Projekt Systementwicklung
WS 2010

Dokumentation CAN-Bus-Gruppe

Christian Steiger

Christian Scheuermann

Inhaltsverzeichnis

Allgemeines

Was ist CAN-Bus

CAN-Bus (Controller Area Network) ist ein asynchroner, serieller Kommunikationsbus, der von Bosch für den Einsatz in Automobilen entwickelt wurde [1],[2]. Er arbeitet mit nur zwei Leitungen (CAN-High und CAN-Low) mit einer Differenzspannungsankopplung und funktioniert prinzipiell mit auch nur einer Leitung, solange die Null-Leitung verbunden ist. Auf dem Bus existiert kein Master. Alle angeschlossenen Teilnehmer funktionieren nach dem CSMA/CA-Verfahren (Carrier Sense Multiplie Access/Collision Avoiding). Dabei lauscht jeder Teilnehmer auf dem Bus, um eine eventuell vorhandene Belegung zu erkennen. Ist der Bus frei, fängt ein Teilnehmer an zu senden. Dabei kommt nach dem Startbit zuerst ein 11-Bit Identifier (Base-Frame). Fangen nun zwei oder mehr Teilnehmer gleichzeitig an zu senden, so setzt sich der Sender mit der niedrigeren ID durch, d.h. auf dem Bus ist der Low-Pegel dominant, der High-Pegel rezessiv (eine Logische 0 überschreibt eine logische 1). Des Weiteren unterstützt der CAN-Bus in der Version 2.0B auch sogenannte Extended-Frame, welcher einen 29 Bit Identifier zulässt. Sowohl das Base-Frame, als auch das Extended-Frame kann als Remote-Frame gesendet werden, wobei hier der Sender eine Nachricht absetzt, dessen Daten von einem Empfänger ausgefüllt werden.

In einem einzigen Datenpaket können zwischen 0 und 8 Byte Nutzdaten übertragen werden. Zur Übertragungssicherung wird eine zyklische Redundanzprüfung eingesetzt.

Beim CAN-Bus ist außerdem noch zu beachten, dass eine ID nicht einen bestimmten Absender kennzeichnet, sondern ein Repräsentant für den Inhalt ist. Ein und das selbe Steuergerät kann unter unterschiedlichen Nachricht-IDs Daten versenden

CAN-BUS beim Opel Astra

Im Opel Astra der Hochschule Darmstadt ist ein CAN-Bus mit einer Busgeschwindigkeit von 500 kbit/s. Die Busgeschwindigkeit wurde im Vorfeld mit Hilfe eines Oszilloskops ermittelt. Näheres hierzu im Kapitel „Aufzeichnungen und Messungen im Fahrzeug“. Außerdem versendet das Fahrzeug von sich heraus ausschließlich Nachrichten im Base-Frame-Format. Es ist allerdings nicht auszuschließen, dass über Remote-Nachrichten auch Geräte im Extended-Frame-Format angefordert werden können.

Entwicklungsumgebung

Als Entwicklungsumgebung setzten wir Eclipse IDE for C/C++ Developers (Helios Release, Build 20100617-1415) auf einem Ubuntu 10.04 ein. Zu Beginn des Projektes arbeiten wir noch mit automatisch generierten Makefiles. Im Laufe des Projektes stellte sich dies jedoch als großes Problem heraus und wir stiegen auf selbst geschriebene Makefiles um. Damit ist es nun bei fast allen unserer Programme möglich, diese ohne Eclipse zu erstellen, indem in der Konsole einfach „make“ aufgerufen wird. Speziell in Verbindung mit Perforce bot dies gewisse Vorteile, da wir für unsere Programme an fast allen stellen Shared Librarys (Ncurses, LibPCan, LibcanIO) benötigten. Diese gingen aber bei jedem Submit in Perforce und anschließendem Herunterladen auf einem anderen PC verloren, so dass sie jedes mal neu bei den Linker-Optionen eingestellt werden mussten.

1 http://de.wikipedia.org/wiki/Controller_Area_Network

2 CAN-Bus 2.0B Specification (BOSCH)

Library NCURSES

In manchen Programmen verwenden wir die NCURSES-Bibliothek. Diese ist allerdings nicht bei der Grundinstallation von Ubuntu dabei und muss somit nachinstalliert werden

```
sudo apt-get install libncurses-dev.
```

Anschließend steht die Bibliothek mit der Compileroption „-lncurses“ zur Verfügung.

NCURSES ist eine Library, die es ermöglicht, an eine beliebige Stelle in der Konsole den Cursor zu setzen und dort auch Daten zu schreiben – was z. B. bei unserem PeakCANInterpreter sehr interessant ist, da dieser permanent die aktuellen Daten des Fahrzeugs anzeigt und somit nicht immer vorher der Bildschirm gelöst werden muss. Auch beim CAN-Bomber war dieses gezielte Schreiben von Vorteil.

Perforce

Perforce als Versionierungstool machte im Rahmen durchaus Sinn. Wir hatten beide Erfahrungen mit anderen Systemen (SVN), aber Perforce stellte uns doch vor gewisse Probleme. Zu aller erst ist zu erwähnen, dass der Begriff „Workspace“ ein paar Unklarheiten aufwarf: Workspace in Eclipse ist was anderes als in Perforce – und diese beiden unter einen Hut zu bekommen, war erst einmal eine Herausforderung. Das Eclipse-Plugin (P4WSAD, zu beziehen unter www.perforce.com) unterstützte uns zwar dabei, aber bis es dann endlich funktionierte, mussten wir beide mehrfach unseren kompletten Workspace (Eclipse und Perforce) löschen. Nachdem Perforce aber endlich funktionierte und wir uns an den P4V-Client gewöhnt hatten, ging die Arbeit relativ einfach von der Hand (auch wenn der Schreibschutz bei kopierten Dateien manchmal schon ein wenig nervig war). Im Allgemeinen unterstützt ein (richtig genutztes) Versionierungstool bei der Entwicklung, speziell wenn einmal eine neue Version nicht funktioniert und ein Vergleich mit der Vorgängerversion einem seine genauen Änderungen aufzeigt. Die häufig gehörten Probleme und Beschwerden anderer Gruppen konnten wir nicht teilen.

Position im Gesamtprojekt

Im Gesamtprojekt ist der CAN-Bus im Fahrzeug angesiedelt. Die Anbindung an das Fahrzeug erfolgt direkt über den ELM als Protokollinterpreter oder den CAN-Bus. Auf die Auswertung der CAN-Daten (und auch der ELM-Daten) setzt die Auswertung und Übertragung der Daten auf. Ohne eine Anbindung an den CAN-Bus und den ELM wäre eine Durchführung des Projektes in diesem Stil nicht möglich gewesen. Somit kann man schon sagen, dass der CAN-Bus eine sehr zentrale Rolle im Gesamtprojekt inne hat. Dies ist allerdings von allen Gruppen zu sagen: in einem größeren Projekt ist das Gesamtergebnis abhängig von allen Einzelteilen. Anders gesagt: ein Gesamtsystem ist nur so stark, wie sein schwächstes Glied.

Unser Ziel war also, eine möglichst vollständig getestete Schnittstelle zur Verfügung zu stellen, um den über uns liegenden Gruppen eine funktionierende Klasse zur Verfügung zu stellen.

Eingesetze Hardware

PCAN-USB

Für unser Projekt bekamen wir zwei PCAN-Usb-Adapter der Firma Peak-System[1] zur Verfügung gestellt, für die es sowohl unter Windows als auch für Linux Treiber gibt, wobei hier speziell der Linux Treiber von Interesse war, da das Projekt vollständig auf Linux aufsetzen sollte.

Die Hauptfeatures des Adapters sind (Bild und Beschreibung der Herstellerseite entnommen):

- Übertragungsraten bis zu 1 Mbit/s.
- Erfüllt die CAN-Spezifikationen 2.0A (11-Bit-ID) und 2.0B (29-Bit-ID).
- Anschluss an CAN-Bus über D-Sub, 9-polig (nach CiA® 102).
- NXP CAN-Controller SJA1000 mit 16 MHz Taktfrequenz.
- NXP CAN-Transceiver PCA82C251.
- 5-Volt-Versorgung am CAN-Anschluss durch Lötjumper zuschaltbar.
- Spannungsversorgung über USB.

1 http://www.peak-system.com/

Treiberinstallation unter Linux

Die Treiberinstallation gestaltet sich als etwas fummelig, funktioniert aber nach abarbeiten der Schritte unter einem Ubuntu 10.04 einwandfrei. Andere Linux-Betriebssysteme konnten wir aufgrund von Zeitmangel nicht testen, die Schritte sollten jedoch bis auf das Installieren der benötigten Pakete die gleichen sein.

Treiber downloaden und entpacken

Der Treiber befindet auf der Herstellerseite, *http://www.peak-system.com*, und ist unter „Support“ => „PCAN-Linux-Gerätetreiber“ => „Driver Download“ als .tar.gz Archiv zu finden. Nachdem sich selbiges auf der Festplatte befindet, entpackt man es und wechselt danach in das dadurch entstandene Verzeichnis mit diesen Befehlen:

```
tar xvf peak-linux-driver.6.20.tar.gz
cd peak-linux-driver.6.20
```

Hinweis: Die Versionsnummer kann sich im laufe der Zeit ändern. Um sich Tipperei zu sparen kann man nachdem man die ersten Buchstaben (peak) eingegeben hat den Pfad oder den Dateinamen mit Tab vervollständigen.

Weitere benötigte Dinge

Für das Kompilieren benötigt der Treiber die aktuellen Kernel-Header und die Entwicklerversion der Bibliothek popt. Unter Ubuntu können diese Pakete per Paketverwaltung hinzugefügt werden, die Installation gestaltet sich damit entsprechend einfach:

```
sudo apt-get install linux-headers-`uname -r`
sudo apt-get install libpopt-dev
```

Hinweis: `uname -r` besorgt sich die Version des aktuell installierten Kernels.

Treiber kompilieren

Der PCAN-Treiber ist in zwei verschiedenen Versionen vorhanden: *netdev* und *chardev*. *chardev* ist dabei der ältere Treiber der direkt mit dem Adapter kommuniziert, und der auch für die mit dem PCAN-Adapter mitgelieferten Beispielprogramme nötig ist. *netdev* ist eine neue Version die per Standard aktiv ist und den Adapter in ein *SocketCAN*[2] Interface einkoppelt das die Adapter-Funktionalität per Sockets zur Verfügung stellt. Im Gegensatz zu *chardev* ermöglicht *netdev* eine Mehrfachbelegung eines Interfaces, es können also mehrere Programme gleichzeitig lesen und schreiben während bei *chardev* nur eine einzige Applikation zugriff auf den Adapter hat.

Hinweis: Wir (Gruppe CAN-Anbindung) arbeiten mit *netdev*, für unsere Programme ist daher die *netdev* Installation erforderlich.

Anweisungen für netdev

Für die *netdev*-Installation reichen die üblichen drei Befehle aus:

```
sudo make clean
sudo make
sudo make install
```

2 http://en.wikipedia.org/wiki/Socketcan oder http://developer.berlios.de/projects/socketcan/

Weitere Schritte für netdev

Mit einem Editor mit Root-Rechten (vi/nano/gedit/etc) die Datei „*/etc/modprobe.d/pcan*“ öffnen, danach die folgenden Zeilen in die Datei kopieren:

```
alias net-pf-29 can
alias can-proto-1 can-raw
alias can-proto-2 can-bcm
stats_timer=0
alias vcan0 vcan
alias vcan1 vcan
alias vcan2 vcan
alias vcan3 vcan
alias can0 pcan
alias can1 pcan
alias can2 pcan
alias can3 pcan
```

Achtung: Ohne diese Zeilen kann der PCAN-Adapter nicht korrekt eingebunden werden!

Tipp: Der gleiche Text findet sich mit einigen zusätzlichen Kommentaren und Optionen auch in der Dokumentation der mit dem Treiber mitgeliefert wird im Ordner „Documentation“ in der Datei „PCAN Driver for Linux_eng.pdf“ auf Seite 21.

Anweisungen für chardev

Bei *chardev* muss dem *make*-Befehl ein Parameter mitgegeben werden:

```
sudo make clean
sudo make NET=NO_NETDEV_SUPPORT
sudo make install
```

Optional

Dieser Befehl dient dazu Warnmeldungen beim aktivieren des Kernelmodules zu verhindern:

```
sudo mv /etc/modprobe.d/pcan /etc/modprobe.d/pcan.conf
```

Kernelmodul aktivieren

Das frisch kompilierte Kernelmodul wird mit dem folgenden Befehl aktiviert:

```
sudo modprobe pcan
```

Achtung: Der Befehl muss nach jedem Reboot erneut ausgeführt werden!

Soll das Modul bei jedem Systemstart automatisch geladen werden ist der folgende Befehl auszuführen:

```
echo 'pcan' | sudo tee -a /etc/modules
```

Als alternative kann */etc/modules* auch mit einem Editor mit Root-Rechten (vi/nano/gedit/etc) editiert werden. *pcan* muss dabei am ende der Datei in einer eigenen Zeile stehen.

Interface aktivieren (nur netdev)

Nach dem Einstecken des Adapter muss folgender Befehl zusätzlich eingeben werden:

```
sudo ifconfig can0 up
```

Falls mehr als ein Adapter anwesend ist, ist der zweite unter can1 erreichbar, der dritte unter can2, und so weiter, für jeden Adapter ist dabei ein eigener *ifconfig*-Aufruf nötig.

Achtung: Der Befehl muss jedes mal wenn ein Adapter eingesteckt wird erneut ausgeführt werden!

Interface automatisch aktiveren

Um die Adapter beim einstecken automatisch zu aktivieren muss die Datei „*/etc/udev/rules.d/45-pcan.rules*“ mit einem Editor mit Root-Rechten (vi/nano/gedit/etc) geöffnet werden. Dort finden sich mehrere Zeilen, die Zeile mit dem Anfang KERNEL=="pcanusb*" suchen und an dessen ende , RUN+="/sbin/ifconfig can%n up" anhängen.

Das Ergebnis sollte so aussehen:

```
# PCAN devices:
#
# special udev rules for Peak System PCAN-USB devices
# www.peak-system.com
#
# klaus.hitschler@gmx.de
#
# $Id:$
#
ACTION!="add", GOTO="pcan_udev_end"

KERNEL=="pcanusb*",    SYMLINK+="pcan%m", MODE="0666", RUN+="/sbin/ifconfig can%n up"
KERNEL=="pcanpci*",    SYMLINK+="pcan%m", MODE="0666"
KERNEL=="pcanpccard*", SYMLINK+="pcan%m", MODE="0666"
KERNEL=="pcanisa*",    SYMLINK+="pcan%m", MODE="0666"
KERNEL=="pcanepp*",    SYMLINK+="pcan%m", MODE="0666"
KERNEL=="pcansp*",     SYMLINK+="pcan%m", MODE="0666"

LABEL="pcan_udev_end"
```

Damit die modifizierte Regel ohne Neustart wirksam wird muss der *udev*-Dienst neugestartet werden, dazu kann der folgende Befehl benutzt werden:

```
sudo service udev reload
```

Ab jetzt werden die Interfaces spätestens nach nach erneutem Einstecken der Adapter automatisch aktiv.

VCAN

VCAN steht für Virtuelles CAN, und ist eine Simulationsschicht die wie der *netdev*-Treiber auch auf SocketCAN aufsetzt und daher über die gleichen Befehle angesprochen werden kann. Das Interface verhält sich dabei genauso wie ein echter CAN-Bus und ist daher ein ausgesprochen nützliches Helferlein für die verschiedensten Testfälle, die richtigen (und in ihrer Zahl begrenzten) Adapter werden damit für andere Aufgaben frei.

Alles was dazu nötig ist findet sich inzwischen in den aktuelleren Linux-Distributionen wie z.B. Ubuntu 10.04, der PCAN-Treiber ist dazu nicht notwendig.

Installation

Für die Installation sind gerade einmal drei Befehle nötig:

```
sudo modprobe vcan
sudo ip link add dev vcan0 type vcan
sudo ifconfig vcan0 up
```

Danach ist das Interface einsatzbereit und unter vcan0 ansprechbar.

Achtung: Nach einem Systemneustart müssen die Befehle erneut eingegeben werden!

Achtung: Software ist nicht gleich Hardware! Das sollte man sich immer vor Augen halten.

VCAN-Interface bei Systemstart automatisch laden

VCAN kann bei jedem Systemstart automatisch geladen werden. Dazu sind lediglich zwei Schritte nötig, die jeweils das Kernelmodul laden und sich um das einrichten des Interface kümmern.

Kernelmodul

Um dass Kernelmodul automatisch zu laden ist der folgende Befehl auszuführen:

```
echo 'vcan' | sudo tee -a /etc/modules
```

Als Alternative kann /etc/modules auch mit einem Editor mit Root-Rechten (vi/nano/gedit/etc) editiert werden. vcan muss dabei am ende der Datei in einer eigenen Zeile stehen.

Interface

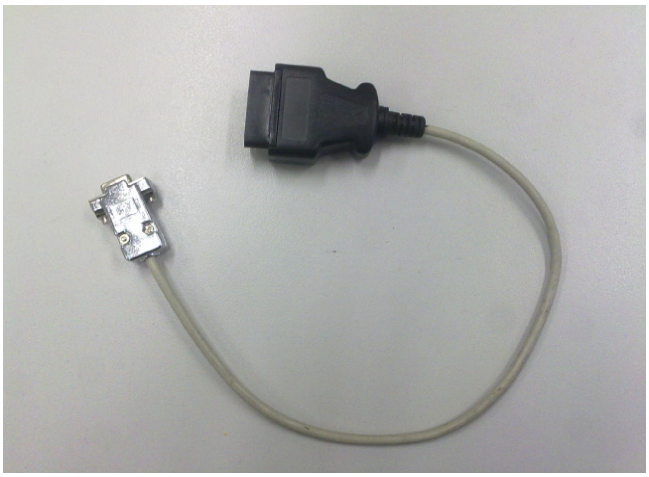
Um das Interface automatisch zu erzeugen und aktivieren muss die Datei „*/etc/rc.local*“ mit einem Editor mit Root-Rechten (vi/nano/gedit/etc) geöffnet werden. Dort werden dann die Befehle „*ip link add dev vcan0 type vcan*“ und „*ifconfig vcan0 up*“ vor der Zeile „*exit 0*“ hinzugefügt, das ganze sollte dann in etwa so aussehen (#-Kommentare in der Datei wegen Irrelevanz entfernt):

```
#!/bin/sh -e

ip link add dev vcan0 type vcan
ifconfig vcan0 up

exit 0
```

Ab dem nächsten Neustart ist das VCAN-Interface jetzt automatisch aktiv.

OBD-DSUB-Kabel

Am Anfang des Projektes setzten wir ein einfaches OBD-CAN-Kabel ein. Dies bestand aus einem OBD-Stecker und einer 9-Pol DSub Buchse, welches wir am ersten Tag abends noch bauten um möglichst schnell Daten vom Fahrzeug zu erhalten.

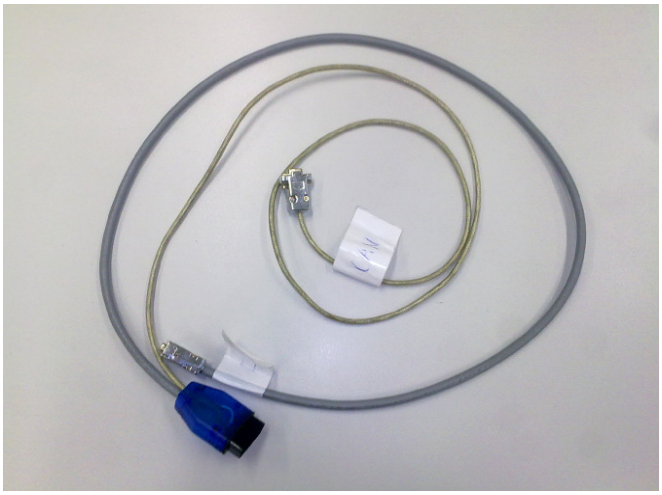


9 Pol DSub	Funktion	OBD-Stecker
2	CAN-High	14
3	Signal Masse	5
7	CAN-Low	6

Belegung OBD-DSub-Kabel

OBD-CAN-ELM-Kabel

In einem fortgeschrittenen Stadium des Projektes wurde es nötig, mit der OBD-Gruppe gemeinsam auf das Fahrzeug zugreifen zu können. Dafür benötigten wir also ein Y-Kabel, dass sowohl einen Anschluss für den ELM und den CAN-Adapter zur Verfügung stellt. Somit löteten wir Anfang der zweiten Woche ein Kabel, dass beide Anschlüsse bot und testeten dies auch kurze Zeit später erfolgreich. Mit diesem Kabel war es auch erstmals möglich, Nachrichten, die der ELM auf dem CAN-Bus absendete mitzulauschen und zu protokollieren, so dass wir später an eine OBD-Over-CAN Schnittstelle gehen konnten (mehr dazu im gesonderten Abschnitt)



9 Pol DSub CAN	9 Pol DSub ELM	Funktion	OBD-Stecker
	7	J1850 Bus+	2
	2	Fahrzeug Masse	4
3	1	Signal Masse	5
7	3	CAN-High	6
	4	ISO 9141-2 K Ausgang	7
	6	J1850 Bus	10
2	5	CAN-Low	14
	8	ISO 9141-2 L Ausgang	15
	9	Batterie Spannung	16

Belegung CAN-DSub, ELM-DSub und OBD-Stecker

Interfaceimplementierung

Wir erkannten zusammen mit der OBD-Gruppe sehr früh, dass es sinnvoll wäre, den über uns liegenden Schichten ein einheitliches Interface zur Verfügung zu stellen. Da wir leider keine Kapazität frei hatten, erstellte die OBD-Gruppe ein Interface, dass nun beide Gruppen implementierten.

Im Laufe des Projektes stellte sich dann noch heraus, dass eine einheitliche Schnittstelle an allen Übergängen der Gruppen sehr sinnvoll wäre. Somit wäre es möglich, an jeder beliebigen Stelle eine evtl. ausgefallene Schicht zu überspringen. Ein Beispiel: Übertragung schlägt fehl, dann könnte zumindest eine Visualisierung und Datenaufzeichnungen im Fahrzeug erfolgen. Also einigten wir uns auf ein einheitliches Interface an allen Übergängen.

CAN.h / CAN.cpp

Unsere Implementierung des Interface erfolgte dann in der Klasse CAN. Diese Klasse beinhaltet einen extra Thread, der beim Erstellen des Objektes erzeugt wird und im Hintergrund permanent die Daten vom CAN-Bus empfängt, auswertet und in internen Variablen speichert. Diese Daten sind über die entsprechenden Getter verfügbar. Da nun aber mehrere Threads auf die selben Variablen zugreifen, wurde es erforderlich, dass sowohl die Getter als auch die Setter threadsicher gemacht wurden, was unter Linux durch Semaphoren ohne Weiteres möglich ist.

Diese Klasse wurde intensiv getestet (Langzeittest: ca. 8 Std. Dauerbetrieb über Nacht mit einem virtuellen CAN-Bus, siehe Kapitel VCAN). Die dazu genutzten Programme sind PeakCANMiniSimulator und PeakCANInterpreter.

Über CAN sind folgende Werte empfangbar und entschlüsselt (Stand: 01.10.2010):

Motordrehzahl, Gaspedalposition, Drosselklappenposition, Drehmoment, Raddrehzahl, Radumdrehungen und Kühlmitteltemperatur. Die Werte Lenkradwinkel haben wir auch noch entdeckt – allerdings sind die Werte relativ schnell als ungültig markiert. Daher nehmen wir sie nicht in unsere Dokumentation auf (für spätere Analysen: Message-ID 0x180). Den genauen Aufbau der Nachrichten siehe Kapitel „CAN-Nachrichten“.

Verwendung

Um die Klasse CAN nutzen zu können, wird die Shared Library LibCanIO benötigt und muss auf dem System installiert sein.

Des Weiteren wird beim Build die Option -lpthread und die Library libcanio.so benötigt.

Hier ein Beispiel:

```
g++ *.cpp -o TestApplication -lpthread /usr/lib/libcanio.so
```

OBD over CAN

Im Opel Astra läuft die OBD-Diagnose intern auch über den CAN-Bus. Dies stellte die OBD-Gruppe schon am Anfang des Projektes fest. Somit war eins unserer Ziele, für diese Gruppe eine Schnittstelle zur Verfügung zu stellen, um die OBD-Anfragen direkt an das Fahrzeug zu senden, ohne dabei einen Protokollinterpreter wie den ELM327 einzusetzen. Nach einer Auswertung der mitgelauschten Nachrichten stellten wir fest, dass die Anfrage vom ELM an das Fahrzeug über die CAN-ID 0x7DF gesendet wird. Die Antwort kommt laut Wikipedia [1] je nach Steuergerät auf unterschiedlichen Ids – beim Opel war nur die ID 0x7E8 zu empfangen. Der Aufbau der Nachrichten war mit den Anfragen direkt beim ELM vergleichbar:

ID	Nutzdaten	Byte 0: weitere Bytes	Byte 1:Mode	Byte 2 (evtl. noch Byte 3): PID
0x7DF	8	2	z. B. 01, 02..	PID Code (1 Byte)
0x7DF	8	3	z. B. 22 ...	PID Code (2 Byte)

OBD-Anfrage, nicht genutzte Bytes sollten auf 0x55 gesetzt werden

ID	Nutzdaten	Byte 0: weitere Bytes	Byte 1: Mode	Byte 2 (evtl. noch 3): PID	1-4 Byte Wert
z. B. 0x7E8	8	3 bis 7	Anfrage + 0x40	Wie Anfrage	Wert

OBD-Antwort

Es war auch ohne weiteres Möglich, diese Nachrichten an den Opel zu senden. Die Antwort vom Fahrzeug konnte auch ausgewertet werden. Leider wurde aus ungeklärten Gründen die Nachricht vom Fahrzeug endlos wiederholt. Auch das Ausschalten der Zündung löschte die Nachrichten auf dem Bus nicht. Erst durch das Anstecken des ELM konnte die Sendungswiederholung unterbunden werden.

Eine Vermutung, woher dies kommen könnte war, dass der Peak CAN-Adapter die empfangenen Nachrichten nicht quittiert (also kein Acknowledge setzt). Wenn auf dem CAN-Bus eine Nachricht nicht bestätigt wird, wird diese im Regelfall vom Sender solange wiederholt, bis die Empfangsbestätigung kommt.

Diese Vermutung widersprach allerdings der Beobachtung im Labor, da hier mit nur 2 angeschlossenen Adaptern (ohne irgendeinen anderen Empfänger) jede Nachricht exakt einmal gesendet wurde. Entfernte man den zweiten Empfänger, so beginnt auch hier die Sendungswiederholung auf dem Bus.

1 OBD-II PIDs (http://en.wikipedia.org/wiki/OBD-II_PIDs)

Im laufe des Projektes entstandene Software

Im Laufe des Projektes entstanden einige Programme. Diese Programme sind im Regelfall kleine Anwendungsbeispiele der verschiedenen Zwischenstufen (vom CAN-Logger, der als erstes mit nur ein paar Zeilen Code entstand und bis zum Ende hin zu einem großen Programm mit Konfigurationsdatei und Kommandozeilenparametern heranreifte, bis hin zu einem MiniSocket-Server und -Client, der die Problematik QT und Exceptions lösen sollte.

CanBomber

Eines der Ziele unserer Gruppe war es das uns zur Verfügung stehende Mercedes-Kombi-Instrument möglichst vollständig anzusteuern um es im laufe des Projektes als Anzeige der vom Fahrzeug übermittelten Daten zu verwenden. Das Problem dabei war dass es praktisch keine Informationen darüber gab welche Nachrichten das Cockpit benötigt um z.B. die Warnlampen auszuschalten, oder die verschiedenen Anzeigen zu kontrollieren.

Mangels alternativen entschlossen wir uns das Cockpit mit einem Hagel aus zahllosen Nachrichten zu bombardieren um zu sehen ob sich irgendetwas auf der Anzeige regt. Als erstes kam dabei ein extrem schmutziger Hack des CanWriters zum Einsatz, der allerdings von Code und Bedienung her kaum den Ansprüchen gerecht wurde. Als wir dann mehr oder weniger per Zufall auf die recht gut Dokumentierte libncurses-Bibliothek stießen und damit eine Möglichkeit hatten schnell und einfach ein simples Terminalinterface zusammenzubauen, war sehr schnell ein für diesen Zweck recht leistungsfähiges Tool geschaffen. Selbiges tauften wir aufgrund seines Einsatzgebietes scherzhaft den „CanBomber“.

Funktionen

- Visuelles Terminalinterface via ncurses.
- Komplett über Tastatur steuerbar.
- Kann bis zu 10 CAN-Nachrichten parallel senden.
- Zeitintervalle in ms für jede Nachricht einzeln setzbar.
- „on the fly“ manipulieren der Can-IDs, des Frame-Formats (Standard oder Extended), sowie des Typs (normale Nachricht oder RTR).
- „on the fly“ manipulieren der Datenbytes:
    - Bits können einzeln gesetzt und gelöscht werden.
    - Datenbyte löschen (0x00) oder setzen (0xFF).
    - Bitshift des Datenbyte-Inhalts nach links oder rechts.
    - Automatische Zufallswerte bei jedem Intervall.
    - Automatischer Inkrement bei jedem Intervall.
    - Automatischer Bitshift bei jedem Intervall.

Hinweis: Der CanBomber wurde erst extrem spät entwickelt (Donnerstag in der zweiten Woche), dadurch hatten wir nur sehr wenig zeit das Programm sauber zu schreiben. Entsprechend gibt es ein oder zwei miese Bugs die noch behoben werden müssten, der Code an sich ist schlecht Dokumentiert und verwendet einige wirklich grausame Krücken um schnell ans Ziel zu kommen, die Effizienz könnte an vielen Stellen ganz erheblich verbessert werden. Dennoch war es ein wirklich extrem nützliches Helferlein ohne das die vorhandene Dokumentation zum Mercedes-Kombi-Instrument nicht einmal ansatzweise so Umfangreich wäre wie es jetzt der Fall ist.

Installation

Achtung: Für die Installation sind Root-Rechte erforderlich, sowohl der PCAN-Treiber (oder die alternative VCAN), als auch die Installation von libCanIO und libncurses erfordern selbige.

Für den CanBomber ist die ebenfalls hier entstandene libCanIO notwendig, genauso wie libncurses in der Entwicklerversion. Unter Ubuntu 10.04 ist die Installation von libncurses jedoch schnell und einfach mit dem folgenden Befehl erledigt:

```
sudo apt-get install libncurses-dev
```

Die Installation von libCanIO gestaltet sich als ebenso einfach, die notwendigen Sourcen sind auf dem Perforce-Server unter „*obt_t_bpse_ws_10/Fahrzeugbusanbindung/CAN/libCanIO*“ zu finden. Nach einem Wechsel in das libCanIO Verzeichnis reichen die folgenden Befehle:

```
make clean
make
make install
```

Danach besorgt man sich auf dem gleichen weg die Sourcen des CanBombers, die ebenfalls auf dem Perforce-Server unter „*obt_t_bpse_ws_10/Fahrzeugbusanbindung/CAN/canbomber*“ zu finden sind. Auch hier reicht ein Wechsel in das canbomber Verzeichnis und das Ausführen von:

```
make clean
make
```

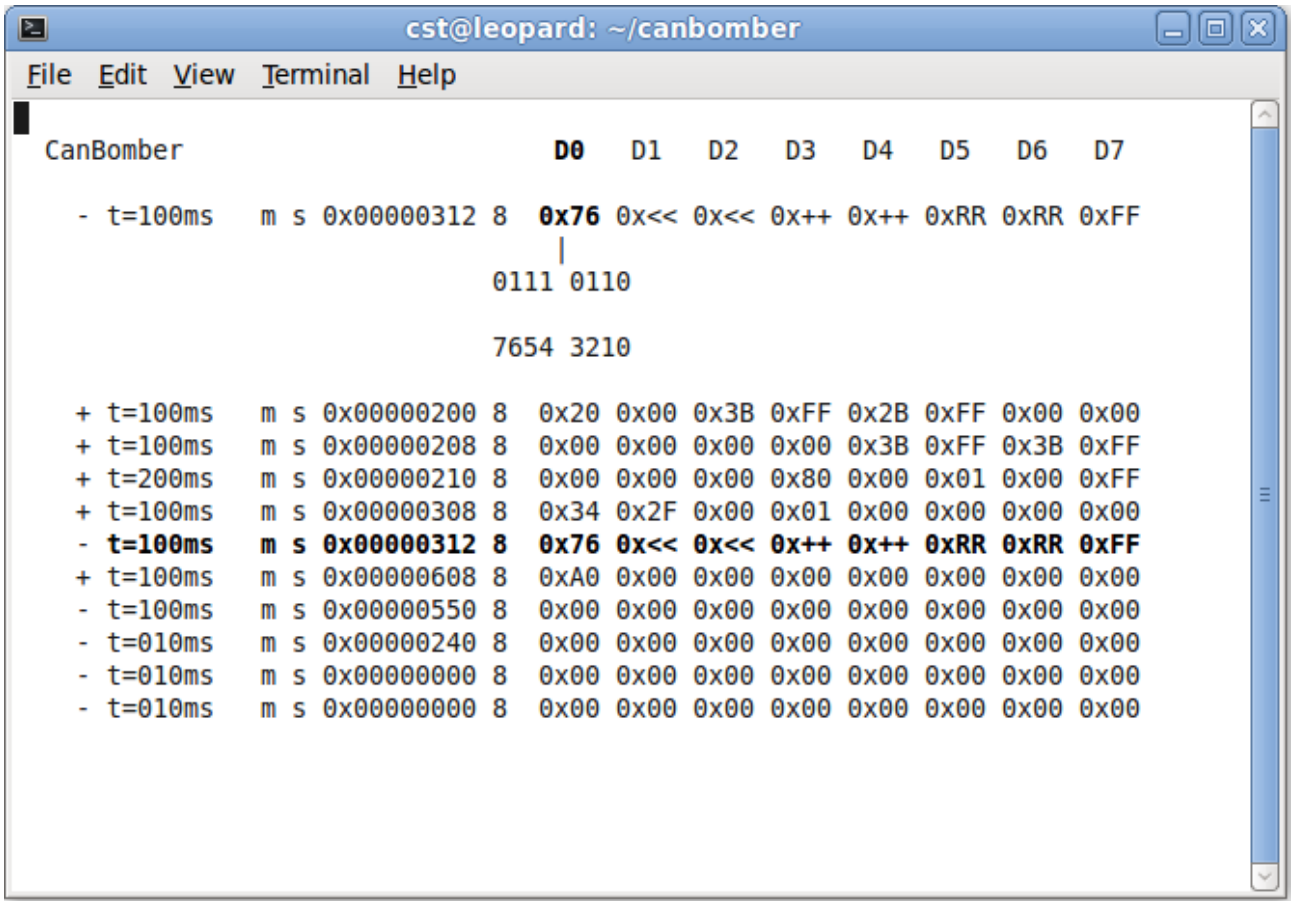
Ab jetzt ist das Programm einsatzbereit.

Bedienung

Das CAN-Interface ist zur Zeit nur über eine Kommandozeilenoption wählbar, z.B.:

```
./canbomber -f=can1
```

Hier ein Screenshot des Programms für weitere Erläuterungen:



Die CAN-Nachricht im oberen Teil ist die momentan ausgewählte, welche in der Auflistung darunter auch entsprechend hervorgehoben ist. In der Ausgewählten Nachricht selbst ist hier das Datenbyte 0 ausgewählt und ebenfalls entsprechend markiert, darunter findet sich eine Auflistung der einzelnen Bits des Datenbytes, welche einzeln gesetzt oder gelöscht werden können.

Die jede CAN-Nachricht ist dabei nach folgendem Schema aufgebaut (von links nach rechts):

- + oder - : Zeigt an ob die Nachricht gerade gesendet wird (+) oder nicht (-).
- t=100ms : Der Zeitabstand in Millisekunden in dem die Nachricht gesendet wird. Werte unter 5ms sollten vermieden werden und können zu Write-Buffer-Overflows führen.
- m (oder r): Nachrichtentyp. m = normale Nachricht, r = RTR-Nachricht.
- s (oder e): s steht für Standard-Frame, e für Extended-Frame.
- Die CAN-Message-ID.
- Anzahl der Datenbytes der Nachricht.
- Datenbytes von D0 bis D7.

0x<<, 0x++ und 0xRR stehen dabei für die bereits erwähnten Automatischen Modi, die das jeweilige Datenbyte bei jedem Senden entsprechend modifizieren. 0x<< ist dabei Bitshift, 0x++ Inkrement und 0xRR ein simpler Zufallszahlengenerator.

Zum Abschluss eine Auflistung der Tastenbelegung:

- Pfeiltasten Oben/Unten = CAN-Message auswählen.
- Pfeiltasten Links/Rechts = Timing, Can-Id, Datenbytes auswählen.
- Tasten 0-7 = Bits 0-7 werden auf 0 oder 1 gesetzt.
- Tasten 8 und 9 = 8 setzt ein Datenbyte auf 0x00, 9 auf 0xFF.
- + und - = Inkrement (+) oder Dekrement (-) um 1 auf den ausgewählten Wert.
- / und * = Bitshift um 1 nach links oder rechts. Beim Intervall Zeit *= 10 oder Zeit /= 10.
- q, w, e = Inkrement um 0x100 (q), 0x10 (w), 0x1 (e). Beim Intervall 100ms, 10ms und 1ms.
- a, s, d = Dekrement um 0x100 (a), 0x10 (s), 0x1 (d). Beim Intervall 100ms, 10ms und 1ms.
- < = Aktiviert oder Deaktiviert das Senden der ausgewählten Nachricht.
- i = Automatisches Inkrement pro Intervall, nur bei Datenbytes (wird angezeigt als 0x++).
- o = Automatischer Bitshift pro Intervall, nur bei Datenbytes (wird angezeigt als 0x<<).
- p = Automatischer Zufallswert pro Intervall, nur bei Datenbytes (wird angezeigt als 0xRR).
- u = Schaltet die Automatik im ausgewählten Datenbyte aus.
- n = Wechselt zwischen Standard-Nachricht und RTR-Nachricht.
- m = Wechselt zwischen Standard-Frame (CAN 2.0A) und Extended-Frame (CAN 2.0B).

Tipp: Zum Experimentieren eignet sich die Kombination aus PeakCanLogger und VCAN ganz hervorragend, man kann nichts kaputtmachen und in Ruhe mit den vorhandenen Optionen spielen.

CAN-Simulator

Der CAN-Simulator empfängt die von der Simulationsgruppe bereitgestellten Daten und speist sie auf den CAN-Bus ein – allerdings kann natürlich auch jede andere Implementierung des OBD-Interface hierfür verwendet werden. Unter anderem haben wir es ebenfalls direkt mit unseren CAN-Aufzeichnungen und dem zwischengeschalteten PeakCANMiniSimulator und unserer CAN-Klasse getestet. Im Allgemeinen kann gesagt werden, dass der CAN-Simulator ausschließlich zur kurzen Erprobung des OBD-Interfaces der Simulationsgruppe verwendet wurde und daher von uns auch nicht die Aufmerksamkeit erfahren hat, die wir anderen Komponenten gewidmet haben. Die Sourcen sind auf dem Perforce-Server unter „*obt_t_bpse_ws_10/Fahrzeugbusanbindung/CAN/CANSimulator*“ zu finden.

Installation

Um den CAN-Simulator nutzen zu können, reicht das Wechseln in das Verzeichnis des CAN-Simulators und anschließend der Befehl:

```
make
```

Bedienung

Da das Programm nur zu Testzwecken verwendet wurde, hier der einfache Aufruf:

```
./CANSimulator
```

libCanIO

libCanIO entstand Mitte der ersten Woche nachdem wir eine Möglichkeit gefunden hatten den leistungsfähigeren *netdev*-Treiber anzusprechen. Geplant war es eine einfach zu bedienende Schnittstelle zu erschaffen mit denen man in wenigen Zeilen Zugriff auf die nötigen Funktionen wie Lesen, Schreiben und Bitrateneinstellungen hat.

Das ganze nahm jedoch extrem viel zeit in Anspruch für vergleichsweise wenig Funktionalität. Die Dokumentation über SocketCAN ist schwer aufzutreiben, und über diverse wichtige Dinge existieren wenig mehr als ein paar Fetzen Code, so dass die meiste zeit mit Recherche anstatt mit Programmieren zugebracht wurde.

Ein großes Problem war dabei das setzen der Bitrate mit dem PCAN-Adapter, eine eigentlich sehr wichtige Funktionalität die eine Schnittstelle bieten sollte. Nach langer aufreibender Suche die zwischenzeitlich sogar in die Kernel-Quellen führte, stellt sich allerdings heraus dass es ein generelles Problem mit SocketCAN ist – Die Hardware ist offensichtlich zu unterschiedlich um eine standardisierte Ansprechmöglichkeit festzulegen, und die Programmierer die das Projekt SocketCAN betreuen finden keine elegante Möglichkeit es über Optionen zu ermöglichen. Zwar gibt es einige Wege und mittel, die aber von unserem PCAN-Adapter nicht unterstützt wurden.

Ebenso problematisch war die Filtermöglichkeit, die nicht in einem vernünftigen Rahmen, also während einer aufgebauten Verbindung, zum laufen zu bekommen war. Auf einer der Seiten wurde erwähnt dass „mit den richtigen Befehlen“ das ganze durchaus möglich ist, welche das sind lies sich allerdings auch hier nach Stundenlanger Suche nicht herausfinden.

Übrig bleibt mehr oder weniger eine Baustelle die nur Zugriff auf die Grundlegenden Funktionen bietet, dabei aber eine Menge Tipperei erspart und den Verbindungsaufbau sowie Lesen, Schreiben und das setzen von einigen Optionen auf Einzeiler statt größeren Blöcken Code beschränkt.

Die libCanIO ist dabei als Shared Library ausgelegt, die einfach in den /usr/lib/ Pfad kopiert wird und dort allen Applikationen ohne große Probleme zur Verfügung steht. Grundgedanke dahinter war das sich mehrere Applikationen die gleiche Bibliothek teilen und Updates schnell und einfach durchgeführt werden können, ohne aufwendiges umher kopieren, oder der alternative mit Pfaden die mehrere male ../../ enthalten und nach dem Hochladen auf den Perforce-Server und späterem herunterladen auf eine andere Maschine nicht mehr funktionieren.

Nachteile sind die dafür benötigten Root-Rechte. Da allerdings für die Kommunikation mit CAN, sei es mit dem PCAN-Adapter oder der alternative VCAN, ebenfalls Root-Rechte für die Installation benötigt werden ist dieser Nachteil mehr oder weniger zu verschmerzen, im Projekt gab es dennoch einige Probleme aufgrund dieser Tatsache.

Funktionen

- Sehr einfacher Verbindungsaufbau.
- Lesen und Schreiben auf den CAN-Bus mit Einzeiler (Lesen Blockiert dabei!).
- Auftretende Fehler werden erkannt und auf stderr ausgeben (abschaltbar).
- Setzen der Optionen Loopback und RecvOwnMsgs möglich.

Installation

Die Installation von libCanIO gestaltet sich als einfach, die notwendigen Sourcen sind auf dem Perforce-Server unter „*obt_t_bpse_ws_10/Fahrzeugbusanbindung/CAN/libCanIO*“ zu finden. Nach einem Wechsel in das libCanIO Verzeichnis reichen die folgenden Befehle:

```
make clean
make
make install
```

Sollten keine Root-Rechte zur Verfügung stehen und die Funktionalität dennoch benötigt werden, ist es das mit Abstand einfachste die libcanio.cpp und libcanio.h aus dem /src Verzeichnis in das eigene Projekt zu kopieren und beim Kompilieren mit einzubinden.

Verwendung

Die Dokumentation der Funktionen befindet sich im Quelltext und sollte alles notwendige erklären. Dennoch hier ein kleines Beispiel, das zeigt was alles nötig ist um eine CAN-Message zu lesen oder zu schreiben.

```
#include <libcanio.h>

int main (void)
{
      // Struct mit CAN-Daten erstellen
      can_frame frame;
      frame.can_id = 0x100;   // CAN-Message-ID
      frame.can_dlc = 8;            // Anzahl der gesendeten Datenbytes
      frame.data[0] = 0x11;   // Inhalt der Datenbytes
      frame.data[1] = 0x22;
      frame.data[2] = 0x33;
      frame.data[3] = 0x44;
      frame.data[4] = 0x55;
      frame.data[5] = 0x66;
      frame.data[6] = 0x77;
      frame.data[7] = 0x88;

      // Can-IO Objekt erstellen
      CanIO can;

      // Verbindung mit CAN-Interface can0 aufbauen,
      // Programm abbrechen wenn ein Fehler aufgetreten ist
      if (can.connect("can0") < 0)
                  return 0;

      // CAN-Message schreiben
      can.writeMsg(&frame);

      // CAN-Message lesen (blockierend!), überschreibt Inhalt von frame.
      can.readMsg(&frame);

      return 0;
}
```

MiniSocketClient und MiniSocketServer

Kurz vor Ende des Projektes schien es ein massives Problem bei der Übertragungsgruppe zu geben, so dass sich die Datenaufzeichnungsgruppe direkt im Fahrzeug anbinden wollte. Allerdings programmierte diese Gruppe in QT – und QT kann nicht mit Exceptions umgehen. Leider war die Implementierung der ELM-Klasse und der CAN-Klasse darauf ausgelegt, dass von beiden Klassen bei nicht unterstützen oder noch nicht empfangenen Werten eine Exception geworfen wird, um dem zugreifenden Programm mit zu teilen, dass dieser Wert von anderer Stelle beschafft werden muss. Ein Beispiel: Die CAN-Klasse unterstützt die Methode „getFullFillLevel“ nicht – den Tankinhalt haben wir bisher noch nicht aus den aufgezeichneten Daten ermitteln können. Versucht nun ein Programm diesen Wert von der CAN-Klasse zu erhalten, so erhält es eine Exception und muss diesen Wert bei der ELM-Klasse anfragen. Diese Logik hat Christoph Becker von der OBD-Gruppe implementiert und als Car-Klasse zur Verfügung gestellt – allerdings werden auch hier wieder bei ungültigen Werten Exceptions ausgelöst, die auch wieder behandelt werden müssen.

Und genau an diesen vielen Stellen lag das Problem: QT brach bei der ersten Exception mit „SIGABORT“ ab. Um eine Exception-Unterstützung in QT zu realisieren, wäre ein komplettes Neubauen des Compilers notwendig gewesen (mit entsprechenden Build-Optionen).

Um dies zu umgehen, schrieben wir den „MiniSocketClient“ und „MiniSocketServer“. Hierin realisierten wir eine TCP-Socket-Verbindung zwischen zwei Programmen auf einem System. Der Client fungierte hierbei als Bereitstellung der Daten für nachfolgende Schichten – natürlich ohne Exceptions.

Der Server sendet die Daten mit Hilfe eines serialisierten Objektes direkt an den Client. Dazu nutzt der Server z. B. Die CAN-Klasse. Diese empfängt im Hintergrund die Werte und stellt diese (natürlich threadsicher) dem Programm zur Verfügung. Der MiniSocketServer überträgt daraufhin die Daten in ein extra Objekt der Klasse „MiniSocket“. Auch MiniSocket ist eine threadsichere Implementierung des OBD-Interface. Allerdings verwendet diese Klasse absolut keine Exceptions, so dass das Objekt nach der Deserialiserung auf der Clientseite direkt ohne try-catch-Blöcke genutzt werden kann.

Nach der Serialisierung des MiniSocket-Objekt wird dieses über die bereits angesprochene TCP-Verbindung zum Client gesendet. Dieser deserialisert das Objekt wieder und kann somit auf die einzelnen Werte zugreifen.

Der Client ist wieder als threadsichere Implementierung des OBD-Interface realisiert und kann somit an jeder beliebigen Stelle im Projekt genutzt werden. Diese Client-Klasse heißt „MiniSocketClient“ und stellt die empfangenen Daten threadsicher zur Verfügung. MiniSocketServer und MiniSocketClient sind Beispielimplementierungen der Klassen.

Die Sourcen zu diesen Programmen sind unter „*obt_t_bpse_ws_10/Fahrzeugbusanbindung/CAN/MiniSocketClient*“ und „*obt_t_bpse_ws_10/Fahrzeugbusanbindung/CAN/MiniSocketServer*“ auf dem Perforce-Server abgelegt.

Aufgrund der späten Entstehung dieses ist die Kommentierung leider sehr mager geraten bis fast nicht vorhanden. Allerdings ist durch die Benennung der einzelnen Methoden deren Funktionalität schon ausreichend erklärt.

Installation

Die Installation der beiden Programme gestaltet sich sehr einfach. Dazu muss nur in den jeweiligen Programmverzeichnissen der Befehl

```
make
```

ausgeführt werden.

Verwendung

Der MiniSocketServer muss zur Verwendung immer zuerst gestartet werden. Leider ist auch noch ein Fehler im Client enthalten, der dazu führt, dass der Server nach dem Beenden des Clients ebenfalls beendet wird.

Server starten:

```
./MiniSocketServer
```

und danach der Client:

```
./MiniSocketClient
```

PeakCANInterpreter

Der PeakCANInterpreter ist ein Beispiel, wie die Klasse CAN genutzt wird. Das Programm nutzt NCURSES und stellt alle empfangenen (und entschlüsselten) CAN-Daten auf dem Bildschirm dar. Wir nutzten es erfolgreich zum Testen der CAN-Klasse.

Die Sourcen des PeakCANInterpreters sind auf dem Perforce-Server unter „*obt_t_bpse_ws_10/Fahrzeugbusanbindung/CAN/PeakCANInterpreter*“ abgelegt.

Installation

Der PeakCANInterpreter verfügt zum Build über einen Makefile. Die Installation gestaltet sich also derart, dass im Programmverzeichnis der Befehl

```
make
```

ausgeführt werden muss.

Verwendung

Um den PeakCANInterpreter nutzen zu können, müssen auf der Maschine entweder die PCAN-Treiber im netdev-Modus installiert oder ein VCAN eingerichtet sein. Es besteht die Möglichkeit beim Starten des Programmes die CAN-Schnittstelle mit anzugeben:

```
./PeakCANInterpreter
```

oder zum Beispiel

```
./PeakCANInterpreter can0
```

PeakCanLogger

Der PeakCanLogger ist eines der ersten Tools die im Laufe des Projektes in unserer Gruppe entstanden sind. Ziel war es einen einfachen Logger zu erschaffen der sich mit dem PCAN-Adapter verbindet und die dort auftretenden Nachrichten anzeigt oder bei Bedarf in eine Datei schreibt.

Zu diesem Zeitpunkt verwendeten wir noch den *chardev*-Treiber, da wir anhand der mitgelieferten Code-Beispiele des Treibers (die nur mit *chardev* funktionieren) schneller ein Programm lauffähig hatten, als mit netdev, der auf SocketCAN[1] aufsetzt. Im Verlauf der nächsten Tage bekamen wir allerdings eine funktionsfähige SocketCAN-Implementierung zum laufen, die den riesigen Vorteil hatte dass sich mehrere Applikationen den gleichen Adapter teilen können.

Die entsprechende Funktionalität wurde daher dem PeakCanLogger hinzugefügt, so dass er als einziges Programm in unserem „Fuhrpark“ sowohl *chardev* als auch *netdev* beherrscht, während die nachfolgenden Tools mit der danach erstellten libCanIO arbeiten.

Das Tool war im Rahmen des Projektes mehrmals zur Aufzeichnung der CAN-Bus-Daten des Fahrzeugs im Einsatz und hat seinen Dienst dort problemlos verrichtet. Die aufgezeichneten Daten können auf dem Perforce-Server unter „*obt_t_bpse_ws_10/Fahrzeugbusanbindung/CAN*“ in Form von .csv Dateien geladen werden.

Funktionen

- Beherrscht *chardev* und *netdev* Implementationen des PCAN-Treibers.
- Vollständig per Kommandozeilenoptionen konfigurierbar.
- Konfigurationsdatei als Möglichkeit zur Festeinstellung bestimmter Optionen.
- Beherrscht einfache Filterfunktionen (einzelne Nachrichten zeigen oder verstecken).
- *nur chardev:* Einstellen der Bitrate.
- Ausgabe in Datei durch einfaches Umleiten mit `./PeakCanLogger > can_log.txt`
- Drei verschiedene Ausgabeformate.

Installation

Der PeakCanLogger setzt den PCAN-Treiber zwingend voraus, für eine Installation sind daher Root-Rechte nötig, sofern der PCAN-Treiber noch nicht installiert ist.

Die Installation gestaltet sich als sehr einfach, die notwendigen Sourcen sind auf dem Perforce-Server unter „*obt_t_bpse_ws_10/Fahrzeugbusanbindung/CAN/PeakCanLogger*“ zu finden. Nach einem Wechsel in das PeakCanLogger Verzeichnis reichen die folgenden Befehle:

```
make clean
make
```

Bedienung

Das Programm erklärt sich sprichwörtlich von selbst, wenn man es mit der Hilfe-Option startet:

```
./PeakCanLogger -?
```

1 http://en.wikipedia.org/wiki/Socketcan oder http://developer.berlios.de/projects/socketcan/

PeakCANMiniSimulator

Der PeakCANMiniSimulator versendet zuvor aufgezeichnete CAN-Nachrichten mit ungefähr dem Originaltiming. Beim Timing traten allerdings Probleme auf: entweder wir nutzen usleep – dann lief das Programm zu schnell, oder wir nutzen eine selbst geschriebene „Sleep"-Methode – hier war das Programm dann allerdings zu langsam.

Der PeakCANMiniSimulator liest dazu eine CSV-Datei ein, welche mit Semikolon getrennt ist. Die ersten beiden Spalten stellen dabei den Timestamp dar. In der 3ten Spalte steht der Frame-Typ (0=Base-Frame, 1=Remote-Base-Frame, 2=Extended-Frame und 3=Remote-Extended-Frame), gefolgt von der ID. Nach der ID kommt die Menge der Nutzdaten (also eine Zahl zwischen 0 und 8) und immer 8 Byte Daten, egal, wie viel tatsächlich benötigt werden. Alle Werte außer dem Timestamp sind hexadezimal.

```
Der Syntax lautet: ./PeakCANMiniSimulator CSV-Datei [optional: Name der CAN-Schnittstelle]
```

Der optionale Name der CAN-Schnittstelle ermöglicht es, innerhalb kurzer Zeit und ohne neu bauen der Anwendung auf z. B. Einen VCAN umzustellen.

Auch hier sind die Sourcen wieder auf dem Perforce-Server zu finden: „*obt_t_bpse_ws_10/Fahrzeugbusanbindung/CAN/PeakCANMiniSimulator*"

PeakCANWriter

PeakCANWriter war nach dem PeakCanLogger das zweite Programm, dass wir im Umfeld dieses Projektes geschrieben haben. Die Funktion ist schnell erklärt: es sendet permanent eine Nachricht auf den CAN-Bus (hier: Ansteuerung der Tachonadel des Mercedes-Kombi-Instrumentes). Da wir es ausschließlich für die ersten Gehversuche benötigten, haben wir es nicht weiter entwickelt. Es beruht auch noch auf den *chardev*-Treibern, welche wir seit dem zweiten Tag nicht mehr verwenden und ist somit mit den unter „Treiberinstallation" beschriebenen Treibern nicht zu verwenden. Um das Programm nutzen zu können, müsste zuerst der *netdev*-Treiber entfernt und als *chardev* neu gebaut werden. Die Befehle dazu lauten (auszuführen im Treiberverzeichnis):

```
sudo rmmod pcan
make clean
make NET=NO
sudo make install
sudo modprobe pcan
```

Der PeakCANWriter ist das einzige unserer Programme, dass noch nicht über ein Makefile verfügt und muss daher mit Eclipse gebaut werden.

SplitCANMessagesByID

SplitCANMessagesByID macht das, was der Name sagt: es splittet eine CSV-Datei auf und erstellt für jede ID eine eigene Datei. Dieses kleine Programm nutzen wir intensiv, um uns einzelne Identifier aus den aufgezeichneten Daten genauer anschauen und analysieren zu können. Einen weiteren Sinn verfolgten wir mit diesem Programm nicht. Theoretisch hätte es an dieser Stelle auch ein kleines Perl-Skript getan, aber dazu hätten wir uns extra hierfür wieder in Perl einarbeiten müssen, was gerade dann schwer fällt, wenn man gerade so richtig schön mit den String-Operationen von C++ beschäftigt ist.

Der Speicherort in Perforce lautet: „*obt_t_bpse_ws_10/Fahrzeugbusanbindung/CAN/SplitCANMessagesByID*“

Verwendung

Um dieses kleine Programm verwenden zu können, muss es (wie alle anderen Programme ja auch), erst einmal gebaut werden:

```
make
```

Der Syntax ist auch hier wieder einfach gehalten:

```
./SplitCANMessagesByID CSV-Datei
```

Die einzelnen erzeugten Dateien liegen im Anschluss im Programmverzeichnis.

Mercedes Benz Kombiinstrument

Das Mercedes-Kombi-Instrument stand uns während unseres Projektes zur Verfügung. Es handelt sich um ein reguläres Cockpit aus einem Mercedes unbekannter Bauart. Ziel war es möglichst viele der vorhandenen Anzeigen, wie Geschwindigkeit, Drehzahl, Kühlmitteltemperatur und andere Dinge wie die Warnlampen zu kontrollieren, um es eventuell als „zweite Anzeige“ für die vom Fahrzeug übertragenen Daten zu verwenden.

Leider ist bis auf Geschwindigkeit und Drehzahl keine Dokumentation über das Ansprechen des Instruments vorhanden oder auffindbar gewesen, so dass eine Menge Experimente mit einer eigens dafür programmierten Software nötig waren (siehe CanBomber).

Aktueller Stand

Ansteuerbar:

- Die meisten Warnlampen: ABS, ESP, Handbremse (links), Motor (rechts), Warndreieck und LIM (Geschwindigkeitsmesser) sowie Tank-Leer (Tankanzeige).
- Geschwindigkeitsanzeige.
- Drehzahlmesser.
- Kühlwassertemperatur.
- Drei verschiedene Alarme mit entsprechender Anzeige im LCD-Display: „*Engine oil level; Stop. ENGINE OFF!*“, „*Tyre pressure; Check Tyres*“ und „*Tyre pressure; Caution tyre defect!*“
- Tempomat-Anzeige im LCD-Display (mit normaler Anzeige wechselnd und permanent), zwei verschiedene Modi (Bildschirmfüllend und als Zusatz zur normalen Anzeige) mit verschiedenen Optionen (Mph blinkend, statt Mph ---), und „Exceeded“ Warnmeldung.

Nicht ansteuerbar / Problematisch:

- „Display defect; Visit Workshop“ Anzeige nach dem anschalten. Man kann die Meldung zwar unterdrücken indem man die Tempomat-Anzeige permanent aktiviert, beseitigen kann man es allerdings nicht.

  Allerdings: Bei den rot beleuchteten Alarm-Nachrichten ist die linke Seite des LCD-Display ziemlich dunkel. Womöglich muss nur ein einfaches Lämpchen ausgetauscht werden um sich der Fehlermeldung zu entledigen? Sollte man sich ansehen, würde das Cockpit bei Demos wesentlich besser dastehen lassen.
- Außentemperaturanzeige, bekommt nach unserem Wissensstand die Temperatur über ein analoges Signal von außen und nicht über den CAN-Bus.
- Tankanzeige. Vielleicht auch über ein analoges Signal, vielleicht ist eine Kombination von CAN-Nachrichten nötig, vielleicht haben wir eine Can-ID einfach nur übersehen.
- Ein paar verbleibende Warnlampen (vor allem rechts, sowie Gurtwarner links).
- Uhrzeit. Ging bisher immer falsch, reagierte nicht auf Nachrichten.
- Speedtronic (Teil der Tempomat-Anzeige), protestiert mit Alarmton und Fehlermeldung.

Technische Daten zur Ansteuerung

Die Ansteuerung der Anzeige erfolgt über eine handvoll verschiedener CAN-Nachrichten. Wichtig dabei ist das diese handvoll Nachrichten gemeinsam in regelmäßigen Abständen an das Cockpit gesendet werden müssen um z.B. Kontrolle über die Warnleuchten zu erhalten, oder zu verhindern dass z.B. der Drehzahlmesser sofort wieder auf 0 RPM zurückfällt.

Die Nachrichten haben dabei die Ids 0x200, 0x208, 0x210, 0x308, 0x312 und 0x608. Selbige haben dabei acht Datenbytes, deren Inhalt zum einfachen Testen bei allen Nachrichten 0x00 sein kann. Werden die Nachrichten richtig gesendet, gehen die Warnlampen die sich auf der linken und rechten Seite befinden aus.

Dazu gibt es noch zwei weitere Ids mit zusätzlichen Funktionen, die ebenfalls Regelmäßig gesendet werden müssen, sofern man sie nutzen möchten.

Die Zeitabstände zwischen dem Senden können dabei bei allen Nachrichten recht großzügig gewählt werden, 50-100ms sind absolut ausreichend, zu große Zeitspannen können allerdings zu Spinnereien führen, in denen die Warnleuchten kurz angehen oder der Alarmton zu hören ist.

Auf den folgenden Seiten befindet sich eine detaillierte Beschreibung über den aktuellen Wissensstand der Nachrichten. Das Wissen wurde dabei durch Brute-Force zusammengetragen, in denen wir das Cockpit schlicht und ergreifend mit hunderten Nachrichten nacheinander bombardiert haben und schauten ob selbiges darauf reagiert. Da wir allerdings selbst mit unserem eigens dafür entwickelten Tool (das wir daraufhin scherzhaft CanBomber nannten), kaum alle möglichen Kombinationen abdecken können, kann es durchaus sein dass weitere Versuche noch weitere Optionen zu tage bringen, insofern könnte sich ein wenig weiteres „bombardieren“ lohnen.

0x200

Steuert Alarmdreieck-Warnlampe und Geschwindigkeit. Muss Regelmäßig zusammen mit 0x208 gesendet werden (Zeitabstand etwa 50-100ms), um die Warnleuchten auf der linken Seite kontrollieren zu können.

Die Datenbytes für die Geschwindigkeit werden zusammen mit 0x208 ausgewertet, insofern sind die Beobachtungen hier nur dann gültig wenn die Nachricht alleine gesendet wird ODER 0x208 die gleichen Mph-Werte beinhaltet, ansonsten ist alles möglich, inklusive zittern des Mph-Zeigers trotz unveränderter Werte. Es handelt sich vermutlich um die Werte der einzelnen Reifen die miteinander verrechnet werden.

Die Geschwindigkeit in Mph kann mit dem folgenden C/C++ Code berechnet werden:

```
hbyte = mph / 10;              // hbyte = Inhalt von Datenbyte 2 und 4
lbyte = mph - (hbyte * 250);   // lbyte = Inhalt von Datenbyte 3 und 5
```

Die Datenbytes 2+3 und 4+5 verhalten sich additiv, d.h. 7,5 Mph und 2,5 Mph ergeben 10 Mph.

Datenbyte 0

Kontrolliert die meisten Warnleuchten auf der linken Seite. Werden die Nachrichten gesendet, und lässt die Lampen per Biteinstellungen leuchten, sorgen sie für weitere Warnmeldungen beim anschalten des Cockpits.

- Bit 1 = Handbremsen-Warnleuchte. Doppelter Alarmton wenn Informationen von Adresse 0x208 komplett fehlen oder das Auto in "Bewegung" ist.
- Bit 2 = ABS-Warnleuchte.
- Bit 3 = ESP/BAS-Warnleuchte.
- Bit 4 = Warndreieck-Leuchte dauerhaft an.
- Bit 5 = Warndreieck-Leuchte blinkt (ignoriert Bit 4).
- Bit 6 = ESP/BAS-Warnleuchte (anderer Fehler?).

Datenbyte 2 + Datenbyte 4

Steuert Geschwindigkeit. 5 Mph pro Schritt, also 25 Mph für 0x05 oder 200 Mph für 0x28.

Achtung: Bit 6 oder Bit 7 führen dazu das die restlichen Bits ignoriert werden, Werte größer als 0x3F (63) sollten also vermieden werden. Da 0x3F allerdings 315 Mph ergeben und damit sowieso nicht angezeigt werden können sollte es damit keine Probleme geben.

Datenbyte 3 + Datenbyte 5

Steuert Geschwindigkeit in sehr kleinen Abständen, 0xFF ergeben 5 Mph.

0x208

Steuert Mph-Anzeige. Muss Regelmäßig zusammen mit 0x200 gesendet werden (Zeitabstand etwa 50-100ms), um die Warnleuchten auf der linken Seite kontrollieren zu können.

Die Datenbytes für die Geschwindigkeit werden zusammen mit 0x200 ausgewertet, insofern sind die Beobachtungen hier nur dann gültig wenn die Nachricht alleine gesendet wird ODER 0x200 die gleichen Mph-Werte beinhaltet, ansonsten ist alles möglich, inklusive zittern des Mph-Zeigers trotz unveränderter Werte. Es handelt sich vermutlich um die Werte der einzelnen Reifen die miteinander verrechnet werden.

Die Geschwindigkeit in Mph kann mit dem folgenden C/C++ Code berechnet werden:

```
hbyte = mph / 10;             // hbyte = Inhalt von Datenbyte 4 und 6
lbyte = mph - (hbyte *  250); // lbyte = Inhalt von Datenbyte 5 und 7
```

Die Datenbytes 4+5 und 6+7 stehen in „Konkurrenz“ zu einander, der höhere der beiden Werte wird bevorzugt, d.h. 7,5 Mph und 2,5 Mph ergeben 7,5 Mph auf der Mph-Anzeige.

Datenbyte 4 + Datenbyte 6

Steuert Geschwindigkeit. 10 Mph pro Schritt, also 50 Mph für 0x05 oder 200 Mph für 0x14.

Achtung: Bit 6 oder Bit 7 führen dazu das die restlichen Bits ignoriert werden, Werte größer als 0x3F (63) sollten also vermieden werden. Da 0x3F allerdings 630 Mph ergeben (*lach*) und damit sowieso nicht angezeigt werden können sollte es damit keine Probleme geben.

Datenbyte 5 + Datenbyte 7

Steuert Geschwindigkeit in sehr kleinen Abständen, 0xFF ergeben 10 Mph.

0x210

Steuert die Tempomat(Limit)-Anzeige im LCD-Display des Cockpits. Kann Bildschirmfüllend angezeigt werden oder als Einzeiler in der Hauptanzeige, mit einigen Optionen wie z.B. blinken lassen der Limit-Mph, kontrollieren der LIM-Leuchte in der Mph-Anzeige oder einer kleinen Warnmeldung. Muss regelmäßig zusammen mit 0x308, 0x312 und 0x608 gesendet werden (Zeitabstand etwa 50-100ms) um die Warnleuchten auf der rechten Seite kontrollieren zu können.

Datenbyte 3

Falls die Limitanzeige per *Datenbyte 5* angeschaltet wird, steuert dieses Datenbyte das Anzeigeverhalten. Ist Bit 7 inaktiv ist die Limitanzeige bei Änderungen an Datenbyte 5 und 8 nur kurz zu sehen (Limit- und Hauptanzeige im Wechsel, solange sich Datenbyte 8 nicht ändert) und schaltet schnell wieder auf die Standardanzeige zurück, bei 0x80 ist die Limitanzeige dagegen permanent zu sehen.

- Bit 7 = Limitanzeige ist permanent zu sehen, wenn *Datenbyte 5 - Bit0* aktiv ist.

Datenbyte 4

Schaltet Limit-Anzeige in Hauptanzeige hinzu, steuert LIM-Lampe.

- Bit 0 = Mit Datenbit 5 - Bit0: Exceeded Schriftzug in Limitanzeige mit Alarmton.
- Bit 3 = Limitanzeige mit 0 MPH in der Hauptanzeige, LIM-Lampe leuchtet.

  Kombinationen:

  Datenbyte 5, Bit 1: LIM-Lampe blinkt.

  Datenbyte 5, Bit 2: LIM-Lampe und Limitanzeige aus.

  Datenbyte 5, Bit 3: --- statt 0 MPH.

  Datenbyte 8: MPH Anzeige.

Datenbyte 5

Steuert die Limit-Anzeige.

- Bit 0 = Aktiviert Limitanzeige mit 0 MPH. Verhindert einen der Alarmtöne beim einschalten des Cockpits (wobei der Fehler eher abgewürgt als beseitigt wird).
- Bit 1 = 0 bzw. --- blinkt.

  Kombinationen:

  Datenbyte 4, Bit 3: LIM-Lampe statt 0 bzw. --- blinkt.

- Bit 2 = Ist Bit 1 aktiv, Speedtronic mit Fehlermeldung (piept).
- Bit 3 = --- MPH statt 0 MPH.
- Bit 6 = "Winter tire limit" statt "Limit".

Datenbyte 8

Ist die Limitanzeige zu sehen steuert dieses Byte die Geschwindigkeit, 0x05 ergeben 5 Mph, 0xFF ergeben 255 Mph.

0x240

Sorgt für einen Alarm mit Warnton und rot beleuchtetem LCD-Display mit der Nachricht *„Engine oil level; Stop. ENGINE OFF!“* sofern man die Nachricht schnell genug sendet. Die Datenbytes haben dabei scheinbar keine weiteren Auswirkungen, vielleicht technische Details für Diagnosegeräte?

0x308

Steuert den Drehzahlmesser, sowie die Warnleuchten „Tank Leer“ und „Motor“. Muss Regelmäßig zusammen mit 0x210, 0x312 und 0x608 gesendet werden (Zeitabstand etwa 50-100ms) um die Motor-Warnleuchte steuern zu können, ansonsten funktionieren nur der Drehzahlmesser und die Tank-Leer-Leuchte.

Die Drehzahl kann mit dem folgenden C/C++ Code berechnet werden:

```
hbyte = rpm / 250;                  // hbyte = Inhalt von Datenbyte 0
lbyte = rpm - (hbyte * 250);  // hbyte = Inhalt von Datenbyte 1
```

Der Code gilt dabei für Drehzahlenwerte wie z.B. 3500 oder 1200 Rpm.

Datenbyte 0

Steuert die Drehzahl. 250 Rpm pro Schritt, also 1250 Rpm für 0x05 oder 6500 Rpm für 0x1A.

Achtung: Bit 6 oder Bit 7 führen dazu das die restlichen Bits ignoriert werden, Werte größer als 0x3F (63) sollten also wie bei 0x200 und 0x208 vermieden werden.

Datenbyte 1

Steuert die Drehzahl in sehr kleinen abständen, 0xFF ergeben 250 Rpm.

Datenbyte 3

Kontrolliert „Tank-Leer“ und „Motor“ Warnleuchten.

- Bit 0 = Tank-Leer-Warnleuchte.
- Bit 1 = Motor-Warnleuchte.

0x312

Muss Regelmäßig zusammen mit 0x210, 0x308 und 0x608 gesendet werden (Zeitabstand etwa 50-100ms) um die Motor-Warnleuchte steuern zu können. Ansonsten sind keine weiteren Informationen bekannt, das Cockpit reagiert scheinbar nicht auf den Inhalt der Datenbytes.

0x550

Sorgt solange es gesendet wird für eine Alarmnachricht mit passendem einmaligen Alarmton, die mit roter Beleuchtung im LCD-Display angezeigt wird. Dabei erscheint unabhängig vom Inhalt der Datenbytes immer die Warnmeldung *„Engine oil level; Stop. ENGINE OFF!“*. Über Datenbyte 0 können zwei weitere Alarmnachrichten zugeschaltet werden, die jeweils einen weiteren Alarmton auslösen und danach im Wechsel mit der „Standardnachricht“ angezeigt werden.

Die Alarmnachrichten werden selbst dann noch angezeigt wenn 0x550 mit Zeitabständen über 500ms gesendet wird.

Es kann dabei durchaus 1 oder 2 Sekunden dauern bis das Cockpit auf die Nachricht reagiert.

Datenbyte 0

- Bit 4 = Zeigt zusätzlich den Alarm „Tyre pressure; Check tyres“ an.
- Bit 5 = Zeigt zusätzlich den Alarm „Tyre pressure; Caution tyre defect!“ an.
- Bit 7 = Unterdrückt die beiden „Tyre pressure“ Meldungen.

0x608

Steuert die Kühlwasser-Temperaturanzeige. Muss Regelmäßig zusammen mit 0x210, 0x308 und 0x312 gesendet werden (Zeitabstand etwa 50-100ms) um die Motor-Warnleuchte steuern zu können.

Die Kühlwasser-Temperatur kann mit dem folgenden C/C++ Code berechnet werden:

```
byte = temp + 0x28;     // byte = Inhalt von Datenbyte 0
```

Der Code gilt dabei für absolute Temperaturwerte wie 40° oder 120°.

Datenbyte 0

Steuert die Kühlwasser-Temperaturanzeige. 1° pro Schritt, es muss der Wert 0x28 (40) zum gewünschten Ergebnis hinzuaddiert werden um z.B. für 0x50 (80) 80° auf der Anzeige zu erhalten.

Achtung: Der Wert 0xFF (255) sollte vermieden werden, ansonsten fällt die Anzeige wieder auf „0“ (das untere Ende der Temperaturskala, die bei 40° aufhört). Der höchste Wert der noch angezeigt wird liegt bei 0xB0 (176).

C++ MKI Klasse

Um das ansteuern des Mercedes-Kombi-Instruments zu vereinfachen haben wir eine kleine Klasse geschrieben, mit denen die bislang verfügbaren Features ziemlich einfach angesteuert werden können, was später folgenden Gruppen die Arbeit hoffentlich etwas erleichtert.

Die Funktionen sind alle entsprechend Dokumentiert, in Eclipse z.B. reicht es mit dem Cursor in der Header-Datei eine weile auf einer der Funktionen zu verharren um an die Informationen zu kommen die nötig sind um sie vernünftig zu nutzen. Vieles ist auch selbsterklärend, setMph(200) stellt z.B. den Geschwindigkeitsmesser auf 200 Mph ein.

Die MKI-Klasse nutzt das in unserer Gruppe ebenfalls entstandene libCanIO, auf das an anderer Stelle in dieser Dokumentation eingegangen wird (auch dessen Installation).

Ein einfaches Beispiel zur Verwendung

```
#include "mki.h"

int main(void)
{
      // create mki object
      mki m;

      // connect to can interface, stop application if we can't connect
      if (m.connect("can0") < 0)
            return -1;

      // set Rpm to 3000
      m.setRpm(3000);

      // set Mph (you can also use m.setKmh(50); instead)
      m.setMph(120);

      // set cooling water temperature
      m.setCWTemp(80);

      // set some of the warning lamps
      m.setLamps(MKI_WARN | MKI_FUEL_LOW | MKI_WARN_BLINK | MKI_ABS);

      // set cruise control options
      m.setDisplay(MKI_LIMIT | MKI_PERMANENT | MKI_MPH_BLINK );
      m.setDisplayMph(234);

      // set one of the alerts
      // m.setAlert( MKI_TIRE_PRESSURE );

      // send the combined data in an endless loop to the cockpit
      // (neccessary, or the display will show stuff only for a single second)
      // every 50ms or so are enougth, use pthreads for nonblocking operation
      while (1) {
                  m.sendData();
                  usleep(50000);
      }

      // everything ok
      return 0;
}
```

Aufzeichnungen und Messungen im Fahrzeug

Ermittlung CAN-Bus Geschwindigkeit

Um überhaupt mit dem CAN-Bus kommunizieren zu können, war es erforderlich, die Geschwindigkeit des CAN-Buses zu ermitteln. Hierzu wurden im Vorfeld zum Projekt mit einem Oszilloskop Messungen auf den Leitungen der OBD-Buchse durchgeführt.

Dabei stellte sich heraus, dass beim Opel Astra die CAN-Leitungen CAN-High und CAN-Low auf den Pins 6 und 14 liegen. Anhand der gewonnenen Messung konnte eine Busgeschwindigkeit von 500 kbit/s ermittelt werden (eine Bit-Zeit beträgt 2,5 µs, daraus errechnet sich dann diese Geschwindigkeit):

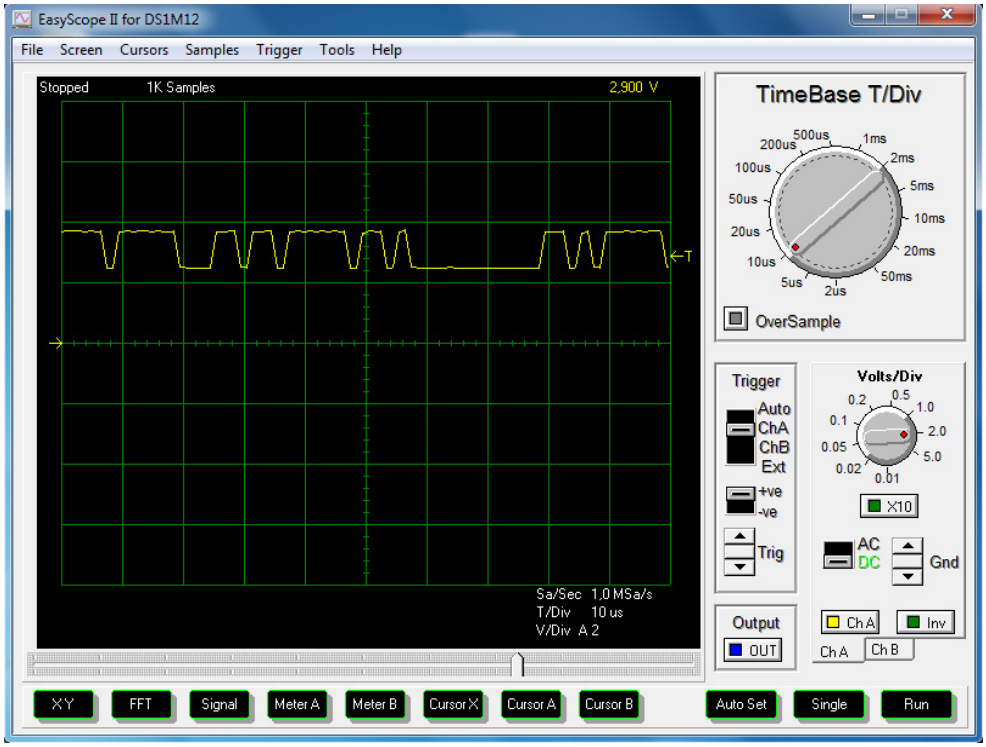


Bild: Messung der CAN-High-Leitung, y=2V/Div, x=10µs/Div, Trigger-Level=2,9 V

An der Aufzeichnung ist schön zu erkennen, dass auf dem CAN-Bus ein „Normalniveau“ von ca. 2,5 V herrscht. Die CAN-High-Signale schlagen Richtung 4 V aus, die CAN-Low-Leitung Richtung 0V.

Datenaufzeichnungen am Opel

Im Laufe des Projektes wurden mehrfach Datenaufzeichnungen am Fahrzeug durchgeführt (in Perforce stehen 3 Aufzeichnungen zur Verfügung). Hierbei wurden folgende CAN-Bus Nachrichten empfangen:

CAN-ID	Auftreten alle x ms	Inhalt der Nachricht
0x110	10	Motordrehzahl, Stellung des Gaspedals, Stellung der Drosselklappe
0x120	10	Drehmoment
0x140	10	unbekannt
0x180	10	unbekannt
0x188	100	unbekannt
0x1A0	50	unbekannt
0x280	10	Rotationsgeschwindigkeit aller 4 Reifen in RPM
0x290	100	Gesamtzahl der Rotationen pro Reifen (0-127)
0x300	100	unbekannt
0x301	100	unbekannt
0x380	100	unbekannt
0x381	100	unbekannt
0x383	100	unbekannt
0x410	100	unbekannt
0x440	1000	unbekannt
0x4A0	1000	unbekannt
0x510	1000	Kühlmitteltemperatur
0x530	1000	unbekannt
0x531	1000	unbekannt

Um ein kleines Rechenbeispiel zu bringen:

Die Nachricht über die Radumdrehungen pro Minute kommt alle 10 ms auf dem CAN-Bus. Ein Fahrzeug, dass sich mit 90 km/h fortbewegt, legt in dieser Zeitspanne 25 cm zurück. Um eine genauere Positionsbestimmung zu ermöglichen, empfiehlt es sich also, zwischen 2 Werten den Mittelwert zu bilden und zu interpolieren.

Messungen am BMW 525 Turing

Da der Opel Astra der Hochschule Darmstadt dem Fachbereich Maschinenbau gehört, kann im Rahmen weiterer Projekte nicht ohne weiteres auf dieses Fahrzeug zurück gegriffen werden. Um dem entgegen zu wirken, stellte Herr Prof. Wietzke seinen BMW 525 Turing zur Verfügung. Dazu mussten allerdings erst einmal Messungen am Fahrzeug durchgeführt werden, um auch hier die CAN-Geschwindigkeit zu ermitteln

Hierzu versuchten wir zuerst mit dem Peak CAN-Adapter auf dem CAN-Bus zu lauschen (es wurden die Geschwindigkeiten 125, 250, 500 und 1000 kbit/s getestet). Nachdem wir dabei keine Aktivität auf dem CAN-Bus fest gestellt haben, sind wir auch hier dem Fahrzeug mit einem Oszilloskop zu Leibe gerückt. Leider stellte sich dabei heraus, dass der BMW entweder auf der OBD-Buchse keinen CAN-Bus führt oder das Fahrzeug nur auf Anfragen auf dem BUS eine Aktivität zeigt.

Wir haben allerdings die Information, dass im Fahrzeug noch eine CAN-Buchse vorhanden wäre. Deren Lage ist uns bis zu diesen Zeitpunkt leider noch nicht bekannt.

CAN-Nachrichten

Da die CAN-Nachrichten am besten als Tabelle dargestellt werden können, siehe Anhang.

Die Liste enthält nur die von uns entschlüsselten Nachrichten. Um aber einen kleinen Überblick zu bieten, sind hier die Nachrichten und ihr Inhalt einmal grob umrissen:

0x110

Diese Nachricht enthält Informationen zur aktuellen Motordrehzahl, der Gaspedalstellung sowie der Drosselklappenstellung.

0x120

Hier sind die Werte zum Drehmoment enthalten.

0x280

Diese Nachricht beinhaltet die Rotationsgeschwindigkeit der einzelnen Reifen in Rotationen pro Minute.

0x290

Und hier sind die Gesamtzahl der Reifenumdrehungen pro Rad enthalten. Leider wird nur jede volle Umdrehung angezeigt, so dass Teildrehungen nicht erkennbar sind.

0x510

Diese Message enthält die Kühlmitteltemperatur.

Hilfestellungen in anderen Gruppen

Headerfile: Serializer.h

In der Übertragungsgruppe gab es Probleme, ein Objekt zu serialisieren. Dazu haben wir einen kleinen Headerfile geschrieben, der die Serialisierung eines Objektes unter C++ mit char* realisiert.

Gruppe Simulation

In der Simulationsgruppe unterstützten wir bei der Implementierung des TCP-Clients und der Simulatorklasse, durch die das OBD-Interface von dieser Gruppe implementierte wurde.

Gruppe Übertragung

Die Übertragungsgruppe hatte Probleme beim Einbinden der Car-Klasse (die gemeinsame Schnittstelle, die von CAN und OBD zur Verfügung gestellt wird). In dieser Klasse werden nacheinander in einem extra Thread die Daten zuerst versucht aus der CAN-Klasse zu besorgen. Wenn die Werte hier nicht vorhanden sind, werden die entsprechenden Werte aus der ELM-Klasse extrahiert. Da allerdings die CAN-Klasse durch ihre direkte Verbindung mit dem CAN-Controller von bestimmten Librarys abhängig ist, gab es an dieser Stelle für die Gruppe massive Integrationsprobleme. Nachdem diese aber behoben waren, konnte die Gruppe mit ihrer Implementierung fortsetzten.

Auch die Übertragungsgruppe implementierte das OBD-Interface.

Gruppe OBD

Mit der Gruppe OBD bestand während der gesamten Projektzeit immer ein reger Austausch, was darauf zurück zu führen ist, dass diese beiden Gruppen direkt am Fahrzeug arbeiten und dementsprechend sich die OBD-Buchse teilen müssen. Dabei wurde unter anderem versucht, den Bluetooth-ELM „offline“ mit Spannung zu versorgen, da für die Implementierung nicht geplant war, immer ans Fahrzeug zu gehen. Leider stellte sich dieses doch recht einfach Unterfangen als größere Herausforderung heraus. Daher haben wir zuerst das OBD-ELM-Kabel durchgemessen, um die Belegung der DSub-Buchse zu haben. Dabei stellte sich heraus, dass sowohl die Batteriespannung, als auch die Masse angeschlossen war. Ein Einspeisen der Spannungsversorgung auf diesen Pins erbrachte aber keinen Erfolg. Daraufhin wurden von der OBD-Gruppe Messungen am Fahrzeug durchgeführt um zu klären, wo die Spannungsversorgung her kommt. Leider erbrachte diese Messung keinen Erfolg. Daher zerlegten wir den ELM um darin evtl. parasitäre Spannungsversorgungen zu suchen – immerhin liegt beim KL-Bus auf der L-Leitung immer 70-100% der Batteriespannung. Aber auch hier konnten wir keinen Erfolg verbuchen. Nach näherer Analyse der Schaltung kam heraus, dass die Spannungsversorgung über die Batterie im Gehäuse selbst offen war – hier fehlte eine Lötbrücke. Da der ELM aber funktionierte, wollten wir dies nicht ändern. Die OBD-Gruppe fragte daraufhin beim Hersteller nach einer entsprechenden Beschreibung, wie das Gerät denn ohne Fahrzeug getestet werden kann – hier stand bis Ende des Projektes die Antwort noch aus.

Nachricht ID 0x110		
Byte	Bit	Bedeutung
0	7	Engine Speed Valid (Valid=0, Invalid=1)
	6	Accelorator Effective Position Valid (Valid=0, Invalid=1)
	5	
	4	Throttle Position Valid (Valid=0, Invalid=1)
	3	
	2	
	1	
	0	
1	7	Engine Speed 16 Bit Phys= N*0,25 --> 0-18383,8 rpm
	6	
	5	
	4	
	3	
	2	
	1	
	0	
2	7	
	6	
	5	
	4	
	3	
	2	
	1	
	0	
3	7	Accelorator Effective Position 8 Bit Phys= N*0,392157 --> 0 - 100%
	6	
	5	
	4	
	3	
	2	
	1	
	0	
4	7	
	6	
	5	
	4	
	3	
	2	
	1	
	0	
5	7	Throttle Position 8 Bit Phys= N*0,392157 --> 0 - 100%
	6	
	5	
	4	
	3	
	2	
	1	
	0	
6	7	
	6	
	5	
	4	
	3	
	2	
	1	
	0	
7	7	
	6	
	5	
	4	
	3	
	2	
	1	
	0	

Nachricht ID 0x120		
Byte	Bit	Bedeutung
0	7	Engine Torque Actual Valid (Valid=0, Invalid=1)
	6	
	5	
	4	
	3	
	2	
	1	
	0	
1	7	Engine Torque Actual 12 Bit
	6	Phys= N*0,25-200 --> -200 - 823,75 Nm)
	5	
	4	
	3	
	2	
	1	
	0	
2	7	
	6	
	5	
	4	
	3	
	2	
	1	
	0	
3	7	
	6	
	5	
	4	
	3	
	2	
	1	
	0	
4	7	
	6	
	5	
	4	
	3	
	2	
	1	
	0	
5	7	
	6	
	5	
	4	
	3	
	2	
	1	
	0	
6	7	
	6	
	5	
	4	
	3	
	2	
	1	
	0	
7	7	
	6	
	5	
	4	
	3	
	2	
	1	
	0	

Nachricht ID 0x180		
Byte	Bit	Bedeutung
0	7	Steering Wheel Angle Protection Value Valid (Valid=0, Invalid=1)
	6	Steering Wheel Angle Valid (Valid=0, Invalid=1)
	5	
	4	
	3	
	2	
	1	
	0	
1	7	Steering Wheel Angle Protection Value 8 Bit
	6	Phys= N*1 --> 0-255
	5	
	4	
	3	
	2	
	1	
	0	
2	7	Steering Wheel Angle 16 Bit
	6	Phys= N*0,0625 --> -2048 - 2047,94° deg
	5	
	4	
	3	
	2	
	1	
	0	
3	7	
	6	
	5	
	4	
	3	
	2	
	1	
	0	
4	7	
	6	
	5	
	4	
	3	
	2	
	1	
	0	
5	7	
	6	
	5	
	4	
	3	
	2	
	1	
	0	
6	7	
	6	
	5	
	4	
	3	
	2	
	1	
	0	
7	7	
	6	
	5	
	4	
	3	
	2	
	1	
	0	

Nachricht ID 0x280		
Byte	Bit	Bedeutung
0	7	Wheel RPM driven Left Valid (Valid=0, Invalid=1)
	6	
	5	Wheel RPM driven Left 14 oder 15 Bit
	4	Phys= N*0,25
	3	
	2	
	1	
	0	
1	7	
	6	
	5	
	4	
	3	
	2	
	1	
	0	
2	7	Wheel RPM nondriven Left Valid (Valid=0, Invalid=1)
	6	
	5	Wheel RPM nondriven Left 14 oder 15 Bit
	4	Phys= N*0,25
	3	
	2	
	1	
	0	
3	7	
	6	
	5	
	4	
	3	
	2	
	1	
	0	
4	7	Wheel RPM driven Right Valid (Valid=0, Invalid=1)
	6	
	5	Wheel RPM driven Right 14 oder 15 Bit
	4	Phys= N*0,25
	3	
	2	
	1	
	0	
5	7	
	6	
	5	
	4	
	3	
	2	
	1	
	0	
6	7	Wheel RPM nondriven Right Valid (Valid=0, Invalid=1)
	6	
	5	Wheel RPM nondriven Right 14 oder 15 Bit
	4	Phys= N*0,25
	3	
	2	
	1	
	0	
7	7	
	6	
	5	
	4	
	3	
	2	
	1	
	0	

Nachricht ID 0x290

Byte	Bit	Bedeutung
0	7	Wheel Rotation Left Driven Rolling Count Valid (Valid=0, Invalid=1)
	6	
	5	Wheel Rotation Left Driven Rolling Count 14 Bit
	4	Phys= N*0,0078125 --> 0 - 127,992 rot
	3	
	2	
	1	
	0	
1	7	
	6	
	5	
	4	
	3	
	2	
	1	
	0	
2	7	Wheel Rotation Left Nondriven Rolling Count Valid (Valid=0, Invalid=1)
	6	
	5	Wheel Rotation Left Nondriven Rolling Count 14 Bit
	4	Phys= N*0,0078125 --> 0 - 127,992 rot
	3	
	2	
	1	
	0	
3	7	
	6	
	5	
	4	
	3	
	2	
	1	
	0	
4	7	Wheel Rotation Right Driven Rolling Count Valid (Valid=0, Invalid=1)
	6	
	5	Wheel Rotation Right Driven Rolling Count 14 Bit
	4	Phys= N*0,0078125 --> 0 - 127,992 rot
	3	
	2	
	1	
	0	
5	7	
	6	
	5	
	4	
	3	
	2	
	1	
	0	
6	7	Wheel Rotation Right Nondriven Rolling Count Valid (Valid=0, Invalid=1)
	6	
	5	Wheel Rotation Right Nondriven Rolling Count 14 Bit
	4	Phys= N*0,0078125 --> 0 - 127,992 rot
	3	
	2	
	1	
	0	
7	7	
	6	
	5	
	4	
	3	
	2	
	1	
	0	

Nachricht ID 0x510

Byte	Bit	Bedeutung
0	7	Engine Coolant Temperatur Valid (Valid=0, Invalid=1)
	6	
	5	
	4	
	3	
	2	
	1	
	0	
1	7	Engine Coolant Temperatur 8 Bit
	6	Phys= N*1-40
	5	
	4	
	3	
	2	
	1	
	0	
2	7	
	6	
	5	
	4	
	3	
	2	
	1	
	0	
3	7	
	6	
	5	
	4	
	3	
	2	
	1	
	0	
4	7	
	6	
	5	
	4	
	3	
	2	
	1	
	0	
5	7	
	6	
	5	
	4	
	3	
	2	
	1	
	0	
6	7	
	6	
	5	
	4	
	3	
	2	
	1	
	0	
7	7	
	6	
	5	
	4	
	3	
	2	
	1	
	0	