

# **Entwurfsdokument**

## **Service-Interface für ein Formula-Student-Fahrzeug**

**Technische Universität Ilmenau  
Softwareprojekt SS 2013  
Gruppe 19**

Christian Boxdörfer  
Thomas Golda  
Daniel Häger  
David Kudlek  
Tom Porzig  
Tino Tausch  
Tobias Zehner  
Sebastian Zehnter

19.06.2013

betreut durch

Dr. Heinz-Dietrich Wuttke, TU Ilmenau  
Oliver Dittrich, fachlicher Betreuer Team StarCraft e.V.

# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>4</b>
<b>2 Feinentwurf</b>	<b>5</b>
2.1 MicroAutoBox II . . . . .	5
2.1.1 Verwendete Blöcke - Ergänzungen . . . . .	5
2.1.2 Config-Dateien . . . . .	9
2.1.3 Aufbau des Simulink-Modells – Signalgenerator . . . . .	11
2.1.4 Signalkollektor . . . . .	19
2.1.5 Signaltransmitter . . . . .	27
2.2 Embedded PC und virtueller Server . . . . .	31
2.2.1 Initialisierung . . . . .	31
2.2.2 Datenübertragung und Datenverarbeitung . . . . .	33
2.2.3 Einfügen in die Datenbank . . . . .	37
2.3 vServer . . . . .	41
2.4 Webseite . . . . .	43
2.4.1 Aufbau der Webseite . . . . .	43
2.4.2 Installation . . . . .	43
2.4.3 Initialisierung der Seiten . . . . .	45
2.4.4 Header . . . . .	46
2.4.5 Startseite . . . . .	46
2.4.6 Registrierung . . . . .	48
2.4.7 Loginvorgang . . . . .	49
2.4.8 Logoutvorgang . . . . .	50
2.4.9 Änderung des persönlichen Passworts . . . . .	51
2.4.10 CSV-Exportfunktionalität . . . . .	52
2.4.11 Nutzerverwaltung . . . . .	53
2.4.12 Online-Anzeige der Nutzer . . . . .	54
2.4.13 Allgemeine Seitenaufbau . . . . .	55
2.4.14 Allgemeine Fahrzeugdaten . . . . .	55
2.4.15 Akkudaten . . . . .	57
2.4.16 Dynamische Daten . . . . .	59
2.4.17 Fahrdynamik . . . . .	61
2.4.18 Motor und Umrichter . . . . .	62
<b>3 Entwicklerdokumentation</b>	<b>63</b>
3.1 MicroAutoBox II . . . . .	63
3.2 Embedded PC . . . . .	63
3.3 vServer . . . . .	63
3.4 Webseite . . . . .	64
3.4.1 Allgemeine Funktionen (general.php) . . . . .	64
3.4.2 Nutzerfunktionen (users.php) . . . . .	66
3.4.3 Konfigurationsdatei (config.php) . . . . .	70
3.4.4 akkudaten.php . . . . .	72
3.4.5 allgemeinefahrzeugdaten.php . . . . .	72
3.4.6 changepassword.php . . . . .	72
3.4.7 dynamischeschedaten.php . . . . .	72
3.4.8 fahrdynamik.php . . . . .	73
3.4.9 index.php . . . . .	73
3.4.10 install.php . . . . .	73

3.4.11	login.php . . . . .	73
3.4.12	logout.php . . . . .	73
3.4.13	motorundumrichter.php . . . . .	73
3.4.14	register.php . . . . .	74
3.4.15	core/init.php . . . . .	74
3.4.16	core/functions/accept.php . . . . .	74
3.4.17	core/functions/akkudaten_sql.php . . . . .	74
3.4.18	core/functions/allgemeinefahrzeugdaten_sql.php . . . . .	74
3.4.19	core/functions/csvexp_sql.php . . . . .	75
3.4.20	core/functions/dynamischedaten_sql.php . . . . .	75
3.4.21	core/functions/fahrdynamik_sql.php . . . . .	75
3.4.22	core/functions/general.php . . . . .	75
3.4.23	core/functions/motorundumrichter_sql.php . . . . .	75
3.4.24	core/functions/users.php . . . . .	75
3.4.25	includes/css/style.css . . . . .	75
3.4.26	includes/overall/footer.php . . . . .	75
3.4.27	includes/overall/header.php . . . . .	76
3.4.28	includes/widgets/csvexp.php . . . . .	76
3.4.29	includes/widgets/loggedin.php . . . . .	76
3.4.30	includes/widgets/manageusers.php . . . . .	76
3.4.31	includes/widgets/onlineusers.php . . . . .	76
3.4.32	includes/widgets/userlist.php . . . . .	76
3.5	Skriptübergreifende Funktionen . . . . .	77
3.6	scripts/allgemeinefahrzeugdaten_script.js . . . . .	78
3.7	scripts/akkudaten_script.js . . . . .	78
3.8	scripts/dynamischedaten_script.js . . . . .	80
3.9	scripts/fahrdynamik_script.js . . . . .	82
3.10	scripts/motorundumrichter_script.js . . . . .	82
	<b>Abbildungsverzeichnis</b>	<b>84</b>
	<b>Anhang</b>	<b>85</b>

# 1 Einleitung

Dieses Entwurfsdokument für das Service-Interface des Formula-Student-Fahrzeugs bündelt Feinentwurf und Entwicklerdokumentation. Es ist daher in drei größere Kapitel untergliedert, die zuerst den Feinentwurf des Softwaresystems behandeln. Hierzu werden vor allem Modellierungen der einzelnen Komponenten des Softwarepakets mit UML aufgezeigt und erläutert. Im Anschluss wird auf die Änderungen zum Grobentwurf hinsichtlich der letzten Phase des Projekts aufgezeigt und gerechtfertigt. Schließlich wird auf ausgewählte Implementierungsabschnitte detailliert eingegangen und diese erläutert. Jedes dieser drei Kapitel ist dabei weiter unterteilt in die Abschnitte MicroAutoBoxII, Embedded PC und Webseite. Dabei umfasst der Teil Embedded PC auch das auf dem vServer laufende Programm zum Empfangen der Daten und Einfügen derselben in die Datenbank, da diese beiden Programme sich sehr ähneln und auch essentiell viele gleiche Klassen benutzen. Das Löschen veralteter Daten auf dem vServer, welches durch ein eigenständiges Programm realisiert wird, wird jeweils im Unterkapitel vServer abgehandelt. Die Unterteilung der Kapitel entspricht also auch den einzelnen Stationen, auf denen Teile unseres Softwarepakets laufen. Die genannte Reihenfolge orientiert sich dabei am Fluss der Daten, wie sie ihren Weg von der MicroAutoBoxII über den Embedded PC und das auf dem vServer laufende Programm zur Webseite zurücklegen.

## 2 Feinentwurf

### 2.1 MicroAutoBox II

#### 2.1.1 Verwendete Blöcke - Ergänzungen

Um den späteren Nutzern des Service-Interfaces den Einstieg in das erzeugte Modell in Simulink anhand konkreter Modellausschnitte zu erleichtern, wurden bereits die für den Grobentwurf benötigten Blöcke im Entwurfsdokument unter 3.2.1 vorgestellt. Jedoch ist es aufgrund mehrerer obligatorischer Anpassungen während der Implementierung des Feinentwurfs notwendig geworden, gegenüber dem Grobentwurf auf eine größere Vielfalt an Blöcken zurückzugreifen. Aus diesem Grund wird nun im Folgenden die jeweilige Funktionalität der erstmalig verwendeten Blöcke erläutert.

#### Math Operations



**Abbildung 2.1:** Produkt-Block zur Multiplikation zweier Signale

##### 1) *Produkt-Block*

Durch den Produkt-Block wird es ermöglicht, zwei Eingangssignale miteinander zu multiplizieren und das entstehende Ausgangssignal im Signalfluss weiterzuführen. In diesem Modell wird dieser Block dazu verwendet, mithilfe des Width-Blockes (siehe Signal Attributes, Width-Block, S. 6) die exakte Größe eines UDP-Paketes für einen bestimmten Datentyp (siehe dSPACE-Blöcke, UDP-Receive-Block, S. 8) zu berechnen. Darüber hinaus bietet dieser gegenüber dem Gain-Block bessere Laufzeiteigenschaften, wobei diesem vorteilhaften Aspekt oft eine größere Anzahl an zu verwendenden Blöcken und Leitungen gegenübersteht.

#### Logic and Bit Operations

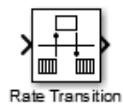


**Abbildung 2.2:** UND-Gatter

### 1) UND-Gatter

Dieser Block bildet als logischer Operator die exakte Funktionsweise eines UND-Gatters nach. Sollten an diesem Block die bis zur einer Anzahl  $n$  anschließbaren Eingangssignale mit dem Datentyp *boolean* allesamt als booleschen Werte eine logische 1 oder allesamt eine logische 0 aufweisen, so liegt am Ausgang des Blocks eine logische Eins an, andernfalls wird bei unterschiedlichen booleschen Werten am Ausgang eine 0 ausgegeben.

## Signal Attributes



**Abbildung 2.3:**  
Rate-Transition-Block



**Abbildung 2.4:**  
Width - Block

### 1) Rate-Transition-Block

Der Rate-Transition-Block bietet die Möglichkeit, Signale vom Output des Blockes, welche mit einer unterschiedlichen Frequenz als der Input des Blockes arbeiten, an den Input anzugeleichen. Dieser Block wird oftmals benötigt, wenn zwei verschiedene Signale unterschiedliche „Sample Times“ aufweisen die nicht manuell durch Änderung einer Sample Time aneinander angeglichen werden können. Innerhalb des Blockes stehen mehrere Optionen zur Verfügung, welche die Integrität der Daten und die Latenzzeiten der Signale beeinflussen. In unserem Falle wurden zur Sicherstellung der korrekten Angleichung bzw. der korrekten Weiterleitung der Signale die beiden Optionen „Ensure data integrity during data transfer“ und „Ensure deterministic data transfer (maximum delay)“ ausgewählt, was gegenüber den dadurch erzielten Vorteilen eine vertretbare Erhöhung der Latenzzeiten zur Folge hat.

### 2) Width-Block

Der Width-Block ermittelt wie sein Name schon vermuten lässt die Breite des anliegenden Input-Signals und leitet diese an den Output weiter. Hierbei können an den Input sowohl Vektoren als auch Busarrays angeschlossen werden.

Folgende Datentypen werden hierbei unterstützt:

- Floating point
- Built-in integer
- Fixed point
- Boolean
- Enumerated

(Quelle: <http://www.mathworks.de/de/help/simulink/blocklist.html>)

## Signal Routing

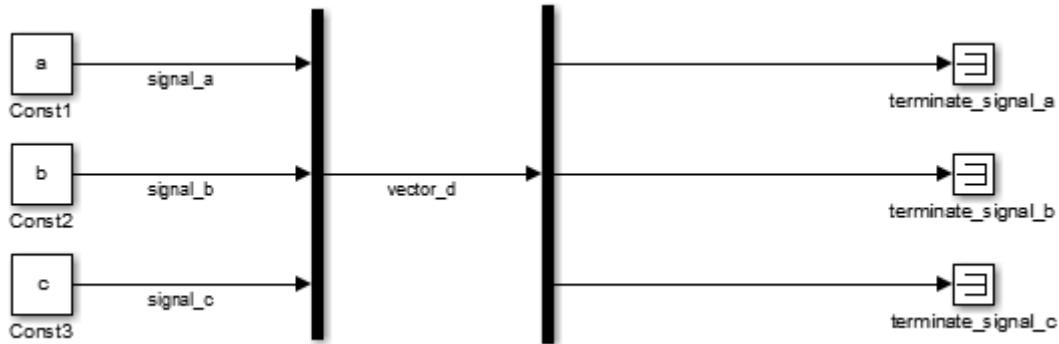


Abbildung 2.5: Modell zur Erläuterung eines Multiplexers / Demultiplexers

### 1) Multiplexer

Analog zum Bus Creator (s. Entwurfsdokument S.10) besitzt auch der Multiplexer die Fähigkeit, mehrere Eingangssignale zum einem einzelnen Signal zu bündeln. Jedoch besteht zwischen den beiden Komponenten ein gravierender Unterschied: während beim Busarray die einzelnen Eingangssignale weiterhin individuell vorliegen und demzufolge nur virtuell gebündelt werden um die Übersichtlichkeit innerhalb des Modells zu erhöhen, werden die Signale beim Multiplexer zu einem reellen Signalvektor zusammengefasst (s. Abb. 2.5). In diesem Beispiel werden die durch die drei Constant-Blöcke Const1-Const3 erzeugten Werte  $a$ ,  $b$  und  $c$  zu dem Signalvektor  $vector\_d$  zusammengefasst, welcher nun die folgende Gestalt besitzt:

$$vector\_d = ( \ signal\_a \ signal\_b \ signal\_c )^T$$

### 2) Demultiplexer

Mit dem Demultiplexer ist es anschließend möglich, aus dem Inputsignal  $vector\_d$  bestimmte Elemente zu entnehmen und auszugeben. In diesem Beispiel werden die drei Elemente des Vektors jeweils wieder einzeln als Skalare herausgeführt und anschließend auf Terminator-Blöcke geleitet. Je nach Anzahl der Inputs  $n$  und der Outputs  $p$  des Demultiplexers ergeben sich u.a. die folgenden Kombinationen:

Tabelle 2.1: Funktionsweise des Demultiplexers

Anzahl der Parameter	Ausgabe des Blockes
$n = p$	$p$ skalare Signale
$p > n$	Fehlermeldung
$p < n$ , $n \ mod \ p = 0$	$p$ Signalvektoren mit $\frac{n}{p}$ Elementen
$p < n$ , $n \ mod \ p = m$	$m$ Vektorsignale mit $\frac{n}{p} + 1$ Elementen und $(p - m)$ Signale mit $\frac{n}{p}$ Elementen

(Quelle: <http://www.mathworks.de/de/help/simulink/slref/demux.html>)

## Simulink Extras

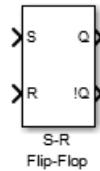


Abbildung 2.6: S-R-Flip-Flop

### 1) S-R-Flip-Flop

Dieser Block bildet die Funktionsweise eines S-R-Flip-Flops nach, welches die folgende bekannte Wertetabelle besitzt:

Tabelle 2.2: Wertetabelle S-R-Flip-Flop

S (Set)	R (Reset)	$Q_n$	$\bar{Q}_n$
0	0	$Q_{n-1}$	$\bar{Q}_{n-1}$
0	1	0	1
1	0	1	0
1	1	0	0

Hierbei ist zu beachten, dass an den beiden Eingängen S (Set) und R (Reset) vom Datentyp *boolean* niemals gleichzeitig zwei logische Einsen anliegen dürfen, die Eingangsbelegung [1, 1] stellt somit eine verbotene Eingangsbelegung dar.

## dSPACE-Blöcke

### 1) UDP-Receive-Block

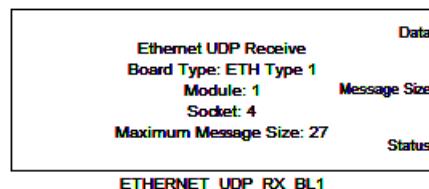
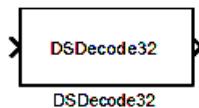


Abbildung 2.7: UDP-Receive-Block

Mittels des UDP-Receive Blockes wird es der MicroAutoBox II ermöglicht, vom Embedded-PC einen Datenstrom vom Datentyp *uint32* zu empfangen. Dies ist notwendig, um einerseits durch den Embedded-PC das zu versendende Datenpaket der MicroAutoBox II auszuwählen und andererseits den Grundstein einer bidirektionalen Kommunikation zwischen den beiden bereits genannten Komponenten für spätere Erweiterungen des Service-Interfaces durch externe Personen zu legen.

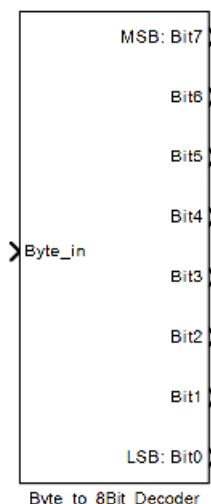
### 2) Decode32-Block



**Abbildung 2.8:** Block zur Dekodierung eines *uni32*-Datenstromes

Der Dekoder-Block DSDecode32 stellt das passende Gegenstück zum Enkoder-Block DSEncode32 dar und basiert ebenfalls wie dieser auf einer sFunction der Firma dSPACE. Er hat die Aufgabe, den empfangenen Datenstrom zu dekodieren und in einem Signalvektor in dem gewünschten Datentyp (Verweis) auszugeben.

### 3) Byte-to-8bit-Dekoder



**Abbildung 2.9:** Byte-to-8-bit-Dekoder

Bei dem Byte-to-8-bit-Dekoder handelt es sich um einen weiteren Block, der auf einer sFunction basiert. Durch ihn wird es möglich, aus einem Byte Daten die einzelnen Bits zu isolieren und diese getrennt an den Outputs weiterzuleiten.

## 2.1.2 Config-Dateien

### signalgenerator\_microautbox.m

Wie schon im Entwurfsdokument beschrieben, dient diese \*.m-File dazu, eine übersichtliche Darstellung über die Werteverläufe aller im Signalgenerator (2.1.3) erzeugten Testsignale zu gewährleisten und eine eventuelle Modifikation ebendieser durch eine ausführliche Kommentierung so komfortabel wie möglich zu gestalten. Des Weiteren wurde um die Benennung der Parameter in dieser Config-Datei zu vereinfachen jeder Funktionsvektor mit dem Präfix „SIGNAL\_“ und jeder Zeitvektor mit dem Präfix „TIME\_“ versehen. Die korrekte Referenzierung dieser Vektoren in Constant- (Verweis) oder RSI-Blöcken (Verweis) ist dem Entwurfsdokument aus Abbildung 3.21 zu entnehmen.

### **config\_datenpaket.m**

Dieses \*.m-File beinhaltet drei Vektoren, welche für den Embedded-PC erforderlichen Informationen über das zu versendende Datenpaket mit den Fahrzeugdaten beinhalteten. Diese drei Vektoren werden jeweils in einem Constant-Block (Verweis) referenziert und bilden mit anderen Paketinformationen das zu versendende „Informationspaket“ welches vor dem eigentlichen Start der Datenübertragung an den Embedded-PC übertragen wird. Der maßgebliche Grund für das Anlegen dieses Config-Files war es, dem Nutzer eine eventuelle Modifikation des Simulink-Modells zu vereinfachen und zusätzlich auf dem Embedded-PC ein hohes Maß an Softcoding zu realisieren, was den Anpassungsaufwand des dort ausgeführten C++ - Programms bei derartigen Veränderungen drastisch reduziert. Im Folgenden soll nun der Zweck der einzelnen Vektoren erläutert werden.

#### 1) VEKTOR\_AUFTHEILUNG\_DATENPaket

Dieser Vektor beschreibt, an welchen Stellen des Datenpaketes welche Datentypen anliegen, um anschließend auf dem Embedded-PC entsprechende Optimierungen bei der Umrechnung/Konvertierung der Daten und die Aufteilung des Datenpaketes anhand der einzelnen Gruppen A) - E) (s. Pflichtenheft) vornehmen zu können. Dieser  $1 \times N$  - Zeilenvektor mit aktuell  $N = 401$  Fahrzeugwerten wird aus Aspekten der Benutzerfreundlichkeit aus einer  $N \times 3$  - Matrix namens MATRIX\_AUFTHEILUNG\_DATENPaket erzeugt, wobei die einzelnen Spalten für die folgenden Einträge stehen:

[ Start — Stop — Datentyp ]

Hierbei ist jeder verwendete Datentyp analog zu den sFunctions *DSEncode32* und *DSDecode32* (Verweis) mit einer Ziffer kodiert:

**Tabelle 2.3:** Kodierung der Datentypen

Datentyp	<i>boolean</i>	<i>int8</i>	<i>uint8</i>	<i>int16</i>	<i>uint16</i>	<i>int32</i>	<i>uint32</i>	<i>float</i>	<i>double</i>
Ziffer	1	2	3	4	5	6	7	8	9

So gibt - um ein Beispiel anzuführen - „1 4 3“ in einer Zeile der Matrix an, dass die ersten vier Elemente des Datenpaketes den Datentyp *unit8* besitzen. Diese Matrix wird nun mittels einiger Befehle in den oben aufgeführten Vektor umgewandelt und somit an das C++ - Programm auf dem Embedded-PC korrekt angepasst. Dies geschieht beispielhaft auf folgende Weise: So wird laut der obigen Bildungsvorschrift bei einer Matrix der Form der Zeilenvektor V mit den Einträgen der jeweiligen Datentypen der Elemente 1-4 erzeugt:

$$M = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 4 & 5 \end{pmatrix}$$

$$V = \begin{pmatrix} 3 & 3 & 5 & 5 \end{pmatrix}$$

## 2) VEKTOR\_PAKETTEILUNG

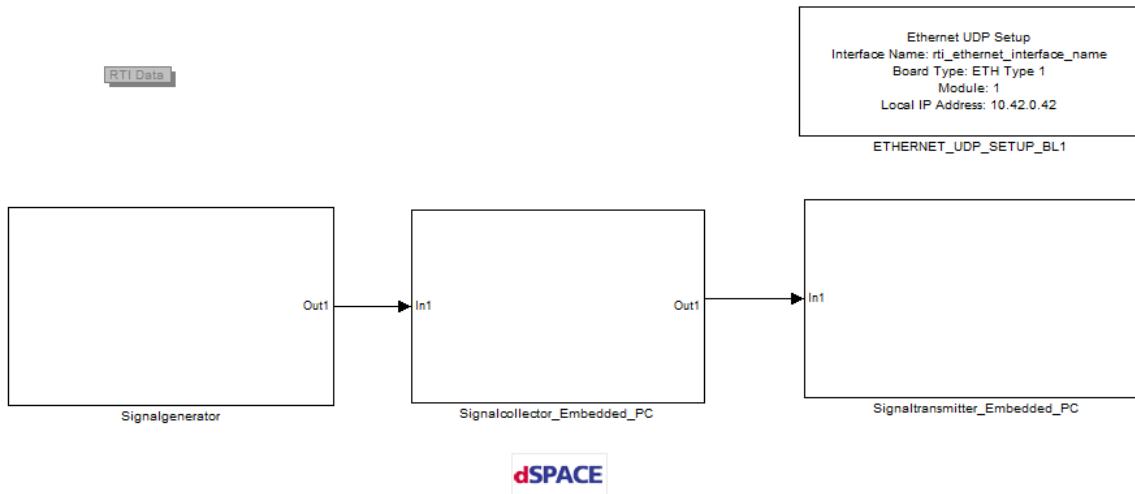
Dieser Vektor enthält als Elemente diejenigen Stellen, bei denen eine Aufteilung der einzelnen Datengruppen der Fahrzeugdaten in die Kategorien A) – E) erfolgt(s. Pflichtenheft). X | Y in der Kommentaren der \*.m-File gibt an, zwischen welchen Stellen das Datenpaket im C++ – Programm getrennt werden muss. Dies verfolgt den Zweck, das auf dem Embedded-PC empfangene Datenpaket in fünf Datenpakete mit den jeweiligen Datengruppen aufzuteilen, um einerseits eine klarere Zuordnung der Daten auf dem vServer zu ermöglichen und andererseits eventuell auftretenden Verbindungsproblemen und einbrechenden Signalstärken bei der Übertragung via UMTS an den vServer zu begegnen.

## 3) VEKTOR\_KOMMASETZUNG

Dieser Vektor  $1 \times N$  - Zeilenvektor mit  $N = 401$  Fahrzeugdaten liefert Informationen darüber, bei welchen Daten bzw. bei welchen Elementen das Komma um eine bestimmte Anzahl von Stellen nach links verschoben werden muss (0:= Keine Stelle, 1:= Um eine Stelle nach links verschieben etc.).

### 2.1.3 Aufbau des Simulink-Modells – Signalgenerator

#### Gesamtaufbau



**Abbildung 2.10:** Gesamtaufbau des Simulink-Modells auf höchster Modellebene

Wie in Abbildung 2.10 dargestellt, unterteilt sich der Aufbau des Simulink-Modells in drei übergeordnete Subsysteme:

- Signalgenerator
- Signalkollektor
- Signaltransmitter

wobei der Signalgenerator den Beginn und der Signaltransmitter das Ende des Signalflusses verkörpert. Des Weiteren ist es auch notwendig, sowohl die Maske RTI\_DATA\_HOST (Verweis Glossar) als auch den UDP-Setup-Block (s. Entwurfsdokument S. 12) auf der obersten Ebene des Simulink-Modells zu platzieren. Im Folgenden soll nun detailliert auf die einzelnen Komponenten und der jeweiligen Funktionalität dieser drei Subsysteme eingegangen werden.

## Signalgenerator

Der Signalgenerator hat die Aufgabe auf der MicroAutoBox II die für einen Systemtest benötigten Testsignale zu generieren, welche zuerst zum Signalcollector (s. 2.1.4) und anschließend über den Signaltransmitter (s. 2.1.5) auf den Embedded-PC via Ethernet-Schnittstelle gelangen und daraufhin über eine GPRS/UMTS-Verbindung an einen virtuellen Server gesendet werden. Diese Signale liegen im Bereich der echten Sensorwerte des Fahrzeuges und können somit gut zu Testzwecken verwendet werden. Da sich eine Modifizierung des in der MicroAutoBox II implementierten Simulink-Modells als sehr aufwändig gestaltet, ist die Generierung der Testsignale enorm wichtig, um die Funktionalität des Simulink-Modells und letztendlich auch des Service-Interfaces zu überprüfen. Vor der Implementierung des finalen Simulink-Modells wird dieser Teil des Modells von einem Mitglied von Team StarCraft e.V. durch eine Verknüpfung zu den realen Sensorwerten, welche vom CAN-Bus des Fahrzeuges geliefert werden, ersetzt.

### Aufbau

Wie im Pflichtenheft aufgeführt, sind die Signale der Fahrzeugdaten in die fünf verschiedenen Gruppen A) – E) namens *Allgemeine Fahrzeugdaten*, *Akkudaten*, *dynamische Daten*, *Fahrdynamikregelung* sowie *Motor- und Umrichterdaten* unterteilt. Diese Gliederung wurde im Signalgenerator ebenfalls beibehalten um ein hohes Maß an Übersichtlichkeit und Modularität zu gewährleisten. Der Signalgenerator ist demnach nochmals in einzelne Subsysteme unterteilt, welche die verschiedenen Kategorien der jeweiligen Fahrzeugdaten repräsentieren. Die generierten Signale werden über einen *Bus Creator* (s. Entwurfsdokument S.10) in ein Bussignal (Verweis Vektorsignal / Bussignal zu meinem Text) umgewandelt und gebündelt über diesen Bus an den Signalkollektor übermittelt, wobei die Reihenfolge der eingehenden Daten erhalten bleibt.

Innerhalb der einzelnen Subsysteme werden nun die simulierten Fahrzeugdaten generiert. Ein Subsystem ist hierbei eine Bündelung mehrerer Blöcke und Komponenten, welche auch wiederum Subsysteme sein können. Dies hat den Zweck, die Übersichtlichkeit innerhalb des Gesamtmodells zu wahren. Weiterhin kann ein solches Subsystem problemlos mehrfach verwendet werden, was den Aufwand beim Erstellen des Modells erheblich reduzieren kann. Am Beispiel *Allgemeine Fahrzeugdaten* soll nun ohne Beschränkung der Allgemeinheit der Aufbau eines Subsystems im Signalgenerator näher erläutert werden.

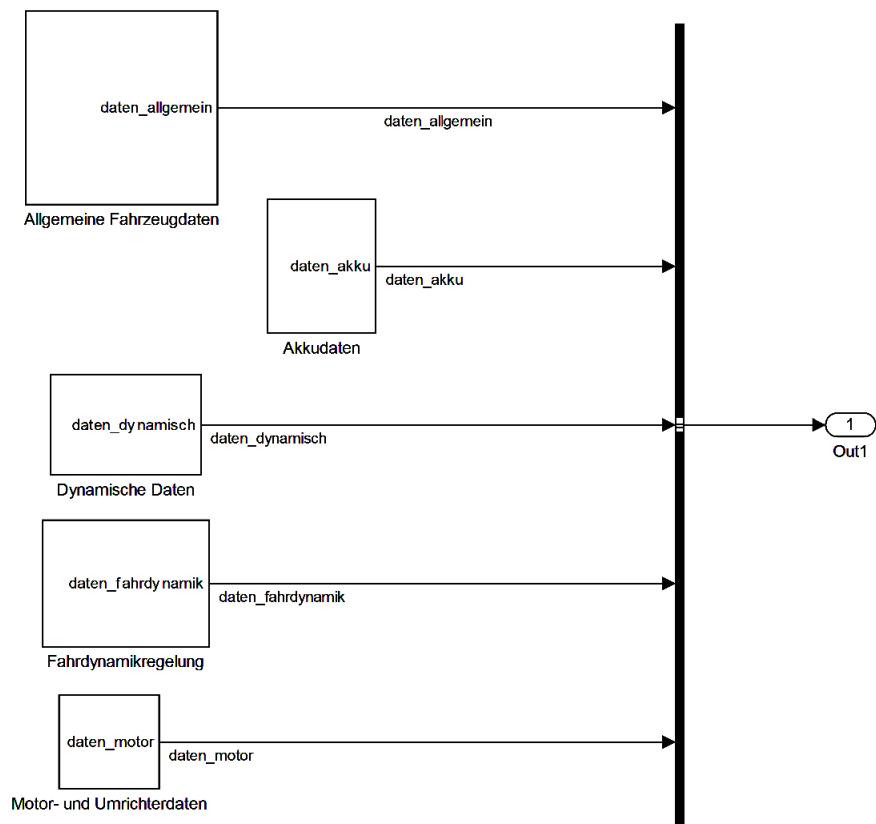


Abbildung 2.11: Unterteilung des Signalgenerators in die einzelnen Subsysteme

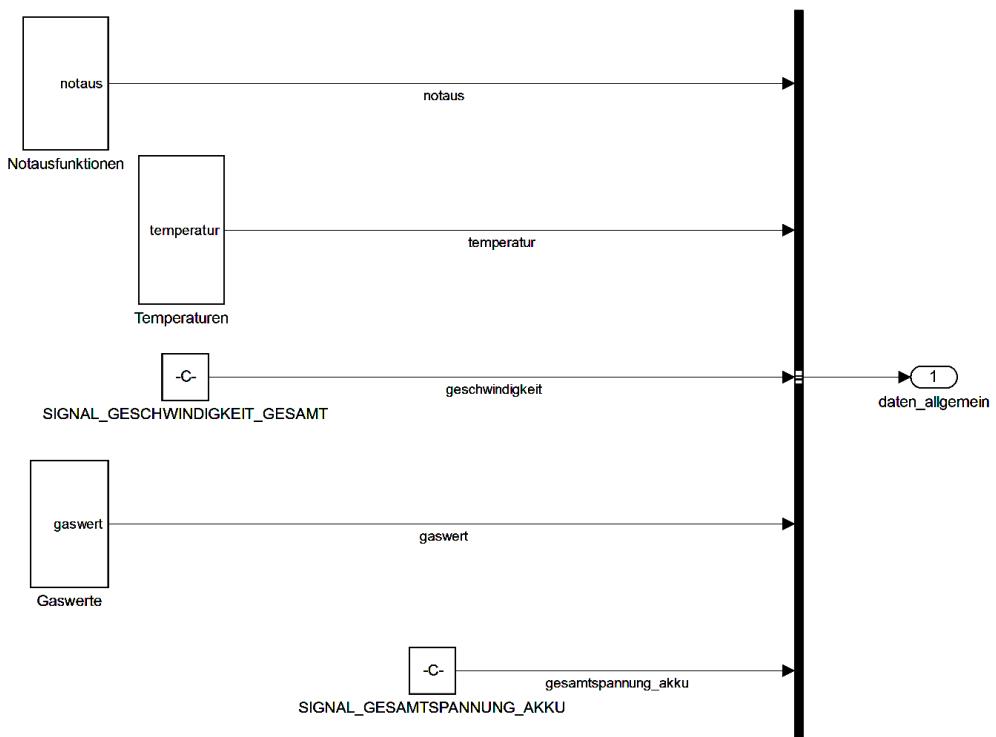


Abbildung 2.12: Aufbau des Subsystems Allgemeine Fahrzeugdaten

Wie in Abbildung 2.12 zu sehen ist, beinhaltet das Subsystem *Allgemeine Fahrzeugdaten* selbst einige Subsysteme sowie andere Simulinkblöcke. Weiterhin befinden sich im Subsystem *Allgemeine Fahrzeugdaten* weitere Subsysteme mit den Namen Notausfunktionen, Temperaturen und Gaswerte. Jedes Subsystem hat eine unterste Ebene in welcher keine Subsysteme mehr vorhanden sind. Weiterhin werden die einzelnen Signale innerhalb eines Subsystems mittels des bekannten Bus Creators zur einem Bussignal gebündelt. Das hat den Sinn, dass das gesamte Subsystem nur einen Output besitzt, was das ganze Modell wesentlich übersichtlicher gestaltet und hierdurch einfacher zu handhaben ist.

### *Erzeugung der einzelnen Signale*

#### A) *Allgemeine Fahrzeugdaten*

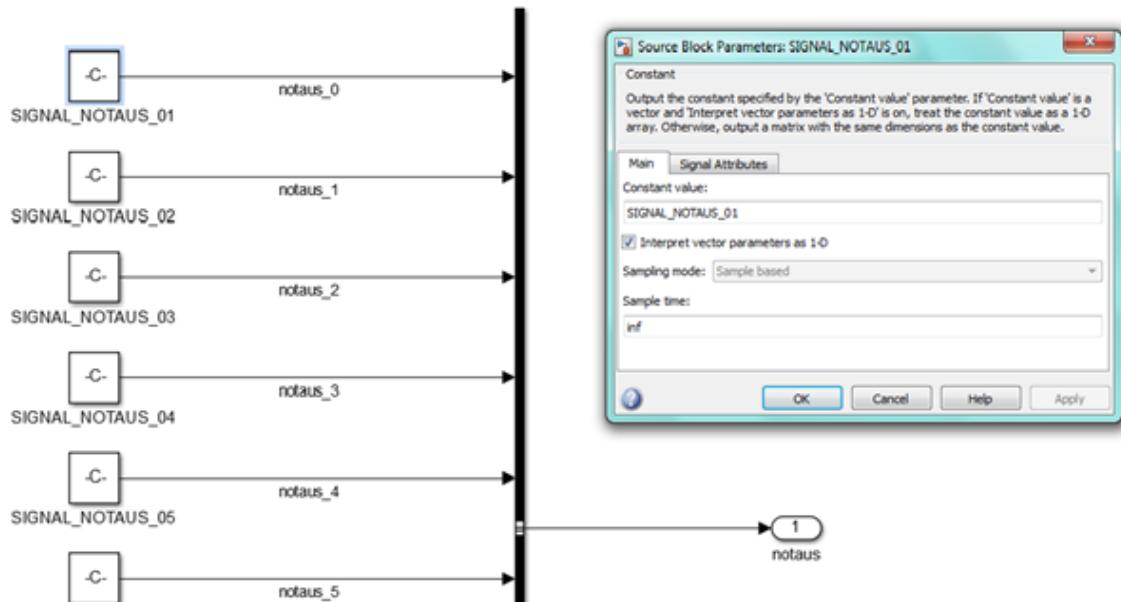
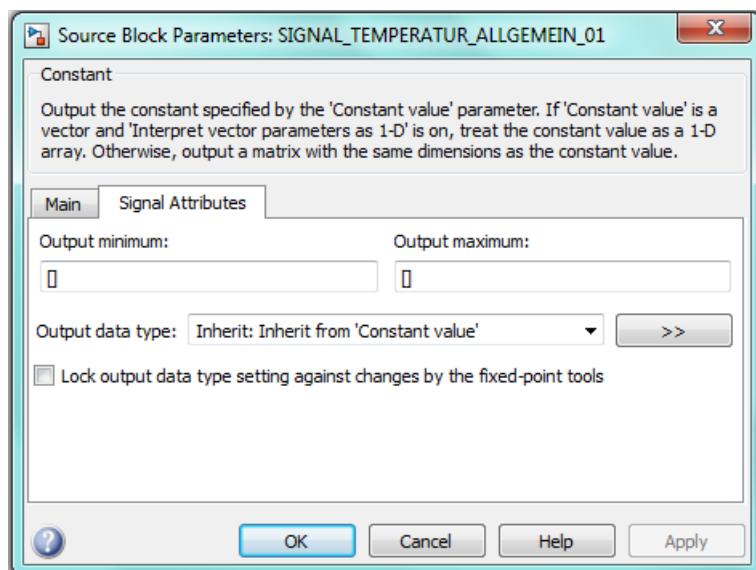


Abbildung 2.13: Erzeugung und Referenzierung der Notaus-Schalter im Simulink-Modell

Die Notaussignale, welche die Zustände der einzelnen Notausschalter des Fahrzeugs symbolisieren, werden über die Constant-Blöcke(s. Entwicklerdokumentation S.8) erzeugt. Sie besitzen den Datentyp *boolean*. Die Parameterwerte werden über das \*.m-File *signalgenerator.m*, welches zuerst diese in den Workspace von Matlab schreibt, von dort referenziert, um die Handhabung des Modells zu erleichtern (s. 2.1.2). Dabei ist SIGNAL\_NOTAUS\_01 der Variablenname, welcher dem System aufgrund des \*.m-Files bekannt ist und direkt aus dem Workspace ausgelesen wird. Dieses Signal wird über einen Bus Creator mit den anderen Signalen des Subsystems zu einem Signal gebündelt und an das Subsystem *Allgemeine Fahrzeugdaten* mit dem Variablenamen „notaus“ übergeben. Im Subsystem *Allgemeine Fahrzeugdaten* wird dieser Output wiederum mit den restlichen Signalen über einen Bus Creator zur einem Ausgabewert mit dem Namen „daten\_allgemein“ zusammengefasst. Im Subsystem *Signalgenerator* werden diese Daten letztendlich mit den restlichen generierten Fahrzeugdaten über einen weiteren Bus Creator gebündelt und an den Signalkollektor

übergeben.

Dies geschieht analog für die weiteren Notaussignalen. Die Temperaturen handhaben sich auf die gleiche Weise wie die Notaussignale, wobei der einzige Unterschied dabei in dem jeweilig verwendeten Datentyp besteht. Da die Datentypen im weiteren Verlauf noch mehrfach konvertiert werden(s. Signalkollektor, Signaltransmitter), spielt er bei der Erzeugung keine Rolle. Somit wird er vom Constant-Block durch die Einstellung „Inherit: Inherit from Constant value“ (s. 2.15) selbst festgelegt. Der Parameter „Gesamtgeschwindigkeit“ wird auch in einem Constant-Block generiert. Dieses befindet sich jedoch nicht nochmals in einem Subsystem, da es nur ein Wert dieser Art ist. Somit liegt er direkt an dem Bus Creator des *Allgemeine Fahrzeugdaten* Subsystems an. Schließlich verhalten sich die Parameter Gaswerte äquivalent zu den Temperatursignalen und der Parameter „Gesamtspannung Akku“ handhabt sich wie die Gesamtgeschwindigkeit.



**Abbildung 2.14:** Einstellungen zum verwendeten Datentyp im Constant-Block

### B) Akkudaten

Die Signalerzeugung im Subsystem *Akkudaten* verläuft ähnlich wie im Subsystem *Allgemeine Fahrzeugdaten*. Das Subsystem ist für die Generierung folgender Signale verantwortlich: Die Zellspannungen(Zelldaten) der Akkus, deren Temperaturen, die maximale sowie minimale Zellspannung, die Gesamtspannung der Akkus, der Ladestrom, die Ladespannung und die Zustände des Balancings (s. Abb. ??).

Die Zelldaten werden über einen „Repeating Sequence Interpolated“ – Block(s. ED S.9) erzeugt. Äquivalent zu den *Allgemeinen Fahrzeugdaten* werden auch hier die Parameterwerte, in diesem Fall SIGNAL\_ZP\_01\_ZELLE\_01, durch Variablen aus dem Config-File signalgenerator\_microautobox.m referenziert, wo diese hinterlegt sind (s. Abb. 2.16). Weiterhin gibt bei diesem Block die Variable im Feld „Vector of time values“ Auskunft darüber, zu welchen diskreten Zeiten sich der Ausgabewert dieses Blockes ändert. Auch dieser Wert(in diesem Fall „TIME\_08“) ist in der oben genannten \*.m-File definiert (s. 2.1.2). Eine Sample time 0.01 bedeutet, dass der Wert alle 0.01 Sekunden abgetastet wird.

Diese Sample time von 0.01 wird im ganzen Modell bei den „Repeating Sequence Interpolated- Blöcken verwendet. Nach der Erzeugung des Wertes „zelle\_001“ wird dieser mittels Bus Creator mit den anderen Werten seines Subsystems gebündelt und als Output „zellenpack\_01“ aus dem Subsystem Zellenpack\_01 ausgegeben.

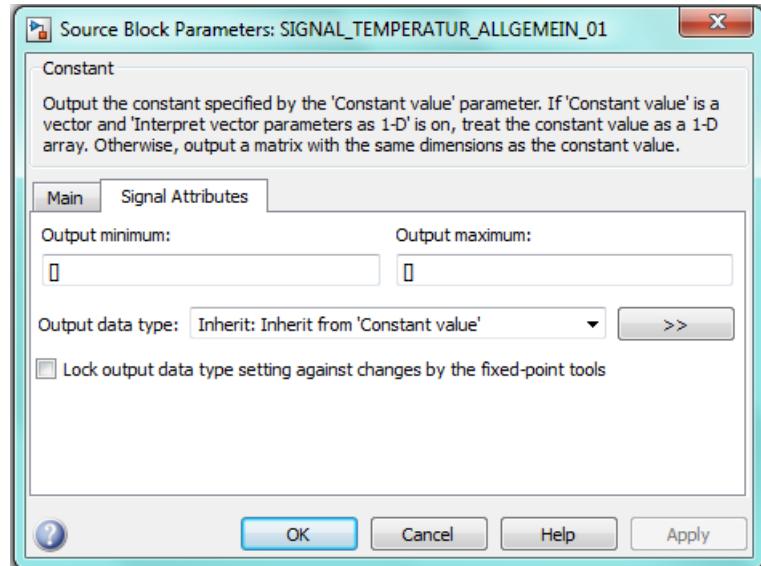


Abbildung 2.15: Einstellungen zum verwendeten Datentyp im Constant-Block

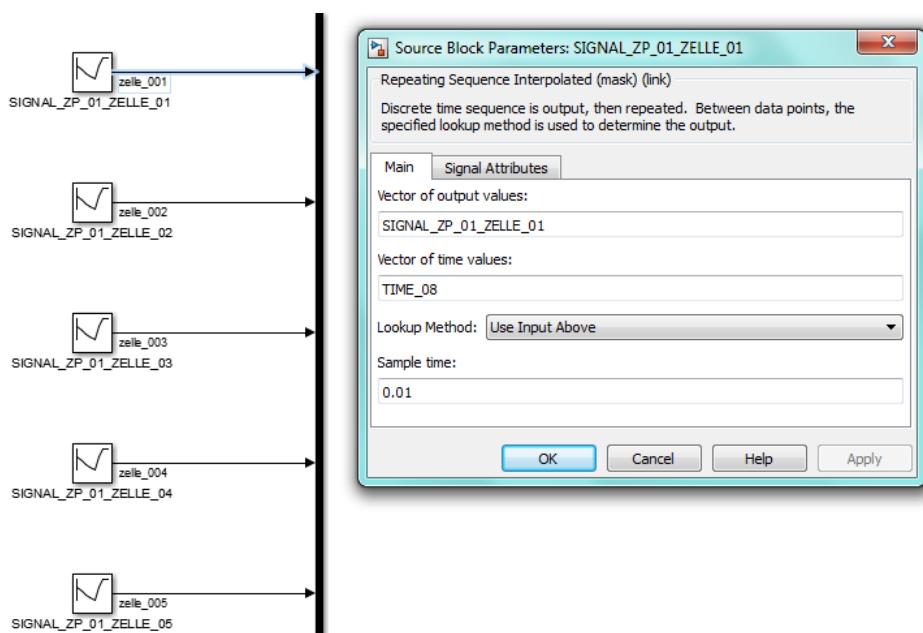


Abbildung 2.16: Erzeugung der Zelldaten über einen RSI-Block mittels eines Signal- und eines Zeitvektors

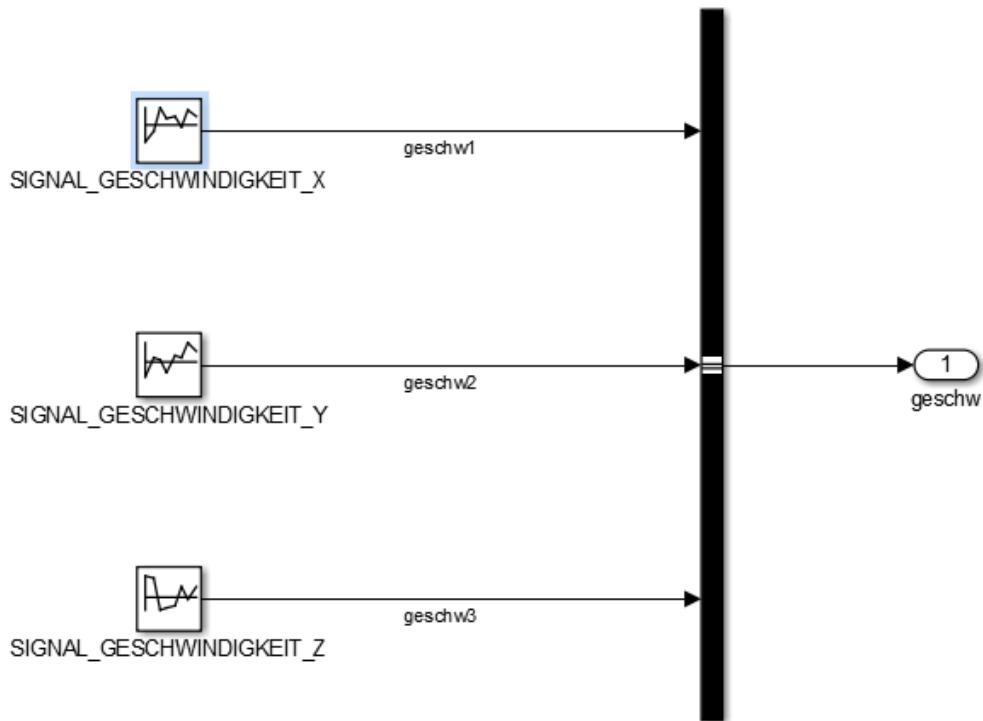
Das übergeordnete Subsystem „Zelldaten“ beinhaltet abermals weitere zwölf Subsysteme namens „Zellpack\_01“ bis „Zellpack\_12“ (s. Abb. ??). Es werden dort nach den Vorgaben des Pflichtenheftes 144 Zelldaten erzeugt, von denen sich jeweils zwölf Zellen in einem Zellenblock befinden. Aus diesem Grund bietet es sich zur Erhöhung der Übersichtlichkeit an, die Signalgenerierung innerhalb der zwölf Subsysteme vorzunehmen. Im Subsystem „Zelldaten“ wird das Signal „zellenpack\_01“ mit den Signalen der anderen zwölf Subsysteme gebündelt und mit den Variablenname „zelldaten“ das Subsystem „Akkudaten“ ausgegeben. An dieser Stelle werden die Signale des kompletten Subsystems wiederholt mit einem Bus Creator gebündelt und als „daten\_akku“ an das Subsystem *Signalgenerator* übergeben. Danach werden, wie beim Subsystem „Allgemeine Fahrzeugdaten“ erläutert, alle Signale zusammengefasst und an den Signalkollektor übertragen, wobei diese Prozedur für alle Zelldaten gleichermaßen abläuft.

Die Signalerzeugung der Akkutemperaturen verläuft analog zu den Zelldaten. Die einzigen Unterschiede sind, dass nur vier Temperaturen pro Zellenpack existieren und der Zeitvektor Wert sich auf „TIME\_10“ bezieht (s. 2.1.2). Die maximale sowie minimale Zellspannung, der Ladestrom sowie die Ladespannung werden direkt mittels „Repeating Sequence Interpolated- Blöcken im Subsystem *Akkudaten* erzeugt und auch direkt in diesem über den Bus Creator mit den anderen Signalen gebündelt. Der Wert für den Zeitvektor beläuft sich bei ihnen auf „TIME\_08“. Die Gesamtspannung verhält sich hierzu weitestgehend gleich, allerdings wird sie durch einen Constant-Block generiert. Die Balancingdaten werden i.A. äquivalent zu den Zelldaten aufbereitet, da zu jeder Zelle ein Balancing-Wert gehört. Lediglich die Signalerzeugung unterscheidet sich hier, da die Werte des Balancings mit Hilfe eines Constant-Blockes in dem Datentyp *boolean* erstellt werden.

### C) Dynamische Daten

Das Subsystem *Dynamische Daten* ist für die Erzeugung folgender Signale zuständig (s. Abb. A.3) : Geschwindigkeit, Beschleunigung, Gierrate, Drehzahlen der Räder, Wassertemperatur, Bremsdruck, Bremskraft, Bremsposition, Federweg, Gaspedalstellung und Lenkwinkel. Alle Signale werden mittels „Repeating Sequence Interpolated- Blöcken und der Einstellung „TIME\_08“ für den Wert Zeitvektors generiert, wobei die Werte der einzelnen Signale im \*.m-File signalgenerator\_microautobox.m (s. 2.1.2) definiert sind.

Das Signal „geschw1“ wird im Subsystem *Geschwindigkeit* erzeugt und mit den anderen Daten im Subsystem über einen Bus Creator gebündelt und als Signal „geschw“ das Subsystem *Dynamische Daten* übergeben, wo es dort mit den anderen Werten wieder mittels Bus Creator gebündelt wird. Das entstandene Signal „daten\_dynamisch“ wird im Subsystem *Signalgenerator* wieder mit den restlichen Signalen bzw. Subsystemen gebündelt und an den *Signalkollektor* geschickt.

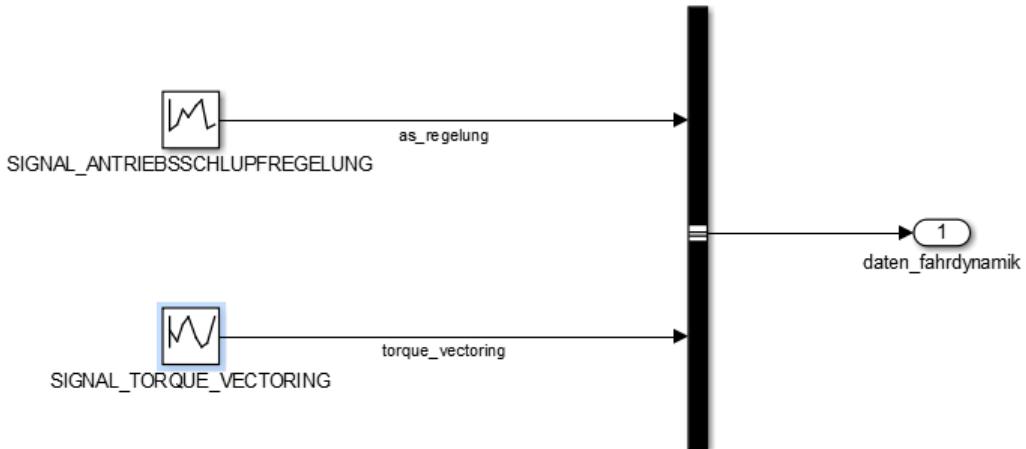


**Abbildung 2.17:** Erzeugung der Testsignale über die Geschwindigkeit des Fahrzeuges in X-, Y- und Z-Richtung mittels dreier RSI-Blöcke

Die Subsysteme *Beschleunigung*, *Gierrate*, *Drehzahlen der Räder*, *Wassertemperatur*, *Federweg* sowie *Gaspedalstellung* verhalten sich absolut äquivalent zu den Geschwindigkeitssignalen. Der Verlauf der Signale „Bremsdruck“, „Bremskraft“ und „Bremsposition“ verhält sich – bis auf deren Erzeugung bereits im Subsystem *Dynamische Daten* – absolut gleich.

#### D) Fahrdynamikregelung

Das Subsystem *Fahrdynamikregelung* ist für die Erzeugung der Testsignale „Antriebschlupfregelung“ und „Torque Vectoring“ verantwortlich (s. Abb. 2.18). Diese werden mit RSI-Blöcken gebildet, welche die gleiche Konfiguration wie bei dem Subsystem *Dynamischen Daten* besitzen. Beide Werte werden jeweils im Subsystem selbst erzeugt und über den Bus Creator gebündelt als Signal „daten\_dynamischän“ das Subsystem *Signalgenerator* übergeben, wobei der weitere Signalverlauf äquivalent zu den anderen Signalen in ebendiesem Subsystem ist.



**Abbildung 2.18:** Erzeugung der Testsignale für die Fahrdynamik des Fahrzeuges mittels zweier RSI-Blöcke

#### E) Motor- und Umrichterdaten

Im Subsystem *Motor- und Umrichterdaten* werden die Signale „DC Strom“, „DC Spannung“, „Motortemperatur“, „Stromgrenze“, „Maximalleistung“, „Lüfterdrehzahl“, „Lüfter“, „Pumpe“ sowie die Wassertemperatur des Motors erzeugt (s. Abb. A.4). Alle Signale werden erneut mit Hilfe von RSI-Blöcken generiert. Die Konfiguration dieser Blöcke gleicht denen des Subsystems *Dynamische Daten*. Die Signalaufbereitung ist bei allen Signalen äquivalent. Allein die Signale der Motortemperatur werden in einem eigenen Subsystem *Motortemperatur* und nicht direkt im Subsystem *Motor- und Umrichterdaten* erzeugt. Es findet auch wieder eine Bündelung der Signale mittels eines Bus Creator statt und das gebündelte Signal „daten\_motor“ wird an das Subsystem *Signalgenerator* übergeben. Der weitere Signalverlauf ist äquivalent zu den anderen Signalen im Subsystem *Signalgenerator*.

### 2.1.4 Signalkollektor

Der Signalkollektor stellt das zweite große Subsystem innerhalb des Simulink-Modells dar. Er hat die Aufgabe, den vom Subsystem *Signalgenerator* (s. 2.1.3) oder von einem späteren Subsystem von Team StarCraft e.V. erhaltenen Busarray wieder in die einzelnen Signale zu unterteilen, geeignet aufzubereiten und diese dann an den *Signaltransmitter* (s. 2.1.5) weiterzuleiten, welcher die erzeugten Testsignale bzw. Sensordaten an den Embedded-PC via Ethernet-Schnittstelle sendet. Während der Aufbereitung der Signale im Signalkollektor werden die folgenden Schritte durchgeführt:

#### 1) Verstärkung der Signale

Abhängig von der Anzahl der Nachkommastellen des Testsignals  $n$  mit  $n > 0$  wird dieses nun durch einen Gain-Block (s. ED S.11) mit dem Faktor  $10^n$  multipliziert, um für den aktuellen Wert des Testsignals jeweils einen ganzzahligen Wert zu erhalten, was zur Vereinheitlichung der zu übertragenden Daten beitragen soll.

### 2) Konvertierung der Datentypen

Die nunmehr ganzzahligen Werte werden von ihren Datentypen *boolean* oder *single* nun einheitlich mittels eines Convert-Blocks in den Datentyp *int16* konvertiert, wodurch alle Signale den gleichen Datentyp aufweisen. Schließlich ist noch eine Konvertierung in den Datentyp *double* vonnöten, da die sFunction *DSEncode32* (s. ED S.12) diesen Datentyp voraussetzt. Allerdings liefert diese sFunction als Ausgabe erneut den Datentyp *int16*, weshalb die vorherige Konvertierung am Eingang der sFunction nicht ins Gewicht fällt.

### 3) Umbenennung der Variablennamen

Mit dem Convert-Block ist es zudem möglich, dem Ausgangssignal unabhängig vom Eingangssignal einen festen bzw. neuen Variablennamen zu vergeben. Dies ist insofern nützlich, da bei einem möglichen Zugriff auf die Daten bzw. die Variablen durch den Embedded-PC oder der Software Control Desk von dSPACE diese immer die gleichen Variablennamen besitzen, unabhängig davon ob der Signalgenerator durch das Subsystem von Team StarCraft e.V. ersetzt wurde oder nicht, wodurch eine Vereinheitlichung des Modells realisiert wird welche die Wartungsaufgaben erleichtert.

Nachdem die Daten entsprechend der obigen Schritte aufbereitet wurden, werden diese erneut durch mehrere Bus Creator und Subsysteme auf einem zentralen Bus Creator zu einem Busarray gebündelt und an einen Encoder (s. ED S.12) übergeben, woraufhin sie nach deren Enkodierung via Ethernet-Schnittstelle an den Embedded-PC versendet werden.

## Aufbau

Die Daten liegen wie bereits in Abschnitt 2.1.3 erläutert am Eingang des Subsystems „Signalkollektor\_EMBEDDED\_PC“ an. Um ebendiese effizient und übersichtlich bearbeiten zu können, muss dieser Busarray wieder in seine einzelnen Signale zerlegt werden. Hierfür wird der Busarray zuerst mittels eines Bus Selectors (s. ED S.10) wieder in die fünf bekannten Hauptkategorien Allgemeine Fahrzeugdaten, Akkudaten, Dynamische Daten, Fahrdynamikregelung sowie Motor- und Umrichterdaten aufgespalten. Diese werden dann anschließend in ihren Subsystemen weiterverarbeitet und aufgespalten. Schlussendlich werden alle Signale erneut gebündelt und mit Hilfe von Bus Creators durch einen Convert-Block in den Datentyp *double* konvertiert, um diese für die sFunction *DSEncode32* passend vorzubereiten (s. Abb. A.5). Im Folgenden sollen daher nun der Aufbau und die Funktionsweise der einzelnen Subsysteme innerhalb des Subsystems „Signalkollektor\_EMBEDDED\_PC“ dargelegt werden.

### Subsystem daten\_allgemein / Allgemeine Fahrzeugdaten

Wie in ABB2 ersichtlich besteht auch dieses Subsystem aus mehreren Subsystemen sowie Simulink-Blöcken. Am Subsystem selbst liegen die Signale als Input vor, wie diese aus dem dazugehörigen Subsystem des Signalgenerators ausgegeben wurden. Diese werden durch einen Bus Selector wieder einzeln aufgetrennt.

Die Signale „gesamtspannung\_akku“ und „geschwindigkeit“ werden mittels des Gain-Blocks um den Faktor 10 vergrößert, um die eine Kommastelle zu beseitigen, damit diese bei der int16-Konvertierung im Convert-Block nicht verloren geht. Zuzüglich generiert der Convert-Block einen festen Variablenamen, welche für den Rest des Simulink-Modells gilt. Üblicherweise wird dem normalen Variablenamen vor Konvertierung noch der Präfix „emb“ hinzugefügt bzw. „daten“ durch „emb“ ersetzt. So geht aus „gesamtspannung\_akku“ der neue Signalname „emb\_gesamtspannung\_akku“ hervor. Diese Umwandlung der Variablennamen wird auch innerhalb der Subsysteme nach der Bündelung mit dem Bus Creator vorgenommen. Aus „daten\_notaus“ entsteht der neue Variablenname „emb\_notaus“. Das Signal „emb\_gesamtspannung\_akku“ wird durch einen Bus Creator mit den anderen Signalen seines Subsystems erneut gebündelt und mit dem Variablenamen „emb\_daten\_allgemein“ an das Subsystem „Signalcollector\_EMBEDDED\_PC“ übergeben. Dort wird diese Kategorie an Signalen erneut mit den anderen Signalen zu einem Busarray zusammengefügt, per Convert-Block in den Datentyp *double* umgewandelt, um es für die Enkodierung aufzubereiten. Schlussendlich werden diese Daten an den Signaltransmitter versandt.

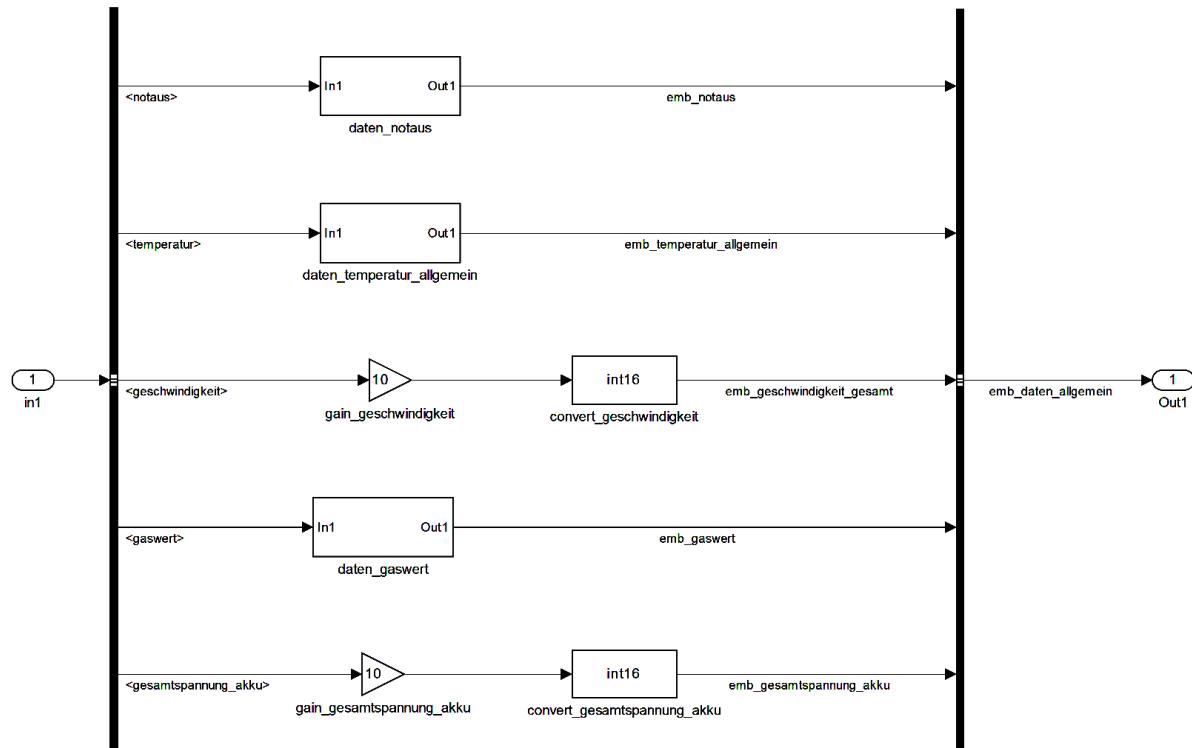


Abbildung 2.19: Übersicht über das Subsystem „daten\_allgemein“ im Signalkollektor

Des Weiteren wird das Subsystem *daten\_notaus* ohne Beschränkung der Allgemeinheit betrachtet (s. Abb. A.6). Innerhalb des Subsystems wird das Busarray mittels eines Bus Selectors in die einzelnen Signale aufgegliedert. Das Signal „notaus\_0“ wird mittels eines Convert-Blockes in den Datentypen *int16* konvertiert und erhält den neuen Variablenamen „emb\_notaus\_01“. Als Rundungsmethode für *boolean* Werte wurde „Floor“ gewählt, da diese auf ganze Zahlen runden. Dies ist nur logisch, da aus „true“ und „false“, eine logische 1 und eine logische 0 resultieren soll.

Danach bündelt ein Bus Creator dieses Signal mit den anderen seines Subsystems und übergibt die Daten an das Subsystem *daten\_allgemein* mit dem Variablenname „emb\_notaus“ (s. ABB3). Anschließend werden die Daten wie oben erläutern weiter verarbeitet (Umbennung des Busarray, Bündelung, Konvertierung und Transfer an Signaltransmitter). Die anderen Signale des Subsystems werden äquivalent aufbereitet. Dieses Vorgehen wird in diesem Modell bei allen boolean Werten im Signalkollektor angewendet.

Zur Verdeutlichung des Unterschieds von boolean und nicht-boolean Werten wird zusätzlich als Beispiel das Subsystem *daten\_temperatur\_allgemein* betrachtet. Die Auftrennung der Signale mittels Bus Selector sowie die Umbenennung durch den Convert-Block gleichen dem Vorgehen des Subsystems *daten\_notaus*. Differierend dazu sind die Multiplikation der Werte mittels Gain-Blöcke und die Rundungsmethode. Diese Unterschiede erschließen sich aus dem vorherrschenden Datentyp, welcher nicht *boolean* ist. Wie schon oben betrachtet werden die Signale mittels Gain-Block um den Faktor 10 vergrößert, um die Kommastelle verschwinden zu lassen, damit die Genauigkeit des Wertes durch den Convert-Block bei der Konvertierung in den Datentyp *int16* nicht beeinträchtigt wird. Weiterhin wird die Rundungsmethode „Round“ verwendet, welche den bekannten geodätischen Runden entspricht und eine zufriedenstellende Genauigkeit liefert. (s. Abb. A.7). Die restliche Aufbereitung bis hin zum Transferieren der Werte gleicht dem oben beschriebenen Vorgehen. Dieses Vorgehen wird in diesem Modell bei allen Werten welche nicht den Datentyp *boolean* aufweisen im Signalkollektor angewendet, abgesehen natürlich vom Gain-Block, dessen Multiplikator je nach Anzahl der erforderlichen Kommastellen unterschiedlich sein kann. Die Verarbeitung der Signale im Subsystem *daten\_gaswert* verlaufen hierzu äquivalent. Der einzige Unterschied dabei ist, dass die Verstärkung des Gain-Blockes 1000 beträgt, da bei diesen Werten eine Genauigkeit von drei Kommastellen vorgegeben ist (s. Pflichtenheft S.4) und diese Kommastellen mit der Vergrößerung beseitigt werden.

#### *Subsystem „daten\_akku“ / Akkudaten*

Wie im letzten Kapitel erwähnt (s. Subsystem „*daten\_allgemein*“), verhalten sich alle Subsysteme im Signalkollektor sehr ähnlich. Es existieren in diesem Subsystem zwei Subsysteme, „*daten\_zelldaten*“ sowie „*daten\_temperatur*“, welche sich um die Verarbeitung von Werten kümmern, welche nicht den Datentypen *boolean* aufweisen. Darüber hinaus existiert noch ein Subsystem mit boolean-Werten – „*daten\_balancing*“ – und einzelne Signale, welche ebenfalls Werte verkörpern, die nicht den Datentyp *boolean* besitzen. Wie erwähnt findet auch eine Signalkonvertierung, eine Umbenennung der Variablennamen, sowie teilweise eine Verstärkung der Signale mittels Gain-Blöcken statt. Die weitere Signal-aufbereitung im Subsystem *Signalcollector\_EMBEDDED\_PC* entspricht der des Subsystems *daten\_allgemein* (s. letzten Abschnitt). Die Verarbeitung der Signale, welche sich nicht nochmals in einem Subsystem befinden, gleicht der aus dem vorherigen Abschnitt. Einzig die Faktoren des Verstärkung der Signale „*zellspannung\_max*“ und „*zellspannung\_min*“ sind aufgrund der vorgegebenen Genauigkeit (s. Pflichtenheft S.4) in diesem Fall zu 1000 gewählt.

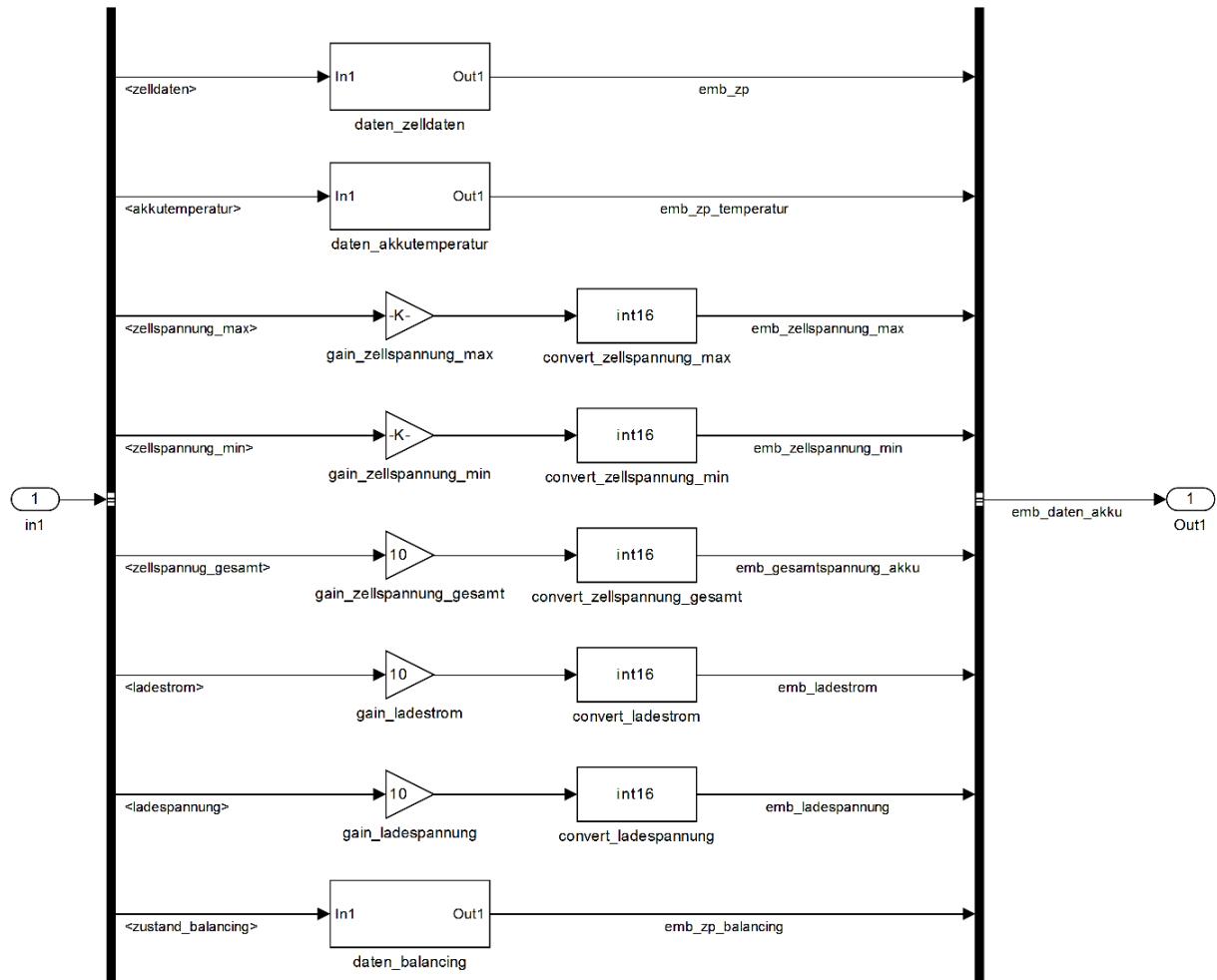


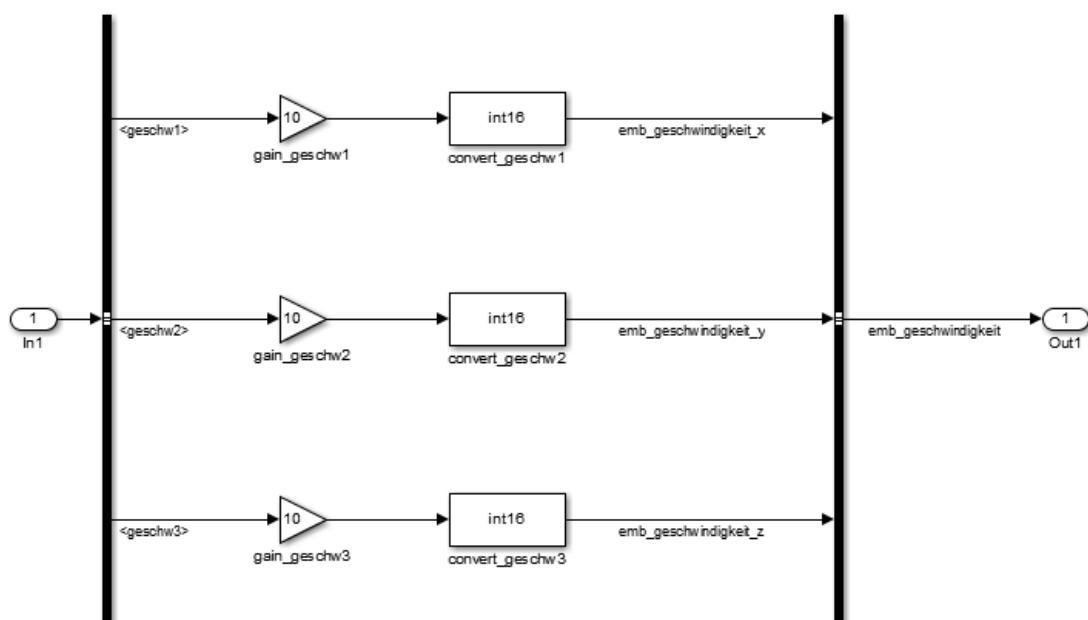
Abbildung 2.20: Übersicht über das Subsystem „daten\_akku“ im Signalkollektor

Die Subsystemstruktur soll anhand des Subsystems „daten\_zelldaten“ (s. Abb. A.8) ohne Beschränkung der Allgemeinheit kurz näher erläutert werden. Ähnlich wie im dazugehörigen Subsystem aus dem Signalgenerator (s. 2.1.3) ist auch dieses System aufgebaut. Das Signal „zelle\_001“ im Subsystem „daten\_zellenpack\_01“ des Subsystems „daten\_zelldaten“ wird mit einem Multiplikator von 1000 durch den Gain-Block versehen und mittels Convert-Block in den Datentyp *int16* umgewandelt und in „emb\_zp\_01\_zelle01“ umbenannt. Dabei kann die Zelle jeweils genau einem Zellenpack zugeordnet werden. Anschließend wird das Signal mit dem Bus Creator mit den restlichen Signalen seines Subsystems gebündelt und mit dem Variablenamen „emb\_zp\_01“ an das Subsystem „daten\_zelldaten“ übergeben. Im Subsystem „daten\_zelldaten“ (s. Abb. A.9) wird das gebündelte Signal nochmals mit den anderen gebündelten Signalen des Subsystems über einen Bus Creator abermals zusammengefasst und mit dem Namen „emb\_zp“ für das Busarry an das Subsystem „daten\_akku“ übermittelt. Dort wird es mit den restlichen Daten wieder gebündelt und an das Subsystem „Signalcollector\_EMBEDDED\_PC“ geliefert. Die weitere Signalverarbeitung ist durch die oberen Abschnitte bereits ausführlich erläutert worden und geschieht für alle Signale des Subsystems „daten\_zelldaten“ auf die gleiche Weise.

Das Subsystem „*daten\_akkutemperatur*“ verhält sich dazu analog, nur kommen darin keine Gain-Blöcke vor, da die Genauigkeit auf ganze Zahlen festgelegt wurde (s. Pflichtenheft S.4). Das Subsystem „*daten\_balancing*“ verhält sich ebenfalls vom Aufbau her dazu gleich und die Signalverarbeitung gleicht natürlich den Werten mit dem Datentyp *boolean*.

### *Subsystem „daten\_dynamisch“/ Dynamische Daten*

Das Subsystem „*daten\_dynamisch*“ beinhaltet mehrere Subsysteme sowie weitere einzelne Signale. Die Datenaufbereitung ist aus den vorherigen Kapiteln bekannt. Das Subsystem weist darüber hinaus keine weiteren Besonderheiten auf, da keine Werte mit dem Datentyp *boolean* darin vorhanden sind. Die Signale „*bremsdruck*“, „*bremskraft*“, „*lenkwinkel\_ddünd*“ und „*bremsposition*“ werden direkt im Subsystem aufbereitet. Die Signale „*bremskraft*“ und „*bremsposition*“ werden zusätzlich mittels Gain-Blöcken mit einer Verstärkung von 1000 versehen. Danach werden alle vier Signale in den Datentyp *int16* konvertiert sowie deren Variablennamen umbenannt und mit den restlichen Signalen des Subsystems gebündelt und an das Subsystem „*Signalcollector\_EMBEDDED\_PC*“ weitergeleitet, wobei der weitere Verlauf aus den obigen Ausführungen zu entnehmen ist. Die Verarbeitung innerhalb der Subsysteme soll ohne Beschränkung der Allgemeinheit beispielhaft am Subsystem „*daten\_geschwindigkeit*“ gezeigt werden. Das Signal „*geschw1*“ wird mittels eines Bus Selectors aus dem Busarray herausgelöst und mit einem Multiplikator von 10 durch den Gain-Block versehen. Der Convert-Block ändert den Datentyp des Signals in *int16* und benennt es in „*emb\_geschwindigkeit\_x*“ um. Infolgedessen wird es durch den Bus Creator mit den anderen Signalen des Subsystems zusammengefasst und als „*emb\_geschwindigkeit*“ ans Subsystem „*daten\_dynamisch*“ weitergeleitet(s.ABB8). Danach wird dieses Busarray mit den weiteren Werten des Subsystems gebündelt und zu dem Subsystem „*Signalcollector\_EMBEDDED\_PC*“ transferiert. Die weitere Verarbeitung erfolgt analog zu den vorherigen Kapiteln.

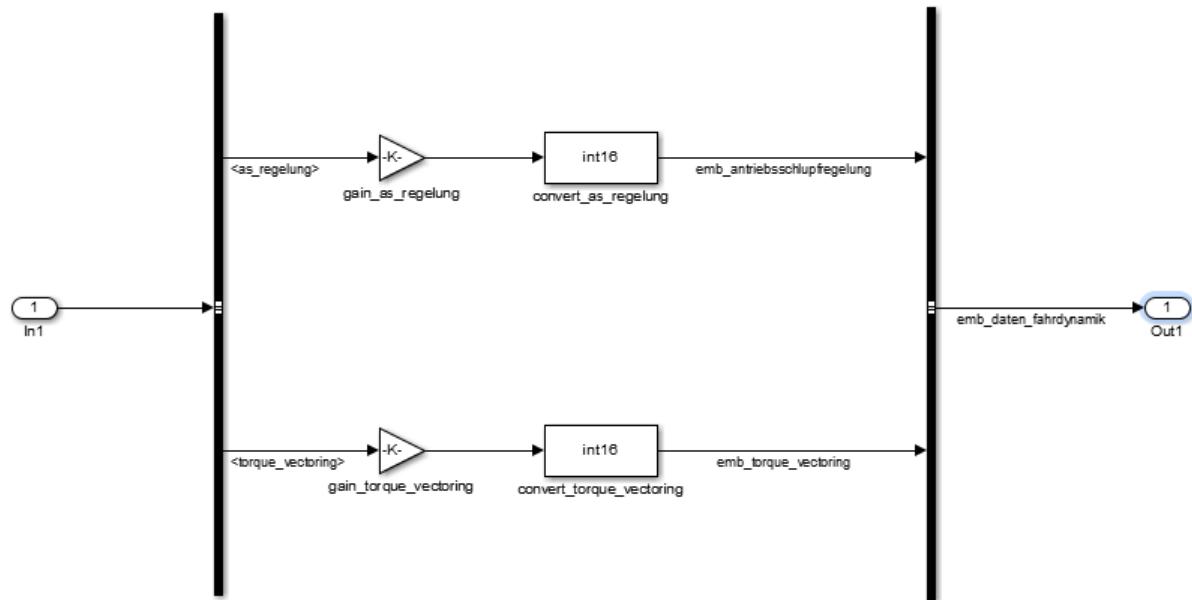


**Abbildung 2.21:** Übersicht über das Subsystem „*daten\_geschwindigkeit*“ im Subsystem „*daten\_dynamisch*“ im Signalkollektor

Die weiteren Subsysteme im Subsystem „daten\_dynamisch“ – „daten\_beschleunigung“, „daten\_gierrate“, „daten\_drehzahl\_rad“, „daten\_wassertemperatur“, „daten\_federweg“ und „daten\_gaspedalstellung“ – verhalten sich dazu äquivalent. Zu bemerken ist nur, dass der Multiplikator bei den Signalen des Subsystems „daten\_gaspedalstellung“ nicht 10 sondern 1000 beträgt.

### *Subsystem „daten\_fahrdynamik“/ Fahrdynamikregelung*

Wie in Abb. 2.22 zu erkennen besteht das Subsystem genau wie das dazugehörige Subsystem im Signalgenerator nur aus zwei Signalen. Beide Werte werden wie üblich aus dem Busarray mittels eines Bus Selectors heraus getrennt. Beiden Signalen wird ein Multiplikator von 1000 durch die Gain-Blöcke zugeführt und die Konvertierung in *int16* und Umbenennung findet durch die Convert-Blöcke statt. Darauf folgend werden beide Signale wieder mit einem Bus Creator gebündelt und das zusammengefasste Busarray wird mit dem Namen „emb\_daten\_fahrdynamik“ an das übergeordnete Subsystem „Signalcollector\_EMBEDDED\_PC“ übergeben. Die Weiterverarbeitung in diesem Subsystem gleicht der aus den vorherigen Kapiteln.



**Abbildung 2.22:** Übersicht über das Subsystem „daten\_fahrdynamik“ im Signalkollektor

### *Subsystem „daten\_motor“/ Motor- und Umrichterdaten*

Dieses Subsystem besitzt keinerlei besondere Eigenschaften, welche nicht bereits in den oberen Subsystemen beschrieben worden wären. Es besteht aus einem Subsystem und verschiedenen einzelnen Signalen (s. Abb. 2.24). Es sind keine boolean-Werte im Subsystem vorhanden. Die Aufbereitung der einzelnen Signale im Subsystem ist wie bei den Signalen in den vorherigen Kapiteln. Die Signale „dc\_strom“, „dc\_spennung“, „lüfterdrehzahl“ sowie „mwassertemperatur“ werden mittels Bus Selector aus dem Busarray separiert, erhalten einen Multiplikator von 10 durch die Gain-Blöcke und werden mittels der Convert-Blöcke in den Datentyp *int16* konvertiert und umbenannt.

Anschließend fasst sie der Bus Creator mit den anderen Signalen des Subsystems zusammen und übergibt die Daten an das Subsystem „emb\_daten\_fahrdynamik“. Die weitere Signalverarbeitung gleicht der aus den vorherigen Kapiteln in diesem Subsystem. Die Signale „stromgrenze“, „maximalleistung“, „lüfter“ und „pumpe“ werden analog behandelt, jedoch werden sie mit den Gain-Blöcken mit einem Multiplikator von 1000 versehen. Die Aufbereitung der Signale im Subsystem „daten\_temperatur\_motor“ ist analog der des Subsystems „daten\_geschwindigkeit“ zu betrachten.

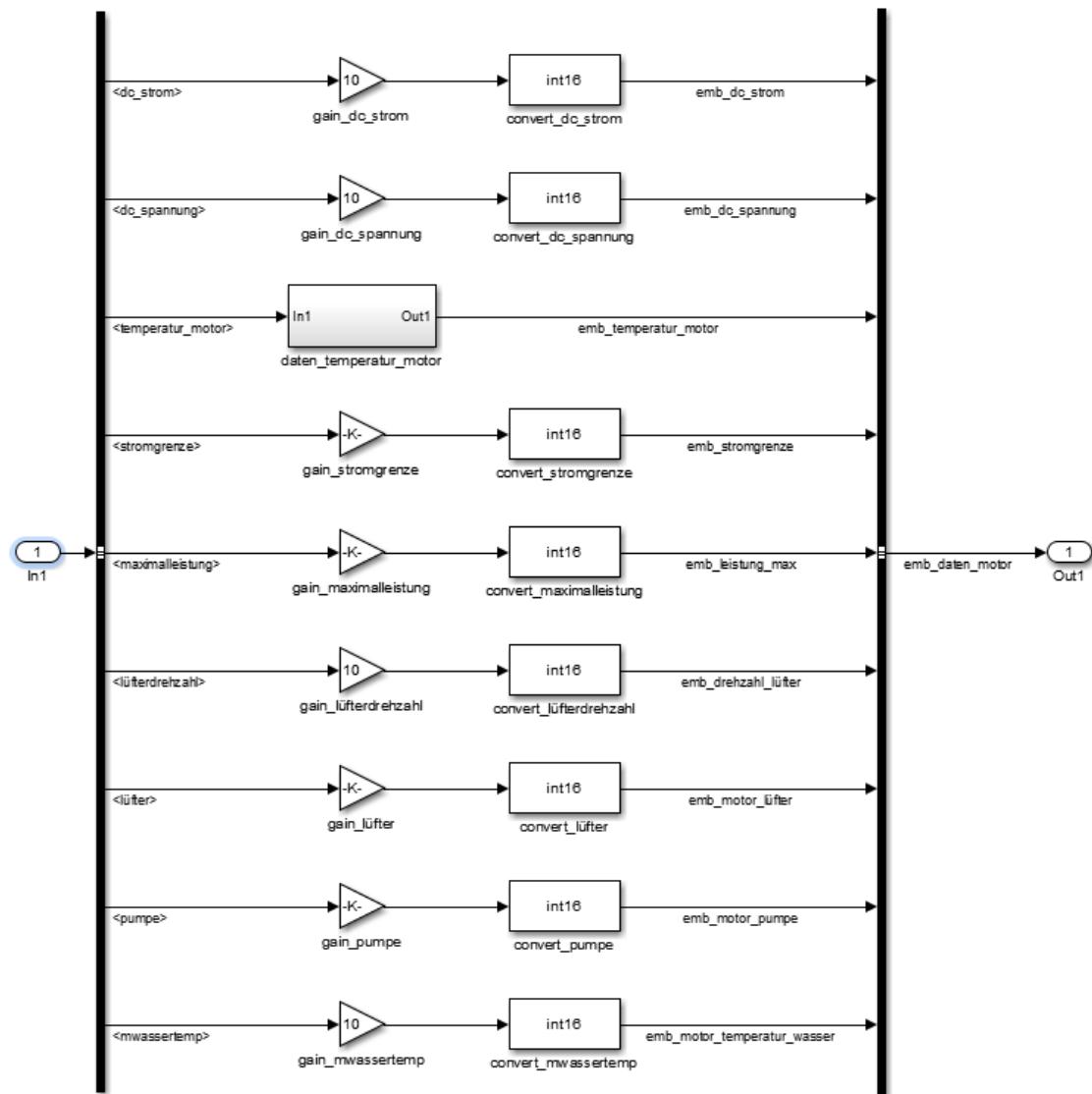


Abbildung 2.23: Übersicht über das Subsystem „daten\_motor“ im Signalkollektor

## 2.1.5 Signaltransmitter

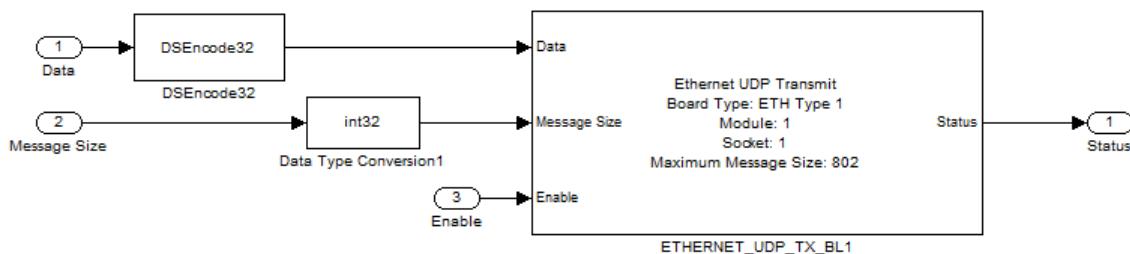
Der Signaltransmitter ist als letztes der drei übergeordneten Subsysteme des Simulink-Modells für die folgenden Aufgaben zuständig:

- Versand der im Signalkollektor (s. 2.1.4) aufbereiteten Fahrzeugdaten via Ethernet-Schnittstelle (UDP) an den Embedded-PC
- Generierung der benötigten Informationen über das zu versendende Datenpaket
- Versand dieser Paketinformationen an den Embedded-PC
- Umsetzung der bidirektionalen Kommunikation zwischen der MicroAutoBox II und dem Embedded-PC

Im Folgenden soll daher nun anhand des Aufbaus des Signaltransmitters in Abb. A.11 die Realisierung der einzelnen Aufgabenbereiche innerhalb dieses Subsystems aufgezeigt werden.

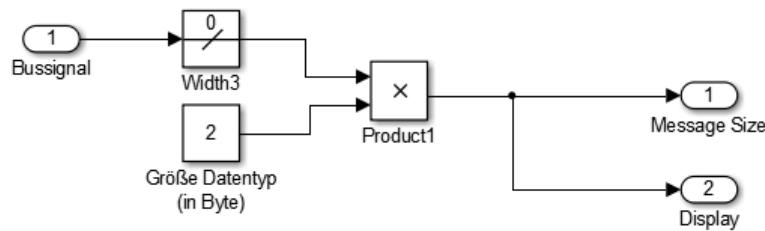
### 1) Versand der Fahrzeugdaten via Ethernet-Schnittstelle an den Embedded-PC

Für diese Aufgabe sind die beiden Subsysteme „MSGSIZE\_DATEN“ und „UDP\_DATEN“ vorgesehen. Die Fahrzeugdaten, welche über den Import-Block in das Subsystem geleitet werden, werden an den Port „Data“ des Subsystems „UDP\_DATEN“ gesendet, woraufhin sie via Ethernet-Schnittstelle an den Embedded-PC übertragen werden. Das Subsystem selbst besitzt den folgenden Aufbau:



**Abbildung 2.24:** Übersicht über das Subsystem „UDP\_DATEN“ im Signaltransmitter

Wie aus den Erläuterungen im Entwurfsdokument bekannt ist (s. S.12-13), werden die Daten von einer sFunction *DSEncode32* vom obligatorischen Datentyp *double* am Eingang des Blockes in den Datentyp *int16* gewandelt und als *uint32* - Datenstrom an den Embedded-PC übertragen. An dem zweiten Port namens „Message Size“ des UDP-Send-Blockes (s. Entwurfsdokument S.13) soll ein konstanter Wert – wie schon der Convert-Block andeutet – im Datentyp *int32* vorliegen, welcher die Paketgröße der zu versendenden Daten angibt. Um für die Fahrzeugdaten auch nach Änderungen des Simulink-Modells durch Hinzufügen oder Reduzieren von Fahrzeugdaten (s. 2.1.3) jeweils automatisch die korrekte Paketgröße zu ermitteln und an den Port „Message Size“ anzulegen, wurde das Subsystem *MSGSIZE\_DATEN* dem Modell hinzugefügt (s. Abb. 2.25).

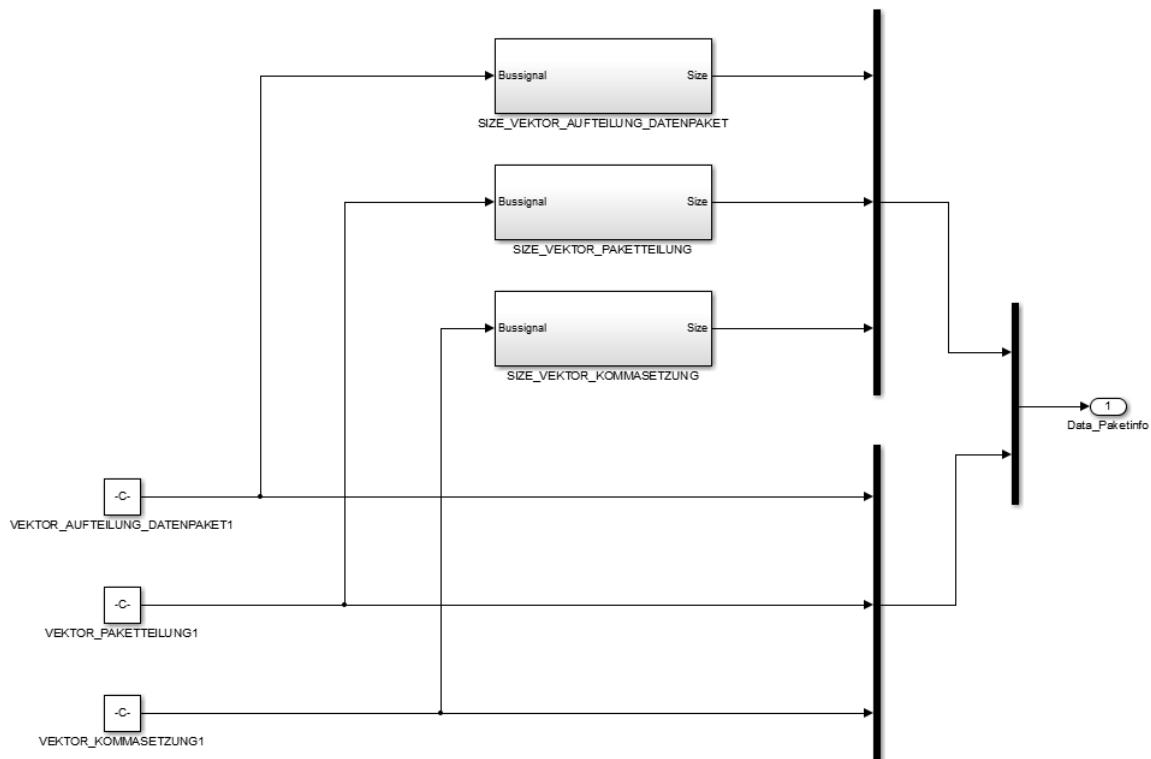


**Abbildung 2.25:** Übersicht über das Subsystem „MSGSIZE\_DATEN“ im Signaltransmitter

In diesem Subsystem wird als Eingangssignal das Busarray der Fahrzeugdaten über einen Width-Block (s. S.5) geleitet und daraus die Breite des Busarrays bestimmt. Dieser Wert wird zusammen mit einem Constant-Block, welcher den Wert 2 besitzt, auf einen Product-Block geleitet und nach einer Konvertierung in den Datentyp *int32* im Subsystem *UDP-DATEN* an den Port „Message Size“ angelegt.

Weiterhin muss im UDP-Send-Block nochmals die automatisch durch das Modell ermittelte Größe des Datenpaketes eingegeben werden, wofür leider noch nicht ein automatisierter Ablauf gefunden werden konnte. Dies dürfte allerdings spätestens in der Testphase behoben sein.

## 2) Generierung der benötigten Informationen über das zu versendende Datenpaket



**Abbildung 2.26:** Übersicht über das Subsystem „DATEN\_PAKETINFO“ im Signaltransmitter

Diese Aufgabe wird ausschließlich durch das Subsystem *DATEN\_PAKETINFO* bewerkstelligt. Hierbei werden in drei Constant-Blöcke die zuvor in der Config-Datei config\_datenpaket.m referenzierten Vektoren (s. 2.1.2) eingebunden und von ebendiesen durch drei Subsysteme deren jeweilige Breiten ausgelesen. Diese Signalvektoren und die jeweilige Größen der Vektoren werden getrennt auf einen Multiplexer geleitet und abschließend nochmals mittels eines Multiplexers zu einem Gesamtsignalvektor „data\_paketinfo“ zusammengefasst, welcher nach dem Verlassen des Subsystems die folgende Struktur aufweist:

$$data\_paketinfo = \begin{pmatrix} SIZE\_VEKTOR\_AUFTEILUNG\_DATENPAKET \\ SIZE\_VEKTOR\_PAKETTEILUNG \\ SIZE\_VEKTOR\_KOMMASETZUNG \\ VEKTOR\_AUFTEILUNG\_DATENPAKET \\ VEKTOR\_PAKETTEILUNG \\ VEKTOR\_KOMMASETZUNG \end{pmatrix}$$

### 3) Versand der Paketinformationen an den Embedded-PC

Der Versand der Paketinformationen an den Embedded-PC erfolgt im Subsystem „UDP\_PAKETINFORMATIONEN“ auf analoge Weise zu dem Versand der Fahrzeugdaten (statt des Subsystems „MSGSIZE\_DATEN“ wird hier nur das dazu inhaltlich äquivalente Subsystem „MSGSIZE\_PAKETINFO“ verwendet). Lediglich bei der Konvertierung wird aufgrund der begrenzten maximalen Paketgröße eines Paketes beim UDP-Protokoll von 1472 byte der Datentyp *unit8* ausgewählt, um den Versand aller Paketinformationen in einem einzigen Paket zu ermöglichen. Demzufolge muss auch im Subsystem „MSGSIZE\_PAKETINFO“ welches die Paketgröße ermittelt, beim Constant-Block der Wert „2“ beim Produktblock durch den Wert „1“ ersetzt oder zusammen mit dem Produktblock entfernt werden.

### 4) Umsetzung der bidirektionalen Kommunikation zwischen der MicroAutoBox II und dem Embedded-PC

Die Umsetzung der bidirektionalen Kommunikation zwischen der MicroAutoBox II und dem Embedded-PC wird durch das Subsystem „EMB\_RECEIVE“ (s. Abb. A.12) und zwei UND-Gattern (s. S.4) realisiert. Ein UDP-Receive-Block – das Gegenstück zum UDP-Send-Block – empfängt von dem Embedded-PC einen Bytevektor, welcher mittels der sFunction *DSDecode32* dekodiert wird. Nachdem dieser anschließend über einen Demultiplexer geleitet wurde, werden die verschiedenen Sample Times des Demultiplexers und der sFunction „Byte\_to\_8Bit\_Decoder“ mittels eines Rate-Transition-Blocks (s. S.5) aneinander angeglichen. Letzterer Block sorgt dafür, dass aus dem gesendeten Bytevektor die einzelnen Bits getrennt herausgeführt werden können. Von diesen 8 Bits werden jedoch nur 4 Bits (Bit 0-3) benötigt, wodurch die anderen 4 Bits frei gewählt werden können. Die Bits 0-3 werden nun nach einer Konvertierung in den Datentyp *boolean* an zwei RS-Flip-Flops (s. S.7) angeschlossen, welche nach einem weiteren Rate-Transition-Block aus dem Subsystem herausgeführt werden. Anschließend werden die Signale an zwei UND-Gatter auf der in Abb. A.11 gezeigten Weise angeschlossen, deren Ausgänge jeweils mit dem Input „Enable“ der beiden Subsysteme „UDP\_DATEN“ und „UDP\_PAKETINFORMATIONEN“ verbunden

sind. Somit lässt sich je nach dem Inhalt des an die MicroAutoBox II übertragenen Bytevektors vom Embedded-PC aus genau steuern, welcher von den beiden UDP-Send-Blöcken – „UDP\_DATEN“ oder „UDP\_PAKETINFORMATIONEN“ – aktiv sein sollen, was durch die nachfolgende Tabelle nochmals verdeutlicht werden soll:

**Tabelle 2.4:** Wertetabelle Bytevektor

Bytevektor (LSB - MSB)	UDP_DATEN	UDP_PAKETINFORMATIONEN
[1010 * * * *]	0	0
[0110 * * * *]	0	1
[1001 * * * *]	1	0

## 2.2 Embedded PC und virtueller Server

An der Datenübertragung und Verarbeitung sind drei von uns entworfene Anwendungen beteiligt. Aus dem in Abschnitt 2.1 bereits beschriebenen Simulink-Modell wird mithilfe der Entwicklungswerzeuge von dSPACE automatisch C-Code erzeugt und für die Ausführung auf der MicroAutoBox II kompiliert. Diese Anwendung wird im Folgenden als *mabxii* bezeichnet. Des Weiteren wird jeweils eine von uns in C++ implementierte Anwendung auf dem Embedded-PC und auf dem vServer laufen, welche von nun an *embpc* und *vserver* genannt werden. Bei deren Entwurf wurde insbesondere darauf geachtet, dass beiden Anwendungen eine möglichst große gemeinsame Code-Basis zugrundeliegt. Dadurch soll zum einen die Fehlerwahrscheinlichkeit minimiert werden, in dem Code-Abschnitte häufiger ausgeführt werden und dadurch Fehler früher erkannt und behoben werden können. Zum anderen soll dadurch Zeit bei der Implementierung und dem Testen eingespart werden, die letztendlich der Qualität der Software zugute kommt.

Im Folgenden wird die konkrete Funktionsweise dieser drei Anwendungen anhand einer schrittweisen Beschreibung eines Kommunikationsverlaufs erläutert. Siehe dazu auch Abbildung 2.27.

### 2.2.1 Initialisierung

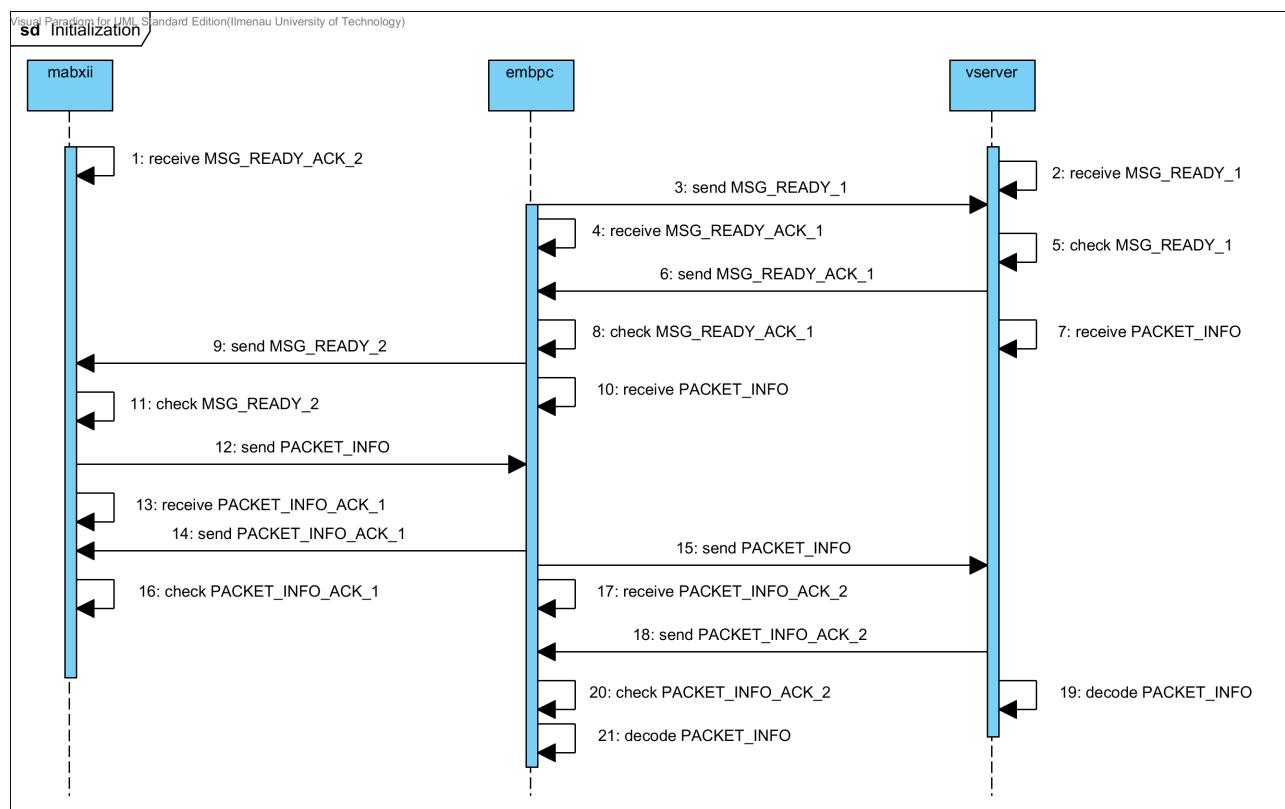


Abbildung 2.27: Initialisierung der Kommunikation

- 1) - 2) Der virtuelle Server sowie die MicroAutoBox II sind nach deren Start in einer Art Wartezustand(1/2: `receive MSG_READY`), in dem sie auf die Nachricht `MSG_READY`

des Embedded-PCs warten. Dabei handelt es sich um einen mit einer bestimmten Bitfolge bestückten Bitvektor der Länge ein Byte. (Siehe Abbildung 2.2.2) Dieser signalisiert beiden Teilnehmern die Bereitschaft des Embedded-PCs.

- 3) - 4) Sobald der Embedded-PC hochgefahren ist, wird die Anwendung *embpc* automatisch ausgeführt. Der PC ist nach fehlerfreiem Start dazu in der Lage, sein erstes Paket zu senden und damit seine Bereitschaft mitzuteilen. Das Paket **MSG\_READY** wird über eine UMTS-Verbindung zunächst an den virtuellen Server gesendet (3: **send MSG\_READY**) und beinhaltet den oben beschriebenen Bitvektor. Nach dem Sendevorgang wartet er auf ein Acknowledgement **MSG\_READY\_ACK** des vServers (4: **receive MSG\_READY\_ACK**).
- 5) - 7) Hat der virtuelle Server diesen Bitvektor erfolgreich empfangen, wird dieser zunächst dekodiert und auf seine Richtig- bzw. Vollständigkeit geprüft (5: **check MSG\_READY**). Danach sendet dieser dem Embedded-PC eine Empfangsbestätigung (6: **send MSG\_READY\_ACK**), wieder in Form eines 8 bit großen Bitvektors bestimmter Kodierung (siehe Abbildung VERWEIS). Anschließend wartet der vServer auf ein Paket **PACKET\_INFO** (7: **receive PACKET\_INFO**). Falls nicht der erwartete Bitvektor empfangen wurde, wird wieder mit Schritt 2: **receive MSG\_READY** fortgefahren.
- 8) - 10) Sobald das Paket **MSG\_READY\_ACK** den Embedded-PC erreicht, wird dieses, wie auf dem vServer, zunächst dekodiert und überprüft (8: **check MSG\_READY\_ACK**). Falls es korrekt empfangen wurde, meldet die Anwendung *embpc* nun ihre Bereitschaft an die MicroAutoBox II, durch das Senden der Nachricht **MSG\_READY** (9: **send MSG\_READY**) und wartet anschließend auf ein Paket (10: **receive PACKET\_INFO**). Falls jedoch nach 5 Sekunden noch kein Acknowledgement des vServers eingetroffen ist, wird wieder mit Schritt 3: **send MSG\_READY** begonnen.
- 11)- 13) Hier wird auf der MicroAutoBox II analog zu schritt 5 und 8 verfahren und das ankommende Paket zunächst überprüft (11: **check MSG\_READY**). Falls nicht das erwartete Paket empfangen wurde, wird mit Schritt 1 fortgefahren. Ansonsten sendet die MicroAutoBox II ein Paket, das die Informationen zur Paketaufteilung, der Kommaverschiebung und Datentypen der einzelnen Werte beinhaltet, an den Embedded PC (12: **send PACKET\_INFO**). Danach wird auf eine Empfangsbestätigung des Embedded-PC gewartet (13: **receive PACKET\_INFO\_ACK**). Trifft diese nicht innerhalb von 5 Sekunden ein, wird die Nachricht erneut verschickt.
- 14) - 17) Nachdem der Embedded-PC das Paket **PACKET\_INFO** empfangen hat, schickt er zunächst eine Empfangsbestätigung an die MicroAutoBox II (14: **send PACKET\_INFO\_ACK**) und leitet anschließend das Paket an den vServer weiter (15: **send PACKET\_INFO**). Nun wartet er auf die Empfangsbestätigung des vServers (17: **receive PACKET\_INFO\_ACK**). Falls innerhalb von 5 Sekunden keine Nachricht eintrifft, wird das Paket erneut verschickt.
- 18) - 20) Sobald der vServer das Paket **PACKET\_INFO** entgegen nimmt, bestätigt er dessen Empfang (18: **send PACKET\_INFO\_ACK**) und fängt anschließend mit der Dekodierung der Paketinformationen an (20: **decode PACKET\_INFORMATION**). Dabei werden die einzelnen Daten (Paketaufteilung, Kommaverschiebung und Datentypen) in einer

jeweils eigenen Datenstruktur abgelegt (VEC\_LAYOUT, VEC\_COMMA, VEC\_DATATYPES). Auf diese wird im späteren Verlauf zugegriffen, um die Fahrzeugdaten korrekt verarbeiten zu können. Der Embedded-PC verfährt nach dem Empfang der Empfangsbestätigung genauso (19: decode PACKET\_INFO).

Der Grund für diesen ausführlichen Initialisierungsprozess ist, dass es für die Verarbeitung der Fahrzeugdaten wichtig ist, dass sowohl der Embedded-PC, als auch der vServer über die Paketinformationen verfügen. Durch deren Größe wäre es aufgrund der geringen Bandbreite von UMTS/GPRS keine Option, diese mit jedem Datenpaket zu übertragen. Falls sich die bisherige Initialisierung in der Testphase als zu fehleranfällig herausstellen sollte, indem z.B. keine Datenübertragung zustande kommt, weil zu viel Paketverlust auftritt, wird dieser Prozess selbstverständlich überarbeitet. Das bisherige Konzept ist also noch nicht endgültig.

## 2.2.2 Datenübertragung und Datenverarbeitung

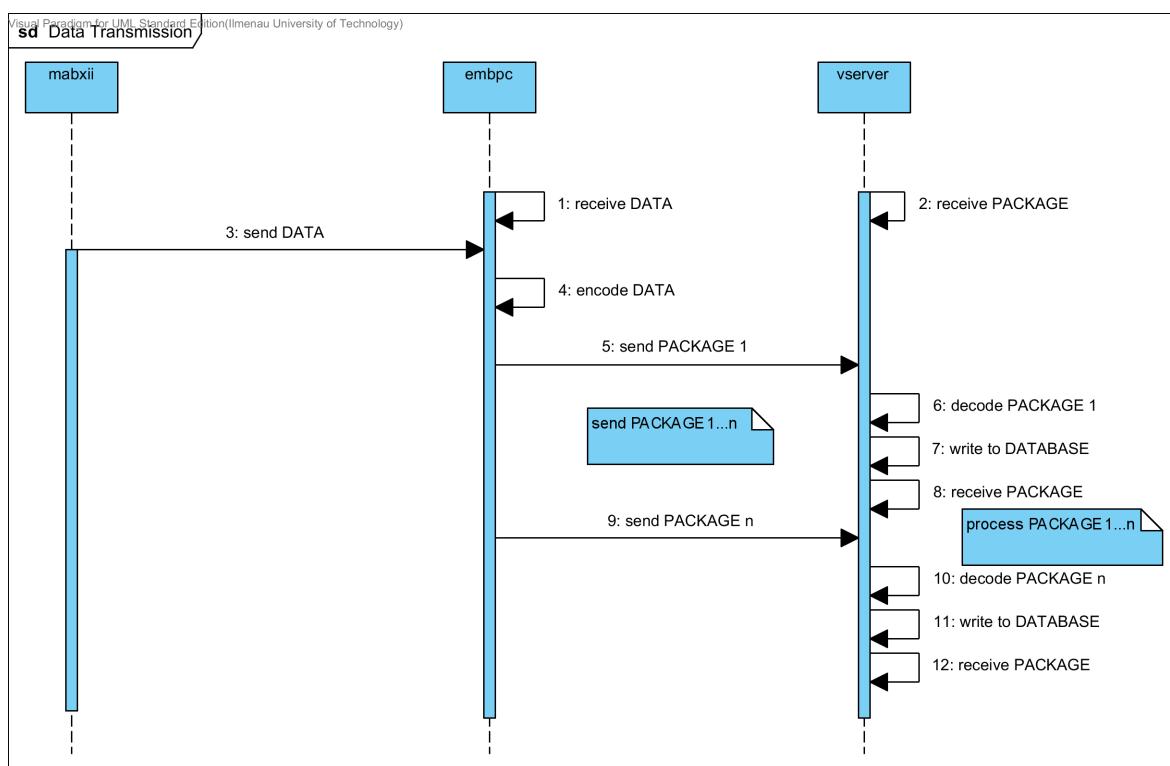


Abbildung 2.28: Übertragung der Fahrzeugdaten

- 1) - 3) Nachdem die MicroAutoBox die Empfangsbestätigung des PACKET\_INFO vom Embedded-PC erhalten hat, beginnt sie nun in einem festen Zeitabstand, UDP-Pakete mit aktuellen Fahrzeugdaten an den Embedded-PC zu senden (3: `send DATA`). Dabei wird nicht berücksichtigt, ob die Pakete den Embedded-PC auch tatsächlich erreichen. Die Konfiguration des Zeitabstandes erfolgt im Matlab/Simulink-Programm mit Hilfe einer sogenannten sample-time (optional-clock). Sowohl der Embedded-PC, als auch der vServer warten nun auf ankommende Daten (1: `receive DATA`, 2: `receive PACKAGE`).

- 4) - 12) Der Embedded-PC nimmt die Fahrzeugdaten entgegen und verarbeitet diese (4: `encode DATA`). Dabei wird der Datensatz, `DATA` entsprechend der Informationen in `VEC_LAYOUT`, in mehrere Pakete geteilt und mit einem `HEADER` versehen. Anschließend werden diese Pakete der Reihe nach an den vServer übertragen (5, 9, . . . : `send PACKAGE 1, . . . ,n`). Dabei wird keine Empfangsbestätigung am Server erzeugt und somit auch keine am Embedded-PC erwartet. Nach Abschluss der Entwurfsphase wurde die Idee umgesetzt, die eingehenden Fahrzeugdaten am Embedded-PC in mehrere Pakete aufzuteilen, um bei eventuell auftretenden Verbindungsproblemen zumindest einzelne Pakete erfolgreich an den virtuellen Server übertragen zu können. Das Programm auf dem PC sorgt dafür, dass jedes Paket vor dem Senden mit einem aktuellen acht Byte Zeitstempel und einer vier Byte großen Identifikationsnummer (ID) versehen ist (`Zeitstempel + ID = HEADER`). Zum Erzeugen des Zeitstempels, mit Nanosekunden-Auflösung wird die Echtzeituhr des PCs genutzt (`clock_gettime(CLOCK_REALTIME, Zeitstempel-Variable)`). Die ID repräsentiert die Paketnummer, daher ist sie also eine Zahl zwischen eins und der Anzahl der getrennten Pakete. Standardmäßig wurde diese in Anlehnung an das Pflichtenheft auf fünf Pakete gesetzt, kann aber im mabxii Programm geändert werden.
- Sobald ein Paket den vServer erreicht, wird es zunächst dekodiert (6: `decode PACKAGE 1`) (wobei `PACKAGE 1` hier nicht zwangsläufig das erste Paket aus `DATA` sein muss – Pakete können sich überholen oder verloren gehen). Dabei wird zunächst der `HEADER` ausgelesen. Der darin enthaltene Zeitstempel wird mit dem lokalen verglichen, ist die Differenz größer als fünf Sekunden, wird das Paket verworfen, ansonsten wird das Paket entsprechend der Daten in `VEC_LAYOUT`, `VEC_COMMA` und `7VEC_DATATYPES` so bearbeitet, dass der Dekoder einen Fahrzeugwert nach dem anderen an die Datenbank zur Verfügung stellen kann. Wurden alle Fahrzeugdaten aus dem Paket ausgelesen und verarbeitet, wird ein neues Paket empfangen.

Tritt ein Neustart des Embedded-PCs mitten im Kommunikationsvorgang auf, führt das erneute Senden des Pakets `MSG_READY` an den virtuellen Server dazu, dass der aktuelle Kommunikationsvorgang abgebrochen und ein neuer vorbereitet wird – der Initialisierungsprozess beginnt von neuem. Auch das Senden des Pakets `MSG_READY` an die MicroAutoBox II dient dazu, einen neuen Kommunikationsvorgang zu starten, denn durch den Empfang des Pakets bricht auch die MicroAutoBox II den aktuell laufenden Kommunikationsvorgang ab und beginnt einen neuen Vorgang. Ein weiterer kritischer Punkt ist das nicht Empfangen der Fahrzeugdaten auf der Seite des Servers. Wenn das Programm nach einem von uns festgelegten Zeitpunkt (`timeout X`) keine Pakete empfangen hat, reagiert es durch ein erneutes Senden des Paketes (`ready_2`) an den Embedded-PC. Daraufhin sendet der PC nach Empfang des Paketes erneut das Paket (`ready_3`), was einen neuen Kommunikationsvorgang auslöst.

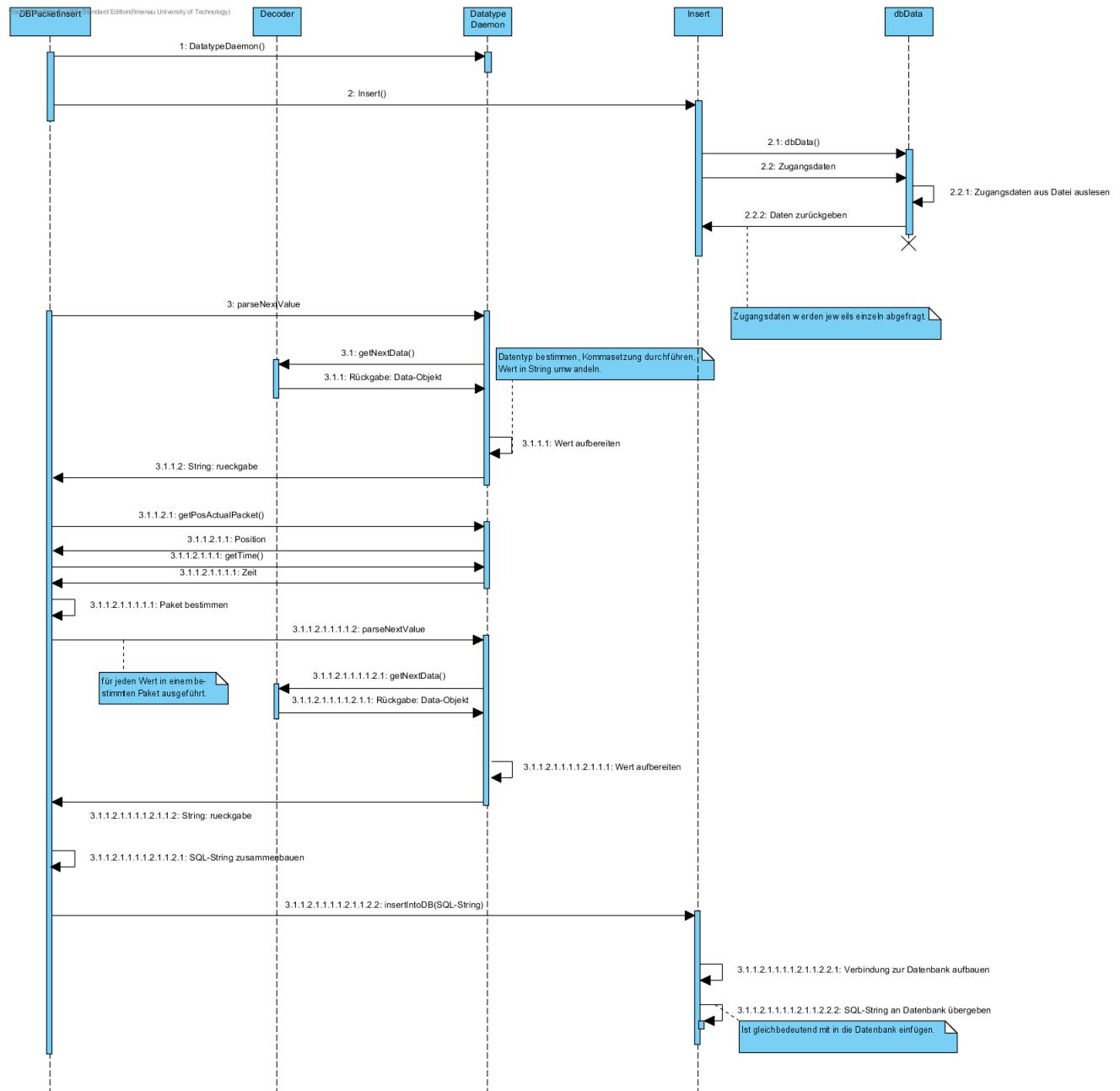
**Tabelle 2.5:** Übersicht über die Inhalte der übertragenen Pakete

Paketbezeichnung	Inhalt	Sender	Empfänger	Bemerkung
MSG_READY_1	Bitvektor: [01011001]	<i>embpc</i>	<i>vserver</i>	<ul style="list-style-type: none"> <li>• Kommunikationsbeginn melden</li> <li>• ggf. Mitteilung über neuen Kommunikationsbeginn</li> </ul>
MSG_READY_ACK_1	Bitvektor: [01010001]	<i>vserver</i>	<i>embpc</i>	Empfangsbestätigung für MSG_READY_1
MSG_READY_2	Bitvektor: [10010101]	<i>embpc</i>	<i>mabxii</i>	<ul style="list-style-type: none"> <li>• Bereitschaft von vServer und Emb-PC melden</li> <li>• ggf. Mitteilung über neuen Kommunikationsbeginn</li> </ul>
PACKET_INFO	Bytestrom: ([VEC_DATATYPES], [VEC_LAYOUT], [VEC_COMMA])	<i>mabxii</i> <i>embpc</i>	<i>embpc vserver</i>	Beinhaltet die für den Emb-PC und vServer notwendigen Informationen für die nachfolgende Verarbeitung der Fahrzeugdaten
PACKET_INFO_ACK_1	Bitvektor: [00010101]	<i>embpc</i>	<i>mabxii</i>	Empfangsbestätigung für PACKET_INFO

**Tabelle 2.6:** Übersicht über die Inhalte der übertragenen Pakete - Fortsetzung

Paketbezeichnung	Inhalt	Sender	Empfänger	Bemerkung
PACKET_INFO_ACK_2	Bitvektor: [01000101]	<i>embpc</i>	<i>vserver</i>	Empfangsbestätigung für PACKET_INFO
DATA	<ul style="list-style-type: none"> <li>• Beinhaltet sämtliche und aktualisierte Fahrzeuginformationen</li> <li>• Der Inhalt wird immer aktuell gehalten</li> </ul>	<i>mabxii</i>	<i>embpc</i>	Wird ohne Abfrage des Empfangs kontinuierlich gesendet
PACKAGE	<ul style="list-style-type: none"> <li>• von <i>embpc</i> in Pakete aufgeteilte aktuelle Fahrzeuginformationen, versehen mit jeweiligen Zeitstempel und ID</li> <li>• ([TIMESTAMP], [ID], [DATA_N])</li> </ul>	<i>embpc</i>	<i>vserver</i>	<ul style="list-style-type: none"> <li>• Standardmäßig werden die Fahrzeugdaten in 5 unterschiedliche Pakete aufgeteilt</li> <li>• Der sich ändernde Informationsinhalt der einzelnen Pakete wird durch den Datenempfang von der MABX II auf dem Emb-PC ständig aktualisiert.</li> </ul>

## 2.2.3 Einfügen in die Datenbank



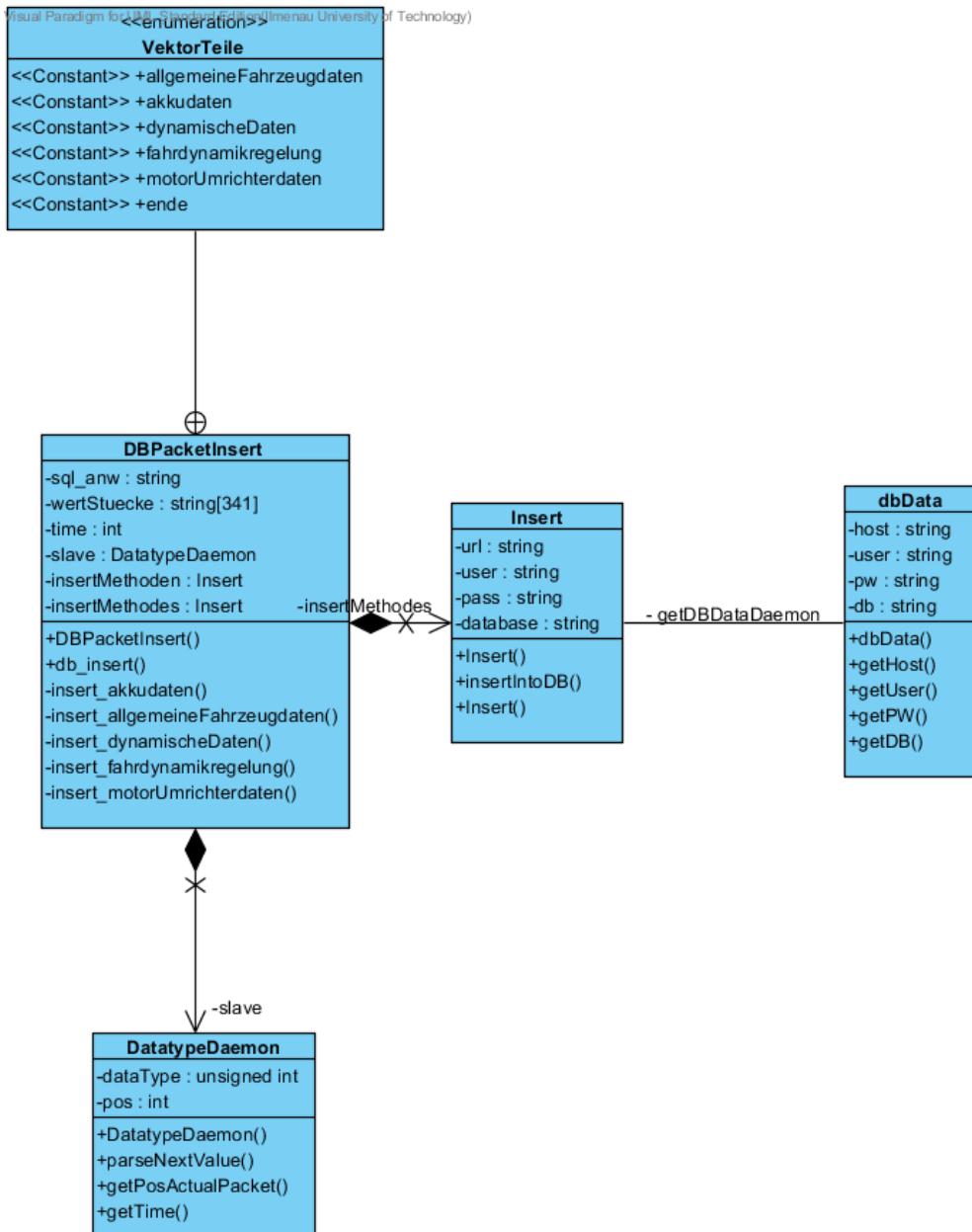
**Abbildung 2.29:** Sequenzdiagramm zum Einfügen in die Datenbank

Dieses Sequenzdiagramm zeigt den typischen Ablauf vom Start des Programms bis zum Ende einer Einfügeoperation in die Datenbank.

Es wird dabei davon ausgegangen, dass die Klasse Decoder bereits existiert.

Zuerst wird ein Objekt von *DBPacketInsert* erzeugt, das wiederum zuerst jeweils ein Objekt der Klasse *DatatypeDaemon* und *Insert* erzeugt. *Insert* erzeugt daraufhin ein Objekt der Klasse *dbData*, welches ihm die Zugangsdaten zur Datenbank zurückgibt. Das Objekt *dbData* wird daraufhin gelöscht. Danach wird bei *DBPacketInsert* die Memberfunktion *db\_insert()* aufgerufen. Leider war zu diesem Zeitpunkt wie in der Einleitung noch nicht Softwareprojekt TU Ilmenau SS 2013

klar, von wem diese Funktion aufgerufen wird, daher findet sie sich hier nicht wieder. Sie löst aber Schritt drei im Diagramm aus, wodurch ein einzelner Wert durch *DatatypeDaemon* geholt wird, den dieser wiederum eingebettet in ein Data-Objekt vom *Decoder* bekommt. Dieser Wert ist immer der Anfang eines Übertragungspakets von Embedded PC zum vServer. Zunächst wird das Paket durch die Position des einzelnen Wertes bestimmt, danach wird *DatatypeDaemon* so oft nach neuen Werten angefragt, bis ein Paket komplett abgehandelt ist. Danach wird der SQL-String aus diesen abgefragten Einzelwerten in einer Unterfunktion zusammengesetzt und anschließend an *insertIntoDB(SQL-String)* des Objekts der Klasse *Insert* übergeben. Dieses baut zuerst eine Verbindung zur Datenbank auf und übergibt dann den SQL-String der Datenbank bzw. dem MySQL-Connector/C++.



**Abbildung 2.30:** Klassendiagramm des virtuellen Servers

#### Klasse *DBPacketInsert*:

Von dieser Klasse soll, wenn das Programm vollständig zusammengebaut ist, einmal ein Objekt erzeugt werden, auf dem dann jeweils nach Erhalt eines Paketes vom Embedded PC die Memberfunktion *db\_insert()* aufgerufen wird. Diese löst dann das Auslesen und Aufbereiten der einzelnen Daten in der Klasse *DatatypeDaemon* aus, setzt die Werte zusammen zu einem SQL-String und übergibt diesen an die Klasse *Insert*. Im Weiteren wird auf diese einzelnen Schritte näher eingegangen.

Wie oben bereits geschildert, fragt *DBPacketInsert* zuerst nur einen Wert von *DatatypeDaemon* an und fügt diesen in ein Stringarray ein. Zu diesem Wert lässt es sich außerdem die Stelle im Übertragungspaket der MikroAutoboxII zum Embedded PC übergeben. Dies geschieht zum einen durch den Aufruf von *parseNextValue()* auf *DatatypeDaemon*, zum

anderen durch den Aufruf von *getPosActualPacket()* auf derselben Klasse. Danach fragt *DBPacketInsert* über *DatatypeDaemon* die Stelle des Wertes im Übertragungspaket von MikroAutoboxII zum Embedded PC ab, woraus er dann das Übertragungspaket von Embedded PC zum vServer ermittelt. Ein solches Paket beinhaltet alle Daten einer Tabelle auf der Datenbank, er nutzt dazu auch die in einer Enumeration definierten Startpunkte der einzelnen Pakete, die den Stellen im Übertragungspaket der MikroAutoboxII zum Embedded PC entsprechen. Diese Information benötigt die Klasse, um jetzt nur so viele Werte über *DatatypeDaemon* in dem Array, welches schon den einen oben genannten Wert enthält, zu sammeln, wie in einem Übertragungspaket sind. Damit füllt der gesamte Inhalt des Arrays eine Tabelle der Datenbank aus.

Für jede Tabelle gibt es schließlich eine Unterfunktion, die entsprechend dem Aufbau der Datenbank die einzelnen Werte des Arrays zusammensetzt und daraus einen gültigen SQL-String baut. Sobald die Unterfunktion abgeschlossen ist, wird der fertige SQL-String an die Memberfunktion *insertIntoDB(string\* anw)* der Klasse *Insert* übergeben.

#### Klasse *DatatypeDaemon*:

Diese Klasse stellt die Schnittstelle zum Decoder dar, indem sie dort *Data*-Objekte anfragt. Aus diesen Objekten extrahiert sie dann jeweils den Datenwert als *Unsigned Integer*, die Position dieses Wertes in dem Übertragungspaket der MikroAutoBoxII an den Embedded PC, den Datentyp vor der Übertragung und den Kommasetzwert.

Einerseits stellt *DatatypeDaemon* der Klasse *DBPacketInsert* die Zeit, als auch die Position des aktuellen Wertes bereit, andererseits stellt sie auch den Zeitstempel des Pakets, den die Klasse auch vom *Decoder* erhält, zur Verfügung. Die Position ist dabei über die Memberfunktion *getPosActualPacket()* und der Zeitstempel über *getTime()* abzufragen. Der Wert hingegen wird von der Klasse zuerst auf seinen Datentyp geprüft, dann gegebenenfalls durch die vom Kommasetzwert vorgegebene Zehnerpotenz geteilt und in den jeweiligen Datentyp verwandelt. Danach wird der Wert unter Verwendung der von der *Boost-Library* angebotenen Funktion *lexical\_cast* in einen *String* konvertiert. Dies geschieht bei Aufruf der Memberfunktion *parseNextValue()*, der aufbereitete und in einen *String* verwandelte Wert bildet dabei den Rückgabewert an den Aufrufer.

#### Klasse *Insert*:

Diese Klasse übernimmt die Kommunikation mit der Datenbank. Dazu erstellt sie erst ein Objekt der Klasse *dbData*, das über die Memberfunktionen *getHost()*, *getUser()*, *getPW()* und *getDB()* jeweils die Adresse, den Benutzer, das Passwort und den Namen der Datenbank zurückgibt.

Wird von *Insert* die Memberfunktion *insertIntoDB(string\* anw)* aufgerufen, so wird zunächst unter Zuhilfenahme des *MySQL-Connector/C++* eine Verbindung mit der Datenbank aufgebaut und an den *Connector* der *String anw* übergeben, der zu diesem Zeitpunkt eine vollständige SQL-Anweisung darstellt. Durch die Übergabe des Strings erfolgt dann das Einfügen der Daten durch den *Connector*.

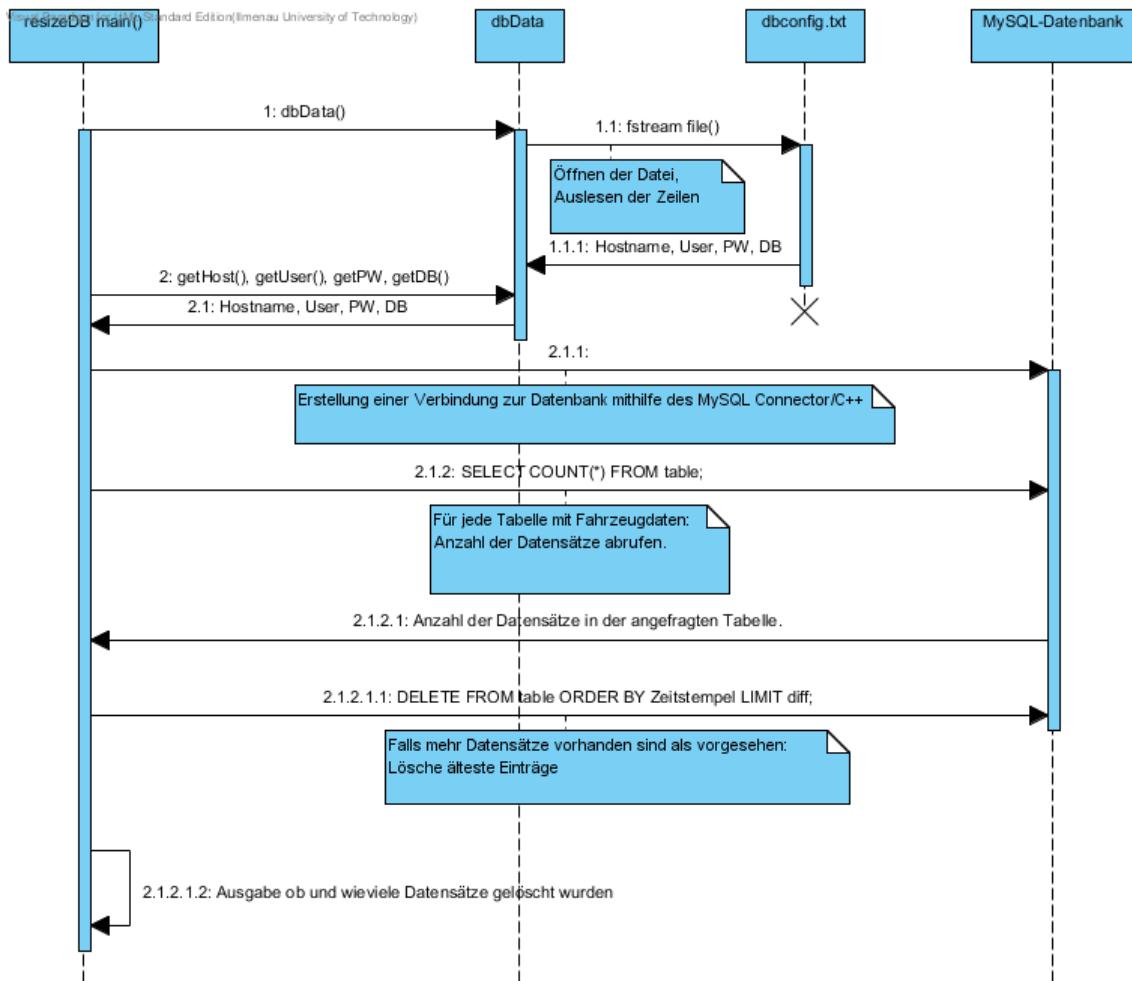


Abbildung 2.31: Sequenzdiagramm des virtuellen Servers

## 2.3 vServer

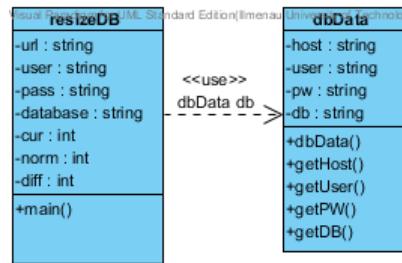
Dieses Sequenzdiagramm beschreibt den Ablauf des Programmes *sizeDB* auf dem virtuellen Server.

Direkt nach dem Aufruf des Programmes wird die Klasse *dbData* erzeugt, welche wiederum die Datei *dbconfig* einliest.

Anschließend fragt *resizeDB* von *dbData* die Zugangsdaten zur Datenbank ab und mit Hilfe dieser Daten wird nun eine Verbindung zum MySQL-Server aufgebaut.

Nachfolgend wird für jede Tabelle die aktuelle Anzahl an Datensätzen abgefragt, sollten mehr Einträge als vorgesehen in der Tabelle liegen, werden sie anschließend über ein weiteres MySQL-Statement aus der Datenbank gelöscht.

Da es auch möglich sein soll, dieses Programm von der Kommandozeile auszuführen, erfolgt zum Schluss eine Ausgabe über die durchgeföhrten Arbeitsschritte.



**Abbildung 2.32:** Klassendiagramm des virtuellen Servers

#### Klasse `resizeDB`:

`resizeDB` stellt ein eigenständiges Programm dar, welches die Fahrzeugdaten-Tabellen der MySQL-Datenbank auf die Anzahl ihrer Einträge überprüft, bei Überschreiten einer vordefinierten Anzahl an Einträgen (regulär 36000) werden die überflüssigen ältesten Einträge automatisch gelöscht.

Dieses Programm wird im Produktiveinsatz periodisch mittels eines vorher definierten *Cronjobs* (also eine periodisch ausgeführte Aufgabe zur Pflege der Daten) aufgerufen und ist unabhängig von den anderen auf dem virtuellen Server laufenden Programmen.

Beim Start werden über die Klasse `dbData` die Zugangsdaten für den MySQL-Server abgerufen und anschließend eine Verbindung zur Datenbank mittels des *MySQL Connector/C++* aufgebaut. War der Verbindungsaufbau erfolgreich, werden die entsprechenden Tabellen mit den Fahrzeugdaten nacheinander überprüft, dazu wird die aktuelle Anzahl an Datensätzen in der Tabelle über ein MySQL-Statement ermittelt und mit der Norm verglichen. Sollten mehr Einträge als vorgesehen in einer Tabelle vorhanden sein, wird diese mit einem weiterem MySQL-Statement nach ihrem Zeitstempel sortiert und die ältesten Einträge werden entfernt. Die Anzahl der gelöschten Einträge richtet sich hierbei nach der Differenz der Anzahl der aktuellen Einträge und dem Referenzwert.

Da dieses Programm auch manuell von der Kommandozeile ausführbar sein soll, etwa um zu Überprüfen ob die Verbindung mit der MySQL-Datenbank erfolgreich ist, erfolgen Ausgaben über den aktuellen Bearbeitungsstand des Programmes.

#### Klasse `dbData`:

Die Klasse `dbData` wird verwendet um die Zugangsdaten für die MySQL-Datenbank aus einer einfachen Textdatei (siehe `dbconfig.txt`) auszulesen.

Um die Werte zu speichern werden die *String*-Variablen `host`, `user`, `pw` und `db` angelegt. Der Konstruktor öffnet bei der Erzeugung eines Objektes dieser Klasse die Datei `dbconfig.txt` aus dem gleichen Verzeichnis und schreibt die Werte der einzelnen Zeilen in die zugehörigen Variablen. Der Zugriff erfolgt mithilfe der vier Memberfunktionen `getHost()`, `getUser()`, `getPW()` und `getDB()`, welche als Rückgabewert den entsprechenden *String* liefern.

## 2.4 Webseite

### 2.4.1 Aufbau der Webseite

Die Webseite setzt sich aus einer Vielzahl von einzelnen PHP- und Java-Skripten zusammen, die gemeinsam die geforderte Funktionalität erfüllen. Um dem Leser einen einigermaßen geordneten Überblick über diese zu verschaffen, möge er das folgende Diagramm näher betrachten. Hier sind alle Skripte und ihre gegenseitigen Zusammenhänge zu finden und stellen den genauen Aufbau der kompletten Webseite dar.

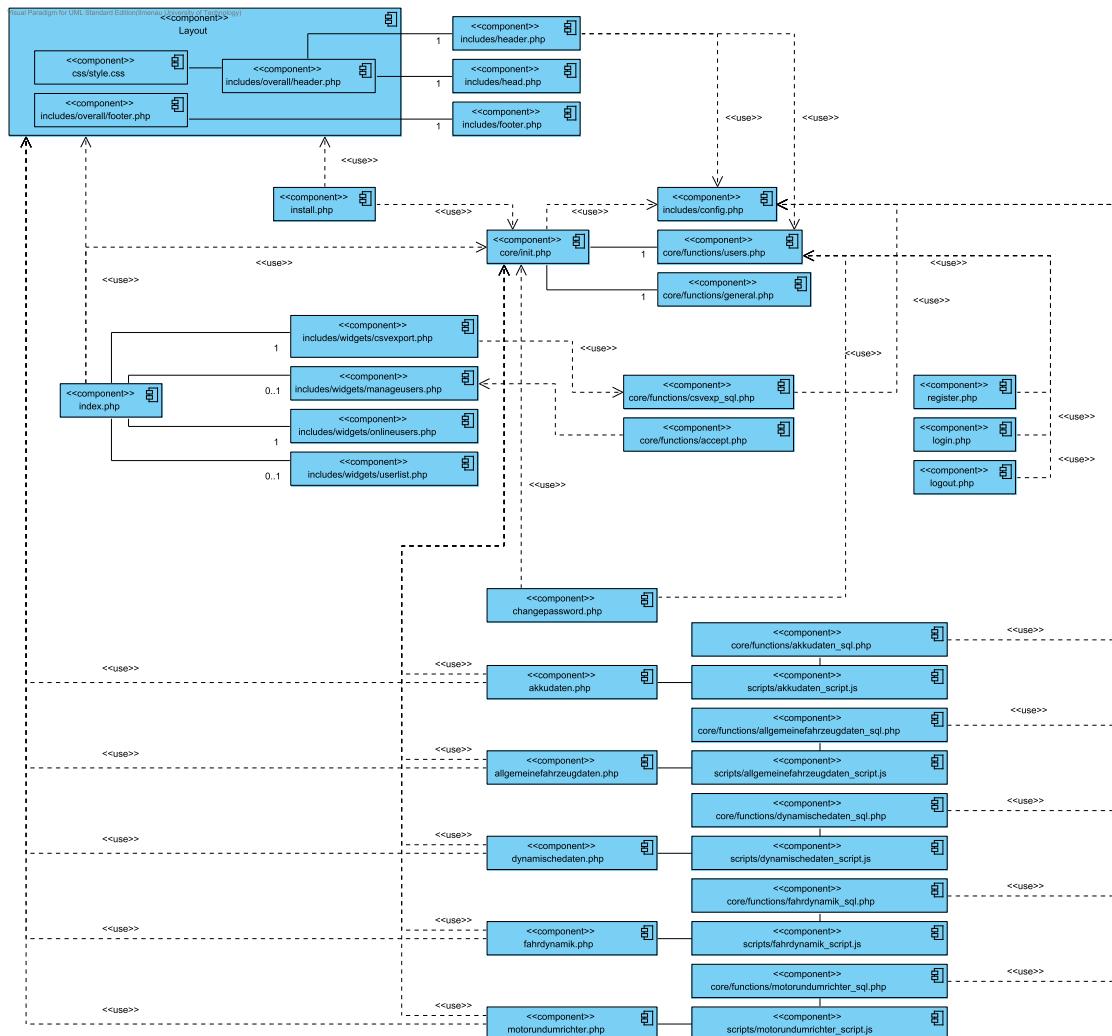


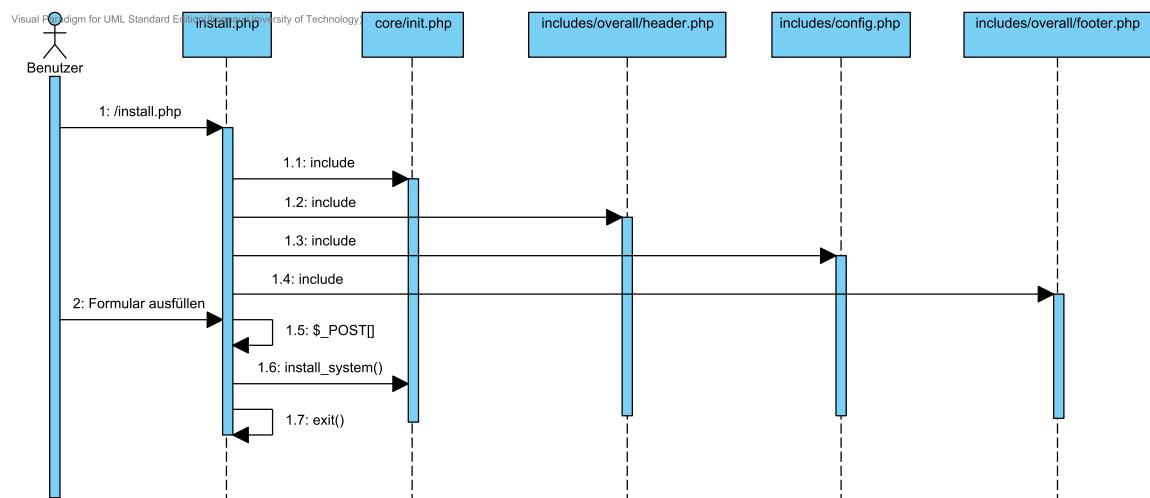
Abbildung 2.33: Komponentendiagramm der Webseite

### 2.4.2 Installation

Die Installation des Systems wird einmalig und nur beim ersten Besuch der Seite vom Nutzer durchgeführt. Sie wird durch Aufruf der `install.php` gestartet. Hierbei bindet das Skript wie auch alle folgenden selbstständigen Seiten die Dateien `core/init.php`, `includes/overall/header.php`, `includes/config.php` und `includes/overall/footer.php` ein. In diesen

Dateien befinden sich neben „globalen“ (es handelt sich hierbei um keine wirklichen PHP-globalen Variablen, sondern lediglich solche, die in allen Skripten verfügbar sind, die *core/init.php* bzw. *includes/config.php* einbinden) Variablendefinitionen und auch selbstdefinierte Funktionen. An dieser Stelle ist insbesondere die Funktion *install\_system()* aus der Datei *core/functions/general.php* zu erwähnen (siehe Eintrag zu *general.php*).

Der Nutzer wird nach dem Durchlauf des Skripts aufgefordert ein Formular auszufüllen. Mit diesen Daten wird dann der Vorstands-Account (= Admin-Account) erzeugt. Anschließend werden alle benötigten Datenbanken, Tabellen und Einträge erzeugt und das Skript anschließend mittels *exit()* terminiert.



**Abbildung 2.34:** Sequenzdiagramm für den Installationsprozess

### 2.4.3 Initialisierung der Seiten

Zu Beginn jeder Seite wird die Datei `core/init.php` eingebunden. Diese sorgt dafür, dass zu Beginn der einzelnen Skripte alle Variablen und Funktionen definiert sind sowie eine neue Sitzung (Session) gestartet wird. Dies wird erreicht, indem die Dateien `include/config.php`, `core/functions/general.php` und `core/functions/users.php` eingebunden werden. Zudem wird für einige Funktionen bereits eine Verbindung zur Datenbank aufgebaut.

Anschließend wird die Funktion `logged_in()` ausgeführt, welche überprüft, ob der Nutzer sich bereits eingeloggt hat. Ist dies nicht der Fall, wird lediglich ein „globales“ Array `$errors[]` erzeugt, indem an vielen Stellen in unterschiedlichen Skripten Fehlermeldungen gesammelt werden können. Hat sich der Benutzer jedoch bereits eingeloggt, wird noch geprüft, ob der Nutzer bereits aktiviert wurde. Falls nicht, wird die aktuelle Sitzung zerstört, der Benutzer auf die Startseite mittels `header()`-Redirect umgeleitet und das Skript beendet. Ist der Nutzer hingegen bereits aktiviert worden, wird ebenfalls das „globale“ Fehlerarray `$errors[]` angelegt und das Skript beendet.

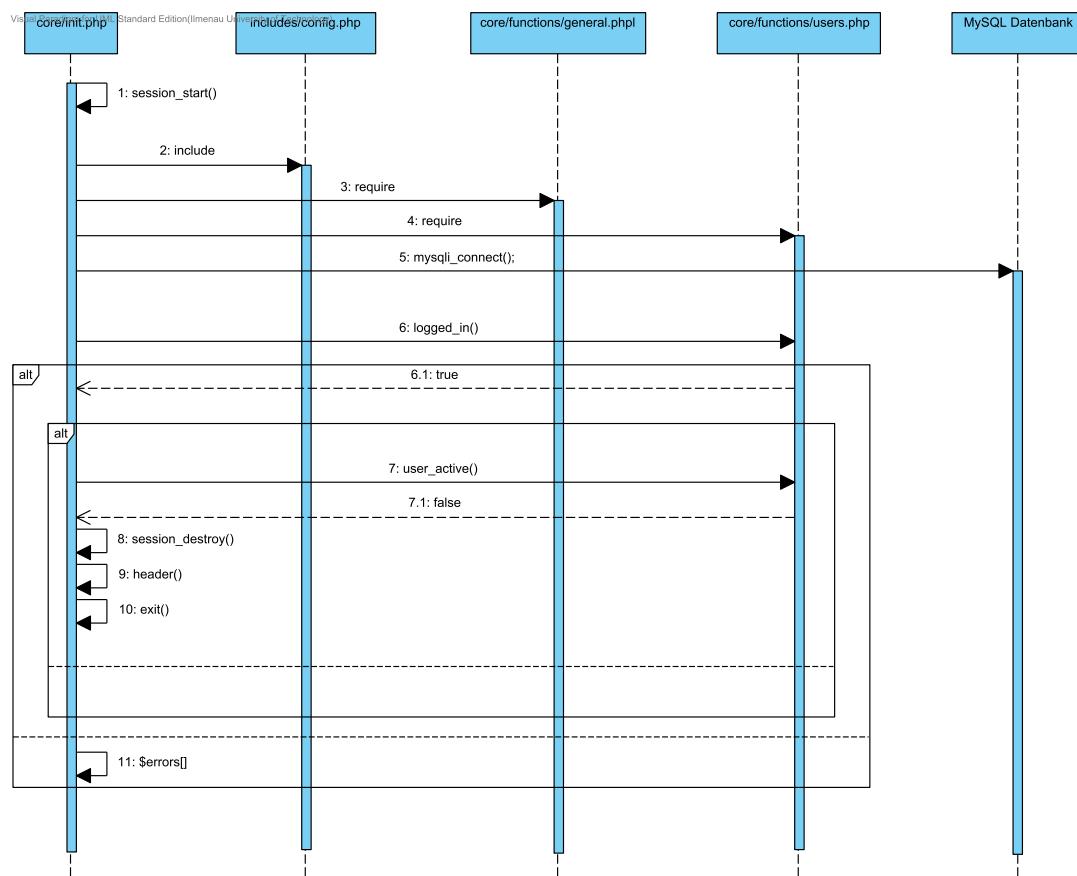


Abbildung 2.35: Sequenzdiagramm für die Initialisierung

## 2.4.4 Header

Die Datei *includes/header.php* wird von der *includes/overall/header.php* eingebunden. Sie setzt vor allen Dingen das Menü um, mittels welchem der Nutzer auf der Seite navigieren kann, insofern er denn eingeloggt ist. Dabei prüft *header.php* zuerst mittels der Funktion *logged\_in()* ob der Nutzer eingeloggt ist. Ist dem so, so zeigt er sowohl Menü als auch das Widget *includes/widgets/loggedin.php* an, welches den Nutzer mit seinem Namen begrüßt, ihm die Möglichkeit bietet sein Passwort zu ändern oder die Startseite aufzurufen. Außerdem wird auf den einzelnen Seiten die über das Menü aufgerufen werden können, jeweils eine Information eingeblendet, ob die Daten bereits „veraltet“ sind. Des Weiteren wird bei Aufruf dieser *header.php* Datei immer der Eintrag in der Online-Nutzer-Tabelle aktualisiert, indem die Funktion *refresh()* ausgeführt wird.

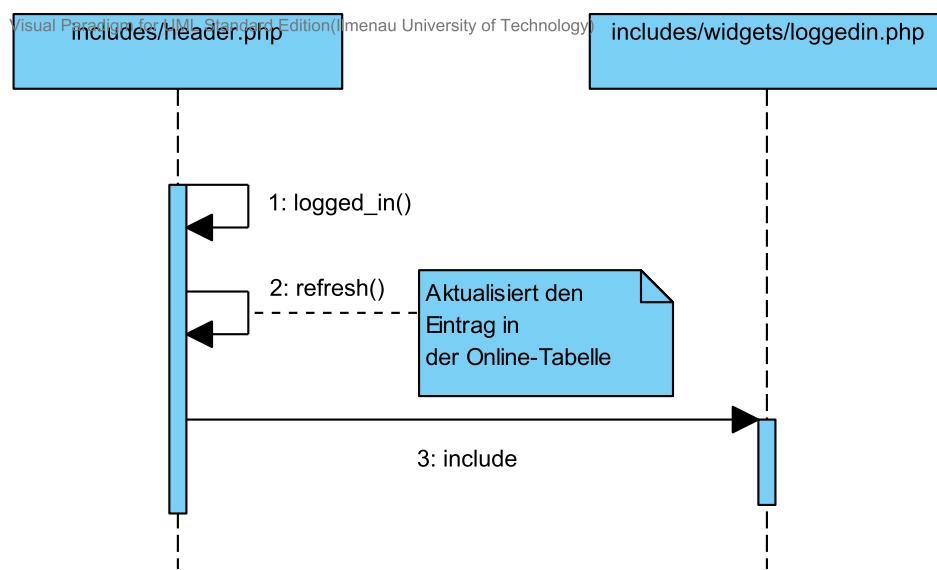


Abbildung 2.36: Sequenzdiagramm der Headerdatei

## 2.4.5 Startseite

Die Startseite ist wie der Name bereits verrät der Startpunkt jedes Aufrufes des Service Interfaces. Nachdem der Benutzer die Seite aufruft werden vom *index.php*-Skript alle nötigen Dateien eingebunden. Die logischen und funktionalen Aspekte werden durch Einbinden der *core/init.php* Datei erreicht. Für das Layout und die HTML-Komponenten wird *includes/overall/header.php* eingebunden (siehe 2.4.13 Allgemeiner Seitenaufbau) wo u. a. auch geprüft wird, ob sich der Nutzer eingeloggt hat oder nicht und ggf. auf die Startseite weitergeleitet wird.

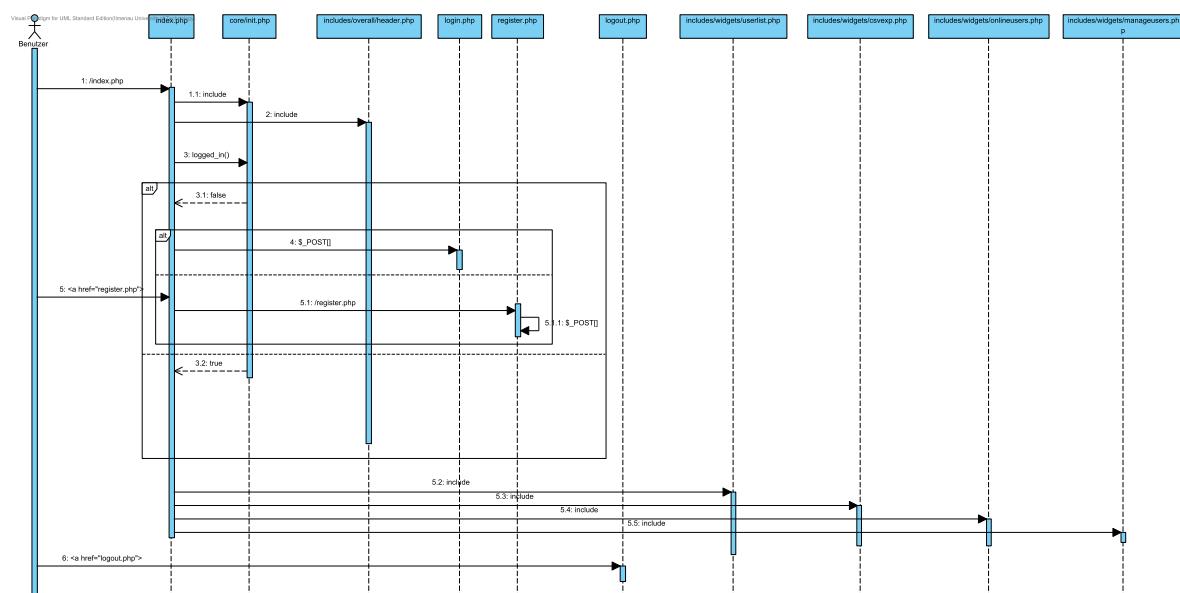
Im Anschluss wird überprüft ob sich der Benutzer bereits eingeloggt hat oder nicht. Für den Fall, dass er noch nicht eingeloggt ist, erscheint ein Formular in welches er E-Mail-Adresse und Passwort eingeben kann. Diese Daten werden anschließend an das Skript *login.php* mittels *POST*-Request verschickt und anschließend validiert sowie verifiziert (siehe 2.4.7). Alternativ kann sich der Benutzer auch einen neuen Account erstellen. Dies erreicht er durch Auswählen des „Jetzt registrieren“-Links, wodurch er auf die Skript-Seite *register.php* weitergeleitet wird. Der Nutzer erhält erneut ein Formular, welches er für die Registrierung

vollständig ausfüllen muss. Diese Daten werden anschließend nach dem Klick auf den Abschicken-Button validiert (siehe 2.4.6).

Hat sich der Nutzer eingeloggt oder war bereits eingeloggt als er die Seite aufrief, werden ihm anstelle des Login-Formulars einige Widgets in Abhängigkeit seiner Rechtegruppe (*isVorstand()*-Aufruf) angezeigt, die da wären

- Möglichkeit zum Export der in der Datenbank befindlichen Fahrzeugdaten (jeder Nutzer; *includes/widgets/csvexport.php*)
- Liste der sich aktuell online befindlichen Nutzer (jeder Nutzer; *includes/widgets/onlineusers.php*)
- Möglichkeit der Verwaltung der Nutzer  
(nur Vorstand; *includes/widgets/manageusers.php*)
- Liste der im System registrierten Benutzer (nur Vorstand; *includes/widgets/userlist.php*)

Des Weiteren wird ihm nach dem Login auch ein horizontales Menü eingeblendet, über welches der Nutzer die Möglichkeit erhält die verschiedenen Unterseiten aufzurufen und sich ggf. wieder auszuloggen.



**Abbildung 2.37:** Sequenzdiagramm der Startseite

## 2.4.6 Registrierung

Der Nutzer erreicht über einen Link auf der Startseite das Registrierungsformular. Dieses bindet zu Beginn die gewohnten `includes/overall/header.php` und die `core/init.php` ein. Anschließend wird geprüft, ob der Nutzer sich bereits eingeloggt hat. Falls ja, so wird er auf die Startseite umgeleitet, andernfalls wird ihm das Registrierungsformular angezeigt. Dieses muss er vollständig ausfüllen, damit seine Registrierungsanfrage verarbeitet wird. Die eingegebenen Daten werden mittels POST-Request an sich selbst geschickt, validiert und anschließend ein neuer Eintrag in der Benutzertabelle angelegt. Dabei wird der Status des Nutzer standardmäßig auf `0`, d.h. *inaktiv*, gesetzt. Anschließend wird der Nutzer mittels `header()`-Umleitung erneut auf die Registrierungsseite umgeleitet und das vorhergehende Skript durch Aufruf der `exit()`-Funktion beendet.

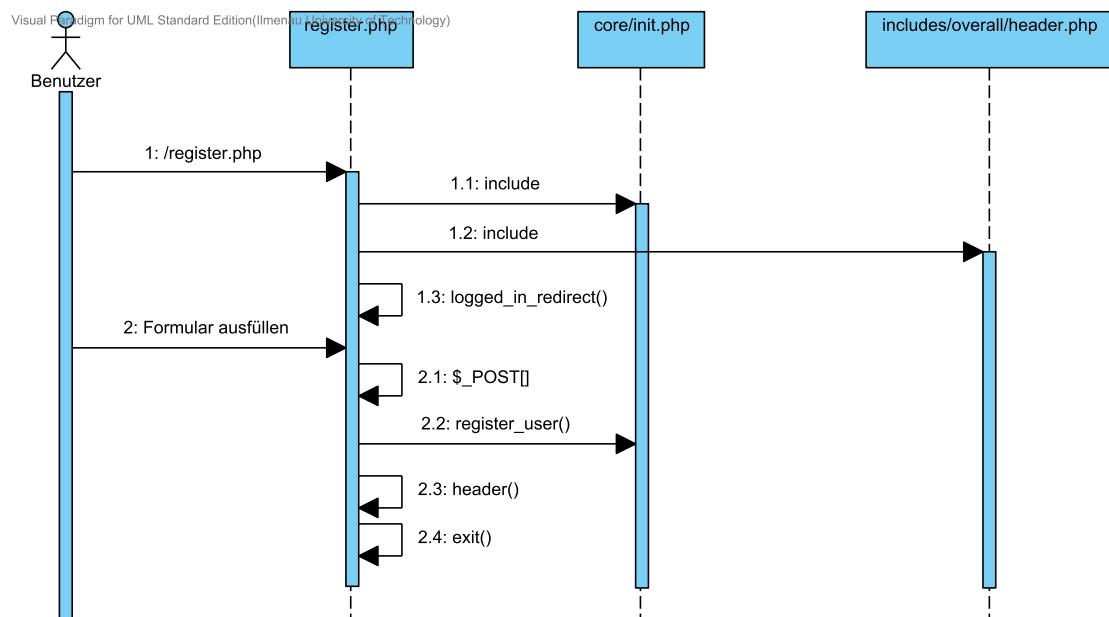


Abbildung 2.38: Sequenzdiagramm des Registriervorgangs

## 2.4.7 Loginvorgang

Nach der Eingabe der Login-Daten wird der Nutzer auf die Seite *login.php* weitergeleitet. Das hier ausgeführte Skript bindet wie gewohnt zuerst die allgemeine Initialisierungsdatei *core/init.php* ein, sowie die für das Layout benötigten *includes/overall/header.php* respektive *includes/overall/footer.php* und überprüft, ob der Nutzer bereits eingeloggt ist. Ist dies der Fall, so wird das Skript beendet und er wird automatisch auf die Startseite weitergeleitet.

Ansonsten führt das Skript anhand der als *POST*-Request übermittelten Daten eine Anfrage aus und verifiziert die Login-Daten. Bei erfolgreicher Verifizierung werden die Sessionvariablen *\$\_SESSION['user\_id']* und *\$\_SESSION['time']* (aktueller UNIX-Zeitstempel) gesetzt und anschließend eine Verbindung zur Datenbank erzeugt. Nach der Auswahl der benötigten Datenbank wird ein neuer Eintrag in der Online-Tabelle erzeugt, der sich aus den beiden Session-Variablen zusammensetzt. Anschließend wird der Nutzer auf die Startseite weitergeleitet und die Ausführung des Skripts mittels *exit()* beendet.

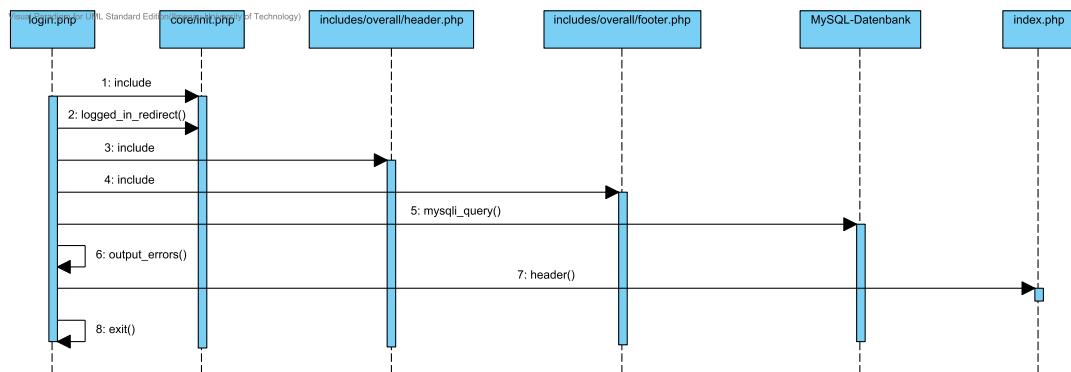


Abbildung 2.39: Sequenzdiagramm des Loginvorgangs

## 2.4.8 Logoutvorgang

Nach dem Aufruf der *logout.php* Datei wird das *core/init.php* Skript eingebunden, woraufhin es die benötigte Datenbank auswählt und anschließend eine SQL-Anfrage an die MySQL-Datenbank sendet, die den „Online“-Eintrag des aktiven Nutzers löscht. Abschließend wird die aktuelle Sitzung mittels *session\_destroy()* beendet, der Nutzer auf die *index.php* mittels *header()*-Weiterleitung weitergeleitet und das Script beendet.

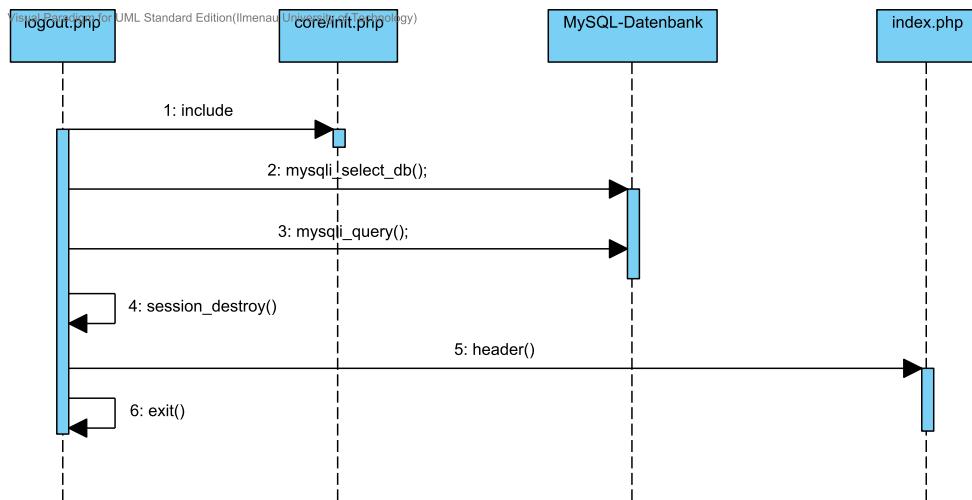


Abbildung 2.40: Sequenzdiagramm des Logoutvorgangs

## 2.4.9 Änderung des persönlichen Passworts

Das Skript *changepassword.php* bindet nach dessen Aufruf zuerst die allgemeine *core/init.php* Datei sowie die für das Layout verantwortliche *includes/overall/header.php* und *include/overall/footer.php* ein und überprüft das Skript ob die aufrufende Person bereits eingeloggt ist.

Der Benutzer kann durch Ausfüllen des ihm an dieser Stelle zur Verfügung gestellten Formulars durch Eingabe seines bisherigen Passworts und seines neuen Passworts (inkl. Wiederholung) ändern. Nach dem Abschicken des Formulars werden die Daten validiert und nach erfolgreicher Validierung durch Aufruf der *change\_password()*-Funktion persistent im System gespeichert. Anschließend wird der Benutzer zurück mittels *header()*-Umleitung auf die *changepassword.php*-Seite geführt.

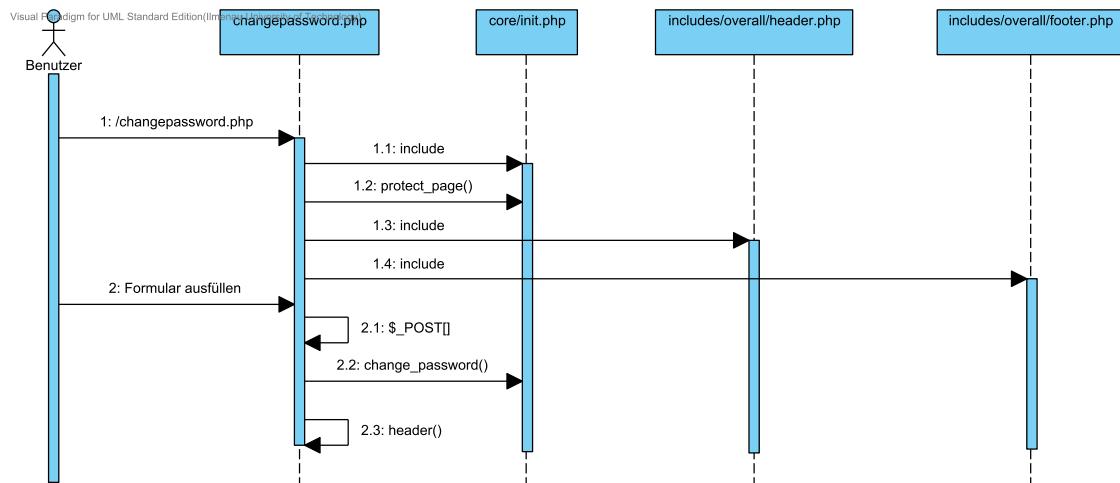


Abbildung 2.41: Sequenzdiagramm der Passwortänderungsfunktion

## 2.4.10 CSV-Exportfunktionalität

Diese Funktion wird ebenfalls als Widget für die Startseite angeboten, wobei eine Liste von Checkboxen zur Verfügung gestellt werden. Durch Auswahl dieser einzelnen Checkboxen können die zu exportierenden Tabellen ausgewählt werden. Die Auswahl wird hierbei mittels POST-Request an die Datei `core/functions/csvexp_sql.php` gesendet, welche dann eine Verbindung zum MySQL-Server aufbaut, die Datenbank auswählt und die gewünschte Abfrage ausführt. Anschließend wird die Verbindung geschlossen und dem Benutzer öffnet sich ein Fenster, dass ihm zum Download der nun erstellen CSV-Datei auffordert.

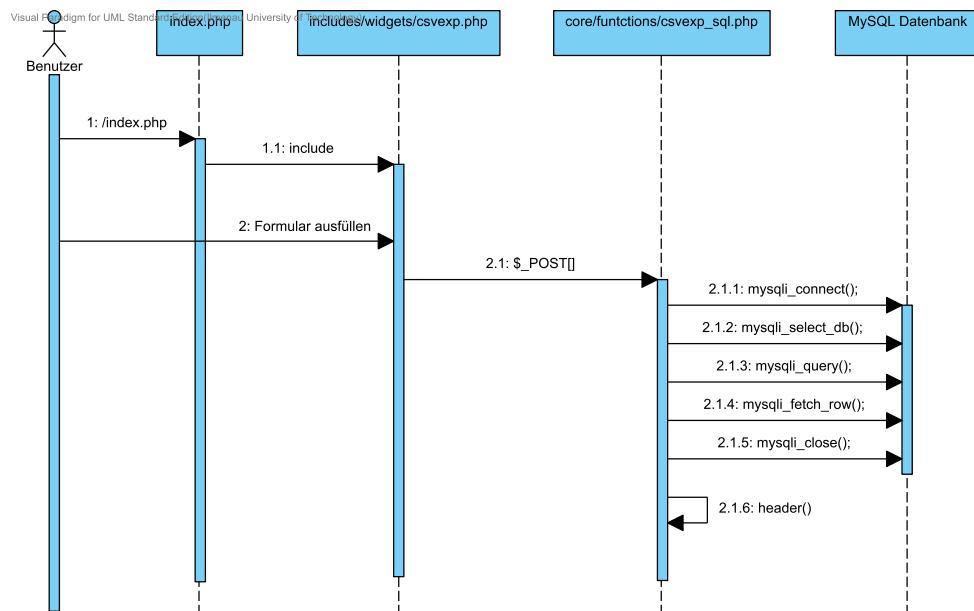


Abbildung 2.42: Sequenzdiagramm des CSV-Export

## 2.4.11 Nutzerverwaltung

Hierbei handelt es sich um genau zu sein um zwei Widgets (*includes/widgets/manageusers.php* und *includes/widgets/userlist.php*), welche auf der Startseite für eingeloggte Benutzer eingebunden werden. Dabei stellt *userlist.php* die im System bereits registrierten Nutzer dar. Hierfür baut das Skript eine Verbindung zur Datenbank auf, wählt die Tabelle der registrierten Nutzer aus (ID, Vorname, Nachname, E-Mail-Adresse, Rechtegruppe, Account-Status) und gibt diese als HTML-Tabelle aus. Die zweite Datei, *manageusers.php*, bietet dem Nutzer ein Formular an, das auf Eingabe einer Nutzer-ID (entnehmbar der Nutzerliste, s.o.), der Auswahl ob der Nutzer aktiviert oder gelöscht werden soll, durch Auswählen einer der beiden Radioboxen sowie der Wahl einer der drei voreingestellten Benutzergruppen mittels einer Dropdown-Box ein Skript (*core/functions/accept.php*) aufruft, welches die übergebenen Daten validiert und anschließend die Änderungen durch Aufbau einer Verbindung zur Datenbank in der Benutzertabelle persistent speichert. Am Ende des Skripts wird der Nutzer automatisch mittels *header()*-Weiterleitung zurück auf die Startseite befördert.

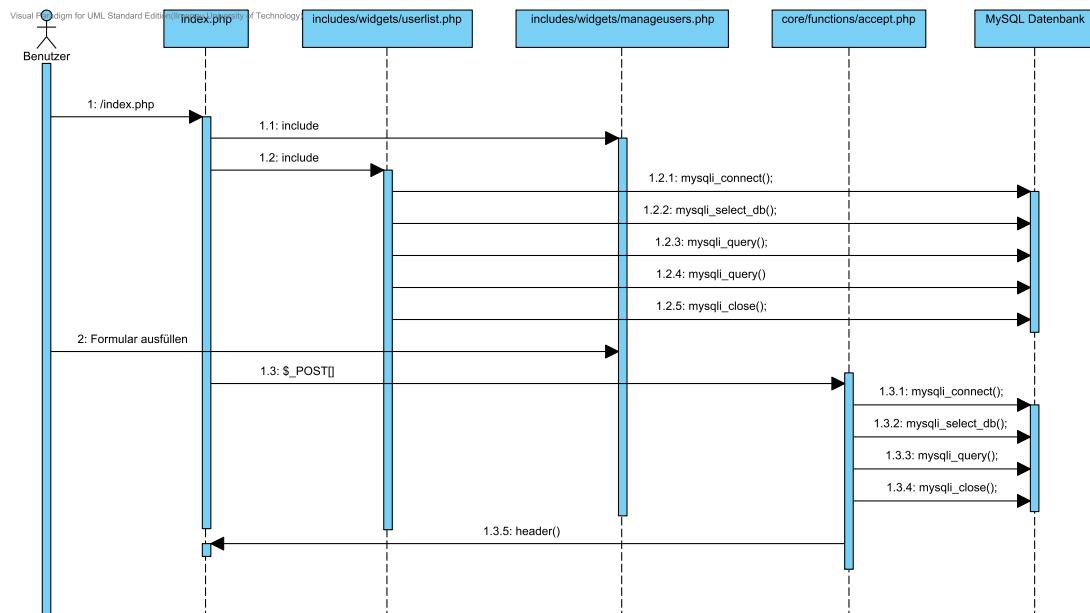


Abbildung 2.43: Sequenzdiagramm der Benutzerverwaltung

## 2.4.12 Online-Anzeige der Nutzer

Hierbei handelt es sich um ein kleines Widget, welches nicht mehr tut, als dass es eine festgelegte SQL-Anfrage an den MySQL-Server stellt. Dabei wird wie gewohnt zuerst die Verbindung eröffnet, die Datenbank ausgewählt und anschließend die Anfrage auf dieser ausgeführt. Das zurückgegebene Ergebnis wird anschließend als HTML-Tabelle ausgegeben und die Verbindung zum MySQL-Server wieder geschlossen.

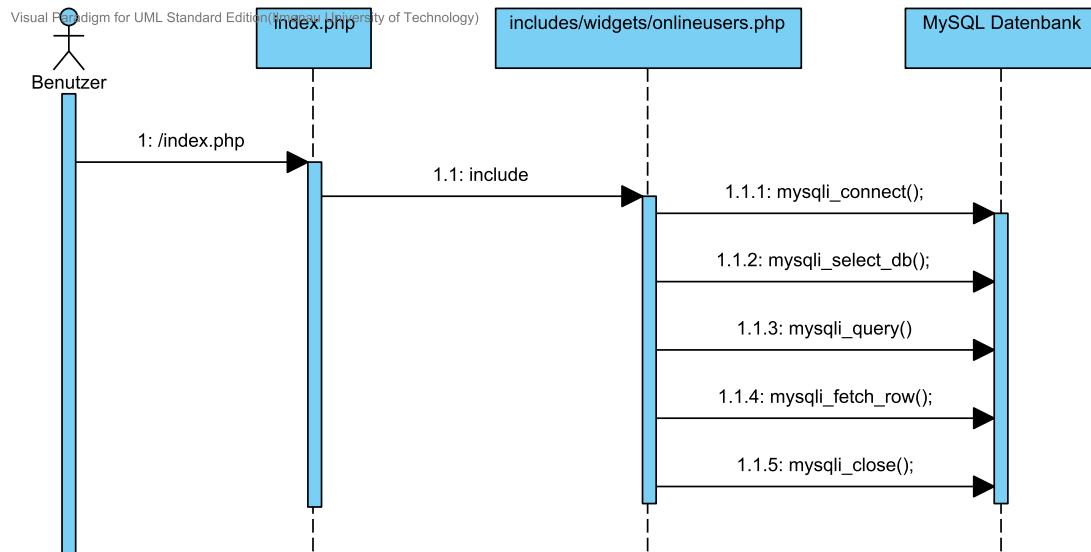
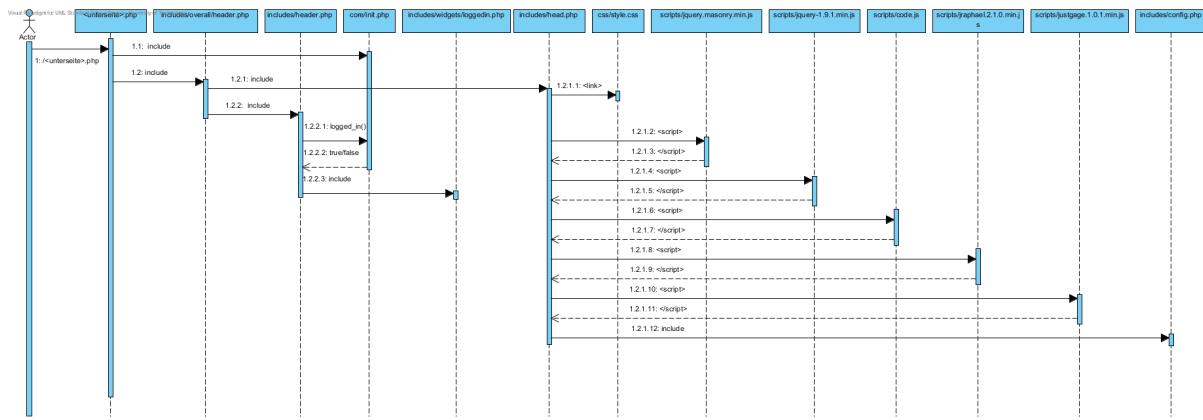


Abbildung 2.44: Sequenzdiagramm der Onlineanzeige

## 2.4.13 Allgemeine Seitenaufbau

Im allgemeinen Seitenaufbau geht es um eine Folge von Seitenaufrufen, Funktionen und Einbindevorgänge. Diese sind für die fünf darstellenden Seiten identisch und werden deshalb an dieser Stelle zusammenfassend veranschaulicht.



### includes/overall/header.php

Beschreibt den Aufbau des Dokuments. Dabei wird der Dokumenttyp bestimmt, danach der Head-, sowie der Navigationsbereich eingebunden.

### includes/head.php

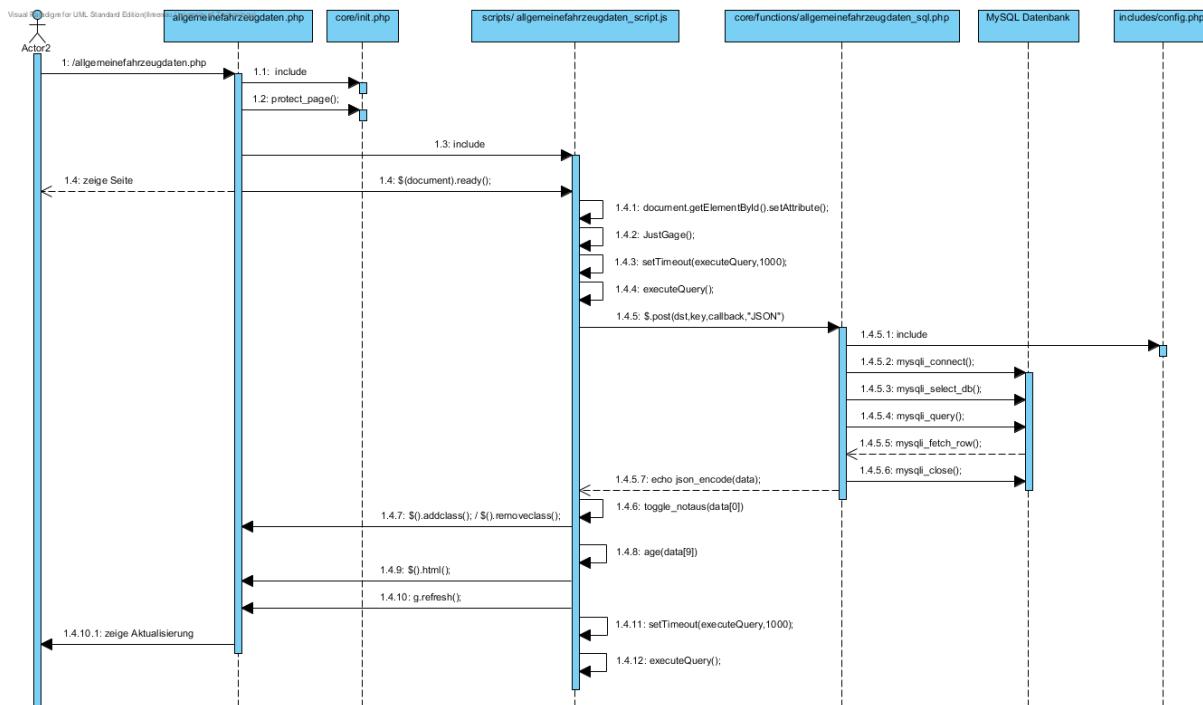
Diese Datei stellt den HTML-Headbereich einer Webseite dar. Hier werden alle allgemein nötigen Skripte, CSS-Dateien und Seiten importiert. Darunter sind die vier Javascript Bibliotheken: jQuery, justGage, raphael und masonry, die alle unter der MIT Lizenz frei verfügbar sind. Weiterhin wird die Datei config.php eingebunden, die relevante Informationen zum Zugriff auf die Datenbanken bereithält.

### includes/header.php

Beinhaltet die Navigationsleiste am Kopf der Seite, die auf allen Seiten präsent ist. An dieser Stelle wird auch geprüft, ob der Nutzer angemeldet ist. Die Navigationsleiste wird nur angemeldeten Nutzern angezeigt.

## 2.4.14 Allgemeine Fahrzeugdaten

Auf dieser Seite werden die im Pflichtenheft vereinbarten allgemeinen Daten dargestellt. Darunter fällt die Geschwindigkeit, die als Halbkreis-Diagramm mittels der JavaScripte justGage und raphael realisiert wird. Die Notausschalter werden als Tabelle dargestellt, wobei ein aktiver Schalter farblich in rot hinterlegt wird. Die Seite enthält 3 wesentliche Dateien: Die PHP-Seite, die den strukturellen Aufbau beinhaltet (3.4.4), das Javascript was in die Seite eingebunden und ausgeführt wird (3.5), sowie eine PHP-Seite, die die MySQL Anfrage für die Seite realisiert (3.4.17).



**protect\_page()** Diese Funktion aus der “init.php“ wird verwendet, um den unberechtigten Zugriff auf die Seite zu vermeiden und den Nutzer auf die Login-Seite weiterzuleiten.

### **\$(document).ready()**

Wenn die Seite komplett aufgebaut ist, wird dies Aktion ausgeführt und startet weitere Funktionen des JavaScripts.

### **document.getElementById().setAttribute**

Ein Aufruf, der bewirkt, dass alle Klassen der Navigationsbuttons verändert werden. Wird fünf mal ausgeführt, wobei alle nicht ausgewählten Schaltflächen grau werden und der ausgewählte Button färbt sich blau.

### **justGage()**

Initialisiert die halbrunde Geschwindigkeitsanzeige, die durch “**g.refresh()**“ aktualisiert wird.

### **setTimeout(executeQuery,1000)**

Dieses Timeout realisiert die Aktualisierung im Sekundentakt. Dabei sorgt der erste Aufruf dafür, dass “executeQuery()“ zum ersten mal aufgerufen wird und jeder weitere Aufruf (innerhalb von “executeQuery()“) sorgt dafür, dass die Funktion kontinuierlich wiederholt wird.

### **executeQuery()**

Aufgabe dieses Aufrufs ist Anforderung des JSON-Arrays, sowie das erneute Selbstaufrufen durch einen Timeout.

```
$.post('allgemeinefahrzeugdaten_sql.php', {permission:"true"}, function(data) {
    "JSON")
```

Dies Funktion ruft die an erster Stelle genannte Seite auf und erwartet als Rückgabe ein Array (data). JSON ist dabei die vereinbarte Notation (Java Script Object Notation). In diesem Array stehen dann der Reihe nach alle aktuellen Werte aus der Datenbank. Des weiteren wird ein array mit einem Wert zusätzlich an die Seite gesendet (Position= permission, Wert=true), die bewirkt, dass kein unbefugter durch direkte Eingabe der Seite den Datensatz bekommen kann.

### **allgemeinefahrzeugdaten\_sql.php**

Diese Seite realisiert die jeweilige Datenbankanfrage und wandelt das Ergebnis mit “json\_encode“ in ein Array um, dass dann an die aufrufende Seite zurückgegeben wird.

### **toggle\_notaus(), removeClass/addClass**

Der Status der Notausschalter wird als String übergeben. Er besteht aus einer Folge von 0 und 1, der aufgetrennt werden muss. Im Anschluss wird mit “(ID).removeClass“ und “(ID).addClass“ jeweils der Status der Schalter (AN/AUS) geändert, und die damit einhergehende Änderung der Hintergrundfarbe (rot für AN/ weiß für AUS).

### **\$(ID).html(val)**

Mit dieser Funktion wird der Wert “val“ an die Stelle im Dokument geschrieben, die die angegeben “ID“ hat.

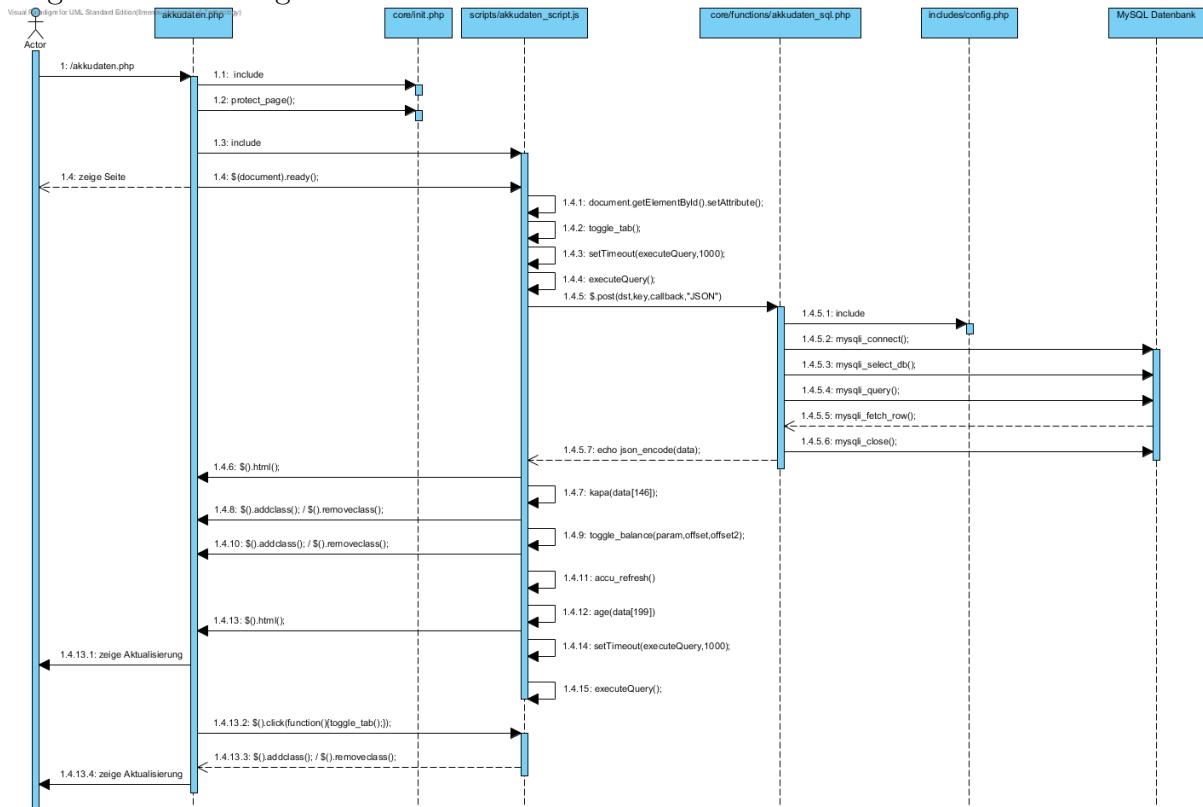
### **age()**

Die Funktion age() dient dazu, die Zeit anzuzeigen, die vergangen ist, seitdem die letzte Aktualisierung in der Datenbank verzeichnet wurde. Dabei wird die UNIX Zeit als Referenzzeit genutzt und auf Sekunden normiert. Der Zeitpunkt in der Tabelle nutzt ebenfalls Zeiteinheit Sekunden. Die Differenz beider Werte gibt die Zeit an, die seit dem letzten Datenbankeintrag vergangen ist. Danach wird diese Zahl in Tage, Stunden, Minuten und Sekunden umgerechnet. Es werden aber aus Platzgründen in der Anzeigefläche nur Stunden, Minuten und Sekunden angezeigt. Die Zeitangabe erscheint genau dann, wenn mehr als 15 Sekunden verstrichen sind und nur die Zeiteinheiten, die tatsächlich mit Werten belegt werden. Ist mehr als eine Stunden vergangen, werden Stunden, Minuten und Sekunden angezeigt. Ist weniger als eine Minute vergangen, werden nur die vergangen Sekunden angezeigt.

## **2.4.15 Akkudaten**

Auf dieser Seite werden alle Spannungen, Stromstärken und Temperaturen bezüglich der Akkumulatoren angezeigt. Dabei entsprechen die dargestellten Daten denen, die im Pflichtenheft vereinbart wurden. Hauptaugenmerk ist die Tabellendarstellung, bei

der bestimmte Tabellenspalten ein-, bzw. ausgeblendet werden können. Informationen zu den 12 Akkumulatorblöcken (jeweils 12 Zellen) sollen permanent angezeigt werden und Informationen zu den 12 Zellen eines Blocks nur auf Wunsch. Des weiteren erfolgt die Anzeige der Gesamtspannung über alle Akkumulatorzellen als Ladebalken im Querformat. Die Seite besteht wieder aus 2 PHP und einer JavaScript Komponente, analog wie 2.4.13 Allgemeine Fahrzeugdaten



Die allgemeine Funktionsweise dieser Seite entspricht 2.4.13 Allgemeine Fahrzeugdaten. Im folgenden wird nur auf Besonderheiten dieser Seite eingegangen.

### accu\_refresh(data)

Diese Funktion aktualisiert die 144 Zellspannungen und berechnet für die zwölf Akkublöcke jeweils minimale, sowie maximale Zellspannung. Der Funktion wird das JSON Array übergeben, dass in executeQuery() geholt wird. In diesem sind alle 144 Zellspannungen gespeichert.

### kapa(data[146],removeClass/addClass)

Der Funktion wird der Wert der Gesamtspannung übergeben. Mit Hilfe dieses Wertes werden dann unter Nutzung von “addclass/removeclass“ zehn Anzeigeflächen so eingefärbt, dass sie einen Wert zwischen 0 und 100 % simulieren.

### toggle\_tab()

Beim Seitenaufbau werden zuerst alle Tabellenspalten erstellt. Sobald der Seitenaufbau fertig ist, wird diese Funktion ausgeführt und versteckt alle Spalten, die sich auf Akkuzellen beziehen.

**toggle\_arrow()** Dies ist eine Funktion, die “toggle\_tab()“ zugeordnet ist. Sie sorgt dafür, dass sich der visuelle Button zum Öffnen und Schließen nach dem Auslösen um 180° dreht. Im Initialzustand zeigt der Pfeil nach unten und nach auslösen des Button dreht dieser sich und zeigt nach oben. Ein weiterer Klick macht den Vorgang rückgängig.

**toggle\_balance(), removeClass/addClass**

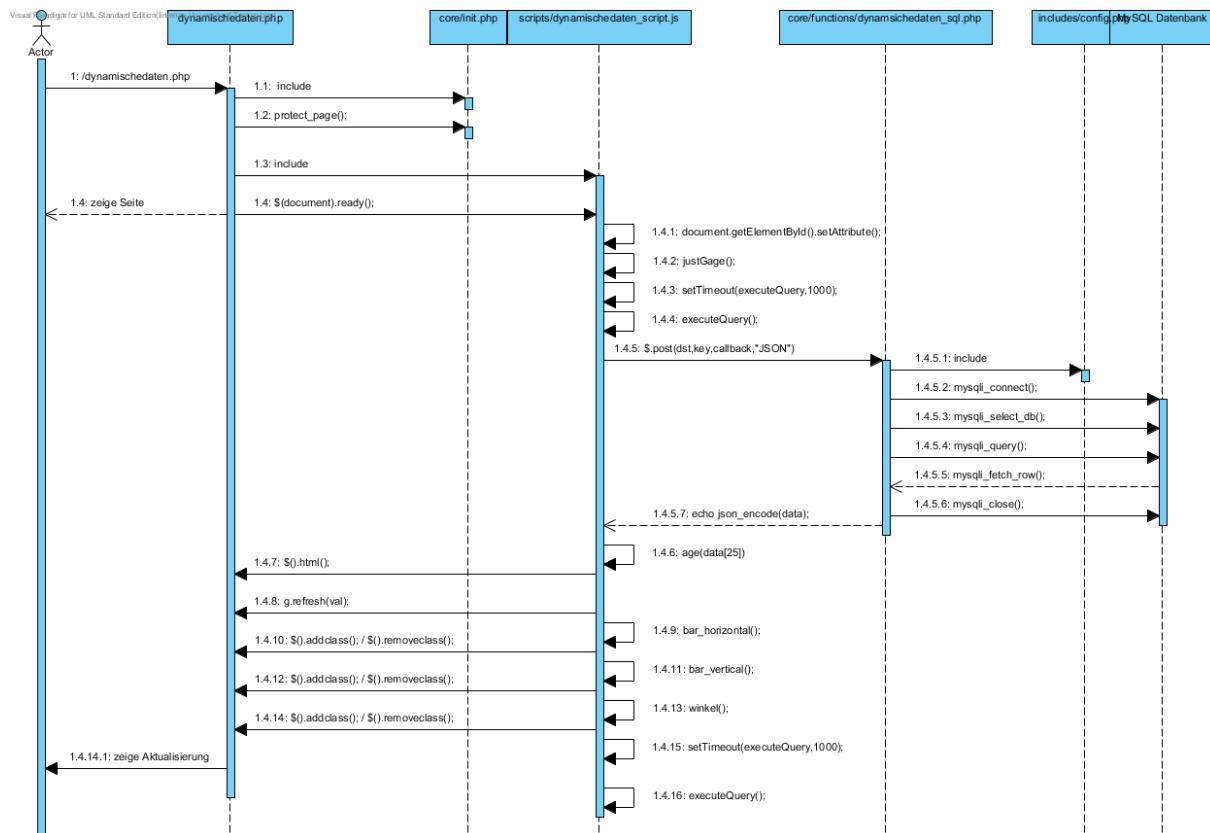
Analog wie in “toggle\_notaus“ 2.4.13 Allgemeine Fahrzeugdaten. Hier wird der Ablauf für zwei Strings durchgeführt, die Balancing-Werte enthalten und bei Aktivität blau eingefärbt werden. Des weiteren wird an dieser Stelle auch gezählt, wie viele Zellen den Status “AN“ haben. Die Anzahl wird in die Spalte des entsprechenden Akkublocks geschrieben und wenn diese größer als 0 ist ebenfalls blau hinterlegt.

## 2.4.16 Dynamische Daten

Der Aufbau ist wieder analog den vorherigen zwei Seiten. Auf dieser Seite wird folgendes visualisiert:

- Drehzahlen der Räder als Halbkreis-Diagramm mit “justGage“
- Die vier Federwege an den Achsen als vertikal stehender Ladebalken
- Gaspedal 1 und 2, Bremsposition, Bremskraft, Bremsdruck, Wassertemperatur 1 und 2 werden als horizontal liegender Ladebalken
- Der Lenkwinkel wird als Balken mit Indikator visualisiert.

Auf der Seite wird die Draufsicht des Formula Student Fahrzeuges angezeigt, das mit der Spitze nach oben zeigt. Darüber befindet sich die Visualisierung des Lenkwinkels und rechts bzw. links befinden sich die Drehzahlanzeigen und die Federwege entsprechend der Achsen und Räder, die in der Fahrzeuggrafik zu sehen sind.



Bereits erwähnte Funktionen verhalten sich analog wie oben beschrieben.

### **bar\_horizontal(), removeClass/addClass**

Bekommt als Eingabe eine ID und einen Wert. Danach wird mittels “removeclass/addclass“ eine entsprechende Anzahl an Flächen farblich hinterlegt. Beschreibt eine horizontalen Ladebalken, der gefüllt wird.

### **bar\_vertical(), removeClass/addClass**

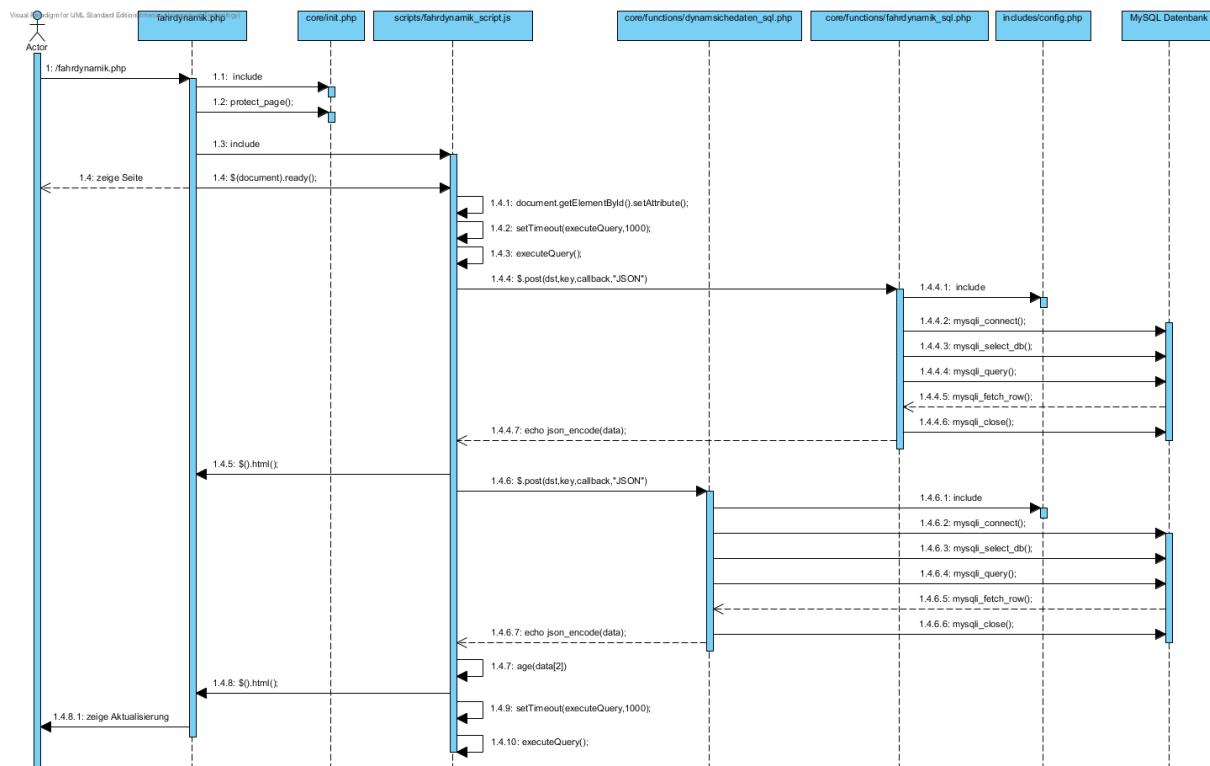
Analog wie “bar\_horizontal()“, nur das die Einfärbung der Flächen vertikal stattfindet.

### **winkel(), removeClass/addClass**

Rechnet den übergebenen Wert vom Wertebereich -180 bis +180 in den Wertebereich -90 bis +90. Danach wird genau eine Fläche eingefärbt, um den Grad des Lenkwinkels darzustellen.

## 2.4.17 Fahrdynamik

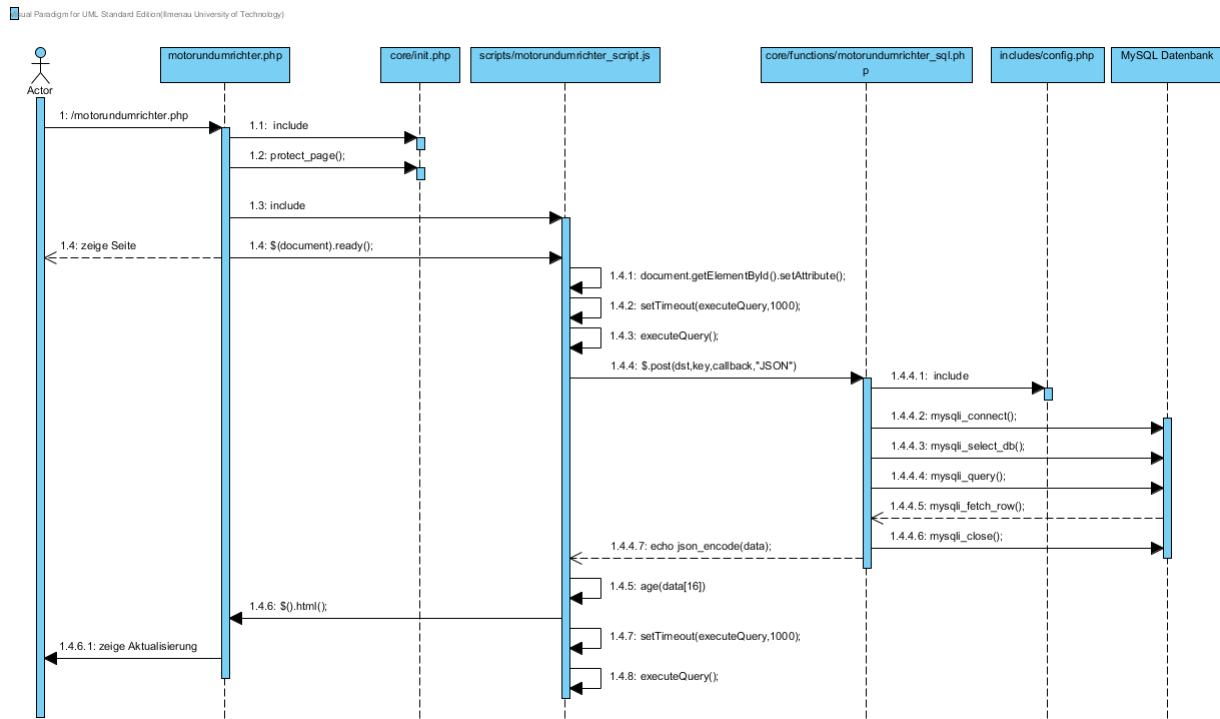
Aufbau der Seiten ist wieder analog den vorhergegangenen. Als Besonderheit auf der Seite ist nur, dass sie auf zwei Tabellen in der Datenbank zugreifen muss. Der Lenkwinkel ist nur in der Tabelle für dynamische Daten vorhanden, und muss deshalb extra abgefragt werden. Somit besteht die Seite im Prinzip aus vier Komponenten.



Funktionserklärung siehe vorhergegangene Seiten.

## 2.4.18 Motor und Umrichter

Diese Seite enthält die Daten, die im Pflichtenheft dafür vereinbart wurden und besteht wieder aus drei Komponenten. Die acht Motortemperaturen werden als Tabelle dargestellt. Es gibt keine weiteren besonderen Merkmale auf dieser Seite.



Funktion wie bereits oben erklärt.

# 3 Entwicklerdokumentation

## 3.1 MicroAutoBox II

Da sich eine eigenständige Entwicklerdokumentation mittels doxygen oder anderen Softwarelösungen bei einem Simulink-Modell nicht anbietet, wurden bereits alle wesentlichen Informationen über das Modell im Feinentwurf unter Punkt 2.1 erläutert.

## 3.2 Embedded PC

Bitte sehen Sie zur Entwicklerdokumentation des Abschnitts Embedded PC in die separate, mit Doxygen erzeugte Entwicklerdokumentation.

## 3.3 vServer

Bitte sehen Sie zur Entwicklerdokumentation des Abschnitts vServer in die separate, mit Doxygen erzeugte Entwicklerdokumentation.

## 3.4 Webseite

Das Benutzersystem ermöglicht es nur bestimmten Personen auf Zugriff zur Seite zu gewähren. Hierbei gibt es die Möglichkeit sich persönlich zu registrieren und anschließend nach der Verifikation und Freischaltung durch den Vorstand (= Administrator) sich auch einzuloggen. Im eingeloggten Zustand stehen dann Seiten zur Verfügung, die als nicht eingeloggter Benutzer nicht ersichtlich sind.

Im Folgenden sollen nun Funktionen und Variablen erläutert werden, sowie die Funktionalität der einzelnen Skripte des Systems.

### 3.4.1 Allgemeine Funktionen (general.php)

In dieser Datei befinden sich Funktionen, die nicht direkt mit dem aktiven Nutzer zu tun haben, sondern allgemein als (Hilfs-)Funktionen dienen.

#### **install\_system(array \$register\_data)**

##### **Author**

Thomas Golda

##### **Beschreibung**

Diese Funktion dient dem Erstellen aller wichtigen Datenbanken, Tabellen und dem Anlegen eines Vorstand-Accounts, sowie dem Einrichten der Benutzergruppen ‚Vorstand‘, ‚Beobachter‘ und ‚Techniker‘. Das übergebene Array beinhaltet hierbei die für die Einrichtung des Vorstand-Accounts nötigen Benutzerinformationen.

#### **logged\_in\_redirect()**

##### **Author**

Thomas Golda

##### **Beschreibung**

Dieser Funktion werden keine Parameter übergeben. Sie verwendet hierbei die Nutzerfunktion *logged\_in()* und leitet einen Nutzer auf die Startseite weiter, wenn er bereits eingeloggt ist. Sie ist insbesondere dann nützlich, wenn man Seiten vor einem unsinnigen Zugriff des Nutzers schützen möchte. So macht es als eingeloggter Benutzer beispielsweise keinen Sinn das Registrierungsformular aufzurufen.

#### **protect\_page()**

##### **Author**

Thomas Golda

**Beschreibung**

Diese Funktion arbeitet ähnlich zu *logged\_in\_redirect()*, prüft ebenfalls durch Zuhilfenahme der Nutzerfunktion *logged\_in()* ob sich ein aufrufender Nutzer bereits eingeloggt hat. Wenn dies nicht der Fall ist, wird er auf die Startseite weiter geleitet. Diese Funktion hilft insbesondere Seiten vor äußereren Zugriffen zu schützen.

**array\_sanitize(array \$item)****Author**

Thomas Golda

**Beschreibung**

Diese Funktion erhält als Übergabe ein Array von Strings, die es mittels der PHP-internen Funktion *mysqli\_real\_escape\_string()* aufbereitet um SQL-Injections beim Zugriff auf die Datenbank zu vermeiden.

**sanitize(string \$data)****Author**

Thomas Golda

**Beschreibung**

Wie *array\_sanitize()* wird auch hier der String von Sonderzeichen bereinigt. Der Unterschied: *sanitize()* arbeitet auf einem einzelnen String.

**output\_errors(array \$errors)****Author**

Thomas Golda

**Beschreibung**

Diese Funktion gibt alle auf einer Seite gesammelten benutzerdefinierten Fehlermeldungen in strukturierter Art und Weise wieder aus und weist den Benutzer somit darauf hin, dass ein Fehler (höchstwahrscheinlich) durch sein Verschulden entstanden ist. Er erhält auch zeitgleich eine Fehlermeldung, die ihn darauf hinweist, wo sein Fehler liegt kann.

### 3.4.2 Nutzerfunktionen (`users.php`)

Die Funktionen in dieser Datei beziehen sich vorrangig auf den aktiven Benutzer und stellen Möglichkeiten dar, auf seine Daten zuzugreifen oder mit diesen zu arbeiten.

#### **change\_password(int \$user\_id, string \$password)**

##### **Author**

Thomas Golda

##### **Beschreibung**

Bekommt eine Nutzer-ID und einen String (= Passwort) übergeben. Das Passwort wird verschlüsselt und anschließend in der MySQL-Datenbank der Eintrag mit der ID `$user_id` in der Benutzertabelle ermittelt. Für diesen Eintrag wird das alte Passwort durch das neue ersetzt.

#### **register\_user(array \$register\_data)**

##### **Author**

Thomas Golda

##### **Beschreibung**

Diese Funktion bekommt ein Array von Strings übergeben und legt einen neuen Datenbankeintrag in der Tabelle für die Benutzer an. Der neu angelegte Nutzer ist standardmäßig deaktiviert und hat somit nicht das Recht sich einzuloggen.

#### **user\_count()**

##### **Author**

Thomas Golda

##### **Beschreibung**

Bestimmt die Anzahl der im System registrierten Benutzer. Nicht mit einberechnet werden noch nicht aktivierte Accounts.

## **user\_data(int \$user\_id)**

### **Author**

Thomas Golda

### **Beschreibung**

Diese Funktion erfordert mindestens ein Argument, welches die ID eines Nutzers ist. Wird sie mit mehr als diesem Parameter aufgerufen, so behandelt sich alle folgenden Parameter (vorzugsweise Strings) als gewünschte auszugebende Parameter.

```
if (Logged_in() == true) {
    $session_user_id = $_SESSION['user_id'];
    $user_data = user_data($session_user_id, 'id', 'passwort', 'vorname', 'nachname', 'email', 'rechte');

    if (user_active($user_data['email']) == false) {
        session_destroy();
        header('Location: index.php');
        exit();
    }
}
```

Abbildung 3.1: Quellcodeauszug aus *core/init.php*

Die Funktion selbst bestimmt zur übergebenen ID die weiterhin angegebenen Parameter und gibt diese als Array aus. So gibt die aus dem Bild ersichtliche aufgerufene Funktion die zur ID gehörigen Daten wie Vor- und Nachname, E-Mail-Adresse, Rechtegruppe und das (verschlüsselte) Passwort.

## **isVorstand()**

### **Author**

Thomas Golda

### **Beschreibung**

Prüft, ob der aktuell aktive Benutzer zur Rechtegruppe ‚Vorstand‘ zugehörig ist und gibt entweder *true* oder *false* zurück, wenn er es nicht ist.

## **isTechniker()**

### **Author**

Thomas Golda

### **Beschreibung**

Prüft, ob der aktuell aktive Benutzer zur Rechtegruppe ‚Techniker‘ zugehörig ist und gibt entweder *true* oder *false* zurück, wenn er es nicht ist.

## isBeobachter()

### Author

Thomas Golda

### Beschreibung

Prüft, ob der aktuell aktive Benutzer zur Rechtegruppe ‚Beobachter‘ zugehörig ist und gibt entweder *true* oder *false* zurück, wenn er es nicht ist.

## timeout()

### Author

Thomas Golda

### Beschreibung

Überprüft, ob der in der Session-Variable `$_SESSION['time']` befindliche UNIX-Zeitstempel älter als `$time` Minuten ist. Standardwert: `$time = 60`. Gibt *true* zurück, falls der Zeitstempel nicht älter als `$time` Minuten ist, *false* andernfalls.

## logged\_in()

### Author

Thomas Golda

### Beschreibung

Gibt zurück, ob der aktive Nutzer eingeloggt ist oder nicht. Dies impliziert auch eine Überprüfung, ob seine letzte Aktivität weiter als eine in der Funktion `timeout()` festgelegte Zeitspanne zurück liegt. Setzt zudem nach dieser Prüfung den Zeitstempel in der Sessionvariable `$_SESSION['time']` auf einen aktuellen Zeitwert.

## refresh()

### Author

Thomas Golda

### Beschreibung

Diese Funktion aktualisiert den Zeitwert in der Tabelle der Nutzer die sich zum aktuellen Zeitpunkt online befinden. Zudem werden alle veraltete Einträge unabhängig des Benutzers gelöscht.

**user\_exists(string \$mail)****Author**

Thomas Golda

**Beschreibung**

Ermittelt mittels einer SQL-Anfrage, ob der Benutzer der zur übermittelten E-Mail-Adresse *\$mail* existiert oder nicht. Gibt im Erfolgsfall *true*, sonst *false* zurück.

**user\_active(string \$mail)****Author**

Thomas Golda

**Beschreibung**

Gibt *true* zurück, wenn der Nutzer mit der E-Mail-Adresse *\$mail* bereits aktiviert wurde, *false* sonst.

**user\_id\_from\_mail(string \$mail)****Author**

Thomas Golda

**Beschreibung**

Bekommt als Argument einen String *\$mail* (= E-Mail-Adresse) und ermittelt mittels einer SQL-Anfrage anschließend die ID des dazugehörigen Nutzers.

**login(string \$mail, string \$password)****Author**

Thomas Golda

**Beschreibung**

Gibt bei Erfolg die Nutzer-ID des zugehörigen Nutzers zurück. Existiert kein Nutzer in der Datenbank mit den angegebenen Zugangsdaten, so gibt die Funktion *false* zurück.

### 3.4.3 Konfigurationsdatei (**config.php**)

Die Konfigurationsdatei wird stets in der core/init.php eingebunden und ist somit auf jeder Seite verfügbar, auf der auch init.php eingebunden wird. Insbesondere liefert sie die Zugangsdaten für den Zugriff auf die MySQL-Datenbanken und speichert auch die Namen der einzelnen Tabellen, sowohl für die Fahrzeugdaten als auch für die Nutzerdaten.

#### **\$dbhost**

Hier kann der Hostname des MySQL-Servers festgelegt werden. Er wird meist vom Hoster mitgeteilt und lautet in den meisten Fällen ‚localhost‘.

#### **\$dbuname**

Hierbei handelt es sich um den Login-Namen der Zugangsdaten zum Datenbankserver. Diesen erhalten Sie ebenfalls von Ihrem Hoster.

#### **\$dbpass**

Hierbei handelt es sich um das Passwort der Zugangsdaten zum Datenbankserver. Diesen erhalten Sie ebenfalls von Ihrem Hoster oder legen Sie selbst in Ihrer Serververwaltung fest.

#### **\$dbname\_fd**

Hiermit legen Sie den Namen der Datenbank fest, in der die Tabellen der Fahrzeugdaten gespeichert werden sollen.

#### **\$dbname\_ud**

Hiermit legen Sie den Namen der Datenbank fest, in der die Tabellen des Benutzersystems gespeichert werden sollen.

#### **\$accu\_data**

Beinhaltet den Namen der Akkudaten-Tabelle.

#### **\$general\_data**

Beinhaltet den Namen der Tabelle für die allgemeinen Fahrzeugdaten.

#### **\$dynamic\_data**

Beinhaltet den Namen der Tabelle für die dynamischen Fahrzeugdaten.

#### **\$engine\_data**

Beinhaltet den Namen der Tabelle für die Motor- und Umrichterdaten.

**\$driving\_data**

Beinhaltet den Namen der Tabelle für die Fahrdynamikdaten.

**\$user**

Beinhaltet den Namen der Tabelle der registrierten Benutzer.

**\$rights**

Beinhaltet den Namen der Tabelle für die verschiedenen Rechtegruppen.

**\$online**

Beinhaltet den Namen der Tabelle für die sich momentan online befindlichen Nutzer.

**\$mail**

Beinhaltet die E-Mail-Adresse des Vorstands oder derjenigen Person, die Benutzer freischalten können soll. (Achtung: Funktionalität nicht implementiert!)

**\$salt**

Ein Teil des Salt für das Passwort. Kann beliebiger String sein.

**\$alg**

Ein Teil des Salt für das Passwort. Gibt den Verschlüsselungsalgorithmus an. (siehe Dokumentation der PHP-Funktion *crypt()*)

**\$rounds**

Ein Teil des Salt für das Passwort. Gibt die Anzahl der Runden an, die der Algorithmus laufen soll. Dabei gilt, je größer umso sicherer, aber auch aufwändiger und langsamer.

**\$cryptsalt**

Endgültiger Salt. Setzt sich aus \$salt, \$alg und \$rounds zusammen.

### 3.4.4 akkudaten.php

→ siehe 2.4.15 Akkudaten

Diese Seite gibt einen Überblick über alle Daten, die die Akkumulatoren betreffen. Zur Anzeige und Visualisierung wurden die unten folgenden Funktion geschaffen. Zum einen regeln sie das Verhalten der Tabelle, z.B. das die aktuellen Werte eingetragen sind und das die Tabelle auf-, bzw. zugeklappt werden kann. Zum anderen auch die Visualisierung der Gesamtspannung als horizontal liegender Ladebalken. Aktualisierungen von Zellspannung, Temperatur und Balancing erfolgen separat. Für Spannung und Balancing wurde dafür eine separate Funktion geschrieben. Die Temperaturaktualisierung wird dabei in der Funktion executeQuery() ausgeführt.

→ siehe **3.6 akkudaten\_script.js**

### 3.4.5 allgemeinefahrzeugdaten.php

→ siehe 2.4.14 Allgemeine Fahrzeugdaten

Diese Seite bietet eine Übersicht über ein Auswahl an Parametern. Näheres dazu im Feinentwurf. Für diese Seite wurde die Funktion toggle\_notaus() geschrieben. Sie regelt, dass das Notausfeld jeweils AN oder AUS anzeigt, bzw. einen roten oder weißen Hintergrund hat. Bis auf die unten genannten Funktionen werden alle weiteren Bearbeitungsschritte direkt in „executueQuery()“ ausgeführt.

→ siehe **3.5 allgemeinefahrzeugdaten\_script.js**

### 3.4.6 changepassword.php

Hierbei handelt es sich um eine Seite, die es dem Nutzer ermöglicht sein Passwort zu ändern. Durch Eingabe seines aktuellen Passwortes und der doppelten Eingabe seines neuen Passwortes wird eine SQL-Anfrage ausgeführt, die das alte Passwort in der Datenbank durch das neue Passwort überschreibt.

→ Für weitere Informationen siehe Feinentwurfsdokument „Passwort ändern“ (2.4.9)

### 3.4.7 dynamischedaten.php

→ siehe 2.4.16 Dynamische Daten

Im Bereich dynamische Daten werden viele Visualisierungen und Veranschaulichungen genutzt. Dazu wurden du nachfolgenden Funktionen definiert, die z.T. auch mehrfach verwendet werden. Hauptaugenmerk sind dabei die als Ladebalken visualisierten Elemente, welche horizontal und auch vertikal angezeigt werden können. Ein weitere, etwa abgewandelte Funktion wird für die Anzeige des Lenkwinkels genutzt, der statt einem Ladebalken nur einen einzigen Strich als Indikator hat.

→ siehe **3.7 dynamischedaten\_script.js**

### 3.4.8 fahrdynamik.php

→ siehe 2.4.17 Fahrdynamik

Da diese Seite nur 3 Werte anzeigen soll und diese sich nicht weiter visualisieren lassen, wurden auch keine weiteren Funktionen an dieser Stelle benötigt. Für weitere Informationen zu dieser Seite, sehen sie im Feinentwurf nach.

→ siehe **3.8 fahrdynamik\_script.js**

### 3.4.9 index.php

Sie ist die Startseite des Systems und liefert im ausgeloggenen Zustand die Möglichkeit, sich mittels eines HTML-Formulares anzumelden bzw. im bereits eingeloggten Zustand sich verschiedene Widgets anzeigen zu lassen.

→ Für weitere Informationen siehe Feinentwurfsdokument „Startseite“(2.4.5)

### 3.4.10 install.php

Diese Datei wird nur beim ersten Mal des Aufrufs der Seite gestartet. Sie bietet ein Formular, welches ausgefüllt werden muss und den ersten Benutzer im System erstellt: den Vorstand. Ihre Hauptaufgabe ist vor allem das Aufrufen der Funktion *install\_system()* (siehe x.1.2).

→ Für weitere Informationen siehe Feinentwurfsdokument „Installation“(2.4.2)

### 3.4.11 login.php

Bekommt Daten vom Formular der Startseite mittels POST-Request übermittelt. Diese werden verifiziert und nach erfolgreicher Verifikation wird ein Eintrag für die Online-Tabelle erstellt. Im Fehlerfall erhält der Nutzer eine Ausgabe über falsche bzw. fehlende Zugangsdaten.

→ Für weitere Informationen siehe Feinentwurfsdokument „Login“(2.4.7)

### 3.4.12 logout.php

Beendet die aktuelle Sitzung, löscht den zum aktiven Nutzer gehörigen Eintrag in der Online-Tabelle und leitet den Nutzer zurück auf die Startseite.

→ Für weitere Informationen siehe Feinentwurfsdokument „Logout“(2.4.8)

### 3.4.13 motorundumrichter.php

→ siehe 2.4.18 Motor und Umrichter

Auch auf dieser Seite werden keine anspruchsvollen Visualisierungen benötigt, so dass die Seite ohne weitere Funktionsdefinitionen auskommt.

→ siehe **3.9 motorundumrichter\_script.js**

### 3.4.14 register.php

Überprüft die bei der Registrierung eingegebenen Daten und schreibt diese nach der Validierung in die Benutzertabelle der Datenbank. Das Passwort wird dabei verschlüsselt und gesalzen (Salt = \$cryptsalt siehe x.1.3) um es sicher zu speichern. Außerdem wird nach der Registrierung der Status des Accounts auf 0 gesetzt, d.h. er muss anschließend manuell aktiviert werden.

→ Für weitere Informationen siehe Feinentwurfsdokument „Registrierung“ (2.4.6)

### 3.4.15 core/init.php

Hierbei handelt es sich um eine Datei, die zu Beginn jeder Inhaltsseite eingebunden wird. Sie wird insbesondere im Header eingebunden, welcher wiederum in jeder Inhaltsseite eingebunden wird und dafür sorgt, dass die Sitzung nach ihrer Erzeugung über jeden Seitenwechsel erhalten bleibt.

→ Für weitere Informationen siehe Feinentwurfsdokument „Initialisierung“ (2.4.3)

### 3.4.16 core/functions/accept.php

Erhält vom Widget *includes/widgets/manageusers.php* per POST-Request Daten. Diese werden ausgewertet, d.h. es wird überprüft, ob eine ID angegeben wurde und falls ja, diese von der eigenen verschieden ist. Des Weiteren wird überprüft, ob eine gewünschte Verwaltungsoption („Löschen“ bzw. „Aktivieren“) ausgewählt wurde, ansonsten wird der Nutzer zurück auf die Indexseite weitergeleitet. Abschließend wird noch die übergebene Rechtegruppe in einen Integer umgewandelt und mit den Daten anschließend eine Anfrage auf die Datenbank ausgeführt.

→ Für weitere Informationen siehe Feinentwurfsdokument „Benutzerverwaltung“ (2.4.11)

### 3.4.17 core/functions/akkudaten\_sql.php

→ siehe 2.4.15 Akkudaten

Analog wie oben funktioniert auch diese Seite, nur das hier auf die Tabelle „\$accu\_data“ zugegriffen wird, die in diesem Fall „akkudaten“ entspricht.

### 3.4.18 core/functions/allgemeinefahrzeugdaten\_sql.php

→ siehe 2.4.14 Allgemeine Fahrzeugdaten

Auf den Seiten mit dem Zusatz SQL werden die eigentlichen Datenbankoperationen ausgeführt. Dazu wird zuerst die Konfigurationsdatei „config.php“ eingebunden um die nötigen Zugriffsinformationen zu erhalten. Danach wird mit den in PHP integrierten mySQLi-Methoden auf die Datenbank zugegriffen und der entsprechende Datensatz ausgelernt. In diesem Fall wird aus der Tabelle „allgemeine\_fahrzeugdaten“ gelesen, die in der Variable „\$general\_data“ aus der Konfigurationsdatei geliefert wurde. Danach wird der Datensatz als JSON, d.h. als Array dekodiert und an die aufrufende Seite zurückgegeben.

### **3.4.19 core/functions/csvexp\_sql.php**

Erhält vom Widget *includes/widgets/csvexp.php* die Informationen als POST-Variablen, welche Checkboxen vor Absenden des Formulars ausgewählt wurden. Diese Informationen werden ausgewertet und anschließend in eine SQL-Anfrage übersetzt die dann die gewünschten Daten ausgibt. Diese Daten werden dem Nutzer jedoch nicht im Browser dargestellt, sondern als CSV-Datei zum Download angeboten.

→ Für weitere Informationen siehe Feinentwurfsdokument „CSV Export“ (2.4.10)

### **3.4.20 core/functions/dynamischedaten\_sql.php**

→ siehe 2.4.16 Dynamische Daten

Besonderheit bei dieser Seite ist, dass sie von 2 unterschiedlichen Skripten aufgerufen wird, da in dieser Tabelle der Lenkwinkel hinterlegt ist, der auf 2 Seiten dargestellt werden muss. Ansonsten analog wie oben für “\$dynamic\_data= dynamische Daten.“

### **3.4.21 core/functions/fahrdynamik\_sql.php**

→ siehe 2.4.17 Fahrdynamik

### **3.4.22 core/functions/general.php**

Beinhaltet allgemeine (Hilfs-)Funktionen für das Benutzersystem.

→ Ausführliche Beschreibung siehe 3.4.1

### **3.4.23 core/functions/motorundumrichter\_sql.php**

→ siehe 2.4.18 Motor und Umrichter

### **3.4.24 core/functions/users.php**

Beinhaltet benutzerbezogene (Hilfs-)Funktionen für das Benutzersystem.

→ Ausführliche Beschreibung siehe 3.4.2

### **3.4.25 includes/css/style.css**

Beinhaltet alle CSS-Formatierungen, die auf der kompletten Seite durchgeführt wurden.  
Für weitere Informationen siehe Kommentare in der Datei.

### **3.4.26 includes/overall/footer.php**

Beinhaltet den globalen Footer, der auf jeder Inhaltsseite eingebunden wird.

### 3.4.27 includes/overall/header.php

Beinhaltet den globalen Header, der auf jeder Inhaltsseite eingebunden wird. Bindet zeitgleich auch die *core/init.php* und *includes/header.php* für die Darstellung des Menüs ein.

### 3.4.28 includes/widgets/csvexp.php

Hierbei handelt es sich um ein Widget, welches zur Auswahl der verschiedenen Fahrzeugdatentabellen jeweils eine Checkbox liefert und diese Daten nach einem Klick an die *core/functions/csvexp\_sql.php* weiterleitet.

→ Für weitere Informationen siehe Feinentwurfsdokument „CSV Export“(2.4.10)

### 3.4.29 includes/widgets/loggedin.php

Dies ist kein Widget wie es die anderen Dateien dieses Ordners sind. Es wird stets im Header eingebunden und liefert im eingeloggten Zustand eine namentliche Begrüßung des aktiven Nutzers. Des Weiteren liefert es auf den einzelnen Inhaltsseiten eine Information über das Alter der Daten. Außerdem gibt es einen Link zur „Passwort ändern“-Funktion sowie zur Startseite.

### 3.4.30 includes/widgets/manageusers.php

Dieses Widget erlaubt es im System registrierte Benutzer zu verwalten, d.h. sie zu aktivieren, zu löschen und ihnen eine Rechtegruppe zuzuweisen. Dies geschieht über ein Formular, dessen Daten über POST-Variablen an die *core/functions/accept.php* weitergereicht werden.  
→ Für weitere Informationen siehe Feinentwurfsdokument „Benutzerverwaltung“(2.4.11)

### 3.4.31 includes/widgets/onlineusers.php

Ein weiteres Widget, welches die Benutzer als HTML-Tabelle ausgibt, die sich gerade eingeloggt im Benutzersystem befinden. Hierbei befindet sich bei jedem Nutzereintrag auch eine relative Zeitangabe zu seiner letzten Aktivität.

→ Für weitere Informationen siehe Feinentwurfsdokument „Online-Anzeige der Nutzer“(2.4.12)

### 3.4.32 includes/widgets/userlist.php

Dieses Widget liefert über die *includes/widgets/onlineusers.php* hinaus eine Liste aller im System registrierten Benutzer. Neben Vor- und Nachname werden auch E-Mail-Adresse, Rechtegruppe, Kontostatus und Online-Status ausgegeben.

→ Für weitere Informationen siehe Feinentwurfsdokument „Online Benutzer“(3.4.31)

## 3.5 Skriptübergreifende Funktionen

Um einen Indikator zu haben, wie alt der aktuellste Datensatz aus der Datenbank ist, wurde die Funktion “age()“ geschrieben. Aus der Differenz der aktuellen Zeit und des Zeitwerts aus der Datenbank ergibt sich die Anzahl an Sekunden, Minuten, bzw. Stunden in denen kein neuer Datensatz in die Datenbank geschrieben wurde. Des weiteren wird hier die Funktion “executeQuery()“ aufgeführt, die ebenfalls ein Hauptbestandteil ist.

### age(param)

#### Author

David Kudlek

#### Beschreibung

In dieser Funktion wird mit der aktuellen UNIX Zeit und dem Zeitpunkt der Daten aus der Tabelle errechnet, wie lange die letzte Aktualisierung zurückliegt. Nach 15 Sekunden Verzögerung wird der aktuelle Wert mittig, unterhalb der Navigationsleiste angezeigt. Der Wert wird in Tagen, Stunden, Minuten und Sekunden errechnet. Es werden nur Stunden, Minuten und Sekunden angezeigt.

#### Parameter:

**int param**      Zeitpunkt, der in der Datenbank gespeichert wurde.

### executeQuery()

#### Author

David Kudlek

#### Beschreibung

In dieser Funktion findet die Aktualisierung sämtlicher Daten statt. In ihr wird ein Array von einer externen Datei angefordert, dann direkt bearbeitet oder es werden Teile der Bearbeitung in separate Funktionen verlagert. Am Ende seiner Ausführung startet sie einen Time-Out, mit der die Funktion executeQuery nach 1000ms erneut aufgerufen wird. Auf diese Art und Weise wird die Funktion kontinuierlich ausgeführt und realisiert so die selbstständige Aktualisierung.

#### Parameter:

bekommt keine Parameter übergeben

## 3.6 scripts/allgemeinefahrzeugdaten\_script.js

### **toggle\_notaus(param)**

#### **Author**

David Kudlek

#### **Beschreibung**

Diese Funktion bekommt einen String übergeben, der mit einer Folge aus 0 und 1 die Notausschalter kodiert.

#### **Parameter:**

**char param**      String aus einer Folge von 0 und 1.

## 3.7 scripts/akkudaten\_script.js

### **toggle\_tab(k)**

#### **Author**

David Kudlek

#### **Beschreibung**

Durch toggle\_tab werden die Untermenüs für die Zellen der Blöcke auf, bzw. zugeklappt. Bei Initialisierung werden sie automatisch geschlossen.

#### **Parameter:**

**int k ∈ [0,11]**      wählt die richtige Zeile aus, deren Untermenü angezeigt/ausgeblendet werden soll.

### **toggle\_arrow(k)**

#### **Author**

David Kudlek

#### **Beschreibung**

Durch toggle\_arrow wird die Richtung der Abbildung im ersten Tabellenfeld um 180° gedreht. Dies dient zur Visualisierung des auf-, bzw. zuklappens der Tabellen mit den Akkuzellen.

**Parameter:**

int  $k \in [0,11]$  wählt die richtige Zeile aus, deren Abbildung gedreht werden soll.

**kapa(param)****Author**

David Kudlek

**Beschreibung**

Kapa steht für die Gesamtkapazität. Mit dieser Funktion wird die Visualisierung als horizontaler Ladebalken realisiert. Die Anzeige besteht aus 10 Container, die je nach dem übergebenen Wert, von links beginnend eingefärbt werden.

**Parameter:**

int param  $\in [0,1000]$

Gibt die aktuelle Gesamtkapazität wieder. Wird durch 100 geteilt, so dass die Anzeige mit 10 farblich veränderbaren Flächen von 0 bis 100% angezeigt werden kann. Ein Fläche entspricht 10%, bzw. 100 Volt.

**accu\_refresh(data)****Author**

David Kudlek

**Beschreibung**

Bei dieser Funktion werden alle 144 Zellspannungen ausgelesen und in die entsprechenden Container geschrieben. Weiterhin wird für je 12 Zellen die maximale, sowie minimale Spannung ermittelt und in die Spalte des entsprechenden Akkumulator-Blocks geschrieben

**Parameter:**

array data Das gesamte JSON-Array, dass aus der Funktion executeQuery geliefert wird

**toggle\_balance(param,offset,offset2)****Author**

David Kudlek

**Beschreibung**

Dieser Funktion werden 2 Strings übergeben, die jeweils 72 Balancingwerte als Folge von 0 und 1 codiert. Bei 0 ist das Balancing der Zelle aus, bei 1 ist es an. Des weiteren wird für je 12 Werte die Gesamtanzahl an aktiveren Balancings gezählt, und in den entsprechenden Akkublock geschrieben.

**Parameter:**

**char param** String als Folge von 0 und 1.

**int offset: 0 oder 72**

Offset für die IDs der Zellen, damit alle 144 Zellen abgedeckt werden.

**int offset2: 0 oder 6**

Offset für die IDs der Akkublöcke, damit alle 12 Akkublöcke abgedeckt werden.

## 3.8 scripts/dynamischedaten\_script.js

### winkel(param)

**Author**

David Kudlek

**Beschreibung**

Diese Funktion visualisiert den Lenkwinkel. Dabei wird ein horizontaler Container verwendet, der 181 Abstufungen hat und jede Abstufung  $2^\circ$  entspricht. Es wird immer genau eine dieser Abstufung entsprechend dem übergebenen Wert eingefärbt. Die Funktion wird mit jedem Aufruf durch Addition von  $90$  auf  $0^\circ$  justiert, bevor die Berechnung des aktuellen Winkels durchgeführt wird. Die Anzahl der Abstufungen wurde auf 181 begrenzt, um die Serverlast so gering wie möglich zu halten, ohne Verlust der Visualisierung.

**Parameter:**

**int param  $\in [-180, 180]$**

Zeigt den aktuellen Lenkwinkel. Wird durch 2 geteilt und dadurch auf die 181 Abstufungen normiert.

### bar\_vertical(param,id)

**Author**

David Kudlek

**Beschreibung**

Aufruf verändert die visuelle Darstellung eines vertikalen stehenden Containers. Dieser enthält 10 Flächen, die eingefärbt werden können. Die Einfärbung beginnt von unten. 0 Federweg entspricht dem voll ausgefülltem Container und 100 dem leeren Container.

**Parameter:**

int param  $\in [0,100]$

- **data[18] (vl), data[19] (vr), data[20] (hl), data[21] (hr)**

Übergibt die Federwege im Bereich 0mm bis 100mm. Wird durch 10 geteilt und dadurch auf Skala normiert.

char id:

- 'vl','vr','hl','hr'

Es wird ein eindeutiger String als Identifikation übermittelt, der zum aktualisieren der Werte benötigt wird. die IDs beschreiben die Achsen: (v)orne, bzw. (h)inten, sowie die Seiten: (l)inks und (r)echts

**bar\_horizontal(param,id,kind)****Author**

David Kudlek

**Beschreibung**

Aufruf verändert die visuelle Darstellung eines horizontal liegenden Containers. Dieser enthält 20 Flächen, die eingefärbt werden können.

**Parameter:**

int param:

- **data[22] (gaseins) data[23] (gaszwei), data[17] (bremse), data[16] (bkraft)  $\in [-100,100]$**

Werte haben Bereich von -100% bis 100%. Der Wert wird durch 10 geteilt, um ihn auf die vorgegebene Skala zu normieren.

- **data[15](+100)  $\in [0,200]$**

Wert hat einen Bereich von 0 bis 200 bar und wird deshalb mit 100 addiert um den korrekten Wertebereich darzustellen. Der Wert wird durch 10 geteilt, um ihn auf die vorgegebene Skala zu normieren.

- **data[13] (wtemp1), data[14] (wtemp2) ∈ [-100,100]**

Werte haben Bereich von  $-100^{\circ}\text{C}$  bis  $100^{\circ}\text{C}$ . Der Wert wird durch 10 geteilt, um ihn auf die vorgegebene Skala zu normieren.

**char id:**

- **'gaseins','gaszwei','bremse','bkraft','bdruck','wtemp1','wtemp2'**

Es wird ein eindeutiger String als Identifikation übermittelt, der zum aktualisieren der Werte benötigt wird.

**int kind ∈ [1,3]** Es wird ein Zahlenwert übermittelt, der die richtige Klasse in der CSS-Datei bestimmt, und so die entsprechende Farbe bestimmt. (1=blau, 2=grün, 3=rot)

## 3.9 scripts/fahrdynamik\_script.js

Es wurden keine weiteren Funktionen für diesen Bereich erstellt.

## 3.10 scripts/motorundumrichter\_script.js

Es wurden keine weiteren Funktionen für diesen Bereich erstellt.

# Abbildungsverzeichnis

2.1	Produkt-Block . . . . .	5
2.2	UND-Gatter . . . . .	5
2.3	Rate-Transition-Block . . . . .	6
2.4	Width - Block . . . . .	6
2.5	Multiplexer / Demultiplexer . . . . .	7
2.6	S-R-Flip-Flop . . . . .	8
2.7	UDP-Receive . . . . .	8
2.8	DSDecode32-Block . . . . .	9
2.9	Byte-to-8-bit-Dekoder . . . . .	9
2.10	Gesamtaufbau Simulink-Modell . . . . .	11
2.11	Subsysteme des Signalgenerators . . . . .	13
2.12	Subsysteme Allgemeine Fahrzeugdaten . . . . .	13
2.13	Testsignale der Notaus-Schalter . . . . .	14
2.14	Enstellungen zum Datentyp im Constant-Block . . . . .	15
2.15	Enstellungen zum Datentyp im Constant-Block . . . . .	16
2.16	Generierung der Zelldaten . . . . .	16
2.17	Generierung der Testsignale für die Geschwindigkeit des Fahrzeuges . . . . .	18
2.18	Generierung der Testsignale für die Fahrdynamik des Fahrzeuges . . . . .	19
2.19	Übersicht über Subsystem „daten_allgemein“ im Signalkollektor . . . . .	21
2.20	Übersicht über das Subsystem „daten_akku“ im Signalkollektor . . . . .	23
2.21	Übersicht über das Subsystem „daten_geschwindigkeit“ im Subsystem „daten_dynamisch“ im Signalkollektor . . . . .	24
2.22	Übersicht über das Subsystem „daten_fahrdynamik“ im Signalkollektor . . . . .	25
2.23	Übersicht über das Subsystem „daten_motor“ im Signalkollektor . . . . .	26
2.24	Übersicht über das Subsystem „UDP_DATEN“ im Signaltransmitter . . . . .	27
2.25	Übersicht über das Subsystem „MSGSIZE_DATEN“ im Signaltransmitter . . . . .	28
2.26	Übersicht über das Subsystem „DATEN_PAKETINFO“ im Signaltransmitter . . . . .	28
2.27	Initialisierung der Kommunikation . . . . .	31
2.28	Übertragung der Fahrzeugdaten . . . . .	33
2.29	Sequenzdiagramm zum Einfügen in die Datenbank . . . . .	37
2.30	Klassendiagramm des virtuellen Servers . . . . .	39
2.31	Sequenzdiagramm des virtuellen Servers . . . . .	41
2.32	Klassendiagramm des virtuellen Servers . . . . .	42
2.33	Komponentendiagramm der Webseite . . . . .	43
2.34	Sequenzdiagramm für den Installationsprozess . . . . .	44
2.35	Sequenzdiagramm für die Initialisierung . . . . .	45
2.36	Sequenzdiagramm der Headerdatei . . . . .	46
2.37	Sequenzdiagramm der Startseite . . . . .	47
2.38	Sequenzdiagramm des Registriervorgangs . . . . .	48
2.39	Sequenzdiagramm des Loginvorgangs . . . . .	49
2.40	Sequenzdiagramm des Logoutvorgangs . . . . .	50

2.41 Sequenzdiagramm der Passwortänderungsfunktion . . . . .	51
2.42 Sequenzdiagramm des CSV-Export . . . . .	52
2.43 Sequenzdiagramm der Benutzerverwaltung . . . . .	53
2.44 Sequenzdiagramm der Onlineanzeige . . . . .	54
3.1 user_data() Funktion . . . . .	67
A.1 Subsystem Akkudaten . . . . .	86
A.2 Subsystem Zelldaten . . . . .	87
A.3 Subsystem Dynamische Daten . . . . .	88
A.4 Subsystem Motor- und Umrichterdaten . . . . .	89
A.5 Übersicht über das Subsystem Signalkollektor . . . . .	90
A.6 Übersicht über das Subsystem „daten_notaus“ im Subsystem Allgemeine Fahrzeugdaten im Signalkollektor . . . . .	91
A.7 Übersicht über das Subsystem „daten_temperatur_allgemein“ im Subsystem Allgemeine Fahrzeugdaten im Signalkollektor . . . . .	92
A.8 Übersicht über das Subsystem „daten_zellenpack_01“ im Subsystem „daten_zelldaten“ im Signalkollektor . . . . .	93
A.9 Übersicht über das Subsystem „daten_zelldaten“ im Subsystem „daten_akku“ im Signalkollektor . . . . .	94
A.10 Übersicht über das Subsystem „daten_dynamisch“ im Signalkollektor . . . . .	95
A.11 Übersicht über das Subsystem Signaltransmitter . . . . .	96
A.12 Übersicht über das Subsystem „EMB_RECEIVE“ im Signaltransmitter . . . . .	97
A.13 Übersicht über die Kommunikation zwischen MicroAutoBox II und Embedded-PC . . . . .	98
A.14 Übersicht über die Kommunikation zwischen Embedded-PC und virtuellem Server . . . . .	99
A.15 Klassendiagramm über Datenübertragung und Verarbeitung . . . . .	100
A.16 Klassendiagramm über Socketerstellung und -benutzung . . . . .	101

# **Anhang**

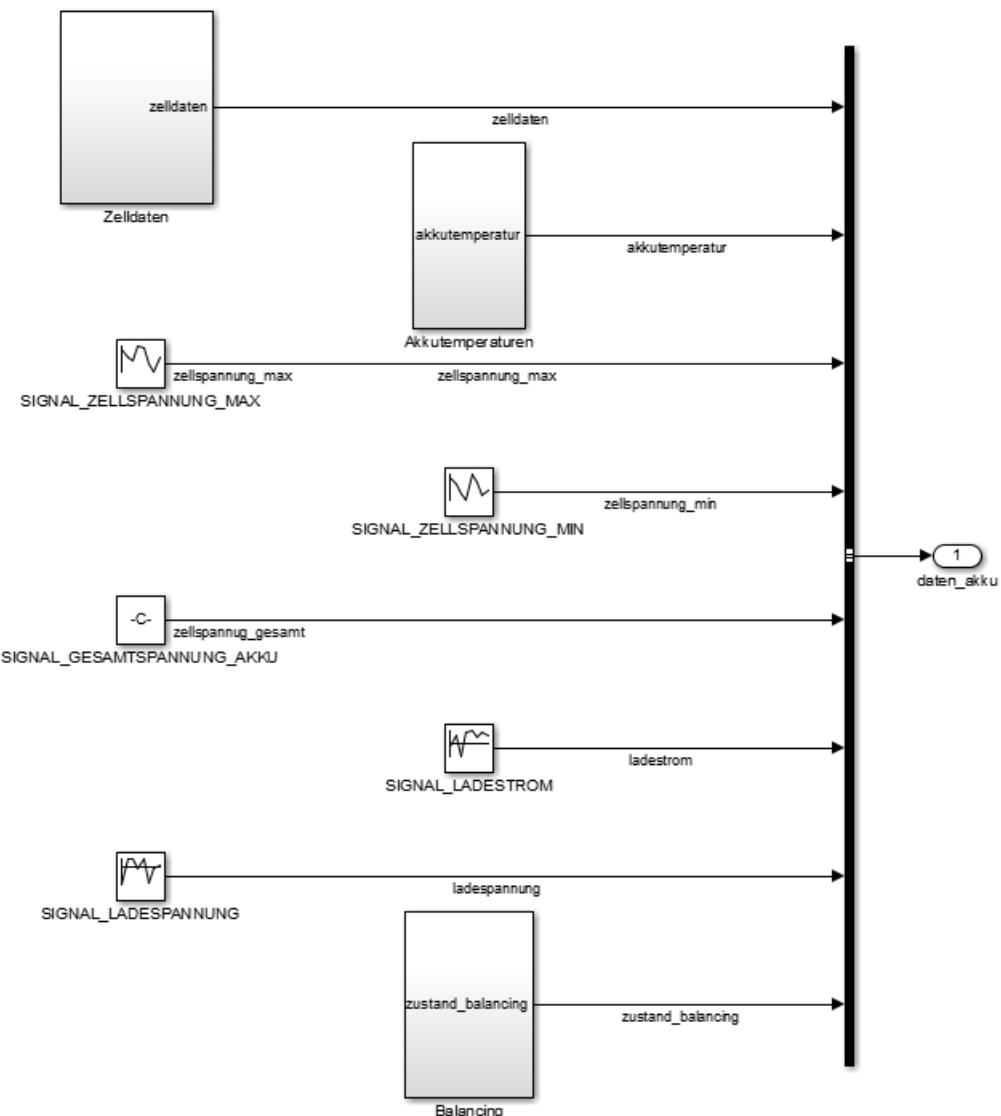


Abbildung A.1: Subsystem Akkudaten

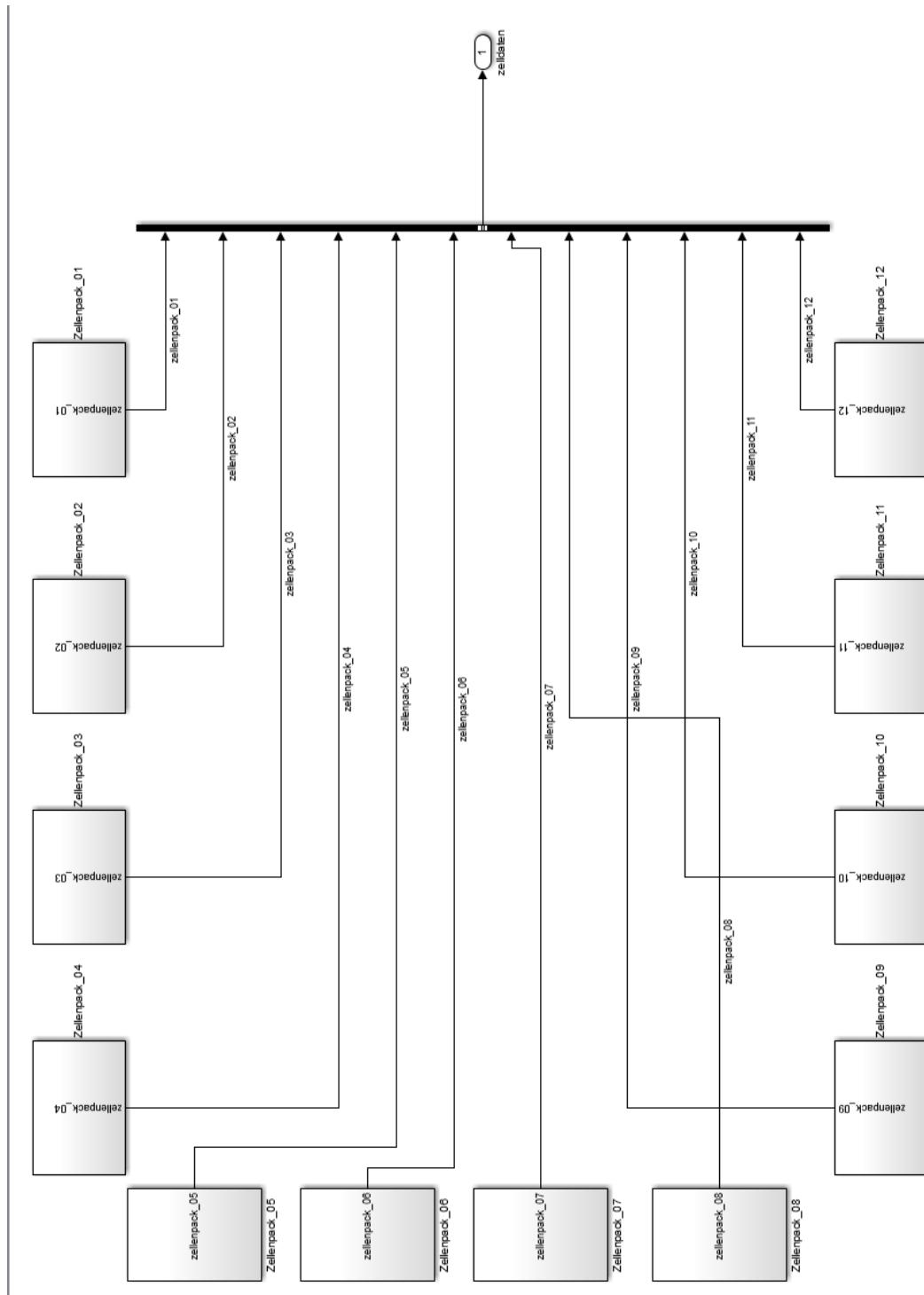


Abbildung A.2: Subsystem Zelldaten

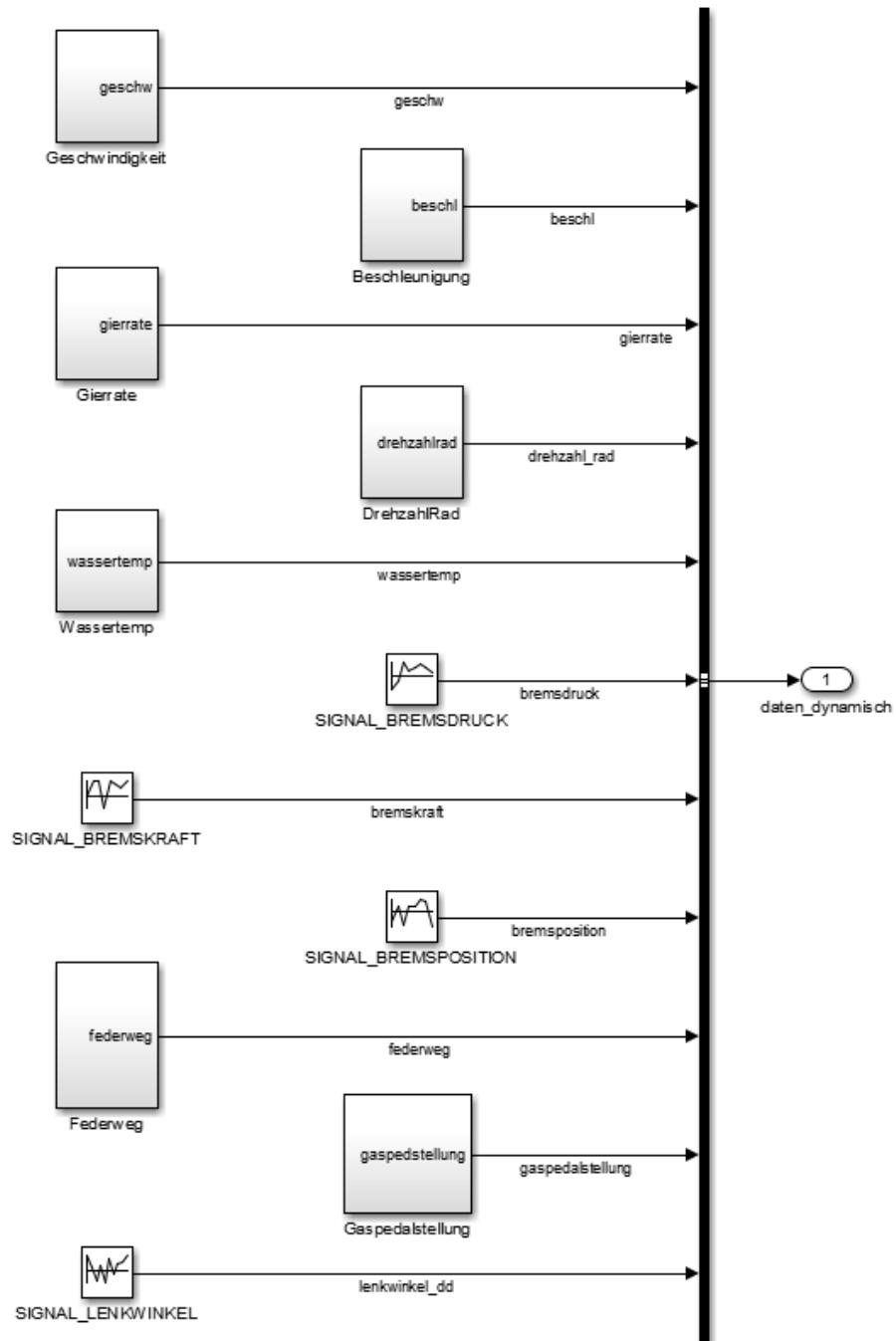


Abbildung A.3: Übersicht über das Subsystem dynamische Daten

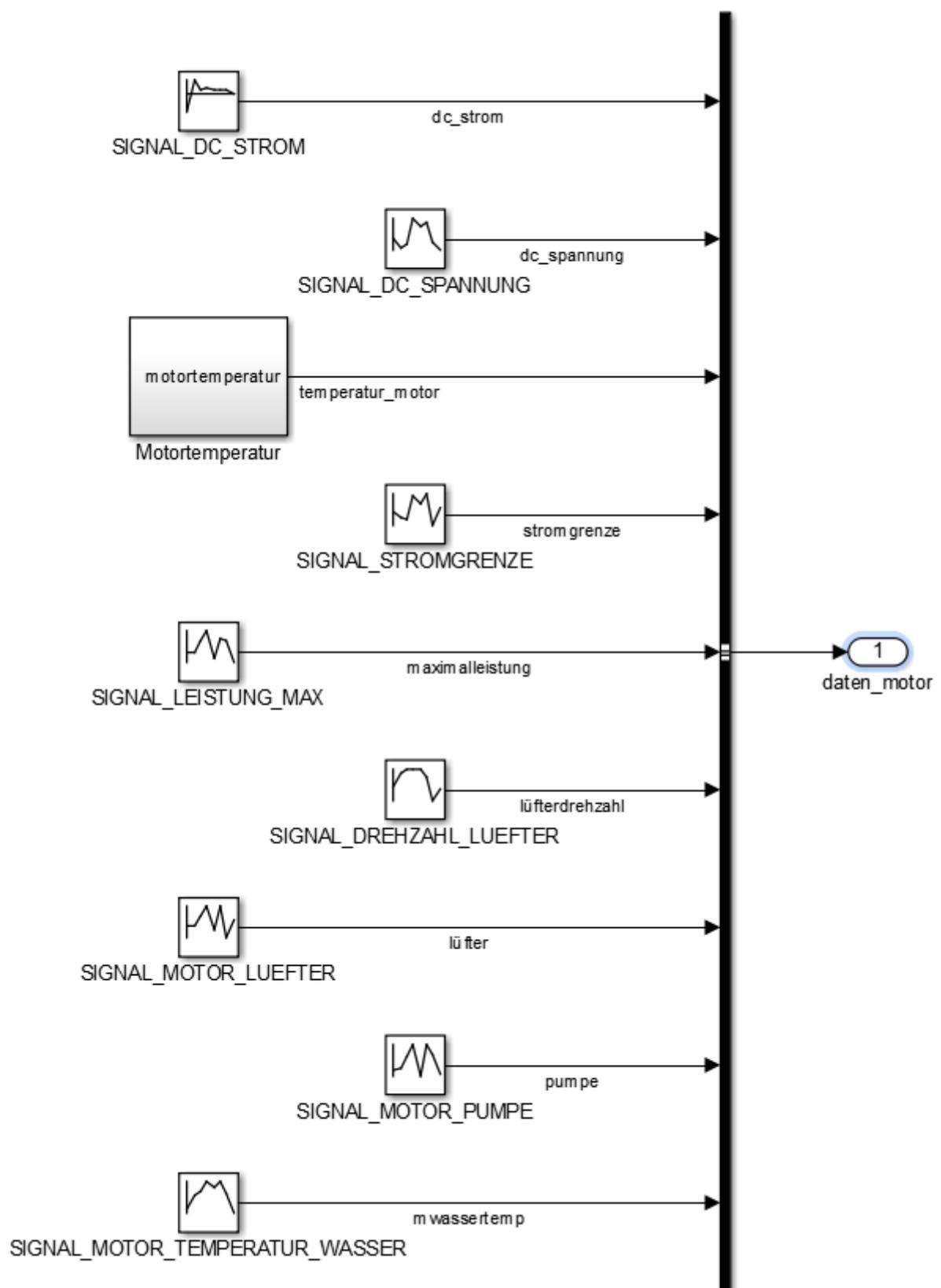


Abbildung A.4: Übersicht über das Subsystem Motor- und Umrichterdaten

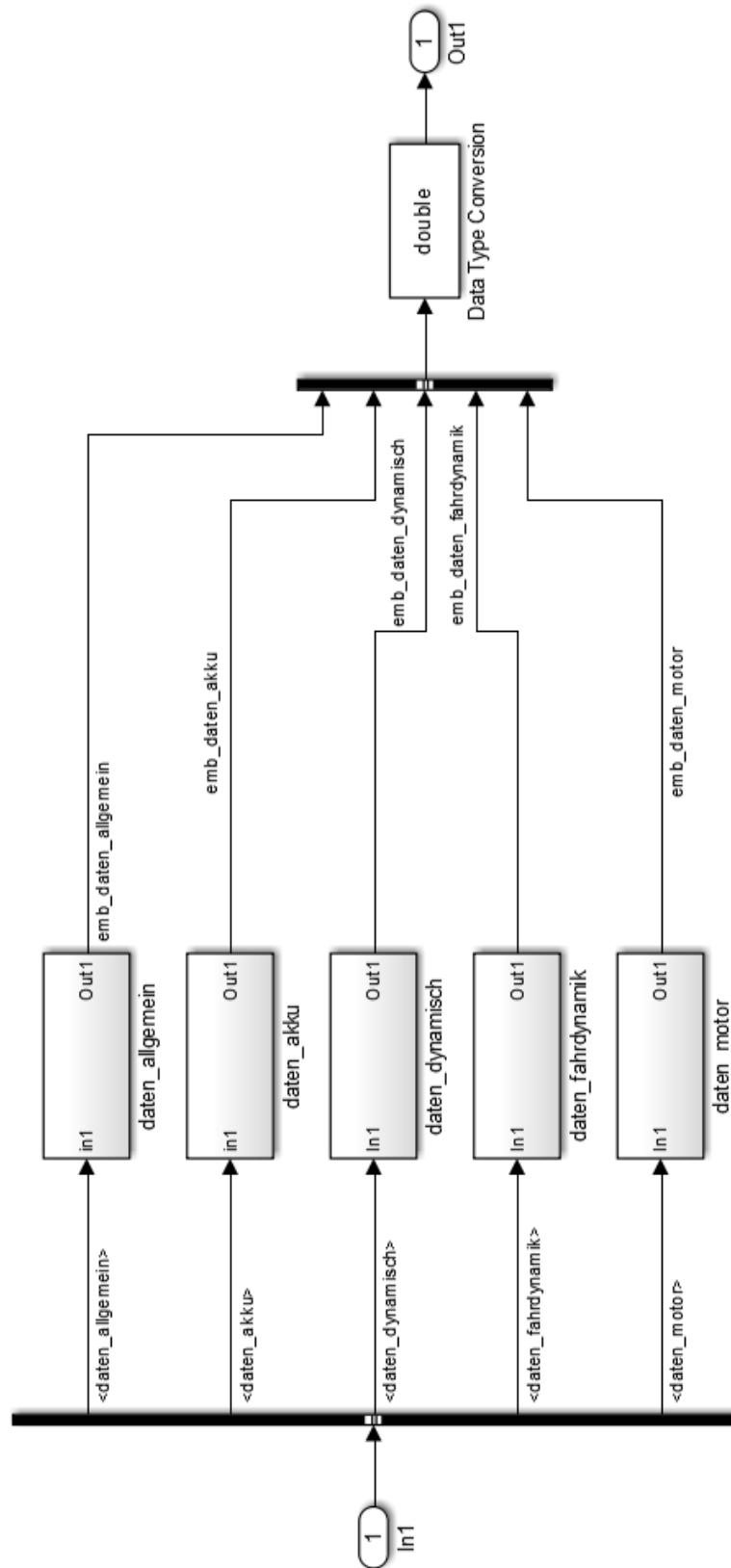


Abbildung A.5: Übersicht über das Subsystem Signalkollektor

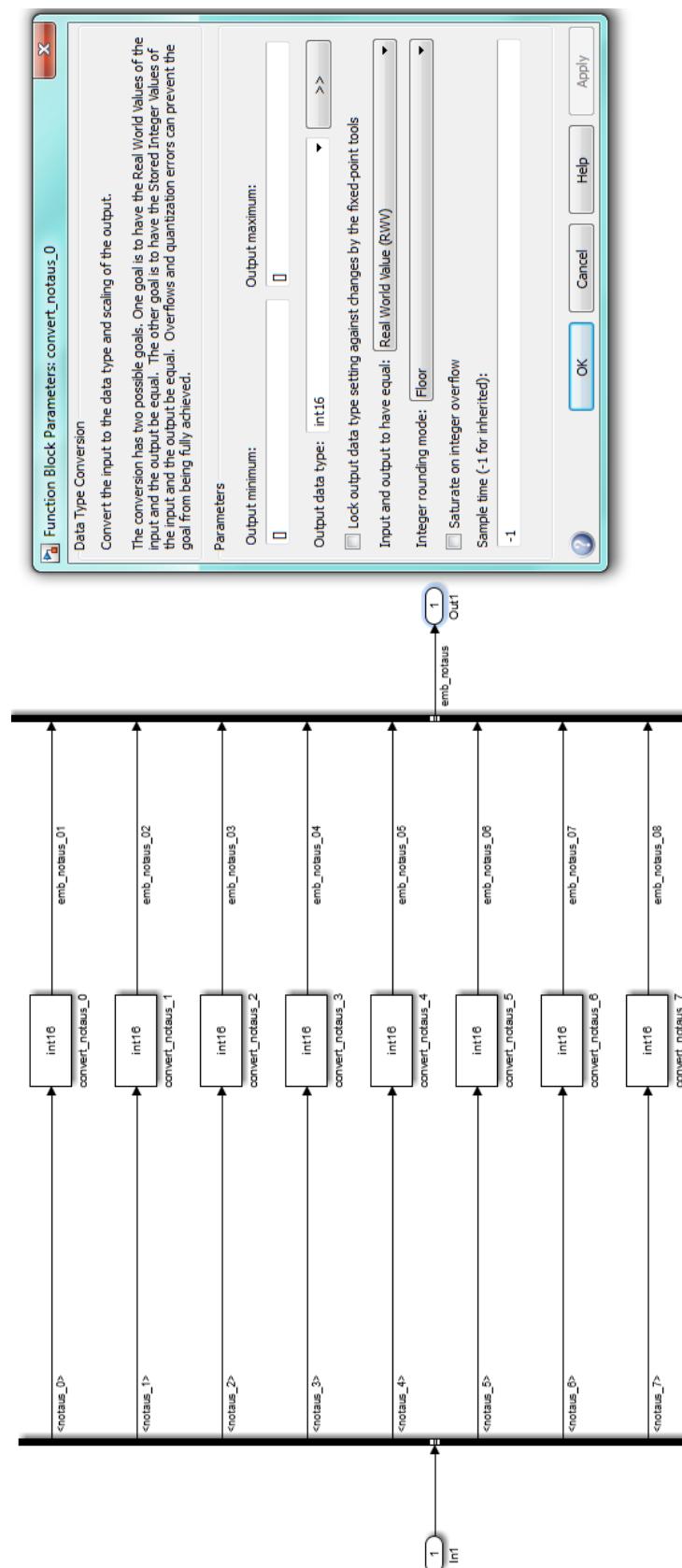


Abbildung A.6: Übersicht über das Subsystem „daten\_notaus“ im Subsystem Allgemeine Fahrzeugdaten im Signalkollektor

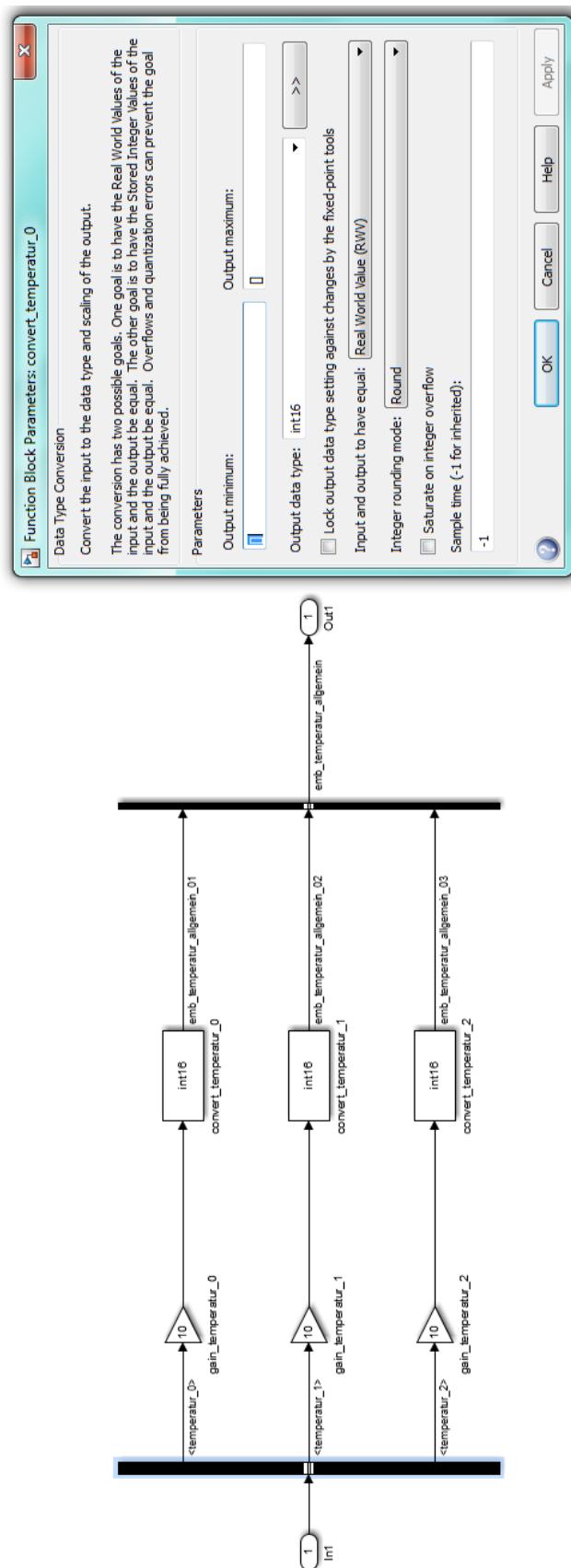
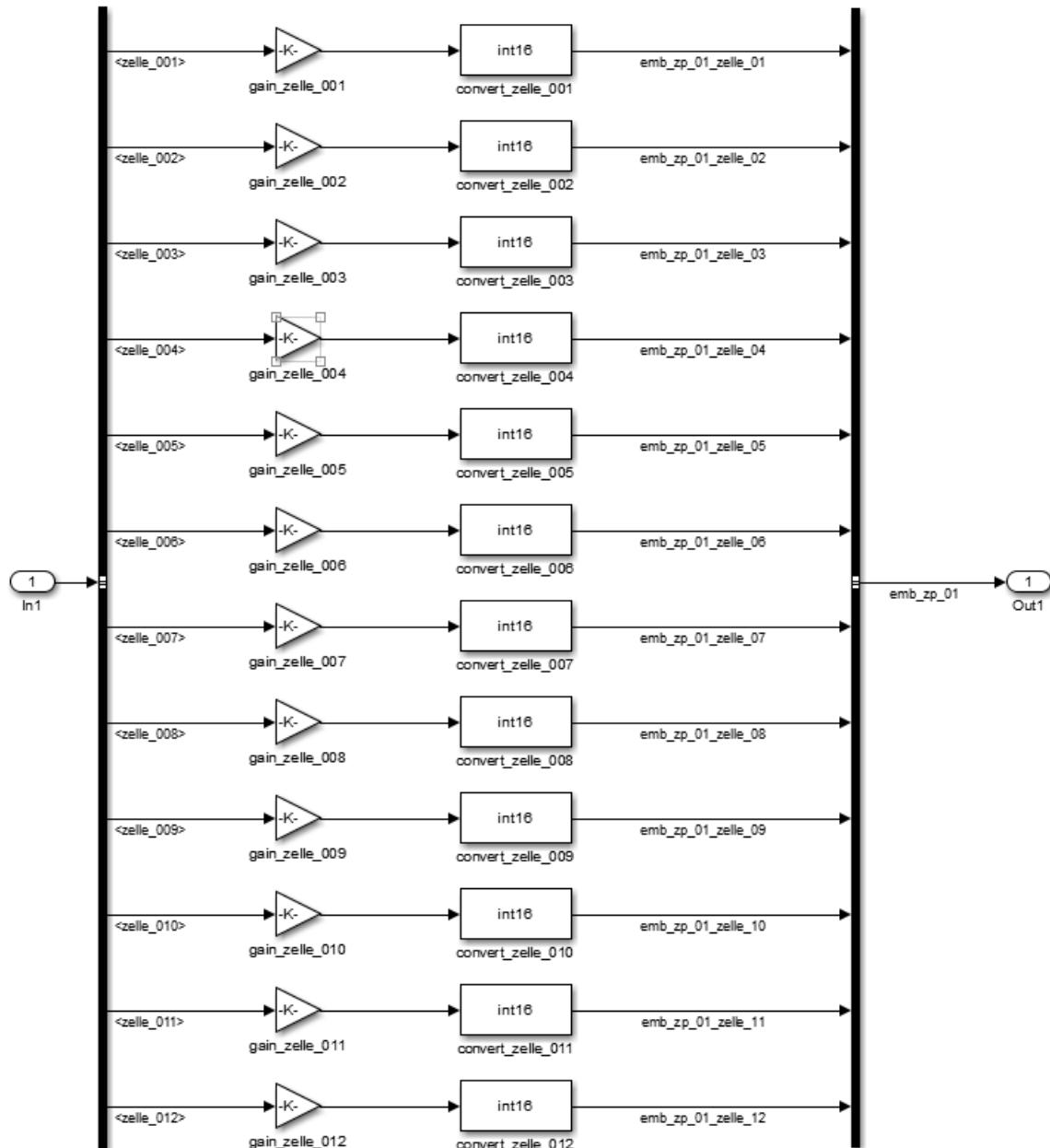
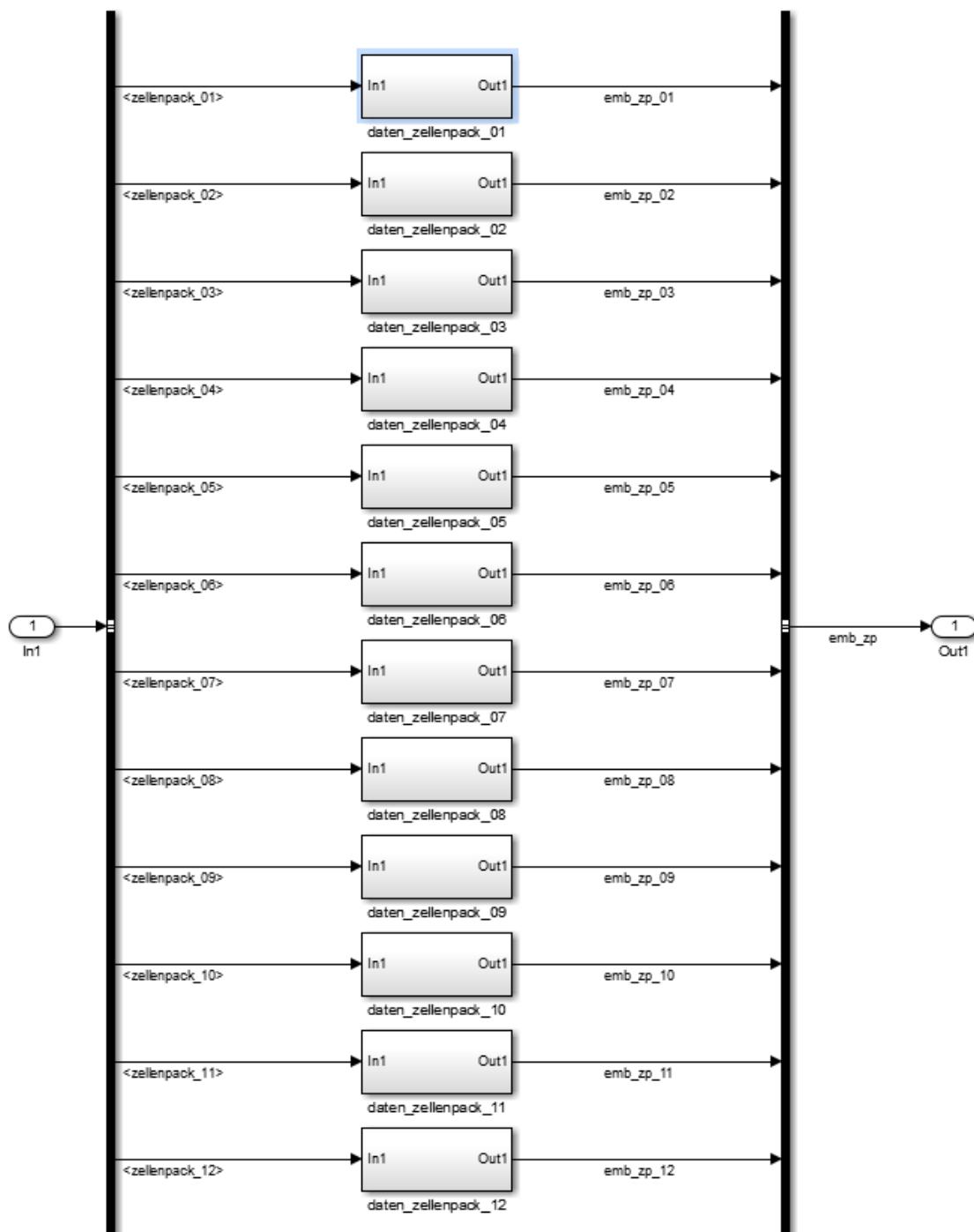


Abbildung A.7: Übersicht über das Subsystem „daten\_temperatur\_allgemein“ im Subsystem Allgemeine Fahrzeugdaten im Signalkollektor



**Abbildung A.8:** Übersicht über das Subsystem „daten\_zellenpack\_01“ im Subsystem „daten\_zelldaten“ im Signalkollektor



**Abbildung A.9:** Übersicht über das Subsystem „daten\_zelldaten“ im Subsystem „daten\_akku“ im Signalkollektor

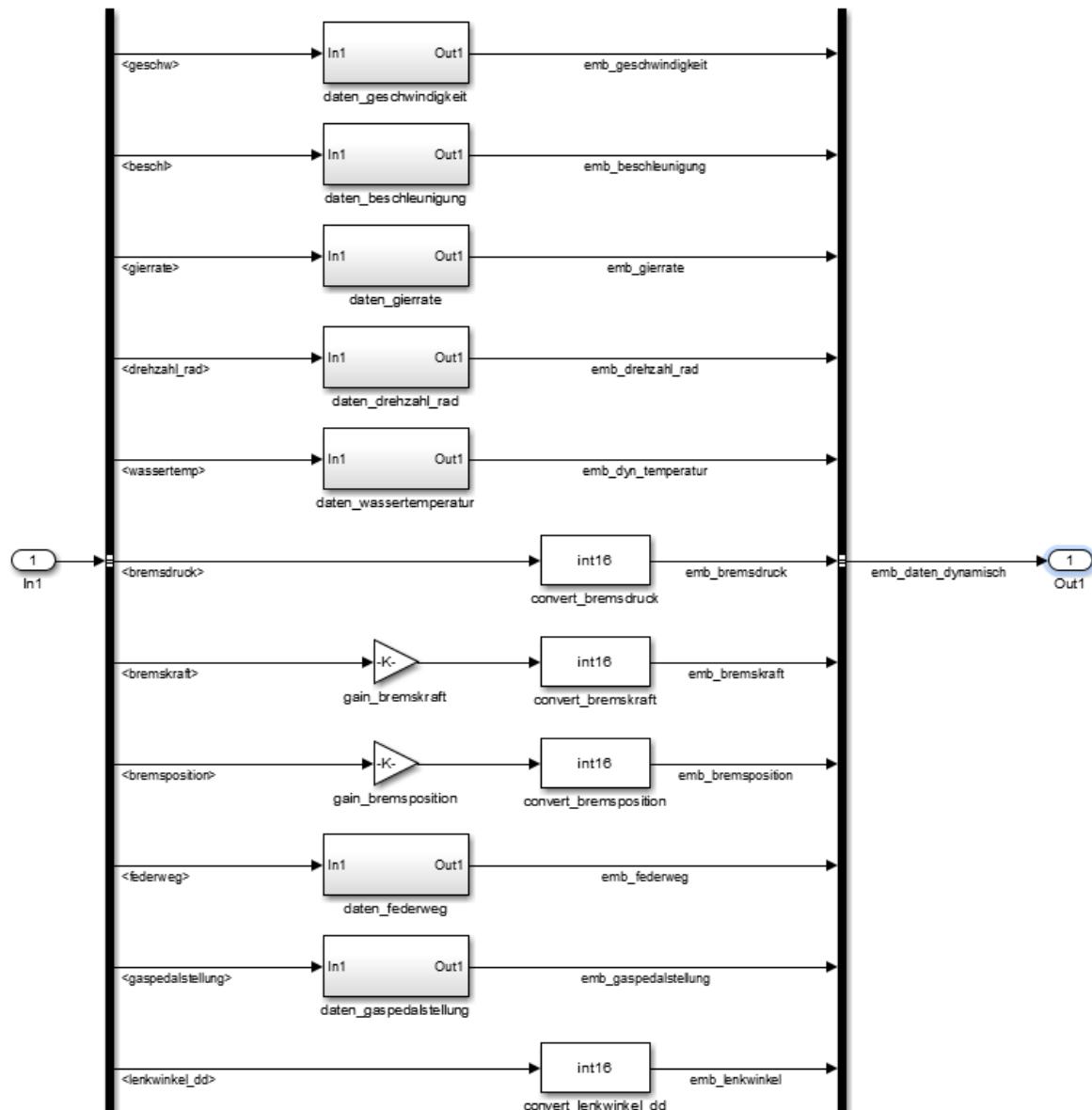


Abbildung A.10: Übersicht über das Subsystem „daten\_dynamisch“ im Signalkollektor

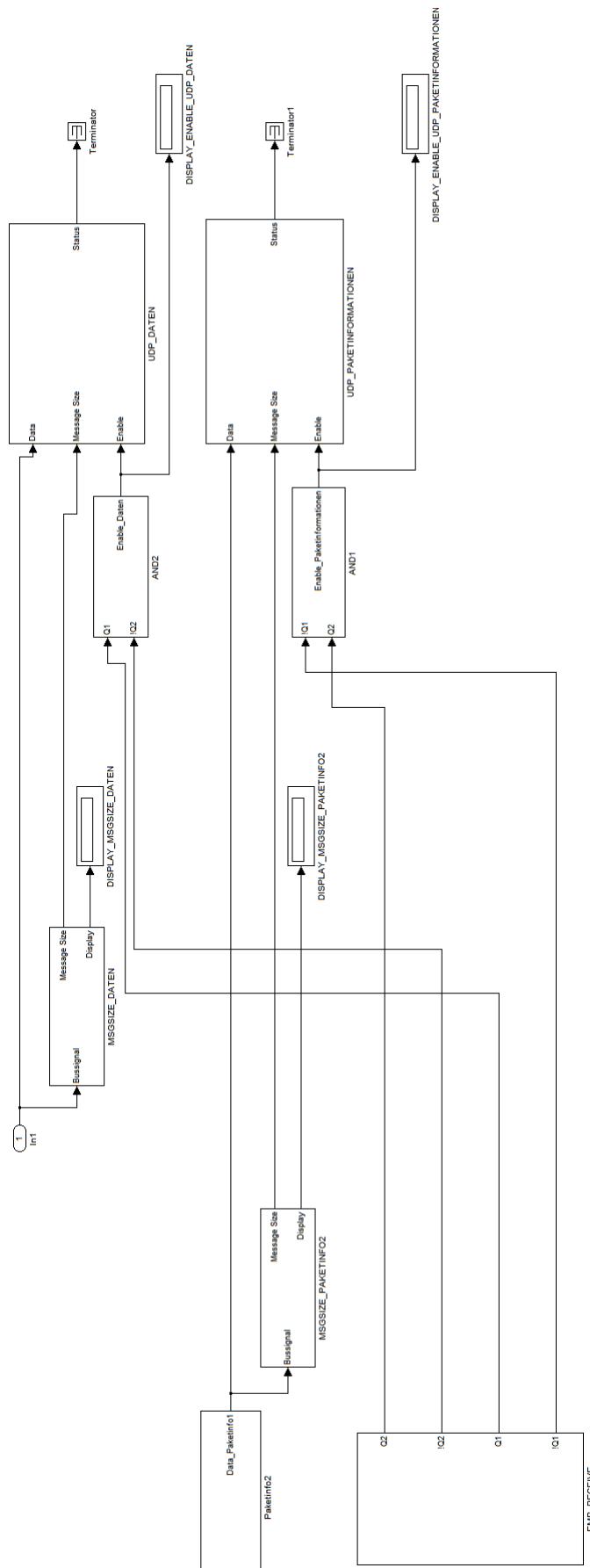


Abbildung A.11: Übersicht über das Subsystem Signaltransmitter

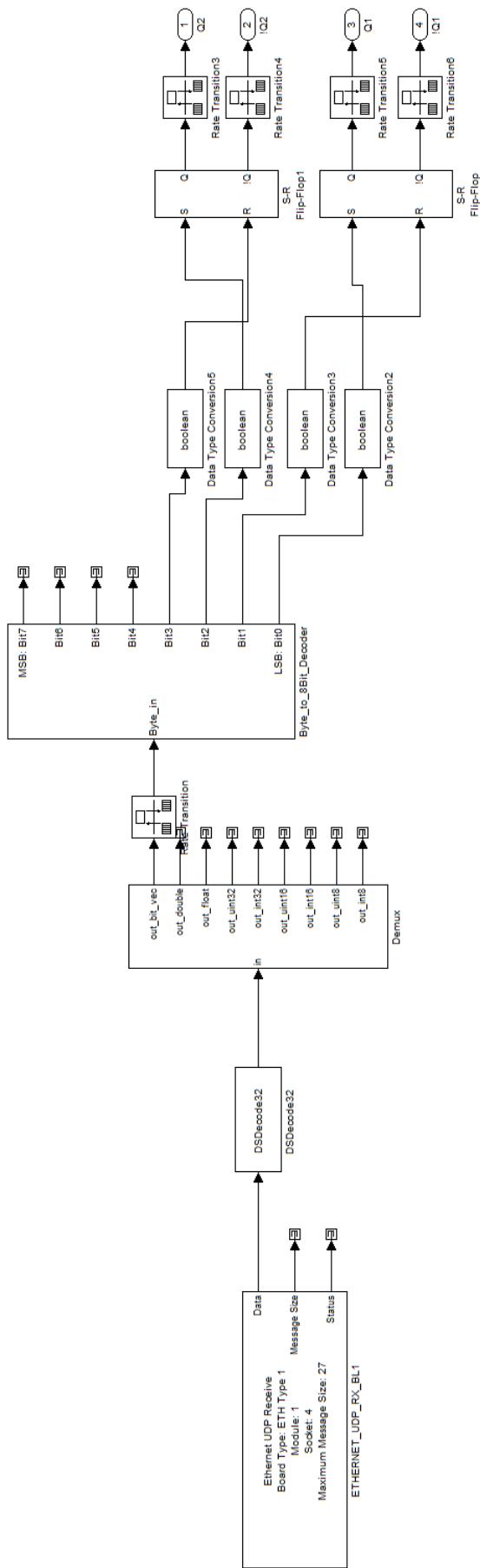


Abbildung A.12: Übersicht über das Subsystem „EMB\_RECEIVE“ im Signaltransmitter

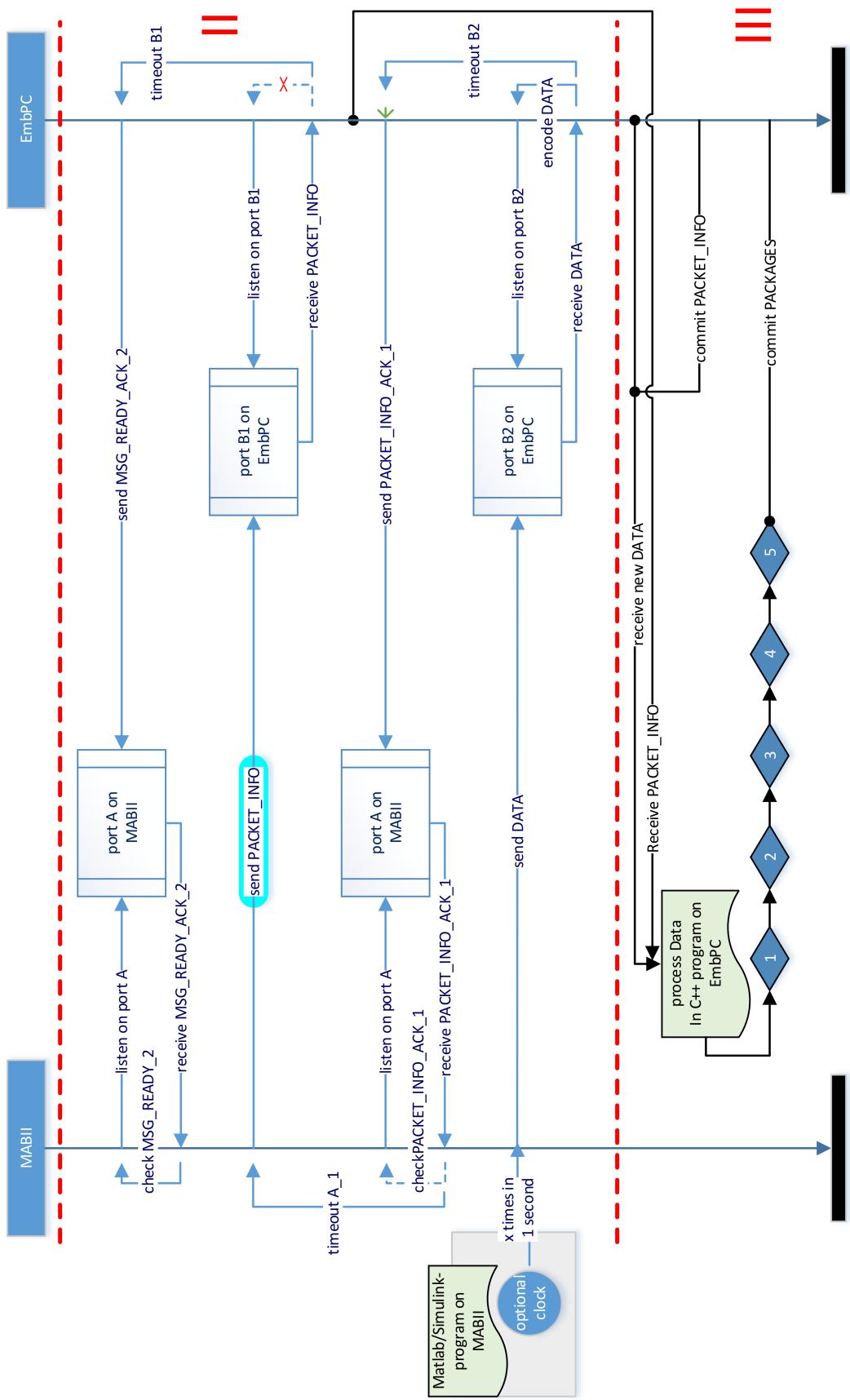


Abbildung A.13: Übersicht über die Kommunikation zwischen MicroAutoBox II und Embedded-PC

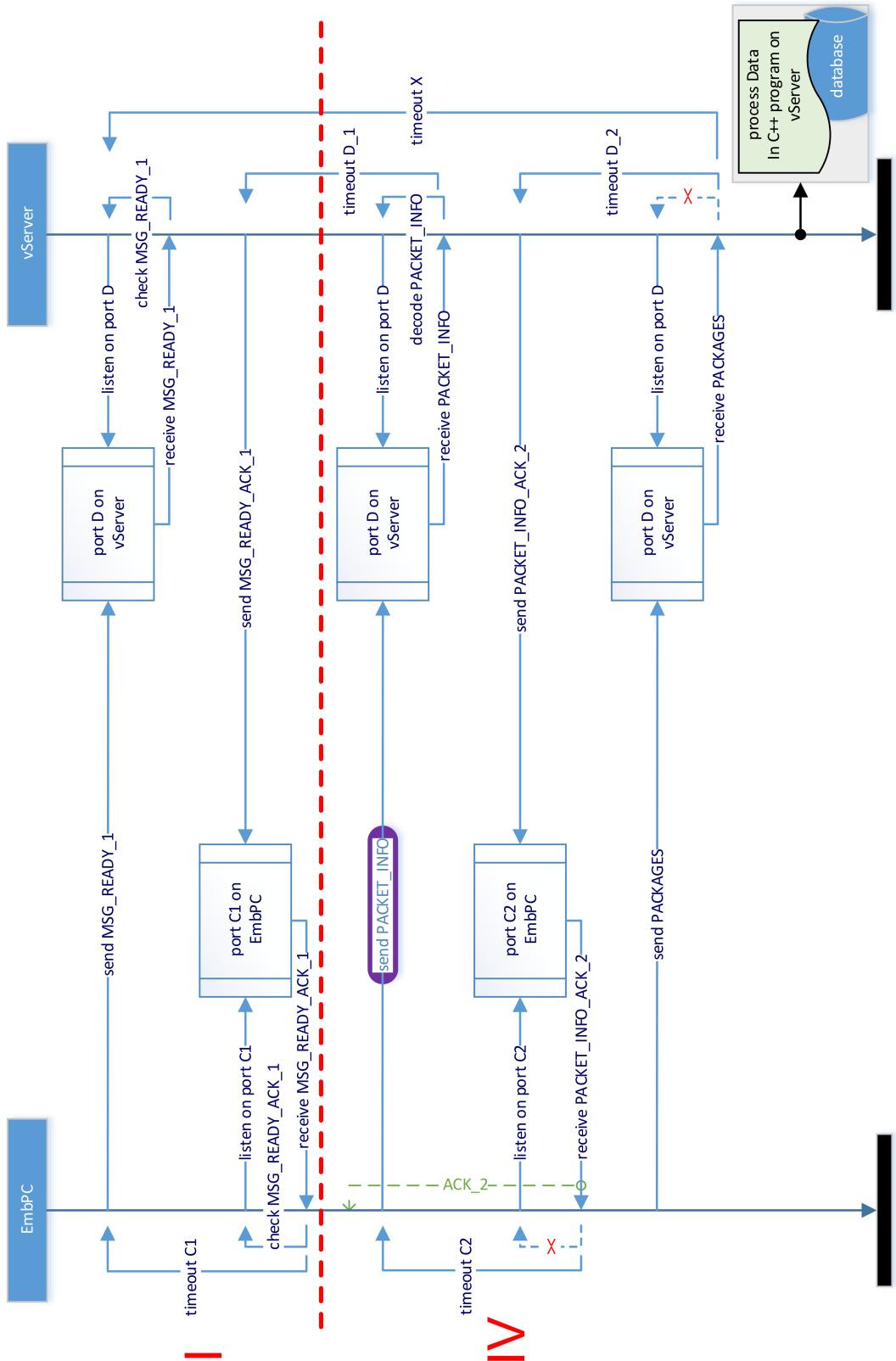
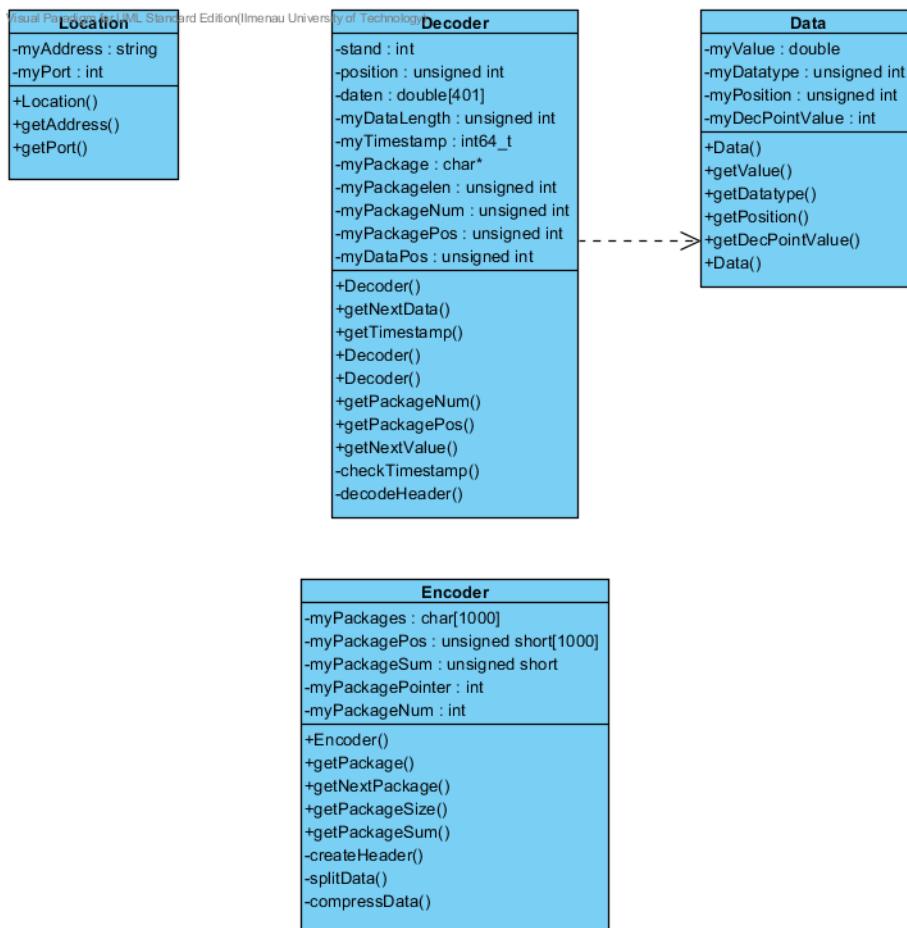


Abbildung A.14: Übersicht über die Kommunikation zwischen Embedded-PC und virtuellem Server



**Abbildung A.15:** Klassendiagramm über Datenübertragung und Verarbeitung

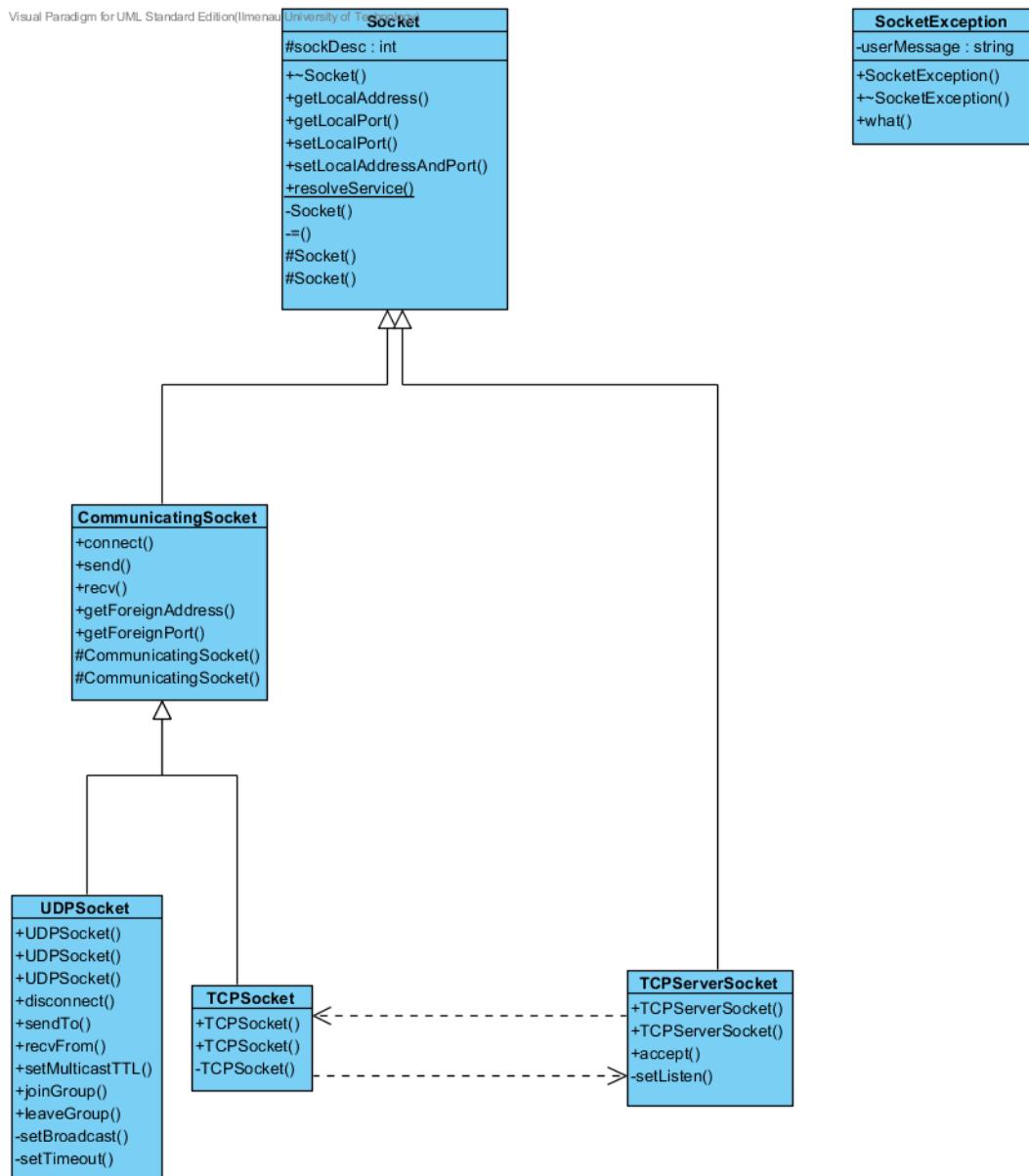


Abbildung A.16: Klassendiagramm über Socketerstellung und -benutzung