenorientierte Betriebssystemschnittstellen (siehe Abschnitt 4.2.1) nach dem Stand der Technik nicht geeignet ist, die auf Seite 64 zusammengefassten Anforderungen an eine Integration des Controller Area Networks in Mehrbenutzersysteme zu erfüllen.

Alternativ wurden die im Rahmen der "Realisierung von vorhandenen Anwendungen" ab Seite 59 identifizierten, funktionalen Anforderungen versuchsweise mit Konzepten aus der Netzwerkprogrammierung abgebildet. Die dabei durch die versuchsweise verwendete Internetprotokollfamilie vorgegebenen Randbedingungen ließen sich auf die Anforderungen des Controller Area Networks nicht ausnahmslos abbilden, weshalb eine neue Netzwerkprotokollfamilie PF_CAN für das Controller Area Network eingeführt wurde.

Zusammenfassend belegt das in diesem Kapitel in Abschnitt 4.3 vorgestellte Konzept zur Integration des Controller Area Networks in der Form einer Netzwerkprotokollfamilie PF_CAN die auf Seite 71 formulierte These.

Die zur Falsifikation der These auf Seite 71 formulierten Fragestellungen Q1 - Q5 können positiv beantwortet werden.

- Q1 Das Netzwerktreiber Modell ist für die CAN Hardware adaptierbar (siehe Abschnitte 4.3.1 und 4.3.2 ab Seite 72).
- Q2 Eine systemweite Schnittstelle zur Konfiguration der CAN Hardware durch den Systemadministrator (beipsielsweise der Bitrate) ist realisierbar (siehe Linux Kernel Dokumentation [19] `"6.5 The CAN network device driver interface"`).
- Q3 Eine Möglichkeit zur inhaltsbasierten Filterung von CAN Botschaften innerhalb des Betriebssystems ist realisierbar (siehe Abschnitt 4.3.7.2 ab Seite 88).
- Q4 Eine Möglichkeit zur CAN Identifier basierten Filterung von CAN Botschaften innerhalb des Betriebssystems ist realisierbar (siehe Abschnitt 4.3.7.1 ab Seite 86).
- Q5 Grundsätzlich sind CAN (Transport-)Protokolle analog zu anderen Kommunikationsprotokollen innerhalb des Betriebssystems in der dargestellen Form realisierbar (siehe Abschnitt 4.3.7.3 ab Seite 90). Die Einhaltung der zeitlichen Anforderungen von konkreten CAN (Transport-)Protokollen wird in der Performanzbewertung in Kapitel 5.1.4 detailliert dargestellt.

Tatsächlich stellt sich die Realisierung von Funktionalitäten innerhalb des Betriebssystemkontext als ein Lösungsansatz dar, der die Verwendung von *aktiver* CAN Hardware zur Realisierung von zeitlichen Anforderungen obsolet machen kann. Sind auf den betrachteten eingebetteten Prozessoren bereits *passive* CAN Controller integriert, könnte ein Konzept mit zusätzlicher *aktiver* CAN Hardware zur Sicherstellung von zeitlichen Anforderungen aus Komplexitäts- und Kostengründen vermieden werden (siehe Problemexposition beim Stand der Technik in Kapitel 3.7).

Das vorgestellte Konzept erfüllt drei elementare Anforderungen:

1. Der leistungsfähige eingebettete Prozessor übernimmt Aufgaben, die bisher nur durch die Prozessoren auf der *aktiven* CAN Hardware erfüllt werden konnten.
2. Es wird durch die Socket Schnittstelle eine Mehrbenutzerfähigkeit hergestellt, die auf Basis einer einzigen Protokollimplementierung beliebig viele Instanzen von Kommunikationsanforderungen nebenläufig realisieren kann.
3. Durch die Verwendung des einfachen Netzwerktreibermodells für die Anbindung von CAN Hardware wird eine Hardwareabstraktion geschaffen, die die Abhängigkeit der CAN Anwendung von einer konkreten CAN Hardware auflöst. Die daraus resultierenden Folgen im Zusammenhang mit CAN Hardwareherstellern werden in Kapitel 6.1 detailliert.

Die Möglichkeit, eine unbeschränkte Anzahl von Akzeptanzfiltern auf jedem einzelnen RAW-Socket konfigurieren zu können, erlaubt einem CAN Anwender die einfache Portierung seiner Software, die bisher beispielsweise auf eine zeichenbasierte Betriebssystemschnittstelle mit einem einzelnen Hardwareakzeptanzfilter aufgesetzt hat.

Selbst die bisher noch nicht betrachtete Anforderung, eine Rückmeldung für die erfolgte Aussendung einer CAN Botschaft zu erhalten, ließe sich über die Echo-Funktionalität des CAN Netzwerkgerätes in Zusammenhang mit der Möglichkeit realisieren, die eigenen ausgesendeten CAN Botschaften auf einem RAW-Socket zu empfangen. Grundsätzlich ist diese Anforderung allerdings als fragwürdig einzustufen, weil damit eine vermeintliche Sicherheit im Kommunikationsverfahren suggeriert wird, welche ohnehin auf der Anwendungsebene durch andere Verfahren gewährleistet werden müsste.

Als ein potenzieller Nachteil des vorgestellten Konzeptes kann der Betrieb der CAN Controller ohne Hardwareakzeptanzfilter gesehen werden. Dieser für einen Mehrbenutzerbetrieb sinnvolle Ansatz, die CAN Controller 'offen' zu betreiben, kann für jeden einzelnen CAN Controller zu einer theoretischen Unterbrechungsanforderungsrate von bis zu 21.276 Unterbrechungen (Interrupts) pro Sekunde führen (1 MBit/s / 47 Bits für das kürzest mögliche CAN Frame). Wenngleich aktuelle Ethernetkarten mit einer Datenrate von 1 GBit/s bei voller Ethernetframelänge ca. 81.000 Interrupts pro Sekunde erzeugen (siehe [22] auf Seite 14), ist eine genaue Bewertung des vorgestellten Konzeptes zu diesem Punkt notwendig, welche in Kapitel 5 (Bewertung) erfolgen wird.

Eine Beschreibung der Implementierung des Filterkonzeptes im Software-Interrupt sowie weitere Details der Linux Implementierung finden sich im Anhang A ab Seite 135.

# 5 Bewertung

In diesem Kapitel wird das erarbeitete Konzept im Hinblick auf seine Performanz und seine Transferierbarkeit in weitere Betriebssysteme und Projekte bewertet. Dazu werden in Abschnitt 5.1 anhand verschiedener, in der Arbeit betrachteter Anwendungsszenarien Performanzvergleiche mit dem Stand der Technik durchgeführt. In Abschnitt 5.2 wird die Anwendung des erarbeiteten, allgemeingültigen Konzeptes zur Integration von Feldbussen in Mehrbenutzerbetriebssysteme auf weitere Betriebssysteme (wie beispielsweise Microsoft Windows) und auf ein eingebettetes 8-Bit Mikrocontrollersystem dargestellt.

## 5.1 Performanzvergleich zum Stand der Technik

Das beschriebene Konzept stellt eine erhebliche Erweiterung des Standes der Technik in Bezug auf Abstraktion und Mehrbenutzerfähigkeit dar. Die angebotene Funktionalität beim Filtern von CAN Botschaften sowie die Möglichkeit mit beliebig vielen Anwendungsprogrammen mit wiederum beliebig vielen Socket-Instanzen auf den CAN Bus zugreifen zu können, erzeugt die Fragestellung nach der Performanz der gefundenen Lösung.

Zur Beantwortung dieser Fragestellung werden in diesem Abschnitt Anwendungsfälle bewertet, deren Verhalten aus dem Stand der Technik bekannt sind und die somit einen messbaren Vergleich zum erarbeiteten Konzept ermöglichen.

Für eine Vergleichbarkeit müssen beide Ansätze auf der selben Hardware, mit dem selben Betriebssystem und der selben Funktionalität gegeneinander geprüft werden. Diese Anforderungen schließen die Verwendung von *aktiver* CAN Hardware beim Performanzvergleich aus.

Für die Messungen in Abschnitt 5.1.1 wurde beispielsweise eine *passive* PEAK PCMCIA CAN Karte IPEH-002091 eingesetzt. Die verwendeten SJA1000 CAN Controller [46] werden bei dieser CAN Karte linear in den Adressraum des Personalcomputers eingeblendet und durch die exklusive Zuweisung eines Interrupts an den PCMCIA Schacht können Abhängigkeiten zu weiterer PC-Hardware im System ausgeschlossen werden.

Zusätzlich realisiert der in der Messung verwendete CAN Treiber der Firma PEAK sowohl das zeichenorientierte Treibermodell als auch die vom Autor dieser Arbeit beigetragene Unterstützung für das Netzwerktreibermodell, wobei sich der Quellcode zur Behandlung von Unterbrechungsanforderungen und zum Zugriff auf die Register des SJA1000 Controllers nicht unterscheidet. Dieser Treiber stellt aufgrund der identischen Basis für den Vergleich der beiden Realisierungsansätze eine optimale Testumgebung dar. Eine Darstellung der Kontroll- und Datenflüsse beider Realisierungsansätze kann der Abbildung 5.1 entnommen werden.

### 5.1.1 Ungefilterte CAN Nachrichten

Die Übertragung ungefilterer CAN Nachrichten von einem einzelnen CAN Controller zu einer einzelnen CAN Anwendung stellt im Vergleich zum Stand der Technik für das erarbeitete Netzwerk-Konzept das schlechtest mögliche Anwendungsszenario dar.

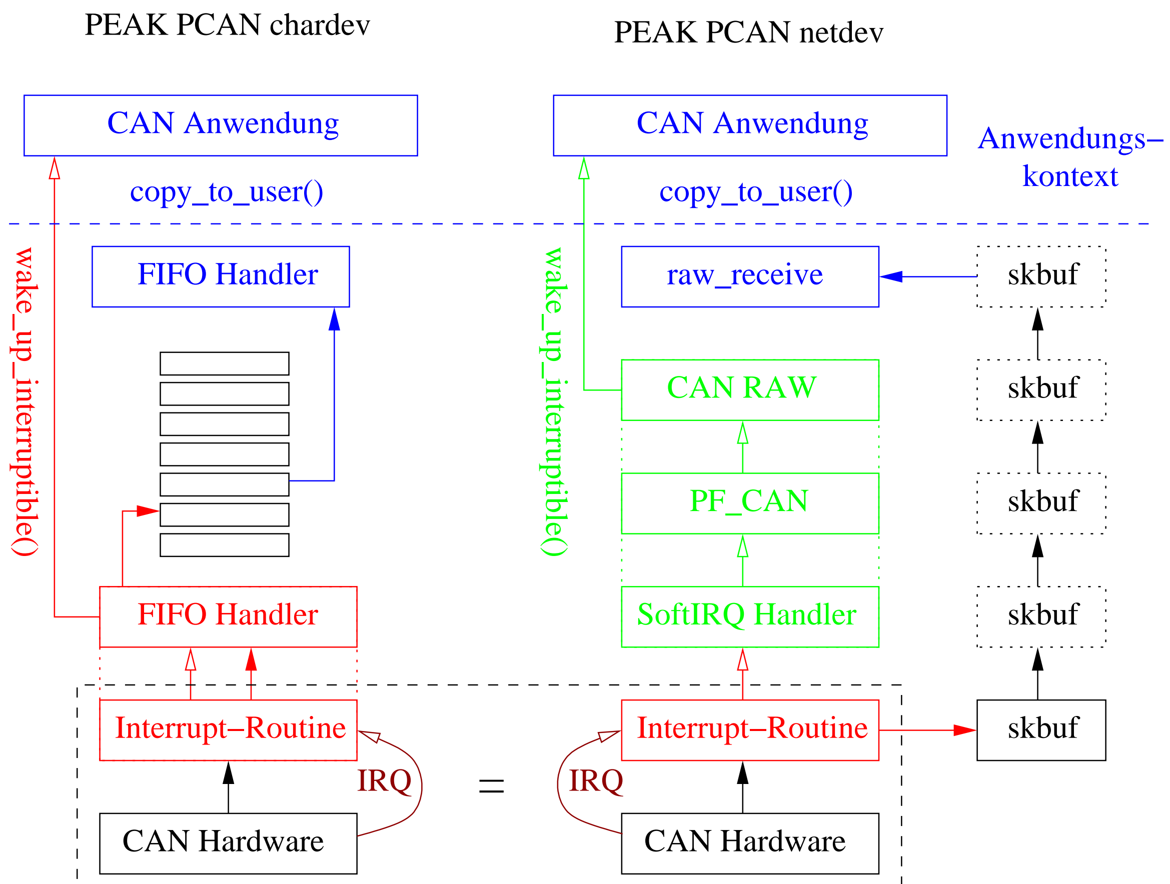



Abbildung 5.1: Testumgebung PEAK PCMCIA CAN Treiber

Das Netzwerk-Konzept nutzt die vom Betriebssystem bereitgestellten Funktionalitäten:

- Dynamische Allozierung von Socket-Buffern (Kapitel 4.3.2) im Hardware-Interrupt
- Verarbeitung der CAN Frames im Software-Interrupt
  - Verteilung der empfangenen CAN Frames an den jeweiligen Socket
  - Variable Länge von Empfangswarteschlangen am jeweiligen Socket
- Freigabe der dynamisch allozierten Socket-Buffer nach Übertragung des CAN Frames an die CAN Anwendung (in den Anwendungskontext)

Für das Einbenutzer-Anwendungsszenario wird für diese Messung die im Netzwerk-Konzept realisierte, universelle Steuerung der Kontroll- und Datenflüsse nicht benötigt.

Die Realisierung des zeichenorientierten PEAK Treibers orientiert sich denn auch an den Anforderungen für eine einzelne CAN Anwendung:

- Ein exklusiver FIFO[1] Puffer zwischen CAN Controller und CAN Anwendung
- Feste Größe des FIFO Empfangspuffers (aktuell 500 CAN Botschaften)
- Zugriff über eine abstrakte Programmierschnittstelle 'FIFO Handler'
- Keine Nutzung von dynamischem Speicher

Die Komplexität der Abläufe bei der zeichenorientierten Schnittstelle ist offensichtlich geringer einzustufen als bei der Übertragung eines CAN Frames beim Netzwerk-Konzept. Die Abbildung 5.1 zeigt auf der linken Seite den zeichenorientierten Ansatz und auf der rechten Seite den netzwerkorientierten Ansatz. Eine Analyse des Quelltextes für den 'FIFO Handler' im PEAK-Treiber hat ergeben, dass der Zugriff auf den FIFO mit dem entsprechenden Locking für den konkurierenden Zugriff auf gemeinsame Speicherbereiche nach dem Stand der Technik ausgeführt wurde.

**Annahme:** *Aufgrund der geringeren Komplexität der FIFO-Lösung bei zusätzlicher Verwendung gleicher Programmcode Bestandteile (Behandlung von Unterbrechungsanforderungen und Übertragen von CAN Frames in den Anwendungskontext) kann von einem performanteren Verhalten der zeichenorientierten Lösung ausgegangen werden. Gestützt wird diese Annahme durch einen alternativen Performanzvergleich [59], welcher am Ende dieses Abschnitts zusätzlich bewertet wird.*

Zur Verifikation der formulierten Annahme wurde ein etabliertes Verfahren zur Messung von Ausführungszeiten innerhalb des Linux Betriebssystems eingesetzt:

Das **LTTng** (Linux Trace Tool next generation - `http://www.lttng.org`) ist ein so genannter Kernel Tracer, der es ermöglicht über präzise Zeitstempel konkurrierende

[1] First-In-First-Out Datenstruktur

Abläufe im Linux Betriebssystem zu verfolgen. Auf diese Weise ist es möglich Performanzprobleme und Probleme mit simultan ausgeführtem Programmcode (bspw. auf Mehrprozessorsystemen) zu analysieren. Die Abbildung 5.2 zeigt die grafische Oberfläche des LTTng (Quelle: `http://www.lttng.org/sites/default/files/resource1.png`) mit der die aufgezeichneten Daten für eine Analyse visualisiert werden können.

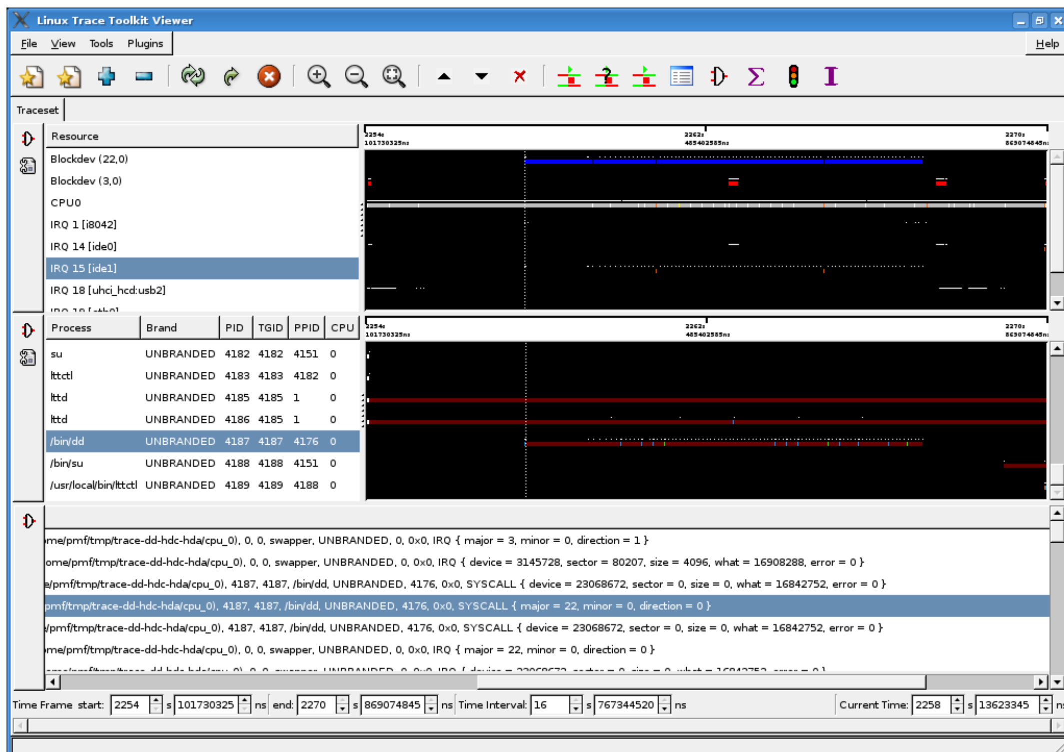



Abbildung 5.2: Grafische Oberfläche von LTTng

Für die Messung wurde ein für LTTng instrumentierter Linux Kernel in der Version 2.6.30 von einem öffentlichen git-Repository auf www.kernel.org genutzt:

`http://git.kernel.org/?p=linux/kernel/git/compudj/linux-2.6-lttng.git;a=shortlog;h=2.6.30-lttng-0.154`

Nach der Installation des LTTng Kernels wird mit

```
modprobe kernel-trace
modprobe net-trace
modprobe fs-trace
mkdir /mnt/debugfs
mount -t debugfs none /mnt/debugfs
```

die benötigte LTTng Testumgebung hergestellt.

Zur Aktivierung der Testmarker für die Messung werden gemäß Abbildung 5.3 mit dem Befehl `echo 1 > [testmarker]` folgende Testmarker aktiviert:

**C1** /mnt/debugfs/ltt/markers/kernel/irq_entry/enable

**C2** /mnt/debugfs/ltt/markers/fs/ioctl/enable

**N1** /mnt/debugfs/ltt/markers/kernel/irq_entry/enable

**N2** /mnt/debugfs/ltt/markers/net/dev_receive/enable

**N3** /mnt/debugfs/ltt/markers/fs/read/enable

Der Messpunkt **N2** ist für den durchzuführenden Vergleich nicht zwingend erforderlich. Zwischen **N1** und **N2** wird ein Socket-Buffer (siehe Kapitel 4.3.2) für das empfangene CAN Frame im Unterbrechungskontext dynamisch alloziert.

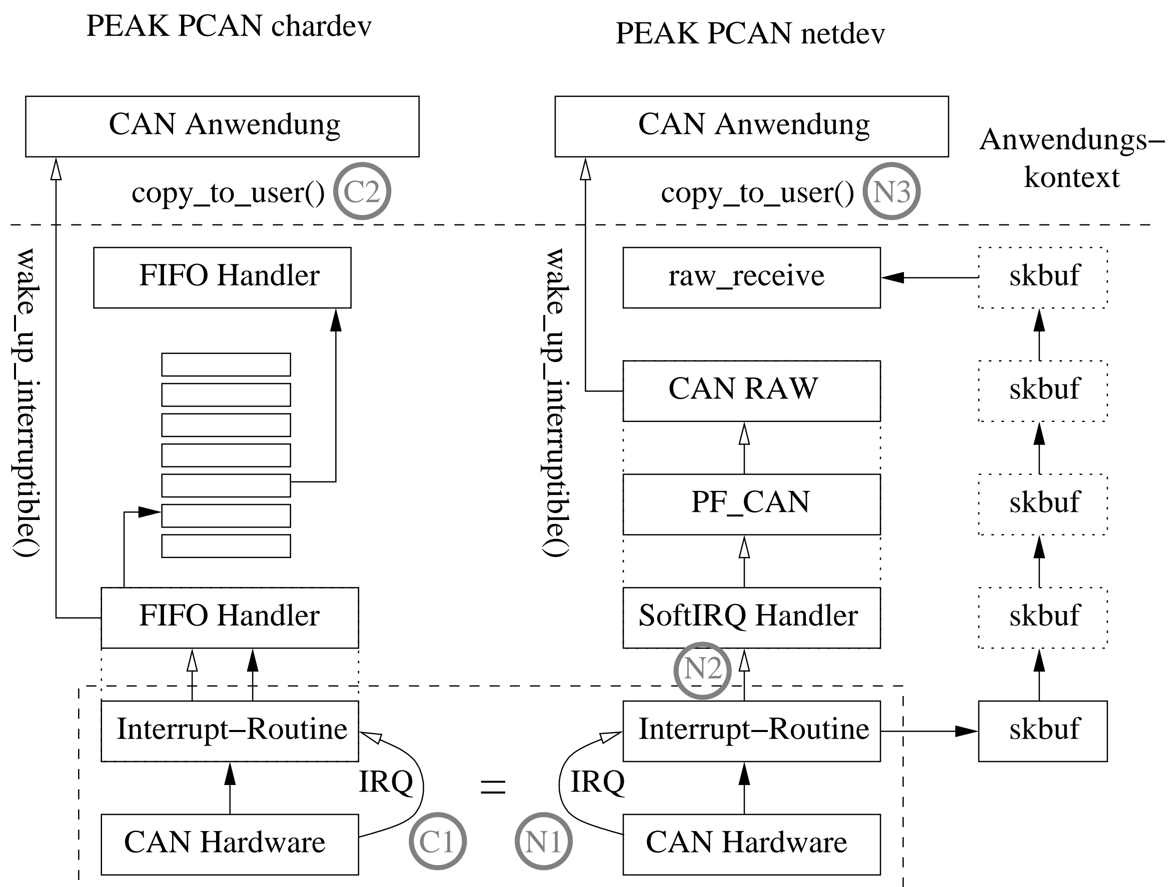



Abbildung 5.3: LTTng Messpunkte am PEAK PCMCIA CAN Treiber

Als CAN Treiber wurde der PEAK PCAN Treiber in der Version 6.11 eingesetzt (siehe http://www.peak-system.com/linux) und für die Messungen jeweils mit und ohne Netzwerktreiberunterstützung übersetzt:

```
make KERNEL_LOCATION=/home/hartko/linux-2.6-lttng NET=NETDEV_SUPPORT
make KERNEL_LOCATION=/home/hartko/linux-2.6-lttng NET=NO_NETDEV_SUPPORT
```

Als CAN Anwendungsprogramme wurden das Werkzeug **candump** (siehe Kapitel C.1.1) für die Netzwerkvariante und das mit dem PCAN Treiber ausgelieferte Werkzeug **receivetest** der Firma PEAK System für die zeichenorientierte Variante eingesetzt.

| Messung | Durchschnitt | min | max | Standardabweichung | Anzahl Messwerte |
|---|---|---|---|---|---|
| C1 → C2 | 112µs | 67µs | 188µs | 22,4µs | 390 |
| N1 → N3 | 93µs | 61µs | 160µs | 9,8µs | 712 |
| N2 → N3 | 26µs | 11µs | 90µs | 4,8µs | 712 |
| N1 → N2 | 67µs | 54µs | 92µs | 7,2µs | 712 |

Tabelle 5.1: Performanzmessung: 'C'hardev vs 'N'etdev

Entgegen der Annahme auf Seite 97 zeigen die Messergebnisse beim PCAN Treiber ein performanteres Verhalten des Netzwerk-Konzeptes. Ein Laufzeitvorteil für das mehrbenutzerfähige Konzept war vor dem Hintergrund der beschriebenen Komplexität bei dem Einbenutzer-Anwendungsszenario nicht zu erwarten. Besonders auffällig ist die gemessene Standardabweichung, die beim zeichenorientierten Konzept mit 22,4µs mehr als doppelt so hoch ausfällt, wie der entsprechende Wert für das Netzwerk-Konzept von 9,8µs.

Dieses unregelmäßigere Verhalten des zeichenorientierten Konzeptes ist auf den konkurrierenden Zugriff auf den FIFO Speicher für empfangene CAN Botschaften zurückzuführen. Das Beschreiben des FIFO im Unterbrechungskontext (CAN-Treiber) sowie das Lesen aus dem FIFO aus dem Anwendungskontext erfordern jeweils ein Sperren der Interrupts für das gesamte System mit der Linux-Kernel Funktion `spin_lock_irqsave()`.

Im Gegensatz zur dargestellten FIFO Implementierung beim zeichenorientierten Konzept realisiert das Netzwerk-Konzept die Bearbeitung eines empfangenen CAN Frames überwiegend im Software-Interrupt-Kontext (siehe grün dargestellte Funktionsblöcke in Abbildung 5.1 auf Seite 96). Die Allokation des Datenspeichers für die jeweilige CAN Botschaft mit einer einmaligen, geringen Verweildauer im Unterbrechungskontext erübrigt das Sperren der Interrupts bei der Übertragung der CAN Botschaft in den Anwendungskontext vollständig.

Performante Netzwerk-Implementierungen sind eine grundsätzliche Anforderung an Mehrbenutzerbetriebssysteme. Die dafür in den Betriebssystemen realisierten, fein-granularen per-CPU Locking-Verfahren und Caching-Algorithmen für Socket-Buffer kommen offenbar auch der Performanz des dargestellten CAN Netzwerk-Konzeptes zugute.

Zu dieser Einschätzung kommt auch ein alternativer Performanzvergleich [59] zwischen dem in Kapitel 4.2.1 diskutierten, zeichenbasierten LinCAN Treiber und dem in dieser Arbeit beschriebenen Netzwerk-Konzept. Jene Messungen sind unter vergleichbaren Testbedingungen mit einer *passiven* PCI CAN Hardware durchgeführt worden. Die durchgeführten Messungen in [59] weisen folgende Unterschiede zur dargestellten Vergleichsmessung mit dem PCAN Treiber auf:

- Es wurden unterschiedliche Linux Kernelversionen betrachtet.

  **2.6.26-2-686** Standard Kernel der Debian 5.0 Linux Distribution

  **2.6.29.4** Standard Kernel, welcher als Basis für Echtzeiterweiterungen dient

  **2.6.29.4-rt16** Linux Kernel 2.6.29.4 mit integrierten Echtzeiterweiterungen

  **2.6.31-rc7** Standard Kernel in neuerer Revision

- Es wurden unterschiedliche CAN Treiber verwendet.

  **Zeichentreiber** LinCAN Treiber (CVS Revision 2009-06-11)

  **Netzwerktreiber** SocketCAN Treiber (SVN Revision 1009)

- Es wurde eine vollständige Round-Trip-Time (RTT) Messung durchgeführt, welche die Zeit vom Senden einer CAN Botschaft aus dem Anwendungskontext bis zum Empfang dieser CAN Botschaft im Anwendungskontext umfasst.

Im Rahmen der Messungen ergibt sich für die betrachteten Linux Kernelversionen ohne Echtzeiterweiterungen zunächst für den LinCAN Treiber ein zeitlicher Vorteil. Der LinCAN Treiber verfügt über eine effizientere Implementierung der Empfangspuffer Behandlung als der PEAK Treiber und verzichtet auf ein aufwändiges Locking beim Ausliefern der CAN Frames in den Anwendungskontext. Der LinCAN Treiber erfüllt damit die auf Seite 97 formulierte Annahme. Die Behandlung der empfangenen CAN Frames durch den Software-Interrupt beim Netzwerk-Konzept ist anhand der Messergebnisse verifizierbar, ebenso wie die Tatsache, dass ein zusätzliches Netzwerkdatenaufkommen (beispielsweise durch Ethernetkarten) die Laufzeit von CAN Nachrichten verlängern kann.

Die Messungen an dem im Vergleich aktuellsten Linux Kernel 2.6.31-rc7 zeigen jedoch eine Performanz des Netzwerk-Konzeptes, die mit dem zeichenbasierten LinCAN Treiber vergleichbar ist. Dieses ist - wie oben erwähnt - auf neue Konzepte beim Caching und beim fein-granularen Locking im Netzwerksubsystem zurückzuführen. Zitat aus [59]:

> In most cases the character device approach has lower overhead but it seems that recent improvements in Linux kernel decrease the overhead of the network device approach. One area where character based drivers still win is when the system is loaded by receiving non-CAN traffic.

Die Auswirkungen dieser ausschließlich im Empfangszweig beobachteten, lastabhängigen Latenzen beim Netzwerk-Konzept sind im Einzelfall in Bezug auf die Anforderungen der jeweiligen CAN Anwendung zu bewerten. Die Zusammenstellung der Messwerte aus [59] ist online verfügbar (`http://rtime.felk.cvut.cz/can/benchmark/1/`, 16.05.2010).

### 5.1.2 CAN Inhaltsfilter

Für den folgenden Vergleich zum Stand der Technik wurde die Aufzeichnung von CAN Daten eines Volkswagen T5 California verwendet, die schon für eine Bewertung der latenten Buslast in Nutzfahrzeugen aufgrund zyklischer Botschaften genutzt wurde (siehe Tabelle 2.1 auf Seite 18).

Wie in Kapitel 2.3.1 (CAN im Kraftfahrzeug) ausführlich beschrieben, werden in Kraftfahrzeugen häufig zyklische CAN Botschaften verwendet, um den Ausfall einer Botschaft erkennbar zu machen. Zur Realisierung dieser Anforderung werden CAN Botschaften auch dann übertragen, wenn sich in den Nutzdaten der CAN Botschaft keine Informationen geändert haben.

Für die folgende Messung wurden zwei gleich lange zeitliche Abschnitte der CAN Aufzeichnung von jeweils 19.2 Sekunden gewählt:

- Fahrzeug im Stand. Zündung aus.
- Fahrzeug in Fahrt.

Die Messwerte wurden mit dem Werkzeug **cansniffer** (siehe Anhang, Kapitel C.1.2) ermittelt, welches jede Änderung in den Nutzdaten visualisieren kann. Die Ablesung der Werte wurde durch das Auslesen der Filterstatistiken des *Broadcast-Managers* (siehe "Inhaltsfilter /proc/net/can-bcm/<socket>“ ab Seite 155) realisiert.

| CAN Bus | Zündung | ungefiltert | gefiltert | Ersparnis | Verworfen [%] |
|---|---|---|---|---|---|
| Komfort-CAN | aus | 3591 | 998 | 2593 | 62,2 % |
| Antriebs-CAN | aus | - | - | - | - |
| Komfort-CAN | an | 4315 | 2211 | 2104 | 48,8 % |
| Antriebs-CAN | an | 18027 | 14631 | 3396 | 18,8 % |

Tabelle 5.2: Performanzmessung: Inhaltsfilterung

Gemäß Tabelle 5.2 ermöglicht die Verwendung eines Inhaltsfilters für zyklische CAN Botschaften für den beschriebenen Anwendungsfall eine Reduzierung des Datenaufkommens um mindestens 18,8 %. Im schlechtesten Fall (Antriebs-CAN während der Fahrt) könnten in dem Zeitraum vom 19,2 Sekunden 3396 Nachrichten an die CAN Anwendung eingespart werden. Das sind gerundet 177 CAN Nachrichten pro Sekunde, die nicht vom Betriebssystemkontext in den Anwendungskontext transferiert werden müssen und die auch nicht von der CAN Anwendung ausgewertet werden müssen.

Der beschriebene Anwendungsfall kann nicht als ein repräsentatives Beispiel einer optimalen Nutzung des *Broadcast-Managers* durch eine CAN Anwendung angenommen werden. Die von der Anwendung tatsächlich benötigten und in den CAN Botschaften kodierten Signale werden nicht explizit herausgefiltert. Zusätzlich könnte die CAN Anwendung die Rate der Aktualiserungen reduzieren, wenn sie beispielsweise nur alle 200ms eine Änderung bestimmter Fahrzeugsignale verarbeiten kann.

Im Rahmen der Studienarbeit [40] sowie der Diplomarbeit [41] von André Naujoks wurde eine eingehende Analyse einer bei der Volkswagen AG entwickelten Software zum Zugriff auf Fahrzeugsignale durchgeführt. Dabei wurde neben Performanzmessungen und der Analyse der Hierachie von Signalabhängigkeiten eine Optimierung der Software beim Zugriff auf den *Broadcast-Manager* durchgeführt (Zitat aus [41] Seite 58ff):

> Das häufige Verwerfen der Daten kann gut an einem Beispiel gezeigt werden. Die Objekte *EngineSpeed* und *VehicleSpeed* werden hier genutzt, um die Reduzierung der CAN-Daten in Tabelle 5.3 darzustellen. Hierzu wurden die Ausgaben des BCM im so genannten proc-Filesystem des Linux-Kernels genutzt, die prozentual und absolut anzeigen, wieviele Datenpakete bereits im Kern des Betriebssystems verworfen wurden.
>
> | | **empfangen** | **verworfen** | **verworfen** [%] |
> |---|---|---|---|
> | CAN-Daten für EngineSpeed | 479500 | 80747 | 17 |
> | CAN-Daten für VehicleSpeed | 248272 | 22752 | 31 |
>
> Tabelle 5.3: Reduzierung des Datenaufkommens durch SocketCAN (Datenaktualisierung auf dem CAN Bus ca. alle 10ms)
>
> Tabelle 5.4 zeigt diese Werte noch einmal allerdings mit einer moderaten Drosselung der Daten auf 200 ms. Eine Drosselung der Daten auf 200 ms ist z. B. bei Datenanzeigen ein guter Erfahrungswert.
>
> | | **empfangen** | **verworfen** | **verworfen** [%] |
> |---|---|---|---|
> | CAN-Daten für EngineSpeed | 479500 | 454670 | 95 |
> | CAN-Daten für VehicleSpeed | 248272 | 225520 | 91 |
>
> Tabelle 5.4: Reduzierung des Datenaufkommens durch SocketCAN bei einer Drosselung auf 200ms
>
> In beiden Tabellen ist deutlich zu sehen, dass die Anwendung deutlich weniger Daten zu verarbeiten hat.

### 5.1.3 Deadline Monitoring

Im vorhergehenden Abschnitt wurde die Möglichkeit zur Reduzierung des Datenaufkommens für CAN Anwendungen durch Inhaltsfilter dargestellt. Mit dem beschriebenen Verfahren entfällt allerdings die Möglichkeit der Ausfallerkennung einer CAN Botschaft durch das so genannte Deadline Monitoring.

Grundsätzlich werden beim Stand der Technik im Kraftfahrzeug die Anforderungen

- Übertragung von Daten
- Ausfallerkennung (Deadline Monitoring)

miteinander verbunden. Dieses auf dem CAN Bus definierte Verfahren muss nicht zwingend an der Programmierschnittstelle zur CAN Anwendung beibehalten werden.

Der in Kapitel B.3 beschriebene *Broadcast-Manager* ermöglicht zusätzlich zur Filterung in den CAN Nutzdaten das Überwachen des Ausbleibens einer CAN Botschaft für eine zuvor definierte Zeit. Das CAN Anwendungsprogramm stellt dazu eine Zeitspanne für eine bestimmte CAN-ID zur Überwachung ein und erhält eine Rückmeldung vom *Broadcast-Manager*, wenn der Ausfall der CAN Botschaft detektiert wurde.

Durch die Verwendung von Zeitgebern im Betriebssystemkontext kann diese Funktionalität effizient umgesetzt werden und die Anwendung muss sich nicht mit der aufwändigen Programmierung von ggf. multiplen Zeitgebern auseinandersetzen.

Ein besonders effizientes Beispiel für die Verwendung des Deadline Monitorings im *Broadcast-Manager* stellt eine Anwendung dar, die beim Ausbleiben von jeglichem CAN Datenverkehr für 180 Sekunden das System abschalten ('herunterfahren') soll.

Pseudocode einer Realisierung nach dem Stand der Technik:

1. Öffne einen RAW-Socket ohne CAN-ID Filter
2. Lese ein CAN Frame mit einem Timeout von 180 Sekunden
   - 2a. Timeout: Fahre das System herunter.
   - 2b. Daten vorhanden: Lese das CAN Frame vom RAW-Socket und gehe zu 2.

Dieses mit dem Betriebssystemaufruf `select(2)` an Punkt 2 mit einer geringen Quelltextkomplexität realisierbare Konzept erzwingt das Auslesen jedes empfangenen CAN Frames. Gemäß dem Beispiel in Tabelle 2.1 auf Seite 18 werden beispielsweise 945 CAN Frames pro Sekunde eingelesen, um sie unmittelbar zu verwerfen, weil lediglich ein Deadline Monitoring durchgeführt werden soll.

Die Realiserung des diskutierten Anwendungsfalls mit dem *Broadcast-Manager* wird dadurch erschwert, dass beim *Broadcast-Manager* eine Ausfallerkennung nur bezogen auf eine einzelne CAN-ID erfolgen kann. Folglich muss dieser (potenziell zyklisch ausgesendete) CAN Identifier zunächst ermittelt werden, bevor das Deadline Monitoring für diesen CAN Identifier eingerichtet werden kann:

1. Öffne einen RAW-Socket ohne CAN-ID Filter (setze `Timeout` = 180 Sekunden)
2. Lese ein CAN Frame mit dem gegebenen `Timeout` von x Sekunden
   2a. Timeout: Fahre das System herunter.
   2b. Daten vorhanden:
      – Lese das CAN Frame vom RAW-Socket (wegen der CAN ID)
      – Schließe den RAW-Socket
3. Öffne einen BCM-Socket und überwache diese CAN-ID (Timeout 175 Sek.)
4. Blockierendes(!) Lesen auf dem BCM Socket bis zum Ausfall der Nachricht
5. Rückmeldung vom BCM Socket: Die überwachte CAN Nachricht ist ausgefallen
   – Schließe den BCM-Socket
   – Öffne einen RAW-Socket ohne CAN-ID Filter (setze `Timeout` = 5 Sekunden)
   – Gehe zu 2.

Im Gegensatz zu der Realisierung nach dem Stand der Technik werden nach diesem Verfahren für die Überwachung des CAN Datenverkehrs im besten Fall (unter Realbedingungen) genau zwei CAN Botschaften von der Anwendung gelesen. Sollte die unter Punkt 2a selektierte CAN Botschaft ausfallen, während der andere CAN Datenverkehr bestehen bleibt, wird automatisch eine neue CAN-ID zur Überwachung gewählt.

Bezogen auf das Anwendungsbeispiel können bei aktivem CAN Datenverkehr in jeder Sekunde 945 Datenübertragungen aus dem Betriebssystemkontext an die CAN Anwendung eingespart werden. Da in diesem Anwendungsfall durch den *Broadcast-Manager* auf eine einzelne CAN-ID geprüft wird (siehe Implementierung SFF-Filter in Kapitel A.2) und beim Empfang dieser einzelnen CAN-ID der Neustart des Zeitgebers im Software-Interrupt im Betriebssystemkontext erfolgt, ist diese gezeigte Realisierung des Deadline Monitorings als eine sehr ressourcen-effiziente Problemlösung zu bewerten.

### 5.1.4 CAN Transportprotokoll ISO 15765-2

Zum Vergleich der im Rahmen dieser Arbeit realisierten Linux-Implementierung des ISO 15765-2 Transportprotokolls [27] mit dem Stand der Technik, wurden verschiedene Software Updates an einem Bedienteil für Klimatisierung durchgeführt. Das verwendete Steuergerät mit der Volkswagen Teilenummer 5K0.907.044 unterstützt die CAN basierte Diagnose nach dem UDS[2] Verfahren und wird beispielsweise im Passat CC eingesetzt.

Abbildung 5.4: UDS-fähiges Climatronic Steuergerät

Im Rahmen der Studienarbeit von Matthias Mierau [37] wurde ein Programm zur Durchführung von Software Updates bei UDS-fähigen Steuergeräten realisiert. Das Programm führt dazu den Ablauf eines so genannten 'Flash-Vorgangs' analog zu einem Volkswagen Diagnosetester VAS 5163[3] durch und setzt dabei auf der beschriebenen Linux-Implementierung des ISO 15765-2 Transportprotokolls auf (siehe Kapitel B.5.1).

Der Diagnosetester VAS 5163 ist eine kommerzielle Diagnosesoftware für das Betriebssystem Windows, die für diesen Vergleich mit einer *aktiven* PCMCIA CAN Hardware (EDICcard2[4]) betrieben wurde. Die EDICcard2 verfügt als *aktive* CAN Hardware über einen Infineon C165 16-Bit Prozessor mit 512kByte RAM, welches für die Firmware der Karte genutzt wird [58]. Die Firmware der EDICcard2 unterstützt die Kommunikationsverfahren verschiedener CAN Transportprotokolle (siehe Kapitel 2.3.2):

> *Für den Betrieb von EDICcard2 stehen verschiedene Softwarepakete mit Betriebssoftware und zusätzliche Fahrzeugprotokolle wie z.B. Diagnostics on CAN (ISO 15765), UDS (ISO 14229), KWP 2000 (ISO 14230), TP 2.0 sowie viele OEM-spezifische Protokolle zur Verfügung. (..)*
>
> *Die Protokollabwicklung im Interface sorgt für robustes Zeitverhalten und erlaubt schnelle Flashprogrammierung der Steuergeräte. (..)* ([58], Seite 1)

Zum Vergleich der Performanz der realisierten Lösung wurden vier Software Updates mit der selben PC Hardware (Dell 830 Laptop) durchgeführt. Zwei Updates mit dem VAS

[2]Unified Diagnostic Services nach ISO 14229-1
[3]http://www.dne-gmbh.de/elektronik/de/gast5163/index.htm
[4]http://www.softing.com/home/de/automotive-electronics/products/hardware/bus-interfaces/edic-card2.php

5163 und der EDICcard2 unter Windows und zwei Updates mit dem Flash-Programm von Matthias Mierau mit unterschiedlichen *passiven* CAN Adaptern unter Linux.

Zum Vergleich der Messwerte wurden die jeweils 2478 Blockstarts für die Übertragung der segmentierten Betriebssoftware für das Klima-Steuergerät bewertet. Der folgende CAN Mitschnitt zeigt einen für den Perfomanzvergleich relevanten Ausschnitt:

```
(..)
(1250590638.721439) <0.001.211> 746 [8] 2B 72 93 FB 05 CA 04 55 <- Data Frame
(1250590638.726236) <0.004.797> 7B0 [8] 02 76 02 AA AA AA AA AA <- Anforderung
(1250590638.727606) <0.001.370> 746 [8] 10 C2 36 03 5F 07 09 66 <- First Frame
(1250590638.730693) <0.003.087> 7B0 [8] 30 0F 00 AA AA AA AA AA <- FlowControl
(1250590638.731933) <0.001.240> 746 [8] 21 08 00 4F 01 42 FF 98 <- Data Frame
(..)
```

Die zeitlichen Abfolgen dieser Sequenz von jeweils fünf CAN Botschaften lassen einen Rückschluss auf die Performanz folgender Zustandsübergänge zu:

**7B0** Die Zeit, die das **Klima-Steuergerät** nach dem Empfang von CAN Botschaften vom Diagnosetester benötigt, um

- einen neuen Datenblock anzufordern
- eine FlowControl Nachricht für eine gestartete Blockübertragung zu senden

**746** Die Zeit, die der **Diagnosetester** benötigt, um einen neuen Datenblock für eine Datenübertragung bereitzustellen und das erste Datenframe daraus zu senden

**746** Die Zeit, die der **Diagnosetester** benötigt, um nach einer empfangenen FlowControl Nachricht das nächste (zweite) Datenframe dieses Datenblocks zu senden

Die Performanz des Klima-Steuergerätes (grüne Werte) ist für den hier angestrebten Vergleich der *Diagnosetester*-Anwendungen nicht relevant.

Eine CAN Botschaft mit 8 Byte Nutzdaten ist im besten Fall 111 Bit lang und benötigt im betrachteten Anwendungsfall (mit 100 kBit/s) auf dem CAN Bus eine minimale Übertragungszeit von 1,11 ms. Diese Übertragungszeit auf dem Medium muss daher von den obigen Messwerten aus dem CAN Mitschnitt abgezogen werden, um eine Aussage über die tatsächliche Anwortzeit des jeweiligen Diagnosetesters zu ermitteln:

`001.370` ms - 1,11 ms = 260µs für die Bereitstellung eines neuen Datenblocks

`001.240` ms - 1,11 ms = 130µs für die Aussendung von Daten nach einem FlowControl

Die Messwerte in den folgenden Tabellen sind in Mikrosekunden [µs] angegeben und um die errechnete Übertragungszeit auf dem Medium (1,11 ms) bereinigt. Die in den

Tabellen angegebenen Messwerte beschreiben jeweils die Zeit, die der betrachtete Diagnosetester benötigt, um für einen neuen Datenblock das erste Datenframe (siehe Tabelle 5.5) bzw. das zweite Datenframe (siehe Tabelle 5.6) zur Verfügung zu stellen.

Anhand der im Folgenden gezeigten Messwerte können die unterschiedlichen Konzepte

- Diagnoseanwendung unter Microsoft Windows mit *aktiver* CAN Hardware
- Diagnoseanwendung unter Linux mit *passiver* CAN Hardware und einer Datenverarbeitung im Software-Interrupt (SoftIRQ)

in ihrer Performanz miteinander verglichen und bewertet werden.

| Messung | Durchschnitt | min | max | Standardabweichung |
|---|---|---|---|---|
| VAS5163 (Messung 1) | 13606 µs | 5 µs | 42755 µs | 7112 µs |
| VAS5163 (Messung 2) | 14960 µs | 10 µs | 55939 µs | 7838 µs |
| Linux EMS PCMCIA | 478 µs | 181 µs | 7643 µs | 270 µs |
| Linux PEAK PCMCIA | 500 µs | 209 µs | 2909 µs | 158 µs |

Tabelle 5.5: ISO-TP: Zeit bis zur Aussendung des ersten Datenframes

Die Zeit zur Bereitstellung des ersten Datenblocks ist primär abhängig von der Implementierung der Diagnoseanwendung. Durch die Verwendung der LibXML2[5]-Bibliothek für die Realiserung unter Linux kann beim Parsen der ODX[6]-Datei mit den Dateninhalten für den Softwaredownload davon ausgegangen werden, das die Flashdaten im Hauptspeicher des Systems vorhanden sind. Die Pufferung der Informationen im Hauptspeicher führt unter Linux zu verhältnismäßig kurzen Antwortzeiten mit einer ebenfalls geringen Standardabweichung.

Die verhältnismäßig stark voneinander abweichenden Antwortzeiten der Realisierung der Diagnoseanwendung unter Windows lässt vermuten, das zur Laufzeit auf den Datenträger zugegriffen wurde und/oder ungünstige Verfahren zum Parsen und Verarbeiten der XML-Strukturen angewendet wurden.

Die minimalen unter Windows gemessenen Antwortzeiten von 5 bzw. 10 Mikrosekunden lassen eine entsprechende Vorhaltung und eine bereits erfolgte Aufbereitung des ersten Datenframes von auszusendenden Daten in der *aktiven* EDICcard2 CAN Hardware vermuten. Die Vermutung des vorgehaltenen First-Frames wird dadurch gestützt, dass die Antwortzeit der betrachteten Windows Lösung beim folgenden (zweiten) Datenframe in Tabelle 5.6 nie kürzer als 23 Mikrosekunden wird.

[5]libxml: The XML C parser and toolkit of Gnome http://xmlsoft.org
[6]**O**pen **D**iagnostic Data **EX**change - XML-Datei gemäß ISO/DIS 22901-1

Die Messwerte aus Tabelle 5.6 zeigen die erwartungsgemäßen Unterschiede in den Antwortzeiten der zwei betrachteten Konzepte mit *aktiver* und *passiver* CAN Hardware auf.

Der eingebettete C165 Prozessor in der *aktiven* CAN Karte sendet die Datenpakete in einem nach oben begrenzten Zeitrahmen von 421 Mikrosekunden aus. Die Performanz des für diesen Anwendungsfall exklusiv genutzten 16-Bit Prozessors führt zu einer durchschnittlichen Antwortzeit von ca. 350 Mikrosekunden.

| Messung | Durchschnitt | min | max | Standardabweichung |
|---|---|---|---|---|
| VAS5163 (Messung 1) | 352 µs | 23 µs | 413 µs | 61 µs |
| VAS5163 (Messung 2) | 350 µs | 28 µs | 421 µs | 61 µs |
| Linux EMS PCMCIA | 166 µs | 52 µs | 4078 µs | 108 µs |
| Linux PEAK PCMCIA | 178 µs | 65 µs | 2552 µs | 84 µs |

Tabelle 5.6: ISO-TP: Zeit bis zur Aussendung des zweiten Datenframes

Die Realisierung des ISO 15765-2 Transportprotokolls kann aufgrund fehlender Echtzeiteigenschaften im Software-Interrupt unter Linux die vorhersagbare Antwortzeit der *aktiven* CAN Hardware erwartungsgemäß nicht erfüllen. Die minimale Antwortzeit ist ca. doppelt so lang wie bei der betrachteten *aktiven* CAN Hardware und im schlechtesten Fall ist die Antwortzeit mit 4 Millisekunden ca. zehn Mal so lang wie bei der *aktiven* EDICcard2 unter Windows. Einzelne Latenzen im Bereich mehrerer Millisekunden sind auch sporadisch bei den Performanzmessungen zum LinCAN Treiber (siehe [59] Abb. 7) aufgetreten und ließen sich dort auf einen fehlerhaften WLAN-Treiber zurückführen.

Obgleich die sporadisch auftretenden Latenzen im Millisekundenbereich quantitativ eine schlechtere Performanz bedeuten, sind sie für den hier zu bewertenden Anwendungsfall in der Praxis tatsächlich nicht relevant, weil erst Zeitüberschreitungen von typischerweise 1000 Millisekunden zu einer Fehlerbehandlung in CAN Transportprotokollen führen.

Durchschnittlich beträgt die Antwortzeit der Linux Lösung mit der *passiven* EMS PCMCIA CAN Hardware 166 Mikrosekunden und ist damit im Durchschnitt doppelt so performant wie die Windows Lösung mit einer *aktiven* CAN Hardware. Diese Messungen werden auch durch die gesamte benötigte Zeit für das Software Update des Klima-Bedienteils bestätigt: Die Linux Lösung ist mit 122,70 Sekunden (EMS) und 123,17 Sekunden (PEAK) ungefähr 2,3 mal schneller als die Windows Lösung (VAS 5163 Diagnosetester) mit 287,55 Sekunden (Messung 1) und 292,10 Sekunden (Messung 2).

Abschließend lässt sich für den Vergleich mit dem Stand der Technik feststellen, dass die Anwortzeiten der ISO 15765-2 Transportprotokoll-Implementierung unter Linux generell für einen produktiven Einsatz geeignet sind. Die Linux Lösung mit einer kostengünstigen, *passiven* CAN Hardware ist - auf der gleichen PC Hardwarebasis - potenziell imstande, die dargestellte Windows Lösung mit *aktiver* CAN Hardware zu ersetzen.

### 5.1.5 Auswirkungen der CAN Treiberperformanz

Im vorherigen Kapitel 5.1.4 wurden die Messungen der Linux Implementierung mit zwei unterschiedlichen, *passiven* PCMCIA CAN Karten duchgeführt:

- PCAN-PC Card dual channel (Fa. PEAK) mit PEAK Treiber (Version v6.11)
- CPC-Card (Fa. EMS Wünsche) mit SocketCAN Treiber (SVN Revision 1038)

Die CAN Hardware unterscheidet sich aus der Sicht des CAN Treiberprogrammierers nur marginal, da beide PCMCIA Karten den verwendeten NXP SJA1000 [46] CAN Controller linear in den Adressraum des PCs einblenden.

Dennoch ist die EMS Karte mit dem SocketCAN Treiber bei der Verwendung der selben Testumgebung im Durchschnitt performanter als die PEAK Karte mit dem PEAK Treiber, wie aus den aus Tabellen 5.5 und 5.6 zu entnehmen ist. Auf eine genaue Quantifizierung des Performanzunterschiedes wurde im Rahmen dieser Arbeit verzichtet, weil sie für die Darstellung des grundsätzlichen Sachverhaltes nicht von Belang ist.

| Messung | Durchschnitt | min | max | Standardabweichung |
|---|---|---|---|---|
| Linux EMS PCMCIA | 166 µs | 52 µs | 4078 µs | 108 µs |
| Linux PEAK PCMCIA | 178 µs | 65 µs | 2552 µs | 84 µs |
| Linux EMS PCMCIA | 478 µs | 181 µs | 7643 µs | 270 µs |
| Linux PEAK PCMCIA | 500 µs | 209 µs | 2909 µs | 158 µs |

Tabelle 5.7: ISO-TP Messungen: Vergleich der Treiberperformanz

Tatsächlich zeigen die Unterschiede zwischen den *passiven* PCMCIA Karten der Fa. PEAK System und der Fa. EMS Wünsche eine Abhängigkeit der Linux Lösung von der Performanz des jeweiligen CAN Treibers. Besonders für den im Kapitel 5.1.4 nicht betrachteten Fall direkt aufeinander folgender Datenpakete ohne zeitliche Verzögerung (ISO-TP Parameter STmin = 0) bestimmt die Performanz des Treibers die maximal mögliche Buslast zur Übertragung von Nutzdaten. In einem solchen Fall können bis zu 585 CAN Frames vom ISO-TP Treiber direkt aufeinanderfolgend in den Sendepuffer des CAN Netzwerkgerätes geschrieben werden (siehe “Identifierbasierte Priorisierung von CAN Sendewarteschlangen” in Kapitel A.9.3), welche vom CAN Treiber danach schnellstmöglich auf den CAN Bus gesendet werden müssen.

Das im Rahmen dieser Arbeit gewählte Konzept bietet demnach nicht nur die Möglichkeit, mit mehreren Anwendungen parallel auf den CAN Bus zugreifen zu können, sondern es ermöglicht in Abhängigkeit vom verwendeten CAN Treiber zusätzlich die vollständige Auslastung des CAN Übertragungsmediums.

## 5.2 Transfer der erarbeiteten Konzepte

Die in dieser Arbeit beschriebene Integration des Controller Area Networks in das Betriebssystem Linux stellt nur *eine* Ausprägung zur Anwendung der erarbeiteten Konzepte dar. Dieses Kapitel zeigt anhand von drei Beispielen, wie die beschriebenen Konzepte und Programmierschnittstellen für Anbindungen des Controller Area Networks in weiteren Mehrbenutzerbetriebssystemen (Microsoft Windows XP und Realtime Linux) sowie auf einem 8-Bit Microcontroller (ohne Betriebssystem) anwendbar sind.

### 5.2.1 Realisierung unter Windows XP

Grundsätzlich läßt sich das beschriebene CAN Subsystem in jedem Mehrbenutzersystem mit einer Socket-Schnittstelle zur Netzwerkkommunikation realisieren. Abbildung 5.5 zeigt eine Umsetzung des Konzeptes mit dem Microsoft Betriebssystem Windows XP, die von der Fachhochschule Braunschweig/Wolfenbüttel im Auftrag der Volkswagen Konzernforschung prototypisch implementiert wurde.

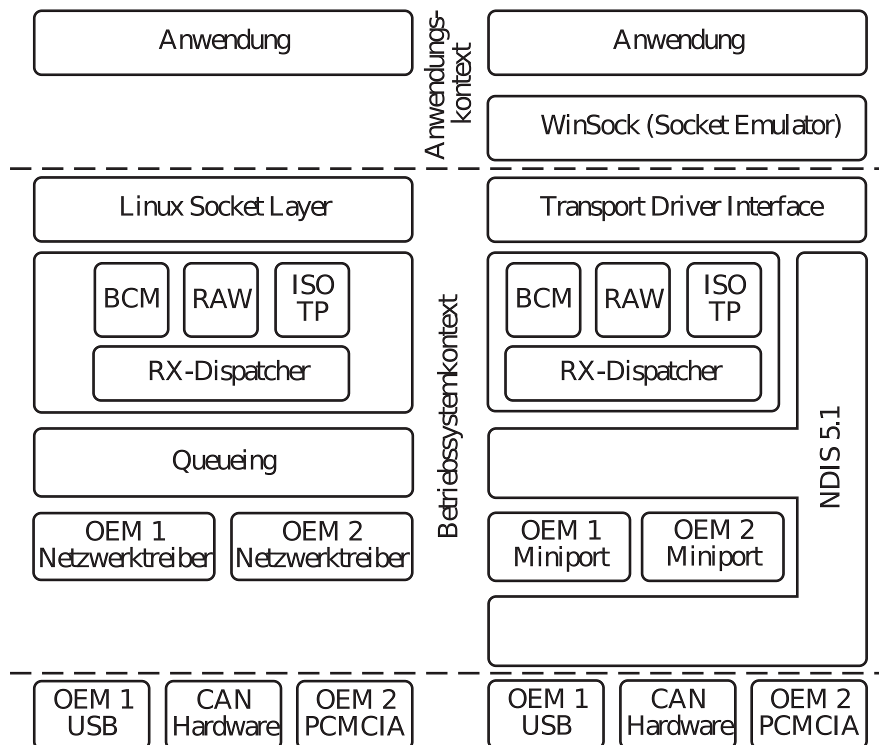



Abbildung 5.5: CAN Netzwerk Konzept unter Windows (rechts) [31]

Wenngleich sich die Struktur des Windows-internen NDIS[7] Treiberkonzeptes von den definierten Treiberschnittstellen in UNIX-ähnlichen Betriebssystemem unterscheidet, konnte für den Anwendungsprogrammierer eine entsprechende Socket-Netzwerkprogrammierschnittstelle für die Protokollfamilie PF_CAN umgesetzt werden.

Zur Realisierung der benötigten CAN Miniport Netzwerktreiber wurden bei der prototypischen Implementierung vorhandene Windows CAN Treiber (siehe Kapitel 3.5) der Firmen PEAK System (PCAN USB) und Kvaser (LAPcan PCMCIA) modifiziert. Dabei sind in beiden Treibern Probleme und Fehler erkannt worden, die durch die nicht offengelegte Schnittstelle (siehe Seite 50) und den Anwendungszugriff über eine herstellerspezifische CAN Bibliothek (siehe Abbildung 3.5) bis dahin verdeckt wurden.

Als Protokolle der Protokollfamilie PF_CAN wurden der RAW-Socket (siehe Kapitel B.2) und der *Broadcast-Manager* Socket (siehe Kapitel B.3) in der Programmiersprache C++ neu implementiert, wobei der Dienstleister nicht die geforderte Portierung der vorhandenen Linux-Quelltexte umsetzte.

Neben des zu erwartenden zusätzlichen Aufwandes für die Wartung und Entwicklung der somit eigenständigen Windows Version führte die Situation bei der Treiberunterstützung zu der Entscheidung, sich im Weiteren auf die quelloffene Linux Realisierung zu fokussieren. Der Quelltext für die unmodifizierten Windows CAN Treiber war nur unter einem Non-Disclosure Agreement (NDA) verfügbar, so dass die Verbreitung eines modifizierten CAN Netzwerktreibers durch Volkswagen aus rechtlichen Gründen ausgeschlossen war.

Mangels Interesse seitens der ursprünglichen Anbieter der CAN Treiber und aufgrund der Einstellung des Projektes durch die Volkswagen Konzernforschung wurde ein Performanzvergleich zum Stand der Technik, wie er in Kapitel 5.1 ausführlich für die Linux Implementierung beschrieben wurde, für die funktionsfähige, prototypische Implementierung des CAN Subsystems unter Windows nicht durchgeführt.

### 5.2.2 RT-SocketCAN

Nach der Festschreibung der Socket-Schnittstelle und der CAN spezifischen Datenstrukturen wurde für das Realtime Linux Projekt Xenomai[8] eine Echtzeitvariante des SocketCAN Projektes realisiert. Aufgrund der Echtzeitanforderungen im Xenomai Projekt wurden bestimmte Konzepte dieser Arbeit bewusst nicht umgesetzt. Unter Anderem:

- Keine Verwendung von dynamischem Speicher
- Statische Strukturen mit fester Anzahl von Geräten und Filtern
- Ein einzelner, fester Empfangspuffer für jedes CAN Gerät

[7] Network Driver Interface Specification (Standard zur Einbindung von Netzwerkkarten)
[8] Xenomai: Real-Time Framework for Linux http://www.xenomai.org

Die RT-SocketCAN Implementierung realisiert ausschließlich einen RAW-Socket und nutzt dabei keine der vorhandenen Netzwerkfunktionalitäten des Linux Kernels. Beim Empfang einer CAN Nachricht wird diese in eine lokale Struktur `rtcan_skb` der Routine zur Bearbeitung der Unterbrechungsanforderung kopiert. Ein Zeiger auf diese Struktur wird an die Empfangsroutine `rtcan_rcv()` übergeben, in der nach einer entsprechenden Filterung die Nutzdaten (das CAN Frame) unmittelbar in den Empfangspuffer des jeweiligen Sockets kopiert wird.

Im Unterschied zu dem in dieser Arbeit beschriebenen CAN Subsystem ist die Anzahl der Filter beliebig aber fest und die Filterlisten sind an die Struktur für das CAN 'Netzwerkgerät' angebunden und nicht in Filterlisten innerhalb der Netzwerkschicht, die es in diesem Konzept nicht gibt.

Weil die Verarbeitung der empfangenen CAN Nachrichten bis zum Kopieren in den Empfangspuffer des Sockets innerhalb des Unterbrechungskontextes erfolgt, sind die Werte zur Erzeugung der statischen Datenstrukturen

- Anzahl der CAN Geräte: XENO_DRIVERS_CAN_MAX_DEVICES (4*)
- Anzahl der Filter pro CAN Gerät: XENO_DRIVERS_CAN_MAX_RECEIVERS (16*)
- Größe des Socket Empfangspuffers: XENO_DRIVERS_CAN_RXBUF_SIZE (1024*)

zur Kompilierungszeit festzulegen. Die Standardwerte sind mit '*' gekennzeichnet.

Zusätzlich kann über die Option XENO_DRIVERS_CAN_LOOPBACK festgelegt werden, ob die in Kapitel 4.3.6 (Lokales Echo gesendeter CAN Nachrichten) beschriebene Funktionalität vom System vorgehalten werden soll.

Trotz der Einschränkugen auf statische Datenstrukturen ist die Programmierschnittstelle des CAN Subsystems für den Anwendungsprogrammierer bei RT-SocketCAN weitestgehend erhalten geblieben. Die Schnittstellen zur CAN Anwendung und zum CAN Treiber sind für eine Migration und für die Wiederverwendbarkeit der vorhandenen Software von entscheidender Bedeutung.

Die Umsetzung der Funktionalitäten der Netzwerkschicht mit Hilfe von statischen Datenstrukturen sind für den Anwender zunächst nicht relevant. In Abhängigkeit vom Anwendungsfall können die vorhandenen Einschränkungen über die Konfiguration zur Kompilierungszeit ausgeräumt werden.

Die Übertragung der Programmierschnittstellen in RT-SocketCAN stellen bei gleichzeitiger Umsetzung mit statischen Datenstrukturen einen interessanten Ansatz zur Implementierung des CAN Subsystems in eingebetteten Systemem ohne dynamische Speicherverwaltung dar. Als Komponente im Realtime Linux Projekt Xenomai wird mit RT-SocketCAN ein vorhersagbares Verhalten für den Zugrif auf den CAN Bus garantiert.

### 5.2.3 Der Broadcast Manager auf einem 8-Bit Microcontroller

Zur Visualisierung von Fahrzeugdaten auf einem Personal Digital Assistant (PDA) stand als Erweiterungsschnittstelle des PDA lediglich eine serielle Schnittstelle nach dem EIA[9] RS-232 Standard zur Verfügung. Die vorhandenen, kommerziellen CAN ↔ RS-232 Adapter, bspw. CAN232 der Firma LAWICEL (`www.can232.com`) mit dem in [13] beschriebenen ASCII Kommunikationsprotokoll, verfügen nur über 'einfache' CAN-ID/Masken Filterfunktionen für CAN Botschaften. Dieser 'einfache' Filter ist bei realen Anwendungsfällen im Fahrzeug gemäß Tabelle 4.1 (auf Seite 83) nicht sinnvoll nutzbar, wodurch der PDA mit der Verarbeitung des vollständigen CAN Datenverkehrs belastet wurde.

Wie in Kapitel 5.1.2 (CAN Inhaltsfilter) aufgeführt, bietet der *Broadcast-Manager* besonders für die Verwendung im Fahrzeugumfeld erhebliche Vorteile, so dass für die beschrieben Anforderungen des Personal Digital Assistant eine Lösung für die Anbindung des *Broadcast-Managers* über eine serielle Schnittstelle erarbeitet werden sollte.

Die Realisierung des *Broadcast-Managers* außerhalb des Linux Betriebssystems erforderte eine Neuimplementierung verschiedener Softwarekomponeten und Schnittstellen, die für die Abbildung der Funktionalität des *Broadcast-Managers* benötigt werden:

- Schnittstelle zur Anwendung (bisher PF_CAN Netzwerksockets)
- Schnittstelle zu CAN ID Filtern (Umfang PF_CAN Protokollmodul)
- Schnittstelle zum Senden von CAN Frames (Umfang PF_CAN Protokollmodul)
- Zeitgeber, beispielsweise zum Deadline Monitoring (bisher Linux Kernel Timer)

In Abbildung 5.6 wird die Einbettung des *Broadcast-Managers* in die Softwarestruktur des 8-Bit Prozessors aus [57] gezeigt.

Für die Anbindung der CAN Anwendung über die serielle Schnittstelle wurde ein ASCII-basiertes Datenprotokoll entworfen und implementiert. Dieses Datenprotokoll erlaubt das sichere Parsen des ASCII Datenstroms über definierte Start- und Endzeichten '<' und '>', sowie einer so genannten **Applikations-ID**, die sich dem Startzeichen anschließt:

**B** Nachricht von und zum Broadcast Manager

**R** Software-Reset

**V** Senden von CAN Nachrichten über den virtuellen CAN (VCAN)

[9] Electronic Industries Alliance - `http://www.eia.org`

Das Versenden von CAN Nachrichten über den virtuellen CAN wurde zu Testzwecken realisiert, um die Funktionalität des realisierten *Broadcast-Managers* ohne CAN Bus testen zu können.

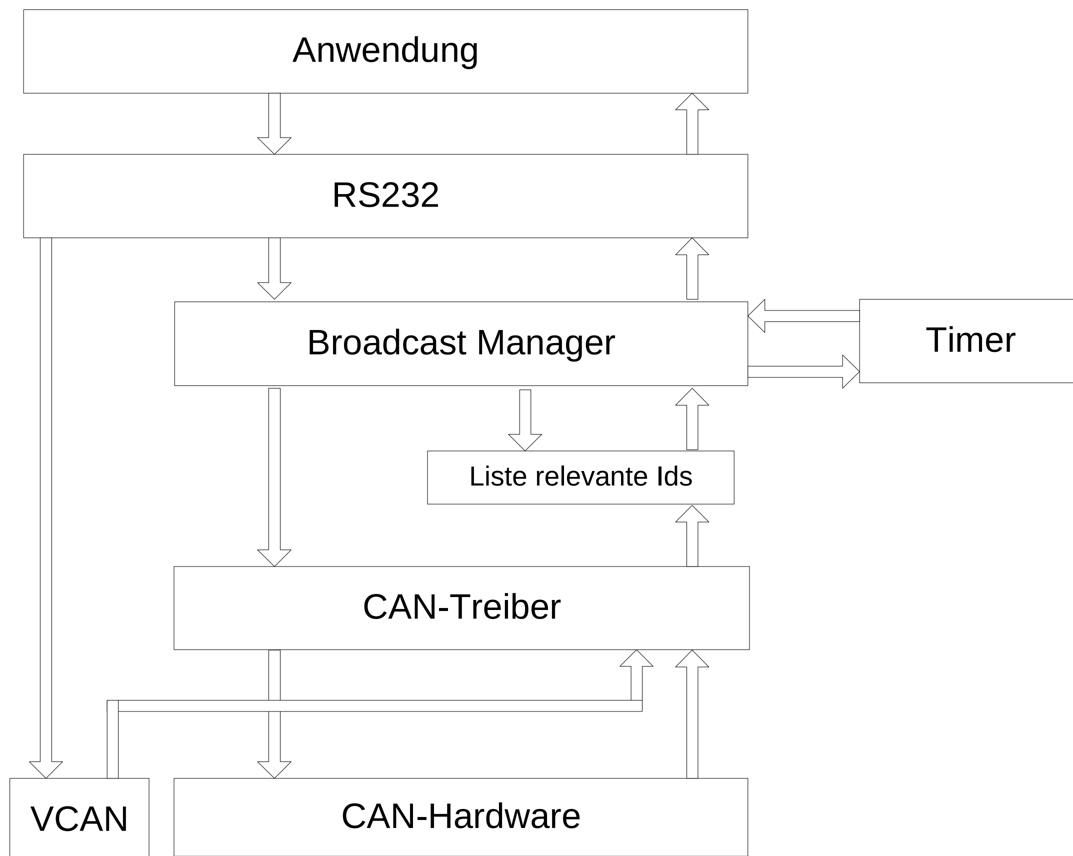



Abbildung 5.6: Die Softwarestruktur auf dem 8-Bit ATmega161 [57]

Die in Linux zur Laufzeit allozierten dynamischen Speicherstrukturen wurden analog der RT-SocketCAN Realisierung (siehe Kapitel 5.2.2) als statische Datenstrukturen ausgeführt, wodurch die maximale Anzahl der Timer beispielsweise auf einen festen Wert begrenzt ist.

Die Befehle an den *Broadcast-Manager* wurden in der Form von bekannten Datenstrukturen übertragen, wie sie für den *Broadcast-Manager* (siehe Kapitel B.3) definiert wurden. Zum Zeitpunkt der Realisierung enthielt die Struktur für das CAN Frame noch ein Strukturelement '**Flag**' für das RTR-Bit, welches in späteren Versionen in die Sruktur für den CAN Identifier überführt wurde.

```
<B01.0300.0500.E803.D007.4400.01.4400.01.08.4E.55.52.4D.45.49.4E.45>

Das bedeutet es sind folgende Elemente in der BCM-Botschaft enthalten:

ApplikationsID:   B         (Nachricht vom und zum BCM)
Opcode:           0x01      (TX_SETUP)
Flag:             0x0003    (SETTIMER| STARTTIMER)
Count:            0x0005    (5mal Intervall 1)
Ival1:            0x03E8    (1000 Millisekunden)
Ival2:            0x07D0    (2000 Millisekunden)
ID:               0x0044
nframes:          0x01
CAN Frame ID:     0x0044
Flag:             0x01
DLC:              0x08
Nutzdaten:        0x4E55524D45494E45
```

Die Anzahl der statischen Datenstrukturen des implementierten *Broadcast-Managers* für das Senden (tx_ops) und Empfangen (rx_ops) wurde im beschriebenen Fall der Umsetzung auf den ATmega161 auf 15 rx_ops und Null tx_ops festgelegt. Die mit dem 8-Bit Microcontroller realisierte Lösung konnte durch die inhaltsbasierte Nachrichtenfilterung die CAN Anwendung auf dem PDA erwartungsgemäß entlasten.

### 5.2.4 Zusammenfassung

Die Realisierung der Funktionalität des *Broadcast-Managers* auf einem 8-Bit Mikrocontroller ist neben der Umsetzung von "RT-SocketCAN“ ab Seite 112 ein Beispiel für die Nutzung der erarbeiteten Konzepte in Systemen ohne die Verwendung einer dynamischen Speicherverwaltung.

Bei der Adaption des neuen Standes der Technik zum Zugriff auf das Controller Area Network an RT-SocketCAN und bei der Realisierung unter Windows XP wurden überwiegend die Abtraktionsschichten und die daraus resultierenden Programmierschnittstellen zum CAN Treiber sowie zum CAN Anwendungsprogrammierer wiederverwendet.

Fazit:

Die im Rahmen dieser Arbeit definierten Abstraktionsschichten zur Verwendung des Controller Area Networks können weitgehend unabhängig von der jeweilig vorhandenen Betriebssystemumgebung angewendet werden. Die Erfahrungen aus den dargestellten Konzepttranfers bieten sich generell zur Entwicklung von CAN Programmierschnittstellen an.

# 6 Zusammenfassung und Ausblick

Im Abschnitt 6.1 wird die vorliegende Arbeit und ihr Beitrag zum Stand der Technik zusammengefasst. Im Rahmen des Abschnitts 6.2 wird auf aktuelle und zukünftige Einsatzmöglichkeiten des erarbeiteten Konzeptes im Fahrzeugumfeld eingegangen und auf mögliche Weiterentwicklungen des CAN Subsystems verwiesen.

## 6.1 Zusammenfassung der Arbeit

Das Controller Area Network ist als Kommunikationsnetz in eingebetteten Systemen ein fester Bestandteil von Anwendungen in Kraftfahrzeugen und im industriellen Umfeld. In mehr als 20 Jahren seit der Verfügbarkeit des CAN Busses haben sich unzählige Hardware- und Software-Lösungen zum Zugriff auf den CAN etabliert. Vom eingebetteten 8-Bit Mikrocontroller bis zum 64-Bit Personalcomputer wird auf dem CAN Bus mit unterschiedlichster CAN Hardware kommuniziert.

Für jede dieser Hardwarelösungen wird eine spezielle CAN Treibersoftware benötigt, welche über ihre spezifische Programmierschnittstelle eine Abhängigkeit der CAN Anwendung zur der (hersteller-) spezifischen Hardware induziert. Als Folge wird eine CAN Anwendung zumeist für die CAN Hardware eines bestimmten Anbieters erstellt und es entsteht zwangsläufig ein Bündel bestehend aus einer CAN Hardware und einer CAN Software. Diese Bündelung erschwert beispielsweise den Wechsel auf die CAN Hardware eines anderen Anbieters, was im wirtschaftswissenschaftlichen Umfeld als 'Vendor-Lock-In' bezeichnet wird: Ein Austausch der Hardware führt zwangsweise zu einer Anpassung der Anwendungssoftware, was mit einem zusätzlichen Risiko und mit zusätzlichen Kosten verbunden ist, welches wiederum die Produktauswahl beeinflusst.

Die Auflösung des Vendor-Lock-In und die Wiederverwendung von CAN Anwendungen unabhängig von der konkreten CAN Hardware durch eine entsprechende Abstraktion ist die Problemstellung und gleichzeitige Zielsetzung dieser Dissertation.

Die Erwartung, durch den Einsatz eines etablierten Mehrbenutzerbetriebssystems analog zu den bekannten Programmierschnittstellen für Massenspeicher, Netzwerkkarten oder Eingabegeräten auch eine hardwareabstrahierende Programmierschnittstelle für das Controller Area Network zu erhalten, wird aufgrund der oben beschriebenen Randbedingungen nicht erfüllt. Nach dem bisherigen Stand der Technik existiert keine mehrbenutzerfähige Programmierschnittstelle, welche die unterschiedlichen Anwendungsszenarien bei der Verwendung des Controller Area Networks abbilden kann.

Die vorliegende Dissertation untersucht anhand von exemplarischen Anwendungsszenarien und den gegebenen technischen Randbedingungen des Controller Area Networks mögliche Alternativen zur Integration einer CAN Programmierschnittstelle in Mehrbenutzerbetriebssysteme. Dazu wird nach der Bewertung des Standes der Technik, der sich für die ermittelten Anforderungen als nicht geeignet erweist, versuchsweise die Technologie der Linux Netzwerkprogrammierung auf das Controller Area Network abgebildet.

Als Beitrag zum Stand der Technik wird im Rahmen dieser Arbeit eine neue Netzwerkprotokollfamilie für das Controller Area Network (PF_CAN) definiert, welche die Anforderungen an eine CAN Programmierschnittstelle in Mehrbenutzerbetriebssystemen erfüllt und das beschriebene Problem mit dem Vendor-Lock-In für CAN Hardware löst.

Die entworfene Programmierschnittstelle orientiert sich am Stand der Technik für Netzwerkprogrammierschnittstellen in Mehrbenutzerbetriebssystemen und ermöglicht den simultanen Zugriff auf den CAN Bus durch mehrere Anwendungen und über verschiedene CAN basierte Kommunikationsprotokolle, wie beispielsweise dem ISO Transportprotokoll 15765-2 [27]. Die erarbeiteten, allgemeingültigen Konzepte für die Integration von eingebetteten Kommunikationsnetzwerken in Mehrbenutzerbetriebssysteme sind auf weitere Betriebssysteme übertragbar, was in Kapitel 5.2 unter anderem am Beispiel von Microsoft Windows XP nachgewiesen wurde.

Aus technischer Sicht werden mit dem dargestellten Konzept die in Kapitel 4.1 zusammengeführten Anwendungsfälle und Anforderungen erfüllt. Die Integration in ein Betriebssystem ohne Echtzeiteigenschaften setzt den damit realisierbaren Anwendungsfällen allerdings grundsätzlich technische Grenzen, bei denen es auf eine zugesicherte Rechenzeit und Antwortzeit ankommt (siehe Kapitel 5.1 und [59]). Können solche Anforderungen nicht wie dargestellt durch Konzepte im Betriebssystemkontext erfüllt werden, sind weiterhin entsprechende Echtzeiterweiterungen der betrachteten Mehrbenutzerbetriebssysteme, ein Echtzeitbetriebssystem oder eine dedizierte, eingebettete Hardwarelösung in Betracht zu ziehen.

Die im Rahmen dieser Arbeit konzipierte und implementierte Protokollfamilie PF_CAN wurde im Jahr 2006 durch die Konzernforschung der Volkswagen AG veröffentlicht (Projektname *SocketCAN*) und ist seit dem 16. April 2008 ein integraler Bestandteil des Netzwerkprotokollstapels des OpenSource Betriebssystems Linux. Die Integration des erarbeiteten CAN Subsystems in das Linux Betriebssystem führte zu einer weltweiten

Beachtung der Protokollfamilie PF_CAN durch CAN Anwendungsentwickler und CAN Treiberentwickler.

Durch die Aufnahme in Linux erfolgte eine defacto Standardisierung der CAN Programmierschnittstellen unter Linux. Analog zu den bereits etablierten Netzwerktechnologien ermöglichen die erarbeiteten CAN Anwendungs- und Netzwerktreiberschnittstellen die Wiederverwendung von Linux CAN Anwendungen auf allen von Linux unterstützten Prozessorarchitekturen und mit unterschiedlicher CAN Hardware.

Mit der Definition der neuen (Standard-)Betriebssystemschnittstellen ergibt sich für die Anbieter von CAN Lösungen im Linux Umfeld die Möglichkeit, ihre Treiber- und Anwendungsimplementationen an die neuen Standardschnittstellen zu adaptieren.

Die CAN Anwendungsschnittstelle wird beispielsweise unterstützt von

- OpenSource Anwendungsentwicklern (bspw. der OpenSource Implementierung des CANopen [5] Protokolls CANFestival, siehe `http://www.canfestival.org`)
- Kommerziellen Anwendungsentwicklern von CANopen [5] Protokollstapeln (bspw. CANopenRT Protokollstapel der Fa. IXXAT GmbH, CANopen Magic Pro Library der Fa. Embedded Systems Academy Inc., UV-Software)

Die CAN Netzwerktreiberschnittstelle wird beispielsweise unterstützt von

- OpenSource Programmierern zur Unterstützung vorhandener CAN Hardware
- CAN Hardwareanbietern für ihre Linux-basierten Produkte (bspw. Firmen Connect Tech Inc., EMS Wünsche e.K., ESD Electronic System Design GmbH, IXXAT GmbH, Kvaser AB, PEAK System GmbH).
- Anbietern für diskrete und integrierte CAN Controller (bspw. Firmen Atmel Corp., Robert Bosch GmbH, Texas Instruments Inc. - Ein Entwickler von TI hat zum Linux Betriebssystem einen HECC (High End CAN Controller) Treiber beigetragen `http://lxr.linux.no/#linux+v2.6.33/drivers/net/can/ti_hecc.c`)

Die defacto Standardisierung der CAN Programmierschnittstellen unter Linux und die Vielzahl im Quelltext frei verfügbarer Beispiele ermöglichen sowohl für den CAN Anwendungsprogrammierer als auch für den CAN Treiberprogrammierer eine erhebliche Reduzierung des Entwicklungsaufwandes [9]. Die Anwendung von etablierten Verfahren und Schnittstellenkonzepten aus dem Umfeld der Informationstechnologie ermöglichen den direkten Wissenstransfer aus der Verwendung anderer Netzwerktechnologien und reduzieren damit den Aufwand für die Einarbeitung in die Programmierung des Controller Area Networks.

## 6.2 Ausblick

Die Realisierung der CAN Programmier- und Treiberschnittstellen analog zu existierenden Netzwerkschnittstellen ermöglichen es dem Anwendungsprogrammierer, Tester und Software-Integrator bei dem Umgang mit dem Controller Area Network vorhandene und bekannte Konzepte und Verfahren anzuwenden. Ein besonderer Vorteil ergibt sich aufgrund der identischen Programmierschnittstellen beispielsweise bei der Integration von drahtlos vernetzten Informationstechnologien in CAN vernetzte Kraftfahrzeuge.

Sowohl beim Aufbau prototypischer Fahrzeugkonzepte auf der Basis von PC-Hardware (Visualisierungen, Fahrerassistenzsysteme, Diagnose, Fehlersuche) als auch bei der Erprobung der Fahrzeug-zu-Fahrzeug-Kommunikation (Car-to-Car) mit Laptops und spezialisierter, Linux-basierter Kleinserien-Hardware werden die Stärken des in dieser Arbeit dargestellten Konzeptes sichtbar. Das beschriebene CAN Subsystem ist seit 2005 in verschiedenen Car-to-Car-Forschungsprojekten (`www.simtd.de`, `www.its.dot.gov/vii`, `www.network-on-wheels.de`) erfolgreich im Einsatz und stellt zukünftig eine integrale Komponente für die Migration der Fahrzeugvernetzung hin zu einer sicheren, IP-basierten Kommunikation (`www.eenova.de/projekte/seis`) dar. Eine sichere, IP-basierte Kommunikation ist dabei auch auf Basis des Controller Area Networks darstellbar (siehe “isotptun (Internet Protocol over CAN)” in Kapitel C.3.5).

Aus wissenschaftlicher Sicht hat sich bei dieser Arbeit das Thema “Identifierbasierte Priorisierung von CAN Sendewarteschlangen” (siehe Kapitel A.9.3) als eine essentielle Funktionalität zur Realisierung bestimmter Anwendungsszenarien beim Zugriff auf das Controller Area Network im neu realisierten Mehrbenutzerkonzept herausgestellt. Die neu entstandenen Möglichkeiten, aufgrund der Socket-basierten Netzwerkfunktionalitäten mit unterschiedlichen CAN Anwendungen parallel auf den CAN Bus schreiben zu können, macht eine anwendungsbezogene Ressourcenvergabe des Kommunikationsmediums erforderlich.

Zusätzlich sind zukünfigte Entwickungen, der in [59] verwendeten Echtzeiterweiterungen des Linux Betriebssystems (RT_PREEMPT) daraufhin zu bewerten, ob sich das erarbeitete Netzwerk-Konzept durch zugesichterte Rechen- und Antwortzeiten für den Einsatz in weiteren CAN Anwendungsszenarien wie beispielsweise einer zeitsteuerten CAN Kommunikation (siehe Kapitel 2.3.3.3) eignet.

Im Anhang werden in Kapitel A.9 Ideen für zukünftige Entwicklungen des CAN Subsystems aufgezeigt. Dabei werden primär Fragestellungen aufgegriffen, welche in der Diskussion mit CAN Anwendern oder im Rahmen dieser Arbeit identifiziert wurden.

# Abkürzungsverzeichnis / Glossar

API: Application Programmers Interface (Programmierschnittstelle für Anwendungsentwickler)

Arbitrierung: Zugriffsverfahren auf das Übertragungsmedium ⇒ 11

Broadcast: Aussendung an Alle. (Jeder CAN Knoten empfängt jede von einem anderen CAN Knoten ausgesendete CAN Botschaft)

Bundle: dt. Bündel - bezeichnet ein mehrteiliges Produkt, welches nur als Komplettpaket erworben werden kann

Callback-Funktion: Rückruffunktion eines Steuerprogramms, welche unter bestimmten Bedingungen (beispielsweise beim Eintreffen von zu verarbeitenden Daten) aufgerufen wird

CAN (Bus): Controller Area Network (Feldbussystem) ⇒ 9

CAN Controller: Integrierter Schaltkreis zur Verarbeitung des CAN Protokolls ⇒ 42

CAN Frame: PDU zum Informationsaustausch auf dem CAN Bus ⇒ 13 (auch *CAN Botschaft* oder *CAN Nachricht*)

CAN Hardware: *aktiv* : CAN Zugriff über einen eingebetteten Prozessor ⇒ 42
*passiv* : CAN Zugriff über einen CAN Controller Baustein ⇒ 42

CAN ID: siehe CAN Identifier

CAN Identifier: eindeutige (siehe Arbitrierung) Adresse einer CAN Nachricht ⇒ 13

CAN Knoten: Teilnehmer an der CAN Kommunikation auf einem CAN Bus

CSMA/CR: Carrier Sense Multiple Access / Collision Resolution ⇒ 11

dominantes Bit: Binärziffer, die auf einem Kommunikationsmedium ein rezessives Bit überschreiben kann ⇒ 11

EFF: Extended (CAN) Frame Format mit 29 Bit CAN Identifier ⇒ 14

eingebettet: (Computer)System, das in einen technischen Kontext *eingebettet* ist und daher u.U. nicht als Computer(System) erkennbar ist

FIFO: First-In-First-Out - abstrakte Datenstruktur mit dem Verhalten, dass das zuerst eingefügte Datenelement auch das erste Element ist, welches entnommen werden kann

Firmware: Betriebssoftware für eingebettete Systeme

Flashen: Übertragen einer (aktualisierten) Betriebssoftware für CAN Steuergeräte über den CAN Bus mittels eines CAN Transportprotokolls

GNU: **G**NU is **n**ot **U**nix

Halb-Duplex: Wechselseitiges Senden und Empfangen von Informationen zwischen zwei Kommunikationsteilnehmern

Hash-Funktion: Streuwertfunktion zur surjektiven Abbildung einer (üblicherweise) großen Definitionsmenge auf eine kleine Zielmenge

I$^2$C: Inter-Integrated Circuit Protokoll ⇒ 2
(Kommunikationsprotokoll zur Vernetzung integrierter Schaltkreise)

IP: Internet Protokoll

ISO: International Organization for Standardization

Linux®: registrierte Handelsmarke von Linus Torvalds

Mehrbenutzerbetriebssystem: Verschiedene Benutzer können in voneinander abgegrenzten Arbeitsumgebungen parallel auf einem System arbeiten

Multitasking: Mehrere Aufgaben werden parallel ausgeführt

OSI: Open Systems Interconnection Modell, Schichtenmodell für Kommunikationsprozesse nach dem ISO/IEC 7498-1:1994 Standard

OSI-Layer: Schicht des OSI Schichtenmodells für Kommunikationsprozesse:

Schicht 7 – Anwendungsschicht (Application Layer)
Schicht 6 – Darstellungsschicht (Presentation Layer)
Schicht 5 – Kommunikationssteuerungsschicht (Session Layer)
Schicht 4 – Transportschicht (Transport Layer)
Schicht 3 – Vermittlungsschicht (Network Layer)
Schicht 2 – Sicherungsschicht (Data Link Layer)
Schicht 1 – Physikalische Schicht (Physical Layer)

OSI-Layer 2: Sicherungsschicht des OSI Schichtenmodells. Sie enthält:

Schicht 2b – Datensicherungsschicht auf der Verbindungsebene (Logical Link Control)

Schicht 2a – Medienzugriffskontrollschicht (Media Access Control)

PCI: Protocol Control Information

PDU: Protocol Data Unit

QNX®: registrierte Handelsmarke von Harman International Industries Incorporated

RAW: (engl. 'roh') bezeichnet den (direkten) Zugriff auf Netzwerk Datenpakete, ohne die Auswertung durch Netzwerk (Transport-)Protokolle

rezessives Bit: Binärziffer, die auf einem Kommunikationsmedium von einem dominanten Bit überschrieben werden kann ⇒ 11

Scheduling: Zeitablaufplanung und -steuerung zur Ressourcenvergabe (Prozessorzeit, Zugriff auf ein Kommunikationsmedium, etc.)

SFF: Standard (CAN) Frame Format mit 11 Bit CAN Identifier ⇒ 14

SMBus: System Management Bus (siehe $I^2C$)

TCP: Transmission Controll Protokoll

TDMA: Time Division Multiple Access (Zeitmultiplex Zugriffsverfahren)

Transportprotokoll: Protokoll zur Übertragung segmentierter Daten über ein Übertragungsmedium ⇒ 21

TTCAN: Time Triggered CAN ⇒ 31

UDP: User Datagram Protokoll

VAPI: Vehicle API (abstrakte Fahrzeugschnittstelle der Volkswagen AG)

Vendor-Lock-In: Strategie zur Kundenbindung ⇒ 54

Voll-Duplex: Gleichzeitiges Senden und Empfangen von Informationen zwischen zwei Kommunikationsteilnehmern

Windows®: registrierte Handelsmarke der Firma Microsoft Corporation

# Literaturverzeichnis


[1] Atmel Corporation: *Atmel AT90CAN128 Datasheet*. Rev. 4250H–CAN–05/06, Mai 2006. – URL http://www.atmel.com/dyn/resources/prod_documents/doc4250.pdf. – Zugriffsdatum: 31.01.2010

[2] Berkaev, D ; Cheblakov, P ; Koop, I ; Ostanin, I ; Kozak, V: Control system for injection channels of VEPP-2000 collider Budker Institute of Nuclear Physics (Veranst.), URL www.inp.nsk.su/~kozak/papers/webph14.pdf. – Zugriffsdatum: 16.05.2010, November 2008

[3] Bloch, Joshua: How to Design a Good API and Why it Matters. In: *Library-Centric Software Design LCSD'05*, URL http://lcsd05.cs.tamu.edu/slides/keynote.pdf. – Zugriffsdatum: 31.01.2010, Oktober 2005

[4] Cach, Petr ; Fiedler, Petr: Internet Draft: IP over CAN Brno University of Technology (Veranst.), URL http://ftp.ist.utl.pt/pub/drafts/draft-cafi-can-ip-00.txt. – Zugriffsdatum: 31.01.2010, September 2001

[5] CAN in Automation GmbH: *CANopen*. – URL http://www.can-cia.org/index.php?id=171. – Zugriffsdatum: 31.01.2010

[6] CAN in Automation GmbH: *The CAN history*. – URL http://www.can-cia.de/index.php?id=161. – Zugriffsdatum: 31.01.2010

[7] Ditze, Michael ; Bernhardi-Grisson, Reinhard ; Kämper, G. ; Altenbernd, Peter: Porting the Internet Protocol to the Controller Area Network. In: *2nd Intl Workshop on real-time LANs in the Internet age (RTLIA 2003)*. Porto, Portugal, 1 Juli 2003. – URL www.hurray.isep.ipp.pt/activities/rtlia2003/full_papers/8_rtlia.pdf. – Zugriffsdatum: 31.01.2010

[8] FMS Standard (Flotten-Management-Schnittstelle). – URL http://www.fms-standard.com. – Zugriffsdatum: 31.01.2010

[9] Fu, Luotao: SocketCAN in automation: A case study. In: *CAN Newsletter* (2010), März, S. 16–18

[10] Förster, Frank: Eine Frage der Treiber-Software. In: *IEE, Automatisierung und Datentechnik* (2004), Januar. – URL http://all-electronics.de/ai/resources/9b4522e54d1.pdf. – Zugriffsdatum: 19.02.2006

[11] Hartkopp, Oliver: Fahrzeuganbindungen durch Standard-IT-Verfahren. In: *Gesellschaft für Informatik, 5. Workshop 'Automotive Software Engineering', Bremen*, 27 September 2007

[12] Hartkopp, Oliver: Access of CAN networks over standard IT interfaces. In: *CAN Newsletter* (2008), September, S. 62–64. – Übersetzung des eingereichten Papers zur 3. Euroforum Konferenz, März 2005, durch die CAN in Automation GmbH

[13] Hartkopp, Oliver: *SLCAN TTY Line Discipline - API Comparison of existing serial CAN Protocol Adapters*, 23 Januar 2008. – URL http://prdownload.berlios.de/socketcan/SLCAN-API.pdf. – Zugriffsdatum: 31.01.2010

[14] Hartkopp, Oliver ; Kaiser, Jörg: Existierende Standard IT Konzepte zur abgesicherten Kommunikation im Fahrzeug. In: *25. VDI/VW-Gemeinschaftstagung „Automotive Security", 19. und 20. Oktober 2009, Ingolstadt*, 19 Oktober 2009

[15] Hartkopp, Oliver ; Miller, Manfred: Integrierte Hard- und Software-Entwicklungsplattform für die Car-to-Car- und Car-to-X-Kommunikation. In: *3. Braunschweiger Symposium – Informationssysteme für mobile Anwendungen, IMA2006*, URL http://www.gzvb.de/fileadmin/user_upload/ima2006_miller_abstract.pdf. – Zugriffsdatum: 31.01.2010, 25 Oktober 2006

[16] Hartkopp, Oliver ; Thürmann, Urs: *Schnittstelle zur Kommunikation zwischen Fahrzeug-Applikationen und Fahrzeug-Bussystemen*. Patentanmeldung. September 2005. – DE102004020880A1 26.04.2004

[17] Hartkopp, Oliver ; Thürmann, Urs: Zugriff auf den CAN-Bus über Standard-IT-Schnittstellen. In: *EUROFORUM Konferenz 'Datenkommunikation im Automobil', 16. und 17. März 2005, Heidelberg*, März 2005

[18] Hartkopp, Oliver ; Thürmann, Urs: *Low Level CAN Framework - Application Programmers Interface*, 20 Februar 2006. – URL http://www.llcf.de/llcf-api-2006-02-20.pdf. – Zugriffsdatum: 31.01.2010

[19] Hartkopp, Oliver ; Thürmann, Urs ; Grandegger, Wolfgang: *Linux Kernel Documentation linux/Documentation/networking/can.txt*, 17 November 2007. – URL http://lxr.linux.no/linux+*/Documentation/networking/can.txt. – Zugriffsdatum: 26.02.2010

[20] Hartwich, Florian ; Müller, Bernd ; Führer, Thomas ; Hugel, Robert: Timing in the TTCAN Network. (2002). – URL www.can-cia.de/fileadmin/cia/files/icc/8/hartwich.pdf. – Zugriffsdatum: 31.07.2010

[21] Intel Corporation: *i82527 Serial Communications Controller (ARCHITECTURAL OVERVIEW)*, Januar 1996. – URL http://user.cs.tu-berlin.de/~mwerner/lect/old/ss06/pek/i82527.pdf. – Zugriffsdatum: 31.01.2010

[22] Intel Corporation: *Interrupt Moderation Using Intel® GbE Controllers*, April 2007. – URL http://download.intel.com/design/network/applnots/ap450.pdf. – Zugriffsdatum: 31.01.2010

[23] ISO 11898-1:2003: *Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling.* ISO, Geneva, Switzerland

[24] ISO 11898-2:2003: *Road vehicles – Controller area network (CAN) – Part 2: High-speed medium access unit.* ISO, Geneva, Switzerland

[25] ISO 11898-3:2004: *Road vehicles – Controller area network (CAN) – Part 3: Low-speed, fault-tolerant, medium-dependent interface.* ISO, Geneva, Switzerland

[26] ISO 11898-4:2004: *Road vehicles – Controller area network (CAN) – Part 4: Time-triggered communication.* ISO, Geneva, Switzerland

[27] ISO 15765-2:2004: *Road vehicles – Diagnostics on Controller Area Networks (CAN) – Part 2: Network layer services.* ISO, Geneva, Switzerland

[28] Johansson/Törngren/Nielsen: *Vehicle Applications of Controller Area Network.* Department of Signals, Sensors and Systems, Royal Institute of Technology, Stockholm, Sweden auch als Kapitel im Handbook of Networked and Embedded Control Systems (ISBN 978-0-8176-3239-7), Birkhäuser Boston, pages 741-765, Januar 2005. – URL http://www.s3.kth.se/~kallej/papers/can_necs_handbook05.pdf. – Zugriffsdatum: 31.01.2010

[29] Kiencke, Uwe: Controller Area Network - from Concept to Reality. In: *1st international CAN Conference (iCC '94)* University of Karlsruhe (Veranst.), 1994

[30] Lackner, Ursula: *Die (R)Evolution der Ökonomie - Einige Implikationen der Netzwerkökonomie gezeigt am Beispiel des Informationssektors.* Diplomarbeit an der Karl-Franzens Universität Graz, April 2002. – URL http://stefan.schleicher.wifo.at/down/da/DA_Lackner_2.pdf. – Zugriffsdatum: 31.01.2010

[31] Lawrenz, Wolfhard: *CAN Controller Area Network - Grundlagen und Praxis.* 5. Wiesbaden : Hüthig, 2010. – Enthält ein Kapitel über SocketCAN von Oliver Hartkopp und Jens Krüger. – ISBN 3-778-52780-0

[32] Leffler/McKusick/Karels/Quarterman: *The Design and Implementation of the 4.3BSD Unix Operating System.* Addison-Wesley Publishing Company, Inc., 1990. – ISBN 3-89319-239-5

[33] Livani, M. A. ; Kaiser, J.: EDF Consensus on CAN Bus Access for Dynamic Real-Time Applications. In: *Lecture Notes on Computer Science, vol. 1388* (1998), S. 1088–1097. – URL http://www-ivs.cs.uni-magdeburg.de/eos/forschung/publications/1998/EOS-1998.000-LK.pdf. – Zugriffsdatum: 27.07.2010

[34] Livani, M. A. ; Kaiser, J. ; Jia, W.: Scheduling Hard and Soft Real Time Communication in the Controller Area Network (CAN). In: *23rd IFAC/IFIP Workshop on Real Time Programming*. Shantou, China, 1998. – URL http://www-ivs.cs.uni-magdeburg.de/eos/forschung/publications/1998/EOS-1998.000-LKJ.pdf. – Zugriffsdatum: 27.07.2010

[35] McCanne, Steven ; Jacobson, Van: *The BSD Packet Filter: A New Architecture for User-level Packet Capture*. URL http://www.tcpdump.org/papers/bpf-usenix93.pdf. – Zugriffsdatum: 31.01.2010, 19 Dezember 1992

[36] Microchip Technology: *MCP2515 Stand-Alone CAN Controller With SPI Interface*, 10 Mai 2007. – URL http://ww1.microchip.com/downloads/en/DeviceDoc/21801e.pdf. – Zugriffsdatum: 31.01.2010

[37] Mierau, Matthias: *Paralleles Update von Steuergeräten*. Studienarbeit an der Fachhochschule Braunschweig/Wolfenbüttel, 30 Januar 2009

[38] Motorola Inc. (Freescale Semiconductors): *Serial Peripheral Interface (SPIV3) Block Description Version 3.06*, Januar 2000. – URL http://www.freescale.com/files/microcontrollers/doc/ref_manual/S12SPIV3.pdf. – Zugriffsdatum: 31.01.2010

[39] Müller, Daniel: *Buslastanalyse und -senkung der CAN-Busse im Automobil*. Masterarbeit an der Fachhochschule Furtwangen, 31 März 2006. – URL http://webuser.hs-furtwangen.de/~reich/Masterarbeiten/MasterThesisDanielMueller.pdf. – Zugriffsdatum: 31.01.2010

[40] Naujoks, André: *Erstellung einer abstrakten Fahrzeugschnittstelle mit anschließendem Benchmarking*. Studienarbeit an der Otto-von-Guericke Universität Magdeburg, Institut für Eingebettete Systeme und Betriebssysteme, April 2007

[41] Naujoks, André: *Analyse und Optimierung einer Fahrzeugschnittstelle*. Diplomarbeit an der Otto-von-Guericke Universität Magdeburg, Institut für Eingebettete Systeme und Betriebssysteme, März 2008

[42] Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug (OSEK): *OSEK Netzwerkmanagement OSEK NM (Version 2.5.3)*, 26 Juli 2004. – URL http://portal.osek-vdx.org/files/pdf/specs/nm253.pdf. – Zugriffsdatum: 31.01.2010

[43] Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug (OSEK): *OSEK Transportprotokoll OSEK COM (Version 3.0.3)*, 20 Juli 2004. – URL http://portal.osek-vdx.org/files/pdf/specs/osekcom303.pdf. – Zugriffsdatum: 31.01.2010

[44] Ohly, Patrick ; Lombard, David N. ; Stanton, Kevin B.: *Hardware Assisted Precision Time Protocol. Design and case study.*, Oktober 2008. –

URL http://www.linuxclustersinstitute.org/conferences/archive/2008/PDF/Ohly_92221.pdf. – Zugriffsdatum: 31.01.2010

[45] Philips Semiconductors (NXP): *I²C-Bus Specification Version 2.1*, Januar 2000. – URL http://www.nxp.com/acrobat/literature/9398/39340011.pdf. – Zugriffsdatum: 31.01.2010

[46] Philips Semiconductors (NXP): *Philips SJA1000 CAN Controller Datasheet*, 04 Januar 2000. – URL http://www.nxp.com/documents/data_sheet/SJA1000.pdf. – Zugriffsdatum: 18.03.2010

[47] Philips Semiconductors (NXP): *Philips TJA1054 Fault-tolerant CAN Transceiver Datasheet*, 23 März 2004. – URL http://www.nxp.com/acrobat/datasheets/TJA1054_3.pdf. – Zugriffsdatum: 31.01.2010

[48] Philips Semiconductors (NXP): *Philips TJA1041 High-speed CAN Transceiver Datasheet*, 29 Juli 2008. – URL http://www.nxp.com/acrobat/datasheets/TJA1041A_4.pdf. – Zugriffsdatum: 31.01.2010

[49] Postel, J.: *Internet Protocol*. RFC 791 (Standard). September 1981 (Request for Comments). – URL http://www.ietf.org/rfc/rfc791.txt. – Zugriffsdatum: 31.01.2010. – Updated by RFCs 1349

[50] Postel, J.: *Transmission Control Protocol*. RFC 793 (Standard). September 1981 (Request for Comments). – URL http://www.ietf.org/rfc/rfc793.txt. – Zugriffsdatum: 31.01.2010. – Updated by RFCs 1122, 3168

[51] Reynolds, J. ; Postel, J.: *Assigned Numbers*. RFC 1700 (Standard). Oktober 1994. – URL http://www.ietf.org/rfc/rfc1700.txt. – Zugriffsdatum: 31.01.2010

[52] Robert Bosch GmbH: *Controller Area Network (CAN) Specification 2.0*. – URL http://www.semiconductors.bosch.de/pdf/can2spec.pdf. – Zugriffsdatum: 31.01.2010

[53] Robert Bosch GmbH: *Mobile Communication Network (MCNet) Specification 1.2*, 08 Juli 1997

[54] Robert Bosch GmbH: *Mobile Communication Network (MCNet) System Concept 0.9*, 17 Juni 1999

[55] Rubini, Alessandro ; Corbet, Jonathan: *Linux Device Drivers*. Second. O'Reilly & Associates, Inc., 2001. – ISBN 0-596-00008-1

[56] SAE J1939-21:2006: *J1939 Data Link Layer – This particular document, SAE J1939-21, describes the data link layer using the CAN protocol with 29-bit Identifiers*. Society of Automotive Engineers (SAE), 400 Commonwealth Drive Warrendale, PA 15096-0001, USA. – Issued by SAE Truck Bus Control And Communications Network Committee

[57] Schick, Andrea: *Soft- und Hardwareentwicklung eines 'Low Level CAN Framework Micro Device' auf der Basis eines Standard Mikrocontrollers der Firma Atmel.* Diplomarbeit an der Fachhochschule Zittau/Görlitz, 09 Dezember 2003

[58] Softing AG: *Bus-Interface EDICcard2 Datenblatt*, 13 Dezember 2007. – URL http://www.softing.com/home/de/pdf/ae/datasheet/hardware/Softing-DB_EDICcard2_D.pdf. – Zugriffsdatum: 30.03.2010

[59] Sojka, Michal ; Pisa, Pavel: Timing analysis of Linux CAN drivers. In: *Eleventh Real-Time Linux Workshop (Open Source Automation Development Lab 2009).* Dresden, Germany, 28 Oktober 2009. – URL http://lwn.net/images/conf/rtlws11/papers/proc/p30.pdf. – Zugriffsdatum: 16.05.2010

[60] Venkateswaran, Sreekrishnan: Writing a Kernel Line Discipline, Linux Magazine, 15 März 2005. – URL http://www.linux-mag.com/id/1891. – Zugriffsdatum: 31.01.2010

[61] Volkswagen AG: *CAN Transportprotokoll TP1.6 (Version 1.6.6)*, 11 Juni 2001

[62] Volkswagen AG: *CAN Transportprotokoll TP2.0 (Version 1.1)*, 02 Februar 2001

[63] Zimmermann, Werner ; Schmidgall, Ralf: *Bussysteme in der Fahrzeugtechnik: Protokolle und Standards.* 2. Wiesbaden : Vieweg, 2007. – ISBN 3-834-80235-2

# Abbildungsverzeichnis


2.1 Das Prinzip eines CAN Netzwerkes . . . . . . . . . . . . . . . . . . . . . . . . 10
2.2 Kollisionsauflösung durch bitweise Arbitrierung [28] . . . . . . . . . . . . . . 12
2.3 Standard Frame Format (SFF) [1] . . . . . . . . . . . . . . . . . . . . . . . . . 14
2.4 Extended Frame Format (EFF) [1] . . . . . . . . . . . . . . . . . . . . . . . . 14
2.5 Prinzip der Kommunikation bei ISO 15765-2 [27] . . . . . . . . . . . . . . . . 26
2.6 Tachograf Botschaftsdefinition 0xFE6C (FMS) [8] . . . . . . . . . . . . . . . 29
2.7 Beispiel einer PDO Kommunikation bei CANopen [8] . . . . . . . . . . . . . 31

3.1 Blockschaltbild des SJA1000 CAN Controllers [46] . . . . . . . . . . . . . . . 39
3.2 Register Layout bei CAN Controllern (Beispiele) . . . . . . . . . . . . . . . . 40
3.3 Mögliche Latenzzeiten bei CAN Schnittstellenkarten [10] . . . . . . . . . . . 42
3.4 *Passive* (links) und *aktive* CAN Hardware . . . . . . . . . . . . . . . . . . . . 43
3.5 CAN Zugriff über Bibliotheksfunktionen (rechts) . . . . . . . . . . . . . . . . 49
3.6 CAN Protokolle im Anwendungskontext . . . . . . . . . . . . . . . . . . . . . 52

4.1 Konzept der Anbindung von Netzwerkgeräten . . . . . . . . . . . . . . . . . . 73
4.2 Protokollfamilien im Linux-Kernel . . . . . . . . . . . . . . . . . . . . . . . . 80
4.3 CAN Protokolle der Protokollfamilie PF_CAN . . . . . . . . . . . . . . . . . 85

5.1 Testumgebung PEAK PCMCIA CAN Treiber . . . . . . . . . . . . . . . . . . 96
5.2 Grafische Oberfläche von LTTng . . . . . . . . . . . . . . . . . . . . . . . . . 98
5.3 LTTng Messpunkte am PEAK PCMCIA CAN Treiber . . . . . . . . . . . . . 99
5.4 UDS-fähiges Climatronic Steuergerät . . . . . . . . . . . . . . . . . . . . . . . 106
5.5 CAN Netzwerk Konzept unter Windows (rechts) [31] . . . . . . . . . . . . . . 111
5.6 Die Softwarestruktur auf dem 8-Bit ATmega161 [57] . . . . . . . . . . . . . . 115

A.1 slcanpty als Emulation einer seriellen CAN Hardware . . . . . . . . . . . . . 159
A.2 Emulation eines zeichenorientierten CAN Treibers . . . . . . . . . . . . . . . 160
A.3 Beispiel einer CAN Queueing Discipline . . . . . . . . . . . . . . . . . . . . . 167
B.1 Unterschiedliche Adressierungen bei Ethernet / CAN . . . . . . . . . . . . . 170
B.2 Das CAN Subsystem im Linux-Kernel . . . . . . . . . . . . . . . . . . . . . . 171
B.3 Das CAN Subsystem-Kernmodul im Linux-Kernel . . . . . . . . . . . . . . . 172
B.4 Beispiel für die Anwendung des Multiplexfilters . . . . . . . . . . . . . . . . 192

# Tabellenverzeichnis


2.1 Buslast bei einem Volkswagen Nutzfahrzeug . . . . . . . . . . . . . . . . . . 18
2.2 Betrachtete CAN Transportprotokolle im Vergleich . . . . . . . . . . . . . 27
2.3 Ausschnitt der CAN Nachrichten an der FMS Schnittstelle eines MAN Nutzfahrzeugs Baujahr 2004 . . . . . . . . . . . . . . . . . . . . . . . . . 28

3.1 Konzeptvergleich: *Aktive* und *passive* CAN Hardware . . . . . . . . . . . 44
3.2 Beispiel für Gerätedateien in Unix Dateisystemen . . . . . . . . . . . . . 46
3.3 Schichtenmodell des CAN Abstraction Layer für Windows . . . . . . . . 51

4.1 Verwendbarkeit 'einfacher' CAN-ID/Masken Filter . . . . . . . . . . . . . 83
4.2 Analogie zwischen Internet- und CAN-Sockettypen . . . . . . . . . . . . 90
4.3 Betriebssystemaufrufe für verschiedene Sockettypen . . . . . . . . . . . . 91

5.1 Performanzmessung: 'C'hardev vs 'N'etdev . . . . . . . . . . . . . . . . . . 100
5.2 Performanzmessung: Inhaltsfilterung . . . . . . . . . . . . . . . . . . . . . 102
5.3 Reduzierung des Datenaufkommens durch SocketCAN (Datenaktualisierung auf dem CAN Bus ca. alle 10ms) . . . . . . . . . . . . . . . . . . . 103
5.4 Reduzierung des Datenaufkommens durch SocketCAN bei einer Drosselung auf 200ms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103
5.5 ISO-TP: Zeit bis zur Aussendung des ersten Datenframes . . . . . . . . . 108
5.6 ISO-TP: Zeit bis zur Aussendung des zweiten Datenframes . . . . . . . . 109
5.7 ISO-TP Messungen: Vergleich der Treiberperformanz . . . . . . . . . . . 110

A.1 Netzwerkgerätespezifische Empfangslisten . . . . . . . . . . . . . . . . . . . 140

# A Details der Linux Implementierung

In diesem Kapitel werden technische Details der Linux Implementierung sowie die Realisierung nicht-funktionaler Anforderungen für eine Integration des Controller Area Networks in Mehrbenutzersyteme vorgestellt, auf die im Rahmen der "Konzeptentwicklung des CAN Zugriffes über Netzwerkschnittstellen" ab Seite 72 aus Gründen der Lesbarkeit und Übersichtlichkeit verzichtet wurde.

Als technische Details werden folgende Themen erläutert:

- Busfehlersignalisierung durch Error Messages
- Filterkonzept im Software-Interrupt
- Zeitstempel für CAN Nachrichten
- Echo Funktionalität für CAN Netzwerktreiber
- Der virtuelle CAN Netzwerktreiber (vcan)

Zusätzlich zu diesen technischen Details ergeben sich aus Kapitel 4.3 nicht-funktionale Anforderungen für einen produktiven Einsatz des entwickelten CAN Subsystems, deren unterschiedliche Lösungen in diesem Kapitel dargestellt werden.

Im Gegensatz zu funktionalen Anforderungen werden dabei Eigenschaften wie Robustheit, Benutzbarkeit, Testbarkeit, Migrationsfähigkeit und Portabilität bewertet. So müssen grundsätzliche Probleme wie das Hinzufügen und Entfernen von CAN Netzwerkgeräten zur Laufzeit gelöst werden, damit das System beim Entfernen eines unter Last arbeitenden CAN-USB-Adapters nicht instabil wird oder gar *abstürzt*[1]. Zusätzlich müssen Möglichkeiten zur Fehlersuche geschaffen werden, um einen tatsächlichen Implementierungsfehler des CAN Subsystems von einer fehlerhaften Benutzung durch den Anwender differenzieren zu können.

- Hot(un)plugging von CAN Geräten
- Anzeige der internen Datenstrukturen zur Laufzeit
- Migration existierender CAN Anwendungen

Abschließend werden Konzepte für zukünftige Entwicklungen aufgezeigt.

[1] "That must be why we're not shipping Windows 98 yet." (Bill Gates' Kommentar zum abgestürzten Windows 98 bei der Neuvorstellung der USB Funktionalität, Fachmesse COMDEX, April 1998)

## A.1 Busfehlersignalisierung durch Error Messages

Mögliche Fehlerzustände der in Kommunikation auf dem CAN Bus und die vom CAN Controller im Fehlerfall erwarteten Verhaltensweisen werden detailliert in der CAN Spezifikation[52] von der Firma Bosch beschrieben. Die durch den CAN Controller detektierten und signalisierten Fehlerzustände auf dem CAN Bus können in kommerziellen Werkzeugen als Zusatzinformation bei der Darstellung des CAN Datenverkehrs aufgenommen und verarbeitet werden.

Dementsprechend wurde für das im Rahmen dieser Arbeit entwickelte CAN Subsystem ein Konzept gewählt, dass eine Übertragung der CAN Fehlerzustände in der Form eines CAN Frames mit einem gesetzten Fehler-Flag realisiert. Diese so genannten "Error Messages" auf dem selben Kanal zu übertragen, wie CAN Botschaften mit Anwendungsinformationen, hat folgende Vorteile:

- Einfache Übertragung des Konzepts auf weitere Betriebssysteme
- Verfügbarkeit der Informationen bei Bedarf an jedem RAW-Socket (siehe B.2)
- Filterung auf bestimmte Fehlerzustände in den Error Messages analog zu den bekannten CAN Identifier Filtern (siehe Kapitel A.2)
- Erzeugung von Error Messages analog zu CAN Botschaften im CAN Netzwerktreiber
- Sicherstellung der korrekten Reihenfolge von Daten und Fehlern im CAN Netzwerktreiber
- Keine Definition einer zusätzlichen Programmierschnittstelle

Der Inhalt einer solchen besonderen CAN Nachricht ist durch das Setzen des Fehler-Flags (`CAN_ERR_FLAG 0x20000000U`) im CAN Identifier einer CAN Botschaft fest definiert und zeigt die Fehlerzustände des CAN Controllers durch eine Kodierung des CAN Identifiers in der folgenden Form an:

```
/* error class (mask) in can_id */
#define CAN_ERR_TX_TIMEOUT   0x00000001U /* TX timeout (by netdevice driver) */
#define CAN_ERR_LOSTARB      0x00000002U /* lost arbitration    / data[0]    */
#define CAN_ERR_CRTL         0x00000004U /* controller problems / data[1]    */
#define CAN_ERR_PROT         0x00000008U /* protocol violations / data[2..3] */
#define CAN_ERR_TRX          0x00000010U /* transceiver status  / data[4]    */
#define CAN_ERR_ACK          0x00000020U /* received no ACK on transmission */
#define CAN_ERR_BUSOFF       0x00000040U /* bus off */
#define CAN_ERR_BUSERROR     0x00000080U /* bus error (may flood!) */
#define CAN_ERR_RESTARTED    0x00000100U /* controller restarted */
```

Die Abbildung der einzelnen Problemklassen in einzelne Bitwerte erlaubt die Filterung nach einzelnen oder mehreren Fehlerzuständen analog der binären Akzeptanzfilterung für CAN Identifier (siehe Kapitel 4.3.7.1 (CAN RAW Protokoll)).

Die unterschiedlichen Fehlerzustände erfordern unter Umständen die Übertragung zusätzlicher Informationen. So ist für den Zustand "lost arbitration" beispielsweise relevant, an welcher Bit-Position die Arbitrierung verloren wurde (siehe Kapitel 2.2.2). Zur Übertragung zusätzlicher Informationen einer Error Message werden die Nutzdaten des verwendeten CAN Frames genutzt. Im genannten Beispiel für den Zustand "lost arbitration" wird die zugehörige Bit-Position im ersten Datenbyte der Error Message abgelegt.

Die Konfiguration des Filters für Error Messages wird unabhängig von der Konfiguration der CAN Akzeptanzfilter der RAW-Sockets durchgeführt. Im Gegensatz zur Möglichkeit auf einem RAW-Socket n ($n \in \mathbb{N}$) Akzeptanzfilter mit Hilfe der Socket-Option CAN_RAW_FILTER definieren zu können, kann nur genau ein Filter für Error Messages mit der Socket-Option CAN_RAW_ERR_FILTER gesetzt werden. Als Voreinstellung ist dieser Error Filter auf `0x00000000U` gesetzt, wodurch keine Fehlerzustände an die Anwendung gesendet werden.

Eine detaillierte Liste der Error Messages ist in der Include Datei
`include/linux/can/error.h`
definiert und die Konstanten für die Socket-Optionen befinden sich in der Include Datei
`include/linux/can/raw.h`

## A.2 Filterkonzept im Software-Interrupt

Die Verteilung von empfangenen CAN Nachrichten (siehe Kapitel 4.3.5) wurde zunächst durch eine einfach verkettete Liste von Filterelementen realisiert. Beim Empfang einer CAN Botschaft mussten jeweils alle Vergleichsoperationen für die einzelnen Filterelemente der verketteten Liste durchgeführt werden:

- Vergleich auf das korrekte CAN Netzwerkgerät
- Vergleich der empfangenen CAN-ID mit der gewünschten CAN-ID (bei einer gewünschten CAN-ID Wert '-1' wurden alle empfangenen CAN-IDs akzeptiert)

Bei einem durchaus realistischen Anwendungsfall bei der Ausführung von fünf **cansniffer** Anwendungen (siehe Kapitel C.1.2) werden $2048 * 5 = 10240$ Filtereinträge für einzelne CAN Identifier generiert, die bei *jedem* empfangenen CAN Frame *vollständig* durchlaufen werden müssen.

Aufgrund der Eigenschaften der empfangenen CAN Nachrichten und der Tatsache, das sie immer von einem konkreten CAN Netzwerkgerät empfangen werden, sind jedoch

effektive Listenstrukturen pro CAN Netzwerkgerät möglich, die die Laufzeit zum Empfangszeitpunkt erheblich reduzieren können.

Die sehr einfach gehaltene Programmierschnittstelle zur Konfiguration einer beliebigen Anzahl von Akzeptanzfiltern pro RAW Socket bietet nach aktuellem Stand folgende Möglichkeiten:

1. Binärer Akzeptanzfilter mit **recv_id ∧ can_mask = can_id ∧ can_mask**
2. Invertierung des binären Akzeptanzfilters
3. Empfang von allen CAN Netzwerkgeräten bei Interface-Index = Null

Der letzte Punkt lässt sich vergleichsweise einfach realisieren, indem zusätzlich zu den erwähnten 'Listenstrukturen pro Netzwerkgerät' eine weitere Struktur angelegt wird, die beim Empfang von CAN Botschaften von allen Netzwerkgeräten zu Filtervergleichen herangezogen wird.

Bei der Analyse der durch den binären Akzeptanzfilter möglichen Konfigurationen können zusätzliche Unterteilungen innerhalb der Netzwerkgerätespezifischen Empfangslisten identifiziert werden. Wird beispielsweise bei einem konfigurierten Filter durch die `can_mask` und die `can_id` *genau ein* 11-Bit Identifier (SFF CAN Frame) selektiert, kann dieses Filterelement einer Liste von Filterelementen hinzugefügt werden, die zur Laufzeit *nur* auf die Übereinstimmung des CAN Identifiers prüft, ohne diesen zuvor mit der `can_mask` zu maskieren.

Der Filteraufwand zur Laufzeit kann in diesem besonderen Fall noch weiter reduziert werden, indem für jeden möglichen SFF CAN Identifier eine eigene Liste von Filterelementen angelegt wird, also eine Tabelle von $2^{11} = 2048$ Filterlisten. Beim Empfang eines SFF CAN Frames kann über den Index des empfangenen CAN-ID Wertes in der Tabelle direkt auf die Filterliste zugegriffen werden und die Konsumenten ohne weitere Prüfungen angesprochen werden.

Das Eingangs beschriebene Anwendungsbeispiel mit 10.240 Filtereinträgen verteilt auf fünf CAN Netzwerkgeräte ist mit diesem Ansatz extrem effizient zur Laufzeit zu behandeln. Allerdings erscheint der Ansatz 2048 Zeiger auf jeweils eine Filterliste nicht sehr effizient mit dem vorhandenen Hauptspeicher umzugehen ($2048 * 4$ Byte $= 8192$ Byte für die Zeigertabelle).

Das Konzept der Aufteilung der per-Netzwerkgerät Listenstrukturen in weitere, in diesem Fall sogar mehr als 2048, Zeigerlisten führt zu einem so genannten Time-Space-Trade-Off, bei dem sich Laufzeit-Performanz und Speicherplatz reziprok verhalten.

Die Netzwerkgerätespezifischen Empfangslisten für das CAN Subsystem wurden auf der Basis vorhandener Anwendungsfälle im Kraftfahrzeug definiert. Die Tabelle mit 2048

Zeigern auf Filtereinträge für SFF Frames ist konkret für Anwendungen im Fahrzeugumfeld optimiert. Bestünde eine analoge Anforderung für den effizienten Zugriff auf einzelne CAN Identifier mit 29 Bit Länge (EFF Frames), würde sich ein entsprechendes Hash-Verfahren zu dessen Realisierung anbieten (siehe Kapitel A.9.2).

Die derzeitige Ausführung der Netzwerkgerätespezifischen Empfangsliste umfasst 2053 Zeigerlisten und benötigt für jedes CAN Netzwerkgerät 8240 Byte im Arbeitsspeicher (32-Bit Linux). Das Konzept und die dadurch mögliche Reduzierung von Vergleichsoperationen zur Laufzeit sollen im Folgenden vorgestellt werden.

Ein Filterelement wird durch eine Registrierung eines Filters durch ein CAN Protokoll (RAW, BCM, Transportprotokolle) bei der Protokollfamilie PF_CAN erstellt. Das Filterelement enthält folgende Strukturelemente:

```
struct receiver {
        struct hlist_node list;
        struct rcu_head rcu;
        canid_t can_id;
        canid_t mask;
        unsigned long matches;
        void (*func)(struct sk_buff *, void *);
        void *data;
        char *ident;
};
```

| Element | Funktion |
|---|---|
| struct hlist_node list | Listenstrukturelement |
| struct rcu_head rcu | Listenverwaltungselement (Read/Copy/Update) |
| canid_t can_id | CAN Identifier für den Binärvergleich |
| canid_t mask | Binärmaske für den Binärvergleich |
| unsigned long matches | Anzahl der akzeptierten Frames durch diesen Filter |
| void (*func)(struct sk_buff *, void *) | Callback Funktion in das CAN Protokoll |
| void *data | Zusätzliche Verwaltungsdaten für das CAN Protokoll |
| char *ident | Zeiger auf eine protokollspezifische Zeichenkette |

Im nicht optimierten Fall einer einzelnen einfach verketteten Liste, würden beim Empfang einer CAN Botschaft für jeden Filtereintrag diese Schritte erfolgen:

- Überprüfung des korrekten CAN Netzwerkgerätes
- Binärvergleich $\mathbf{recv_id} \wedge \mathbf{can_mask} = \mathbf{can_id} \wedge \mathbf{can_mask}$
- Aufruf der Callback-Funktion bei Akzeptanz des Filters

Durch die Netzwerkgerätespezifischen Empfangslisten wird der erste Vergleich ("Überprüfung des CAN Netzwerkgerätes") für jedes einzelne Filterelement obsolet.

Zusätzlich wird beim Eintragen der Werte in das Filterelement die Maskierung **can_id** $\wedge$ **can_mask** sofort berechnet und in den `can_id` Wert des Filters gespeichert. Zur Laufzeit ergibt sich dadurch für den Binärvergleich eine entsprechende Einsparung zu

$$\textbf{recv_id} \wedge \textbf{can_mask} = \textbf{can_id}_{maskiert}$$

Für die verschiedenen Filterlisten sind zur Laufzeit folgende Vergleichsoperationen notwendig:

| Filterliste | Vergleichsoperation |
|---|---|
| rx_err | recv_id $\wedge$ ERR_FLAG => recv_id $\wedge$ can_mask = can_id$_{maskiert}$ |
| rx_all | (keine) |
| rx_fil | recv_id $\wedge$ can_mask = can_id$_{maskiert}$ |
| rx_inv | recv_id $\wedge$ can_mask != can_id$_{maskiert}$ |
| rx_eff | recv_id = can_id$_{maskiert}$ (zukünftig ggf. Hash-Funktion - siehe A.9.2) |
| rx_sff[2048] | (keine) |

Tabelle A.1: Netzwerkgerätespezifische Empfangslisten

Durch die Vorverarbeitung und die Anordnung in spezifische Filterlisten können die Vergleichsoperationen zur Laufzeit erheblich reduziert werden. Die in Kapitel 4.1 im Rahmen der Anforderungsanalyse formulierten Mehrbenutzer-Anforderungen an die Akzeptanzfilter lassen sich ohne Vergleichsoperationen abbilden. Eine Ausnahme stellt nur ein Anwendungsfall aus Kapitel 4.1.2 dar, wenn zusätzliche Akzeptanzfilter ("KONFIGURATION DER EMPFANGSFILTER DES CAN CONTROLLERS") definiert werden.

## Fazit

Die Realisierung der Akzeptanzfilter im Softwareinterrupt ist zur Umsetzung der CAN Integration in Mehrbenutzersysteme nötig. Allerdings lässt sich die Akzeptanzfilterung nicht mit einer Komplexität von $\mathcal{O}(1)$ realisieren, da das aktuelle Filterkonzept Laufzeitabhängigkeiten in Bezug auf bestimmte Eingangsgrößen aufweist:

**d** Anzahl der Devices (CAN Netzwerkgeräte)

**f** Anzahl der Filter (für ein bestimmtes Netzwerkgerät)

**f**$_{all}$ Anzahl der Filter (für den Empfang von allen Netzwerkgeräten)

Im schlechtesten Fall kann von einer Komplexität von $\mathcal{O}(\mathrm{d} + \mathrm{f} + \mathrm{f}_{all})$ beim Empfang von CAN Botschaften ausgegangen werden. Tatsächlich ist diese Komplexität für bisherige Anwendungsfälle mit 'realen' CAN Netzwerkgeräten nicht vorhanden, weil beispielsweise zehn Anwendungen die auf fünf reale CAN Busse gleichzeitig zugreifen schon zu den komplexesten Anwendungsfällen nach dem Stand der Technik zählen.

Die in Kapitel A.5 vorgestellte Möglichkeit, zur Laufzeit beliebig viele 'virtuelle' CAN Netzwerkgeräte zu erzeugen, kann bei der Komplexitätsbetrachtung zu 'großen' Werten für 'd' (Anzahl der Devices) führen, wobei die Art der Filterverwendung 'f' und '$f_{all}$' voraussichtlich gleich bleiben wird. Zur Entfernung der (offensichtlich kritischen) Anzahl der Netzwerkgeräte aus der Komplexitätsbetrachtung muss das lineare Suchen nach dem betroffenen Netzwerkgerät in der Liste der netzwerkgerätespezifischen Empfangslisten unterbleiben. Dieses kann beispielsweise dadurch realisiert werden, dass in der Datenstruktur des Netwerkgerätes eine Referenz auf die zugehörige netzwerkgerätespezifische Empfangsliste angelegt wird.

Seit Linux Kernel 2.6.26 (freigegeben am 13. Juli 2008) existiert in der Datenstruktur für Netzwerkgeräte (`struct net_device`) ein für diese Zwecke verwendbarer Zeiger mit der Bezeichnung `ml_priv` ('mid-layer private'). Die folgende Änderung am Linux Kernel setzt dieses Konzept um und wurde erfolgreich erprobt:

```
--- net/can/af_can.c (Revision 1033)
+++ net/can/af_can.c (Arbeitskopie)
@@ -374,6 +374,13 @@
         struct dev_rcv_lists *d = NULL;
         struct hlist_node *n;

+        if (!dev)
+                return &can_rx_alldev_list;
+
+        /* do not search when we have a direct reference */
+        if (dev->ml_priv)
+                return (struct dev_rcv_lists *)dev->ml_priv;
+
         /*
          * find receive list for this device
          *
@@ -895,6 +902,7 @@
                 d->dev = dev;

                 spin_lock(&can_rcvlists_lock);
+                dev->ml_priv = d;
                 hlist_add_head_rcu(&d->list, &can_rx_dev_list);
                 spin_unlock(&can_rcvlists_lock);
```

Es verbleibt im schlechtesten Fall eine Komplexität von $\mathcal{O}(f + f_{all})$ beim Empfang von CAN Botschaften. Diese Verbesserung wird ab Linux Kernel 2.6.34 integriert sein.

## A.3 Zeitstempel für CAN Nachrichten

Für eine Analyse von Netzwerken wird beispielsweise unter Linux beim Empfang von Ethernetframes im Socket-Buffer (siehe Kapitel 4.3.2) die Empfangszeit abgelegt. In eingebetteten Systemen kann das Erstellen des Zeitstempels bei Bedarf auch abgeschaltet werden, weil der Zugriff auf die entsprechenden Hardwarezeitgeber und zusätzliche Verfahren zur Erhöhung der Genauigkeit des Zeitstempels Rechenzeit benötigen.

Abhängig von der jeweiligen CAN Hardware bieten verschiedene CAN Lösungen den Zugriff auf einen Hardware-Zeitstempel, der beim Empfang einer CAN Botschaft gesetzt wird und zusammen mit den Inhalten der CAN Botschaft ausgelesen werden kann.

Die spezifischen CAN Hardware-Zeitstempel haben allerdings im Zusammenhang mit der Integration in Mehrbenutzersysteme verschiedene Nachteile:

- Unterschiedliche zeitliche Auflösungen
- Unterschiedliche oder keine Konzepte zur Drift-Kompensation
- Keine Synchronisierung der Zeitbasis zu anderen CAN Controllern im PC
- Es gibt CAN Hardware ohne Hardware-Zeitstempel

In Anbetracht der zu erwartenden Probleme bei der Einbindung der verschiedenen CAN Hardware-Zeitstempel wurde auf deren Verwendung vollständig verzichtet.

Stattdessen wird die im Linux Kernel vorhandenen Infrastruktur für Zeitstempel von Netzwerkgeräten mit einer Auflösung von bis zu einer Nanosekunde verwendet. Das CAN Subsystem verwendet dadurch den Stand der Technik analog zu Ethernetkarten. Dieses erlaubt zudem die Verwendung der bekannten Programmierschnittstellen zum Auslesen von Zeitstempeln aus der Socket-Schnittstelle (siehe Kapitel B.1.6).

Die Genauigkeit des Netzwerkzeitstempels wird bei dem gewählten Verfahren durch die Latenzzeit für eine Unterbrechungsanforderung bestimmt. Diese ist nicht deterministisch im Bereich von 10-200µs anzunehmen[2], was bei einer minimalen zeitlichen Ausdehnung eines CAN Frames von 47µs für eine detaillierte Busanalyse im direkten Vergleich mit kommerziellen Lösungen unter Verwendung von *aktiver* CAN Hardware nicht hinreichend genau ist.

Für eine detaillierte Busanalyse ist eine CAN Hardware mit Zeitstempeln erforderlich. Im Hinblick auf eine mögliche Realisierung von Hardware Zeitstempeln für das CAN Subsystem sei auf die Arbeiten von Patrick Ohly bei der Firma Intel[44] verwiesen, der für entsprechende Analysen von *aktiven* 10GBit Ethernetkarten eine Infrastruktur zur Übertragung von Hardware-Zeitstempeln in das Linux Betriebssystem eingebracht hat.

[2] `http://trego.co.il/images/MovingToLinux.pdf`, Folie 23

## A.4 Echo Funktionalität für CAN Netzwerktreiber

In Kapitel 4.3.6 (Lokales Echo gesendeter CAN Nachrichten) wurde die Motivation für ein Echo Funktionalität auf der Ebene der CAN Netzwerktreiber erläutert. Im Folgenden wird die zur Umsetzung der Echo Funktionalität realisierte Infrastruktur im CAN Subsystem beschrieben und welche Voraussetzungen eine Protokollimplementierung für die Protokollfamilie PF_CAN erfüllen muss, um die Mehrbenutzerfunktionalität zu unterstützen.

Zum Versenden eines Socket-Buffers (siehe Kapitel 4.3.2) über ein Netzwerkgerät, muss zunächst die Datenstruktur des Socket-Buffers mit `sock_alloc_send_skb()` mit einer ausreichenden Nutzdatenlänge für eine CAN Botschaft erzeugt werden. Neben den Nutzdaten müssen in die Datenstruktur des Socket-Buffers eine Referenz auf die eigene Socket-Struktur und eine Referenz auf das zu verwendende Netzwerkgerät kopiert werden.

Grundsätzlich kann ein solchermaßen vorbereiteter Socket-Buffer von einem Netzwerkprotokoll mit Hilfe der Funktion `dev_queue_xmit()` vom Betriebssystem in die entsprechende Sendewarteschlange des Netzwerkgerätes eingehängt werden. Allerdings ist dadurch die Echo Funktionalität nicht gegeben (siehe PACKET-Socket in Kapitel 4.3.3).

Anstelle der Funktion `dev_queue_xmit()` wird für das Versenden von Socket-Buffern mit CAN Dateninhalten von der PF_CAN Infrastruktur eine Funktion `can_send()` zur Verfügung gestellt, welche analog zu `dev_queue_xmit()` den Socket-Buffer an das entsprechende CAN Netzwerkgerät weitergibt:

```
/**
 * can_send - transmit a CAN frame (optional with local loopback)
 * @skb: pointer to socket buffer with CAN frame in data section
 * @loop: loopback for listeners on local CAN sockets (recommended default!)
 *
 * Return:
 *  0 on success
 *  -ENETDOWN when the selected interface is down
 *  -ENOBUFS on full driver queue (see net_xmit_errno())
 *  -ENOMEM when local loopback failed at calling skb_clone()
 *  -EPERM when trying to send on a non-CAN interface
 *  -EINVAL when the skb->data does not contain a valid CAN frame
 */

int can_send(struct sk_buff *skb, int loop){}
```

Die Funktion `can_send()` führt zusätzliche Prüfungen an dem übergebenen CAN Socket-Buffer durch:

- Prüfung auf die korrekte Nutzdatenlänge für CAN Botschaften
- Prüfung auf die korrekte Nutzdatenlänge innerhalb der CAN Botschaften
- Prüfung auf das Vorhandensein des verwendeten CAN Netzwerkgerätes
- Prüfung, ob das CAN Netzwerkgerät betriebsbereit ist

Zusätzlich werden die für CAN Inhalte üblichen Konstanten (beispielsweise ETH_P_CAN) gesetzt sowie die Zeiger für Ethernet-typische Protokollinformationen zurückgesetzt, da diese für das Controller Area Network nicht relevant sind.

Für die Ausführung der Echo Funktionalität muss der Funktionsparameter `loop` auf einen Wert ungleich Null gesetzt werden - dieses stellt den Standard Anwendungsfall dar, der derzeit nur durch eine Konfiguration des RAW-Socket für die jeweilige RAW-Socket Instanz abgeschaltet werden kann (via `setsockopt()` siehe Kapitel B.2).

Abhängig von dem Funktionsparameter `loop` werden als Pakettyp (packet_type) des Socket-Buffers folgende Werte gesetzt:

**PACKET_LOOPBACK** wenn für das Packet ein Echo ausgeführt werden soll

**PACKET_HOST** wenn für das Packet kein Echo ausgeführt werden soll

Diese Pakettypen werden vom CAN Netzwerkgerät ausgewertet und die Echo Funktionalität entsprechend umgesetzt. Ein CAN Socket-Buffer, der vom System empfangen wurde, wird gemäß dem Kommunikationsverfahren des Controller Area Networks mit einem Pakettyp PACKET_BROADCAST gekennzeichnet.

Die Verfügbarkeit der Echo Funktionalität auf der CAN Treiberebene kann mit Hilfe des Bitwertes 'IFF_ECHO' in den so genannten Interface-Flags von Netzwerkgeräten geprüft werden.

Ist ein CAN Netzwerkgerät nicht fähig ein CAN Socket-Buffer nach erfolgreichem Versenden auf dem CAN Bus als Echo in die Empfangswarteschlange einzuhängen, kann dieses vom CAN Subsystem übernommen werden. Dazu wird im Rahmen von `can_send()` nach einem erfolgreichen Übermitteln des Socket-Buffers an das CAN Netzwerkgerät, der Socket-Buffer dupliziert und in die Empfangswarteschlange eingehängt. Die korrekte Reihenfolge der von CAN Anwendungen auf dem lokalen System empfangen CAN Botschaften kann bei diesem Verfahren nicht sichergestellt werden, weil die Arbitrierung der CAN Botschaften auf dem CAN Bus zu einer unterschiedlichen Reihenfolge der CAN Botschaften auf dem Medium geführt haben kann.

## A.5 Der virtuelle CAN Netzwerktreiber (vcan)

Der virtuelle CAN Bustreiber realisiert ein logisches CAN Netzwerkgerät, über das Anwendungen auf einem System ohne real vorhandene CAN Hardware kommunizieren können.

Die Idee entspricht der eines so genannten Loopback-Device, wie es für Internet Protokoll Verbindungen auf dem lokalen System genutzt wird. Durch die unterschiedliche Form der Adressierung im Internet Protokoll zu der Adressierung von Informationen im Controller Area Network kann das existierende Loopback-Device allerdings nicht verwendet werden. Beim Controller Area Network ergibt sich eine qualifizierte Angabe einer CAN Nachricht aus dem CAN Identifier und dem CAN Bus auf dem dieser Identifier gesendet wird: Eine Nachricht mit der CAN-ID 0x123 auf can0 kann grundsätzlich eine andere Bedeutung haben, als eine Nachricht mit der CAN-ID 0x123 auf can1.

Deshalb benötigt man im Allgemeinen mehr als ein virtuelles CAN Netzwerkgerät, wenn man beispielsweise ein Gateway-Steuergerät für ein Kraftfahrzeug realisieren oder simulieren möchte, das im realen Fahrzeug mit 5 verschiedenen CAN Bussen interagiert.

Die Motivation für das lokale Zurückspiegeln von CAN Nachrichten (loopback) ist in Kapitel 4.3.6 dargelegt. Die Installation und Einrichtung von virtuellen CAN Netzwerkgeräten wird in Kapitel B.1.3 beschrieben.

Mögliche Anwendungen für virtuelle CAN Netzwerkgeräte sind:

- CAN Softwareentwicklung ohne vorhandene Hardware
- Wiedergabe von aufgezeichneten CAN Log Dateien (siehe Kapitel C.1.4)
- Reduzierung von Programmierschnittstellen
- Verknüpfung von nicht-CAN Daten zur Aufzeichnung in Log Dateien

Als Anwendungsbeispiel für die zwei letztgenannten Punkte kann die Umsetzung von GPS Positionsinformationen auf den virtuellen CAN betrachtet werden. Dabei wurde beispielhaft ein Anwendungsprogramm realisiert, das über einen so genannten GPS Daemon[3] Positionsinformationen erhält und diese in Form von CAN Nachrichten auf den virtuellen CAN sendet.

Die GPS Informationen können auf diese Weise für verschiedene CAN Anwendungen genutzt werden, ohne das die CAN Anwendung sich zusätzlich mit dem Zugriff auf den GPS Daemon und der Dekodierung der gelieferten Datenformate auseinandersetzen muss.

[3]Ein Programm, das in einem Unix System im Hintergrund abläuft und bestimmte Dienste zur Verfügung stellt

Bei einer Aufzeichnung von Fahrzeugdaten von verschiedenen CAN Bussen kann mit diesem Verfahren beispielsweise der virtuelle CAN Bus 'vcan0' mit aufgezeichnet werden, auf dem die GPS Informationen gesendet werden. Man erhält dadurch in der Log Datei einen Mitschnitt, der die Fahrzeugsignale mit der jeweiligen Position des Fahrzeugs abbilden kann.

Der virtuelle CAN Netzwerktreiber ist Teil des Linux Kernels seit der Version 2.6.25 und kann in zwei Modi betrieben werden, die über einen Modulparameter 'echo' definiert werden können:

**echo = 0** Der VCAN Treiber gibt sich dem CAN Basismodul als Netzwerktreiber zu erkennen, der kein Loopback auf der Treiberebene unterstützt, wodurch das CAN Basismodul den Loopback einer entsprechenden CAN Nachricht selbst ausführt. In diesem Modus konsumiert der VCAN eine zu sendende CAN Nachricht ohne sie als empfangenes Paket zurückzuschicken. Dieses ist zur Vermeidung von unnötiger Prozessorlast die Grundeinstellung des virtuellen CAN Treibers.

**echo = 1** Der VCAN Treiber gibt sich dem CAN Basismodul als Netzwerktreiber zu erkennen, der Loopback auf der Treiberebene unterstützt. Ankommende CAN Nachrichten werden als empfangenes Paket wieder an das CAN Basismodul geschickt.

## A.6 Hot(un)plugging von CAN Geräten

Der Linux Kernel bietet im Umgang mit Netzwerktreibern so genannte Notifier-Listen an, über die sich die Statusänderungen von Netzwerkgeräten bis hin zu ihrer Entfernung aus dem System verfolgen lassen. Als besondere Herausforderung stellt sich das Entfernen von aktiv genutzter CAN Hardware dar, die beispielsweise als USB-Geräte oder PCMCIA-Karten im System vorhanden sind.

Für Linux Netzwerktreiber kann sich der Interessent für Statusänderungen mit seinem persönlichen Strukturelement in die Notifier-Liste einhängen. Das persönliche Strukturelement enthält einen Zeiger auf die Funktion, die bei einer Änderung eines Gerätestatus aufgerufen wird.

Die möglichen Statusänderungen sind in der Datei `linux/include/linux/notifier.h` beschrieben:

```
/* netdevice notifier chain */
#define NETDEV_UP        0x0001  /* For now you can't veto a device up/down */
#define NETDEV_DOWN      0x0002
#define NETDEV_REBOOT    0x0003  /* Tell a protocol stack a network interface
                                    detected a hardware crash and restarted
                                    - we can use this eg to kick tcp sessions
                                    once done */
#define NETDEV_CHANGE    0x0004  /* Notify device state change */
#define NETDEV_REGISTER 0x0005
#define NETDEV_UNREGISTER        0x0006
#define NETDEV_CHANGEMTU         0x0007
#define NETDEV_CHANGEADDR        0x0008
#define NETDEV_GOING_DOWN        0x0009
#define NETDEV_CHANGENAME        0x000A
#define NETDEV_FEAT_CHANGE       0x000B
#define NETDEV_BONDING_FAILOVER 0x000C
```

Für das CAN Subsystem werden nur die rot markierten Nachrichten ausgewertet, wobei vor der Bewertung der Nachricht zunächst geprüft werden muss, ob es sich um ein CAN Netzwerkgerät handelt (Gerätetyp ARPHRD_CAN). Über die Notifier-Listen werden grundsätzlich Statusänderungen für alle Typen von Netzwerkgeräten - wie beispielsweise LAN oder WLAN Adaptern - angezeigt.

Das CAN Basismodul nutzt in `af_can.c` die Notifier NETDEV_REGISTER und NETDEV_UNREGISTER, um die per-Interface Listenstrukturen anzulegen bzw. zu löschen (siehe Kapitel A.2). Das Löschen dieser Listenstruktur ist jedoch erst möglich wenn alle eingetragenen CAN Filter von den jeweilig geöffneten Sockets der CAN Protokolle gelöscht wurden. Die Reihenfolge des Aufrufens der Notifier Funktionen durch die Notifier-Liste ist unbestimmt. Daher markiert das CAN Basismodul die per-Interface Listenstruktur zur Löschung, wenn die Struktur zum Zeitpunkt der Bearbeitung durch das CAN Basismodul noch aktive Filter enthält. Ist die Notifier-Liste vollständig abgearbeitet, sollten auch die geöffneten Sockets der CAN Protokolle ihre Filter für das betroffene Netzwerkgerät entfernt haben. Nach Entfernung des letzten Filters wird dann die zum Löschen markierte Struktur entfernt.

Die auf dem CAN Basismodul aufsetzenden CAN Protokolle (genauer: ihre instanziierten Sockets) nutzen den Notifier NETDEV_UNREGISTER für das beschriebene Löschen der angelegten CAN Filter. Als zweiter Notifier wird allerdings auch noch NETDEV_DOWN genutzt, um einer Anwendung die Information über ein heruntergefahrenes Netzwerkgerät signalisieren zu können. So kann ein Versuch zum Lesen oder Schreiben auf einen geöffneten CAN Socket eine Fehlermeldung ENETDOWN zurückgeben und entsprechend von der Anwendung verarbeitet werden. Da das Netzwerkgerät auch wieder hochgefahren werden kann, werden die Filterlisten bei einem NETDEV_DOWN nicht verändert.

## A.7 Anzeige der internen Datenstrukturen zur Laufzeit

Das CAN Subsystem bietet über das so genannte procfs Dateisystem im Verzeichnis `/proc/net/can` einen Einblick in seine internen Datenstrukturen und Statistiken. Die Informationen können vom Benutzer beispielsweise mit

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

abgefragt werden und dienen als Unterstützung zur Fehlersuche sowie zur Bewertung von Performanz und Lastzuständen des CAN Subsystems. Die Schnittstelle zu den internen Datenstrukturen wurden im Rahmen der Realisierung des CAN Subsystems implementiert.

Im Folgenden werden die einzelnen Einträge erläutert.

```
user@host:~> ls -l /proc/net/can/
total 0
-rw-r--r-- 1 root root 0 30. Jul 14:24 rcvlist_all
-rw-r--r-- 1 root root 0 30. Jul 14:24 rcvlist_eff
-rw-r--r-- 1 root root 0 30. Jul 14:24 rcvlist_err
-rw-r--r-- 1 root root 0 30. Jul 14:24 rcvlist_fil
-rw-r--r-- 1 root root 0 30. Jul 14:24 rcvlist_inv
-rw-r--r-- 1 root root 0 30. Jul 14:24 rcvlist_sff
-rw-r--r-- 1 root root 0 30. Jul 14:24 reset_stats
-rw-r--r-- 1 root root 0 30. Jul 14:24 stats
-rw-r--r-- 1 root root 0 30. Jul 14:24 version
```

Interne Datenstrukturen der PF_CAN Infrastruktur:

**stats** Statistiken - Beschreibung ab Seite 149

**reset_stats** Zurücksetzen von Statistiken - Beschreibung ab Seite 150

**rcvlist_xxx** Empfangslisten für Akzeptanzfilter - Beschreibung ab Seite 152

**version** Softwareversion des CAN Subsystem - Beschreibung ab Seite 154

Zusätzlich besteht die Möglichkeit den Status und die Filterquote für die inhaltsbasierten Filter des *Broadcast-Managers* (siehe Kapitel 4.3.7.2 und Kapitel B.3) auszulesen. Die Beschreibung der *Broadcast-Manager* Filterlisten findet sich ab Seite 155.

## Statistiken /proc/net/can/stats

Über die angebotenen Statistiken kann man sich über das aktuelle CAN Datenaufkommen informieren oder sich beispielsweise das Verhältnis der von Applikationen subskribierten CAN Nachrichten zu der Anzahl aller vom CAN Bus empfangenen Nachrichten anzeigen lassen.

Die Informationen werden als Grundeinstellung mit dem Start des CAN Subsystems jede Sekunde aktualisiert. Werden die Statistiken nicht benötigt, besteht die Möglichkeit beim Laden des CAN Basismodules `can.ko` den Parameter `stats_timer=0` anzugeben.

Die statistischen Angaben schließen auf jeder Zeile mit einer in Klammern eingefassten eindeutigen Markierung ab - beispielsweise mit (RXMR). Mit dieser eindeutigen Markierung soll eine automatisierte, textuelle Verarbeitung der Daten ermöglicht werden, die in einer Zeile jeweils mit einem numerischen Wert beginnen und mit der eindeutigen Markierung enden.

```
user@host:~> cat /proc/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)
```

## Zurücksetzen von Statistiken /proc/net/can/reset_stats

Das Zurücksetzen der statistischen Informationen kann durch interne Überläufe von Zählern oder vom Benutzer selbst initiiert werden. Über das intern initiierte Zurücksetzen der statistischen Informationen informiert eine zusätzliche Zeile (STR). Das vom Benutzer initiierte Zurücksetzen der statistischen Informationen wird in einer zusätzlichen Zeile (USTR) gezählt. An diesem Beispiel sind die Auswirkungen bezogen auf die obige Ausgabe der Statistiken in einem laufenden System gut zu erkennen:

```
user@host:~> cat /proc/net/can/reset_stats
Scheduled statistic reset #1.
user@host:~> cat /proc/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)
       1 user statistic resets (USTR)
```

Beim Zurücksetzen der Statistik werden die Werte für alle CAN Frame bezogenen Statistiken gelöscht.

## Zusammenfassung der Statistikelemente

TXF (transmitted frames)
Gesamtanzahl gesendeter CAN Frames

RXF (received frames)
Gesamtanzahl empfangener CAN Frames

RXMF (matched frames)
Gesamtanzahl an CAN Protokolle/Sockets weitergeleiteter CAN Frames

RXMR (total match ratio in %)
Verhältnis insgesamt weitergeleiteter zu empfangenen CAN Frames in %

TXR (total tx rate in frames/s)
Gesamt Framerate für gesendetete CAN Frames in Frames/s

RXR (total rx rate in frames/s)
Gesamt Framerate für empfangene CAN Frames in Frames/s

CRXMR (current match ratio in %)
Verhältnis in der letzten Sekunde weitergeleiteter zu empfangenen Frames in %

CTXR (current tx rate in frames/s)
Framerate in der letzten Sekunde für gesendetete CAN Frames in Frames/s

CRXR (current rx rate in frames/s)
Framerate in der letzten Sekunde für empfangene CAN Frames in Frames/s

MRXMR (max match ratio)
Maximalwert des Verhältnisses weitergeleiteter zu empfangenen Frames in %

MTXR (max tx rate in frames/s)
Maximalwert der Framerate für gesendetete CAN Frames in Frames/s

MRXR (max rx rate in frames/s)
Maximalwert der Framerate für empfangene CAN Frames in Frames/s

CRCV (current receive list entries)
Aktuelle Anzahl im CAN Basismodul eingetragener CAN Filter

MRCV (maximum receive list entries)
Maximalwert der Anzahl im CAN Basismodul eingetragener CAN Filter

STR (statistic resets)
Anzahl der durchgeführten Zurücksetzungen der Statistik Werte

USTR (user statistic resets)
Anzahl der davon durch den Anwender initiierten Zurücksetzungen der Statistik

## Filterstrukturen /proc/net/can/rcvlist_*

CAN Subsystem-Module können sich beim CAN Basismodul für den Empfang von einzelnen CAN-IDs (oder Bereichen von CAN-IDs) von bestimmten CAN-Netzwerkgeräten registrieren. Diese Registrierung führt zu einem Filtereintrag in einer zugehörigen Empfangsliste, bei der zu jedem registrierten CAN Filter 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.

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

**rcvlist_all** In dieser Liste sind CAN Filter eingetragen, die von einem CAN Bus alle empfangenen CAN Frames benötigen. Beispielsweise sind dieses die so genannten RAW-Sockets mit ihrem Filter in der Grundeinstellung (siehe Kapitel B.2).

**rcvlist_fil** In dieser Liste sind CAN Filter eingetragen, die nur einen über Bitmasken definierten Bereich von CAN Frames benötigen (z.B. 0x200 - 0x2FF). Zusätzlich werden in dies Liste Filter für RTR CAN Frames eingetragen.

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

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

**rcvlist_sff** In dieser Liste sind CAN Filter 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. Details siehe Kapitel 4.3.5

Zum Anzeigen aller Empfangslisten, kann man zur Vereinfachung folgendes eingeben:

```
user@host:~> cat /proc/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
  (vcan5: no entry)
  (any: no entry)


receive list 'rx_eff':
  (can1: no entry)
  (can0: no entry)
  (vcan5: no entry)
  (any: no entry)


receive list 'rx_err':
  (can1: no entry)
  (can0: no entry)
  (vcan5: no entry)
  (any: no entry)


receive list 'rx_fil':
  device   can_id   can_mask  function  userdata   matches  ident
   can1      200    80000700  f8c995ac  f0e5b380         0  raw
  (can0: no entry)
  (vcan5: no entry)
  (any: no entry)


receive list 'rx_inv':
  (can1: no entry)
  (can0: no entry)
  (vcan5: no entry)
  (any: no entry)


receive list 'rx_sff':
```

```
  (can1: no entry)
  device   can_id   can_mask  function  userdata   matches  ident
   can0      123    c00007ff  f8c86bec  e2e14380        29  bcm
   can0      456    c00007ff  f8c86bec  ea954880         0  bcm
   can0      789    c00007ff  f8c86bec  e30e6200       130  bcm
   can0      3FF    c00007ff  f8c86bec  deaf2580        14  bcm
   can0      740    c00007ff  f8c93680  e48322c4       178  tp20
  (vcan5: no entry)
  (any: no entry)
```

Um eine einzelne Liste anzuzeigen:

```
user@host:~> cat /proc/net/can/rcvlist_sff

receive list 'rx_sff':
  (can1: no entry)
  device   can_id   can_mask  function  userdata   matches  ident
   can0      123    c00007ff  f8c86bec  e2e14380        29  bcm
   can0      456    c00007ff  f8c86bec  ea954880         0  bcm
   can0      789    c00007ff  f8c86bec  e30e6200       130  bcm
   can0      3FF    c00007ff  f8c86bec  deaf2580        14  bcm
   can0      740    c00007ff  f8c93680  e48322c4       178  tp20
  (vcan5: no entry)
  (any: no entry)
```

## Versionsinformation /proc/net/can/version

Die CAN Subsystem Versionsinformationen können für eine Anwendung z.B. durch das Auslesen der Datei `/proc/net/can/version` abgefragt werden.

```
user@host:~> cat /proc/net/can/version
rev 20090105 abi 8
```

### Inhaltsfilter /proc/net/can-bcm/<socket>

Der *Broadcast-Manager* (siehe Kapitel 4.3.7.2 und Kapitel B.3) realisiert einen Filter, der auf Änderungen des Nutzdateninhaltes vom CAN Frames konfigurierbar ist. Jeder geöffnete **BCM**-Socket kann mit beliebig vielen Filtern konfiguriert werden, die gemäß ihrer Konfiguration CAN Nachrichten an den Anwender weiterreichen oder nur intern eine aktualisierte Version der empfangenen Daten ablegen.

Für einen geöffneten Socket legt der *Broadcast-Manager* einen Dateieintrag mit der (eindeutigen) Adresse der Socketstruktur im Verzeichnis `/proc/net/can-bcm` an. Der Status der einzelnen konfigurierten Filter kann durch das Auslesen dieser Datei angezeigt werden, wobei bei Empfangsfiltern nur Filter angezeigt werden, die mindestens ein CAN Frame empfangen haben:

```
user@host:~> cat /proc/net/can-bcm/eef3f900
>>> socket eef3f900 / sk ee93f800 / bo ee93f800 / dropped 0 / bound vcan1 <<<
rx_op: 123 vcan1 [1]d # recv 1981 (2323) => reduction: 15%
rx_op: 456 vcan1 [1]d # recv 232 (232) => reduction: 0%
rx_op: 789 vcan1 [1]d # recv 1 (464) => reduction: near 100%
rx_op: 321 vcan1 [1]d # recv 22 (464) => reduction: 96%
rx_op: 654 vcan1 [1]d # recv 237 (461) => reduction: 49%
rx_op: 234 vcan1 [1]d # recv 53 (233) => reduction: 78%
rx_op: 567 vcan1 [1]d # recv 2177 (2325) => reduction: 7%
rx_op: 232 vcan1 [1]d # recv 8 (2277) => reduction: near 100%
tx_op: 042 vcan1 [1] t1=1000000 # sent 6
```

Die Ausgaben sind dabei wie folgt zu interpretieren:

Titelzeile
- **socket / sk / bo** Adressen betriebssysteminterner Strukturen.
- **dropped** Anzahl verworfener Updates, die dem Anwendungsprogramm aufgrund eines vollen Socket Empfangspuffers nicht zugestellt werden konnten.
- **bound** CAN Netzwerkgerätename an den der Socket gebunden ist (auch: 'any').

rx_op
- **can_id** CAN Identifier für diesen Empfangsfilter
- **interface** CAN Netzwerkgerätename für diesen Empfangsfilter
- **nframes** 1 ⇒ Standard Filter, 2..n ⇒ Multiplex Filter
- **opts** d: RX_CHECK_DLC, timeo: Empfangstimeout in $\mu$s, thr: Throttle in $\mu$s.
- **# stats** recv <weitergeleitete Frames> (<empfangene Frames>) Reduktion in %

tx_op
- **can_id** CAN Identifier für diesen Sendeauftrag
- **interface** CAN Netzwerkgerätename für diesen Sendeauftrag
- **nframes** Anzahl zirkulär gesendeter CAN Frames
- **opts** t1: Zykluszeit in $\mu$s, t2: Zykluszeit in $\mu$s.
- **# stats** sent <Anzahl gesendeter CAN Frames>

## A.8 Migration existierender CAN Anwendungen

Wie in Kapitel 3 beschrieben, stellt der Wechsel auf eine andere Programmierschnittstelle immer ein technisches und finanzielles Risiko dar. In diesem Abschnitt soll beschrieben werden, wie eine existierende CAN Anwendung auf das Netzwerk Konzept migriert werden kann und welche Änderungen sich dabei für den Anwendungsentwickler ergeben.

### A.8.1 Anpassung existierender CAN Anwendungen

In Kapitel 3.4 wurde der Zugriff auf den CAN Bus nach dem Stand der Technik über die zeichenorientierte Treiberschnittstelle anhand von vier unterschiedlichen Beispielen dargestellt.

Das Beispiel für das Zugriffsverfahren 1b (direkter Zugriff auf die Gerätedatei mit binären Strukturen) wird von den CAN Treibern CAN4LINUX und LinCAN mit einer Struktur für das CAN Frame realisiert, die auf den gemeinsamen Ursprung der beiden CAN Treiber (LDDK Projekt von Klausi Schröter, FU Berlin) zurückgeht:

```
/**
* The CAN message structure.
* Used for all data transfers between the application and the driver
* using read() or write().
*/
typedef struct {
    /** flags, indicating or controlling special message properties */
    int             flags;
    int             cob;         /**< CAN object number, used in Full CAN  */
    unsigned   long id;          /**< CAN message ID, 4 bytes  */
    struct timeval  timestamp;   /**< time stamp for received messages */
    short      int  length;      /**< number of bytes in the CAN message */
    unsigned   char data[CAN_MSG_LENGTH]; /**< data, 0...8 bytes */
} canmsg_t;
```

Die Struktur ist auf einem 32-Bit Linux 4+4+4+8+2+8 = 30 Byte lang und enthält neben den Informationen, die eine CAN Botschaft beschreiben zusätzlich

**flags** ein Bitfeld für EFF, RTR sowie CAN Controller Stati wie BUS-OFF

**cob** Ein Wert für das Controller spezifische CAN Objekt (siehe Mailboxes im i82527[21])

**timestamp** Einen Zeitstempel für den Empfangszeitpunkt des CAN Frames

Exakt diese zusätzlichen und Controller spezifischen Informationen wurden bei der Definition des CAN Frames für das CAN Subsystem bewusst vermieden (siehe Begründung in Kapitel 4.3.2.1 (Die Datenstruktur des CAN Frames)). Die für die Arbitrierung relevanten Bitwerte für EFF und RTR wurden in den CAN Identifier übernommen und führen zu einem 16 Byte langen CAN Frame in dem 3 Bytes ungenutzt sind:

```
/**
 * struct can_frame - basic CAN frame structure
 * @can_id:  the CAN ID of the frame and CAN_*_FLAG flags, see above.
 * @can_dlc: the data length field of the CAN frame
 * @data:    the CAN frame payload.
 */
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)));
};
```

Für das Versenden von CAN Nachrichten ergeben sich für den Programmierer nur geringe Unterschiede, die die Anpassung des vorhandenen Quelltextes erforderlich machen:

| CAN4LINUX | CAN Subsystem |
|---|---|
| `canmsg_t tx;` | `struct can_frame tx;` |
| | |
| `tx.id = 0x123;` | `tx.can_id = 0x123;` |
| `tx.flags = 0;` | |
| `tx.length = 2;` | `tx.can_dlc = 2;` |
| `tx.data[0] = 0x11;` | `tx.data[0] = 0x11;` |
| `tx.data[1] = 0x22;` | `tx.data[1] = 0x22;` |
| | |
| `write(fd, tx, sizeof(tx));` | `write(fd, tx, sizeof(tx));` |

Beim Aussenden einer RTR Botschaft:

| CAN4LINUX | CAN Subsystem |
|---|---|
| `canmsg_t tx;` | `struct can_frame tx;` |
| | |
| `tx.id = 0x123;` | `tx.can_id = 0x123 \| CAN_RTR_FLAG;` |
| `tx.flags = MSG_RTR;` | |
| `tx.length = 0;` | `tx.can_dlc = 0;` |
| | |
| `write(fd, tx, sizeof(tx));` | `write(fd, tx, sizeof(tx));` |

Fazit:

Die potenziellen Änderungen im Quelltext beim Senden von CAN Botschaften können als minimal betrachtet werden.

Beim Empfang von CAN Botschaften kann über die entsprechenden Filtereinstellungen der Zugriff auf CAN Botschaften und Error Messages konfiguriert werden. Die mögliche Trennung in der Form einer Aufteilung dieser Nachrichtenarten auf zwei unterschiedliche Socket-Instanzen kann zu einer Vereinfachung des Quelltextes führen. Lediglich der optionale Zugriff auf den Zeitstempel erfordert eine zusätzliche Anpassung des Quelltextes (siehe Kapitel B.1.6).

### A.8.2 Weiterverwendung existierender closed-source CAN Anwendungen

Ist eine CAN Anwendung nur in binärer Form vorhanden, weil es sich beispielsweise um ein kommerzielles Produkt oder eine kommerzielle CAN Bibliothek handelt, ist die Anpassung der CAN Anwendung wie im vorherigen Beispiel nicht möglich.

Um die binären CAN Anwendungen unter Linux betreiben zu können, müssen die von der Anwendung vorausgesetzten Schnittstellen zum Betriebssystem vorhanden sein. Eine Bereitstellung dieser vorausgesetzten Schnittstellen könnte beispielsweise durch eine Emulation der vorausgesetzten Schnittstellen erfolgen, die ihrerseits auf die geschaffenen Möglichkeiten des CAN Zugriffs über Netzwerkschnittstellen aufsetzt.

Die Emulation einer Betriebssystemschnittstelle ist abhängig von der Qualität der Dokumentation der Schnittstelle und ihrer korrekten Realisierung. Nutzt beispielsweise ein Anwendungsprogramm einen Fehler oder ein nicht spezifiziertes Verhalten der Schnittstelle aus, ist dieses in der Emulation zu berücksichtigen, was wiederum einen erheblichen Entwicklungs- und Wartungsaufwand nach sich ziehen kann.

Eine im Rahmen dieser Arbeit erfolgreich realisierte Emulation einer CAN Betriebssystemschnittstelle ist in Abbildung A.1 dargestellt.

Die Emulation eines CAN Adapters, der von der CAN Anwendung über die serielle Schnittstelle angesprochen wird, wurde mit einer in Unix-Systemen vorhandenen Betriebssystemschnittstelle des "Pseudo-TTY" realisiert. Pseudo-TTYs können beispielsweise eine "serielle Schnittstelle" für Terminal-Programme emulieren.

Das Werkzeug **slcanpty** (siehe Kapitel C (Werkzeuge)) erzeugt ein Pseudo Terminal für ein vorhandenes CAN Anwendungsprogramm, dass eine serielle CAN Hardware mit dem SLCAN/LAWICEL Kommunikationsprotokoll [13] erwartet. Dabei wird das erzeugte Pseudo Terminal einem bestimmten CAN Netzwerkgerät zugewiesen, auf das die von

der Anwendung gesendeten CAN Daten gesendet werden bzw. dessen empfangene CAN Frames an das SLCAN Anwendungsprogramm weitergeleitet werden.

```
user@host:~> slcanpty /dev/ptyc0 can0
```

Im gezeigten Beispiel wird für das Anwendungsprogramm eine serielle Schnittstelle `/dev/ttyc0` erzeugt und diese mit der CAN Netzwerkschnittstelle `can0` verbunden.

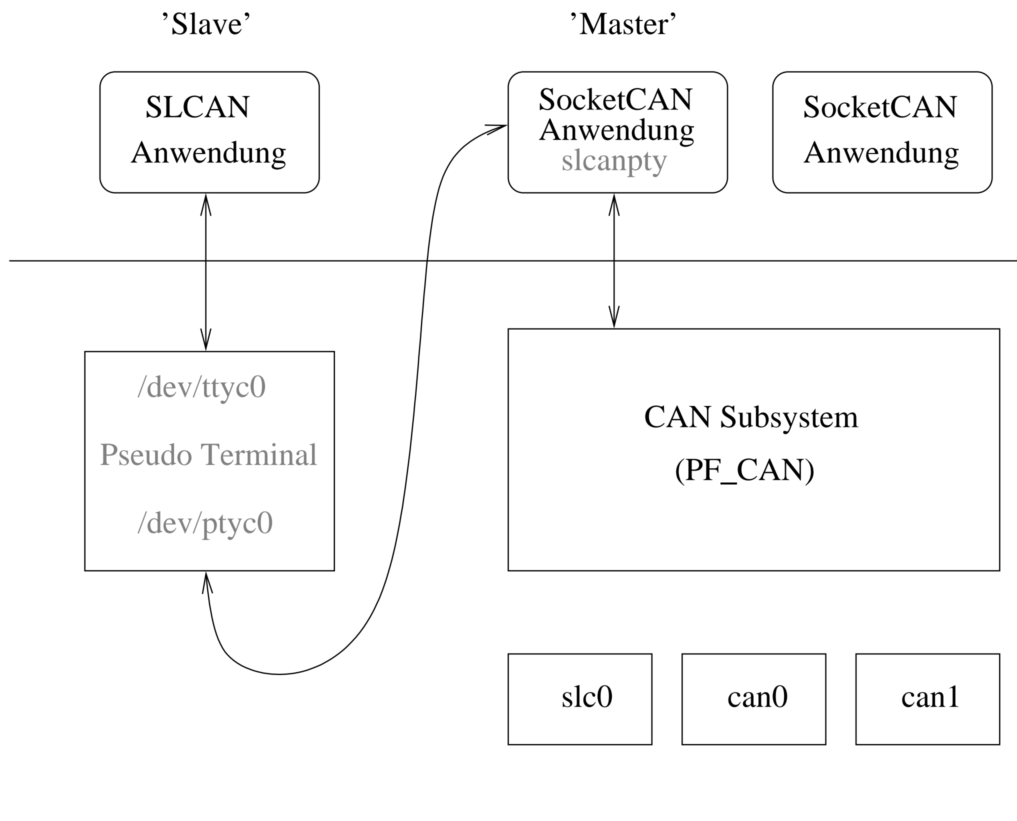



Abbildung A.1: slcanpty als Emulation einer seriellen CAN Hardware

Die beschriebene **slcanpty** Emulation wurde mit der Demo-Version des kommerziellen Werkzeuges "CANreport" der Firma Port erfolgreich erprobt.

Die Einschränkung der Demo-Version auf eine CAN Bitrate von 125kBit/s konnte von der realisierten **slcanpty** Emulation allerdings nicht abgebildet werden, weil diese die Einstellung der Bitrate des CAN Netzwerktreibers nicht implementiert. Aufgrund der Tatsache, dass praktisch beliebig viele Pseudo Terminals auf diese Weise eingerichtet werden können, besteht die Möglichkeit auch mehrere SLCAN Anwendungsprogramme auf gleichen oder verschiedenen CAN Bussen parallel arbeiten zu lassen.

Für die Emulation einer zeichenorientierten CAN Treiberschnittstelle, wie sie in Kapitel 3.4 (CAN Programmierschnittstellen unter Linux) beschrieben wurde, ist die Realisierung eines Kernel-Treibers notwendig. Ein solcher Treiber könnte beispielsweise aus dem ursprünglichen zeichenorientierten Treiber abgeleitet werden und auf diese Weise die bereitgestellte Schnittstelle zur CAN Anwendung beibehalten.

Für das Senden und Empfangen von CAN Botschaften müsste ein solcher Treiber innerhalb des Linux Kernel auf die Schnittstellen des CAN Subsystems zugreifen, die üblicherweise von den CAN Protokollen genutzt werden:

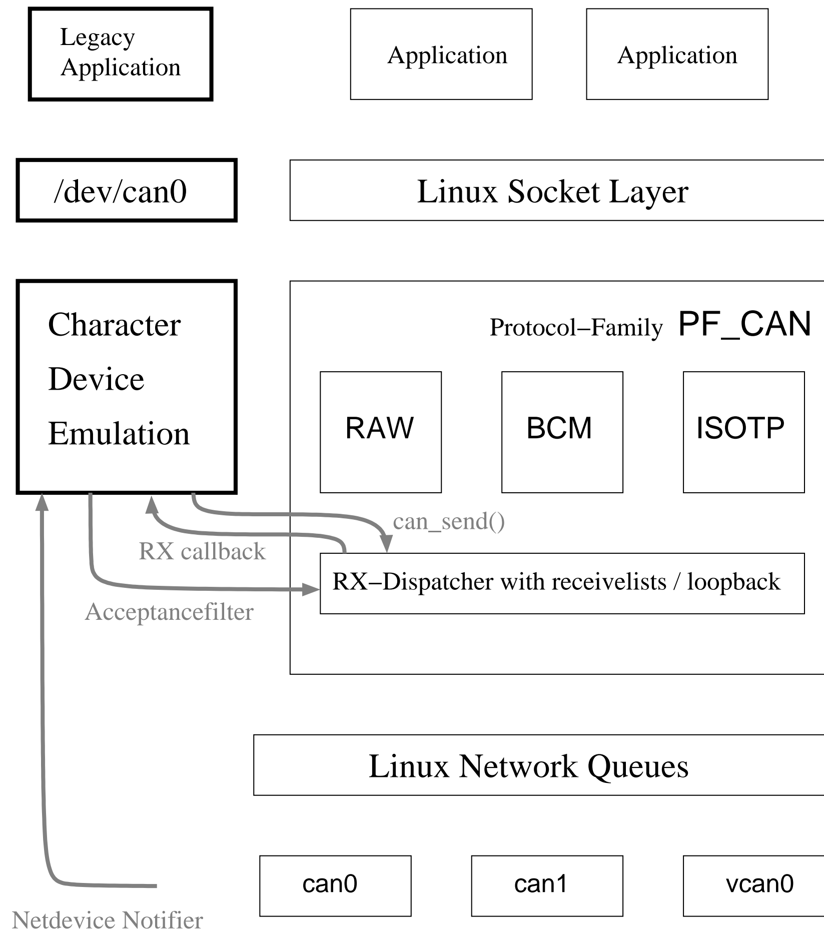



Abbildung A.2: Emulation eines zeichenorientierten CAN Treibers

Für eine solche Emulation müssten beispielsweise auch die Statusänderungen der Netzwerkgeräte verfolgt werden, um das Entfernen der gesetzten Akzeptanzfilter sicherstellen zu können (siehe Kapitel A.6). Zusätzlich muss eine Konfigurationsmöglichkeit realisiert werden, die der Emulation das entsprechend zu nutzende CAN Netzwerkgerät (bspw. 'can0') zuweist.

## A.9 Konzepte für zukünftige Entwicklungen

### A.9.1 Nutzung aktiver CAN Hardware zur Nachrichtenfilterung

Im Kapitel 4.3.5 (Filtern und Verteilen empfangener CAN Nachrichten) wurde auf die potenziellen Nachteile einer Filterung im Software-Interrupt eingegangen und warum die Verwendung von Controller-spezifischen Eigenschaften keine Lösung für die Anforderungen für Mehrbenutzerbetriebssystemen darstellt.

Die realisierten Filter (siehe Kapitel A.2) bieten die für eine Verwendung in einem Mehrbenutzersystem nötige Flexibilität, wobei der CAN Netzwerktreiber allerdings ohne Filter betrieben werden muss.

Eine Filterung auf der Ebene des CAN Netzwerkgerätes wird dann interessant, wenn nur wenige einzelne CAN Identifier oder bestimmte Bereiche von CAN Identifiern benötigt werden. In diesem Fall würde die Anzahl der von der (*aktiven*) CAN Hardware in das System übertragenen CAN Nachrichten reduzierbar sein.

Wenn ein Anwender beispielsweise einen RAW-Socket mit dem voreingestellten Filter (= ohne Filterung) öffnet, muss das CAN Netzwerkgerät alle empfangenen CAN Nachrichten an das System weiterleiten. Bei der Verwendung des Interface Index '0' (also aller CAN Netzwerkgeräte) müsste jedes CAN Netzwerkgerät alle empfangenen CAN Nachrichten an das System weiterleiten. Ein Filter auf Treiberebene wäre obsolet.

Die Nutzung *aktiver* CAN Hardware zur Filterung von CAN Nachrichten ist daher nur für spezielle Anwendungsszenarien sinnvoll. Für eine Beibehaltung der aktuell zur Verfügung gestellten Filterfunktionalitäten ist für jede Änderung an den Filtern im CAN Subsystem der entsprechende Filter zusätzlich an den CAN Netzwerktreiber weiter zu leiten. Dieses kann beispielsweise durch eine Funktion erfolgen, die vom Netzwerktreiber zur Verfügung gestellt wird.

Allerdings ist dabei die Performanz dieser zur Verfügung gestellten Treiberfunktion von entscheidener Bedeutung, um die Effizienz der verwendeten RCU[4]-Listen zu erhalten.

Bei eingehender Betrachtung einer Filterung von CAN Nachrichten auf Treiberebene unter Beibehaltung der beschriebenen (Mehrbenutzer-)Filterfunktionen, kann von einer derartigen Realisierung nur abgeraten werden.

Für den Einsatz in einem konkreten Anwendungsfall ist eine Filterung von CAN Nachrichten allerdings in der Form eines systemweiten Filters auf der Treiberebene vorstellbar. Dabei könnte ein entsprechend autorisierter Anwender (z.B. der Administrator des

[4]Read-Copy-Update: Ein Zugriffsverfahren für das effiziente Ein- und Aushängen von Elementen aus einer verketteten Liste

Systems) die Filterliste für ein CAN Netzwerkgerät über eine neu zu schaffende Programmierschnittstelle systemweit festlegen. Die CAN Anwendungsprogramme können einen derart definierten Filter zwar für sich zusätzlich einschränken, eine Ausweitung des Filters durch die CAN Anwendung beispielsweise durch die Verwendung des vordefinierten Filters (s.o.) könnte dabei allerdings nicht umgesetzt werden.

Eine solche durch den Systemadministrator konfigurierbare Filterung auf Treiberebene ist grundsätzlich nicht auf die Verwendung einer *aktiven* CAN Hardware beschränkt. Allerdings ist die Verarbeitung von Filtern im Unterbrechungskontext als kritisch zu bewerten. Eine solche Akzeptanzfilterung im Unterbrechungskontext ist in der Folge mit einer Komplexität von $\mathcal{O}(1)$ zu realisieren, um Laufzeitabhängigkeiten in Bezug auf die Anzahl der verwendeten Filter ausschließen zu können.

Ein Beispiel für einen Akzeptanzfilter mit einer Komplexität von $\mathcal{O}(1)$ stellt ein konfigurierbares Bitfeld dar, bei dem durch zwei Bit-schiebe Operationen und einen indexierten Speicherzugriff ermittelt werden kann, ob das einer CAN ID zugehörige Bit gesetzt ist und somit diese CAN ID weitergeleitet werden soll.

Die systemweite Filterung im CAN Treiber bietet analog einer so genannten Firewall[5] die Möglichkeit, einen CAN Teilnehmer und die zugehörigen CAN Anwendungen in ihren Kommunikationsmöglichkeiten sende- und empfangsseitig zu beschränken.

### A.9.2 Hash Verfahren für EFF Nachrichtenfilter

Im Kapitel A.2 (Filterkonzept im Software-Interrupt) wurde die Realisierung der verschiedenen Filterlisten für die jeweiligen CAN Netzwerkgeräte erläutert.

Zur Reduzierung von Vergleichsoperationen zur Laufzeit werden die von CAN Anwendungsprogrammen definierten Filter möglichst in spezifischen Filterlisten angeordnet. Für die Filterung nach einzelnen 11-Bit CAN Identifiern von SFF Frames existiert zu diesem Zweck ein Feld von 2048 Filterlisten.

Die Filterung nach einzelnen EFF Frames mit einen 29-Bit CAN Identifier wird durch eine lineare Liste von Filterelementen realisiert. Aufgrund eines fehlenden realen Anwendungsfalles für die Filterung nach einzelnen EFF Frames wurde bis zum aktuellen Zeitpunkt keine performantere Lösung implementiert.

Denkbar wäre für diese Liste eine Hash-Funktion, welche die einzelnen zu filternden 29-Bit CAN Identifier beispielsweise auf ein Feld von n Filterlisten abbildet. Da über eine zu erwartende Verteilung der CAN Identifier keine Aussage gemacht werden kann und die Funktion beim Empfang jedes EFF Frames ausgeführt werden muss, ist eine besonders effiziente Hash-Funktion mit einer Komplexität von $\mathcal{O}(1)$ zu wählen.

[5]engl. Brandmauer: Ein Verfahren zum Filtern von Adressen und Diensten beim Internetprotokoll

Erfahrungsgemäß unterscheiden sich bei 29-Bit Identifiern die relevanten (unterschiedlichen) Bereiche innerhalb der CAN ID in Abhängigkeit vom Anwendungsfall. So werden beispielsweise die Bits 25-29 oder 0-7 zur Unterscheidung von Dateninhalten verwendet. Da die übrigen Bitwerte des CAN Identifiers dabei konstant bleiben, könnte sich eine Exklusiv-ODER Verknüpfung (Symbol ⊕) für eine Abbildung des 29-Bit Identifiers auf einen Feldindex mit geringerer Bitlänge als potenziell effiziente Hash-Funktion herausstellen. Die Funktionalität einer solchen Hash-Funktion kann binär wie im folgenden Beispiel dargestellt werden:

```
 1 1010 1100 0111 0101 0110 1010 1100         (29-Bit CAN-ID: 0x1AC756AC)

01 1010 1100 0111 0101 0110 1010 1100         (10 Bit High)
01 1010 1100 0111 0101 0110 1010 1100         (10 Bit Mid)
01 1010 1100 0111 0101 0110 1010 1100         (10 Bit Low)


  0110101100         (10 Bit High)
⊕ 0111010101         (10 Bit Mid)
⊕ 1010101100         (10 Bit Low)

  1011010101         (0x2D5 : 10 Bit Hashwert der CAN-ID 0x1AC756AC)
```

Analog zu der indexierten Filterliste für 11-Bit CAN Identifier in Tabelle A.1 auf Seite 140 wird eine Filterliste mit 1024 Elementen ($2^{10}$) für 29-Bit Identifier in die netzwerkgerätespezifischen Empfangslisten eingefügt.

Im Gegensatz zur indexierten Filterliste für 11-Bit CAN Identifier muss bei der über den Hash-Wert indexierten Filterliste für 29-Bit CAN Identifier die Übereinstimmung der empfangenen ID zu der ID in den Filterelementen weiterhin geprüft werden. Der Unterschied zur aktuellen Implementierung des CAN Subsystems besteht in der potenziell geringeren Anzahl von zu prüfenden Filterelementen, wenn die Hash-Funktion eine möglichst gleichverteilte Anzahl von Filterelementen in die 1024 Filterlisten ermöglicht.

Zur Bewertung der Gleichverteilung der CAN Identifier in den Filterlisten für EFF Nachrichtenfilter wurde ein Programm realisiert, welches die Hash-Werte für CAN Identifier mit verschiedenen Wertabständen (engl. Offsets) für alle CAN Identifier berechnet hat. Der folgende Quelltext enthält den relevanten Ausschitt dieses Programms zur versuchsweisen Verifikation der Hash-Funktion:

```
unsigned int effhash(unsigned int can_id)
{
        unsigned int ret;

        ret  = (can_id>>20) & 0x3FF;
        ret ^= (can_id>>10) & 0x3FF;
        ret ^=  can_id      & 0x3FF;

        return ret;
}

(..)
        for (i=0; i < entries; i++) {
                hash = effhash(can_id);
                can_id += offset;
                can_id &= 0x1FFFFFFF;
                t[hash]++;
        }
(..)
```

Die gewählte Hash-Funktion `effhash()` hat in Bezug auf die beschriebenen Anwendungsfälle in verschiedenen Testläufen erwartungsgemäß eine gleichverteilte Anordung der Filterelemente in der indexierten Filterliste erzeugt.

Aufgrund des zusätzlichen benötigten Hauptspeichers für jedes CAN Netzwerkgerät, sollte bei einer Implementierung eine Konfigurierbarkeit dieser Funktionalität zur Kompilierungszeit vorgesehen werden.

### A.9.3 Identifierbasierte Priorisierung von CAN Sendewarteschlangen

Mit der Möglichkeit, den CAN Bus parallel durch unterschiedliche Anwendungen nutzen zu können, ergeben sich neue Fragestellungen zur Sicherstellung eines 'fairen' Zugangs zum CAN Bus in Mehrbenutzerbetriebssystemen. Nach dem bisherigen Stand der Technik bestand eine mehr oder weniger direkte Beziehung zwischen der CAN Anwendung und dem CAN Controller. Mit der Möglichkeit, verschiedene CAN Protokolle und deren zeitliche Anforderungen gleichzeitig auf einem System zu verarbeiten, wird ein allgemeingültiges Verfahren zur Vergabe der CAN Bus Ressourcen erforderlich.

Die in Kapitel 2.3 dargestellten CAN Anwendungen definieren unterschiedliche Anforderungen an den Datendurchsatz und das zeitliche Antwortverhalten. Im Folgenden wird eine mögliche Adaption von exisitierenden Netzwerkwarteschlangen an die Belange des Controller Area Networks diskutiert, was sich als ein zukünftiges Thema für eine wissenschaftliche Arbeit auf der Basis dieser Dissertation anbietet.

Verschiedene CAN Controller, wie beispielsweise der Controller im Atmel AT90CAN128, verfügen über mehrere so genannte Message Objekte zum Senden von CAN Nachrichten. Ist mehr als ein Message Objekt (MOb) mit ausgehenden CAN Frames belegt, stellt sich die Frage nach der Aussendungsreihenfolge der zu sendenden CAN Frames.

In der Spezifikation des AT90CAN128[1] steht dazu:

> *Then, the CAN channel scans all the MObs in Tx configuration, finds the MOb having the highest priority and tries to send it.*

Für die Verarbeitung der CAN Botschaften ergeben sich durch die CAN Identifier-gesteuerte Priorisierung (siehe Kapitel 2.2.2) verschiedene Vor- und Nachteile:

Vorteile

+ Eine höher priorisierte CAN Botschaft wird schneller gesendet
+ Folgt dem Konzept der Arbitrierung auf dem CAN Bus (CSMA/CR)

Nachteile

– Kein vorhersagbares Sendeverhalten bei niedrig priorisierten CAN Frames
– Aussendung potenziell veralteter Informationen bei mehrfacher Verdrängung
– Kann wegen einer fehlenden Warteschlange nicht bei virtuellen CAN Netzwerkgeräten (siehe Kapitel A.5) genutzt werden

Bei der Verwendung mehrerer Message Objekte verletzt das Verhalten des AT90CAN128 die Anforderung **(A11)** Verzicht auf proprietäre Eigenschaften von CAN Hardware aus Kapitel 4.1 (Anforderungsanalyse). Stellt der Entwickler des CAN Netzwerktreibers sicher, dass beim verwendeten CAN Controller jeweils nur *ein* Message Objekt genutzt wird, kann eine eventuell vorhandene hardwarespezifische Strategie des CAN Controllers nicht zum Einsatz kommen. Die Reduzierung auf ein einzelnes Message Objekt birgt jedoch in Abhängigkeit von der verwendeten CAN Hardware das Risiko, dass der CAN Controller nicht seinen maximal möglichen Datendurchsatz beim Senden von CAN Frames erreichen kann.

Zur Sicherstellung eines 'fairen' Zugangs zum CAN Bus für verschiedene CAN Anwendungen auf einem Mehrbenutzersystem stellt sich die Frage, ob eine Sortierung von ungesendeten CAN Botschaften, beispielsweise nach der durch den CAN Identifier vorgegebenen Priorität, sinnvoll ist. Dazu wird im Folgenden ein Anwendungsszenario zum Software-Update von Steuergeräten über das ISO 15765-2 Transportprotokoll betrachtet (siehe Kapitel 5.1.4).

Um eine Kommunikation weiterer CAN Teilnehmer auf dem genutzen CAN Bus während des Software Updates sicherzustellen, wurden im referenzierten Beispiel für die Transportkanäle die CAN Identifier 0x7B0 sowie 0x746 verwendet. Die Transportkanäle sind damit in dem Bereich der 10% der CAN Identifier mit der geringsten Priorität auf dem Medium. Wird die maximal mögliche PDU-Länge von 4095 Datenbytes ohne Bestätigung durch die Gegenseite versendet, generiert der ISO-TP Protokolltreiber $4095/7 = 585$ CAN Botschaften, welche aufeinanderfolgend mit der CAN ID 0x746 in den Sendepuffer des Netzwerkgerätes geschrieben werden.

Bei einer optimal angenommenen Übertragungszeit von 1,11 Millisekunden pro CAN Frame ist das CAN Netzwerkgerät minimal 649 ms mit der Aussendung der CAN Botschaften blockiert. Eine weitere CAN Anwendung auf dem selben System, welche in einem Abstand von 100 ms aktualisierte Anwendungsdaten mit dem CAN Identifier 0x123 sendet, würde durch die 649 ms Blockierung in ihrer Funktionalität massiv beeinträchtigt. Das beschriebene Anwendungsszenario erfordert folglich eine Sortierung der ungesendeten CAN Botschaften.

Wie im Kapitel 4.3.1 (Netzwerktreibermodell für CAN Hardware Treiber) beschrieben, stellt das Linux Betriebssystem für das Senden von Netzwerkpaketen eine Infrastruktur zur Verfügung, was ein vergleichsweise einfaches Treibermodell für Netzwerktreiber ermöglicht. Die ausserhalb des Netzwerktreibers realisierte Warteschlange und das Verhalten der Warteschlange wird vom Betriebssystem bereitgestellt (siehe Abbildung 4.1 auf Seite 73).

Für die Steuerung des Datenflusses beim Internet Protokoll besteht im Linux Kernel die Möglichkeit zum 'Traffic Shaping', was frei übersetzt den '(Daten-)verkehr in Form' bringt. Mit dem Traffic Shaping können beispielsweise Peer-to-Peer Datenkommunikationsverbindungen in ihrer zur Verfügung stehenden Bandbreite begrenzt werden, um weiterhin eine performante Nutzung des WWW-Browsers oder des E-Mail Programmes auf dem System zu ermöglichen.

Das Traffic Shaping realisiert die Steuerung des Datenstromes durch so genannte Queueing Disciplines (Qdisc). Die durch die verschiedenen Queueing Disciplines realisierbaren Steuerungsmöglichkeiten orientieren sich dabei überwiegend an den Inhalten und Diensten der Internet Protokollfamilie.

Für die Realisierung einer Queueing Discipline für CAN Botschaften müsste die Qdisc auf den Datenbereich des Socket-Buffers (siehe Kapitel 4.3.2) zugreifen, um aus dem darin enthaltenen CAN Frame den CAN Identifier für eine weitere Verarbeitung zu extrahieren.

In Abbildung A.3 wird eine mögliche Funktionalität einer Queueing Discipline für das Controller Area Network gezeigt.

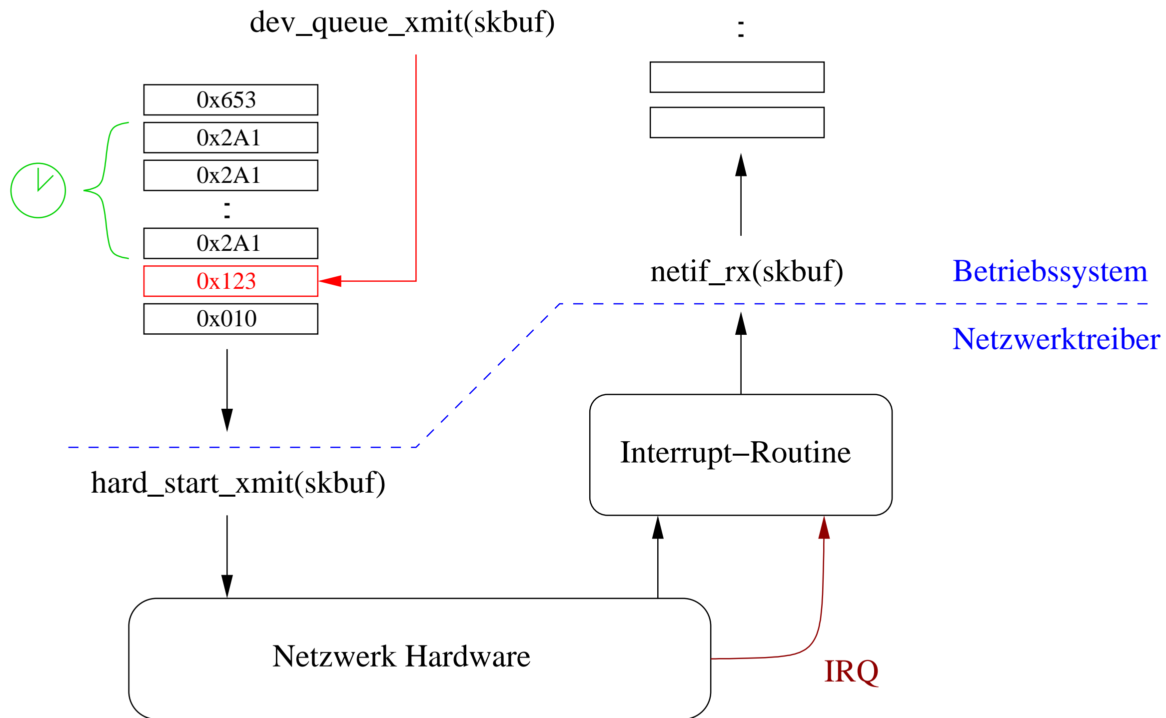



Abbildung A.3: Beispiel einer CAN Queueing Discipline

Der rote Pfeil zeigt die Einordnung des Socket-Buffers (skbuf) gemäß der Priorisierung durch den CAN Identifier. Bei der Einordnung von Socket-Buffern mit gleicher CAN ID ist darauf zu achten, dass die zeitliche Reihenfolge eingehalten wird, um eine Vertauschung in der Reihenfolge der Nachrichten auf dem CAN Bus ausschließen zu können.

Neben der standardmäßigen FIFO QDisc und dem gezeigten Verfahren zur Priorisierung gemäß dem CAN Identifier sind noch weitere Queueing Disciplines für den CAN Bus verwendbar. Das so genannte 'Stochastic Fair Queueing' (SFQ) richtet beispielsweise für verschiedene definierte Kommunikationsarten eigene Warteschlangen ein, die dann nach dem Round-Robin-Verfahren geleert werden. Im oben diskutierten Beispiel würde sich beispielsweise eine Warteschlange für die ISO-TP Kommunikation (CAN Identifier 0x746) und eine Warteschlange für die restliche ausgehende CAN Kommunikation nach dem SFQ Verfahren anbieten.

Zusammenfassend stellen die Linux Queueing Disciplines, bei entsprechender Unterstützung der CAN spezifischen Priorisierungskonzepte, eine potenzielle hardwareunabhängige Möglichkeit zur Realisierung einer CAN Traffic Shaping Lösung dar.

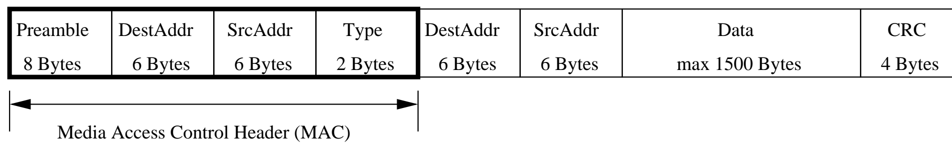
# B Die Programmierschnittstelle für das CAN Subsystem in Linux

In diesem Kapitel wird die Programmierschnittstelle für das realisierte CAN Subsystem detailliert beschrieben. Im Gegensatz zu den technischen Beschreibungen im Kapitel 4.2.2 steht hierbei die Sicht des Anwenders im Vordergrund. Teile dieses Kapitels sind nach umfangreicher Überarbeitung und Aktualisierung aus der bereits Anfang 2006 veröffentlichten Socket-API Beschreibung [18] übernommen worden. Die folgende Kurzeinleitung ist für den direkten Einstieg in die Programmierschnittstelle gedacht und setzt die Lektüre der Dissertation nicht voraus.

Wesentliche Komponenten des CAN Subsystems 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.

Die Abbildung B.1 zeigt in vereinfachter Form die Unterschiede zwischen einem Ethernet Frame und einem CAN Frame, wie sie auf ihrem jeweiligen Medium gesendet werden. Besonders markant sind die Unterschiede bei den für das Internet-Protokoll benötigten Adressen im Gegensatz zu dem 11 bzw. 29 Bit langen CAN Identifier.



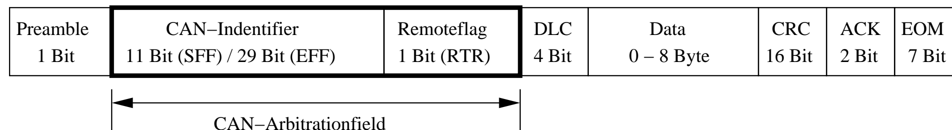




Abbildung B.1: Unterschiedliche Adressierungen bei Ethernet / CAN

Diese Unterschiede führen auch dazu, dass die Socket Adressstruktur `sockaddr_can` sich nicht ganz analog zu der bekannten `sockaddr_in` Struktur für die Internet Protokollfamilie (PF_INET) definieren lies. Durch die Bindung eines CAN Sockets an üblicherweise ein einzelnes CAN Interface, ist die gewählte Adressstruktur ähnlich der Adressstruktur für den sogenannten Packet Socket für die Protokollfamilie PF_PACKET - siehe packet(7). Der Ablauf eines Verbindungsaufbaus und die Benutzung geöffneter Sockets zum Datenaustausch sind jedoch stark an Verfahren angelehnt, wie sie von TCP/IP bekannt sind.

Neben diesem Dokument sind daher auch die aktuellen Linux 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) für das CAN Subsystem relevant. Außerdem bieten die Manual Pages ip(7), udp(7), tcp(7) und packet(7) einen Einblick in Grundlagen, auf denen auch das CAN Subsystem basiert.

Das CAN Subsystem ist neben den bekannten Protokollfamilien, wie z.B. den Protokollen der Internetprotokollfamilie (PF_INET) im Linux-Kernel integriert (siehe Abbildung B.2). Dazu wurde eine neue Protokollfamilie PF_CAN eingeführt. Durch die Realisierung der verschiedenen höheren CAN-Protokolle als Kernelmodule, können zeitliche Randbedingungen im Kernel-Kontext eingehalten werden, die auf der Anwenderschicht in dieser Form nicht realisierbar wären. Dieses ist beispielsweise für CAN Transportprotokolle relevant, die Antwortzeiten im Bereich von Millisekunden definieren.

Sollen auf einem System parallel mehrere CAN Anwendungen zur Ausführung kommen, kann jede Anwendung unabhängig beliebig viele CAN Sockets öffnen. Dabei kommt auch bei mehreren Socket-Instanzen immer derselbe Code zur Ausführung.

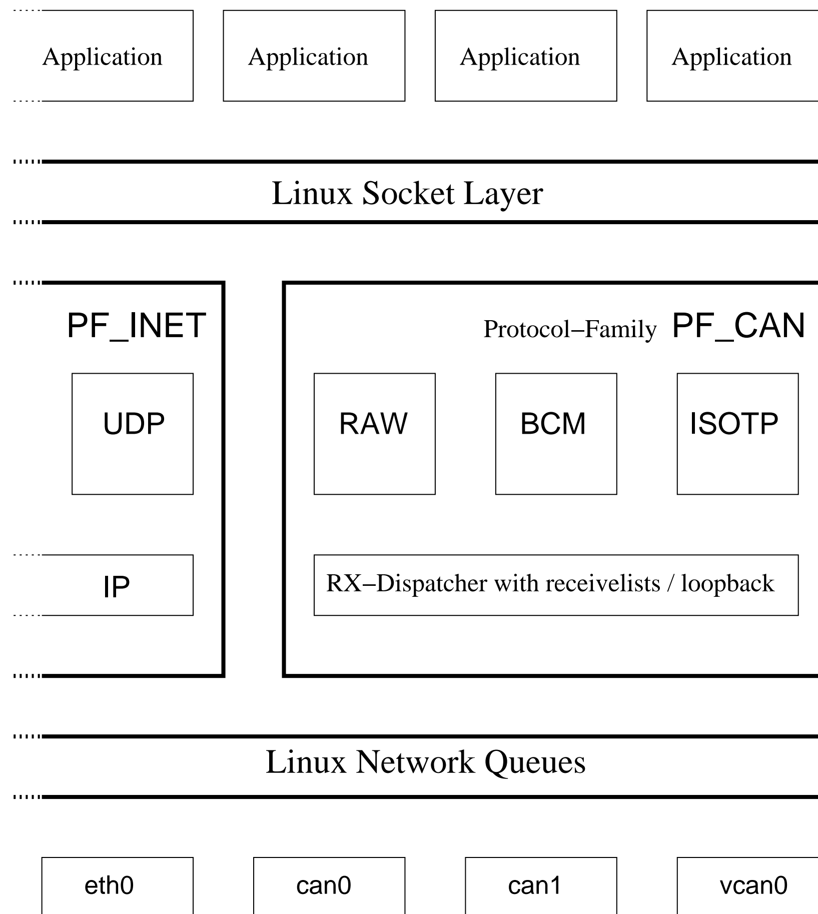



Abbildung B.2: Das CAN Subsystem im Linux-Kernel

Das CAN Subsystem 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. Der RAW-Socket entspricht am ehesten der bisherigen Sicht des CAN Anwendungsprogrammierers auf den CAN Bus - mit dem Unterschied nun mehrere voneinander unabhängige CAN Anwendungen zeitgleich starten zu können, die auch auf dem selben CAN Bus arbeiten können.

Eine Besonderheit stellt der so genannte RX-Dispatcher des CAN Subsystem dar. Durch die Art der Adressierung der CAN Frames kann es mehrere 'Interessenten' an einer empfangenen CAN-ID geben. Durch die CAN Subsystem-Funktionen rx_register() und rx_unregister() können sich die Protokollmodule beim CAN Subsystem-Kernmodul für ein oder mehrere CAN-IDs von definierten CAN-Netzwerkgeräten registrieren, die ihnen beim Empfang automatisch zugestellt werden. Das CAN Subsystem-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.

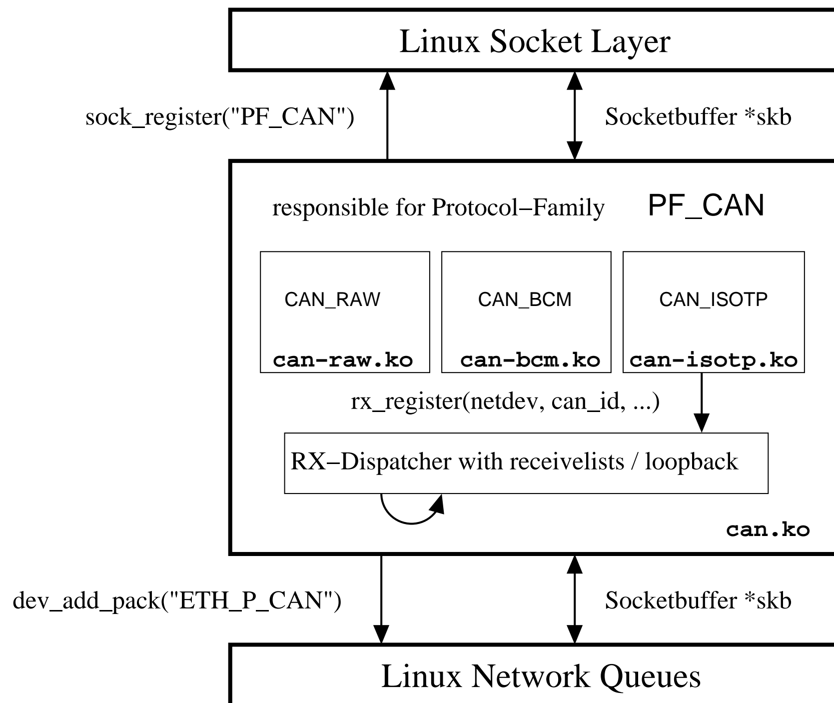



Abbildung B.3: Das CAN Subsystem-Kernmodul im Linux-Kernel

Für die Anbindung der CAN-Netzwerktreiber an das CAN Subsystem wurde ein neuer 'Ethernet-Protokoll-Typ' ETH_P_CAN eingeführt, was sicherstellt, dass alle im System eingehenden CAN Frames dem RX-Dispatcher zugeleitet werden. Dazu meldet sich das CAN SubsystemModul `can.ko` 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.

## B.1 Generelle Hinweise

In diesem Abschnitt wird auf grundsätzliche Zusammenhänge bei der Inbetriebnahme des CAN Subsystems eingegangen und die Basis für die Anwendung der einzelnen CAN Protokolle gelegt.

### B.1.1 CAN Protokolle

Wie in Abbildung B.3 gezeigt, bestehen die Module des CAN Subsystems aus

**can.ko** Dem Basismodul mit Filterlisten von CAN Identifiern, die von den CAN Protokollen angefordert wurden. Dieses Modul bietet die Schnittstellen für CAN Protokolle zu den CAN Netzwerkgeräten.

**can-raw.ko** Dieses CAN Protokoll setzt auf das Basismodul auf und leitet die empfangenen CAN Frames gemäß den vom Anwender gesetzten Filtern an den Anwender weiter. Über den RAW-Socket gesendete CAN Frames werden zum Versenden in die Warteschlange des jeweiligen Netzwerktreibers eingehängt.

**can-bcm.ko** Dieses CAN Protokoll setzt auf das Basismodul auf und kann eine Filterung im Dateninhalt von CAN Frames ausführen, sowie CAN Frames zyklisch auf den CAN Bus senden.

Diese drei Module sind Bestandteil des Linux Kernel seit der Version 2.6.25.

Für ältere Linux Kernel Versionen - auch für den Kernel 2.4 - besteht die Möglichkeit die Module aus dem SVN Versionsverwaltungssystem herunter zu laden, in dem die OpenSource Entwicklung des SocketCAN Projektes[1] stattfindet. In dem SVN Versionsverwaltungssystem werden auch die Werkzeuge aus Anhang C sowie das CAN Protokoll ISO-TP (ISO 15765-2) weiterentwickelt.

### B.1.2 CAN Netzwerktreiber

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 CAN Subsystem 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.

[1] http://developer.berlios.de/projects/socketcan

Wenngleich ein Treiber nach dem Netzwerk-Treibermodell aufgrund der im Linux Kernel vorhandenen Infrastruktur einfacher zu realisieren ist, sind die bei einer kommerziellen CAN Hardware beigelegten Treiber für das CAN Subsystem häufig 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 erste Linux Netzwerktreiber für kommerzielle CAN Hardware wurde mit der Unterstützung des Autors dieser Arbeit für die Produkte der Fa. PEAK System realisiert. Von PEAK System (`www.peak-system.com`) wird eine Reihe *passiver* CAN Hardware angeboten, die auf dem Philips/NXP SJA1000 [46] CAN Controller basieren:

**PCAN-ISA** PC-ISA I + II CAN-Interface one and two channel

**PCAN-PC/104** PC/104 CAN-Interface one and two channel

**PCAN-Dongle** PC-Parallel Port CAN-Interface one channel

**PCAN-PCI** PC-PCI CAN-Interface one or two channel

**PCAN-PCI Express** PC-PCI Express CAN-Interface one or two channel

**PCAN-PC Card** PC-PC Card (PCMCIA) CAN-Interface one or two channel

**PCAN-PC/104plus** PC/104plus CAN-Interface one and two channel

**PCAN-USB** PC-USB CAN Interface

Eine entsprechende Installationsanleitung liegt dem Treiber bei und geht auch auf die Installation der CAN Netzwerktreiber ein.

Der PEAK Linux Treiber ist ein weitgehend konfigurierbarer aber monolitischer Treiber, der für eine Anzahl unterschiedlicher Hardware genutzt werden kann, um den Quellcode-Umfang möglichst gering zu halten.

In dem SocketCAN SVN Versionsverwaltungssystem auf www.berlios.de werden modulare CAN Treiber entwickelt, die sich in das übliche Konzept der Linux Kernel Entwicklung einpassen. Dazu wird eine Infrastruktur realisiert, die das Konfigurieren und Entwickeln von CAN Netzwerktreibern vereinheitlicht und erheblich erleichtert. Aktuell wird folgende CAN Hardware unterstützt:

**generic Platform Bus based SJA1000 driver (SJA1000)** This driver adds support for the SJA1000 chips connected to the "platform bus" (Linux abstraction for directly to the processor attached devices). Which can be found on various boards from Phytec (`www.phytec.de`) like the PCM027, PCM038.

**EMS CPC-PCI and CPC-PCIe Card (SJA1000)** This driver is for the one or two channel CPC-PCI and CPC-PCIe cards from EMS Dr. Thomas Wuensche (`www.ems-wuensche.de`).

**EMS CPC-CARD Card (SJA1000)** This driver is for the one or two channel CPC-CARD cards from EMS Dr. Thomas Wuensche (www.ems-wuensche.de).

**IXXAT PCI Card (SJA1000)** This driver is for the IXXAT PC-I 04/PCI card (1 or 2 channel) from the IXXAT Automation GmbH (www.ixxat.de).

**PEAK PCAN PCI Card (SJA1000)** This driver is for the PCAN PCI, the PC-PCI CAN plug-in card (1 or 2 channel) from PEAK Systems (www.peak-system.com).

**MPL PIPCAN CAN module driver (SJA1000)** This driver adds support for the PIP-CAN module used on some SBC boards from MPL AG (www.mpl.ch).

**Kvaser PCIcanx and Kvaser PCIcan PCI Cards (SJA1000)** This driver is for the the PCIcanx and PCIcan cards (1, 2 or 4 channel) from Kvaser (www.kvaser.com).

**Freescale MPC5200 onboard CAN controller (MSCAN)** The Motorola Scalable Controller Area Network (MSCAN) definition is based on the MSCAN12 definition which is the specific implementation of the Motorola Scalable CAN concept targeted for the Motorola MC68HC12 Microcontroller Family. This driver supports the Freescale MPC5200 onboard dualCAN controller (www.freescale.com).

**Atmel AT91 onchip CAN controller** for the CAN controller in Atmel's AT91SAM9263.

**TI High End CAN Controller** Driver for TI HECC (High End CAN Controller) module found on many TI devices. The specifications are available from www.ti.com.

**Microchip MCP251x SPI CAN controllers** Uses the Kernel SPI Subsystem.

**CANcard / CANcard2 / EDICcardC / EDICcard2 (Softing)** Softing CAN pcmcia cards with on-board CPU (www.softing.com). This driver needs an extra Softing Firmware v4.6 which is also available on the SocketCAN project site.

**Serial / USB serial CAN Adaptors (slcan)** CAN driver for several 'low cost' CAN interfaces that are attached via serial lines or via USB-to-serial adapters using the LAWICEL ASCII protocol (www.lawicel.com, http://www.can232.com). The driver implements the tty linediscipline N_SLCAN based on the Serial Line IP (SLIP) driver, that is used for dial-up Internet Protocol networking.

### B.1.3 virtuelle CAN Netzwerktreiber

Der virtuelle CAN Bus-Treiber realisiert ein logisches CAN-Netzwerkgerät, über das Anwendungen auf einem System ohne real vorhandene CAN Hardware kommunizieren können.

Die Idee entspricht dem eines Loopback-Device, wie man es vom Internet Protokoll kennt. Allerdings ergibt sich eine qualifizierte Angabe einer CAN Nachricht aus dem CAN Identifier und dem CAN Bus auf dem dieser Identifier gesendet wird. Daher wird man in der Praxis häufig mehr als eine Instanz des virtuellen CAN Treibers benötigen.

Das VCAN Modul `vcan.ko` ist Bestandteil des Linux Kernel seit der Version 2.6.25. Für ältere Linux Kernel Versionen - auch für den Kernel 2.4 - besteht die Möglichkeit sich dieses Modul aus dem SVN Versionsverwaltungssystem herunter zu laden.

Nach dem Laden des VCAN Moduls (z.B. mit `modprobe vcan`) werden bei Kernelversionen kleiner als 2.6.24 automatisch vier virtuelle CAN Geräte erzeugt (vcan0 .. vcan3). Dieser Wert ist mit einem Modulparameter 'numdev=<Anzahl>' beim Laden des Modules veränderbar.

Seit Kernel 2.6.24 wird das so genannte Netlink Interface des Linux Kernels genutzt, um zur Laufzeit Instanzen von Software-Netzwerkgeräten wie dem virtuellen CAN Bus erzeugen und entfernen zu können. Dazu muss das Werkzeug ip(8) aus dem iproute2 Softwarepaket auf dem System installiert sein. Beispiele für die Anwendung von ip(8):

**ip link add type vcan** Create a virtual CAN network interface.

**ip link add dev vcan42 type vcan** Create a virtual CAN network interface with a specific name 'vcan42'.

**ip link del vcan42** Remove a (virtual CAN) network interface 'vcan42'.

### B.1.4 Konfigurieren und Hochfahren von CAN Netzwerktreibern

Wie auch bei anderer Netzwerk Hardware müssen CAN Netzwerkgeräte vor ihrer Nutzung 'hochgefahren' werden. Dazu wird das übliche Werkzeug ifconfig(8) verwendet:

**ifconfig can0 up** Fährt das Netzwerkgerät 'can0' hoch und macht es so benutzbar für Andwender.

**ifconfig can0 down** Fährt das Netzwerkgerät 'can0' herunter.

**ifconfig** (ohne Parameter) Zeigt alle aktiven Netzwerkgeräte an.

Beim Einsatz einer 'realen' CAN Hardware muss die Bitrate für das CAN Netzwerkgerät zuvor gesetzt werden. Dieses ist abhängig von der jeweils verwendeten Hardware und wird bei der aktuellen CAN Treiberinfrastruktur über die so genannte Netlink Konfigurationsschnittstelle mit Hilfe des ip(8) Werkzeuges eingestellt, z.B. mit:

**ip link set can0 type can bitrate 125000**

Es besteht auch die Möglichkeit hardwareunabhängige eigene Timingwerte zu setzten. Nach der Einstellung der Bitrate kann das CAN Netzwerkgerät hochgefahren werden. Hinweise zur Verwendung von ip(8) finden sich in der Dokumentation des CAN Subsystems im Linux Kernel [19] in Abschnitt 6.5 .

### B.1.5 CAN Datenstrukturen

Für die Kommunikation auf dem CAN Bus wird eine neue Protokoll-Familie `PF_CAN` im Socket-Layer implementiert. Aus Anwender- bzw. Programmierersicht werden die CAN-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.

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

```
/**
 * struct sockaddr_can - the sockaddr structure for CAN sockets
 * @can_family:  address family number AF_CAN.
 * @can_ifindex: CAN network interface index.
 * @can_addr:    protocol specific address information
 */
struct sockaddr_can {
        sa_family_t can_family;
        int         can_ifindex;
        union {
                /* transport protocol class address information (e.g. ISOTP) */
                struct { canid_t rx_id, tx_id; } tp;

                /* reserved for future CAN protocols address information */
        } 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 unterschiedlichen CAN Identifiern eine virtuelle Punkt-zu-Punkt-Verbindung ab.

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

```
/*
 * Controller Area Network Identifier structure
 *
 * bit 0-28     : CAN identifier (11/29 bit)
 * bit 29       : error frame flag (0 = data frame, 1 = error frame)
 * bit 30       : remote transmission request flag (1 = rtr frame)
 * bit 31       : frame format flag (0 = standard 11 bit, 1 = extended 29 bit)
 */
typedef __u32 canid_t;
```

```
/**
 * struct can_frame - basic CAN frame structure
 * @can_id:  the CAN ID of the frame and CAN_*_FLAG flags, see above.
 * @can_dlc: the data length field of the CAN frame
 * @data:    the CAN frame payload.
 */
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)));
};
```

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 sind, 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;
```

Der 64 Bit Zugriff erlaubt in der Programmiersprache C eine 'Struktur' oder eine 'Union' über das 8 Byte Array zu definieren ('casten'). Im Beispiel ist dieses ein vorzeichenloser 64 Bit Wert. Aufgrund der zu erwartenden Probleme mit der Byte-Order auf Little/Big Endian Systemen ist hierbei allerdings größte Vorsicht geboten, weil dabei schnell unportabler Quellcode erzeugt werden kann.

Die C Typdefinition für den CAN Identifier `canid_t` enthält auch Bitwerte für besondere Kennzeichnungen, wie der Kennzeichnung für die 29 Bit Identifier des Extended Frame Format (EFF). Entprechende Bitwert Definitionen und Bitmasken sind ebenfalls in der Include-Datei `include/linux/can.h` definiert.

### B.1.6 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 CAN Anwendungen verwendet:

```
struct timeval tv;
int s; /* socket */

(..)

read(s, &msg, sizeof(msg));

if (ioctl(s, SIOCGSTAMP, &tv) < 0)
        perror("SIOCGSTAMP");
else
        printf("(%ld.%06ld) ", tv.tv_sec, tv.tv_usec);

(..)
```

Die Zeitstempel, die mit SIOCGSTAMP gelesen werden können, haben unter Linux eine Auflösung von einer Mikrosekunde und werden beim Empfang eines CAN Frame im CAN-Networkdevice automatisch gesetzt. Es existieren auch Zeitstempel mit einer Auflösung von Nanosekunden, auf die hier aber nicht eingegangen wird.

## B.2 CAN Raw Protokoll 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 read(2) oder recvfrom(2)) ist als Interface-Index Null einzutragen. Im obigen Beispiel also `addr.can_ifindex = 0`. Das Senden von CAN Frames über einen solchen RAW-Socket muss dann über sendmsg(2) erfolgen, um anhand der dabei mitgegebenen Adressinformationen den CAN Bus zu definieren, auf dem das CAN Frame ausgesendet werden soll.

### B.2.1 CAN Identifier Filter

Der RAW-Socket bietet eine Filterfunktion mit der Bereiche von CAN-IDs aus dem Datenstrom ausgefiltert werden können. Dazu nutzt der RAW-Socket die durch das

CAN Subsystem Basismodul bereitgestellen Filtermöglichkeiten und stellt sie dem CAN Anwender für jeden Socket separat zur Verfügung. Zum Setzen der CAN Filter kann noch vor dem Aufruf von bind(2) mit dem Systemaufruf setsockopt(2) ein Array von CAN Filtern gesetzt werden. In diesem Beispiel sollen alle CAN-IDs von 0x200 - 0x2FF (auch extended CAN Frames und RTR Frames) 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.

Die Definition der CAN Filter ist in der Include-Datei `include/linux/can.h` definiert und verhält sich ähnlich wie bekannte CAN Filter, die in CAN Hardware Controllern realisiert sind:

```
/**
 * struct can_filter - CAN ID based filter in can_register().
 * @can_id:   relevant bits of CAN ID which are not masked out.
 * @can_mask: CAN mask (see description)
 *
 * Description:
 * A filter matches, when
 *
 *          <received_can_id> & mask == can_id & mask
 *
 * The filter can be inverted (CAN_INV_FILTER bit set in can_id) or it can
 * filter for error frames (CAN_ERR_FLAG bit set in mask).
 */
struct can_filter {
        canid_t can_id;
        canid_t can_mask;
};

#define CAN_INV_FILTER 0x20000000U /* to be set in can_filter.can_id */
```

Ein weiteres Beispiel mit vier Filtern:

```
struct can_filter rfilter[4];

rfilter[0].can_id   = 0x80001234;   /* Exactly this EFF frame */
```

```
rfilter[0].can_mask = (CAN_EFF_FLAG | CAN_RTR_FLAG | CAN_EFF_MASK);

rfilter[1].can_id   = 0x123;        /* Exactly this SFF frame */
rfilter[1].can_mask = (CAN_EFF_FLAG | CAN_RTR_FLAG | CAN_SFF_MASK);

rfilter[2].can_id   = 0x200 | CAN_INV_FILTER; /* all, but 0x200-0x2FF SFF */
rfilter[2].can_mask = (CAN_EFF_FLAG | CAN_RTR_FLAG | 0xF00);


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));
```

Wie beschrieben gilt grundsätzlich die Filterregel

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

was allerdings auch bedeutet, dass der Bitwert für extended CAN Frames mit 29 Bit Identifier (CAN_EFF_FLAG) sowie der Bitwert für Remote Transmission Requests (CAN_RTR_FLAG) bei der Definition der Filtermaske im Allgemeinen gesetzt sein sollten, will man nicht auf die CAN ID 0x123 (= 0x123 SFF) und 0x80000123 (= 0x123 EFF) gleichzeitig filtern. Ein nicht gesetztes Bit lässt immer beide Möglichkeiten zu.

Wird beim Erzeugen und Binden des RAW-Sockets kein Filter gesetzt, wird als Grundeinstellung ein Filter gesetzt, der alle CAN Frames (ausser Error Frames) durchlässt (wie im obigen Beispiel der `filter[3]`). Grundsätzlich gilt:

- Die Grundeinstellung ist ein Filter, der alle (Daten-) CAN Frames passieren lässt.
- Es können beliebig viele CAN Filter (0 .. n) gesetzt werden - auch kein Filter.
- Beim Setzen von Filtern, wird der vorherige Filtersatz ersetzt.
- Die einzelnen Filter sind voneinander unabhängig (logisch ODER).

Für die Anwendung der Filterfunktion muss für die Definition der setsockopt(2) Funktionsparameter die Include-Datei `include/linux/can/raw.h` eingebunden werden. Eine Veränderung des Filters zur Laufzeit ist über weitere Aufrufe von setsockopt(2) möglich. Für 'interaktive' Tests mit den beschriebenen CAN Filtern sei an dieser Stelle auf das Werkzeug **candump** verwiesen, dass im Kapitel C.1.1 beschrieben ist. Bei **candump** können als Kommandozeilenparameter alle bekannten Filtermöglichkeiten (ausgenommen Null Filter) für eine gegebene CAN Netzwerkschnittstelle angegeben und ausprobiert werden, welche wiederum anhand der Filterlistenansicht (siehe Kapitel A.7) verifiziert werden können.

## B.3 CAN Broadcast Manager Protokoll Sockets

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 von Botschaftslängen
- Beantworten von RTR-Botschaften
- Timeoutüberwachung von Botschaften
- Reduzierung der Häufigkeit von Änderungsnachrichten (Throttle-Funktion)

### B.3.1 Kommunikation mit dem Broadcast-Manager

Im Gegensatz zum RAW-Socket (Kapitel B.2) und den Transportprotokoll-Sockets (Kapitel B.4) 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 - head of messages to/from the broadcast manager
 * @opcode:    opcode, see enum below.
 * @flags:     special flags, see below.
 * @count:     number of frames to send before changing interval.
 * @ival1:     interval for the first @count frames.
 * @ival2:     interval for the following frames.
 * @can_id:    CAN ID of frames to be sent or received.
 * @nframes:   number of frames appended to the message head.
 * @frames:    array of CAN frames.
 */
struct bcm_msg_head {
        __u32 opcode;
        __u32 flags;
        __u32 count;
        struct timeval ival1, ival2;
```

```
        canid_t can_id;
        __u32 nframes;
        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/ERR 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 Multiplex-Nachrichten auch wenn dieses aufgrund der gleichen Länge nicht nötig wäre. Siehe Seite 188.

**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 muss 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 193.

**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.

## B.3.2 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;
} 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.can_id    = 0x42; /* obsolete when using TX_CP_CAN_ID */
msg.frame.can_dlc   = 3;
msg.frame.data[0]   = 0x12;
msg.frame.data[1]   = 0x34;
msg.frame.data[2]   = 0x56;

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.

### B.3.2.1 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).

#### B.3.2.2 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.

#### B.3.2.3 Aussenden verschiedener Nutzdaten (Multiplex)

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.

### B.3.3 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.

### B.3.4 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.

### B.3.5 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.

### B.3.6 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 187) 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

#### B.3.6.1 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 Timeoutüberwachung 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.

#### B.3.6.2 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.

#### B.3.6.3 Nachrichtenfilterung (Nutzdaten - simple)

Analog der Übertragung der Nutzdaten bei TX_SETUP (siehe Seite 187) 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.

#### B.3.6.4 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 188).

Zur Filterung von Multiplex-Nachrichten werden mindestens zwei `can_frame`'s (`nframes` $\geq$ 2) 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.

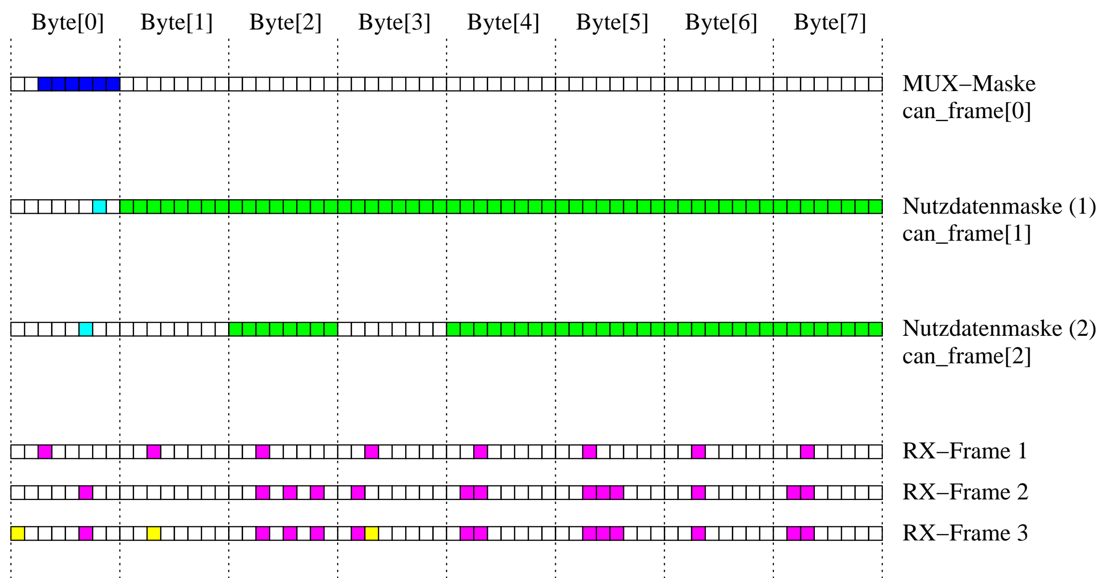



Abbildung B.4: Beispiel für die Anwendung des Multiplexfilters

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 B.4) ist die MUX-Maske im Byte 0 auf `0x3F` 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.

#### B.3.6.5 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 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.

#### B.3.6.6 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 muss Null sein und so werden folglich auch keine Nutzdaten (`can_frame`'s) an den *Broadcast-Manager* geschickt. 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 ist eine Funktionalität, die auch mit einem RAW-Socket und mit RAW-Filtern (siehe Kapitel B.2.1) realisiert werden kann.

#### B.3.6.7 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.

### B.3.7 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.

### B.3.8 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.

### B.3.9 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.

## B.4 CAN Transportprotokoll Sockets (Stream)

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 201.

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 - the sockaddr structure for CAN sockets
 * @can_family:  address family number AF_CAN.
 * @can_ifindex: CAN network interface index.
 * @can_addr:    protocol specific address information
 */
struct sockaddr_can {
        sa_family_t can_family;
```

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

                /* reserved for future CAN protocols address information */
        } 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.

```
/* 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.tp.tx_id  = 0x760;
addr.can_addr.tp.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.tp.tx_id  = 0x440;
addr.can_addr.tp.rx_id  = 0x760;

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

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

close(s);
```

### B.4.1 Tracemode

Wie schon beim RAW-Socket (siehe Kapitel B.2) 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. Bei bisher realisierten Transportprotokollen besteht grundsätzlich die 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_addr.tp.tx_id  = 0x248;
addr1.can_addr.tp.rx_id  = 0x448;

addr2.can_family = AF_CAN;
addr2.can_ifindex = ifr.ifr_ifindex; /* also can0 */
addr2.can_addr.tp.tx_id  = 0x448;
addr2.can_addr.tp.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.

## B.4.2 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 IMPL.) */
  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.

Entgegen bisherigen Implementierungen unterstützt diese Realisierung für das CAN Subsystem die dynamische Identifiervergabe nicht im Rahmen der TP1.6-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 CAN Subsystem über Broadcastnachrichten auf der Benutzerebene mit RAW-Sockets realisiert. Um die maximale Zeitspanne zwischen dem Aushandeln der Identifier und der Eröffnung des Transportkanals einhalten zu können, muss sichergestellt sein, dass die Transportprotokollmodule bei der Kanaleröffnung bereits geladen sind.

### B.4.3 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.

Entgegen bisherigen Implementierungen unterstützt diese Realisierung für das CAN Subsystem 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 CAN Subsystem über Broadcastnachrichten auf der Benutzerebene mit RAW-Sockets realisiert. Um die

maximale Zeitspanne zwischen dem Aushandeln der Identifier und der Eröffnung des Transportkanals einhalten zu können, muss sichergestellt sein, dass die Transportprotokollmodule bei der Kanaleröffnung bereits geladen sind.

### B.4.4 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.

## B.5 CAN Transportprotokoll Sockets (Datagram)

Im Gegensatz zu Stream Sockets auf dem CAN Bus, wie sie in Kapitel B.4 beschrieben sind, erfolgt bei Datagramm Sockets kein expliziter Verbindungsaufbau. Allerdings wird auch bei diesen CAN-Transport-Protokollen eine virtuelle Punkt-zu-Punkt-Verbindung auf dem CAN Bus auf zwei CAN-IDs abgebildet. Die Verbindung wird mit den angegebenen Adressen analog zu UDP/IP Sockets durch einen bind(2) Systemaufruf definiert und kann danach mit den üblichen read(2) und write(2) Systemaufrufen genutzt werden. Das aufwändige Verfahren mit listen(2) und accept(2) auf der Seite des Servers einer Kommunikationsverbindung entfällt.

### B.5.1 ISO-Transportprotokoll (ISO 15765-2)

Zur Nutzung eines CAN ISO 15765-2 Transport-Protokolles wird ein Socket vom Typ DGRAM geöffnet:

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

Protokollspezifische Parameter können nach dem Öffnen eines Sockets mit setsockopt(2) und getsockopt(2) gesetzt bzw. gelesen werden. Für das ISO-TP Protokoll sind die Parameter für diese Systemaufrufe in der Datei `include/linux/can/isotp.h` aufgeführt:

```
#define CAN_ISOTP_OPTS          1
#define CAN_ISOTP_RECV_FC       2

struct can_isotp_options {

        __u32 flags;            /* set flags for isotp behaviour.       */
                                /* __u32 value : flags see below        */

        __u32 frame_txtime;     /* frame transmission time (N_As/N_Ar)  */
                                /* __u32 value : time in nano secs      */

        __u8  ext_address;      /* set address for extended addressing  */
                                /* __u8 value : extended address        */

        __u8  txpad_content;    /* set content of padding byte (tx)     */
                                /* __u8 value : content on tx path      */

        __u8  rxpad_content;    /* set content of padding byte (rx)     */
                                /* __u8 value : content on rx path      */
};
```

```
struct can_isotp_fc_options {

        __u8  bs;               /* blocksize provided in FC frame      */
                                /* __u8 value : blocksize. 0 = off     */

        __u8  stmin;            /* separation time provided in FC frame */
                                /* __u8 value :                        */
                                /* 0x00 - 0x7F : 0 - 127 ms            */
                                /* 0x80 - 0xF0 : reserved              */
                                /* 0xF1 - 0xF9 : 100 us - 900 us       */
                                /* 0xFA - 0xFF : reserved              */

        __u8  wftmax;           /* max. number of wait frame transmiss. */
                                /* __u8 value : 0 = omit FC N_PDU WT   */
};


/* flags for isotp behaviour */

#define CAN_ISOTP_LISTEN_MODE   0x01    /* listen only (do not send FC) */
#define CAN_ISOTP_EXTEND_ADDR   0x02    /* enable extended addressing */
#define CAN_ISOTP_TX_PADDING    0x04    /* enable CAN frame padding tx path */
#define CAN_ISOTP_RX_PADDING    0x08    /* enable CAN frame padding rx path */
#define CAN_ISOTP_CHK_PAD_LEN   0x10    /* check received CAN frame padding */
#define CAN_ISOTP_CHK_PAD_DATA  0x20    /* check received CAN frame padding */
#define CAN_ISOTP_HALF_DUPLEX   0x40    /* half duplex error state handling */
```

Durch die Vielzahl möglicher Kommunikationsparameter in ISO-TP und deren Anwendungsmöglichkeiten sind auch die über setsockopt(2) einstellbaren Parameter entsprechend umfangreich. Für eine Auswahl verschiedener Anwendungsbeispiele sei an dieser Stelle auf den Quellcode der ISO-TP Werkzeuge aus Kapitel C.3 verwiesen, die vom SocketCAN SVN Versionsverwaltungssystem herunter geladen werden können. Mit den dort realisieren Programmen, werden alle Optionen genutzt, die an dieser Stelle beschrieben werden. Es bietet sich damit auch an, das Verhalten der ISO-TP Implementierung auf dem CAN Bus bezüglich der verschiedenen Optionen zu studieren, da die Werkzeuge über die Kommandozeile ein Setzen der hier beschriebenen Parameter ermöglichen.

Die zur Laufzeit auch nach dem bind(2) noch änderbaren Parameter werden im Folgenden beschrieben. Da die Namensgebung der Strukturelemente innerhalb der Strukturen `can_isotp_options` und `can_isotp_fc_options` eindeutig ist, wird für eine Bessere Lesbarkeit im Folgenden nur der Elementname verwendet.

### struct can_isotp_options

**flags** Definiert über Bitwerte das Verhalten des ISO-TP Protokolls:

**CAN_ISOTP_LISTEN_MODE** Der ISO-TP Socket empfängt Daten, reassembliert sie und leitet Sie an das Anwendungsprogramm weiter - allerdings ohne

die üblichen FlowControl (FC) CAN Nachrichten an den Absender zu schicken. Dieses Verhalten wird z.B. benötigt, wenn man ein Werkzeug zum Mitlesen einer ISO-TP Verbindung realisiert, wie den **isotpsniffer** (siehe Kapitel C.3.4).

**CAN_ISOTP_EXTEND_ADDR** Das ISO-TP Protokoll verfügt über die Möglichkeit einer erweiterten Adressierung, die im Datenbereich (`data[]`) des CAN Frames realisiert wird und somit die Nutzdaten pro CAN Frame entsprechend um ein Byte reduziert. Bei gesetztem Bitwert wird der Adresswert aus dem Element `ext_address` für die erweiterte Adressierung genutzt.

**CAN_ISOTP_TX_PADDING** Ein CAN Frame kann maximal 8 Byte im Datenbereich (`data[]`) transportieren. Wenn in einem ISO-TP CAN Frame diese 8 Byte nicht genutzt werden, besteht die Möglichkeit im letzten CAN Frame einer ISO-TP PDU weniger als 8 Byte zu übertragen, was auch auch Gründen der Buslast zu empfehlen ist. Gemäss der ISO-TP Spezifikation kann das letzte CAN Frame einer Datenübertragung aber auch mit einem zu bestimmenden Bytewert auf eine Länge vom 8 Byte aufgefüllt werden. Bei gesetztem Bitwert wird das Füllbyte aus dem Element `txpad_content` für das Auffüllen des letzten CAN Frames genutzt.

**CAN_ISOTP_RX_PADDING** Soll die Konsistenz der Füllbytes von der Gegenseite geprüft werden (siehe CAN_ISOTP_CHK_PAD_*), wird mit dem Setzen dieses Bitwertes der Bytewert des erwartete Füllbyte der Gegenseite aus dem Element `rxpad_content` übernommen.

**CAN_ISOTP_CHK_PAD_LEN** Werden bei gesetztem Bitwert von der Gegenseite ISO-TP CAN Frames empfangen, die nicht auf 8 Byte aufgefüllt sind, so werden diese ISO-TP PDUs verworfen.

**CAN_ISOTP_CHK_PAD_DATA** Werden bei gesetztem Bitwert von der Gegenseite ISO-TP CAN Frames empfangen, deren Füllbyte nicht dem Wert aus dem Element `rxpad_content` entspricht, so werden diese ISO-TP PDUs verworfen.

**frame_txtime** In der ISO-TP Spezifikation werden die Transferzeiten, die ein gesendetes CAN Frame den Bus physikalisch belegt mit den Netzwerkschicht Parametern N_Ar (receive) und N_As (send) beschrieben. Dieser in diesem Element in Nanosekunden angegebene Wert wird gemäß der Spezifikation zu den von der Gegenseite geforderten Wartezeiten (aus `stmin`) addiert. Generell bietet dieser zur Laufzeit veränderbare Wert aber auch die Möglichkeit die Sendeverzögerung lokal zu erhöhen, wenn die Kommunikation mit den von der Gegenseite gegebenen `stmin` Werten nicht stabil möglich ist.

**ext_address** Gibt den Bytewert für die erweiterte Adressierung an, sofern diese durch den Bitwert CAN_ISOTP_EXTEND_ADDR aktiviert wurde.

**txpad_content** Gibt den Bytewert für das sendeseitige Füllbyte an, sofern das Füllen der ISP-TP PDUs durch den Bitwert CAN_ISOTP_TX_PADDING aktiviert wurde.

**rxpad_content** Gibt den Bytewert für die Konsistenzprüfung der Füllbytes von der Gegenseite an, sofern der Bitwert CAN_ISOTP_RX_PADDING aktiviert wurde.

### struct can_isotp_options

**bs** Gibt die ISO-TP Blocksize (BS) an, die im Rahmen von Flow Control (FC) CAN Frames an die Gegenseite gesendet werden, damit diese ihre Nutzdaten entsprechend sendet.

**stmin** Gibt den minimalen Abstand zwischen zwei ISO-TP CAN Frames an, die im Rahmen von Flow Control (FC) CAN Frames an die Gegenseite gesendet werden, damit diese ihre Nutzdaten entsprechend sendet.

**wftmax** Ist der Empfänger kurzzeitig nicht in der Lage, Daten von der Gegenseite zu empfangen, kann er bis zu `wftmax` sogenannte Wait Frames senden, bis der Empfänger selbst den Empfang fortsetzt oder abbricht. Diese im Empfänger realisierte Abbruchsbedingung ist in der aktuellen Implementierung nicht realisiert.

Nach der (optionalen) Konfiguration des geöffneten ISO-TP Sockets mit setsockopt(2) werden die entprechenden Kommunikationsadressen (CAN IDs) mit bind(2) übergeben. Im Gegensatz zu bekannten Adressen aus im Internet Protokoll muss der Anwender beim Verbindungsaufbau nicht nur die CAN-ID seines Kommunikationspartners, sondern auch seine eigene angeben.

Die bei bind(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 - the sockaddr structure for CAN sockets
 * @can_family:  address family number AF_CAN.
 * @can_ifindex: CAN network interface index.
 * @can_addr:    protocol specific address information
 */
struct sockaddr_can {
        sa_family_t can_family;
        int         can_ifindex;
        union {
                /* transport protocol class address information (e.g. ISOTP) */
                struct { canid_t rx_id, tx_id; } tp;

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

Im folgen Beispiel soll eine ISO-TP Verbindung aufgebaut werden, wobei als lokale Adresse die CAN-ID 0x740 verwendet wird und die Gegenseite auf der CAN-ID 0x760

sendet. In dem Beispiel wird sendeseitig das Auffüllen der ISO-TP CAN Frames mit dem Wert 0x42 eingestellt und eine Nachricht ('Hello World') an die Gegenseite gesendet. Dieses Beispiel ist dahingehend vereinfacht, dass auf eine Fehlerbehandlung teilweise verzichtet wird. Eine mögliche Fehlerbehandlung ist aber in den Quelltexten zu den Beispielprogrammen in Kapitel C.3 realisiert.

```
int s;
struct sockaddr_can addr;
struct ifreq ifr;
static struct can_isotp_options opts;
unsigned char buf[] = "Hello World";
int buflen = 12;

if ((s = socket(PF_CAN, SOCK_DGRAM, CAN_ISOTP)) < 0) {
    perror("socket");
    exit(1);
}

opts.flags = CAN_ISOTP_TX_PADDING;
opts.txpad_content = 0x42;
setsockopt(s, SOL_CAN_ISOTP, CAN_ISOTP_OPTS, &opts, sizeof(opts));

addr.can_family = AF_CAN;
strcpy(ifr.ifr_name, "can0");
ioctl(s, SIOCGIFINDEX, &ifr);
addr.can_ifindex = ifr.ifr_ifindex;
addr.can_addr.tp.tx_id = 0x740;
addr.can_addr.tp.rx_id = 0x760;

if (bind(s, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
    perror("bind");
    close(s);
    exit(1);
}

write(s, buf, buflen);

/*
 * due to a Kernel internal wait queue the PDU is sent completely
 * before close() returns.
 */
close(s);
```

Nach dem bind(2) könnte beispielsweise auch mit einem read(2) oder einem select(2) blockierend auf ankommende Daten gewartet werden.

Ein Beispiel für den Tracemode (CAN_ISOTP_LISTEN_MODE) ist im Quelltext zu dem Werkzeug **isotpsniffer** (siehe Kapitel C.3.4) verfügbar. Das grundsätzliche Verfahren zum Mitlesen von Transportprotokollen wird im Kapitel B.4.1 erläutert.

# C Werkzeuge

In diesem Kapitel werden die in dieser Arbeit genutzten Werkzeuge für den Umgang mit dem CAN Subsystem des Linux Kernel beschrieben. Die vorgestellte Liste erhebt keinen Anpruch auf Vollständigkeit und bezieht sich auf die vom Autor realisierten Werkzeuge, die zum Herunterladen auf dem SocketCAN SVN Versionverwaltungssystem bereitgestellt sind (siehe http://developer.berlios.de/svn/?group_id=6475):

**Anonymous SVN Access via SVN** This project's BerliOS Developer SVN repository can be checked out through anonymous (svnserve) SVN with the following instruction set.

```
svn checkout svn://svn.berlios.de/socketcan/trunk
```

**Anonymous SVN Access via HTTP** This project's BerliOS Developer SVN repository can be checked out through anonymous HTTP with the following instruction set.

```
svn checkout http://svn.berlios.de/svnroot/repos/socketcan/trunk
```

**Developer SVN Access via SSH** Only project developers can access the SVN tree via this method. SSH2 must be installed on your client machine. Substitute <user> with the proper value. Enter your site password when prompted.

```
svn checkout svn+ssh://<user>@svn.berlios.de/svnroot/repos/socketcan/trunk
```

**Developer SVN Access via HTTPS** Only project developers can access the SVN tree via this method. Substitute <user> with the proper value. Enter your site password when prompted.

```
svn checkout https://<user>@svn.berlios.de/svnroot/repos/socketcan/trunk
```

Die beschriebenen Werkzeuge geben zumeist einen Hilfe Text aus, wenn sie ohne bzw. mit falschen Parametern aufgerufen werden. In dieser Übersicht sollen daher nur zusätzliche Hinweise gegeben werden, die sich aus den gegebenen Hilfe Texten nicht unmittelbar ergeben.

## C.1 Anzeige und Erzeugung von CAN Daten

### C.1.1 candump

**candump** ist ein Programm, das über den RAW-Socket CAN Frames von einem oder mehreren CAN-Interfaces einliest und in lesbarer Form - auf Wunsch mit Zeitstempeln - ausgibt.

Mit `candump -?` oder ohne Parameter aufgerufen, erscheint folgender Hilfetext.

```
Usage: candump [options] <CAN interface>+
  (use CTRL-C to terminate candump)

Options: -t <type>   (timestamp: (a)bsolute/(d)elta/(z)ero/(A)bsolute w date)
         -c          (increment color mode level)
         -i          (binary output - may exceed 80 chars/line)
         -a          (enable additional ASCII output)
         -S          (swap byte order in printed CAN data[] - marked with '‘' )
         -s <level>  (silent mode - 0: off (default) 1: animation 2: silent)
         -b <can>    (bridge mode - send received frames to <can>)
         -B <can>    (bridge mode - like '-b' with disabled loopback)
         -l          (log CAN-frames into file. Sets '-s 2' by default)
         -L          (use log file format on stdout)
         -n <count>  (terminate after receiption of <count> CAN frames)
         -r <size>   (set socket receive buffer to <size>)
         -d          (monitor dropped CAN frames)

Up to 16 CAN interfaces with optional filter sets can be specified
on the commandline in the form: <ifname>[,filter]*

Comma separated filters can be specified for each given CAN interface:
 <can_id>:<can_mask> (matches when <received_can_id> & mask == can_id & mask)
 <can_id>~<can_mask> (matches when <received_can_id> & mask != can_id & mask)
 #<error_mask>       (set error frame filter, see include/linux/can/error.h)

CAN IDs, masks and data content are given and expected in hexadecimal values.
When can_id and can_mask are both 8 digits, they are assumed to be 29 bit EFF.
Without any given filter all data frames are received ('0:0' default filter).

Use interface name 'any' to receive from all CAN interfaces.

Examples:
candump -c -c -ta can0,123:7FF,400:700,#000000FF can2,400~7F0 can3 can8
candump -l any,0~0,#FFFFFFFF    (log only error frames but no(!) data frames)
candump -l any,0:0,#FFFFFFFF    (log error frames and also all data frames)
candump vcan2,92345678:DFFFFFFF (match only for extended CAN ID 12345678)
candump vcan2,123:7FF (matches CAN ID 123 - including EFF and RTR frames)
candump vcan2,123:C00007FF (matches CAN ID 123 - only SFF and non-RTR frames)
```

Anmerkungen:

**-t** <**type**> Anzeigeform des Zeitstempels

**(a)bsolute** Absoluter Unix Zeitstempel in Sekunden seit 1.1.1970

```
(1231777817.918224)  vcan0  4B6  [8] A3 22 A2 7D 84 1F BA 26
(1231777818.118093)  vcan0  3B6  [8] 4F 88 1F 27 11 10 22 37
(1231777818.318225)  vcan0   C2  [2] 14 1C
```

**(d)elta** Zeitdifferenz zwischen den CAN Frames

```
(0.200154)  vcan0  7DF  [8] A3 C2 9D 5F 7A 90 0A 68
(0.200127)  vcan0  28E  [7] DC 72 C1 26 81 29 1A
(0.200181)  vcan0  491  [8] B0 61 1C 72 BE 5E 69 47
```

**(z)ero** Absolute Zeit, die bei Programmstart bei Null beginnt

```
(0.000000)  vcan0  29E  [8] 18 F0 BC 16 26 9E 38 03
(0.196377)  vcan0  216  [8] FC 42 CD 12 BE 35 BE 65
(0.396520)  vcan0  425  [8] B4 B0 8D 55 B0 80 26 18
```

**(A)bsolute w date** Absoluter Unix Zeitstempel seit 1.1.1970 als Datum

```
(2009-01-12 17:40:52.042241)  vcan0   E3  [8] 8B FF 23 4E 58 B9 FF 5E
(2009-01-12 17:40:52.238370)  vcan0  796  [8] DD AE B1 16 9B 06 CB 1C
(2009-01-12 17:40:52.438556)  vcan0  72B  [6] 16 D7 D9 28 65 2F
```

**-c** Zur Unterscheidung der Daten von verschiedenen CAN Interfaces können die jeweiligen ausgegebenen Zeilen automatisch coloriert werden. Dabei können drei Stufen angegeben werden:

```
  (ohne) (1231777818.238370)  vcan0  796  [8] DD AE B1 16 9B 06 CB 1C
      -c (1231777818.238370)  vcan0  796  [8] DD AE B1 16 9B 06 CB 1C
   -c -c (1231777818.238370)  vcan0  796  [8] DD AE B1 16 9B 06 CB 1C
-c -c -c (1231777818.238370)  vcan0  796  [8] DD AE B1 16 9B 06 CB 1C
```

**-i** Ausgabe in binärer Form

```
vcan0 73D [8] 10011010 10111001 00000010 00111011 11101100 01111110 11001000 01101010
```

**-a** Zusätzliche Ausgabe in Textform (soweit darstellbar)

```
(1231780767.914897)  vcan0  549  [8] 5D D7 85 30 CA 60 B4 20   ']..0.'. '
(1231780768.115042)  vcan0  668  [8] 03 1D 1D 56 11 F0 1A 1D   '...V....'
(1231780768.315185)  vcan0  2CF  [6] A6 D8 25 74 49 E3         '..%tI.'
```

**-S** dreht vor der Ausgabe die Byteorder des Datenteiles des CAN Frames und zeigt dieses durch so genannte Backquotes ` vor den einzelnen Datenelementen an. Dieses gilt jeweils auch für die Darstellung in der Textform bzw. in der Binärform.

```
(1231781018.497372)  vcan0  419  [3] 79`25`DB
(1231781018.697545)  vcan0  385  [8] 45`03`04`71`61`33`96`65
(1231781018.897700)  vcan0  55D  [4] 15`F9`D6`B2
```

**-s** <**level**> Reduziert die Ausgabe von **candump** auf eine Animation, die sich mit jedem empfangenen CAN Frame ändert (-s 1) oder deaktiviert die Ausgabe (-s 2)

**-b** <**can**> sendet die empfangenen CAN Frames zusätzlich auf dem CAN Interface <can> aus (die CAN Frames werden dabei lokal zurückgespiegelt)

**-B** <**can**> sendet die empfangenen CAN Frames zusätzlich auf dem CAN Interface <can> aus (die CAN Frames werden dabei nicht lokal zurückgespiegelt)

**-l** Schreibt die empfangenen CAN Frames im Logfile Format im aktuellen Verzeichnis in eine Datei mit aktueller Uhrzeit, z.B. `candump-2009-01-12_184810.log`

**-L** Anstelle der o.g. Möglichkeiten der Ausgabeformatierung kann die Ausgabe hiermit auf das Logfile Format gesetzt werden. Eine solche Ausgabe lässt sich z.B. mit dem Werkzeug netcat (nc(1)) als Stream auf einen entfernten Rechner umleiten und kann dort mit Hilfe des Werkzeugs **canplayer** aus Kapitel C.1.4 wieder auf reale oder virtuelle CAN Bussewiedergegeben werden.

```
(1231789887.926875) vcan0 136#D1A22411087C
(1231789888.127018) vcan0 5D4#8B0B5958
(1231789888.327184) vcan0 56F#6A8D111BDBFF0E52
```

Das Logfile Format ist als reduzierte Textdarstellung ein kompaktes aber noch 'menschenlesbares' Format, das als Bibliotheksfunktionen auf dem SocketCAN SVN vorliegt. Das Komprimieren dieses textbasierten Logfile Formates mit gzip hat sich als effizienter herausgestellt, als ein binäres Logfile Format zu realisieren.

```
<can_id>#{R|data}

can_id can have 3 (standard frame format) or 8 (extended frame format)
       hexadecimal chars

data has 0 to 8 hex-values that can (optionally) be seperated by '.'

Examples:
123# -> standard CAN-Id = 0x123, dlc = 0
12345678# -> extended CAN-Id = 0x12345678, dlc = 0
123#R -> standard CAN-Id = 0x123, dlc = 0, RTR-frame
7A1#r -> standard CAN-Id = 0x7A1, dlc = 0, RTR-frame
123#00 -> standard CAN-Id = 0x123, dlc = 1, data[0] = 0x00
123#1122334455667788 -> standard CAN-Id = 0x123, dlc = 8
123#11.22.33.44.55.66.77.88 -> standard CAN-Id = 0x123, dlc = 8
32345678#112233 -> error frame with CAN_ERR_FLAG (0x2000000) set
```

**-n** <**count**> Beendet das Programm nach der Aussendung von <count> CAN Frames.

**-r** <**size**> Definiert die Größe der Socket-Empfangspuffer.

**-d** Überwacht den möglichen Verlust empfangener CAN Frames, der durch das Scheduling der Anwendung in dem Mehrbenutzerbetriebssystem auftreten kann.

Zu jedem der bis zu 16 angegebenen CAN Interfaces können maximal 30 Komma-separierte Filter angegeben werden, die in den jeweiligen geöffneten RAW-Socket eingetragen werden. Die Beschränkung auf 16 Sockets und jeweils 30 Filter lässt sich im Quelltext von **candump** ändern.

### C.1.2 cansniffer

Der **cansniffer** nutzt den *Broadcast-Manager* Socket und visualisiert Änderungen im Datenbereich von CAN Frames. Dabei können erkannte Unterschiede interaktiv unterdrückt werden und die Daten sowohl in hexadezimaler, binärer oder textueller Form dargestellt werden.

Mit `cansniffer -?` oder ohne Parameter aufgerufen, erscheint folgender Hilfetext.

```
Usage: cansniffer [can-interface]
Options: -m <mask>  (initial FILTER default 0x00000000)
         -v <value> (initial FILTER default 0x00000000)
         -q         (quiet - all IDs deactivated)
         -r <name>  (read sniffset.name from file)
         -b         (start with binary mode)
         -B         (start with binary mode with gap - exceeds 80 chars!)
         -c         (color changes)
         -f         (filter on CAN-ID only)
         -t <time>  (timeout for ID display [x100ms] default: 50, 0 = OFF)
         -h <time>  (hold marker on changes [x100ms] default: 10)
         -l <time>  (loop time (display) [x100ms] default: 2)
Use interface name 'any' to receive from all can-interfaces

commands that can be entered at runtime:

q<ENTER>       - quit
b<ENTER>       - toggle binary / HEX-ASCII output
B<ENTER>       - toggle binary with gap / HEX-ASCII output (exceeds 80 chars!)
c<ENTER>       - toggle color mode
#<ENTER>       - notch currently marked/changed bits (can be used repeatedly)
*<ENTER>       - clear notched marked
rMYNAME<ENTER> - read settings file (filter/notch)
wMYNAME<ENTER> - write settings file (filter/notch)
+FILTER<ENTER> - add CAN-IDs to sniff
-FILTER<ENTER> - remove CAN-IDs to sniff

FILTER can be a single CAN-ID or a CAN-ID/Bitmask:
+1F5<ENTER>    - add CAN-ID 0x1F5
-42E<ENTER>    - remove CAN-ID 0x42E
-42E7FF<ENTER> - remove CAN-ID 0x42E (using Bitmask)
-500700<ENTER> - remove CAN-IDs 0x500 - 0x5FF
+400600<ENTER> - add CAN-IDs 0x400 - 0x5FF
+000000<ENTER> - add all CAN-IDs
-000000<ENTER> - remove all CAN-IDs

if (id & filter) == (sniff-id & filter) the action (+/-) is performed,
which is quite easy when the filter is 000
```

- Die Ausgabe auf dem Terminal wird mit der Häufigkeit aktualisiert, die im Kommandozeilenparameter '-l' vorgegeben wird - standardmäßig 200ms.
- Die Eingabe von Filterbefehlen zur Laufzeit wird aufgrund der Ausgabe der Daten nicht angezeigt.
- Der Zeitstempel gibt die Differenz zum vorherigen Empfangszeitpunkt für die jeweilige CAN-ID an.

Beispiele für **cansniffer** Darstellungen:

```
cansniffer vcan1
```

```
/  time    ID  data ...                    < cansniffer vcan1 # l=2 h=10 t=50 >
0.200197  351  80 AD 2A 2E 0C 75 74 10 ..*..ut.
0.198582  353  00 D8 19 B1 80 19       ......
0.199963  3c3  67 00 00 00 B7 A0 00 F8 g.......
0.606804  3e1  10 CE 16 06 5A 00 82    ....Z..
```

```
cansniffer -c vcan1
```

```
/  time    ID  data ...                    < cansniffer vcan1 # l=2 h=10 t=50 >
0.200197  351  80 AD 2A 2E 0C 75 74 10 ..*..ut.
0.198582  353  00 D8 19 B1 80 19       ......
0.199963  3c3  67 00 00 00 B7 A0 00 F8 g.......
0.606804  3e1  10 CE 16 06 5A 00 82    ....Z..
```

```
cansniffer -b vcan1
```

```
|  time    ID  data ...                    < cansniffer vcan1 # l=2 h=10 t=50 >
0.198766  351  1000000000110101001000010110111000001010011101000111010000010000
0.200346  353  000000000111010000010100101101001000000000011001
0.200711  3c3  0100010100000000000000000000000010110111011000000000000001011010
0.000000  3e1  00010000110011100000111100000111010110110000000010000010
```

```
cansniffer -B -c vcan1
```

```
/  time    ID  data ...                    < cansniffer vcan1 # l=2 h=10 t=50 >
0.199327  351  10000000 10010111 01010001 00100010 00001101 01110011 01110011 00010000
0.200493  353  00000000 01101000 00011111 10110110 10000000 00011001
0.200530  3c3  00100010 00000000 00000000 00000000 10110111 01010000 00000000 10001101
0.201721  3e1  00010000 11001110 00010010 00000101 01011001 00000000 10000010
```

### C.1.3 cansend

Mit **cansend** kann ein einzelnes CAN Frame an ein angegebenes CAN Interface gesendet werden.

Mit `cansend -?` oder ohne Parameter aufgerufen, erscheint folgender Hilfetext.

```
Usage: cansend <device> <can_frame>.
```

Als <can_frame> wird die kompakte Darstellung erwartet, wie sie auch im Logfile Format Verwendung findet:

```
<can_id>#{R|data}

can_id can have 3 (standard frame format) or 8 (extended frame format)
       hexadecimal chars

data has 0 to 8 hex-values that can (optionally) be seperated by '.'

Examples:
123# -> standard CAN-Id = 0x123, dlc = 0
12345678# -> extended CAN-Id = 0x12345678, dlc = 0
123#R -> standard CAN-Id = 0x123, dlc = 0, RTR-frame
7A1#r -> standard CAN-Id = 0x7A1, dlc = 0, RTR-frame
123#00 -> standard CAN-Id = 0x123, dlc = 1, data[0] = 0x00
123#1122334455667788 -> standard CAN-Id = 0x123, dlc = 8
123#11.22.33.44.55.66.77.88 -> standard CAN-Id = 0x123, dlc = 8
32345678#112233 -> error frame with CAN_ERR_FLAG (0x2000000) set
```

Bei einer ungültigen Angabe eines CAN Frames wird eine entsprechende Fehlermeldung ausgegeben:

```
hartko@vwagwolkf320:~> cansend vcan0 1234#ABC

Wrong CAN-frame format!

Try: <can_id>#{R|data}
can_id can have 3 (SFF) or 8 (EFF) hex chars
data has 0 to 8 hex-values that can (optionally) be seperated by '.'

e.g. 5A1#11.2233.44556677.88 / 123#DEADBEEF / 5AA# /
     1F334455#1122334455667788 / 123#R for remote transmission request.
```

Das angegebene Beispiel hat gleich zwei Fehler:

- 1234 als CAN-ID ist nicht 3 oder 8 Zeichen lang
- ABC ist als Datenlänge nicht durch zwei teilbar

### C.1.4 canplayer

Der **canplayer** schreibt eine Log Datei zeitrichtig auf CAN Interfaces, d.h. er ermöglicht die Wiedergabe von aufgezeichneten Daten auf CAN Interfaces. Weil der **canplayer** auch von der Standardeingabe (stdin) lesen kann, muss zur Anzeige des Hilfetextes `canplayer -?` eingegeben werden:

```
Usage: canplayer <options> [interface assignment]*

Options:              -I <infile>  (default stdin)
                      -l <num>     (process input file <num> times)
                                   (Use 'i' for infinite loop - default: 1)
                      -t           (ignore timestamps: send frames immediately)
                      -g <ms>      (gap in milli seconds - default: 1 ms)
                      -s <s>       (skip gaps in timestamps > 's' seconds)
                      -x           (disable local loopback of sent CAN frames)
                      -v           (verbose: print sent CAN frames)

Interface assignment:  0..n assignments like <write-if>=<log-if>
e.g. vcan2=can0 ( send frames received from can0 on vcan2 )
extra hook: stdout=can0 ( print logfile line marked with can0 on stdout )
No assignments => send frames to the interface(s) they had been received from.

Lines in the logfile not beginning with '(' (start of timestamp) are ignored.
```

- Wenn Zuweisungen definiert werden, sind die ursprünglichen CAN Interfaces in ihrer Zuweisung komplett aufgehoben, d.h. es können nicht einzelne Interfaces neu zugewiesen werden und die übrigen ursprünglichen Zuweisungen bleiben erhalten. Wenn Zuweisungen in der Kommandozeile durchgeführt werden, müssen alle gewünschten Zuweisungen eingetragen werden - unter Umständen auch can2=can2
- Mit der Schleifenvariable (loop) '-l <num>' kann die Anzahl angegeben werden, wie häufig die Eingabedatei wiedergegeben werden soll. '-li' bzw. '-l i' gibt die Eingabedatei unendlich häufig wieder. Dieser Parameter kann nicht angewendet werden, wenn die Daten über die Standardeingabe (stdin) gelesen werden.
- Um in eine Log Datei Kommentare einpflegen zu können, werden alle Zeilen ignoriert, die nicht mit einer geöffneten Klammer '(' beginnen.

### C.1.5 cangen

Zu Entwicklungs- und Testzwecken kann mit dem Werkzeug **cangen** CAN Datenverkehr auf einem gegebenen CAN Interface erzeugt werden. Dabei besteht die Möglichkeit sowohl die Inhalte der zu generierenden CAN Daten zu definieren als auch die Zeitabstände mit der die CAN Daten gesendet werden.

Mit `cangen -?` oder ohne Parameter aufgerufen, erscheint folgender Hilfetext.

```
cangen: generate CAN frames

Usage: cangen [options] <CAN interface>
Options: -g <ms>       (gap in milli seconds - default: 200 ms)
         -e            (generate extended frame mode (EFF) CAN frames)
         -I <mode>     (CAN ID generation mode - see below)
         -L <mode>     (CAN data length code (dlc) generation mode - see below)
         -D <mode>     (CAN data (payload) generation mode - see below)
         -p <timeout>  (poll on -ENOBUFS to write frames with <timeout> ms)
         -i            (ignore -ENOBUFS return values on write() syscalls)
         -x            (disable local loopback of generated CAN frames)
         -v            (increment verbose level for printing sent CAN frames)

Generation modes:
'r'        => random values (default)
'i'        => increment values
<hexvalue> => fix value using <hexvalue>

When incrementing the CAN data the data length code minimum is set to 1.
CAN IDs and data content are given and expected in hexadecimal values.

Examples:
cangen vcan0 -g 4 -I 42A -L 1 -D i -v -v   (fixed CAN ID and length, inc. data)
cangen vcan0 -e -L i -v -v -v              (generate EFF frames, incr. length)
cangen vcan0 -D 11223344DEADBEEF -L 8      (fixed CAN data payload and length)
cangen vcan0 -g 0 -i -x                    (full load test ignoring -ENOBUFS)
cangen vcan0 -g 0 -p 10 -x                 (full load test with polling, 10ms timeout)
cangen vcan0                               (my favourite default :)
```

### C.1.6 canbusload

Für eine Bewertung der aktuellen Buslast auf dem CAN Bus kann anhand der empfangenen CAN Frames die Länge der Frames auf dem physikalischen Medium in Bit berechnet werden:

**standard CAN Frames (SFF)** 47 Bit + Länge der Nutzdaten (in Bit)

**extended CAN Frames (EFF)** 67 Bit + Länge der Nutzdaten (in Bit)

Bei Bittiming Parametern, die durch die hardwareunabhängige Konfigurationsschnittestelle für den CAN Controller definiert werden (siehe B.1.4 auf Seite 176) ist es nicht möglich die Bitrate des CAN Busses automatisiert abzufragen. Daher muss die Bitrate für die jeweiligen CAN Interfaces beim Starten von **canbusload** mit angegeben werden. Die Ausgabe der Buslast erfolgt im Abstand von jeweils einer Sekunde und gibt dabei die in dieser Sekunde akkumulierten Werte aus.

Mit `canbusload -?` oder ohne Parameter aufgerufen, erscheint folgender Hilfetext.

```
Usage: canbusload [options] <CAN interface>+
  (use CTRL-C to terminate canbusload)

Options: -t (show current time on the first line)
         -c (colorize lines)
         -b (show bargraph in 5% resolution)
         -r (redraw the terminal - similar to top)

Up to 16 CAN interfaces with mandatory bitrate can be specified on the
commandline in the form: <ifname>@<bitrate>

The bitrate is mandatory as it is needed to know the CAN bus bitrate to
calcultate the bus load percentage based on the received CAN frames.
For each given interface the data is presented in one line which contains:

(interface) (received CAN frames) (used bits total) (used bits for payload)

Example:

user@host:~> canbusload can0@100000 can1@500000 can2@500000 can3@500000 -r -t -b -c

canbusload 2008-05-27 15:18:49
 can0@100000  805  74491  36656  74%  |XXXXXXXXXXXXXX......|
 can1@500000  796  75140  37728  15%  |XXX.................|
 can2@500000    0      0      0   0%  |....................|
 can3@500000   47   4633   2424   0%  |....................|
```

## C.2 Konvertierung von Log-Dateien

Für das Speichern von CAN Datenverkehr in Dateien existieren verschiedene Formate, die von den entsprechenden CAN Werkzeugen unterstützt werden. Neben binären Formaten werden dabei auch Textdateien verwendet, die eine Konvertierung in andere Formate erleichtern.

Werden bei der Entwicklung verschiedene Werkzeuge von verschiedenen Herstellern genutzt, ist eine Konvertierung der Logdateien nötig, um aktuelle Logdateien oder auch eine über längere Zeit aufgebaute Sammlung an Logdateien für verschiedene Werkzeuge nutzbar machen zu können.

Die im folgenden beschriebenen Werkzeuge konvertieren das SocketCAN Logfile Format in das so genannte ASC Format, welches beispielsweise von den Werkzeugen CANoe® und CANalyser® der Firma Vector Informatik verwendet wird.

Die Werkzeuge arbeiten als Grundeinstellung mit den Daten von der Standardeingabe (stdin) und schreiben die Ergebnisse in die Standardausgabe (stdout). Dieses kann beispielsweise dazu genutzt werden mit **candump** Ausgaben im ASC Format zu erzeugen:

```
candump vcan0 vcan1 -L | log2asc vcan0 vcan1
```

### C.2.1 log2long

Überführt eine SocketCAN Logdatei in eine lesbarere Form, wie sie von dem Werkzeug `candump -ta -a` bekannt ist:

```
user@host:~> cat candump-2009-01-13_142924.log
(1231853364.856407) vcan0 4A8#96800400FE00A04C
(1231853364.857559) vcan0 380#2075F99481000000
(1231853364.858760) vcan1 289#32023000
user@host:~> cat candump-2009-01-13_142924.log | log2long
(1231853364.856407)  vcan0  4A8  [8] 96 80 04 00 FE 00 A0 4C   '.......L'
(1231853364.857559)  vcan0  380  [8] 20 75 F9 94 81 00 00 00   ' u......'
(1231853364.858760)  vcan1  289  [4] 32 02 30 00               '2.0.'
```

### C.2.2 asc2log

Konvertiert eine ASC Datei in das Format für eine SocketCAN Logdatei.

`asc2log -?` zeigt einen Hilfetext an:

```
Usage: asc2log
Options: -I <infile>  (default stdin)
         -O <outfile> (default stdout)
```

### C.2.3 log2asc

Konvertiert eine SocketCAN Logdatei in eine ASC Datei.

`log2asc -?` zeigt einen Hilfetext an:

```
Usage: log2asc [can-interfaces]
Options: -I <infile>  (default stdin)
         -O <outfile> (default stdout)
         -4 (reduce decimal place to 4 digits)
         -n (set newline to cr/lf - default lf)
```

Die CAN Interfaces, deren Daten konvertiert werden sollen, sind auf der Kommandozeile anzugeben. Eine Konvertierung von Error Frames (z.B. Bus Error Events) wird nicht durchgeführt. Die Datumsangaben in der ASC Darstellung werden aus dem Unix Zeitstempel im SocketCAN Logfile errechnet.

```
user@host:~> cat candump-2009-01-13_142924.log
(1231853364.856407) vcan0 4A8#96800400FE00A04C
(1231853364.857559) vcan0 380#2075F99481000000
(1231853364.858760) vcan1 289#32023000
user@host:~> cat candump-2009-01-13_142924.log | log2asc vcan0 vcan1
date Tue Jan 13 14:29:24 2009
base hex  timestamps absolute
no internal events logged
   0.000000 1  4A8             Rx   d 8 96 80 04 00 FE 00 A0 4C
   0.001152 1  380             Rx   d 8 20 75 F9 94 81 00 00 00
   0.002353 2  289             Rx   d 4 32 02 30 00
user@host:~> cat candump-2009-01-13_142924.log | log2asc -4 vcan0 vcan1
date Tue Jan 13 14:29:24 2009
base hex  timestamps absolute
no internal events logged
   0.0000 1  4A8             Rx   d 8 96 80 04 00 FE 00 A0 4C
   0.0011 1  380             Rx   d 8 20 75 F9 94 81 00 00 00
   0.0023 2  289             Rx   d 4 32 02 30 00
user@host:~> cat candump-2009-01-13_142924.log | log2asc -4 vcan1
date Tue Jan 13 14:29:24 2009
base hex  timestamps absolute
no internal events logged
   0.0023 1  289             Rx   d 4 32 02 30 00
```

## C.3 CAN ISO-TP Transportprotokoll

Für die im Folgenden beschriebenen Werkzeuge ist ein Basiswissen über das CAN ISO-TP Protokoll (ISO15765-2) [27] erforderlich. Es wird ein gemeinsames Beispiel einer einzelnen Datenübertragung gezeigt, dass die Werkzeuge **isotpsend**, **isotprecv**, **isotpdump** und **isotpsniffer** im Verbund darstellt. Die Eingabe und Ausgabe der Daten auf der Standardeingabe und Standardausgabe sollen eine einfache Benutzung der Werkzeuge für erste Tests ermöglichen. Für die Realisierung einer ISO-TP Anwendung ist eine Implementierung in der Programmiersprache 'C' oder 'C++' zu empfehlen, wobei der Quelltext der gezeigten Werkzeuge eine Ausgangsbasis darstellen können.

Das gezeigte Beispiel demonstiert auch das Auffüllen von ISO-TP CAN Nachrichten - hier mit dem Wert 0x42. Das Füllbyte für eine Datenrichtung ist dabei immer identisch und kann von beiden Kommunikationspartnern auf Korrekheit geprüft werden.

- Der Sender der Nachricht überprüft dabei die Korrektheit der vom ihm empfangenen FlowControll (FC) Nachrichten (siehe rote Füllbytes)
- Der Empfänger der Nachricht überprüft dabei die Korrektheit der vom ihm empfangenen Nutzdaten Nachrichten (SF/CF) (siehe blaue Füllbytes)

Dieser Ausschnitt zeigt das folgende gemeinsame Beispiel als Ausgabe von **candump**:

```
user@host:~> candump -td vcan0
 (0.000000)  vcan0  321  [8] 10 0F 11 22 33 44 55 66
 (0.000077)  vcan0  123  [8] 30 00 02 42 42 42 42 42
 (0.002090)  vcan0  321  [8] 21 77 88 99 AA BB CC DD
 (0.002037)  vcan0  321  [8] 22 EE FF 42 42 42 42 42
```

## C.3.1 isotpsend

Liest aus der Standardeingabe (stdin) eine bis zu 4095 Zeichen lange Nachricht als Hexadezimalwerte ein und sendet diese als ISO-TP PDU gemäß den angegebenen Kommandozeilen Parametern auf den CAN Bus.

```
Usage: isotpsend [options] <CAN interface>
Options: -s <can_id>  (source can_id. Use 8 digits for extended IDs)
         -d <can_id>  (destination can_id. Use 8 digits for extended IDs)
         -x <addr>    (extended addressing mode. Use 'any' for all addresses)
         -p <byte>    (set and enable padding byte)
         -P <mode>    (check padding in FC. (l)ength (c)ontent (a)ll)
         -t <time ns> (transmit time in nanosecs)

CAN IDs and addresses are given and expected in hexadecimal values.
The pdu data is expected on STDIN in space separated ASCII hex values.
```

Beispiel für das Versenden einer Nachricht aus 15 Bytes:

```
A="11 22 33 44 55 66 77 88 99 AA BB CC DD EE FF"
echo $A | isotpsend -s 321 -d 123 -p 42 -Pa vcan0
```

## C.3.2 isotprecv

Liest gemäß den angegebenen Kommandozeilen Parametern eine ISO-TP PDU vom CAN Bus und gibt diese als Hexadezimalwerte auf der Standardausgabe (stdout) aus.

```
Usage: isotprecv [options] <CAN interface>
Options: -s <can_id>  (source can_id. Use 8 digits for extended IDs)
         -d <can_id>  (destination can_id. Use 8 digits for extended IDs)
         -x <addr>    (extended addressing mode.)
         -p <byte>    (set and enable padding byte)
         -P <mode>    (check padding in SF/CF. (l)ength (c)ontent (a)ll)
         -b <bs>      (blocksize. 0 = off)
         -m <val>     (STmin in ms/ns. See spec.)
         -w <num>     (max. wait frame transmissions.)
         -l           (loop: do not exit after pdu receiption.)

CAN IDs and addresses are given and expected in hexadecimal values.
The pdu data is written on STDOUT in space separated ASCII hex values.
```

Beispiel für den Empfang der oben versendeten Nachricht:

```
user@host:~> isotprecv -s 123 -d 321 -p 42 -Pa -l vcan0 -m 02
11 22 33 44 55 66 77 88 99 AA BB CC DD EE FF
```

### C.3.3 isotpdump

**isotpdump** nutzt wie das Werkzeug **candump** aus Kapitel C.1.1 den RAW Socket und nicht den ISO-TP Socket. Dieses Werkzeug zeigt die ISO-TP CAN Nachrichten an und dekodiert und visualisiert dabei die Protokollinformationen. Die empfangenen Daten werden aber nicht gemäß der Spezifikation bearbeitet.

```
Usage: isotpdump [options] <CAN interface>
Options: -s <can_id> (source can_id. Use 8 digits for extended IDs)
         -d <can_id> (destination can_id. Use 8 digits for extended IDs)
         -x <addr>   (extended addressing mode. Use 'any' for all addresses)
         -c          (color mode)
         -a          (print data also in ASCII-chars)
         -t <type>   (timestamp: (a)bsolute/(d)elta/(z)ero/(A)bsolute w date)

CAN IDs and addresses are given and expected in hexadecimal values.
```

Beispiel:

```
user@host:~> isotpdump -s 123 -d 321 -c vcan0
  vcan0  321  [8]  [FF] ln: 15   data: 11 22 33 44 55 66
  vcan0  123  [8]  [FC] FC: 0 = CTS # BS: 0 = off # STmin: 0x02 = 2 ms
  vcan0  321  [8]  [CF] sn: 1    data: 77 88 99 AA BB CC DD
  vcan0  321  [8]  [CF] sn: 2    data: EE FF 42 42 42 42 42
```

Zum Vergleich dazu die Ausgabe von **candump** (nachträglich koloriert):

```
user@host:~> candump -td vcan0
 (0.000000)  vcan0  321  [8] 10 0F 11 22 33 44 55 66
 (0.000077)  vcan0  123  [8] 30 00 02 42 42 42 42 42
 (0.002090)  vcan0  321  [8] 21 77 88 99 AA BB CC DD
 (0.002037)  vcan0  321  [8] 22 EE FF 42 42 42 42 42
```

### C.3.4 isotpsniffer

Der **isotpsniffer** nutzt zwei ISO-TP Sockets (für zwei Datenrichtungen) mit der Option, den Datenstrom nur mitzulesen (CAN_ISOTP_LISTEN_MODE). Dabei werden die empfangenen Informationen gemäß der ISO-TP Spezifikation zusammengesetzt und angezeigt ohne das die üblichen Bestätigungsnachrichten gesendet werden.

```
Usage: isotpsniffer [options] <CAN interface>
Options: -s <can_id> (source can_id. Use 8 digits for extended IDs)
         -d <can_id> (destination can_id. Use 8 digits for extended IDs)
         -x <addr>   (extended addressing mode.)
         -c          (color mode)
         -t <type>   (timestamp: (a)bsolute/(d)elta/(z)ero/(A)bsolute w date)
         -f <format> (1 = HEX, 2 = ASCII, 3 = HEX & ASCII - default: 3)
         -h <len>    (head: print only first <len> bytes)

CAN IDs and addresses are given and expected in hexadecimal values.
```

Beipiel:

```
user@host:~> isotpsniffer -s 123 -d 321 -c vcan0
  vcan0  321  [15]  11 22 33 44 55 66 77 88 99 AA BB CC DD EE FF  - '."3DUfw........'
```

Zum Vergleich dazu die Ausgabe von **isotpdump** und **candump** (koloriert):

```
user@host:~> isotpdump -s 123 -d 321 -c vcan0
  vcan0  321  [8]  [FF] ln: 15   data: 11 22 33 44 55 66
  vcan0  123  [8]  [FC] FC: 0 = CTS # BS: 0 = off # STmin: 0x02 = 2 ms
  vcan0  321  [8]  [CF] sn: 1    data: 77 88 99 AA BB CC DD
  vcan0  321  [8]  [CF] sn: 2    data: EE FF 42 42 42 42 42
```

```
user@host:~> candump -td vcan0
 (0.000000)  vcan0  321  [8] 10 0F 11 22 33 44 55 66
 (0.000077)  vcan0  123  [8] 30 00 02 42 42 42 42 42
 (0.002090)  vcan0  321  [8] 21 77 88 99 AA BB CC DD
 (0.002037)  vcan0  321  [8] 22 EE FF 42 42 42 42 42
```

### C.3.5 isotptun (Internet Protocol over CAN)

Mit [4] und [7] existieren verschiedene Konzepte der Abbildung der Internetprotokoll-kommunikation auf die Adressierungsmöglichkeiten des Controller Area Networks. Der vom Autor in [14] dargestellte und bewertete Ansatz verwendet die unter Linux vorhandenen, so genannten Internetprotokoll-Tunnel zusammen mit der in dieser Arbeit dargestellten Realisierung des ISO 15765-2 Protokolls für das CAN Subsystem.

Der dafür im Linux Betriebssystem genutzte Tunnel Treiber ermöglicht die Simulation eines Netzwerkgerätes durch eine Anwendung (im Anwendungskontext). Nachdem eine solche Anwendung eine besondere Gerätedatei `/dev/net/tun` geöffnet hat, kann sie mit einem Systemaufruf (mit Administrator / root Rechten) ein Netzwerkgerät erzeugen:

```
int t;
struct ifreq ifr;

t = open("/dev/net/tun", O_RDWR);

memset(&ifr, 0, sizeof(ifr));
ifr.ifr_flags = IFF_TUN | IFF_NO_PI;
strncpy(ifr.ifr_name, "ctun%d", IFNAMSIZ);

ioctl(t, TUNSETIFF, (void *) &ifr);

/* now we have a ctun0 (or ctun1 or ...) netdevice connected to filedescriptor t */
```

Die Anwendung kann nun Daten in die Datei 't' schreiben, die sich für den Nutzer des Netzwerkgerätes 'ctun0' als empfangenes Internet Protokoll Paket darstellen. Umgekehrt werden an das Netzwerkgerät 'ctun0' gesendete Internet Protokoll Pakete der Anwendung verfügbar gemacht, indem sie von der Datei 't' Daten einliest.

Das ISO-TP Protokoll ermöglicht PDUs mit einer Länge von bis zu 4095 Bytes segmentiert auf dem CAN Bus zu übertragen. Da ist es naheliegend ein Ethernetframe mit einer Maximum Transmission Unit (MTU) von 1500 Bytes versuchsweise in ISO-TP PDUs zu verpacken und auf den CAN Bus zu übertragen. Der im folgenden beschriebene IP-over-ISOTP Tunnel erzeugt ein Point-to-Point Netzwerkgerät 'ctunX' und ermöglicht auf einfache Weise eine IP Kommunikation über den CAN Bus.

Dazu wird das im Folgenden beschriebene Werkzeug **isotptun** auf beiden Rechnern eingesetzt, die über den CAN Bus miteinander verbunden sind. Bei der Kommunikation über ISO 15765-2 werden vom ISO-TP Treiber bis zu 585 CAN Frames en-bloc in die Sendepuffer der verwendeten CAN Netzwerkgeräte geschrieben. Zur Sicherstellung der Kommunikation ist die Größe der Sendewarteschlange entsprechend anzupassen.

```
Usage: isotptun [options] <CAN interface>

This program creates a Linux tunnel netdevice 'ctunX' and transfers the
ethernet frames inside ISO15765-2 (unreliable) datagrams on CAN.

Options: -s <can_id>  (source can_id. Use 8 digits for extended IDs)
         -d <can_id>  (destination can_id. Use 8 digits for extended IDs)
         -x <addr>    (extended addressing mode.)
         -p <byte>    (padding byte rx path)
         -q <byte>    (padding byte tx path)
         -P <mode>    (check padding. (l)ength (c)ontent (a)ll)
         -t <time ns> (transmit time in nanosecs)
         -b <bs>      (blocksize. 0 = off)
         -m <val>     (STmin in ms/ns. See spec.)
         -w <num>     (max. wait frame transmissions.)
         -h           (half duplex mode.)
         -v           (verbose mode. Print symbols for tunneled msgs.)

CAN IDs and addresses are given and expected in hexadecimal values.
Use e.g. 'ifconfig ctun0 123.123.123.1 pointopoint 123.123.123.2 up'
to create a point-to-point IP connection on CAN.
```

Beispiel zur Realisierung einer Internetprotokollkommunikation über CAN:

Rechner1 ist mit 'can1' über den CAN Bus an den 'can2' von Rechner2 angeschlossen und die CAN Verbindung ist etabliert und getestet (z.B. mit **cansend** und **candump**). Auf beiden Rechnern werden zwei Terminals für den Benutzer 'root' geöffnet, weil **isotptun** das Terminal während der Ausführung des Tunnelprogramms blockiert.

Rechner1:

```
isotptun -s 123 -d 321 -v can1
ifconfig ctun0 123.123.123.1 pointopoint 123.123.123.2 up
```

Rechner2:

```
isotptun -s 321 -d 123 -v can2
ifconfig ctun0 123.123.123.2 pointopoint 123.123.123.1 up
```

Nun ist es möglich auf Rechner2 die Internet Protokoll Verbindung zu Rechner1 zu nutzen und sich beispielsweise einzuloggen oder Daten zu senden:

```
user@rechner2:~> ping 123.123.123.1
user@rechner2:~> ssh user@123.123.123.1
user@rechner2:~> scp user@123.123.123.1:myfile.tar.gz .
```