Dokumentation

CAN-Bus
Controller Area Network

Gliederung

- CAN-Bus
  - Funktionsweise (Signalübertragung)
- OBD
- Diagnosekommunikation (ELM327 als Vermittler)
- C++ Programm

CAN-Bus

- Bus-Topologie → lineare Anordnung

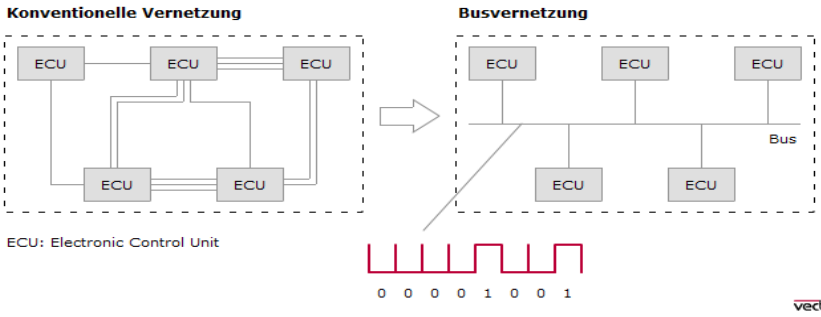


Abb. 1 Bustopologie

- 2-Draht-Bussystem → Twisted Pair:
  Can-high - und Can-low - Leitung
- Die Steuergeräte sind über kurze Stichleitungen an den Bus angeschlossen
- jedes Steuergerät verfügt über eine Busanbindung, die aus einem CAN-Controller und einem Transceiver besteht

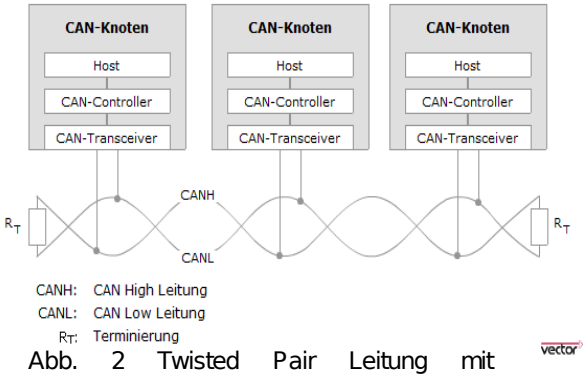


Abb. 2 Twisted Pair Leitung mit angeschlossenen Steuergeräten

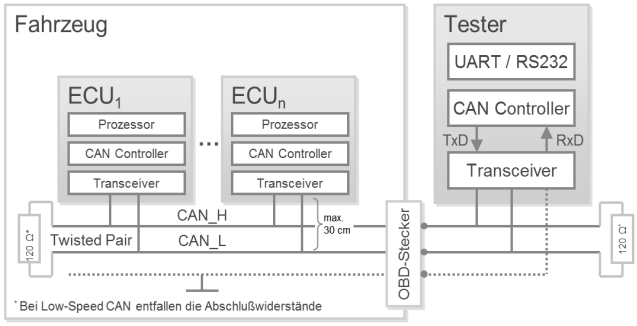


Abb. 3 Leitung mit Tester

- 3 verschiedene CAN Klassen, die sich in Übertragungsgeschwindigkeiten unterscheiden:
  - A Diagnose
  - B Komfort Systeme (Low-Speed-CAN)
  - C Antrieb und Fahrwerkkontrolle (High-Speed-CAN)

Funktionsweise

- Kommunikationsprinzip: Multi-Master Prinzip
  → jedes Steuergerät kann mit jedem anderen Steuergerät kommunizieren
- Steuergeräte können senden und empfangen gleichzeitig
- CAN Controller verarbeitet die vom Mikrocontroller/-prozessor im Steuergerät oder vom Transceiver gesendeten Daten und leitet sie weiter
- Transceiver wandelt die empfangenen Daten in ein elektrisches Signal um und sendet sie über die Busleitung an die anderen Steuergeräte und umgekehrt

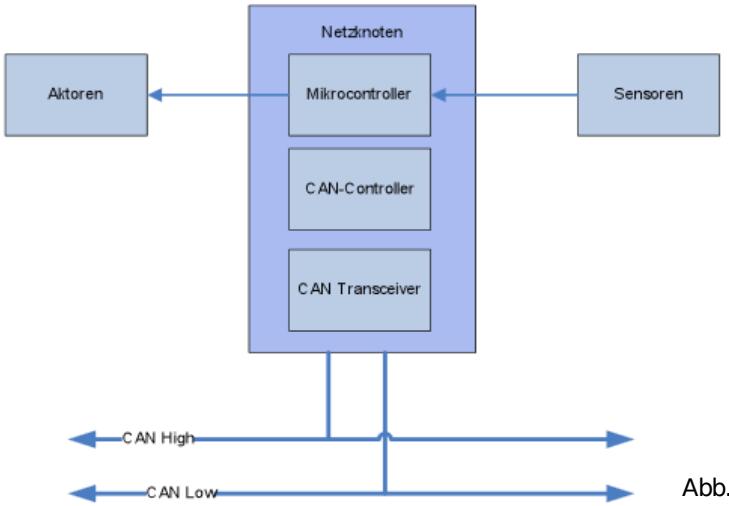


Abb. 4 Netzknotenanbindung

- Signalübertragung von Spannungen:

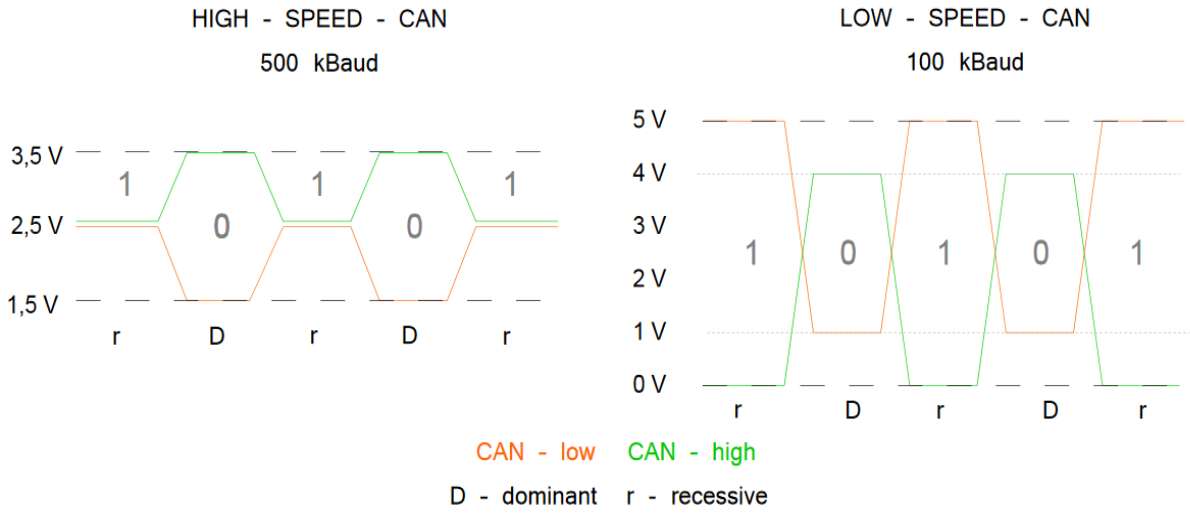


Abb. 5 Signalübertragung

- logische [1] = rezessiv logische [0] = dominant

- äußere Störungen wirken sich auf beide Leitungen gleichermaßen aus
- Signal der einen Leitung ist komplementär zu dem Signal der anderen Leitung → Störungsaufhebung

- Datenübertragung erfolgt in seriellen Datenpaketen, deren Aufbau standardisiert ist
- Datenprotokoll:
  - Anfang 1 Bit → Anfang = dominant
  - Statusfeld (Identifier) 11 bzw. 29 Bit → Datenart / Inhalt & Priorität
  - RTR 1 Bit → Daten anfordern [1] oder senden [0] (Remote Transmission Request)
  - Kontrollfeld 6 Bit → Anzahl der Datenbits (0–8 Bytes)
  - Datenfeld max. 64 Bit → Daten
  - Sicherungsfeld 16 Bit → Erkennen von Übertragungsfehlern durch „Prüfsumme“ aus Daten
  - Bestätigungsfeld 2 Bit → Empfänger bestätigen korrekten Empfang

    Fehler [0 1] / kein Fehler [1 1]
  - Ende 7 Bit → 7 x [1]
  - Ruhezustand min. 3 Bit → trennen aufeinanderfolgende Nachrichten

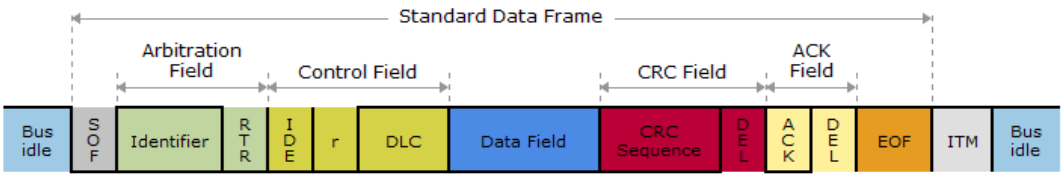
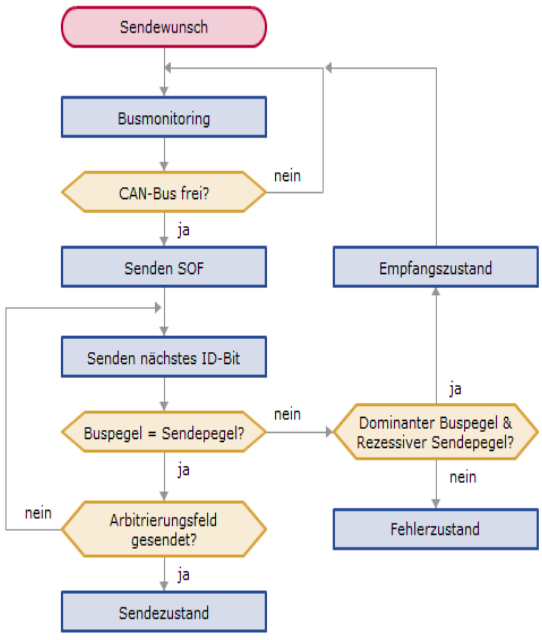


Abb. 6 Datenprotokoll

vector

- bitweise Arbitrierung
  - wenn mehrere Steuergeräte gleichzeitig den Sendevorgang beginnen, muss entschieden werden, wer Vorrang hat
  - die Nachricht mit der höchsten Priorität darf senden
  - entsprechend der Priorität wird jeder Nachricht ein Identifier aus 11 bzw. 29 Bit zugeordnet
  - Jeder sendende Teilnehmer vergleicht den von ihm gesendeten Buspegel mit dem Pegel, der tatsächlich auf dem Bus liegt
  - Die Teilnehmer senden den Identifier ihrer Nachricht bis diese sich in einem Bit unterscheiden → [0] überschreibt [1]
  - Die Teilnehmer, die eine [1] gesendet und eine [0] beobachtet, stellt seinen Sendevorgang sofort ein
  - → niedriger Identifierwert = hohe Priorität
  - Bei verlorener Arbitrierung wird das Senden automatisch wiederholt, sobald der Bus wieder frei ist



vector

Abb. 7 Arbitrierung

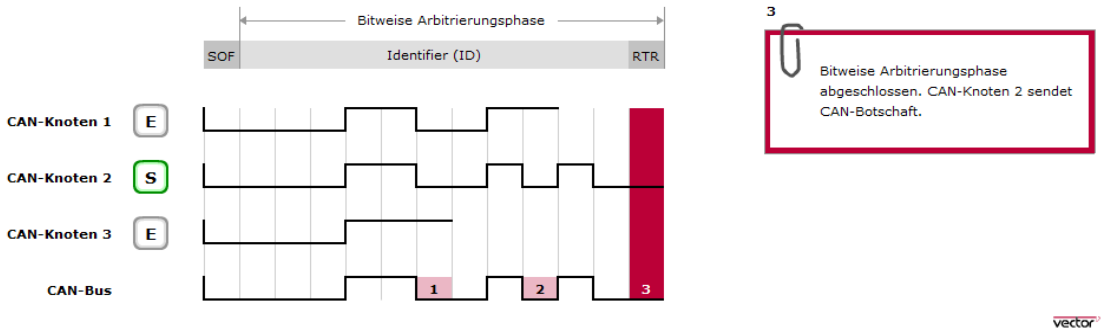


Abb. 8 Beispiel zur Arbitrierung

- Auch wenn ein Empfänger einen Fehler meldet, wird das Senden sofort wiederholt
- Stuff-Bits
  - werden eingefügt, wenn 5 Bits hintereinander mit dem selben Signal gesendet werden
  - dienen der Synchronisation von Sender und Empfänger
  - Stuff-Bits sieht man nur, wenn man die Spannungen auf dem Bus mit einem Oszilloskop misst
  - Beim Senden und Empfangen werden Stuff-Bits automatisch eingefügt und wieder heraus genommen

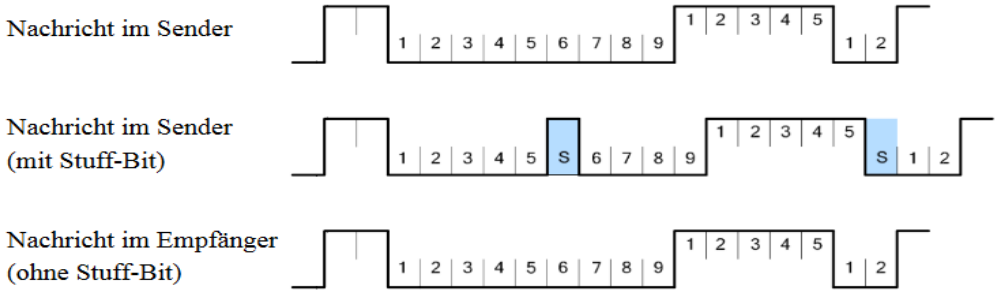


Abb. 9 Stuff-Bits

OBD – On-Board-Diagnose

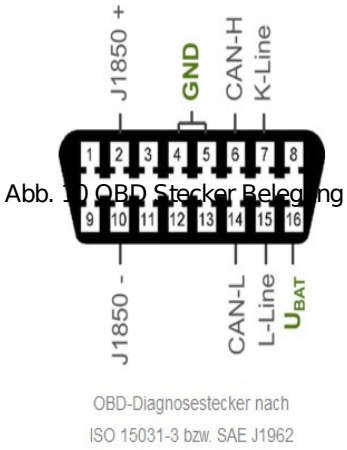


Abb. 10 OBD Stecker Belegung

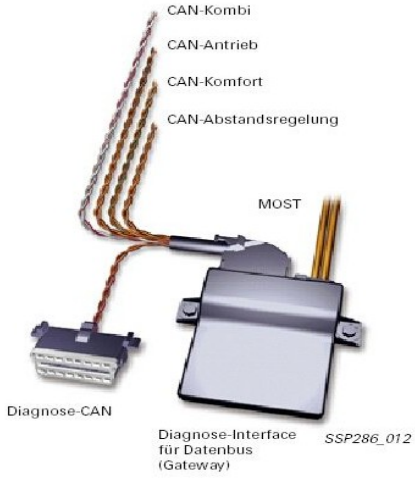


Abb. 11 Übersicht mit Gateway

- Diagnoseschnittstelle
- Überwachung aller abgasrelevanten Daten in allen Fahrzeugen
- Speicherung von auftretenden Fehlern
- Schnittstelle um gespeicherte und Echtzeit- Daten auszulesen

Diagnosekommunikation

Abb. 12 verwendeter Elm327

- Elm237 dient als Vermittler zwischen dem CAN-Bus und dem Computer
- Kommunikation erfolgt nur bei Bedarf
- Kommunikationsprinzip: Request – Response
- Rollenverteilung festgelegt:
  Diagnosetester fragt an → ein oder mehrere Steuergeräte antworten

- SID (Service Identifier) kennzeichnet den Inhalt der Botschaft
  - 0x01 – Abfrage von Messdaten
  - 0x03 – Fehlercodes lesen
  - 0x09 – Abfrage von Fahrzeuginformationen wie Fahrgestellnummer
  - ...
- PID (Parameter Identifier) Umrechnung
  - 0x00 – PID's, die unterstützt werden
  - 0x05 – Kühlwassertemperatur A-40
  - 0x0C – Motorumdrehungen (256*A + B)/4
  - 0x0D – Geschwindigkeit A
  - 0x1F – Zeit seit Motorstart 256*A + B
  - ...

Allgemein:

Request	SID	PID	
Response	SID + 0x40	PID	Data

Beispiel:

Request	01	05	
Response	41	05	66

0x66 → 102 102-40 = 62°C

Abb. 13 Request – Response Kommunikation

C++ -Program

- 3 verschiedene Teile:
  - Header-Datei mit einer Klasse
  - C++-Datei mit den Methoden der Klasse
  - Main-Programm mit weiteren Funktionen

Header-Datei

```
#ifndef SERIALCLASS_H_INCLUDED
#define SERIALCLASS_H_INCLUDED

#define ARDUINO_WAIT_TIME 2000

#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

class Serial
{
    private:
        //Serial comm handler
        HANDLE hSerial;
        //Connection status
        bool connected;
        //Get various information about the connection
        COMSTAT status;
        //Keep track of last error
        DWORD errors;

    public:
        //Initialize Serial communication with the given COM port
        Serial(char *portName);
        //Close the connection
        ~Serial();
        //Read data in a buffer, if nbChar is greater than the
        //maximum number of bytes available, it will return only the
        //bytes available. The function return -1 when nothing could
        //be read, the number of bytes actually read.
        int ReadData(char *buffer, unsigned int nbChar);
        //Writes data from a buffer through the Serial connection
        //return true on success.
        bool WriteData(char *buffer, unsigned int nbChar);
        //Check if we are actually connected
        bool IsConnected();


};


#endif // SERIALCLASS_H_INCLUDED
```

Serial.h-Datei

```
#include "SerialClass.h"

Serial::Serial(char *portName)
{
    this->connected = false;                                    //We're not yet connected
    //Try to connect to the given port throuh CreateFile
    this->hSerial = CreateFile(portName,
            GENERIC_READ | GENERIC_WRITE,
            0,
            NULL,
            OPEN_EXISTING,
            FILE_ATTRIBUTE_NORMAL,
            NULL);
    if(this->hSerial==INVALID_HANDLE_VALUE)     //Check if the connection was successfull
    {
        if(GetLastError()==ERROR_FILE_NOT_FOUND)      //If not success full display an Error
            {
            //Print Error if neccessary
            printf("ERROR: Handle was not attached. Reason: %s not available.\n", portName);
        }
        else
        {
            printf("ERROR!!!");
        }
    }
    else
    {
        DCB dcbSerialParams = {0};     //If connected we try to set the comm parameters
        if (!GetCommState(this->hSerial, &dcbSerialParams))                 //Try to get the current
        {
            //If impossible, show an error
            printf("failed to get current serial parameters!");
        }
        else
        {
            //Define serial connection parameters for the arduino board
            dcbSerialParams.BaudRate=CBR_38400;
            // dcbSerialParams.BaudRate=CBR_9600;
            dcbSerialParams.ByteSize=8;
            dcbSerialParams.StopBits=ONESTOPBIT;
            dcbSerialParams.Parity=NOPARITY;
            //Setting the DTR to Control_Enable ensures that the Arduino is properly
            //reset upon establishing a connection
            dcbSerialParams.fDtrControl = DTR_CONTROL_ENABLE;
             //Set the parameters and check for their proper application
             if(!SetCommState(hSerial, &dcbSerialParams))
             {
                printf("ALERT: Could not set Serial Port parameters");
             }
             else
             {
                 //If everything went fine we're connected
                 this->connected = true;
                 //Flush any remaining characters in the buffers
                 PurgeComm(this->hSerial, PURGE_RXCLEAR | PURGE_TXCLEAR);
                 //We wait 2s as the arduino board will be reseting
                 Sleep(ARDUINO_WAIT_TIME);
             }
        }
    }
}
```

```
Serial::~Serial()
{
    if(this->connected)            //Check if we are connected before trying to disconnect
    {
        this->connected = false;        //We're no longer connected
        CloseHandle(this->hSerial);             //Close the serial handler
    }
}
int Serial::ReadData(char *buffer, unsigned int nbChar)
{
    DWORD bytesRead;                //Number of bytes we'll have read
    unsigned int toRead;                //Number of bytes we'll really ask to read
    //Use the ClearCommError function to get status info on the Serial port
    ClearCommError(this->hSerial, &this->errors, &this->status);
    if(this->status.cbInQue>0)          //Check if there is something to read
    {
        //If there is we check if there is enough data to read the required number
        //of characters, if not we'll read only the available characters to prevent
        //locking of the application.
        if(this->status.cbInQue>nbChar)
        {
            toRead = nbChar;
        }
        else
        {
            toRead = this->status.cbInQue;
        }
        //Try to read the require number of chars, and return the number of read bytes on success
        if(ReadFile(this->hSerial, buffer, toRead, &bytesRead, NULL) )
        {
            return bytesRead;
        }
    }
    return 0;               //If nothing has been read, or that an error was detected return 0
}
bool Serial::WriteData(char *buffer, unsigned int nbChar)
{
    DWORD bytesSend;
    //Try to write the buffer on the Serial port
    if(!WriteFile(this->hSerial, (void *)buffer, nbChar, &bytesSend, 0))
    {
        //In case it don't work get comm error and return false
        ClearCommError(this->hSerial, &this->errors, &this->status);
        return false;
    }
    else
        return true;
}
bool Serial::IsConnected()
{
    return this->connected;             //Simply return the connection status
}
```

Main-Programm

```
#include <stdio.h>
#include <tchar.h>
#include "SerialClass.h"        // Library described above
#include <string>
#include <fstream>
#include <iomanip>
#include <iostream>

using namespace std;

const char* pfad1 = "C:\\Users\\hile\\Desktop\\Schwarz-Programm\\versuch1.txt";
const char* pfad2 = "C:\\Users\\hile\\Desktop\\Schwarz-Programm\\daten1.txt";
fstream versuch(pfad1, ios::out|ios::app);
fstream daten(pfad2, ios::out|ios::app);

void write(Serial*SP,char *data, int &len)
{
    char data_write[200] = {0};
    int nbChar = 0;
    cout << endl << "_____" << endl;
    versuch << endl << "_____" << endl;
    strcpy(data_write, data);
    nbChar = len;
    if(SP->WriteData(data_write,nbChar))
    {
        for(int i=0;i<nbChar;i++)
            if(data_write[i] == 0x0d)
                data_write[i] = 0x0a;
        for(int i=0;i<nbChar;i++)
        {
           cout << data_write[i];
           versuch << data_write[i];
        }

        cout << " wurde gesendet" << endl;
        versuch << " wurde gesendet" << endl;
    }
    Sleep(1000);
    versuch.close();
    return;
}

void read(Serial*SP, char *data_read, int &received_Bytes, int &nbbytes)
{
    received_Bytes = SP->ReadData(data_read,nbbytes);
    if(received_Bytes > 0)
    {
        data_read[received_Bytes] = 0;
        for(int i=0;i<received_Bytes;i++)
            if(data_read[i] == 0x0d)
                data_read[i] = 0x0a;
    }
    return;
}
```

```
void supported(char*data_read, int&received_Bytes)                    //die Funktion funktioniert leider
nicht
{
   for(int i=0;i<received_Bytes;i++)
   {
      char buffer[32];
      itoa(data_read[i],buffer,2);
      cout << setw(4) << setfill('0') << buffer << " ";
      versuch << setw(4) << setfill('0') << buffer << " ";
      daten << setw(4) << setfill('0') << buffer << " ";
   }
   cout << endl;
   versuch << endl;
   daten << endl;
   versuch.close();
   daten.close();
   return;
}

int value(char*data_read,int i, int j)
{
   char* buff[100];
   char conv[10] = "0x00";
   conv[2] = data_read[i];
   conv[3] = data_read[j];
   int value = strtol(conv,buff,16);
   return value;
}

void geschw(char*data_read)
{
   daten << "Geschwindigkeit: ";
   versuch << "Geschwindigkeit: ";
   int A = value(data_read,12,13);
   versuch << dec << A << " " << "km/h" <<endl;
   daten << dec << A << " " << "km/h" <<endl;
   versuch.close();
   daten.close();
   return;
}
void umdreh(char*data_read)
{
   daten << "Motor-Umdrehungen: ";
   versuch << "Motor-Umdrehungen: ";
   int A = value(data_read,12,13);
   int B = value(data_read,15,16);
   int erg = ((256*A)+B)/4;
   versuch << dec << erg << " " << "rpm" << endl;
   daten << dec << erg << " " << "rpm" << endl;
   versuch.close();
   daten.close();
   return;
}
void temp(char*data_read)
{
   daten << "Motor-Kuehlwassertemperatur: ";
   versuch << "Motor-Kuehlwassertemperatur: ";
   int A = value(data_read,12,13);
   int erg = A-40;
   versuch << dec << erg << " " << "°C" << endl;
   daten << dec << erg << " " << "°C" << endl;
   versuch.close();
   daten.close();
   return;
```

```
}
```

```
void zeit(char*data_read)
{
   daten << "Zeit seit Motorstart: ";
   versuch << "Zeit seit Motorstart: ";
   int A = value(data_read,12,13);
   int B = value(data_read,15,16);
   int erg = (256*A)+B;
   versuch << dec << erg << " " << "s" << endl;
   daten << dec << erg << " " << "s" << endl;
   versuch.close();
   daten.close();
   return;
}

void print(Serial*SP,char *data, int &len, int sleeptime)
{
   char data_read[200] = {0};
   int nbbytes = 200;
   int received_Bytes = 0;

   write(SP, data, len);
   Sleep(sleeptime);
   read(SP,data_read, received_Bytes, nbbytes);

   if(received_Bytes > 0)
   {
      cout << "Anzahl zurueckgeschickter Bytes: " << received_Bytes << endl;
      versuch << "Anzahl zurueckgeschickter Bytes: " << received_Bytes << endl;
      if(!strcmp(data,"01 00\r"))
      {
         for(int i=0;i<received_Bytes;i++)
         {
            cout << data_read[i];
            versuch << data_read[i];
         }
         cout << endl;
         versuch << endl;
         for(int i=0;i<received_Bytes;i++)
         {
            printf("%x ",data_read[i]);
            versuch << hex << int (data_read[i]) << " ";
         }
         cout << endl;
         versuch << endl;
         supported(data_read,received_Bytes);
      }
      else
      {
         for(int i=0;i<received_Bytes;i++)
         {
            cout << data_read[i];
            versuch << data_read[i];
         }
         cout << endl;
         versuch << endl;
         for(int i=0;i<received_Bytes;i++)
         {
            printf("%x ",data_read[i]);
            versuch << hex << int (data_read[i]) << " ";
         }
         cout << endl;
         versuch << endl;

         if(!strcmp(data,"01 0D\r"))
```

```
            {
                geschw(data_read);
            }
            else if(!strcmp(data,"01 0C\r"))
            {
                umdreh(data_read);
            }
            else if(!strcmp(data,"01 05\r"))
            {
                temp(data_read);
            }
            else if(!strcmp(data,"01 1F\r"))
            {
                zeit(data_read);
            }
            else
            {
                daten << endl << endl << endl;
                for(int i=0;i<received_Bytes;i++)
                {
                    daten << data_read[i];
                }
                daten << endl;
            }
        }
    }
    versuch.close();
    daten.close();
}

int _tmain(int argc, _TCHAR* argv[])
{
            printf("Welcome to the serial test app!\n\n");
            Serial* SP = new Serial("\\\\.\\COM6");    //Com23
            if (SP->IsConnected())
    {
        cout << "We're connected" << endl;
    }
    else
    {
        return -1;
    }

    char data[200] = {0};
    int len = 0;
    data[0] = 0x0D;
    len = 1;
    print(SP,data,len,0);
    strcpy(data, "atz\r");
    len = 4;
    print(SP,data,len,0);
    strcpy(data, "atz\r");
    len = 4;
    print(SP,data,len,0);
    print(SP,data,len,0);

    ///Welche PIDs werden unterstützt
    strcpy(data, "01 00\r");
    len = 6;
    print(SP,data,len,500);

    ///Geschwindigkeit, Motorumdrehung, Kühltemperatur und Zeit seit Motorstart auslesen als
Schleife
    for(int g=0;g<500;g++)
```

```
    {
        strcpy(data, "01 0D\r");
        len = 6;
        print(SP,data,len,500);

        strcpy(data, "01 0C\r");
        len = 6;
        print(SP,data,len,500);

        strcpy(data, "01 05\r");
        len = 6;
        print(SP,data,len,500);

        strcpy(data, "01 1F\r");
        len = 6;
        print(SP,data,len,500);
    }

    SP->~Serial();
    return 0;
}
```

Quellen:

http://kfztech.de/kfztechnik/elo/can/can_grundlagen_1.htm

https://www.emotive.de/index.php/de/doc/car-diagnostic-systems/contents
oder: http://www.emotive.de/wiki/index.php?title=Category:Fahrzeugdiagnose

http://www.fachschule-fuer-technik-mhl.de/downloads/projekte/pa_kfz_001.pdf

https://prof.beuth-hochschule.de/uploads/media/AusarbeitungCAN-Bus.pdf

https://en.wikipedia.org/wiki/OBD-II_PIDs

Bildquellen:

Abb. 1, 2, 6, 7, 8
http://elearning.vector.com/index.php?wbt_ls_kapitel_id=1329921&root=376493&seite=vl_can_introduction_de

Abb. 3
https://www.emotive.de/index.php/de/doc/car-diagnostic-systems/bus-systems/can

Abb. 4, 9
https://prof.beuth-hochschule.de/uploads/media/AusarbeitungCAN-Bus.pdf

Abb. 10
http://www.emotive.de/wiki/index.php?title=OBD_on_CAN

Abb. 11
http://kfztech.de/kfztechnik/elo/can/can_grundlagen_1.htm

Abb. 12
http://www.autodia.de/produkte/autodia-e327