

Projekt Systementwicklung: OnBoard Diagnose und Telemetrie

vom:
20.09.2010 – 01.10.2010

Dokumentation der OBD-Gruppe

Mitglieder:

Christoph Becker
Dirk Vogel
Stephan Fries
Dario Cundari
Andreas Schröder

Dozent:

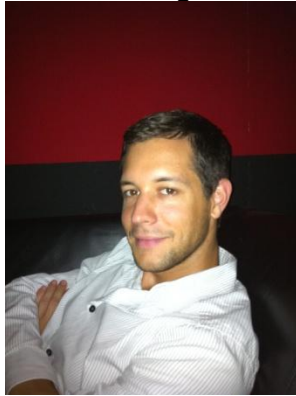
Prof. Dr. Joachim Wietzke

Bilder der Personen:

Christoph Becker



Dirk Vogel



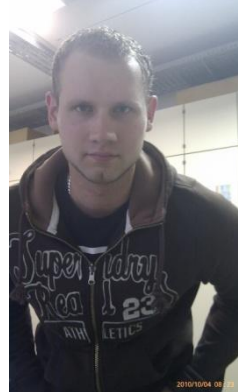
Stephan Fries



Dario Cundari



Andreas Schröder



Inhaltsverzeichnis

1. Übersicht	5
1.1 Protokolle	5
1.2 Protokollinterpreter	5
1.3 Simulator	6
2. Anbindung des ELM-Chipsatzes via Serielle Schnittstelle	8
3. Anbindung des ELM-Chipsatzes via Bluetooth	12
3.1 Vor der Realisierung festgelegte Anforderungen	12
3.2 Inwieweit erfüllt das Modul die festgelegten Anforderungen?	12
3.3 Entwicklungsumgebung	12
3.4 Struktur des Moduls	13
3.4.1 Verbindungsaufbau	14
3.4.2 Initialisierung	15
3.4.3 Verbindungsabbau	15
3.5 Prinzip von get1Byte() und get2Byte()	15
3.6 Testdurchführungen	17
3.6.1 Erfolgreiche Tests	17
3.6.2 Gescheiterte Tests	17
3.7 Überlegungen zur Realisierung weiterer Schritte	17
4. Auswertung der Daten mit der ELM-Klasse	18
4.1 OBD2 PID's	18
4.2 Schnittstellenspezifikation	18
4.3 Allgemeines zur Klasse	19
4.4 Aufbau und Funktionsweise der get-Funktionen	20
4.5 Zusätzliche Informationen	21
4.5.1 ELM-Klasse in Verbindung mit der CAN- und GPS-Klasse	21
4.5.2 Private-Methoden der ELM-Klasse	21
5. Anwendungsbibliothek	22
5.1 Begriffsdefinition	22
5.2 Hinweise für die Weiterentwicklung	22
5.2.1 Allgemeine Hinweise	22
5.2.2 Spezielle Hinweise	22
5.3 Projekt kompilieren	23
5.3.1 Benötigte Bibliotheken	23
5.3.2 Kompilieren in Eclipse	24

5.3.3 Kompilieren in der Kommandozeile.....	25
5.4 Quellcode-Dokumentation erstellen.....	25
5.5 Klassendiagramme	26
5.5.1 Klasse Connection	26
5.5.2 Klasse OBD	27
5.5.3 Klasse Car	28
5.5.4 Klasse ServicesStatusList.....	30
6. GPS-Modul	31
6.1 Beschreibung der wichtigsten Klassen und Methoden	33
6.2 Einbindung in die Anwendungsbibliothek	34
7. Kamerakomponente	35
7.1 Verwendung anderer USB-Kameras unter Linux	37
8. Links.....	38
8.1 OBD2 / ELM	38
8.2 Serielle Verbindung.....	38
8.3 Bluetooth.....	38
8.4 GPS.....	38
8.5 Kamera.....	38
8.6 Wiki-Links	39
8.7 Perforce.....	39
8.8 Sonstiges	39
9. Literatur	39

1.Übersicht

Ziel unserer Arbeit war es die Daten, die über die OBD-Schnittstelle zur Verfügung stehen auszulesen und in einer Anwendungsbibliothek den Gruppen in einer höheren Abstraktionsebene und unkomplizierten Art und Weise zur Verfügung zu stellen.

Zielfahrzeug war ein Opel Astra 1.7CDTI auf den die Software abgestimmt wurde und demnach eventuell in Zukunft noch auf andere Fahrzeuge bzw. Protokolle portiert werden muss.

1.1 Protokolle

Je nach Fahrzeug können andere Protokolle zum Einsatz kommen über die, die OBD Informationen laufen. Demnach kommen auch je nach Protokoll die Daten in leicht unterschiedlicher Form an.

Protokollübersicht:

- ISO 9141-2
- ISO 14230-4 (KWP2000 fast und slow)
- ISO 15765-4 (CAN 250/500 kB 11/29 Bit ID)
- SAE J1850 PWM und VPW

1.2 Protokollinterpreter

Um OBD-Daten am Fahrzeug auszulesen wurde ein Protokollinterpreter genutzt, dieser besitzt einen Befehlssatz und kann (je nach Modell) entweder über die Serielle Schnittstelle oder Bluetooth angesprochen werden. So kann unabhängig vom Protokoll des jeweiligen Fahrzeuges, mit ein und demselben Interface auf relativ viele Fahrzeugtypen zugegriffen werden.

Wir entschieden uns für das ElmCan Interface auf Basis des ELM-327 Chipsatzes (v1.3a). Einmal mit Serieller und einmal mit Bluetooth Anbindung.

Der ElmCan:



Unsere Wahl fiel deshalb auf diesen Controller, da er 1. den umfangreichsten Befehlssatz besitzt und viele andere Controller wie z.B. der AGV4000 Abwärtskompatibel zu diesem Controller sind. Und 2. Da dieser Controller die meisten Protokolle unterstützt (Alle unter 1.1 genannten).

Eine Übersicht des Befehlssatzes befindet sich auf folgender Seite:

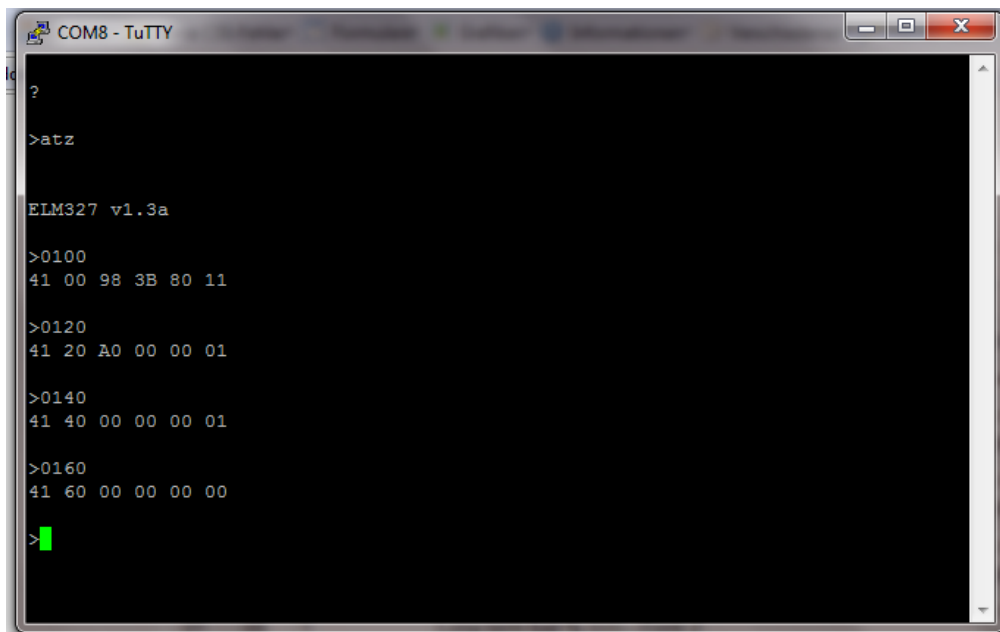
<http://elmelectronics.com/dsheets.html>

Der ELM arbeitet mit 38.400 Baud, 8 Datenbit, 1 Stoppbit, keine Parität.

Um den ELM via Terminal anzusprechen eignen sich unter anderem folgende Programme:

- Minicom (Linux)
- Hyperterminal (Windows XP)
- TuTTY (Windows Vista/7)

Ein Auszug wie die Benutzung via Terminal aussehen kann:



```
COM8 - TuTTY
?
>atz
ELM327 v1.3a
>0100
41 00 98 3B 80 11
>0120
41 20 A0 00 00 01
>0140
41 40 00 00 00 01
>0160
41 60 00 00 00 00
>
```

Hierbei testeten wir das Ganze an dem Opel Astra. Der Befehl „atz“ bewirkt ein restart des ELM die folgenden PID's (Parameter ID's) 0100, ...,0160 fragen die vom Fahrzeug unterstützten Daten ab.

Software für den ElmCan:

- [ScanMasterELM](#)
- [ScanMaster-PPC](#)
- [PCMSCAN](#)
- [Pocket Scanner Plus](#)
- [PowerDyn](#)
- [Rev](#) (nur über iTunes App Store)

1.3 Simulator

Um nicht immer zum Testen ans Auto zu müssen hatten wir einen Simulator zur Verfügung. Es handelte sich hierbei um den [Diamex AGV MiniSim](#).

Diamex AGV MiniSim:



Dieser ermöglicht die Simulation von Fehlercodes, unterstützt verschiedene Protokolle, bis zu 2 ECU's und es lassen sich Drehzahl, Kühlmitteltemperatur und Geschwindigkeit über Potentiometer einstellen.

Dies erleichtert natürlich die Entwicklung erheblich da direkt im Labor getestet und entwickelt werden kann. Leider stand uns dieser Simulator nur 3 Tage zur Verfügung, da er gleich am Anfang kaputt ging und erst gegen Ende der 2. Woche ein Austausch Simulator vom Hersteller zurückkam.

Durch den Simulator lässt sich unter anderem sehr schön herausfinden in welcher Form die verschiedenen Protokolle antworten. Somit kann man, wenn in Zukunft weitere Protokolle unterstützt werden sollen, die Software direkt auf die verschiedenen Protokolle anpassen.

2. Anbindung des ELM-Chipsatzes via Serielle Schnittstelle

Die Klasse Serial ist dafür zuständig einen Port für die serielle Übertragung zu öffnen und auf diesem dann Daten zu übertragen.

Es ist so gehandhabt, dass alle üblichen virtuellen USB Ports durchprobiert werden und für den geöffneten Port wird eine Meldung ausgegeben.

Das Öffnen des Ports und die Initialisierung geschieht beim Aufruf des Konstruktors automatisch.

Die Schnittstelle wird im sogenannten „Raw“ Modus benutzt.

Das bedeutet, dass es keine Hard- oder Software Flusssteuerung gibt, sondern die Daten einfach ankommen, wann sie ankommen und wie sie ankommen.

Durch eine while Schleife soll sichergestellt werden, dass immer die richtige Anzahl Bytes die benötigt wird auch empfangen werden. Fall es dabei zu Fehlern kommt wird man mit Hilfe einer Exception informiert.

Nachdem der serielle Port (Hier ttyUSB0) mit „open()“ geöffnet wurde, kann mit „read()“ und „write“ darauf zugegriffen werden.

Die Schnittstelle ist außerdem auf 8N1 (8 Datenbit ein Stoppbit, keine Parität) und die Baudrate 38400 eingestellt. Sämtliche Einstellungen werden in der Struct „termios“ vorgenommen.

```
struct termios {
    tcflag_t c_iflag;    /* input specific flags (bitmask) */
    tcflag_t c_oflag;    /* output specific flags (bitmask) */
    tcflag_t c_cflag;    /* control flags (bitmask) */
    tcflag_t c_lflag;    /* local flags (bitmask) */
    cc_t      c_cc[NCCS]; /* special characters */
};
```


Beispiel Initialisierung mit struct termios options:

```
struct termios options;

fcntl(fd, F_SETFL, 0);

if(tcgetattr(fd, &options) != 0) return (-1);

cfsetspeed(&options, B38400);           //Baudrate auf 38400

options.c_cflag &= ~(PARENB);           //Kein Parity bit
options.c_cflag &= ~(CSTOPB);           //1 Stopbit
options.c_cflag &= ~(CSIZE);           // Datenbits
options.c_cflag |= CS8;                 //8 Datenbits

options.c_cflag |= (CLOCAL | CREAD )    ;
//options.c_iflag = (IGNPAR | ICRNL);
options.c_lflag &= ~(ICANON | ECHOE|ECHO|ISIG);           //kein echo
//options.c_oflag = 0;

options.c_oflag &= ~(OPOST)    ;
//setzte "raw" input
options.c_cc[VMIN] = 15        ;
options.c_cc[VTIME] = 15       ;

tcflush(fd, TCIOFLUSH);
tcsetattr(fd,TCSANOW,&options);
if(tcsetattr(fd, TCSAFLUSH, &options) != 0) return (-1); //Falls optionen zu
setzen nicht geklappt hat

//fcntl(fd, F_SETFL, FNDELAY);
fcntl(fd, F_SETFL, 0);
```

Mit Hilfe von Bitmasken werden dann die einzelnen Steuerbits entweder gesetzt oder gelöscht.

Die Serial Klasse implementiert alle Methoden, die durch das Connection Interface vorgegeben wurden:

```
class Serial :public Connection {

private:
    void send(char* PID);
    void clearBuf();
    int openport(int port);
    int getfd();

public:
    Serial();
    virtual ~Serial();
    void clearErrorCodes(void);
    const string getErrorCodes(void);
    const string get1Byte(const string PID);
    const string get2Byte(const string PID);

};
```

Da das Fahrzeug je nach Anfrage entweder 1 oder 2 relevante Bytes als Antwort parat hat, gibt es dafür jeweils eine eigene Methode:

```
string Serial::get1Byte(const char* PID)
{
    strcpy(res1, "");
    tcflush(fd,TCIFLUSH);
    write(fd,PID,5);
    count1 = 0;
    do {
        count1 += read(fd, &empfangen[count1], 12);
        if(count1 > 2 && empfangen[0] != '4' && empfangen[1] != '1')
            break;
    }
    while(count1 < 12);

    empfangen[count1] = 0; // Das selbe wie oben

    if(empfangen[0] != '4' && empfangen[1] != '1') {
        return empfangen;
    }

    //TESTAUSGABE
    //cout << "Empfangen Buffer: " << empfangen << endl;

    res1[0]=empfangen[6];
    res1[1]=empfangen[7];

    return res1;
}
```

Den Methoden wird die benötigte PID als String übergeben.

Sie schicken dann die PID zum ELM und interpretieren die Antwort.

Die Methoden stellen jeweils sicher dass der Serielle Puffer geleert ist und die richtige Anzahl an Daten empfangen wird. Anschließend werden die für die Werte relevanten Daten extrahiert und als Return-Wert übergeben.

In der nächst Höheren Schicht werden dann die nötigen Berechnungen durchgeführt um aus den ASCII Zeichen, die einen Hex-Wert darstellen sollen die endgültigen Werte wie Drehzahl oder Geschwindigkeit zu kreieren.

Die Methode für die Fehlercodes gibt die aufgetretenen Fehler als zusammenhängenden String zurück. Dieser ist bereits von allen irrelevanten Bytes und Leerzeichen befreit.

In der nächst Höheren Ebene werden aus den ASCII(Hex) Zeichen dann die (leider nicht genormten) Fehlercodes erstellt (z.B. P0002).

Bevor die Methoden funktionieren, müssen ein paar Einstellungen am ELM Interface vorgenommen werden:

```
//ELM Initialisieren
tcflush(fd, TCIOFLUSH);
send("\r");
sleep(1);
send("atz\r"); //reset
sleep(1);
send("ate0\r"); //echo aus
sleep(1);
send("at10\r"); //newline aus
sleep(1);
send("010C\r");
sleep(2);
tcflush(fd,TCIOFLUSH);
cout << "...finished!" << endl;
```

Zwischen dem Versenden der einzelnen Steuerbefehle wird jeweils eine Sekunde gewartet, da Tests gezeigt haben, dass ansonsten manche Daten vom ELM ignoriert werden.

Die Steuerbytes werden mit der Methode send() versendet.

3. Anbindung des ELM-Chipsatzes via Bluetooth

3.1 Vor der Realisierung festgelegte Anforderungen

Es soll ein Modul entwickelt werden, das automatisch eine drahtlose Verbindung zum "ELM 327 Bluetooth – Adapter" aufbaut.

Des Weiteren soll das Modul Methoden zur Verfügung stellen, die ein sicheres Auslesen der OBD-Informationen gewährleistet.

3.2 Inwieweit erfüllt das Modul die festgelegten Anforderungen?

Das Modul erfüllt alle festgelegten Anforderungen. Allerdings arbeitet das Modul nicht mit allen Klassen der Library zusammen. Dies wird unter dem Punkt 5.2 nochmal genauer erläutert.

3.3 Entwicklungsumgebung

Dieses Modul wurde mit dem Entwicklungswerzeug "Eclipse C++" unter Ubuntu Linux entwickelt.

Zusätzlich wurde das Paket "libbluetooth-dev" mit dem Paketverwaltungsmanager "Synaptic" unter Ubuntu Linux installiert, um die Bluetooth-Bibliothek "BlueZ" benutzen zu können.

Schritt für Schritt:

1. Das Betriebssystem "Ubuntu" kann man sich kostenlos unter dem folgenden Link herunterladen: <http://www.ubuntu.com/>
2. Die Entwicklungsumgebung "Eclipse C++" steht auf der Homepage von Eclipse auch kostenlos zur Verfügung: <http://www.eclipse.org>.
 - ⇒ gz-Archiv mit der der Kommandozeile - tar xfvz eclipse.tar.gz – auspacken.
 - ⇒ tz-Archiv mit - tar xfv eclipse.tar - auspacken
 - ⇒ Programm mit der Datei "Eclipse.exe" starten.

3. Um die Bluetooth-Bibliothek nutzen zu können, muss vorher das Paket "libbluetooth-dev" installiert werden. Dies ist bequem mit dem Standardpaketverwaltung "Synaptic" möglich:
 - ⇒ System
 - ⇒ Systemverwaltung
 - ⇒ Synaptic-Paketverwaltung starten
 - In der Schnellsuche "libbluetooth-dev" eingeben
 - Paket zum installieren vormerken
 - Im Menü auf dem Button –Anwenden- klicken, um das Paket zu installieren.
4. Jetzt muss nur noch eine Einstellung in Eclipse angepasst werden.
 - ⇒ Project
 - ⇒ Properties
 - C/C++ Build
 - Settings
 - GCC C++ Linker
 - Libraries
 - add: **bluetooth**

3.4 Struktur des Moduls

Die drahtlose Verbindung wird mit der Klasse

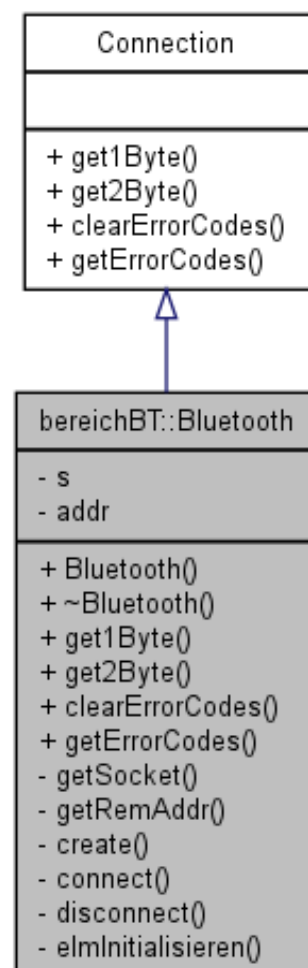
"Bluetooth" realisiert, die von der Klasse "Connection" die folgenden abstrakten Methoden erbt:

```
+ get1Byte()
+ get2Byte()
+ clearErrorCodes()
+ getErrorCodes()
```

get1Byte() und get2Byte()

Die zwei Methoden haben als Parameter einen konstanten String für die **PID's** und liefern die gewünschten Bytes als String.

Der String besteht aus 4 Hexadezimalzahlen, die mit der passenden Formel in der Klasse "ELM" durch die verschiedenen Methoden ausgewertet werden.



PID's sind Indikatoren für die vom Motorsteuergerät unterstützten Informationen.

Der Adapter ELM kann die verfügbaren PID's abfragen und ausgeben. Je nachdem welche PID man haben will, werden 1 oder 2 Bytes zurückgegeben. Welche PID's wie viel Bytes zurückliefern und die passenden Formeln für die Auswertung, kann man unter dem folgenden Link nachschlagen: http://en.wikipedia.org/wiki/OBD-II_PIDs

clearErrorCodes() besitzt kein Parameter und gibt kein Wert zurück. Diese Methode verschickt lediglich ein Steuerbefehl "04\r" um alle im Auto vorhandenen Fehlercodes zu löschen. Nachdem alle Fehlercodes erfolgreich gelöscht wurden, antwortet der ELM mit der Zahl "44". Dies wird in der Methode zur Sicherheit, dass der Befehl erfolgreich ausgeführt wurde, überprüft.

getErrorCodes() gibt als Rückgabewert ein String mit allen Fehlercodes zurück.

3.4.1 Verbindungsaufbau

Der Verbindungsaufbau erfolgt durch die privaten Methoden:

- `bdaddr_t getRemAddr()`
- `int getSocket ()`
- `bool create ()`
- `bool connect (bdaddr_t chosenAddr)`
- `bool disconnect ()`

Die Verbindung wird im Konstruktor der Bluetooth-Klasse in folgender Reihe hergestellt.

1. **createSocket()** initialisiert ein Socket "s" mit den Parametern `AF_BLUETOOTH`, `SOCK_STREAM`, `BTPROTO_RFCOMM`.
2. **bdaddr_t getRemAddr()** sucht in der Nähe nach vorhandenen Bluetooth-Geräten. Falls sich darunter auch der ELM-327 mit der MAC-Adresse "00:06:66:03:01:AD" befindet, liefert die Methode die Bit-Adresse des Adapters zurück. Wenn der ELM nicht vorhanden sein sollte, wird ein Exception geworfen.
3. **connect(bdaddr_t chosenAddr)** baut eine Verbindung mit der übergebenen Bitadresse auf.

3.4.2 Initialisierung

Nachdem die Verbindung steht, muss erst mal der ELM initialisiert werden. Die Initialisierung führt die Methode `elmInitialisieren()` aus. Die Methode verschickt nacheinander folgende Steuerbefehle aus:

- `\r`
- `atz\r`
- `ate0\r`
- `atl0\r`

Durch `"atz\r"` wird ein Reset des ELM durchgeführt. Durch den Befehl `"ate0\r"` wird das Echo in der Antwort ausgeschaltet und durch `"atl0\r"`, das Newline.

Durch die Initialisierung muss man nicht mehr extra das Echo oder Newline herausfiltern und hat sich dadurch viel Arbeit beim Auslesen gespart.

3.4.3 Verbindungsabbau

Der Verbindungsabbau erfolgt automatisch beim Löschen des Objektes bzw. Zeigers. Im Destruktor von der Klasse-Bluetooth wird die Methode `disconnect()` aufgerufen. Die wiederum den kreierten Socket wieder frei gibt.

3.5 Prinzip von `get1Byte()` und `get2Byte()`

Das Prinzip von `get1Byte()` und `get2Byte()` ist wahrscheinlich nicht gleich einleuchtend, deshalb wird es hier etwas ausführlicher behandelt.

Welche Methode benutzt wird, wird durch die Anzahl der Bytes, die man erwartet entschieden.

Nachdem die PID mit der Methode `write()`, die die Library Bluetooth zur Verfügung stellt, versendet hat. Wird die Antwort in der While-Schleife ausgewertet.

Bevor aber die `read()` Methode zum Einsatz kommt, wird mit der `select()` Methode getestet, wann der Socket zum Lesen bereit ist. Ist der Socket innerhalb von 3 Sekunden bereit, fängt das Programm an, die Daten in den Puffer `cWert` zu schreiben. Die Daten werden anschließend in die String Variabel `sWert` geschrieben um die komfortablen String-Methoden nutzen zu können. Wenn in dem String `"NO DATA"` gefunden wird, bedeutet dies, dass das Auto die jeweilige PID nicht unterstützt und die Methode liefert `"NO DATA"` zurück. Falls aber `"NO DATA"` nicht vorhanden ist und genau 12 Bytes gelesen wurden, werden die richtigen Bytes ausgewertet.

Codeausschnitt aus der Methode get1Byte():

```
int timer = 2;
while(true)
{
    tv.tv_sec = 3;
    retval = select(s+1, &rfd, NULL, NULL, &tv);
    if(retval) {
        bytes_read = read(s, cWert, sizeof(cWert));
        bytes_all = bytes_all + bytes_read;
        sWert.append(cWert);
        pos2 = sWert.find("NO DATA\r\r>");
        if(pos2 != string::npos)
        {
            return "NO DATA";
        }
        if(bytes_all == 12)
        {
            pos = sWert.find_first_of('>');
            if(pos != string::npos)
            {
                reWert1[0] = sWert[6];
                reWert1[1] = sWert[7];
                reWert1[2] = '\0';
                break;
            }
        }
    }
    timer--;
    if(timer == 0) return "NO DATA";
}
```

Eine Antwort hat immer folgende Struktur: "41 05 35". Das Erste Byte ist immer die SID. Das zweite Byte gibt den Befehl an und das dritte Byte ist der eigentliche Wert, den man benötigt.

	Type	Value
Wert[1]	char	49 '1'
Wert[2]	char	32 ' '
Wert[3]	char	48 '0'
Wert[4]	char	53 '5'
Wert[5]	char	32 ' '
Wert[6]	char	51 '3'
Wert[7]	char	53 '5'
Wert[8]	char	32 ' '
Wert[9]	char	13 '\r'
Wert[10]	char	13 '\r'
Wert[11]	char	62 '>'

3.6 Testdurchführungen

3.6.1 Erfolgreiche Tests

Nachdem die "ELM-Klasse" mit dem Modul Bluetooth verknüpft wurde, konnten erste Tests durchgeführt werden.

In der "ELM-Klasse" wird ein Zeiger "m_Connection" erstellt, der auf das Objekt "Bluetooth" zeigt. Die Methoden in der "ELM-Klasse" nutzen die Methoden "get1Byte" und "get2Byte" um an die jeweilige Information ranzukommen. Die Informationen werden, wie unter Punkt 3. erwähnt, mit der passenden Formel ausgewertet und zurückgegeben.

Für den Test wurden die Objekte wie folgt erstellt:

```
Connection *verbindung = new  
bereichBT::Bluetooth();  
ELM * p_elm = new ELM(verbindung);
```

Danach wurde mit dem Zeiger "p_elm", neun verschiedene Methoden der "ELM-Klasse" in einer Schleife 10-mal aufgerufen und auf einer Konsole erfolgreich ausgegeben.

Dieser Test benötigte insgesamt 16 Sekunden.

D.h. innerhalb einer Sekunde können 5,6 Werte abgefragt werden.

ELM
- m_Connection
+ ELM() + ~ELM() + getEngineLoadValue() + getEngineCoolantTemperature() + getIntakeManifoldPressure() + getEngineRPM() + getVehicleSpeed() + getIntakeAirTemperature() + getMAFAirFlowRate() + getThrottlePosition() + getFuelRailPressure() + getTroubleCodes() + deleteTroubleCodes() + getWheelRPM() + getSteeringWheelDirection() + getFuelType() + getFuelFillLevel() + getAcceleratorPosition() + getSteeringWheelProtection() + getEngineTorque() + getGear() + getWheelRotation() + getLatitude() + getLongitude() + getFullGPSData() - stringToInteger() - integerToString() - asciiToInteger() - codeToString()

3.6.2 Gescheiterte Tests

Durch die Fertigstellung der Anwendungsbibliothek konnten weitere Tests durchgeführt werden. Statt der Klasse "ELM" wurde der Klasse "CAR" ein Zeiger auf "Bluetooth" übergeben. Dieser Test scheiterte aus unerforschten Gründen. Wahrscheinlich ist es aber ein Timing-Problem, da im Debug-Modus die richtigen Werte auf der Konsole angezeigt werden.

3.7 Überlegungen zur Realisierung weiterer Schritte

Das Programm wurde bis jetzt so konzipiert, dass nur eine Verbindung mit dem ELM 327 (MAC-Adresse: 00:06:66:03:01:AD) hergestellt wird. Man könnte das Programm so erweitern, dass auch Verbindungen zu anderen ELM-327-Bluetooth Geräten zur Verfügung gestellt werden.

4. Auswertung der Daten mit der ELM-Klasse

Die Daten, die an den ELM-Chip gesendet und empfangen werden sind ASCII-Zeichen in Hex-Form. Somit muss um z.B. die Motortemperatur herauszufinden die Zeichenfolge „0105“ gesendet werden. Der ELM antwortet dann wiederum mit einer Kryptischen Zeichenfolge, die bis auf die relevanten Antwortbytes schon in der Seriellen Klasse „bereinigt“ wird. Die Umrechnung dieser Bytes in reale Temperatur übernimmt dann wiederum die ELM-Klasse.

4.1 OBD2 PID's

Für alle Fahrzeugdaten, die in OBD2 genormt sind gibt es zugehörige PID's (Parameter ID's). Diese ID's sind an das Steuergerät bzw. den ELM zu senden um die zugehörigen Daten zu bekommen.

Alle OBD2-PID's können auf folgender Seite eingesehen werden mit Erklärung, Anzahl Antwortbytes und zugehöriger Umrechnungsformel:

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

Das heißt nicht, dass ein Fahrzeug all diese Daten unterstützen muss, im Regelfall werden je nach Fahrzeug nur einige wenige unterstützt. Auch um herauszufinden welche Daten das Fahrzeug unterstützt gibt es eigene PID's.

4.2 Schnittstellenspezifikation

Die folgenden Funktionen bietet die ELM-Klasse anderen Klassen als Schnittstelle an:

```
const double getEngineLoadValue(void);           // in %
const double getEngineCoolantTemperature(void);  // in °C
const double getIntakeManifoldPressure(void);    // in kPa
const double getEngineRPM(void);                 // in U/min
const double getVehicleSpeed(void);              // in Km/h
const double getIntakeAirTemperature(void);      // in °C
const double getMAFAirFlowRate(void);            // in g/s
const double getThrottlePosition(void);          // in %
const double getFuelRailPressure(void);          // in kPa
const vector<string> getTroubleCodes(void);
const void deleteTroubleCodes(void);
```

Alle Funktionen (bis auf getTroubleCodes() und deleteTroubleCodes()) geben einen double, in dem der entsprechende Wert gespeichert ist, zurück. Somit müssen sich die oberen Schichten nicht mehr mit den einzelnen Bits die vom Motorsteuergerät zurückgesendet werden herumschlagen sondern rufen einfach bequem eine der Methoden auf und erhalten die gewünschten Daten als Double Werte zurück.

Spezialfall ist einmal die Methode getTroubleCodes(), die einen vector von strings zurückgibt. Je ein String enthält einen Fehlercode. Diese müssen dann in einer Liste von

Fehlercodes nachgeschlagen werden, um die genaue Beschreibung des Fehlers zu erhalten.

Fehlercode Beschreibungen können auf folgender Seite nachgeschlagen werden:

<http://www.engine-codes.com/>

Die Methode `deleteTroubleCodes()` gibt nichts zurück, beim Aufruf werden einfach alle Fehlercodes die im Motorsteuergerät abgespeichert sind gelöscht.

4.3 Allgemeines zur Klasse

Bisher wurden nur die Methoden zur Abfrage von Motordaten implementiert, die unser Testfahrzeug der Opel Astra unterstützt. Es wurde noch keine Abfrage implementiert, die herausfindet welche Daten das Fahrzeug unterstützt, d.h. wenn die Software mit einem anderen Fahrzeug getestet wird, das einige der Daten zur Abfrage nicht bereitstellt so dauert die Abfrage einige Zeit länger und das Motorsteuergerät antwortet mit einem „NO DATA“ dieses wird von der Klasse auch explizit abgefangen und eine exception geworfen.

In Zukünftigen Versionen wäre es von daher günstiger komplett auf exceptions zu verzichten und bevor Anfragen an das Motorsteuergerät gesendet werden können, herauszufinden welche Daten das Auto unterstützt, um so eine größere Kompatibilität zu anderen Fahrzeugen zu erreichen.

Desweiteren wurde die Software bis jetzt nur auf das Protokoll ISO 15765-4 (CAN 11/500) welches der Opel benutzt ausgelegt. Bei Fahrzeugen ab Baujahr 2008 wird nur noch dieses Protokoll eingesetzt, jedoch arbeiten andere Fahrzeuge noch mit anderen Protokollen wo die Antwortbytes in anderer Form zurück kommen können. Wenn dies der Fall ist arbeitet diese Klasse eventuell nicht ordnungsgemäß. Ein Lösungsansatz für dieses Problem wäre für jedes Protokoll eine eigene Klasse zu erstellen, die die empfangenen Daten dann spezifisch auswertet. Dazu müsste das verwendete Protokoll beim Verbindungsaufbau herausgefunden werden und dann je nachdem ein entsprechendes Objekt der jeweiligen Protokollklasse instanziiert werden.

4.4 Aufbau und Funktionsweise der get-Funktionen

Beispiel anhand der Methode getEngineRPM :

```
/**
 * PID for EngineRPM
 */
#define EngineRPM "010C\r"

.
.
.

const double ELM::getEngineRPM(void) {
    double Result;
    char wert[6] = EngineRPM;
    string returnValue = m_Connection->get2Byte(wert);
    if(returnValue.substr(0,7) == "NO DATA") {
        throw "EngineRPM: Die Angeforderte Information wird von diesem Fahr-
zeug nicht unterstützt";
    }

    int A = asciiToInteger(returnValue.substr(0, 2));
    int B = asciiToInteger(returnValue.substr(2, 2));
    Result = ((A * 256.) + B) / 4; // Umrechnungsformel
    return Result;
}
```

Das Char-Array „wert“ wird mit der Zeichenfolge „010C“ und einem abschließenden CR, um dem Befehl beim Senden über die Serielle Schnittstelle mit einer Eingabe zu bestätigen (sonst wird der Befehl nicht ausgeführt), gefüllt.

„010C“ ist die PID für die Drehzahl. Die Methode get2Byte() wird von der Seriellen Klasse zur Verfügung gestellt und wird in diesem Fall aufgerufen, da die Daten für die Motordrehzahl 2Bytes als Antwort enthalten. Andere PID's geben nur 1Byte zurück je nachdem muss die entsprechende Funktion aufgerufen werden.

Wenn der String „NO DATA“ als Return-Wert zurück kommt, dann wird dieser Wert von dem Fahrzeug bzw. dem Motorsteuergerät nicht unterstützt und es wird eine Exception ausgelöst.

Als nächstes bekommen die beiden Integerwerte A und B die Bytes 1 und 2 des Return-Wertes als Dezimalzahl zugewiesen. Diese Umrechnung der ASCII-Zeichen in Hex-Form wird durch die Methode asciiToInteger() durchgeführt.

Nun kann mit der entsprechenden Umrechnungsformel für diese PID, der Eigentliche Wert ausgerechnet werden und dieser wird in Form eines Double-Wertes als Return-Wert zurückgegeben.

4.5 Zusätzliche Informationen

4.5.1 ELM-Klasse in Verbindung mit der CAN- und GPS-Klasse

Die ELM-Klasse stellt nicht das eigentliche Interface nach außen dar. Sie wird vielmehr durch Funktionen der CAN-Klasse, die von der CAN-Gruppe erstellt wurde und der GPS-Klasse ergänzt. Da wir eng mit der CAN-Gruppe in Zusammenarbeit standen einigten wir uns auf ein gemeinsames Interface. Dieses Interface stellt die CAR-Klasse zur Verfügung. Darin sind alle Methoden der ELM sowie der CAN Klasse enthalten und zusätzlich noch die Methoden zur GPS-Positionsbestimmung, allerdings alle rein virtuell. Daher müssen die Methoden, die die CAN-Klasse und die GPS-Klasse zur Verfügung stellt auch in der ELM-Klasse implementiert werden, diese enthalten nichts weiter als exceptions. Somit wird sichergestellt, dass diese Klasse diese Daten nicht unterstützt.

4.5.2 Private-Methoden der ELM-Klasse

Die ELM-Klasse besitzt desweiteren 4 Private-Methoden. Diese werden hier anhand des Quellcodes kurz erläutert:

```
/**
 * Converts a string to an integer
 * @param value Convert this string into an integer
 * @return      The converted integer
 */
const int stringToInteger(const string value);

/**
 * Converts an integer into a string
 * @param value A given value
 * @return      The converted integer
 */
const string integerToString(const int value);

/**
 * Converts a ascii-value to an integer
 * @param value Convert this hex-value into an integer
 * @return      The converted integer
 */
const int asciiToInteger(const string value);

/**
 * Converts a error-code into a readable string
 * @param value The 2 Bytes (= 4 characters) from the elm-response
 * @return      The readable string
 */
const string codeToString(const string value);
```

5. Anwendungsbibliothek

5.1 Begriffsdefinition

Begriff	Definition
ELM-Controller	ELM-Controller bezeichnet in dieser Dokumentation wahlweise den ElmCan-II USB oder den ElmCan-II Bluetooth.
PID	PID (= Parameter ID) sind Codes, mit denen man Informationen aus dem Fahrzeug abrufen kann.
Quellcodeordner	Damit ist der Ordner gemeint, in dem sich die Quellcodes des Projekts befinden.

5.2 Hinweise für die Weiterentwicklung

5.2.1 Allgemeine Hinweise

- Für die Installation von Bibliotheken werden root-Rechte benötigt.
- Bei Änderungen von Dateien in Perforce sollte man sich mit ALLEN betroffenen Teammitgliedern absprechen, damit nicht mehrere Personen die gleiche Datei bearbeiten.
- Die Aufgaben der einzelnen Teammitglieder sollten möglichst klar definiert sein.
- Eclipse kompiliert nur den Inhalt von gespeicherten Dateien. Vor dem kompilieren werden Dateien allerdings nicht automatisch gespeichert.
- Bei einer Compiler-Fehlermeldung, die etwas über „verirrte Zeichen“ sagt, muss man die Datei entweder mit einem Programm wie Textpad (www.textpad.com) korrigieren, oder man versucht die betreffende Zeile zu löschen und anschließend neu zu schreiben (NICHT Copy&Paste nutzen!).

5.2.2 Spezielle Hinweise

- Die Rückgabewerte des ELM-Controllers haben immer (?) 2 Byte (= 4 Hexadezimalzeichen) mehr. Das 1. Byte enthält die Device-ID. Das 2. Byte ist immer gleich. Danach fängt der eigentliche Rückgabewert an.
- Die Fehlerbehandlung erfolgt in allen Klassen über Exceptions.
- Das Projekt lässt sich momentan nur unter Linux kompilieren, da Header-Dateien eingebunden sind, die nur unter Linux zur Verfügung stehen.

5.3 Projekt kompilieren

5.3.1 Benötigte Bibliotheken

Für das Kompilieren werden folgende Bibliotheken benötigt:

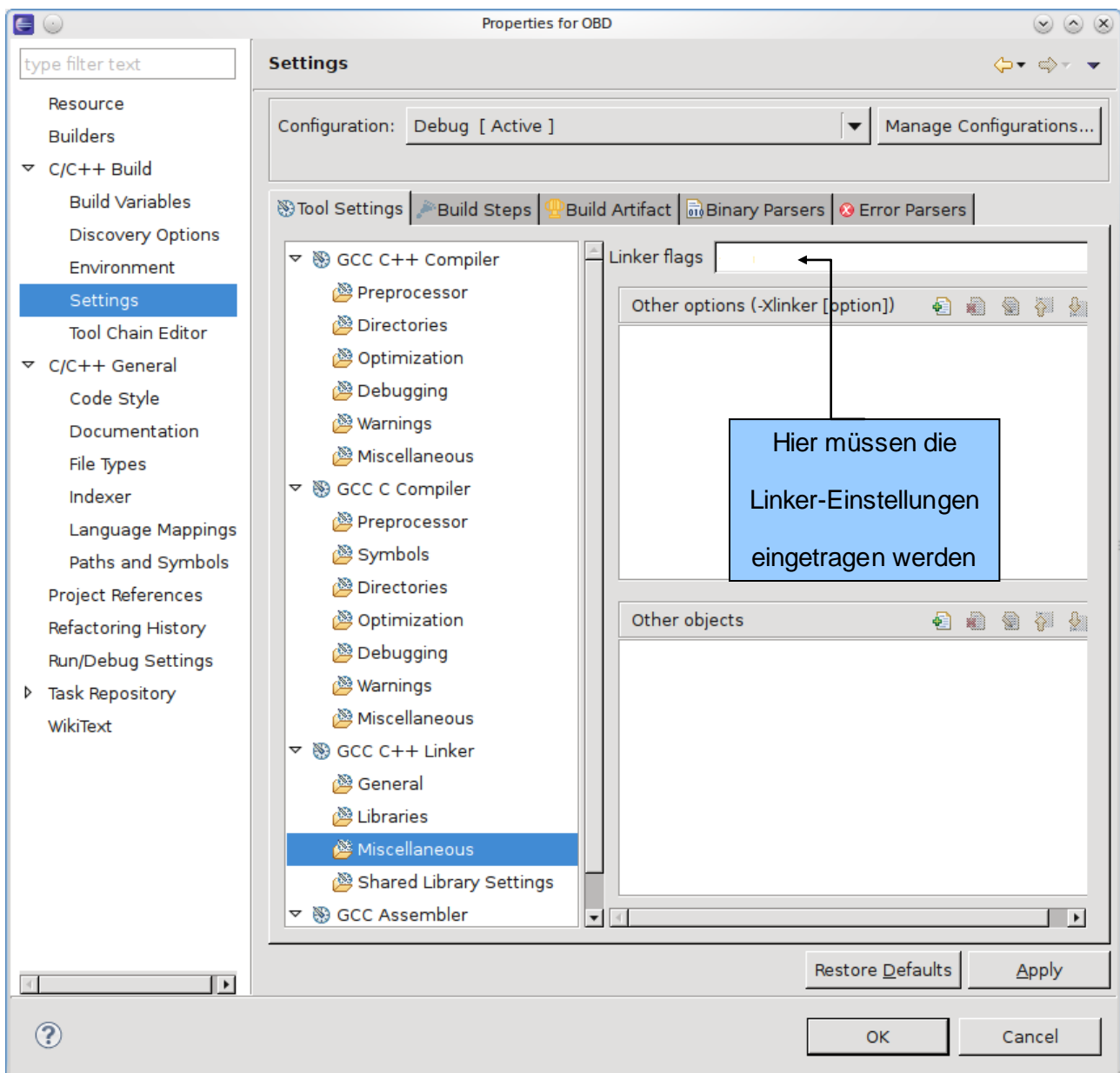
- BlueZ (libbluetooth) Für die Anbindung des ELM per Bluetooth
- libCANIO Für die Anbindung des CAN
→ Mehr Infos siehe Dokumentation der CAN-Gruppe.

5.3.2 Kompilieren in Eclipse

Um das Projekt in Eclipse kompilieren zu können, muss man ALLE .cpp- und .h-Dateien aus dem Quellcodeordner in Eclipse einfügen. Anschließend muss bei den Linker-Einstellungen folgende Zeile hinzugefügt werden:

```
-lpthread -lblueetooth
```

Der folgende Screenshot zeigt, wo man die Einstellungen vornehmen muss:



5.3.3 Kompilieren in der Kommandozeile

Um das Projekt in der Kommandozeile kompilieren zu können, genügt es, wenn man in den Quellcodeordner wechselt und dort den Befehl

```
make
```

ausführt. Das Projekt wird dann kompiliert und eine Beispielanwendung, welche nahezu alle Klassen benutzt wird unter dem Namen „test.out“ als ausführbare Datei erzeugt.

5.4 Quellcode-Dokumentation erstellen

Für die Erstellung der Quellcode-Dokumentation wird folgende Software benötigt:

Doxygen Für die Generierung der Dokumentation

Graphviz Für die Generierung der Diagramme innerhalb der Dokumentation

Sind beide Programme (z.B. über die Paketverwaltung) installiert, genügt es, in der Kommandozeile in den Quellcodeordner zu wechseln und dort den Befehl

```
doxygen Doxyfile
```

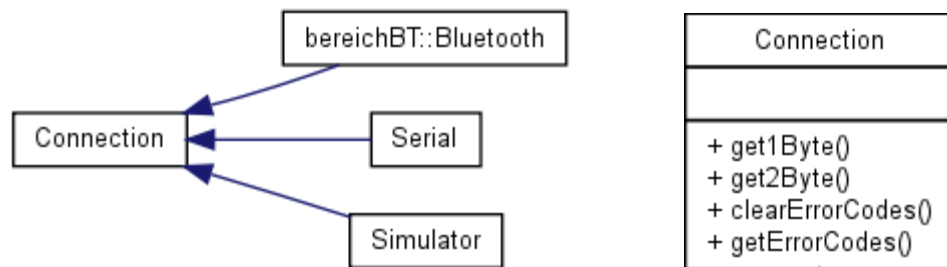
auszuführen. Nun wird die Dokumentation im Unterordner „doc“ erstellt.

Hinweis: Die in diesem Dokument benutzten Klassendiagramme stammen alle aus der erzeugten Quellcode-Dokumentation.

5.5 Klassendiagramme

5.5.1 Klasse Connection

Die Klasse „Connection“ dient als Interface für verschiedene Verbindungen zum ELM-Controller. Sie stellt lediglich virtuelle Methoden zur Verfügung, welche von den geerbten Klassen „Bluetooth“, „Serial“ und „Simulator“ verschieden implementiert werden.

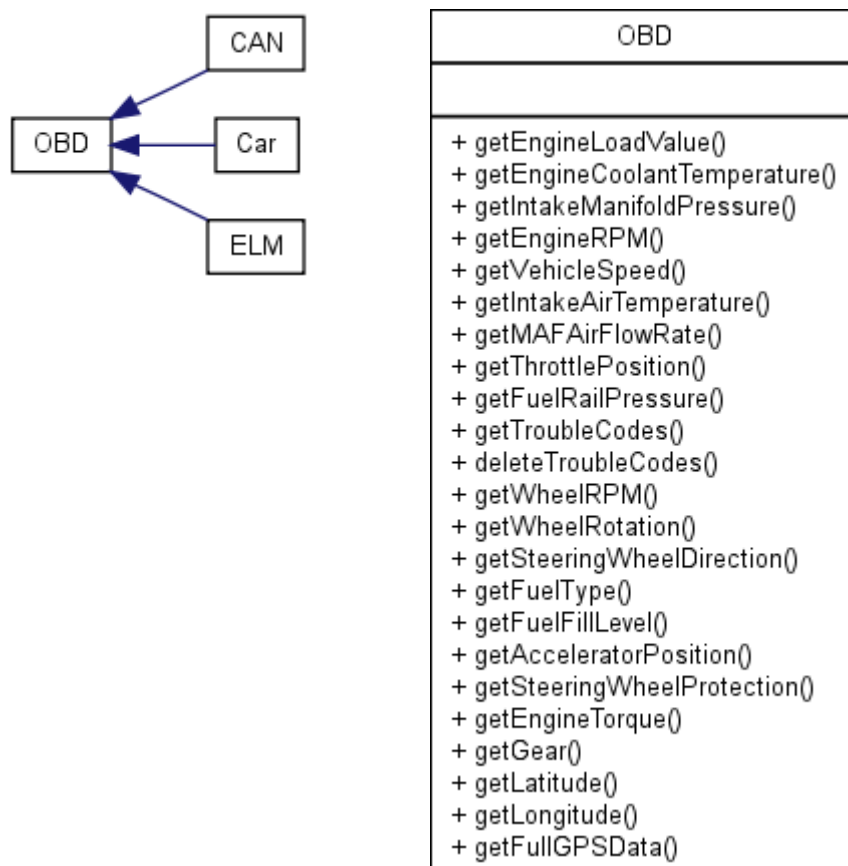


Möchte man nun eine weitere Verbindungsart zum ELM-Controller implementieren, genügt es eine neue Klasse zu erstellen, welche von der Klasse „Connection“ erbt. Nun muss man nur noch alle virtuellen Methoden implementieren.

5.5.2 Klasse OBD

Die Klasse „OBD“ wurde ursprünglich als Schnittmenge zwischen den verfügbaren Daten des ELM-Interfaces und des CAN-Interfaces konzipiert. Da so allerdings keine gemeinsame Daten-Schnittstelle zwischen den Gruppen realisiert werden konnte, wurde die Klasse dann erweitert um die Vereinigungsmenge zwischen den verfügbaren CAN-Daten und ELM-Daten zu bilden.

Später wurden dann noch die GPS-Daten in die Interface-Klasse „OBD“ aufgenommen. Die Klasse „OBD“ bildet damit eine gemeinsame Schnittstelle zwischen den einzelnen Gruppen.



Wie auch schon bei der Klasse „Connection“ stellt die Klasse „OBD“ nur virtuelle Methoden zu Verfügung. Somit müssen alle Klassen, welche von „OBD“ erben, alle Methoden implementieren. Von unserer Gruppe wurden hierbei die Implementierungen „ELM“ sowie „Car“ bereitgestellt.

5.5.3 Klasse Car

Die Klasse „Car“ vereint die implementierten Funktionalitäten der Klassen „CAN“ und „ELM“ in einer Klasse. Dies hat für andere Gruppen den Vorteil, dass sie sich nicht auf eine Schnittstelle festlegen müssen. Diese Klasse war zu Beginn des Projekts nicht vorgesehen, da man anfangs im Auto nur CAN oder ELM-Controller anschließen konnte. Erst durch einen selbstgebauten Adapter der CAN-Gruppe war es möglich CAN und ELM-Controller parallel im gleichen Auto zu betreiben.

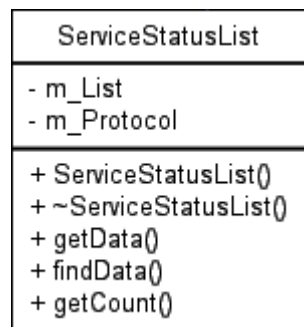
Zusätzlich stellt die Klasse den anderen Gruppen auch erstmals die GPS-Daten zur Verfügung. Die Funktionalität hierzu stammt aus dem Framework des ICM.

Da Zugriffe auf den ELM-Controller das zugreifende Programm eine gewisse Zeit blockiert, wurde zudem ein Puffer für alle unterstützten Daten hinzugefügt. Bei jedem Aufruf eines Getters wird der zwischengespeicherte Wert sofort zurückgegeben. Im Hintergrund fragt ein 2. Thread kontinuierlich den ELM-Controller und den CAN ab um die gepufferten Werte aktuell zu halten.

Car
<ul style="list-style-type: none"> - m_Connection - m_ELM - m_CAN - m_GPS - m_ChildThread - m_ThreadSafeSemaphore - m_ExternConnection - m_mode - m_EngineLoadValue - m_EngineCoolantTemperature - m_IntakeManifoldPressure - m_EngineRPM - m_VehicleSpeed - m_IntakeAirTemperature - m_MAFAirFlowRate - m_ThrottlePosition - m_FuelRailPressure - m_TroubleCodes - m_WheelRPM - m_SteeringWheelDirection - m_FuelType - m_FuelFillLevel - m_AcceleratorPosition - m_SteeringWheelProtection - m_EngineTorque - m_Gear - m_WheelRotation
<ul style="list-style-type: none"> + Car() + ~Car() + updateAll() + getEngineLoadValue() + getEngineCoolantTemperature() + getIntakeManifoldPressure() + getEngineRPM() + getVehicleSpeed() + getIntakeAirTemperature() + getMAFAirFlowRate() + getThrottlePosition() + getFuelRailPressure() + getTroubleCodes() + deleteTroubleCodes() + getWheelRPM() + getSteeringWheelDirection() + getFuelType() + getFuelFillLevel() + getAcceleratorPosition() + getSteeringWheelProtection() + getEngineTorque() + getGear() + getWheelRotation() + getLatitude() + getLongitude() + getFullGPSData() - updateEngineLoadValue() - updateEngineCoolantTemperature() - updateIntakeManifoldPressure() - updateEngineRPM() - updateVehicleSpeed() - updateIntakeAirTemperature() - updateMAFAirFlowRate() - updateThrottlePosition() - updateFuelRailPressure() - updateTroubleCodes() - updateWheelRPM() - updateSteeringWheelDirection() - updateFuelType() - updateFuelFillLevel() - updateAcceleratorPosition() - updateSteeringWheelProtection() - updateEngineTorque() - updateGear() - updateWheelRotation()

5.5.4 Klasse ServiceStatusList

Die Klasse „ServiceStatusList“ ist dazu gedacht um eine Informationsanzeige über die unterstützten PIDs zu liefern. Dazu wird beim Erstellen der Klasse eine Liste der bekannten PIDs abgerufen indem nacheinander die Getter (vom Typ OBD) abgefragt werden. Schlägt solch eine Abfrage fehl, nimmt die Klasse an, dass der Wert nicht unterstützt wird.



6. GPS-Modul

Auf die Funktionsweise der Positionierungsberechnung von GPS wird nicht näher eingegangen und kann bei Bedarf unter:

http://de.wikipedia.org/wiki/Global_Positioning_System

nachgelesen werden. Wissen über die Positionsberechnung wird für die Implementierung nicht benötigt, da Sie innerhalb der GPS-Geräte stattfindet.

Die Übertragung von GPS-Positionierungsdaten zwischen GPS-Geräte und PCs sowie mobilen Endgeräten erfolgt überwiegend über das NMEA 0183 Protokoll (siehe http://de.wikipedia.org/wiki/NMEA_0183). Diese gibt spezielle Schnittstellen für die Übertragung, festgelegte Baudraten und Format, Art, Anzahl der Übertragenen Daten vor.

Das uns vorliegende GPS-Gerät **Navilog NL-402U** wird über USB angeschlossen. Über `/dev/ttyACM0` kann durch ein `read()` der kontinuierliche Datenstrom ausgelesen werden. Diese Daten werden mittels max. 80 druckbaren ASCII-Zeichen gesendet und mit einem Zeilenende `<CR><LF>` voneinander getrennt. NMEA spezifiziert solche Datensätze, sogenannte NMEA sentences (siehe <http://www.nmea.de/nmea0183datensaetze.html>). Es gibt verschiedene Datensätze, wobei der sogenannte Recommended Minimum Sentence C (RMC) sich als wichtigster herausstellt, da jeder Empfänger diesen Satz ausgeben muss um NMEA-Konform zu sein. Jeder Satz fängt mit \$ oder ! Zeichen an, gefolgt von der Geräte ID und der Datensatz ID. Darauf folgt, jeweils durch Kommata getrennt die in den NMEA sentences definierten Datenfelder.



Abbildung 1: Navilog NL-402U

Der RMC Satz ist wie folgt definiert und muss entsprechend geparkt werden (Quelle: Wikipedia):

```
$GPRMC,162614,A,5230.5900,N,01322.3900,E,10.0,90.0,131006,1.2,E,A*13
$GPRMC,HHMMSS,A,BBBB.BBBB,b,LLLLL.LLLL,I,GG.G,RR.R,DDMMYY,M.M,m,F*PP
```

Symbol	Bedeutung
HHMMSS	Zeit (UTC)
A	Status (A für <i>OK</i> , V bei Warnungen)
BBBB.BBBB	Breitengrad
B	Ausrichtung (N für <i>North</i> , nördlich; S für <i>South</i> , südlich)
LLLLL.LLLL	Längengrad
L	Ausrichtung (E für <i>East</i> , östlich; W für <i>West</i> , westlich)
GG.G	Geschwindigkeit über Grund in Knoten
RR.R	Kurs über Grund in Grad bezogen auf geogr. Nord
DDMMYY	Datum (Tag Monat Jahr)
M.M	magnetische Abweichung (<u>Ortsmissweisung</u>)
M	Vorzeichen der Abweichung (<i>E</i> oder <i>W</i>)
	Signalintegrität :
	A = Autonomous mode,
	D = Differential Mode,
F	E = Estimated (dead-reckoning) mode
	M = Manual Input Mode
	S = Simulated Mode
	N = Data Not Valid
PP	hexadezimale Darstellung der Prüfsumme

Zur Zeit ist nur der RMC-Satz mit den Werten Zeit, Datum, Breitengrad und Längengrad implementiert. Das Auslesen der Positionsdaten über die zur Verfügung gestellten GPS-Geräte wurde unter der Verwendung der Klassen des InCarMultimedia-Frameworks vom 22.09.2010 realisiert. Alle für das GPS relevanten Programmdateien befinden sich im Repository unter „lobd_t_bpse_ws_10\Fahrzeugbusanbindung\GPS_Module“.

6.1 Beschreibung der wichtigsten Klassen und Methoden

CGpsSystemNMEA stellt die Implementierung des NMEA-Protokolls dar, greift auf das Gerät zu und Verpackt die Daten in eine GpsData-Struktur.

CGpsSystemNMEA::waitOnPort() wartet ohne zu pollen durch einen select()-Befehl auf die Lesebereitschaft des GPS-Devices

CGpsSystemNMEA::readGpsData(GpsData& gpsdata) liest vom Gerät in den Lesepuffer, separiert die Datensätze und übergibt sie dem Parser

CGpsSystemNMEA::parseGPS(const char* msg, GpsData& result) parst die Datensätze gemäß den einzelnen Datensatzdefinitionen.

CGpsDataContainerAdapter gibt durch get-Befehle Zugriff auf die GpsData Struktur des GpsDataContainers.

CGpsDataContainer realisiert threadsafe eine Art Doppelpufferung der GpsData Struktur um die Möglichkeit zu schaffen gleichzeitig Lese- und Schreibzugriffe auf die Datenstruktur ausführen zu können und die dadurch entstehenden Konsistenzprobleme zu verhindern.

6.2 Einbindung in die Anwendungsbibliothek

Die Funktionalitäten sind durch die Methoden:

```
const double getLatitude(void);
```

```
const double getLongitude(void);
```

```
const CGpsDataContainerAdapter getFullGPSData(void);
```

unter Verwendung der Membervariablen (Initialisierung im Konstruktor)

```
CGpsModul * m_GPS;
```

innerhalb der Car-Klasse der Anwendungsbibliothek verfügbar.

Werden die Methoden `getLongitude` und `getLatitude` nacheinander aufgerufen, so kann es passieren, dass die Werte zu verschiedenen Positionierungsmessungen gehören, müssen also als unabhängig betrachtet werden.

Der `CGpsDataContainerAdapter` enthält alle Werte des unterstützten GPS-Satzes, wobei nur Datum, Uhrzeit, Längen- und Breitengrad momentan implementiert sind. Sinnvoll wäre die Erweiterung in Hinsicht der Rückgabewerte der Klassenmethoden `CGpsSystemNMEA::parseGPS` und `CGpsSystemNMEA::readGpsData` des ICM-Frameworks. Zusätzlich zum Feststellen ob ein Datensatz fehlerfrei gelesen wurde, kann unter Einbeziehung des Valid-Flags des RMC-Satzes, die Gültigkeit dieses Datensatz festgestellt werden.

7. Kamerakomponente

Als Zusätzliches Feature zur Fahrzeuganbindung über OBD und GPS sollte es möglich sein über eine Kamera im Fahrzeuginneren Aufzeichnungen während der Fahrt zu machen. Dies ermöglicht eine Reihe von neuen Anwendungsszenarios. Denkbar wäre eine Blackbox für Automobile, bei dem die Fahrt der letzten 2 Minuten von der Videokamera auf ein persistentes Speichermedium gepuffert wird und nach einem Unfall wieder abgespielt werden kann. Aber auch die Synchronisierung mit den Aufgezeichneten OBD- bzw. CAN-Daten ergeben ein besseres Bild vom Fahrverhalten des Fahrers und lassen die zustande kommenden Messwerte besser verstehen, da man den Straßenverlauf rückwirkend visualisieren kann.

Zur Verfügung stand eine Logitech QuickCam Pro 9000 (Abb. 2). Erste Überlegungen waren, sie über Video4Linux anzusprechen. Zu Ein entsprechendes Codebeispiel (<http://v4l2spec.bytesex.org/>) wurde umgesetzt und auch erfolgreich getestet. In diesem Beispiel wird die Kamera geöffnet und es wird über einen sogenannten Memorymapped Speicherbereich, der den Kameraspeicher in dem sich die Videobilder der Kamera befinden auf den Arbeitsspeicher des Rechners abbildet. In der Funktion „process_image“ kann dann das Rohbild zu einem komprimierten abspielbaren Format weiterverarbeitet werden und anschließend abgespeichert oder über das Netzwerk gestreamt werden.



Abbildung 2: Logitech Quick Cam

Aufgrund der Tatsache, dass die V4L-API sehr mächtig ist, sich der Aufwand dadurch auch schnell vergrößert und die weitere Verarbeitung des Videobildes zu einem abspielbaren Format zusätzlich Einarbeitungszeit in Anspruch nehmen würde, wurde wegen des Zeitdrucks eine vorläufige und eher rudimentäre Lösung implementiert und bedarf bei Übernahme weiterer Fehlerabsicherung bzw. -behandlung.

In Perforce wurde unter „\obd_t_bpse_ws_10\Fahrzeugbusanbindung\Camera_Module“ in den Dateien CCamera.h und CCamera.cpp ein Coderahmen angelegt. Die Kamerafunktionen und deren Zustände werden in der Klasse CCamera gekapselt. Die Methode „startRecording“ ruft das Kommandozeilentool mencoder mit dem String:

```
„mencoder tv:// -tv driver=v4l2:width=640:height=480:device=“ + m_device + " -ovc  
lavc -lavcopts vcodec=mpeg4 -vf scale=640:480 -o " + path + " tv:// &“
```

über die system()-Funktion auf, welche Kommandozeilenbefehle ausführen kann. Die variablen m_device steht hier für den zuvor eingestellten Devicenamen, welcher in Linux unter /dev/ zu finden ist. Path für den in der Parameterübergabe angegeben Speicherort des Videos. Das &-Zeichen am Schluss des Befehls gibt an, dass das Programm als Hinter-

grundprozess gestartet werden soll und der system()-Aufruf nicht auf die Beendigung des aufzurufenden Programms warten soll, somit nicht blockierend wirkt.

Die „stopRecording“ Methode beendet ordnungsgemäß durch das Senden des SIGINT-Signals die Ausführung des Programms. Da sich zum Aufruf der Methode noch Videobilder im Puffer des mencoder-Tools befinden, die noch nicht kodiert und abgespeichert wurden, wartet die Methode durch einen sleep() aufruf einige Sekunden um auch diese Bilder abspeichern zu können. Der Sleepwert ist abhängig von der Performanz des Systems und variiert dementsprechend.

Das System geht dabei von der Annahme aus, dass es jeweils nur ein Prozess namens mencoder zur Laufzeit gibt und dieser auch nur aus unserer Anwendung gestartet wird. Fehler beim Programmablauf von mencoder werden nicht abgefangen und eine Überwachung des Prozessstatus ist nicht implementiert, was bei unvorgesehenen Programmablauf von mencoder zu Inkonsistenzen bei den Kamerazuständen (m_recording) führen kann.

In der Methode takeStillImage() kann der Shellbefehl:

```
./videodog -x 640 -y 480 -w 3 -d /dev/video0 -j -f standbild.jpg
```

für die Aufnahme eines Einzelbildes der Kamera implementiert werden, sofern das Tool videodog installiert ist.

Als Fortführung der Implementierung könnte die C++ Video Capture Library **avcap** (<http://sourceforge.net/projects/libavcap/>) dienen, welche Plattformunabhängig die Aufzeichnung von Aufzeichnungsgeräten unter Linux, Windows und MacOS realisiert. Unter der Verwendung einer solchen Bibliothek, die man in das eigene Programm einbindet ist das Auftreten von Fehlern wesentlich niedriger und die Fehlerbehandlung wesentlich einfacher und sicherer.

7.1 Verwendung anderer USB-Kameras unter Linux

Die USB-Kamera wird mit Hilfe der V4L2-API angesprochen, die ab Kernel 2.5 integriert ist. Nachfolgend wird erklärt wie man feststellen kann ob eine USB-Kamera standardmäßig unterstützt wird:

1. USB-Kamera einstecken (Logitech Webcam Pro 9000 Treiber ist bereits auf den Laborrechner in V4L2 bereits vorhanden -> keine weitere Einstellungen nötig)
2. Falls andere Kamera verwendet wird **'lsusb'** in Kommandozeile eingeben um Geräte ID und Bezeichnung herauszufinden. Wenn keine Bezeichnung vorhanden ist, linux-usb.org/usb.ids herunterladen und in **/usr/local/share** kopieren und nochmal versuchen. (**lsusb | grep -i' video'** listet alle usb video-geräte auf)
3. z.B. über **'mplayer tv :// -tv driver=v4l:width =640: height =480: device =/dev/video0'** oder **'\$ mplayer tv :// -tv driver=v4l2:width =640: height =480: device =/dev/video0'** Kamera ansprechen um Treibersupport zu testen.
4. Falls kein Treibersupport vorhanden, über ID u Bez. Treiber suchen und Installieren (z.B. gspca-Treibersammlung für verschiedene Webcams -> <http://mxhaard.free.fr>)

8. Links

8.1 OBD2 / ELM

<http://www.obd-2.de/index.html>

<http://elmelectronics.com/dsheets.html>

<http://www.blafusel.de/>

<http://www.blafusel.de/obd/agvsim.html>

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

<http://www.engine-codes.com/>

8.2 Serielle Verbindung

<http://www.easysw.com/~mike/serial/serial.html>

8.3 Bluetooth

<http://people.csail.mit.edu/albert/bluez-intro/>

8.4 GPS

http://de.wikipedia.org/wiki/Global_Positioning_System

http://de.wikipedia.org/wiki/NMEA_0183

<http://www.nmea.de/nmea0183datensaetze.html>

8.5 Kamera

<http://v4l2spec.bytesex.org/>

<http://sourceforge.net/projects/libavcap/>

<http://mxhaard.free.fr>

8.6 Wiki-Links

https://wiki.h-da.de/fbi/technische-sys-teme/index.php/Anbindung_ans_Fahrzeug,_Treiberschicht,_Protokoll,_Applikationsbibliotheken

https://wiki.h-da.de/fbi/technische-systeme/index.php/Benutzer:Stcrbeck/Protokoll_OBD

8.7 Perforce

//depot/obd_t_bpse_ws_10/Fahrzeugbusanbindung/Interface/

8.8 Sonstiges

<http://www.textpad.com>

<http://www.ubuntu.com/>

<http://www.eclipse.org>

9. Literatur

Plate Jürgen: Linux Hardware Hackz. Messen, Steuern und Sensorik mit Linux. / Hanser Fachbuch (2007)

Schäffer Florian: Fahrzeugdiagnose mit OBD: OBD I, OBD II sowie KW 1281. / Elektor-Verlag (2009)

Schröder Joachim; Glockel Tilo; Dillmann Rüdiger: Embedded Linux: Das Praxisbuch. / Springer Berlin (2009)