HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN
FACHHOCHSCHULE AMBERG - WEIDEN

FAKULTÄT ELEKTRO- UND INFORMATIONSTECHNIK

Studiengang Software - Systemtechnik

Diplomarbeit von

Tobias B E Y R L E

Entwurf und Realisierung einer Anbindung von Radarsensoren an eine ECU mittels SSC- und CAN Bus

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN
FACHHOCHSCHULE AMBERG - WEIDEN

FAKULTÄT ELEKTRO- UND INFORMATIONSTECHNIK

Studiengang Software - Systemtechnik

Diplomarbeit von

Tobias B e y r l e

Entwurf und Realisierung einer Anbindung von Radarsensoren an eine ECU mittels SSC- und CAN Bus

Gutachter:	**Prof. Dr. Alfred Höß**
Zweitgutachter:	**Prof. Wolfgang Schindler**
Bearbeitungsbeginn:	**01.04.2007**
Bearbeitungsende:	**14.03.2008**

Bestätigung gemäß § 31 Abs. 7 RaPO

Name und Vorname
der Studentin / des Studenten: Tobias Beyrle

Ich bestätige, dass ich die Diplomarbeit mit dem Titel:

Entwurf und Realisierung einer Anbindung von Radarsensoren an eine ECU mittels SSC- und CAN Bus

selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benützt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Datum:

Unterschrift:

Entwurf und Realisierung einer Anbindung von Radarsensoren an eine ECU mittels SSC- und CAN Bus

Diplomarbeit im Studiengang Software-Systemtechnik,
Fakultät Elektro- und Informationstechnik,
Hochschule für angewandte Wissenschaften
Fachhochschule Amberg-Weiden

Gutachter:	Prof. Dr. Alfred Höß
Zweitgutachter:	Prof. Wolfgang Schindler
Betreuer in Firma:	Dipl.-Ing.(FH) Marc Steuerer
Ausgabedatum:	01.04.2007 - Sommersemester 2007
Abgabedatum:	14.03.2008 - Wintersemester 2007/08

Zusammenfassung

Die Diplomarbeit beschreibt den Aufbau und die Umsetzung einer Software, die als Bestandteil eines bereits bestehenden Frameworks, Daten von 3 Radarsensoren per CAN- bzw. SSC-Bus überträgt und innerhalb des Frameworks z.B. der Datenverarbeitung zur Verfügung stellt. Dem Framework zu Grunde liegt das Echtzeitbetriebssystem OSEK/OS, das im Automotiveumfeld weit verbreitet ist. Die Hardware besteht aus einem von VDO Automotive entwickelten Steuergerät, auf dem ein Freescale MPC5200 (32-Bit, Power PC) Prozessor verbaut ist. Außerdem geht diese Arbeit auf die Grundlagen des SSC-Busses ein und beschreibt auch den CAN-Treiber der die Kommunikation mit den Radarsensoren ermöglicht. Am Ende dieses Dokuments sind Testfälle, Messungen und eine Bewertung des entwickelten Systems zu finden.

Schlüsselworte: ECU, Radarsensor, CAN-Bus, SSC-Bus

Inhaltsverzeichnis

KAPITEL 1

Einleitung

1.1. Motivation

Nach der Zielsetzung durch die Europäische Union, die Zahl der Verkehrsunfälle mit tödlichem Ausgang bis zum Jahr 2010 um 50 Prozent zu senken, wurde so genannten Fahrerassistenzsystemen eine große Bedeutung beigemessen, dieses ehrgeizige Ziel zu realisieren.

Das Bundesministerium für Bildung und Forschung (BMBF) initialisierte daraufhin unter anderem das Autosafe-Projekt, das zum Ziel hatte, ein System zu entwickeln, das den Autofahrer mit Hilfe von diversen Sensoren in kritischen Situationen unterstützen soll, um Unfälle zu vermeiden. Dabei finden neben Videokameras, Lasersensoren selbstverständlich auch Radarsensoren Verwendung. Die Radartechnologie bietet den Vorteil relativ unempfindlich gegenüber Witterungseinflüssen, wie z.B. Regen, Nebel und auch Lichteinflüssen zu sein (siehe [Sas05]). Es gibt derzeit bereits in Lkws und in der Mercedes S-Klasse der Daimler AG Radarsysteme, die unter der Bezeichnung Bremsassistent PLUS als Sonderausstattung geordert werden können. Allerdings ist dieses System komplett eigenständig, da die Verarbeitung der Daten direkt in den Sensoren passiert. Bei der hier gewählten Realisierung wird die Verarbeitung der Daten auf ein eigenes Steuergerät ausgelagert, auf dem man die Daten direkt mit anderen Applikationen verknüpfen kann. Dies führt zum Einen zu einer gesteigerten Leistungsfähigkeit, da die Hardware des Steuergerätes leistungsfähiger ist, als die im Sensor integrierte und zum Anderen, zu einer besseren Zugänglichkeit zu den Daten, was eine breitere Nutzung zulässt, z.B. für sog. Komfortsysteme oder andere Assistenzsysteme.

VDO Automotive AG hat einen BMW 5er als Testfahrzeug aufgebaut (siehe Abbildungen 1.1 , 1.2). In ihm sind hinter der vorderen Stoßstange die beiden Medium Range und der Long Range Radar Sensor verbaut. Im Kofferraum befinden sich vier PCs, die in Verbindung mit sog. Gateways teilweise die Funktionen erfüllen, die später von der ECU übernommen werden

sollen. Wie man auf den Bildern sehen kann (Abbildung 1.3), ist einiges an Hardware nötig um die gleiche Funktionalität auf PC-Basis im Fahrzeug bereitzustellen, wie es die hier beschriebene Applikation auf Embedded-Basis tut. Je ein PC ist für die Verarbeitung der Daten der vorderen und der hinteren Sensoren verantwortlich. Damit er die Daten der Sensoren, die per SSC-Bus übertragen werden empfangen kann, benötigt er ein Gateway. Dies ist ein elektronisches Gerät, das die Radar-Daten vom Sensor empfängt und sie über eine Ethernetverbindung und DLC-Protokoll an den PC weiterleitet (vlg. [Kla04]).

Abbildung 1.1.: Testfahrzeug und Cornerspiegel(Radarreflektor)

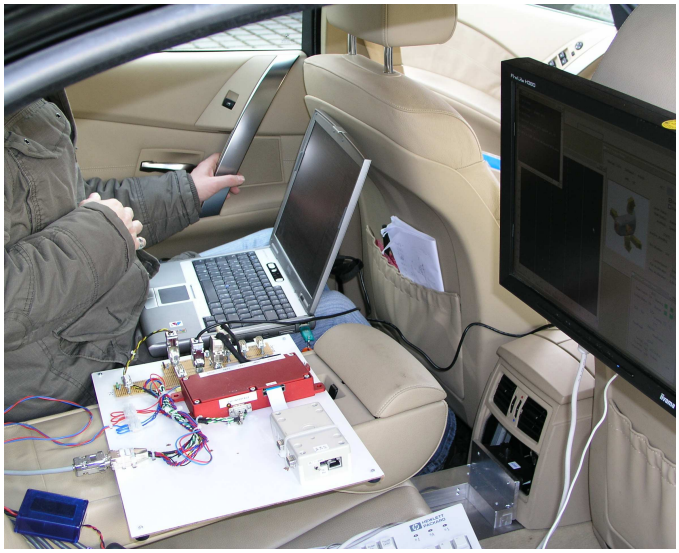

Abbildung 1.2.: Testfahrzeug (innen)

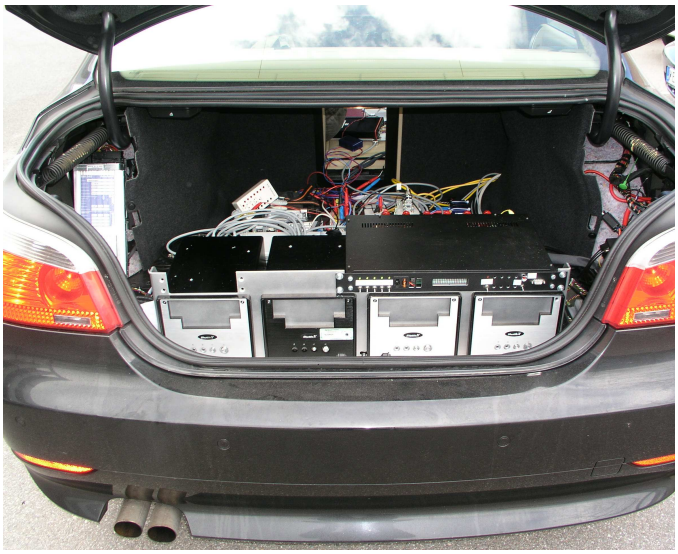

Abbildung 1.3.: Testfahrzeug Kofferraum

1.2. Projekt Autosafe

Ziel des Autosafe Projektes, in dessen Rahmen diese Diplomarbeit bearbeitet wurde, ist es, ein integrales Sicherheitssystem für Straßenfahrzeuge bereitzustellen, welches den Fahrer bei allen Phasen des Fahrens unterstützen soll. Abbildung 1.4 zeigt die einzelnen Phasen, die dabei eingesetzten Systeme und den Umfang, in welchem Ausmaß das integrale Sicherheitssystem Einfluss darauf nimmt.

Bei der Realisierung sollen Hardware- und Software-Module entwickelt werden, um später einzelne Komponenten in bestehende Fahrsicherheitssysteme integrieren zu können. Dabei sollten diese Module besonders leistungsstark, gut skalierbar und leicht in vorhandene Systeme integrierbar sein.

Entscheidend bei einem solchen System ist die Leistungsfähigkeit der eingesetzten Sensoren um die Umgebung zu erfassen. Da ein einzelner Sensorentyp viel zu einseitig und anfällig für Störungen ist, werden diverse Technologien und Sensortypen verwendet, wie z.B. Lidar, Radar und Videokameras. Jede Technologie liefert dabei unabhängig von den anderen, Informationen über die Umgebung. Werden alle gewonnen Sensordaten global ausgewertet, bietet das ein großes Potential im Hinblick auf die Leistungsfähigkeit des Sensorsystems.

Das Projekt, das vom Bundesministerium für Bildung und Forschung gefördert wird, ist auf drei Jahre angelegt. Dabei sind folgende Firmen Partner des Verbundprojektes Autosafe: VDO Automotive AG (ehemals Siemens VDO), Infineon Technologies AG, Siemens Restraint Systems GmbH und die Porsche AG. Weiterhin ist als Unterauftragsnehmer die HAW Amberg-Weiden am Projekt beteiligt.

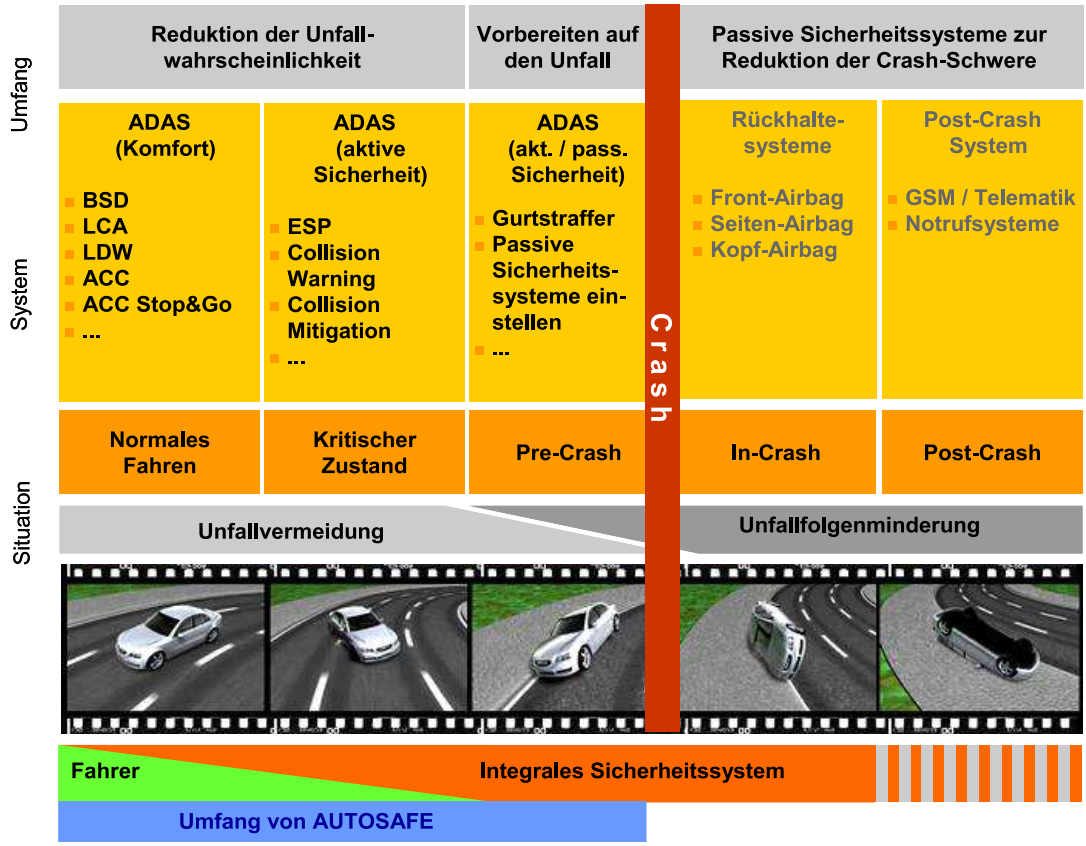

Abbildung 1.4.: Phasen des Fahrens

1.3. Aufbau der Arbeit

Die Arbeit gliedert sich in vier große Bereiche. Zu Beginn werden die Arbeitsmittel beschrieben. Dies umfasst neben der verwendeten Hard- und Software auch Bestandteile des Systems, wie die Radarsensoren und das Steuergerät (ECU). Außerdem werden die Bussysteme, die bei dieser Arbeit verwendet wurden, beschrieben.

Der zweite Teil, das Kapitel Grundlagen, beschäftigt sich erst mit der Zeitabschätzung der verschiedenen Kommunikationsschnittstellen und mit dem zu Grunde liegenden Framework, in das die Software integriert werden soll.

Der Hauptteil der Arbeit behandelt die Funktionsweise und die Umsetzung der entwickelten Software. Dabei wird der Aufbau und der Ablauf ebenso beschrieben, wie wichtige Klassen und die Fehlerbehandlung.

Im letzten Kapitel dieser Arbeit wird die gemessene Leistung des Systems beurteilt, aufgetretene Probleme erläutert und Maßnahmen zu weiteren Leistungssteigerung vorgestellt.

KAPITEL 2

Hardware

2.1. Tools

In diesem Abschnitt werden verwendete Tools und Hilfsmittel vorgestellt, die zum Gelingen der Arbeit beigetragen haben.

2.1.1. CodeWarrior

Die CodeWarrior Entwicklungsumgebung wurde ursprünglich von der Firma Metrowerks entwickelt. 1999 wurde diese Firma von Motorola aufgekauft und später zusammen mit ihrer Halbleitersparte ausgelagert. Das damit neu gegründete Unternehmen Freescale, entwickelt weiter unter dem Namen Codewarrior, Entwicklungsumgebungen. Für diese Arbeit wurde die Version 5.70 des Codewarriors verwendet (siehe Abbildung 2.1). Er vereint Projektmanager, Source Browser, Editor, Debugger, Kompiler in einer Multi-Window Anwendung. Mit dieser IDE ist es möglich sowohl Software für den Embedded-Bereich bis hin zu Desktopanwendungen zu programmieren. Ausgelegt ist er jedoch für das Programmieren von Mikrocontrollern. Die Entwicklungsumgebung unterstützt dabei diverse Plattformen, wie z.B. den MCP5200 auf PowerPC-Basis und auch mehrere Programmiersprachen. Zusätzlich zu seiner Standardfunktionalität, lässt sich das Entwicklungswerkzeug durch diverse Plug-Ins in seinem Umfang erweitern, z.B. durch Erweiterungen für OSEK und Linux Unterstützung (siehe [Fre07]).

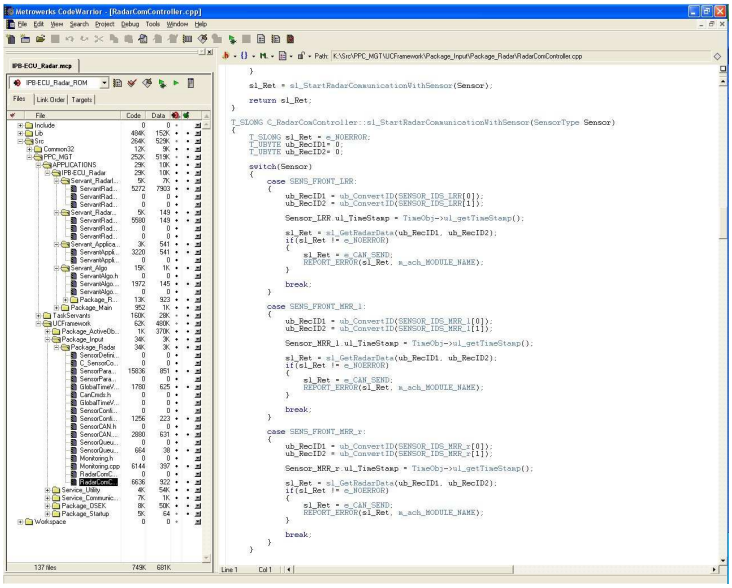

Abbildung 2.1.: CodeWarrior Entwicklungsumgebung von Freescale

2.1.2. BDI / BDI-Flasher

Das BDI-2000 von Abatron (Abbildung 2.2) ist ein On-Chip Debugger, bzw. On-Chip Emulator. Verbunden wird es über die JTAG-Schnittstelle mit der Ziel-Hardware und seriell oder per Ethernet mit dem PC. Das BDI unterstützt diverse Mikroprozessoren, wie z.B. den hier verwendeten MCP5200 PowerPC von Freescale. Der Vorteil gegenüber einem Software-Debugger ist, dass auch nach einem Fehler bzw. einem Systemcrash die Möglichkeit besteht, die Vorgänge nachzuvollziehen, die zu dieser Situation geführt haben.

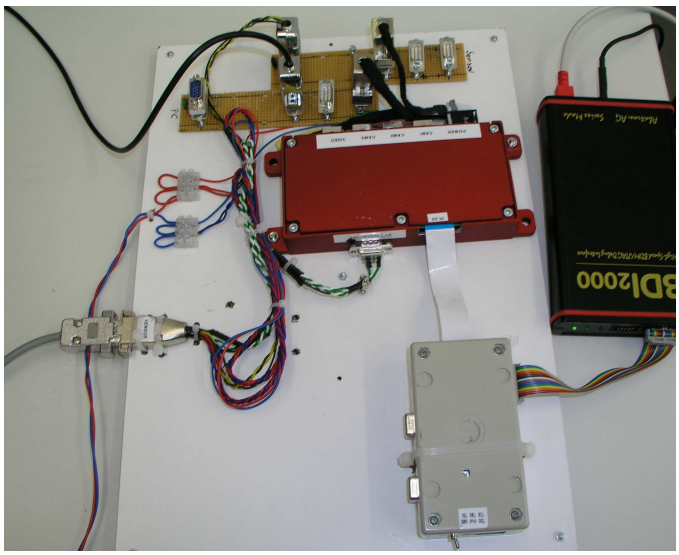

Abbildung 2.2.: Testaufbau mit ECU(rot), BDI(schwarz) und Peripherie

Von VDO Automotive wurde passend zum BDI ein Softwaretool bereit gestellt (Abbildung 2.3), mit dem man z.B. den Flash-Speicher auf der ECU bereinigen oder auch einen ROM-Version der Software auf die Hardware aufspielen kann. Außerdem bietet dieses Tool die Möglichkeit, Konfigurationsdateien auf die ECU zu schreiben, die dann von der Software im Betrieb ausgelesen werden können (siehe [ABA]).

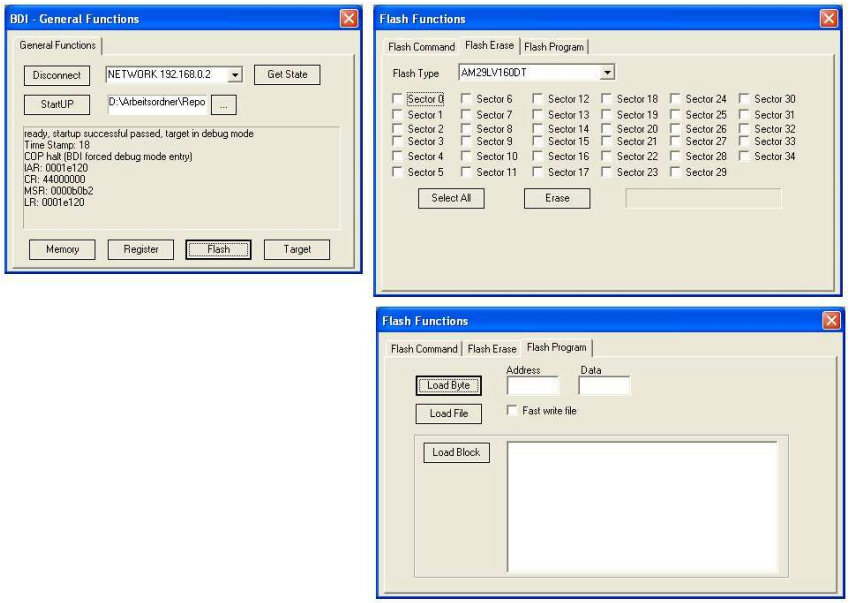

Abbildung 2.3.: BDI Software Tool

2.1.3. PCAN / Softing

Da man an einem Standard PC keine CAN-Bus Schnittstelle vorfindet, war es nötig diese nachzurüsten. Dies ist relativ einfach möglich, da es von diversen Herstellern CAN-Interfaces gibt, die man z.B. über USB mit dem PC verbinden kann. Während dieses Projekts kamen drei verschiedene Varianten von PC-CAN Interfaces zum Einsatz.

- PC-USB zu CAN Interface der Firma Peak (einkanalig)
- CANusb Interface der Firma Softing (einkanalig)
- PCMCIA CANCard der Firma Softing (zweikanalig)

Abbildung 2.4.: PC-USB zu CAN Interface der Firma Peak

Abbildung 2.5.: CANusb Interface der Firma Softing

Abbildung 2.6.: PCMCIA CAN-Card der Firma Softing

Zum PC-USB zu CAN Interface der Firma Peak gibt es ein kleines Software Tool, das es erlaubt relativ einfach einzelne CAN-Nachrichten zu verschicken und zu empfangen. Es bietet jedoch keine Möglichkeit eine komplexere Kommunikation zwischen zwei oder mehr Teilnehmern aufzuzeichnen. Aus diesem Grund wurde diese Hardware auch nur dafür benutzt Kommandos an die ECU zu schicken oder am Anfang des Projektes mit den Radarsensoren zu kommunizieren. Da es aber auch komplexeren Nachrichtenaustausch über den CAN-Bus gibt, z.B. bei der Konfiguration der Radarsensoren, mussten auch noch andere Hilfsmittel eingesetzt werden. Dazu wurde die Hardware der Firma Softing verwendet, im Zusammenspiel mit einem Programm der Firma Vector, namens CANalyzer, das im Automotive-Bereich weit verbreitet ist.

2.1.4. CANalyzer

Die CANalyzer Software der Firma Vector ist ein umfangreiches Analysewerkzeug, um den Datenverkehr auf diversen Bussystemen zu bestimmen. Während der Entwicklung wurde es ausschließlich für die Auswertung der Kommunikation des CAN-Busses verwendet. Ohne die Möglichkeit die genaue Abfolge der Pakete auf dem Bus zu analysieren, wäre die Entwicklung einer Klasse, die die Kommunikation zwischen ECU und Radarsensor bewerkstelligt, nahezu unmöglich gewesen. Auch im späteren Verlauf der Arbeit war es von Vorteil die Kommunikation zwischen ECU und der Anwendersoftware analysieren zu können. (vgl. [Vec])

2.1.5. Filterterminal

Das ADAS-Framework bietet die Möglichkeit, Nachrichten über eine serielle Schnittstelle an der ECU zu verschicken. Diese Nachrichten lassen sich direkt im Quellecode erzeugen. Dazu muss allerdings die Headerdatei *DebugSerialLevel7.h* eingebunden sein. Verwenden kann man dies sehr gut, um Fehler oder Stati zu melden. Um die Nachrichten am PC empfangen und lesen zu können, muss dort ein Programm laufen, dass die serielle Schnittstelle abhört. Während dieses Projektes wurde dazu das Programm Filter Terminal in der Version 0.95195 verwendet (siehe Abbildung 2.7).

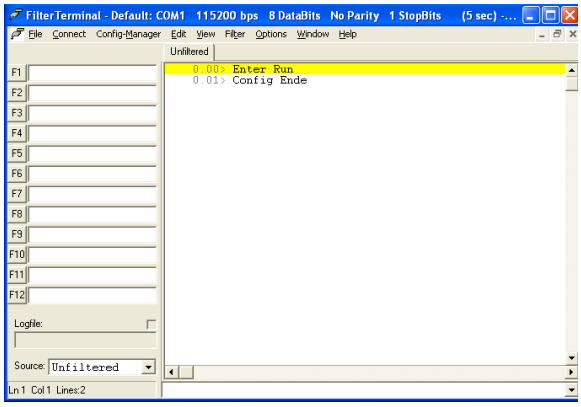

Abbildung 2.7.: Filter Terminal Programm

2.2. ECU

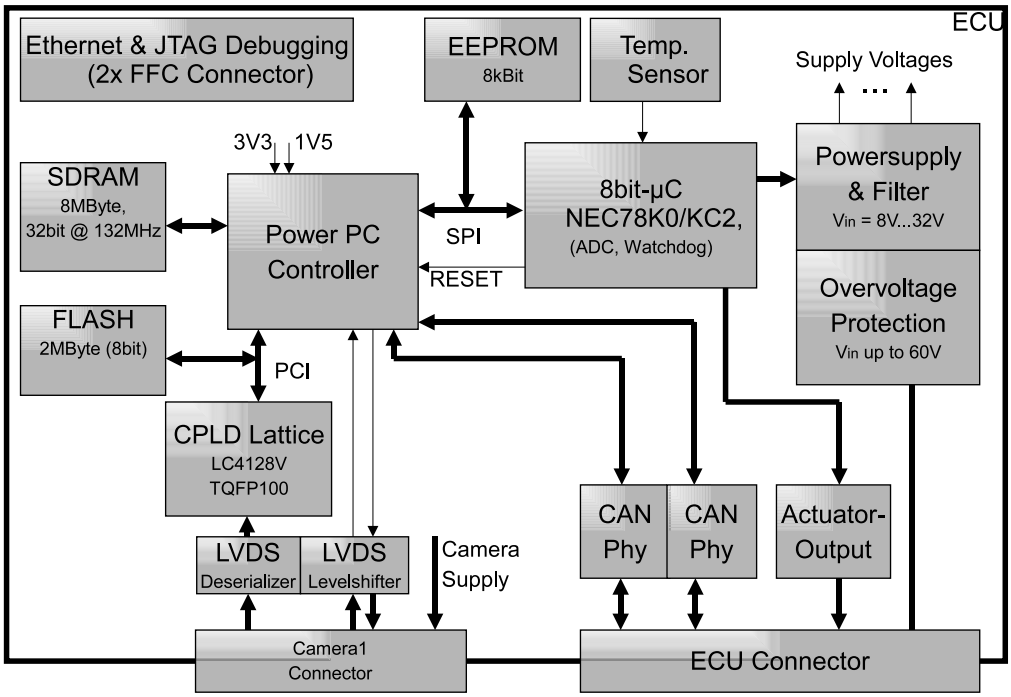

Abbildung 2.8.: Blockschaltbild der ECU

Der MCP5200 von Freescale (ehemals Motorola) ist ein 32-Bit MIPS Prozessor. Er basiert auf der PowerPC Architektur und ist speziell für den Einsatz im Fahrzeug modifiziert (spezifiziert für eine Umgebungstemperatur von -40 bis +85°C). Er arbeitet mit einer Taktgeschwindigkeit von 400 MHz und kann sowohl SDRAM als auch DDR SDRAM mit 133 MHz anbinden. Außerdem verfügt er über diverse Schnittstellen wie z.B. USB, ATA, Ethernet, I²C, JTAG und zweimal CAN-Bus. Ein CAN-Bus wird dazu verwendet, um zwischen Radarsensoren und ECU zu kommunizieren und der weite, um die ECU zu steuern.

Der PowerPC MCP5200E hat werksseitig Fehler in seinem Programmable Serial Controller (PSC). Beim PSC handelt es sich um eine serielle Schnittstelle des Mikrocontrollers, die je nach Anwendungsfall ein anderes Protokoll unterstützt. Die Protokolle, die vom MCP5200 unterstützt werden sind: I^2C, UART, IrDA, Codec, Codec-SPI und AC97. Die Daten der ersten Übertragung, die er empfängt sind nicht korrekt. Dies wurde bei der Implementierung bedacht und dadurch auftretende Probleme behoben. (siehe [Fre04])

2.3. Radarsensoren

In den nächsten beiden Abschnitten werden die Radarsensoren, die bei diesem Projekt verwendet werden vorgestellt und auch mit anderen Sensoren, die für ähnliche Aufgaben verwendet werden verglichen.

2.3.1. Verwendete Sensoren

Abbildung 2.9.: Medium Range Radar Sensor

Das System arbeitet mit drei Sensoren im 24GHz Band, die keinerlei Vorverarbeitung vornehmen, sondern die Radar-Rohdaten ausgeben. Bei diesen Sensoren handelt es sich um Vorserienmodelle. Die Sensoren, die im Testfahrzeug verbaut sind, wurden getestet und liefern gute Ergebnisse. Da das Testfahrzeug für den Entwicklungszeitraum für andere Nutzer weiter verfügbar sein musste, wurden der FH zwei mittel weitreichende Sensoren zur Verfügung gestellt. Ein weitreichender Sensor war nicht verfügbar.

Die beiden mittel-weitreichenden Sensoren sind in Höhe der vorderen Stoßstange links und rechts am Fahrzeug angebracht und ein weitreichender Sensor ist mittig am Fahrzeug in gleicher Höhe montiert. Jeder Sensor verfügt über 2 Receiver, d.h. über zwei Antennenarrays, die die reflektierten Radarwellen auffangen, die vom Sensor ausgesendet werden. Diese empfangenen Wellen werden mittels A/D-Wandler in 16 Bit Werte gewandelt. Bei einem solchen Wert spricht man auch von Sample. Pro Messvorgang werden 256 Sample erzeugt, die vom Sensor nach Anforderung, über den SSC-Bus an die ECU geschickt werden.

Die beiden Sensoren mit mittlerer Reichweite nehmen bei einer Bandbreite von 500 MHz, Objekte bis 35m Entfernung wahr. Der weitreichende Sensor detektiert Objekte in einer Entfernung bis zu 100m. Dabei ist die Entfernungsauflösung, also die Genauigkeit im Bezug auf die Entfernung, auf wenige Zentimeter genau (siehe [Sie06]).

2.3.2. Vergleich zu anderen Sensoren

Bisher werden meist Sensoren der 77GHz Klasse im Fahrzeugbereich eingesetzt. Die Continental AG bietet solche Sensoren mit integrierter Verarbeitung bereits an (vgl. [Con]). Nachteil dieses Systems gegenüber dem hier entwickelten, ist in erster Linie der Preis. Als weiterer Vorteil ist anzuführen, dass auf der hier verwendeten Hardware nicht nur die Radarapplikation ausgeführt werden kann, sondern gleichzeitig auch noch andere Komponenten des Advanced Driver Assistent System (ADAS). Dies führt zu einem hoch integrierten Sicherheitssystem, das flexibel und schnell auf unterschiedlichste Sensorinformationen reagieren kann. Zusätzlich ist es leicht an neue Gegebenheiten, wie z.B. neue Sensortechnologie oder bessere Auswertealgorithmen anzupassen. Das Gesamtsystems ist außerdem gut skalierbar. Dies bedeutet, dass einzelne Komponenten weggelassen werden können.

2.4. Bus-Systeme

Wichtige Bestandteile der Applikation sind die zwei verschiedenen Bus-Systeme, die unterschiedliche Aufgaben haben. In den nächsten Abschnitten werden diese näher erläutert.

2.4.1. CAN

Der CAN-Bus ist ein von Bosch entwickeltes Multimaster-Bus-System für die Vernetzung von Steuergeräten (im Auto). Dabei handelt es sich um einen asynchronen, seriellen Bustyp, der nach dem CSMA/CR Verfahren arbeitet. Kollisionen auf dem Bus werden durch eine Bitarbitrierung vermieden. Eine CAN-Nachricht beginnt immer mit ihrer ID. Je niedriger die ID, desto höher ihre Priorität. Versuchen zwei oder mehr Teilnehmer gleichzeitig zu senden, setzt sich folglich der Teilnehmer, der die Nachricht mit der niedrigsten CAN-ID versendet. Unterbrochen werden kann das Senden einer Nachricht nicht. Eine normale Nachricht kann bis zu acht Byte Daten beinhalten. Typischerweise wird der CAN-Bus mit zwei Drähten (und zusätzlicher Masseleitung) ausgeführt, auf denen komplementäre Signale gesendet werden, um Gleichstromstörungen zu vermeiden.

In Fahrzeugen wird der Bus normalerweise mit einer Übertragungsgeschwindigkeit von 500kbit/s betrieben, was eine theoretische Leitungslänge von 100m entspricht. Da es bei dieser Applikation maßgeblich auf Schnelligkeit ankommt, findet die Kommunikation mit den Radarsensoren bei 1Mbit/s statt, was die maximale Übertragungsgeschwindigkeit darstellt. Die maximale Leitungslänge verringert sich dadurch auf 40m. Diese Grenzen ergeben sich daraus, dass je größer die Übertragungsgeschwindigkeit ist, desto kürzer liegt das Signal auf dem Bus (Bit-Zeit; Bit/Sekunde). Mit zunehmender Leitungslänge steigt jedoch die Zeit, die ein Signal

braucht, bis es am anderen Ende des Busses angekommen ist (Ausbreitungsgeschwindigkeit). Das Signal muss also länger auf dem Bus liegen, als es braucht um sich über den gesamten Bus auszubreiten, denn der Empfänger braucht auch etwas Zeit, um auf ein Signal zu reagieren.

Da der CAN-Bus weit verbreitet und an anderer Stelle (siehe Literaturverzeichnis) schon mehrfach ausführlich beschrieben wurde, wird hier auf eine weitere Beschreibung verzichtet. (siehe [K. 99], [ISO90], [ISO99])

2.4.2. SSC

Beim SSC-Bus handelt es sich um einen seriellen Bus, der synchron zu einem Clock-Signal Daten sendet, bzw. empfängt. Dabei erfolgt die Übertragung der Daten über ein mindestens vieradriges Twisted-Pair-Kabel, wobei immer je zwei miteinander verdrillte Adern für die Daten und die Clock-Impulse verwendet werden. Da die Signale auf diesen beiden Adernpaare immer symmetrisch gesendet werden, hat der SSC-Bus eine sehr geringe Störempfindlichkeit. Die Grundspannung auf den beiden Clock- bzw. Datenleitungen beträgt 2 Volt. Der Spannungsunterschied zwischen high und low 1 Volt.

Das Bussystem arbeitet nach dem Master-Slave-Prinzip, d.h. immer nur ein Teilnehmer (der Master) kann senden. Im hier beschriebenen Anwendungsfall ist immer nur ein Radarsensor Master und die ECU arbeitet als Empfänger, also Slave. Welcher Sensor als Master fungiert, wird zuvor explizit per CAN-Nachricht festgelegt.

Da der Bus mit 10 MHz getaktet ist und nur sehr wenig Protokoll-Overhead hat, erreicht er einen sehr hohen Datendurchsatz, was in diesem Anwendungsfall von hoher Bedeutung ist. Die Sensor Daten, Protokoll-Overhead und Headerdaten, die pro Übertragungsvorgang gesendet werden, belaufen sich auf 8224 Bit. Bei einer Bitrate von 10MBit/s dauert eine Übertragung rechnerisch 860μs. Pro Sensor sind jedoch zwei Übertragungen nötig. Das bedeutet, dass über CAN der zweite Receiver des Sensors zum Master auf dem SSC-Bus gemacht werden muss. Dies quitiert der Receiver mit einer ACK-Meldung bevor die ECU den Receiver zum Senden veranlässt, was wieder per CAN-Nachricht geschieht.

Gesendet werden auf dem SSC-Bus keine Pakete oder Nachrichten, sondern ein kontinuierlicher Bit-Stream. Es ist also unabdingbar zu wissen, wie die ankommenden Bit zu interpretieren sind. Der Sensor schickt eigentlich Word (also 16 Bit Werte), aber zur einfacheren Verarbeitung, werden sie auf der ECU als zwei Byte, also (8 Bit Werte) interpretiert (siehe dazu auch Abschnitt 4.1.2).

Das Protokoll, das zum Einsatz kommt, ist sehr rudimentär. Die ersten beiden Byte einer Übertragung haben die festen Werte 0xB3 und 0x85 und werden als Startsequenz bezeichnet. Anschließend folgen 24 Byte Header-Daten bevor 512 Byte Nutzdaten des Up-Sweep gesendet werden. Direkt im Anschluss kommen 24 Byte Header-Daten und 512 Byte Nutzdaten des Down-Sweep. Die letzten beiden Byte dienen als Stopsequenz. Sie haben die Werte 0xA1 und 0xCD.

Die Header-Daten werden nicht ausgewertet, da die darin enthaltenen Informationen nicht benötigt werden.

KAPITEL 3

Grundlagen

3.1. Zeitabschätzung

Bevor man mit der Implementierung beginnen kann, ist es nötig, Flaschenhälse im System zeitmäßig abzuschätzen. In dieser Applikation ist der Datenaustausch zwischen Sensoren und ECU die Engstelle. Aus diesem Grund wird eine Abschätzung der Leistungsfähigkeit vorgenommen.

3.1.1. SSC-Kommunikation

Bei der Datenübertragung eines Receivers werden zwei Byte Startsequenz, 24 Byte Header (Up-Sweep), 512 Byte Daten (Up-Sweep), 24 Byte Header (Down-Sweep), 512 Byte Daten (Down-Sweep) und zwei Byte Stopsequenz übertragen. Pro Kommunikationsvorgang werden 1076 Byte = 8608 Bit. Bei einer Übertragungsgeschwindigkeit von 10MBit/s, mit der der SSC-Bus betrieben wird, ergibt sich eine rechnerische Übertragungsdauer von 860 μs.

Messungen haben diese berechneten Ergebnisse bestätigt (siehe Abbildung 3.1). Die Übertragung von zwei Receivern, also einem Sensor, war mit einer gemessenen Dauer von zwei ms ebenfalls im Bereich dessen, was zuvor berechnet wurde.

Die Buslast beträgt ca. sechs Prozent. Annahme ist, dass alle 30 ms von einem Sensor Daten übertragen werden, d.h. bei drei Sensoren sendet jeder Sensor 11x pro Sekunde.

$$1 Sekunde / 30 ms = 33$$

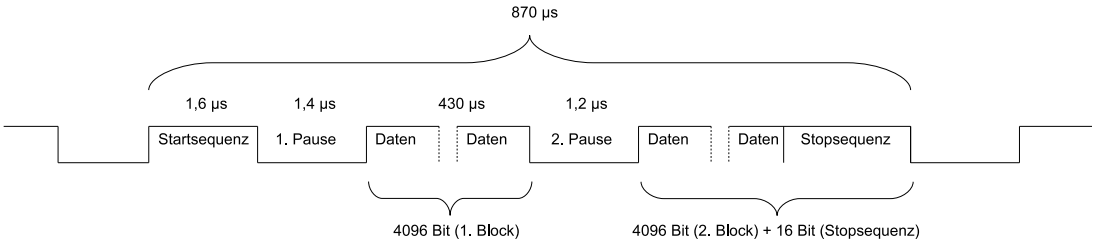

Abbildung 3.1.: SSC-Übertragung für einen Receiver

Wenn man die Buslast auf Basis der Übertragung eines Receivers berechnet, kommt man auf folgendes Ergebnis: Es finden ca. 66 Übertragungen mit 8608 Bit statt.

$$66 \cdot 8608Bit = 568.128Bit$$

Die Übertragungsrate beträgt 10 MBit/s, also 10.000.000 Bit/s. Daraus ergibt sich eine Buslast von

$$568.128Bit/10.000.000Bit/s = 0,057s$$

, was 5,7 Prozent entspricht.

3.1.2. CAN-Kommunikation ECU-Sensor

Gesteuert wird der Radarsensor über CAN-Nachrichten. Um Radardaten vom Sensor zu erhalten, müssen fünf Nachrichten mit je vier Datenbyte an den Sensor geschickt und zwei ACK-Nachrichten vom Sensor erhalten werden. Eine ACK-Nachricht besitzt nur ein Datenbyte. Bei der hier verwendeten Übertragungsgeschwindigkeit von 1 MBit/s ist die minimale Übertragungsdauer von CAN-Nachrichten mit vier Datenpaketen $100\mu s$ (vgl. [IXX]). Wie im vorherigen Abschnitt wird angenommen, dass alle 30 ms die Daten eines Sensors abgerufen werden, was zu 33 Übertragungen pro Sekunde führt. Zur Vereinfachung der Abschätzung wird angenommen, dass auch die ACK-Nachrichten 100 μs Übertragungszeit benötigen Pro Übertragung ist der Bus dabei

$$6 \cdot 100\mu s = 600\mu s$$

belegt. Jede Sekunde werden in einem Zeitraum von

$$600\mu s \cdot 33 = 19,8ms$$

Daten über den Bus geschickt. Daraus ergibt sich eine Buslast von 1,98 Prozent.

$$19,8ms/1s = 0,0198$$

3.1.3. CAN-Kommunikation ECU-PC

Derzeit wird auf der ECU Peak . Dies soll erst in den nächsten Ausbaustufen der Software geschehen. Die Daten, die von der ECU übertragen werden, sind derzeit noch Peaklisten. Da diese Peaklisten aber immer unterschiedlich groß sind, werden unterschiedlich viele Nachrichten über den CAN-Bus übermittelt, was eine Zeit- und Lastabschätzung etwas schwierig macht. Aus diesem Grund wird angenommen, dass durchschnittlich sechs Peaks, sowohl im Up- als auch im Down-Sweep pro Sensor erkannt werden.

Das Senden der Peaklist besteht aus drei verschiedenen CAN-Nachrichten Typen:
(siehe auch Anhang A, oder Abbildung 3.2)

CMD_PEAKLIST_NOF_PEAKS	CAN-ID 7D1
CMD_PEAKLIST_UP_PEAKS	CAN-ID 7D2
CMD_PEAKLIST_DOWN_PEAKS	CAN-ID 7D3

CAN-ID	Anzahl Datenbyte	Datenbyte 1	Datenbyte 2	Datenbyte 3	Datenbyte 4	Datenbyte 5	Datenbyte 6	Datenbyte 7
7D1	7	0x03	0x03	0x01	0xCC	0x07	0x00	0x00
7D2	6	0x60	0x00	0x17	0x01	0xC0	0x02	
7D2	6	0xBE	0x00	0x75	0x81	0xA7	0x01	
7D2	6	0xD6	0x00	0x11	0x81	0xCB	0x01	
7D3	6	0x60	0x00	0x17	0x01	0xC0	0x02	
7D3	6	0xBD	0x00	0x7C	0x81	0xB3	0x01	
7D3	6	0xD7	0x00	0x63	0x01	0xC3	0x01	

Abbildung 3.2.: Peaklisten per CAN-Nachrichten verschicken

Die Peaklisten-Übertragung eines Sensors, der sowohl im Up- als auch im Down-Sweep sechs Peaks detektiert hat, besteht aus einer Nachricht mit sieben Daten Byte und 12 Nachrichten mit je sechs Daten Byte. Die Minimaldauer für die Übertragung von Nachrichten mit acht Datenbyte bei 1MBit/s ist 139μs (vgl. [IXX]). Dieser Minimalwert wird allen CAN-Nachrichten, auch wenn sie ein oder zwei Datenbyte weniger haben zugrunde gelegt. Für die Übertragung der Peaklist eines Sensors benötigt die ECU also ca.

$$139\mu s \cdot 13 = 1,81ms$$

33 Sensordaten werden pro Sekunde ausgewertet und übertragen, was zu einer Sendezeit von

$$33 \cdot 1,81ms = 59,73ms$$

pro Sekunde auf dem CAN-Bus führt. Das entspricht einer Buslast von 5,97 Prozent.

3.2. Radarsensoren

Damit die Radarsensoren Daten liefern, die man verarbeiten kann, muss jeder Sensor konfiguriert werden, bevor er verwendet wird. Einige Parameter die im Anhang D beschrieben sind, bzw. im Headerfile *C_SensorConfigData.h* zu finden sind, müssen aus diesem Grund an die Sensoren geschickt werden. Allerdings werden manche Werte im Vorfeld mit einem Faktor multipliziert (siehe Tabelle). Um die Parameter zu übertragen wird der CAN-Bus verwendet. Wie die einzelnen Nachrichten konkret aussehen, wird im Kapitel 4.5 beschrieben.

SIGNAL_LENGTH	ohne Faktor	Anzahl der Samples
ADC_CHANNEL	ohne Faktor	Nummer das A/D-Wandlers
TAKT_DELAY	Faktor 10^6	Zeit in Mikrosekunden
ADC_SAMPLE_TIME	Faktor $25 \cdot 10^9$	Zeit in Nanosekunden
START_DELAY	Faktor $6,4 \cdot 10^6$	Zeit in Mikrosekunden
PWM_DELAY	Faktor $4 \cdot 10^6$	Zeit in Mikrosekunden
SAMPLE_DELAY	Faktor $12,75 \cdot 10^6$	Zeit in Mikrosekunden
HF_CLOCK_PERIOD	Faktor $25 \cdot 10^9$	Zeit in Nanosekunden
HF_GATE_WIDTH	Faktor $25 \cdot 10^9$	Zeit in Nanosekunden
HF_CLOCK_MODE	ohne Faktor	
LIN_MODE	ohne Faktor	Linearisierungsmodus
DEBUG_VARIABLES	ohne Faktor	

3.3. OSEK / ADAS-Framework

Als Basis läuft auf der ECU ein Echtzeitbetriebssystem namens OSEK. Darauf aufgesetzt steht das ADAS-Framework dem Programmierer zu Verfügung. In den folgenden beiden Abschnitten wird dieses Kombination vorgestellt.

3.3.1. OSEK Betriebssystem

Das OSEK Betriebssystem (siehe [OSE]) ist ein Echtzeitbetriebsystem, das von vielen führenden deutschen und französischen Automobilherstellern zusammen mit der Robert Bosch AG und der Siemens AG ins Leben gerufen wurde. Das Haupteinsatzgebiet dieses Betriebssystems ist der Embedded-Bereich mit Schwerpunkt Automotive. Das OSEK Betriebsystem benötigt minimale Systemressourcen und bietet Portabilität von Softwaremodulen auf Quellcode Ebene. Dies ist nur über definierte Schnittstellen und Hardewareabstraktion möglich, die es für fast jeden Mikrocontroller im Automotive-Bereich gibt. Darüber hinaus unterstützt es das Multitasking, Interruptbehandlung, Versand von Nachrichten, Alarme, Ereignisse und mehrere Betriebsmodi.(siehe [Wer04])

OSEK Bestandteile:

OSEK OS: Betriebssystem für eingebettete Systeme
OSEK COM: Kommunikation zwischen Programmteilen
OSEK NM: Überwachung, Konfiguration und Verwaltung eines Netzwerks
OSEK OIL: Beschreibungssprache für Betriebssystemobjekte

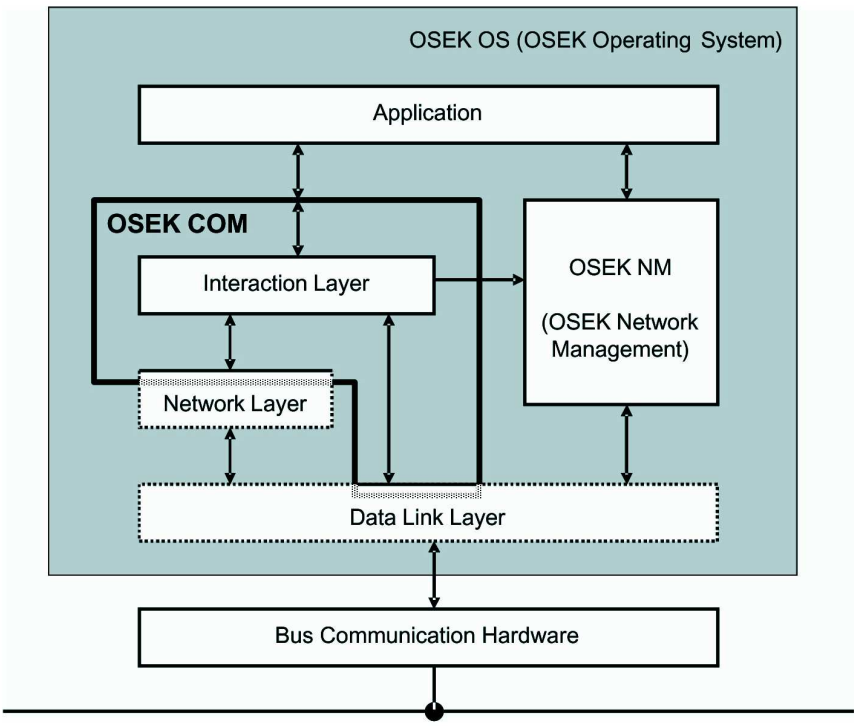

Abbildung 3.3.: OSEK Aufbau

3.3.2. ADAS-Framework

Das ADAS Framework setzt auf dem OSEK Betriebssystem auf und ist größtenteils in C++ programmiert. Es benutzt in erster Linie das Design Patter Active Object (siehe [R.]), das hier dazu verwendet wird, Nachrichten zwischen Tasks auszutauschen ohne den Mechanismus, der im OSEK Betriebssystem eingebaut ist, zu verwenden. Außerdem kapselt es Funktionalität hinter einem Interface, so dass diese wiederverwendbar werden. Das Framework besteht aus diversen Klassen, die die zugrunde liegende Hardware abbildet und deren Funktionalität dem Anwender zur Verfügung stellt.

Da das ADAS-Framework auf unterschiedlicher Hardware betrieben werden können soll, hat man sich dazu entschlossen, nicht die Standardvariablentypen der Programmiersprache wie Integer, Double oder Float zu verwenden, sondern eigene Datentypen zu definieren. Dies hat den Vorteil, dass die neu definierten Datentypen auf jeder Hardware so angepasst werden können, dass sie die selbe Anzahl an Bit haben, bzw. den gleichen Wertebereich abdecken.(siehe [M. 03a]).

Auch für das Design, bzw. der Aufbau der Funktionen folgt eigenen Regeln (siehe [Rei03]). Besonders zu erwähnen ist, dass alle Funktionen schon durch ihren Namen angeben, was für ein Variabletyp ihr Rückgabewert ist. Die Funktion *v_ConvertTaktPeriode* gibt nichts zurück, da ihr Rückgabewert vom Typ *void* ist. Im Gegensatz dazu gibt die Funktion *sl_Reset* einen Wert vom Typ T_SLONG zurück. Zu erkennen ist das an dem Buchstaben, oder der Buchstabenkombination ganz zu Beginn der Funktion, vor dem Unterstrich. Auch Variablen werden auf diese Art

gekennzeichnet. Falls es sich um eine private Variable einer Klasse handelt ist diese zusätzlich mit einem kleinen m und einem Unterstrich markiert, wie z.B. die Variable *m_sl_MemoryAdr*.

3.3.3. ADAS-Message

Die ADAS-Message ist eine vom Framework bereitgestellte Funktionalität, um zwischen verschiedenen Tasks, Informationen auszutauschen. Folgender Condeausschnitt zeigt alle nötigen Schritte, um eine ADAS-Message zu erzeugen und zu verschicken.

Listing 3.1: Erstellen und Versenden einer ADAS-Message

```
T_SLONG C_ServantApplication::sl_StartSingleStep(SensorType Sensor)
{
  T_SLONG sl_Ret = e_NOERROR;
  m_SensorCmd = Sensor;

  C_ADASMessage ADASMsg;
  C_AOProxy &rc_Proxy = C_ProxyFactory::rc_GetProxy(C_ProxyFactory::e_RADAR_INPUT);

  ADASMsg.v_CreateMessage(      C_ADASMessage::e_USER_CMD,
                                0,
                                S_ServantRadarInputCmd::e_START_SINGLE_STEP,
                                (T_VOID*) &m_SensorCmd);

  sl_Ret = rc_Proxy.sl_SendMessage(ADASMsg);
  if(sl_Ret != e_NOERROR)
  {
              REPORT_ERROR(sl_Ret, "APLLICATION_TASK_ERROR:_StartSingleStep");
  }
  return sl_Ret;
}
```

Zuerst wird ein Objekt vom *Typ C_ADASMessage* angelegt. Jeder Task hat einen Proxy, an den die Nachrichten für den entsprechenden Task adressiert werden. Deswegen wird anschließend eine Referenz auf diesen Proxy erzeugt. Bisher enthält die angelegte ADAS-Message keinerlei Daten. Durch die Funktion *v_CreateMessage* wird diese leere Hülle nun mit Inhalt versehen. Der erste Parameter gibt den Typ der ADAS-Message an. In diesem Projekt werden ausschließlich Nachrichten vom Typ *e_USER_CMD* erzeugt. Es gibt aber auch Nachrichten vom Typ *e_SYSTEM_CMD*, die vom Framework selbst erzeugt werden, um Daten zwischen Framework-Tasks auszutauschen.

Bei ADAS-Messages unterscheidet man zwischen synchronen und asynchronen Nachrichten. Der Unterschied zwischen beiden Arten ist, dass bei synchronen Nachrichten der aktuelle Task angehalten wird, bis bei ihm wieder eine ADAS-Message eintrifft, die ihn mit seinen Aufgaben fortfahren lässt. Asynchrone Nachrichten werden nur an den Empfängertask verschickt, jedoch wird der versendende Task nicht unterbrochen, sondern läuft weiter. Bei der Umsetzung dieser Diplomarbeit fanden nur asynchrone ADAS-Messages Verwendung. Sollte man eine synchrone Nachricht erzeugen wollen, also an dieser Stelle warten wollen, bis eine Antwort auf die Nachricht eingegangen ist, muss man als zweiten Parameter den eigenen Proxy mit übergeben. Diese Funktion wird in diesem Fall nicht verwendet. Die hier erzeugte Nachricht enthält das Kommando *e_START_SINGLE_STEP*, das im struct *S_ServantRadarInputCmd* definiert ist (siehe auch Anhang A).

Listing 3.2: ADAS-Message Subcommands des Input-Task

```
struct S_ServantRadarInputCmd
{
        // available subcommands
        enum E_SUBCOMMAND_INPUT
        {
                e_RADAR_DATA_COMPLETE           = 100,
                e_START_TRANSMISSION            = 110,
                e_CONFIG_SENSOR                 = 120,
                e_GET_SENSOR_DATA               = 130,
                e_START_DUMMY_RUN               = 140,
                e_START_NEXT_RADAR_COM          = 150,
                e_START_SINGLE_STEP             = 160,
                e_SEND_SENSOR_CONFIG            = 170,
                e_FREE_MEMORY                   = 180,
                e_RESET                         = 190,
                e_ADD_SENSOR_TO_QUEUE           = 200,
                e_GET_CALLED_SENSOR_DATA        = 210
        };
};
```

Als letzten Parameter kann man der Nachricht noch einen Void-Pointer übergeben. In diesem exemplarischen Fall wird ein Pointer auf ein Objekt vom Typ *m_SensorCmd* übergeben, das in einen Void-Pointer gecastet wird. Die erzeugte Nachricht ist nun mit allen nötigen Informationen versehen worden und muss nun nur noch mit der Funktion *sl_SendMessage* des Proxy verschickt werden.

Wie der Empfänger-Task ADAS-Messages verarbeitet, wird im Abschnitt 4.2.1 beschrieben.

KAPITEL 4

Umsetzung

4.1. SSC-Kommunikation

Die SSC-Kommunikation ist ein wesentlicher Bestandteil des Gesamtsystems, da über diesen Bus die Radar-Rohdaten an das System übermittelt werden.

4.1.1. Initialisierung

Die Kommunikation über die SSC-Schnittstelle ist ein sehr wichtiger Bestandteil dieser Applikation, da über sie die Daten der Radarsensoren an die ECU übertragen werden. Der SSC-Bus ist an die ECU über das sog. Serial Peripheral Interface (SPI) angeschlossen. Die Begriffe SSC und SPI sind nahezu gleichbedeutend, wobei SSC-Bus der Begriff ist, der von VDO-Automotive verwendet wird und SPI in Dokumenten von Freescale zu finden ist (siehe [Wik]). Die ECU verfügt über einen Hardware Buffer von 512 Byte, der die eingehenden Daten zwischenspeichert.

Im Input-Task findet die SSC-Kommunikation zwischen Sensor und ECU statt. Dort wird während der Initialisierung der SPI-Port eingerichtet und die zugehörigen Callback-Funktionen werden angelegt. Der zugehörige Code sieht wie folgt aus:

Listing 4.1: Initialisierung der SSC-Schnittstelle

```
sl_Ret = Factory.rc_GetBoardInstance().sl_OpenSerial(   1,
                                                        10000000,
                                                        v_RecSSCStream,
                                                        v_DummyTransmit,
                                                        static_cast<T_VOID*>(this),
                                                        e_SPI_SLAVE_MODE,
                                                        1,
                                                        512-m_sl_AlarmLevel,
```

```
                                                    m_uba_InputBuffer,
                                                    m_sl_DataLength,
                                                    m_uba_TransmitBuffer,
                                                    257);

    if (sl_Ret != e_NOERROR)
    {
            sl_Ret = e_SERIALERROR;
            REPORT_ERROR(sl_Ret, "Installation SSC-Callback");
            return sl_Ret;
    }

    sl_Ret = Factory.rc_GetBoardInstance().sl_GpioOpen(29,
                                            ub_PORT_OUTPUT | ub_PORT_PUSHPULL,
                                            b_TRUE);
    if (sl_Ret != e_NOERROR)
    {
            sl_Ret = e_GPIOERROR;
            REPORT_ERROR(sl_Ret, "GpioOpen ERROR: Init SSC-Interface");
            return sl_Ret;
    }
```

In der Funktion *sl_OpenSerial*, die vom Framework bereit gestellt wird, werden diverse Parameter übergeben, um die SSC-Schnittstelle korrekt einzurichten. Der erste Parameter mit dem Wert 1 gibt an, welcher SPI-Port konfiguriert werden soll. Der zweite Parameter mit dem Wert 10 Millionen gibt die Baudrate an, was 10 MBit/s entspricht. Anschließend werden zwei Funktionszeiger übergeben. Dabei handelt es sich um den Receive-Callback und den Transmit-Callback. Damit sind zwei Funktionen gemeint, die interruptgesteuert sind. Wobei die eine aufgerufen wird, wenn Daten an dieser Schnittstelle empfangen werden und die andere wenn Daten versendet werden.

Beide Funktionen sind ebenfalls Bestandteil des Input-Task. Der Transmit-Callback ist jedoch nur eine Dummy-Funktion, also ohne Inhalt, da in dieser Anwendung keine Daten von der ECU über den SSC-Bus verschickt werden. Als fünften Übergabewert muss immer ein this-Zeiger, der statisch nach void gecastet wird, übergeben werden. Der sendende Radarsensor fungiert auf dem SSC-Bus immer als Master, da beim Master-Slave Prinzip, nach dem dieser Bus arbeitet, immer nur Master Daten senden können. Die ECU ist Empfänger und somit Slave. Dies wird mit der Konstante *e_SPI_SLAVE_MODE* als sechster Parameter beschrieben.

Der nächste Parameter mit dem Wert 1 gibt den Transmit-Alarmlevel an. Da die ECU keine Daten über den SSC-Bus verschickt, ist er nicht relevant. Als nächstes wird der Receive-Alarmlevel übergeben. Dieser Wert gibt an, wann der Hardware-Buffer, in dem die eingehenden Daten zwischengespeichert werden, ausgelesen werden soll. Dies geschieht dann auf Interrupt-Ebene in der Receive-Callback Funktion. Da der Buffer 512 Byte groß ist, kann der Wert zwischen 1 und 512 liegen. Ist die angegebene Anzahl an Bytes eingegangen, wird der Receive-Callback ausgelöst und die Funktion *v_RecSSCStream* aufgerufen. Dabei ist jedoch zu beachten, dass der Alarmlevel nach umgekehrter Logik funktioniert. Möchte man, nachdem 100 Byte eingegangen sind, den Callback auslösen, muss man den Alarmlevel auf 512 - 100 = 412 setzten.

Der neunte Parameter gibt einen Speicherbereich an (*m_uba_InputBuffer*), in den die Daten des Hardware-Buffers kopiert werden, wenn der Alarmlevel erreicht ist. Man muss jedoch wissen, dass mehr Byte kopiert werden, als durch den Alarmlevel vorgegeben wurden. Das liegt daran, dass zwischen dem Erreichen des Alarmlevel und dem Auslösen des Callback, bzw. dem

Auslesen der Daten aus dem Hardware Buffer, eine kleine Zeitdifferenz liegt. In dieser Zeit werden auf Grund der hohen Übertragungsgeschwindigkeit noch weitere Byte empfangen. Wenn es also dann tatsächlich zum Auslesen des Buffer kommt, liegen mehr Byte in ihm als durch das Alarmlevel vorgegeben und diese werden auch alle ausgelesen. Wieviele Byte das sind, lässt sich jedoch nicht berechnen. Das daraus resultierende Problem und seine Lösung wird im nächsten Abschnitt (SSC-Callback) beschrieben.

Der letzte Parameter von Bedeutung gibt die Größe des Speicherbereichs(*m_uba_InputBuffer*) an. Die letzten beiden Parameter geben den Speicher des Transmitbuffer und dessen Größe an. Beide Werte sind wieder irrelevant, da keine Daten verschickt werden.
Außerdem muss das GPIO-Register 29 noch, wie im Codeausschnitt zu sehen, initialisiert werden. Der Wert *b_TRUE*, der als letztes übergeben wird, deaktiviert den Port. An anderer Stelle (SensorCAN.cpp, *sl_StartTransmission*) wird der Port, kurz bevor Daten über ihn empfangen werden sollen, mit der Funktion *sl_GpioSet* aktiviert.

4.1.2. SSC-Callback

Der SSC-Callback ist die Funktion, die die Daten vom Bus einliest und speichert. Nach dem Ende der Übertragung informiert sie die Anwendung, dass neue Daten vorliegen.

Wie in Abschnitt 2.2 beschrieben, hat der verwendete Prozessor (MCP5200 von Freescale) einen Hardwarefehler. Dieser führt dazu, dass die erste Übertragung an SSC-Daten ein Byte zu wenig hat. Alle folgenden Übertragungen sind dann fehlerfrei. Aus diesem Grund gibt es in der Anwendung einen so genannten Dummyrun, der beim Hochfahren der Applikation, bzw. nach einem Reset ausgeführt wird. Dieser Dummyrun führt dazu, dass ein Sensor Daten schickt, die dann verworfen werden. Anschließend kann die reguläre Kommunikation über den SSC-Bus stattfinden.

Der Parameter pubRxBuffer ist ein Zeiger auf den Hardware-Buffer. Der Parameter slRxBufferSize gibt an, wie viele Daten im Hardwarebuffer liegen. Der Dummyrun wird über sie Zustandsvariable *m_sl_Dummyrun* gesteuert. Sie verändert ihren Wert von null bis zwei, was anzeigt, dass der Dummyrun beendet ist.

Der Callback wird jedesmal automatisch aufgerufen, wenn der Alarmlevel, der bei der Initialisierung angegeben wurde, erreicht ist. In der ersten for-Schleife werden die Daten aus dem Hardwarebuffer in den bereitgestellten Speicherbereich kopiert. Dabei werden die ausgelesenen Daten im Hardware-Buffer gelöscht, so dass wieder Platz für neue Daten ist. Da immer eine unterschiedliche Anzahl von Byte im Hardwarebuffer bereit liegt, kann man nicht mit Sicherheit davon ausgehen, dass am Ende einer Übertragung der Alarmlevel nochmal erreicht werden kann. Das bedeutet, es könnten noch Daten im Hardwarebuffer liegen, die nicht mehr heraus kopiert werden. Da es auch nicht möglich ist den Alarmlevel dynamisch anzupassen, muss man das Problem einfacher lösen.

Die while-Schleife im Callback wird dann ausgeführt, wenn weniger Byte erwartet werden bis der Alarmlevel erreicht werden kann und solange noch nicht alle Byte empfangen wurden. Mit der Funktion *sl_ReadManualRxBuffer* wird der Hardwarebuffer in einer Art "Polling-Modus" ausgelesen. Anschließend wird das GPIO-Register wieder aktiviert, was die SPI-Schnittstelle

vom SSC-Bus trennt. Als letztes wird eine ADAS-Message an den Input-Task versendet, um ihm mitzuteilen, dass die Daten nun zur Verfügung stehen.

Listing 4.2: SSC Callbackfunktion

```
T_VOID C_ServantRadarInput::v_RecSSCStream(      T_UBYTE* pubRxBuffer,
                                                 T_SLONG slRxBufferSize,
                                                 T_VOID* pv_UserData)
{
  T_SLONG sl_Ret = e_NOERROR;
  if(m_pub_DataBuffer == 0)
  {
        sl_Ret = e_NULL_POINTER;
        REPORT_ERROR(sl_Ret, "Receive SSC ERROR: Pointer to Databuffer is not correct!");
  }

  if(m_sl_DummyRun == 0)          //Dummyrun nötig, da Fehler in der Hardware
  {
        m_sl_DummyRun++;          //Dummyrun Zustandsvariable
        m_sl_DataLength--;        //beim Dummyrun wird ein Byte zu wenig übertragen
  }

  for (T_SLONG sl_Index = 0; sl_Index < slRxBufferSize; ++sl_Index)
  {     //Auslesen der Daten vom Hardwarebuffer
        m_pub_DataBuffer[m_sl_BufDataCounter++] = pubRxBuffer[sl_Index];
        //Daten werden an die Stelle, auf die der Zeiger
        m_sl_InputCounter++;
        //m_pub_DataBuffer zeigt, geschrieben
  }
  C_BoardFactory Factory;

  while(m_sl_InputCounter > m_sl_DataLength-m_sl_AlarmLevel &&
        m_sl_InputCounter < m_sl_DataLength)
  {     //Falls die empfangene Datenmenge so groß ist, dass der Alarmlevel nicht
        //mehr erreicht werden kann, aber noch nicht alle Daten der Übertragung
        //ausgelesen wurden wird nun der Hardwarebuffer manuell ausgelesen bis alle
        //Daten vorhanden sind

        T_SLONG sl_RemainingSize = m_sl_DataLength-m_sl_InputCounter;
        T_BOOL b_IsEmpty = false;
        Factory.rc_GetBoardInstance().sl_ReadManualRxBuffer(1,
                                                    &m_pub_DataBuffer[m_sl_BufDataCounter],
                                                    sl_RemainingSize,
                                                    b_IsEmpty);

        m_sl_InputCounter += sl_RemainingSize;
        m_sl_BufDataCounter += sl_RemainingSize;
  }

  if (m_sl_InputCounter >= m_sl_DataLength)
  {     //wenn alle Daten empfangen wurden, wird der SSC-Port deaktiviert
        m_sl_InputCounter = 0;
        sl_Ret = Factory.rc_GetBoardInstance().sl_GpioSet(29, b_TRUE);
        if (sl_Ret != e_NOERROR)
        {
          sl_Ret = e_GPIOERROR;
          REPORT_ERROR(sl_Ret, m_ach_MODULE_NAME);
         return;
        }

        if(m_sl_DummyRun > 1)
        {
          sl_recCount++;                // Receivercounter erhöhen
          if(sl_recCount > 1)           //falls der zweite Receiver schon ausgelsen wurde
          {
```

```
                m_sl_BufDataCounter = 0;
                sl_recCount = 0;          // Counter wieder auf 0 setzen
            }
            C_ADASMessage C_Msg;
            C_AOProxy &rc_Proxy = C_ProxyFactory::rc_GetProxy(C_ProxyFactory::e_RADAR_INPUT);
            C_Msg.v_CreateMessage(C_ADASMessage::e_USER_CMD,
                                  0,
                                  S_ServantRadarInputCmd::e_RADAR_DATA_COMPLETE,
                                  0);

            sl_Ret = rc_Proxy.sl_SendMessage(C_Msg);
            if(e_NOERROR != sl_Ret)
            {
                REPORT_ERROR(sl_Ret, "Error. Send ADAS-Msg to RadarInputServant");
            }
          }

          if(m_sl_DummyRun == 1 )
          {
            m_sl_DataLength++;
            m_sl_DummyRun++;
            m_sl_BufDataCounter = 0;
          }
    }
}
```

Nach der Übertragung der Radardaten eines Receivers liegen die Daten im Speicher. Da zur späteren Auswertung immer die Daten von beiden Receivern eines Sensors benötigt werden, findet im Programm im Anschluss an die erste Übertragung, gleich die zweite Übertragung statt. Um die Daten des Sensors aufzunehmen, steht im Programm die Struktur SensorData bereit (vgl. Anhang C). Dabei handelt es sich um ein einfaches Array in dem die Daten liegen.

Da die Daten keine reine Nutzdaten sind, sondern auch Protokolloverhead und Headerdaten beinhalten, können sie so nicht an die Signalverarbeitung übergeben werden. Deswegen gibt es die Struktur *S_AlgoData*, die Zeiger auf den Beginn der relevanten Speicherbereiche enthält. Neben den Zeigern auf die Daten enthält sie auch noch den Sensor Typ, von dem diese Daten stammen und einen Zeitstempel. Diese Struktur lässt sich auch gut an den Algo-Task übergeben (siehe Abbildung 4.2).

Der Sensor schickt, wie im Abschnitt 2.4.2 zu lesen, einen kontinuierlichen Bitstream. Die ECU liest diesen Stream byteweise ein und speichert die Bytes in einem Array. Der Sensor schickt aber eigentlich keine Byte (8 Bit Werte) sondern Word (16 Bit Werte). In diesem Format braucht auch die Signalverarbeitung im Algo-Task die eingegangen Daten. Da die Daten nicht vertauscht wurden, sondern in der richtigen Reihenfolge im Speicher liegen, kann man sie einfach umdefinieren (siehe Abbildung 4.1). Da Daten an den Algo-Task nur per ADAS-Message übergeben werden können und ADAS-Messages dies nur per Void-Pointer machen, wird dem Task ein Zeiger auf die Struktur AlgoData übergeben (siehe Abbildung 4.2). Die darin enthaltenen Zeiger, sind Zeiger auf Word, was genau zu dieser Redefinition führt, wie sie beschrieben wurde.

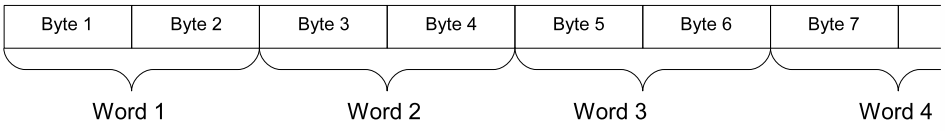

Abbildung 4.1.: Byte als Word definieren

4.2. Die Tasks

In diesem Kapitel werden die vier Tasks des Systems beschrieben. Bei den Tasks handelt es sich um unterbrechbare Task, d.h. höher priore Task und Interrupts halten sie so lange an, bis sie wieder an der Reihe sind. Ein unterbrochener Task fährt dann mit seinem Ablauf fort, wenn alle Interrupts und Tasks mit höherer Priorität nicht mehr aktiv sind. Die Priorität eines Task ist in diesem System umgekehrt Proportional zu seiner ID, d.h. je niedriger sein ID, desto höher prior ist er. Festgelegt werden die Prioritäten im Headerfile *cfg.h* des Framework.

Applikation Task	*ID 0*
Input Task	*ID 2*
Output Task	*ID 3*
Algo Task	*ID 4*

4.2.1. Aufbau der Tasks

Alle vier Tasks, der Applikation-Task, der Input-Task, der Algo-Task und der Output-Task sind fast identisch aufgebaut. Der einzige Task, der etwas vom Aufbau der übrigen abweicht, ist der Applikation-Task. Er kann lediglich ADAS-Messages an andere Tasks verschicken, nicht jedoch empfangen. Neben den obligatorischen Bestandteilen einer jeden Klasse wie Konstruktor und Destruktor haben alle Tasks die Routine *sl_DoBeforeMsgLoop*. Diese Funktion wird beim Starten der Software ein einziges Mal ausgeführt. Sie ist also eine Art Initialisierungsfunktion. Als Pendant dazu gibt es die *sl_PostMsgLoop*.

ADAS-Messages werden in der Funktion *sl_ProcessCommand* abgearbeitet, die an den Task geschickt wurden. Diese Funktion gibt es auch im Applikation-Task. Dort ist sie jedoch nicht implementiert, weswegen keine Nachrichten empfangen werden können. Die Funktion *sl_ProcessResponse* fängt die Antwort auf eine versendete ADAS-Message ab. In ihr wird erst unterschieden, um welchen Typ ADAS-Message es sich handelt (*e_USER_CMD* oder *e_SYSTEM_CMD*). Anschließend wird in einem Switch-Case-Block anhand der Subcommands weiter unterschieden. Im vorherigen Abschnitt ist in einem Beispiel beschrieben, wie man ADAS-Messages verschickt. Die Nachricht, die verschickt wurde, war an den Input-Task adressiert mit dem Subcommand *e_START_SINGLE_STEP*. Hier wird genau dieses Beispiel nochmal aufgegriffen und die Verarbeitung exemplarisch gezeigt.

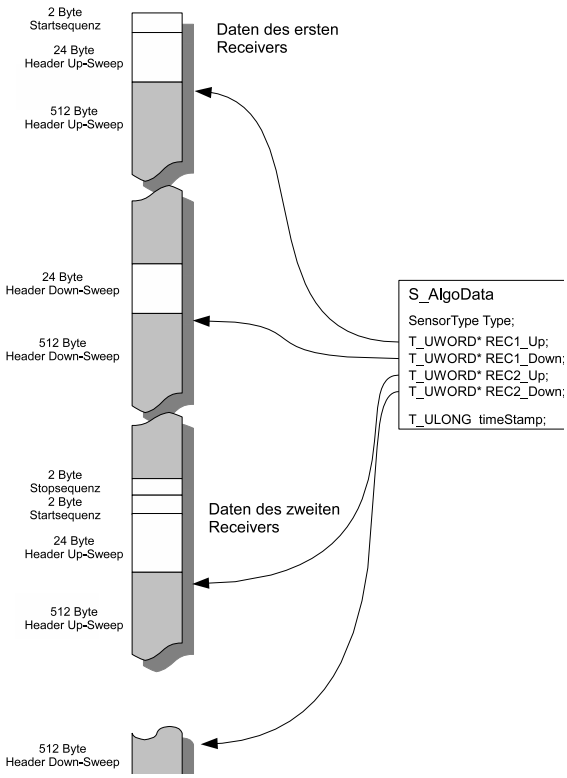

Abbildung 4.2.: SSC-Daten

Listing 4.3: Empfang von ADAS-Messages

```
T_SLONG C_ServantRadarInput::sl_ProcessCommand(C_ADASMessage* pc_Message)
{
  T_SLONG sl_Ret = e_NOERROR;
  switch(pc_Message->e_GetCommand())
  {
        case C_ADASMessage::e_USER_CMD : // user commands
          switch(pc_Message->sl_GetSubCommand())
          {
            case S_ServantRadarInputCmd::e_START_SINGLE_STEP:
            {
                  const T_VOID* v_Sensor = pc_Message->pv_GetInputData();
                  SensorType* p_Sensor = (SensorType*)v_Sensor;
                  RadarComController.b_SingleStep = b_TRUE;

                  sl_Ret = RadarComController.sl_AddSensorToQueue(*p_Sensor);
                  if(sl_Ret != e_NOERROR)
                  {
                        ErrorMonitoring->v_ErrorAddSensorToQueue(*p_Sensor);
                        REPORT_ERROR(e_SINGLE_STEP_ERROR, m_ach_MODULE_NAME);
                  }
                  sl_Ret = RadarComController.sl_StartRadarCommunication();
                  if(sl_Ret != e_NOERROR)
                  {
                        ErrorMonitoring->v_ErrorStartRadarCom(*p_Sensor);
                        REPORT_ERROR(e_START_RADAR_COM, m_ach_MODULE_NAME);
                  }
```

```
                sl_Ret = pc_Message->sl_SendResponse(C_ADASMessage::e_RESPONSE_OK);
                break;
        }
```

Zu Beginn wird der mitgeschickte Void-Pointer wieder in einen Pointer des Typs SensorType zurück gecasted. Genau das wurde der ADAS-Message beim Erstellen auch mitgegeben. Anschließend werden diverse Funktionen und die dazugehörige Fehlerbehandlung ausgeführt, die hier nicht näher betrachtet werden sollen. Wichtig ist jedoch die Funktion *pc_Message->sl_SendResponse(C_ADASMessage::e_RESPONSE_OK)*, die ganz am Ende aufgerufen wird. Mit dieser Funktion wird dem Framework bekannt gegeben, dass die ADAS-Message verarbeitet wurde und sie somit aus dem Nachrichtenspeicher gelöscht werden kann.

Zusätzlich gibt es in den Tasks noch die Funktion *sl_Response*. Dort treffen die Bestätigungen der Empfänger-Tasks ein, die mit sl_SendResponse verschickt wurden. An dieser Stelle könnte man bei Bedarf noch Programmlogik einbauen. In dieser Applikation ist dies jedoch nicht nötig.

4.2.2. Applikation

Die ECU verfügt über zwei physikalische CAN-Kanäle. Ein Kanal wird für die Kommunikation mit den Radarsensoren verwendet, der andere um Befehle zu empfangen bzw. Antworten zu senden. Der Applikation-Task besitzt einen Callback, der auf Nachrichten, die über den zweiten CAN-Kanal gesendet werden, reagiert. In dieser Funktion wird überprüft, ob es sich bei der gesendeten Nachricht um einen Befehl für die ECU handelt. Es wird jedoch nur nach CAN-IDs unterschieden.

Welche Befehle die ECU akzeptiert, ist im Anhang B zu lesen. Für jedes dieser Kommandos gibt es eine zugehörige Funktion, die vom Applikation-Task aus eine ADAS-Message verschickt, um andere Task vom eingegangen Befehl zu informieren. Normalerweise wird immer nur der Input-Task benachrichtigt, der dann das Kommando umsetzt. Beim Start-Debug Kommando jedoch wird zusätzlich noch der Algo-Task informiert.

4.2.3. Input

Der Input-Task ist der eigentliche Haupttask der Software. In ihm kommt die RadarComController Klasse zum Einsatz, die wie der Name schon sagt, ein Controller ist und die Applikation steuert. Die Hauptfunktionalität der Anwendung ist das Auslesen der Radardaten aus den Sensoren. Genau das geschieht in diesem Task. Um mit dem Radarsensor kommunizieren zu können, benötigt der Task einen CAN-Channel, um Daten an die Sensoren schicken und von einem zugehörigen Callback auch Antworten empfangen zu können. Der Input-Task besitzt auch zum Empfangen von SSC-Daten die nötigen Callbackfunktion. (Klassendiagram siehe Anhang E)

4.2.4. Output

Wie der Applikation-Task hat auch der Output-Task einen Zeiger auf den CAN-Kanal, über den die ECU Befehle empfangen kann. Allerdings verschickt der Output-Task über diesen Kanal Nachrichten. Bei diesen Nachrichten kann es sich um Fehlermeldungen, Sensordaten, die Sensorkonfiguration oder Debugdaten handeln (siehe Anhang A).

4.2.5. Algo

Der Algo-Task ist zuständig für die Auswertung der Radardaten. Er wird vom Input-Task aufgerufen, sobald die Daten für eine Auswertung zur Verfügung stehen. Dies geschieht mittels einer ADAS-Message, die den Zeiger auf die Daten enthält. In seiner einzigen Funktion, *sl_StartSigProc*, werden dann die Algorithmen des Signalverarbeitungspaketes aufgerufen, die von anderen Mitarbeitern des Autosafe-Projektes implementiert wurden.

4.3. SensorCan Klasse

Die Steuerung der Radarsensoren erfolgt über den CAN-Bus. Um die Funktionalität der Sensoren dem Rest der Applikation möglichst einfach zur Verfügung zu stellen, wurde die SensorCan Klasse implementiert. Sie hat einen Zeiger auf den CAN-Kanal, der mit den Sensoren verbunden ist und stellt folgende Funktionen bereit:

sl_Broadcast	Diese Funktion schickt eine CAN-Nachricht über den Bus, auf den alle angeschlossenen Sensoren mit einer Acknowledge-Nachricht antworten.
sl_StartMeasurement	Mit dieser Funktion werden ein oder mehrere Radarsensoren dazu veranlasst eine Aufnahme durchzuführen.
sl_SetSSCMaster	Mithilfe dieser Funktion kann man einen Sensor zum Master auf dem SSC-Bus bestimmen. Der Sensor antwortet mit einer Acknowledge-Nachricht, wenn der Befehl erfolgreich ausgeführt wurde.
sl_StartTransmission	Nachdem man zuvor einen Sensor zum Master gemacht hat, kann man ihn mit dieser Funktion dazu veranlassen, seine letzten Messdaten über den SSC-Bus zu verschicken.
sl_StartParam	Diese Funktion schickt an einen Sensor eine Nachricht, die ihn in den Konfigurationsmodus versetzte. Der Sensor antwortet darauf mit einer Acknowledge-Nachricht

sl_SendConfig Während eines Konfigurationsvorganges werden mehrere Nachrichten an den Sensor geschickt. Durch diese Funktion wird dies einfach möglich gemacht.

sl_CheckID Ob in einem Byte mehr als ein Bit gesetzt ist, wird durch diese Funktion überprüft.

Die Funktion *sl_StartMeasurement* benötigt zwei Byte als Parameter. Jedes Byte ist als Bitmaske zu verstehen. Jedes Bit steht für einen Receiver. Soll z.B. Receiver 0 und 1 zur gleichen Zeit eine Messung durchführen, müssen dieser Funktion die Byte 0x03 und 0x00 übergeben werden. Sollen z.B. Receiver 4 und 5 gemeinsam messen, müssen die Byte 0x30 und 0x00 übergeben werden, da der Receiver 4 durch das Byte 0x10 und Receiver 5 durch das Byte 0x20 angesprochen wird. In der jetzigen Variante dieser Funktion spielt es keine Rolle, welcher Wert im zweiten

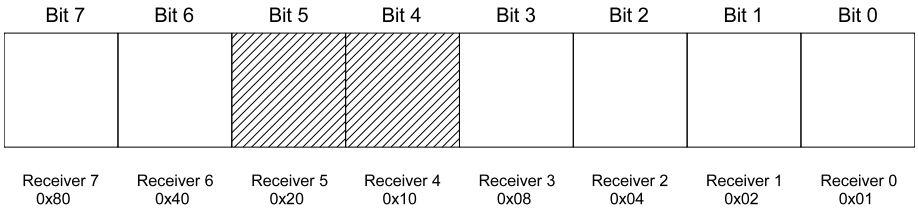

Abbildung 4.3.: Receiver Bitmaske

übergebenen Byte steht, da er intern auf 0 gesetzt wird. Sollten irgendwann mehr als acht Sensoren, bzw. Receiver mit IDs die größer als sieben sind, verwendet werden, muss diese Funktion angepasst werden, da dann auch das zweite Byte ausgewertet werden muss.

Im Gegensatz zur Funktion *sl_StartMeasurement* dürfen bei der *sl_SetSSCMaster* in den beiden übergeben Byte nicht mehr als ein Bit gesetzt sein, da nur ein Receiver als Master auf dem SSC-Bus fungieren darf. Dies wird überprüft und falls die Bedingung nicht erfüllt ist, gibt die Funktion einen Fehler an die aufrufende Funktion zurück.

Genau wie bei der *sl_SetSSCMaster* Funktion, darf auch bei der *sl_StartTransmission* nur ein Bit in den beiden übergeben Byte gesetzt sein. Nur ein Master kann Daten über den Bus senden. Es darf immer nur ein Teilnehmer zu einem Zeitpunkt als Master auf dem Bus fungieren.

Da es im Framework einen Fehler gibt, der bisher nicht behoben wurde, ist es nötig die Funktion *Factory.rc_GetBoardInstance().sl_TransmitSerial(1, m_uba_TransmitBuffer, 10)* aufzurufen (erster Parameter: PSC Nummer, zweiter Parameter: Buffer mit Byte die gesendet werden sollen, dritter Parameter: Anzahl der zu sendenden Byte). Allerdings wird mit dem Aufruf dieser Funktion nicht wirklich etwas über den SSC-Bus verschickt. Die ECU ist erstens gar nicht Master und zweitens ist die SPI-Schnittstelle des Prozessors noch vom SSC-Bus getrennt. Denn bevor die ECU Daten über den SSC-Bus empfangen kann, muss auch das GPIO-Register 29 aktiviert werden, damit der SPI-Port, der an den SSC-Bus angeschlossen ist, mit dem Prozessor verbunden wird.

Die Broadcast-Funktion versendet eine CAN-Nachricht, auf die alle Receiver mit den IDs von 0-7 mit einer Acknowledge-Nachricht antworten.

Mit der CAN-Nachricht, die die Funktion *sl_StartParam* verschickt, wird der ausgewählte Sensor in den Parametrisierungsmodus versetzt. Direkt nach dieser Nachricht sollten nun die Parameterisierungsdaten verschickt werden, da nach kurzer Zeit der Sensor automatisch aus dem Parametrisierrungsmodus zurückkehrt.

Die Funktion *sl_SendConfig* versendet Parameterisierungsdaten. Sie wird ausschließlich in der SensorParametrisation Klasse verwendet.

4.4. RadarComController Klasse

Die RadarComController Klasse ist die zentrale Klasse dieses Programms, die den Ablauf steuert und fast alle wichtigen Daten hält. Sie ist Bestandteil des Input-Task.
Liste wichtiger Daten die der RadarComController verwaltet:

SensorData Sensor_LRR	Sturktur zur Speicherung der Radardaten des Longe Rage Radar -Sensor (vgl. Anhang C)
SensorData Sensor_MRR_r	Sturktur zur Speicherung der Radardaten des Medium Rage Radar-Sensor, der rechts verbaut ist (vgl. Anhang C)
SensorData Sensor_MRR_l	Sturktur zur Speicherung der Radardaten des Medium Rage Radar-Sensor, der links verbaut ist (vgl. Anhang C)

S_AlgoData AlgoData_LRR	Die Struktur *S_AlgoData* beinhaltet Zeiger auf den Speicherbereich, der in den SensorData-Objekten angelegt wurde.
S_AlgoData AlgoData_MRR_r	Außerdem ist in ihr auch der Sensortyp zu dem diese Daten gehören vermerkt. Diese Struktur hilft die benötigten Daten,
S_AlgoData AlgoData_MRR_l	bzw. Zeiger auf die Daten, einfacher an den Algo-Task zu übergeben.

Weiterhin hat die RadarComController Klasse eine Instanz der SensorParametrisation Klasse, die dazu dient, die Sensoren zu parametrisieren. Des Weiteren verfügt er über einen Zähler, der speichert, wieviele Sensoren bereits parametrisiert wurden. Außerdem speichert die Controller Klasse welche Sensoren erkannt wurden und welche Sensoren gestört sind. Auch eine Zustandsvariable die angibt, ob sich das System im Einzelschritt- oder Dauerbetrieb arbeitet, wird von der Controller Klasse verwaltet. Eine weitere wichtige Komponente ist die Sensorqueue. In ihr wird gespeichert, welche Sensordaten als nächstes angefordert werden müssen, um sie dem Algo-Task zu übergeben.

Die RadarComController Klasse regelt nicht nur die Kommunikation zu den Sensoren, sowohl über CAN- als auch SSC-Bus, sonder steuert auch den Ablauf zwischen den einzelnen Tasks. Außerdem delegiert und überwacht sie die Parametrisierung der Sensoren durch die SensorParametrisation Klasse.

4.5. SensorParametrisation Klasse

Diese Klasse hat die Aufgabe die Sensoren mit Parametern zu versorgen, die veranlassen, dass sie im Rahmen ihrer Spezifikation arbeiten und auswertbare Daten liefern. Alle wichtigen Parameter für jeden Receiver und jeden Betriebsmodi sind in der Datei *C_SensorConfigData.h* abgelegt(siehe Anhang D). Diese Werte werden vor ihrer Übertragung an den Sensor teilweise noch mit bestimmten Faktoren multipliziert und auf zwei Byte verteilt, falls sie Werte größer als 256 annehmen können.

Ursprünglich sollten die Radarsensoren in drei Modi betrieben werden können. Bisher wurden aber nur zwei in der Sensorsoftware implementiert und nach jetzigem Stand reichen diese beiden auch aus, bzw. wird nur einer davon verwendet. Der Betriebsmodus, der verwendet wird und daher von Bedeutung ist, ist der FMCW Modus.

Die Parametrisierung eines Receivers geschieht über den CAN-Bus. Dafür werden zehn CAN-Nachrichten verschickt (siehe [M. 03b]). Die erste Nachricht ist eine Initialisierungsnachricht und ist immer gleich. Anschließend werden immer drei CAN-Nachrichten pro Modus gesendet, also drei Nachrichten für den FMCW Modus, drei Nachrichten für den CW Modus (nicht relevant) und drei Nachrichten für den nicht implementierten Modus RES (nicht relevant). Alle Nachrichten haben acht Datenbyte.

1. CAN-Nachricht FMCW:

Byte [0]: SignalLength low Byte
Byte [1]: SignalLength high Byte
Byte [2]: ADC Channel
Byte [3]: Taktdelay
Byte [4]: ADCSampletime low Byte
Byte [5]: ADCSampletime high Byte
Byte [6]: StartDelay low Byte
Byte [7]: StartDelay low Byte

SignalLength hat den Wert 256 und ist in der *C_SensorConfigData.h* als *SIGNAL_LENGTH* definiert. Dieser Wert gibt an, wieviele Sample aufgenommen werden. Es werden also pro Messung 256 Word (16 Bit Variable) an Daten generiert. Bevor er aber übertragen werden kann, wird er durch die Funktion *v_ConvertSignalLength* mittels Shift-Operator auf zwei Byte verteilt.

Der ADC-Channel gibt an, welchen der integrierten Analog-Digital-Konverter der Sensor verwenden soll. Dies ist bei jedem Sensor ein anderer. Das Taktdelay ist durch die Hardware

des Sensors gegeben. Allerdings wird es vor dem Übertragen noch mit einem Faktor von 10^6 multipliziert, da es in μs angegeben ist.

Die ADCSampletime wird ebenfalls auf zwei Byte mittels Bitshift verteilt. Sie gibt den Zeitraum an, den der A/D-Wandler benötigt, das analoge Eingangssignal zu digitalisiere. Bevor das jedoch geschieht wird der Wert mit dem Faktor $25 \cdot 10^9$ multipliziert. Auch dieser Wert sollte nicht verändert werden, da er von der Hardware des Sensors abhängig ist.

Der Wert StartDelay gibt an, wie groß die Verzögerung zwischen Eingang des Befehls zur Übertragung der Radardaten und Versenden der Daten liegt. Der Umrechnungsfaktor für diesen Wert beträgt $6,4 \cdot 10^6$. Er wird ebenfalls auf zwei Byte verteilt.

2. CAN-Nachricht FMCW:

Byte [0]: PWM Delay
Byte [1]: Sample Delay
Byte [2]: Takt Periode low Byte
Byte [3]: Takt Periode high Byte
Byte [4]: Takt Torwidth low Byte
Byte [5]: Takt Torwidth high Byte
Byte [6]: HF Clock Mode
Byte [7]: Lin Mode

Die Werte der zweiten CAN-Nachricht sollten nicht verändert werden, da auch sie größtenteils von der Hard- bzw. Software des Sensors abhängen. Der Parameter PWMDelay wird mit einem Faktor von $4 \cdot 10^6$ multipliziert, Sample Delay mit einem Faktor von $12,75 \cdot 10^6$.

Die nächsten beide Werte, Takt Periode und Takt Torwidth (*SENSOR_HF_CLOCK_PERIOD* und *SENSOR_HF_GATE_WIDTH*) werden jeweils mit dem Faktor $25 \cdot 10^9$ multipliziert und auf 2 Byte aufgeteilt. Die letzten beiden Werte, werden ohne Konvertierung in die CAN-Nachricht geschrieben.

3. CAN-Nachricht FMCW:

Byte [0]: fester Wert 0x00
Byte [1]: fester Wert 0x00
Byte [2]: fester Wert 0x19 (bei CW und RES 0x0E)
Byte [3]: Debug Variable 1
Byte [4]: Debug Variable 2
Byte [5]: Debug Variable 3
Byte [6]: Debug Variable 4
Byte [7]: Debug Variable 5

Die Debug Variablen werden ohne Konvertierung in die CAN-Nachricht geschrieben und übertragen. Über die Debug Variable an Stelle vier wird die Bandbreite geregelt, mit der der Sensor arbeiten soll. Der Sensor rechnet diesen Wert aber intern um, d.h. er multipliziert ihn mit 20. Per CAN übertragen wird jedoch genau der Wert, der auch in der Headerdatei *C_SensorConfigData.h* gespeichert ist.

4.6. Ablauf

Die nächsten Abschnitte beschreiben, welche Abläufe im Programm stattfinden, wenn ein Befehl an die ECU geschickt wird.

4.6.1. Konfiguration

Die Konfiguration der Sensoren kann durch zwei Ereignisse ausgelöst werden. Zum Einen werden den Sensoren automatisch beim Hochfahren des System konfiguriert und zum Anderen wenn das Kommando *CMD_SENSOR_RECONFIGURE* empfangen wird. Beim Einschalten des Systems wird im Input-Task automatisch die *sl_DoBeforeMsgLoop* Funktion ausgeführt, die neben der Initialisierung des CAN-Channel, der SSC-Schnittstelle auch die Initialisierung der RadarComController-Klasse veranlässt. Bei der Initialisierung dieser Klasse wird auch die Funktion *sl_Reset* aufgerufen. Auch über den *CMD_SENSOR_RECONFIGURE* Befehl, der im Applikation-Task empfangen wird, wird die *sl_Reset* über den Umweg einer ADAS-Message, die an den Input-Task adressiert ist, aufgerufen.

In der *sl_Reset* Funktion sind alle nötigen Abläufe gebündelt, die nötig sind, um das System von alten Abläufen zu bereinigen und neu starten zu können. Dazu gehört auch die Konfiguration der Radarsensoren. Damit das System nochmals neu starten kann, führt RadarComController Klasse einen Reset der Module SensorQueue, SensorCANChannel und SensorParametrisation aus. Nachdem alle drei Module wieder im Ursprungszustand sind, ruft die *sl_Reset* Funktion die *sl_StartParametrisation* im SensorParametrisation Objekt auf. Damit wird der Parametrisieungsvorgang gestartet.

Um den Receivern Zeit zu geben, auf den Broadcast mit ACK-Nachrichten zu reagieren, wird ein Timer gestartet, der nach Ablauf von $3\mu s$ die Funktion *v_StartSensorParam* aufruft. Direkt nach dem Starten des Timer wird über den SensorCANChannel Klasse ein Broadcast auf dem CAN-Kanal ausgeführt. Dies führt dazu, dass alle angeschlossenen Sensoren mit einer ACK-Message antworten.

Um feststellen zu können welche Sensoren angeschlossen sind, wird eine Broadcast-Nachricht über den CAN-Kanal geschickt. Alle Receiver antworten darauf mit einer ACK-Message. Diese Antworten werden im CAN-Callback (siehe Abbildung 4.4 (1)) empfangen. Die ID jedes Receivers der eine ACK-Message schickt wird im SensorParametrisation Objekt, im Array *sla_Receiver* gespeichert und zusätzlich der Zähler *sl_NofReceiver* erhöht. Bevor die Broadcast-Nachricht auf den Bus gelegt wird, startet ein Timer von 3 ms, der nach Ablauf seiner Zeit die Funktion *v_StartSensorParam* aufruft. In dieser Zeit haben alle angeschlossenen Receiver geantwortet und wurden im SensorParametrisation Objekt registriert.

Die Funktion *v_StartSensorParam* verschickt nun an den Input-Task eine ADAS-Message mit dem Befehl, mit der Parametrisierung der Receiver zu beginnen. Aus dem Input-Task wird die Funktion *sl_ParameteriseDetecedReceiver* des RadarComController aufgerufen, die wiederum die Funktion *sl_ReceiverDetected* im SensorParametrisation Objekt aufruft. Hier wird überprüft ob es sich bei der übergebenen Receiver ID (aus dem Array *sla_Receiver*) um eine gültige Receiver ID handelt, bevor die Funktion *sl_ParameteriseReceiver* angestoßen wird.

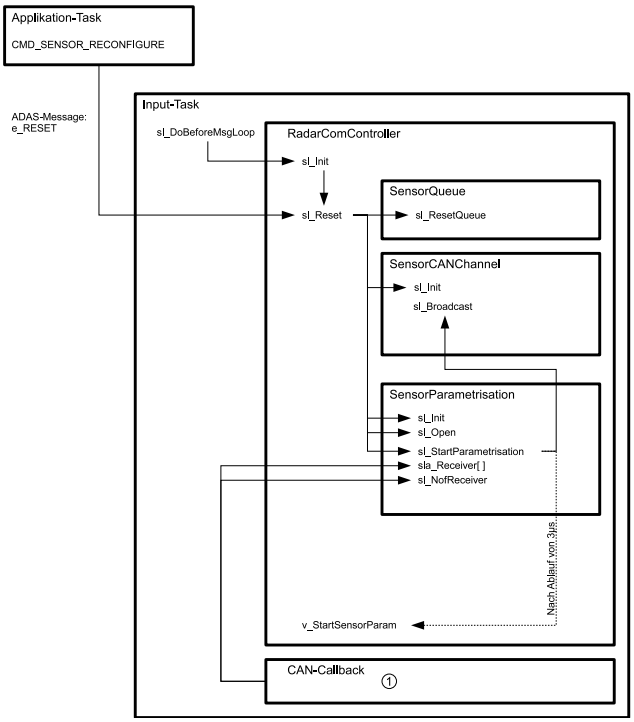

Abbildung 4.4.: Ablauf der Konfiguration Teil 1

Diese Routine ruft zuerst die SensorCANChannel-Funktion *sl_StartParam* auf, die an den entsprechenden Receiver eine CAN-Nachricht verschickt, die ihn in den Parametrisierungsmodus versetzt. Dieser Modus hält aber nicht lange an, d.h. nun müssen sofort die Parameter verschickt werden. Da es für jeden Receiver eine eigene Funktion gibt, wählt die *sl_ParameteriseReceiver* Routine die jeweils richtige aus. Alle Parametrisierungs-Funktionen greifen zum Verschicken der Parameter auf die Funtktion *sl_SendConfig* des SensorCANChannel zu. Sind alle zehn CAN-Nachrichten, die die Parameter enthalten, erfolgreich an den Receiver übermittelt worden, schickt er eine ACK-Message zurück. Diese Nachricht wird wieder CAN-Callback im Input-Task aufgefangen (siehe Abbildung 4.5 (2)). Von dort aus wird nun wieder eine ADAS-Message verschickt, die dazu führt, dass der nächste Receiver parametrisiert wird. Ist der letzte Receiver parametrisiert, wird keine weitere ADAS-Message mit dem Befehl *e_CONFIG_SENSOR* verschickt, sondern der Befehl *e_START_DUMMY_RUN* (siehe Abbildung 4.6 (1)). Dieser Befehl startet einen Übertragung über den SSC-Bus. Die dabei übertragenen Daten werden verworfen, da sie ungültig sind (siehe Kapitel 2.2). Dabei wird zuerst die Funktion *sl_CheckDetectedReceiver* ausgeführt, die überprüft, ob auch beide Receiver eines Sensors erkannt wurden. Wurde nur ein Receiver erkannt, wird der betroffene Sensor im Array BrokenSensors gespeichert und der Counter *sl_NofBrokenSensor* erhöht. Sind korrekterweise beide Receiver eines Sensors erkannt worden, wird dieser Sensor im Array DetectedSensors vermerkt und der Counter *sl_NofDetectedSensor* erhöht.

Außerdem wird die Funktion *sl_DummyRun* ausgeführt. Sie verschickt über den SensorCAN-Channel zwei Nachrichten, die einen beliebigen, vorher erkannten Receiver eine Messung

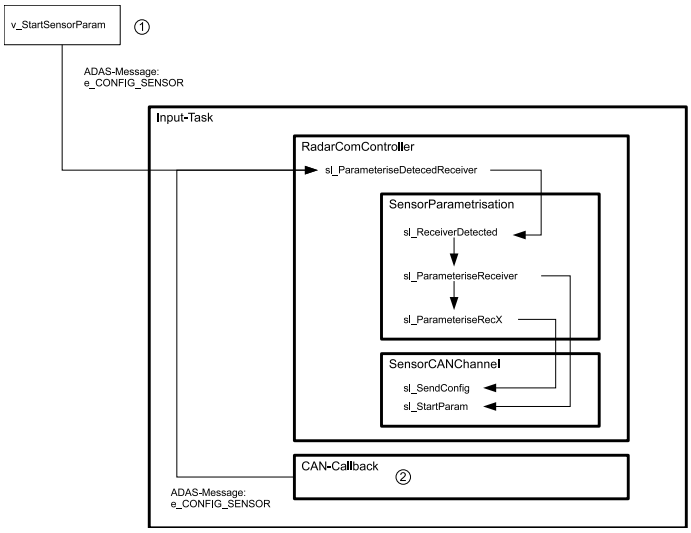

Abbildung 4.5.: Ablauf der Konfiguration Teil 2

durchführen lässt und anschließend als Master auf den SSC-Bus setzt. Darauf antwortet der Receiver mit einer ACK-Message, die im CAN-Callback im Input-Task empfangen wird (siehe Abbildung 4.6 (2)). Von dort aus wird eine ADAS-Message verschickt, die über die Funktion *sl_StartSensorTransmission* im RadarComController den Receiver dazu veranlässt, Radardaten über den SSC-Bus zu verschicken. Da nun Daten von der ECU empfangen wurden, ist der Fehler nun umgangen worden. Alle weiteren Übertragungen haben nun die richtige Anzahl von Byte, und nicht eines zu wenig wie die erste.

Damit ist der Resetvorgang abgeschlossen und das System ist wieder betriebsbereit. Allerdings gibt es auch einige Fehler, bei denen ein Software-Reset nicht ausreicht. Dann muss man die ECU hardwaremäßig reseten, was nur durch Trennen von der Spannungsquelle möglich ist.

4.6.2. Sensor Konfiguration auslesen

Für den Anwender ist es wichtig zu wissen, wie das System arbeitet. Besonders wie die Sensoren konfiguriert sind, ist für eine Auswertung von Bedeutung. Deswegen kann man mit dem Kommando *CMD_START_GET_SENS_CONFIG* die ECU dazu veranlassen, die aktuelle Konfiguration per CAN zu verschicken.

Empfangen wird der Befehl, der von Außen kommt, im Applikation-Task. Von dort wird er via ADAS-Message an den Input-Task, bzw. den RadarComController weitergereicht. Das SensorParametrisation-Objekt stellt die Funktion getSensorConfig zur Verfügung, die aufgerufen wird. Mittels Referenz gibt diese Funktion ein Objekt vom Typ CSensorConfig zurück. Aus dem RadarComController wird ein Zeiger auf dieses Objekt per ADAS-Message an den Output-Task verschickt, wo in der Funktion *sl_SendConfigToPC* alle nötigen Daten extrahiert werden und per CAN an den PC übertragen werden. Als erstes wird eine CAN-Nachricht (*CMD_NOF_SENSORS*) verschickt, die dem PC mitteilt, wieviele Sensoren an das

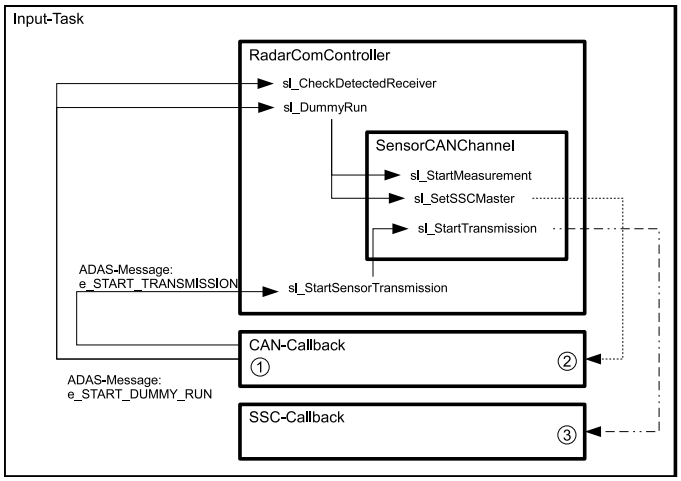

Abbildung 4.6.: Ablauf Dummyrun

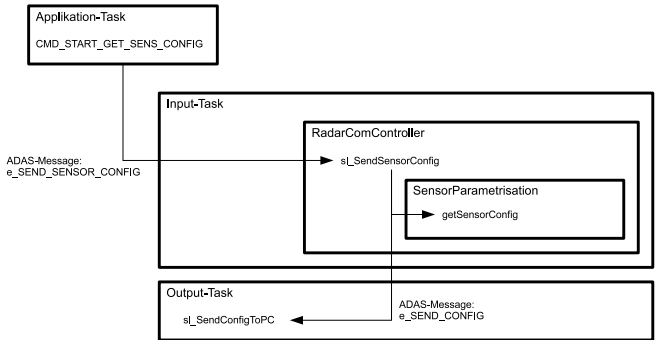

Abbildung 4.7.: Ablauf Konfiguration Auslesen

System angeschlossen sind. Anschließend wird für jeden Sensor eine Nachricht vom Typ *CMD_SENSOR_CONFIG* und vom Typ *CMD_SENSOR_ORIENTATION* verschickt (siehe Anhang A). Diese Nachrichten beinhalten alle wichtigen Informationen, die man zur Auswertung der Radardaten benötigt.

4.6.3. Dauerbetrieb

Der Dauerbetrieb ist die Betriebsart, die später die Standardbetriebsart sein soll. So lange aber nicht die komplette Signalverarbeitung auf der ECU implementiert ist, kann dieser Modus nicht vernünftig verwendet werden. Da aber sowohl der Einzelschritt Modus (Singlestep) als auch der Debug Modus, nahezu identisch sind, wird anhand des Dauerbetriebs die Funktionsweise der Software erklärt.

Gestartet wird der Dauerbetrieb mit dem CAN-Kommando *CMD_START_MEAS* (siehe auch Anhang A). Vom Applikation-Task aus werden zwei ADAS-Messages an den Input-Task, bzw.

den RadarComController verschickt. Die Nachricht *e_RESET_SINGLE_STEP* ist in Abbildung 4.8 nicht eingezeichnet. Durch sie wird im RadarComController das Flag *b_SingleStep* auf false gesetzt. Die andere ADAS-Message (*e_GET_SENSOR_DATA*), die ebenfalls an den Input-Task versendet wird, ruft zuerst die *sl_AddSesnorToQueue* auf. Je nachdem welcher Wert sich in der ursprünglich eingegangen CAN-Nachricht im ersten Byte befunden hat, wird dadurch ein einzelner oder alle Sensoren, die ans System angeschlossen sind, in der SensorQueue eingetragen. Die Sensorqueue regelt, von welchem Sensor als nächstes Daten angefordert werden sollen. Über die Funktionen *sl_StartRadarCom*, *sl_StartRadarComWithSensor* und *sl_GetRadarData*

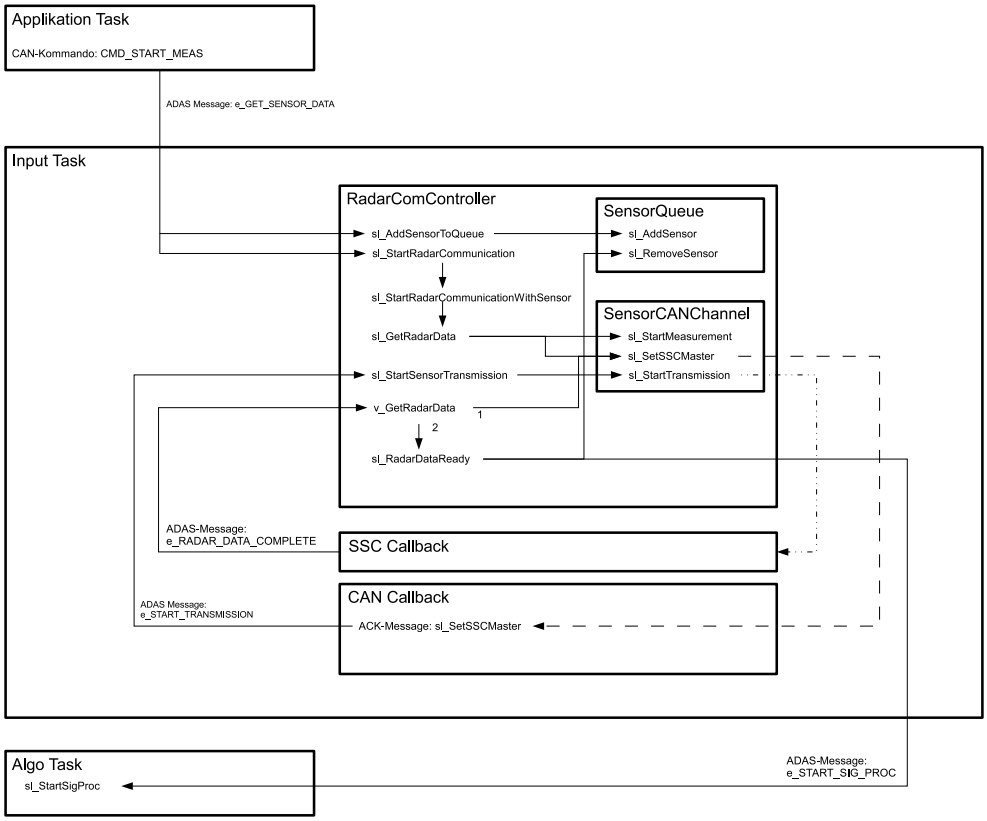

Abbildung 4.8.: Ablaufdiagramm Dauerbetrieb Teil 1

wird eine Messung des aktuellen Sensors angestoßen und anschließend wird er zum Master auf dem SSC-Bus. Dieses Kommando beantwortet der Sensor mit einer ACK-Message, die im CAN-Callback aufgefangen wird. Von dort aus wird mittels ADAS-Message, die Funktion *sl_StartSensortransmission* angesprungen. Diese schickt den entsprechenden Befehl an den Sensor, der daraufhin mit der Übertragung der Daten des ersten Receivers beginnt. Die via SSC-Bus gesendeten Daten, werden im SSC-Callback empfangen und gespeichert. Sind alle Daten eingegangen, wird eine ADAS-Message an den Input-Task verschickt, die die Funktion *v_GetRadarData* aufruft. Diese setzt, wenn sie zum ersten mal aufgerufen wird, den zweiten Receiver des Sensors als Master auf dem SSC-Bus, was dieser ebenso wie der erste Receiver mit

einer ACK-Message beantwortet. Der weitere Ablauf ist identisch, wie die erste Übertragung, bis zu dem Punkt, wo die Funktion *v_GetRadarData* aufgerufen wird. Sie startet nun keine dritte Kommunikation, sondern ruft die Funktion *sl_RadarDataReady* auf.

Diese Funktion entfernt nun den Sensor, der gerade Daten geliefert hat aus der SensorQueue und schickt eine ADAS-Message an den Algo-Task. Mit dieser Nachricht bekommt die Signalverarbeitung einen Verweis auf die Radardaten mitgeteilt und von welchem Sensor die Daten geliefert wurden. Mit diesen Informationen können die Daten nun ausgewertet werden.

Nachdem der Algo-Task seine Berechnungen beendet hat, verschickt er eine Nachricht an den Output-Task, um die berechneten Ergebnisse per CAN-Bus zu verschicken und eine ADAS-Message an den Input Task (siehe Abbildung 4.9). Die Nachricht an den Input-Task

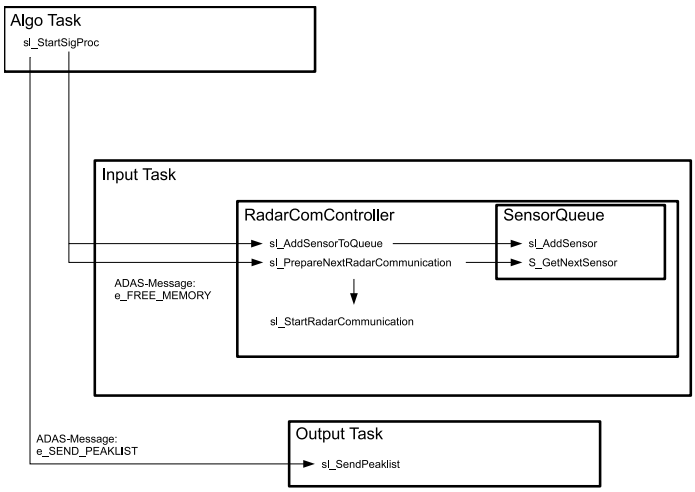

Abbildung 4.9.: Ablaufdiagramm Dauerbetrieb Teil 2

(*e_FREE_MEMORY*) bewirkt, dass ein Flag deaktiviert wird, das anzeigt, ob die Daten, die im Speicher des jeweiligen Sensors sind, bereits ausgewertet wurden. Außerdem fügt sie den Sensor, dessen Daten gerade im Algo Task verwendet wurden, an letzter Position in der SensorQueue hinzu.

Des weiteren wird die Funktion *sl_PrepareNextRadarCom* ausgeführt. Diese Funktion gibt den Sensor aus der SensorQueue zurück, der als nächstes Radardaten liefern soll.

Zuletzt wird nun die Funktion *sl_StartRadarCommunication* aufgerufen, die den Vorgang wieder von vorne beginnen lässt (siehe Abbildung 4.8).

4.6.4. Einzelschritt Modus

Der Unterschied zwischen Einzelschritt Modus und Dauerbetrieb ist marginal. Wenn das Kommando *CMD_SINGLE_STEP* vom Applikation-Task empfangen wird, werden ebenfalls zwei Nachrichten an den Input-Task versendet. Die erste heißt aber im Gegensatz zum Dauerbetrieb diesmal *e_SET_SINGLE_STEP*, was dazu führt, dass das entsprechende Flag im RadarCom-Controller auf true gesetzt wird.

Ansonsten ist der Ablauf mit dem Dauerbetrieb identisch. Erst wenn die Nachricht *e_FREE_MEMORY* vom Algo-Task nach Beendigung der Signalverarbeitung an den Input-Task geschickt wird (siehe Abbildung 4.9), gibt es wieder einen Unterschied. Die Funktion *sl_AddSensorToQueue* wird nicht aufgerufen und somit der gerade ausgewertet Sensor nicht erneut zur SensorQueue hinzugefügt. Wenn also alle angeforderten Sensoren einmal Daten geliefert haben, ist die SensorQueue leer und es werden keine weiteren Messungen durchgeführt, bis wieder ein neuer Befehl empfangen wird.

4.6.5. Debug Modus

Der Debug Modus funktioniert genauso wie der Einzelschritt Modus. Allerdings schickt er vom Applikation-Task aus eine zusätzliche Nachricht an den Algo-Task. Dies bewirkt, dass nach der Signalverarbeitung zusätzlich zu den Peaklisten auch noch weitere Nachrichten vom Output-Task verschickt werden, die die FFT-Daten und das Threshold-Level beinhalten (CAN-Kommando *CMD_SAMPLE_DATA*, siehe auch Anhang A).

4.7. ErrorMonitoring

Die ErrorMonitoring-Klasse (*Monitoring.cpp*) ist als Singleton designed, d.h. von ihr gibt es nur eine einzelne Instanz (siehe [Eri05]). Dies Instanz wird jedoch von diversen Klassen verwendet. Dazu gehören :

ServantRadarInput	(Input-Task)
ServantRadarOutput	(Output-Task)
ServantAlgo	(Algo-Task)
RadaraComController	

Die Aufgabe der ErrorMonitoring Klasse ist die Fehlerbehandlung und das Melden aufgetretener Fehler. Dafür greift sie auf den Output-Task zurück, indem sie die Fehler per ADAS-Message an den Output-Task leitet und dieser sie dann per CAN nach Außen überträgt. Eine Auflistung sämtlicher Fehlernachrichten und ihr Aufbau ist in Anhang B zu finden. Weiterhin sind auch Timeout in der ErrorMonitoring realisiert.

4.7.1. Timeout

Die Timeout überwachen, ob in einer bestimmten Zeit ein Ablauf von Funktionen erfolgreich ausgeführt wurde. In dieser Applikation gibt es drei Timeout. Der erste überwacht, ob die Kommunikation zwischen Radarsensor und ECU eine gewisse Zeit nicht überschreitet und ein zweiter Timeout überwacht die Zeit, die der Algo-Task zum Berechnen der Peaklisten benötigt. Der letzte prüft, wie lange ein Sensor benötigt, um auf den Befehl sl_SetMaster (macht den Sensor zum Master auf dem SSC-Bus) mit einer ACK-Nachricht zu antworten

Ein Timeout wird immer durch das Starten eines Timers in Gang gesetzt. Das passiert beim SensorComTimeout über die Funktion *sl_InstallSensorComTimeout*. Die Funktion wird im Input Task aufgerufen, bevor die Übertragung der SSC-Daten beginnt (wenn die ADAS-Message *e_START_TRANSMISSION* empfangen wird ; siehe Abbildung 4.8). Der Timer wird in diesem Fall auf 3000 μs gesetzt, was 3 ms entspricht. Da eine Übertragung im Normalfall ca. 2 ms benötigt, ist dieses Zeitfenster groß genug. Nach Ablauf dieser Zeitspanne wird eine beim Anlegen des Timeout bestimmte Funktion in diesem Fall *v_RadarCom* aufgerufen. Diese Funktion wird auf jeden Fall ausgeführt, selbst wenn vorher die *v_ClearSensorComTimeout* Routine ausgeführt wurde. Diese Funktion, die ausgeführt wird, wenn die Datenübertragung abgeschlossen ist (siehe Abbildung 4.8; in der Funktion *sl_RadarDataReady*), stoppt nicht etwa den Timer, sondern setzt nur ein Flag, das in der Timeoutfunktion *v_RadarCom* überprüft wird. Rein theoretisch könnte man den Timer auch anhalten, also den Timeout deinstallieren. Diese Prozedur wäre aufwendig und würde zu lange dauern. Aus diesem Grund wurde diese Funktionalität so vereinfacht implementiert.

Wenn dieser Timeout auftritt, also das Flag nicht vor Ablauf der festgelegten Zeit gesetzt wird, versucht der Timeout die Kommunikation erneut zu starten und zählt einen Fehlerzähler hoch. Dies macht er mehrere Male. Überschreitet er eine in der Funktion festgelegte Schranke, wird

die Kommunikation nicht noch einmal neu gestartet, sondern wird eine Fehlernachricht vom Typ *EC_SENSOR_COM_TIMEOUT* per CAN-Bus verschickt.

Der zweite Timeout, der im System integriert ist, überwacht die Signalverarbeitung. Eigentlich könnte man auf ihn auch verzichten, da im Algo-Task, in dem die Signalverarbeitung passiert, keine kritischen Abläufe stattfinden, wie z.B. eine Kommunikation mit einem externen Gerät. Da dieses Softwarepaket aber von anderen Entwicklern des Projekts stammt, wird die Berechnungsdauer damit überwacht. Gerade im Entwicklungsprozess, in dem immer wieder Änderungen am bestehenden Code gemacht werden oder neue Pakete hinzugefügt werden, bot dieser Timeout eine gute Überwachungsmöglichkeit der Leistung des Systems.

Der Algo-Timeout wird ebenfalls mit einer Dauer von 3 ms installiert, bevor die ADAS-Message *e_START_SIG_PROC* an den Algo-Task versendet wird. Beendet wird er mit der Funktion *v_ClearAlgoTimeout*, nachdem die Signalverarbeitung abgeschlossen ist und bevor die Ergebnisse an den Output-Task übermittelt werden. Sollte das Flag nicht rechtzeitig gesetzt werden, wird die Signalverarbeitung nicht noch einmal neu gestartet, sondern die Kommunikation mit dem nächsten Sensor begonnen. Da die Signalverarbeitung relativ lange braucht, um Ergebnisse zu liefern, wären die Daten nach einem zweiten Durchlauf schon veraltet. Wenn ein einzelner Sensor einmal keine Daten liefert, führt das im Gesamtsystem zu keinem Problem. Wichtiger sind vielmehr möglichst aktuelle Daten zu verarbeiten.

Falls während des Betriebs die Verbindung zwischen ECU und Radarsensoren verloren geht, wird dies unter anderem durch den dritten Timeout ermittelt. Dieser Timeout wird mit einer Dauer von 10 ms gestartet, kurz bevor der SetMaster Befehl via CAN an einen Receiver verschickt wird. Meldet sich der Receiver rechtzeitig mit einer ACK-Message zurück, wird diese im CAN-Callback des Input-Task empfangen und dieser Timeout, wie die beiden anderen, mittels Flag deaktiviert. Wird der Timeout nicht rechtzeitig deaktiviert, ist dies ein kritischer Fehler. Das Programm verschickt sofort eine Fehlernachricht per CAN und versucht einen Reset der Software durchzuführen, indem an den Input-Task die ADAS-Message mit dem Befehl e_RESET geschickt wird. Dies führt unter anderem dazu, dass das System evtl. wieder eine Verbindung mit den Sensoren herstellen kann, sofern es nur eine temporäre Störung war, die zum Kommunikationsverlust geführt hat. Außerdem würde erkannt werden, falls ein Sensor teilweise oder ganz ausgefallen wäre.

4.7.2. Fehlerüberwachung

Die ErrorMonitoring Klasse überwacht eine Vielzahl von Vorgängen und meldet aufgetretene Fehler über den Output-Task. Welche Fehlermeldungen es gibt und wie diese aufgebaut sind, ist im Anhang B zu lesen.

Die Funktion *sl_StartSensorTransmission* aus der RadarComController-Klasse, die das Übertragen der Radardaten veranlasst, wird von der Funktion *v_ErrorStartTransmission* überwacht. Sollte beim Starten der Übertragung ein Fehler auftreten, wird fünf mal versucht, die Übertragung erneut zu starten. Dazu wird eine ADAS-Message vom Typ *e_GET_SENSOR_DATA* an den Input-Task verschickt (siehe Abbildung 4.8). Der Sensor für den das Übertragen der Daten nicht gestartet werden konnte, wird durch dieses Kommando wieder am Ende in die SensorQueue eingetragen und das Programm fährt mit einem anderen Sensor fort.

Schlägt das Konfigurieren der Sensoren fehl, d.h. die RadarComController Funktion *sl_ParameteriseDetecedReceiver* gibt einen Fehler zurück (Input-Task bei der Verarbeitung der ADAS-Message *e_CONFIG_SENSOR*), wird dies von der ErrorMonitoring Klasse an den Output-Task gemeldet, der die Fehlernachricht per CAN verschickt.

Es werden auch Fehler, deren Auftreten weniger wahrscheinlich ist, abgefangen und nach Außen übertragen. Während des Entwicklungsprozesses ist dies ein gutes Hilfsmittel Fehler frühzeitig zu erkennen und auch Designschwächen zu beheben. Als Beispiel hierfür ist die Funktion *v_ErrorAddSensorToQueue*, die einen Fehler meldet, falls es ein Problem beim Hinzufügen eines Sensors zur Queue gibt. Bisher trat ein solcher Fehler noch nicht auf. Falls aber das System einmal erweitert wird, ist es vorstellbar, dass das System mehr Sensoren verwalten soll, als die SensorQueue verwalten kann.

Alle Funktionalitäten des Programms werden über die ADAS-Messages gesteuert, da dies die einzige Möglichkeit ist, zwischen Tasks Daten und Befehle auszutauschen. Auf dieser Ebene ist es auch relativ einfach, mögliche Fehler abzufangen. Außerdem gibt es eine überschaubare Anzahl von Möglichkeiten, wo Fehler auftreten können. Aus diesem Grund werden fast alle ErrorMonitoring Funktionen bei der Verarbeitung von ADAS-Messages in den *sl_ProcessCommand* Funktionen der Tasks verwendet.

Wenn Fehler mittels CAN-Nachricht von der ECU gemeldet wurden, reicht es im Normalfall, die ECU mittels *CMD_SENSOR_RECONFIGURE* Befehl zu reseten, um den Fehler zu beheben und in den Ursprungszustand zu versetzten. Anschließend muss man die ECU mit dem entsprechenden Befehl wieder neu starten (z.B. mit *CMD_START_MEAS* oder *CMD_START_DEBUGMODUS*). Sollte die ECU darauf nicht reagieren, hilft nur noch ein Hardwarereset, also das Steuergerät kurz von der Spannung trennen und anschließend wieder verbinden.

KAPITEL 5

Zusammenfassung

5.1. Evaluierung

Vorgabe ist es, ein System zu entwickeln, dass höchstens 30 ms benötigt um Daten eines Sensors zu Verarbeiten. Dazu gehört auch die Daten vom Sensor anzufordern und zu übertragen. Wie lange die Rechenzeit der Algorithmen ist, die benötigt werden um Objekte aus den Peaklisten zu generieren und dann diese Objekte zu tracken, also ihre Bewegung zu verfolgen, kann noch nicht gesagt werden. Diese Teilbereiche des Autosafe-Projektes befindet sich noch in der Entwicklung. Anzunehmen ist jedoch, dass diese komplexen Berechnungen einen Großteil der verfügbaren Zeit in Anspruch nehmen werden. Aus diesem Grund hat die Schnelligkeit des Systems eine hohe Priorität.

In die Software wurde die Möglichkeit integriert, Zeitmessungen während des Betriebs durchzuführen. Diese Messungen benötigen ein Minimum an Ressourcen, um den Ablauf möglichst wenig zu beeinflussen. An verschiedenen Punkten des Ablaufes wurden Messpunkte gesetzt um auch einzelne Abschnitte messtechnisch zu erfassen und nicht nur den Gesamtablauf. Dies bot die Möglichkeit Einsparpotenzial besser zu lokalisieren.

Gemessen wurde an sieben Stellen während eines Ablaufes (siehe 5.1). Gestartet wird die Zeitmessung beim Empfang des CAN-Kommandos (*CMD_START_MEAS*). Messpunkt eins ist in der Funktion *sl_GetRadarData* in der eine CAN-Nachricht an den Sensor verschickt wird, die ihn eine Messung machen lässt. Messung zwei wird beim Starten der Datenübertragung über den SSC-Bus gemacht und Messung drei nach Beendigung der Kommunikation. Da pro Sensor zwei Receiver Daten übertragen sind Messung vier und fünf auch jeweils vor und nach der SSC-Übertragung. Messung sechs ist dann direkt vor der Signalverarbeitung und die letzte Messung, Nummer sieben findet nach der Datenverarbeitung statt. Die Zeitmessungen ergaben für mehrere Durchläufe und für alle drei Sensoren, gleiche Ergebnisse.

Messung 1:	0 ms	Starten der Messung
Messung 2:	7 ms	Beginn der ersten SSC-Übertragung
Messung 3:	8 ms	Ende der ersten SSC-Übertragung
Messung 4:	8 ms	Beginn der zweiten SSC-Übertragung
Messung 5:	9 ms	Ende der zweiten SSC-Übertragung
Messung 6:	9 ms	Beginn der Signalverarbeitung
Messung 7	: 12 ms	Ende der Signalverarbeitung

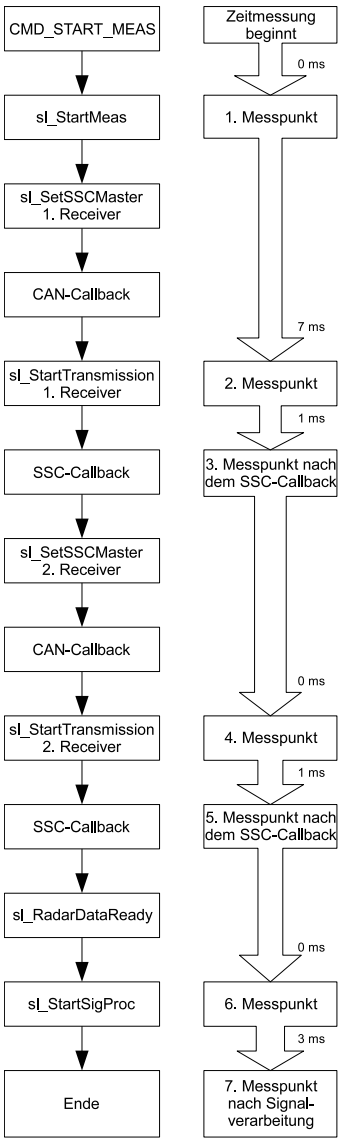

Abbildung 5.1.: Zeitmessung

An dieser Tabelle ist gut zu sehen, dass die Messung nahezu sofort nach Eingang des Befehles gestartet wird. Anschließend dauert es jedoch sieben Millisekunden bis der Sensor mit der Übertragung beginnt. Dies liegt daran, dass der Messvorgang auch Zeit benötigt und die Daten nicht sofort bereit stehen. Die SSC-Übertragung dauert (wie in Abschnitt 3.1.1 beschrieben) $870\mu s$. Da die Zeitmessung möglichst wenig Ressourcen in Anspruch nehmen soll, ist ihre kleinste messbare Zeiteinheit, eine Millisekunde. Aus der Tabelle ist zu sehen, dass jede SSC-Übertragung eine Millisekunde dauert, was der Berechnung entspricht.

Die Signalverarbeitung benötigt für ihre Berechnungen drei Millisekunden, bis die Peaklisten erzeugt wurden.

Der wichtigste Faktor zur Bewertung des Systems, neben der Performance, ist die Stabilität der Übertragung. Bei Tests im stehenden Fahrzeug, aber auch bei Fahrten mit dem Testauto der VDO Automotive AG, die mehrere Minuten dauerten, wurde kein einziger Fehler von der ECU gemeldet. Bei diesen Tests war die Fehlerbehandlung so eingestellt, dass jeder Fehler, auch wenn er nur einmal aufgetreten ist und danach selbstständig behoben wurde, per CAN gemeldet wurde.

Bei Dauertests über wenige Stunden traten ebenfalls keine Fehler auf. Die Qualität der übertragenen Daten wurde mit einzelnen Aufnahmen geprüft. Dazu wurde an vorher ausgemessenen Stellen ein Corner-Spiegel, also ein Radarreflektor gestellt und die Position mit der aus den Radardaten berechneten Position verglichen.

Auf PC-Basis gibt es eine Software (DAiSY: Driver Assistance and Savety System), mit der man unter anderem Peaklisten, die von der ECU versendet werden, auswerten und grafisch darstellen kann (siehe Abbildung 5.2). Die Situation der Aufnahme kann man im unteren, rechten Bereich des Bildes gut erkennen. Das voraus fahrende Fahrzeug befindet sich direkt vor dem Testfahrzeug und ist ein paar Meter entfernt. Auf der linken Seite des Bildes sieht man eine Visualisierung der beiden mittel weitreichende Sensoren. Bei etwas mehr als 20m, sieht man auf dem Bild eine Anhäufung von farbigen Kästchen (markiert durch einen weißen Kreis). Jedes Kästchen steht für einen Peak. Diese abgebildeten Peaks sind auch in den Peaklisten am rechten Bildrand zu finden. Peaks aus einer Peakliste des Up-Sweeps (im Bild: Sweeptype 170) sind dabei etwas größer dargestellt, als Peaks aus einer Peakliste des Down-Sweeps (Sweeptype 240).

Die Entfernung der Peaks lässt sich ganz einfach aus der FFT-Cell errechnen. Die beiden Medium Range Radarsensoren haben einen Umrechnungsfaktor von 0,29 , der Longe Range Radarsensor hat einen Umrechnungsfaktor von 1,49 . Die beiden Peaks des linken Medium Range Radarsensor liegen also in einer Entfernung von:

$$70,8 \cdot 0,29 = 20,5m$$

$$72,7 \cdot 0,29 = 21,1m$$

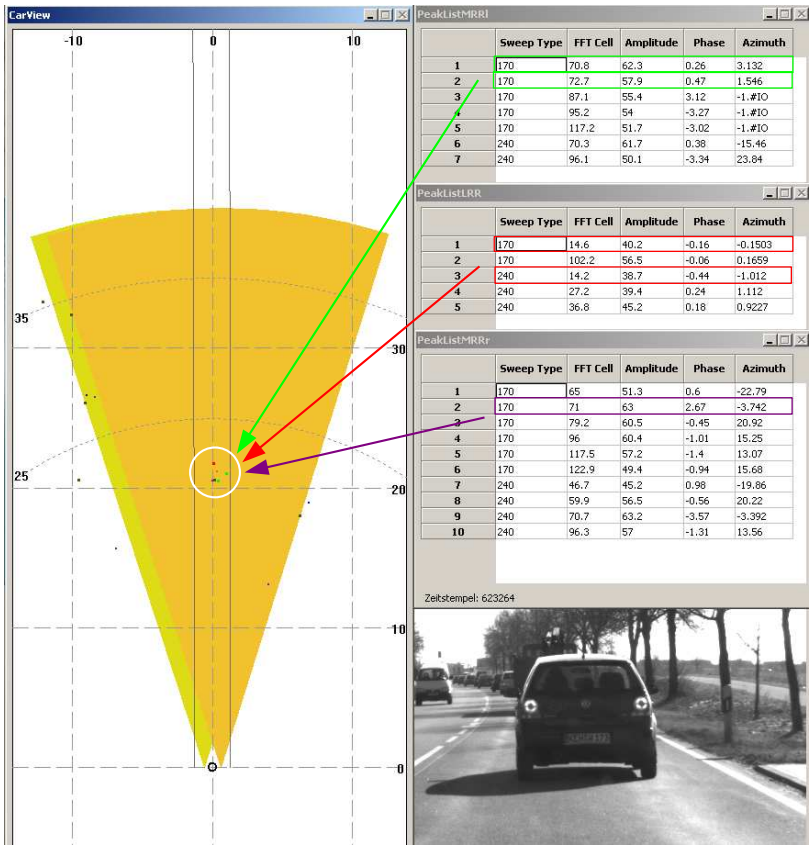

Abbildung 5.2.: DAiSy Software

5.2. Probleme

Während der Diplomarbeit gab es etliche Probleme die teilweise nicht komplett befriedigend gelöst werden konnten. Die größte Hürde bei der Planung und Umsetzung war, an zuverlässige Informationen zu kommen. Sowohl für die Hardware (ECU und Radarsensoren), als auch für die Software (ADAS-Framework) gab es nur sehr spärliche Dokumentation. Die einzige Möglichkeit an nötige Informationen zu kommen, war der direkte Kontakt zu Ansprechpartnern in den Partnerfirmen des Autosafe-Projektes. Diese mussten manchmal jedoch auch mühsam Informationen in ihrem Unternehmen zusammentragen.

Probleme ergaben sich auch aus der Tatsache, dass an der Hochschule nur Medium Range Radarsensoren zur Verfügung standen und Fehler erst auftraten, als man im Testfahrzeug die Möglichkeit hatte, alle Sensoren an das System anzuschließen.

Das Problem, das durch den Fehler auf dem Prozessor entstand (siehe Abschnitt 4.1.2), konnte relativ einfach gelöst werden, nachdem bekannt wurde, dass ein solcher Fehler existiert.

Bei manchen Fehlern, die auf der ECU auftreten, reicht es nicht, das System neu zu konfigurieren um es wieder in Gang zu setzten. Wie dieses Problem behoben wird, ist im nächsten Abschnitt zu lesen.

Es gibt ein Zeitverzögerung von sieben Millisekunden zwischen dem Befehl an den Radarsensor, dass er nun als Master auf dem SSC-Bus fungieren soll und seiner Antwort, in Form einer ACK-Message auf dem CAN-Bus. Bisher verstreicht diese Zeit ungenutzt.

5.3. Ausblick

Wie im vorherigen Abschnitt schon angesprochen gibt es noch ein paar Punkte die verbessert werden können. Dazu gehört z.B. eine richtige Reset-Funktion, die per CAN-Nachricht ausgelöst werden kann und die ECU komplett neu startet. Da es die Möglichkeit gibt das OSEK Betriebssystem zu reseten, wäre das genau die Lösung für die Fehler, die sich mit dem *CMD_SENSOR_RECONFIGURE* (siehe Anhang A) nicht beheben lassen.

Ebenso wäre es nötig die sieben Millisekunden Lücke die im Abschnitt Probleme angesprochen wird zu beheben. Da diese Lücke im Ablauf dadurch zustande kommt, dass die Messung Zeit benötigt, wäre eine erste einfach Lösung, alle Sensoren gemeinsam eine Messung starten zu lassen. Dies würde dazu führen, dass man nicht bei jedem Sensor diese Lücke hat, sondern nur einmal pro Messzyklus. In einem zweiten Schritt könnte man das Messen der Radarsensoren bereits veranlassen, während der Algo-Task noch an den Messdaten der vorherigen Messung rechnet. Diese beiden Verbesserungen würde zu einer Performancesteigerung merklich beitragen.

5.4. Resümee

Viele Dinge waren vor dieser Arbeit völlig unklar, da es wenig oder keine Erfahrung gab, was die Implementierung eines solchen Systems anging. Bereiche des Projektes, die man im Vorfeld als kritisch betrachtet hatte, ließen sich dann aber oft problemlos umsetzten. Zu Beginn gab es z.B. Zweifel daran, dass die Datenübertragung über den SSC-Bus stabil genug ist, da die Übertragungsgeschwindigkeit mit 10 MBit sehr groß ist. Mittlerweile hat sich aber gezeigt, dass die Übertragung wenig fehleranfällig ist. In den bisherigen Testläufen konnte kein Datenverlust festgestellt werden.

Auch war es fraglich, ob der verwendete Prozessor, bzw. die ECU leistungsfähig genug ist, alle nötigen Berechnungen schnell genug auszuführen. Da noch nicht alle Algorithmen auf der ECU implementiert sind, kann diese Frage auch nicht abschließend beantwortet werden, aber bisher scheint das Zeitlimit von 30 ms pro Sensor eingehalten werden zu können.

Das System das entwickelt wurde, entspricht den Anforderungen, das an es gestellt wurde. Es gibt aber noch Verbesserungspotenzial. Besonders die Stabilität der Applikation spricht für sich. Dennoch ist es notwendig die Software in weiteren Test, besonders unter Einsatzbedingungen im Testfahrzeug weiter zu überprüfen um etwaige Fehler aufzuspüren.

Literaturverzeichnis

[ABA] ABATRON AG: *Brochure BDI2000*. http://www.abatron.ch/fileadmin/user_upload/products/pdf/BDI2000.pdf.

[Con] Continental AG: *ACC mit Radar- oder Infrarot-Technologie*. http://www.conti-online.com/generator/www/de/de/cas/cas/themen/produkte/elektr_brems_und_sicherheit/fahrerassistenzsysteme/radar_und_infrarottechnologie_de.html. letzter Aufruf der Seite 13.03.08.

[Eri05] Erich Gamma, Richard Helm, Ralph Johnnson, John Vlissides: *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison - Wesley, 04/2005. ISBN 0201633612.

[Fre04] Freescale Semiconductor: *MPC5200 (L25R) Errata Rev. 4*. http://www.freescale.com/files/32bit/doc/errata/MPC5200E.pdf, September 2004.

[Fre07] Freescale: *CodeWarrior Development Tools*, 2007.

[ISO99] ISO: *ISO-WD 11898-2: Road Vehicles - Interchange of digital information - Part 1: High speed medium access unit and medium dependent interface*, 1999.

[ISO90] ISO: *ISO-WD 11898-1: Road Vehicles - Interchange of digital information - Part 1: Controller Area Network data link layer and medium access control*, 19990.

[IXX] IXXAT Automation GmbH: *Controller Area Network - Einführung*. http://www.ixxat.de/can_introduction_de,929,147.html. letzter Aufruf der Seite 13.03.08.

[K. 99] K. Etschberger: *Controller Area Network - Grundlagen, Protokolle, Bausteine, Anwendungen*. Carl Hanser Verlag, München, Wien, 1999. ISBN 3-446-19431-2.

[Kla04] Klaus Dembowski: *Netzwerke*. Markt + Technik, 2004. ISBN 3827267390.

[M. 03a] M. Koeppl: *C/C++ Naming Conventions for SV C RS*. Siemns VDO, November 2003.

[M. 03b] M. Töns: *ADAS Kommunikation Protokoll*. Siemens VDO, August 2003.

[Mar08] Marc Steuerer, Tobias Beyrle: *CAN Nachrichten*. HAW Amberg Weiden, Februar 2008.

[OSE] OSEK VDX Konsortium: *OSEK VDX Portal*. `http://www.osek-vdx.org/`. letzter Aufruf der Seite 13.03.08.

[R.] R. Greg Lavender, Douglas C. Schmidt: *Active Object , An Object Behavioral Pattern for Concurrent Programming*. `http://www.cs.wustl.edu/~schmidt/PDF/Act-Obj.pdf`.

[Rei03] Reinhard Hamperl: *C/C++ Programming Guideline (embedded Software) for SV C RS*. Siemns VDO, Dezember 2003.

[Sas05] Sascha Ries: *Hinderniserkennung*, Januar 2005.

[Sie06] Siemens VDO: *Sensor Requirements Autosafe*. Version 1.2, Mai 2006.

[Vec] Vector Informatik GmbH: *Quick Introduction to CANalyzer*.

[Wer04] Werner Zimmermann, Ralf Schmidgall: *Bussysteme in der Fahrzeugtechnik - Protokolle und Standards*. Vieweg, 2004. ISBN 978-3-8348-0235-4.

[Wik] Wikipedia: *Serial Peripheral Interface*. `http://de.wikipedia.org/wiki/Serial_Peripheral_Interface`. letzter Aufruf der Seite 13.03.08.

Abkürzungsverzeichnis

A/D Analog / Digital

AC97 Audio Codec '97

ACK Acknowledge

ADAS Advanced Driver Assistance System

ATA Advanced Technology Attachment

BMBF Bundesministerium für Bildung und Forschung

CSMA/CR Carrier Sense Multiple Access / Collision Resolution

ECU Electronic Control Unit

FMCW Frequency Modulated Continuous Wave

GPIO General Purpose Input/Output

I^2C Inter-Integrated Circuit

IDE integrierte Entwicklungsumgebung

JTAG Joint Test Action Group

MIPS Microprocessor without interlocked pipeline stages

OSEK Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug

PCMCIA Personal Computer Memory Card International Association

PSC Programmable Serial Controller

RADAR Radio Detection And Ranging

SPI Serial Peripheral Interface

SSC Synchronous Serial Channel

UART Universal Asynchronous Receiver Transmitter

USB Universal Serial Bus

Abbildungsverzeichnis

Quelltext-Verzeichnis

ANHANG A

CAN-Kommandos

Mit diesen CAN-Kommandos lässt sich die Applikation auf der ECU steuern, bzw. antwortet die ECU. Zu beachten ist, dass hier Dezimalzahlen aufgeführt sind, CAN-Identifier werden aber im hexadezimalen Zahlenformat übertragen (siehe [Mar08]).

Listing A.1: enum CanCmds

```
enum CanCmds
{
  CMD_START_MEAS              = 2000, // startet Messung
  CMD_START_SINGLE_STEP       = 2030, // startet den Singlestepmodus
  CMD_STOP_MEAS               = 2031, // stoppt Messung

  CMD_PEAKLIST_NOF_PEAKS      = 2001, // liefert Anzahl der Peaks zurück
  CMD_PEAKLIST_UP_PEAKS       = 2002, // liefert Peakliste Up
  CMD_PEAKLIST_DOWN_PEAKS     = 2003, // liefert Peakliste Down

  CMD_ERROR_MSG               = 2004, // liefert Error von ECU

  CMD_START_DEBUG_MODUS       = 2010, // startet Debug-Modus auf ECU
  CMD_SAMPLE_DATA             = 2011, // liefert die FFT Daten und den Threshold

  CMD_GET_TIME_MEAS           = 2015, // liefert Zeiten an PC
  CMD_TIME_MEAS               = 2016, // Antwortadresse

  CMD_START_GET_SENS_CONFIG   = 2022, // fordert die Sensorkonfig. von ECU an
  CMD_NOF_SENSORS             = 2023, // liefert die Anzahl der Sensoren
  CMD_SENSOR_CONFIG           = 2024, // liefert die Sensorkonfiguration
  CMD_SENSOR_ORIENTATION      = 2025, // liefert die Sensororientierung

  CMD_SENSOR_RECONFIGURE      = 2040, // Sensoren werden erneut konfiguriert
};
```

Nachfolgend ist eine kurze Beschreibung der CAN Pakete gegeben:

CMD_START_MEAS
startet eine Dauermessung, d.h. die ECU läuft selbstständig und liefert ständig die gewonnen Ergebnisse zurück.
Nachrichtenlänge: 0

CMD_STOP_MEAS
stop eine Messung, die mit CMD_START_MEAS gestartet wurde.
Nachrichtenlänge: 0

CMD_START_SINGLE_STEP
startet eine einzelne Messung. Es kann angegeben werden, für welchen Sensor die Messung ausgeführt werden soll.
Nachrichtenlänge: 1

- Byte: [0] Sensor Type, enthält den Sensor, für welchen die Messung durchgeführt werden soll. Es stehen die Typen aus dem enum SensorType zur Verfügung.

CMD_PEAKLIST_NOF_PEAKS
liefert die Anzahl der Peaks in den Up- und Down Peaklisten zurück
Nachrichtenlänge: 7

- Byte: [0] Anzahl der UP Peaks
- Byte: [1] Anzahl der Down Peaks
- Byte: [2] Enthält den Sensor Type für welchen die Listen sind. Der Typ korrespondiert mit dem enum SensorType.
- Byte: [3] Zeitstempel (low Byte); Der gesamte Zeitstempel hat die Länge von vier Byte
- Byte: [4] Zeitstempel
- Byte: [5] Zeitstempel
- Byte: [6] Zeitstempel (high Byte)

CMD_PEAKLIST_UP_PEAKS
enthält die Up Sweep Peaklisten für die letzte Messung.
Nachrichtenlänge: 6

- Byte: [0] FFT Zelle: der Wert muss ins float-Format konvertiert werden mit Faktor 10(low Byte)
- Byte: [1] FFT Zelle (high Byte)
- Byte: [2] Phase: der Wert muss ins float-Format konvertiert werden mit Faktor 100(low Byte)
- Byte: [3] Phase: (high Byte)
- Byte: [4] Amplitude: der Wert muss ins float-Format konvertiert werden mit Faktor 10(low Byte)
- Byte: [5] Die Amplitude (high Byte)

CMD_PEAKLIST_DOWN_PEAKS
enthält die Down Sweep Peaklisten für die letzte Messung.
Nachrichtenlänge: 6

Byte: [0]	FFT Zelle: der Wert muss ins float-Format konvertiert werden mit Faktor 10(low Byte)
Byte: [1]	FFT Zelle (high Byte)
Byte: [2]	Phase: der Wert muss ins float-Format konvertiert werden mit Faktor 100(low Byte)
Byte: [3]	Phase (high Byte)
Byte: [4]	Amplitude: der Wert muss ins float-Format konvertiert werden mit Faktor 10(low Byte)
Byte: [5]	Amplitude (high Byte)

CMD_ERROR_MSG
Die Error-Message besteht aus maximal 5 Paketen. Es kommt immer auf den Art der Fehlernachricht an.
Nachrichtenlänge : 5

Byte: [0]	Fehler-ID
Byte: [1]	abhängig von Fehler-ID
Byte: [2]	abhängig von Fehler-ID
Byte: [3]	abhängig von Fehler-ID
Byte: [4]	abhängig von Fehler-ID

CMD_START_DEBUG_MODUS
startet eine Debug Übertragung. Dies bedeutet, dass die ECU zusätzlich noch das Spektrum für den angegeben Sensor mit überträgt. Es kann angegeben werden für welchen Sensor die Messung ausgeführt werden soll. Jedoch kann immer nur ein einzelner Sensor abgefragt werden, nie alle gleichzeitig.
Nachrichtenlänge : 1

Byte: [0]	Sensor Type: Enthält den Sensor, für welchen die Messung durchgeführt werden soll. Es stehen die Typen aus dem enum SensorType zur Verfügung. Ausnahme: SENS_ALL funktioniert hierbei nicht.

CMD_SAMPLE_DATA
enthält das Spektrum für den angewählten Sensor.
Nachrichtenlänge: 8

- Byte: [0] Amplitudenspektrum Up Sweep: der Wert muss is float-Format konvertiert werden mit Faktor 10 (low Byte)
- Byte: [1] Amplitudenspektrum Up Sweep (high Byte)
- Byte: [2] Threshold Up Sweep: der Wert muss ins float-Format konvertiert werden mit Faktor 10 (low Byte)
- Byte: [3] Threshold Up Sweep (high Byte)
- Byte: [4] Amplitudenspektrum Down Sweep: der Wert muss ins float-Format konvertiert werden mit Faktor 10 (low Byte)
- Byte: [5] Amplitudenspektrum Down Sweep (high Byte)
- Byte: [6] Threshold Down Sweep: der Wert muss ins float-Format konvertiert werden mit Faktor 10 (low Byte)
- Byte: [7] Threshold Down Sweep (high Byte)

CMD_START_GET_SENS_CONFIG
fordert die Konfiguration der Sensoren an, welche an die ECU angeschlossen sind.
Nachrichtenlänge: 0

CMD_NOF_SENSORS
enthält die Anzahl der angeschlossenen Sensoren.
Nachrichtenlänge: 1

- Byte: [0] Anzahl der Sensoren

CMD_SENSOR_CONFIG
enthält die Konfiguration der Sensoren. Damit sind der Umrechnungsfaktor von FFT Zelle in Entfernung, sowie der Umrechnungsfaktor für die Geschwindigkeit gemeint.
Nachrichtenlänge: 5

- Byte: [0] Enthält den Sensor Type für welchen die Konfigurationsdaten bestimmt sind. Der Typ korrespondiert mit dem enum SensorType.
- Byte: [1] Umrechnungsfaktor in Entfernung: [m]; der Wert muss ins float-Format konvertiert werden mit Faktor 100 (low Byte)
- Byte: [2] Umrechnungsfaktor in Entfernung (high Byte)
- Byte: [3] Umrechnungsfaktor in Geschwindigkeit: [m/sec]; der Wert muss ins float-Format konvertiert werden mit Faktor 100 (low Byte)
- Byte: [4] Umrechnungsfaktor in Geschwindigkeit (high Byte)

CMD_SENSOR_ORIENTATION
enthält die Positionen an welchen die Sensoren am Fahrzeug montiert sind.
Nachrichtenlänge: 7

- Byte: [0] Enthält den Sensor Type für welchen die Konfiguration ist. Der Typ korres -pondiert mit dem enum SensorType.
- Byte: [1] X-Position: [m]; der Wert muss ins float-Format konvertiert werden mit Faktor 100 (low Byte)
- Byte: [2] X-Position: [m] (high Byte)
- Byte: [3] Y-Position: [m]; der Wert muss ins float-Format konvertiert werden mit Faktor 100 (low Byte)
- Byte: [4] Y-Position: [m] (high Byte)
- Byte: [5] Z-Position: [m]; der Wert muss ins float-Format konvertiert werden mit Faktor 100 (low Byte)
- Byte: [6] Z-Position: [m] (high Byte)

CMD_SENSOR_RECONFIGURE
Alle Sensoren werden erneut konfiguriert und ein Dummyrun wird ausgeführt.
Nachrichtenlänge: 0

CMD_START_GET_TIME_MEAS
fordert die Ergebnisse der Zeitmessungen welche in der ECU durchgeführt wurden an.
Nachrichtenlänge: 0

CMD_TIME_MEAS
enthält die Ergebnisse der Zeitmessungen, mit der Position an welcher die Messung ausgeführt wurde.
Nachrichtenlänge: 7

- Byte: [0] Nummer der aktuellen Messung
- Byte: [1] Anzahl aller durchgeführten Messungen
- Byte: [2] Position ID, an welcher die Messung durchgeführt wurde
- Byte: [3] Zeitmessung (low Byte); der gesamte Zeitstempel hat die Länge von vier Byte
- Byte: [4] Zeitmessung
- Byte: [5] Zeitmessung
- Byte: [6] Zeitmessung (high Byte)

ANHANG B

Error-Messages

Passiert in der Applikation ein Fehler, der nicht selbstständig behoben werden kann, wird dieser Fehler per CAN-Nachricht gemeldet. Dafür ist ein eigener CAN-Nachrichtentyp definiert worden (CMD_ERROR_MSG). Error-Messages haben eine Größe von 5 Byte. Dabei enthält das erste Byte einen Errorcode wie er in diesem enum definert ist.

Listing B.1: enum ErrorCodes

```
enum ErrorCodes
{
        EC_CONFIG_FAILED                = 100,
        EC_CANT_ADD_SENSOR_TO_QUEUE     = 110,
        EC_CANT_RESET_ECU               = 120,
        EC_SIG_PROC_FAILED              = 130,
        EC_CANT_SEND_CONFIG             = 140,
        EC_INCORRECT_SCCDATA            = 150,
        EC_RADAR_COM_FAILED             = 170,
        EC_SENSOR_COM_TIMEOUT           = 180,
        EC_INSTALL_TIMEOUT              = 190,
        EC_START_TRANSMISSION           = 200,
        EC_START_NEXT_COM               = 210,
        EC_PREPARE_NEXT_COM             = 220,
        EC_SEND_PEAKLIST                = 230,
        EC_SEND_DEBUG                   = 240,
        EC_SEND_TIME_MEAS               = 250

};
```

Die anderen 4 Byte enthalten ja nach Errorcode verschiedene Daten. Wird ein Byte nicht explizit mit Daten belegt, enthält es den Wert 0. Nachfolgend wird die Bedeutung der einzelnen Nachrichten ausführlich erläutert.

EC_CONFIG_FAILED:
Die Konfiguration der Sensoren ist fehlgeschlagen.
verwendete Byte: 5
- Byte: [0] Error-ID
- Byte: [1] Anzahl der erkannten Sensoren
- Byte: [2] Anzahl der nicht konfigurierten Sensoren
- Byte: [3] erster Sensor, der fehlerhaft war (z.B. nur ein aktiver Receiver)
- Byte: [4] Anzahl der bisher fehlgeschlagenen Konfigurationen

EC_CANT_ADD_SENSOR_TO_QUEUE:
Ein Sensor konnte nicht der SensorQueue hinzugefügt werden.
verwendete Byte: 3
- Byte: [0] Error-ID
- Byte: [1] Anzahl der Versuche, den Sensor der Queue hinzuzufügen
- Byte: [2] Sensortyp

EC_CANT_RESET_ECU:
ECU konnte nicht reseted werden.
verwendete Byte: 2
- Byte: [0] Error-ID
- Byte: [1] Anzahl der Resetversuche

EC_SIG_PROC_FAILED:
Die Signalverarbeitung wurde nicht fehlerfrei durchgeführt.
verwendete Byte: 3
- Byte: [0] Error-ID
- Byte: [1] Anzahl der Versuche die Signalverarbeitung durchzuführen
- Byte: [2] Sensor, dessen Signalverarbeitung fehlerhaft ist

EC_CANT_SEND_CONFIG:
Das Senden der Sensor Konfigurationen ist fehlgeschlagen
verwendete Byte: 2
- Byte: [0] Error-ID
- Byte: [1] Anzahl der Versuche die Konfiguration zu senden

EC_INCORRECT_SCCDATA:
Die Radardaten die über den SSC-Bus empfangen wurden sind fehlerhaft.
verwendete Byte: 3
- Byte: [0] Error-ID
- Byte: [1] Anzahl der Versuche korrekte Daten vom Sensor zu empfangen
- Byte: [2] Sensor, der die Daten nicht fehlerfrei überträgt

EC_RADAR_COM_FAILED:

Die Kommunikation mit einem Radarsensor hat nicht funktioniert.
verwendete Byte: 3
Byte: [0] Error-ID
Byte: [1] Anzahl der fehlerhaften Kommunikationsversuche
Byte: [2] Sensor der nicht fehlerfrei kommuniziert

EC_SENSOR_COM_TIMEOUT:
Die Kommunikation mit dem Sensor hat zu lange gedauert. Ein Timeout wurde ausgelöst.
verwendete Byte: 3
Byte: [0] Error-ID
Byte: [1] Anzahl wie oft der Timeout ausgelöst wurde
Byte: [2] Sensor, bei dem der Timeout ausgelöst wurde

EC_INSTALL_TIMEOUT:
Ein Timeout konnte nicht gestartet / installiert werden.
verwendete Byte: 1
Byte: [0] Error-ID

EC_START_TRANSMISSION:
Die Übertragung der Radardaten ist fehlgeschlagen.
verwendete Byte: 3
Byte: [0] Error-ID
Byte: [1] Anzahl der der Versuche, die Übertragung zu starten
Byte: [2] Sensor der Daten schicken soll

EC_START_NEXT_COM:
Das Starten des nächsten Kommunikationszyklus mit einem Radarsensor war nicht erfolgreich.
verwendete Byte: 3
Byte: [0] Error-ID
Byte: [1] Anzahl der Versuche die Kommunikation zu starten
Byte: [2] Sensor mit dem sie Kommunikation gestartet werden soll

EC_SEND_PEAKLIST:
Die Peaklistdaten konnten nicht nach Außen übertragen werden.
verwendete Byte: 2
Byte: [0] Error-ID
Byte: [1] Anzahl der Versuche die Peakliste zu verschicken

EC_SEND_DEBUG:
Das Senden der Debug-Daten ist fehlgeschlagen.
verwendete Byte: 2
Byte: [0] Error-ID
Byte: [1] Anzahl der Versuche die Debug-Daten zu verschicken

EC_SEND_TIME_MEAS:
Das Senden der Zeitmessdaten ist fehlgeschlagen.
verwendete Byte: 2
Byte: [0] Error-ID
Byte: [1] Anzahl der Versuche die Timemeasurement Daten zu verschicken

ANHANG C

Sensor Definitionen

In der Header-Datei SensorDefinition.h werden wichtige Definitionen und Datenstrukturen festgelegt.

Listing C.1: Sensor Definitionen

```
#ifndef _SENSOR_DEFINITION_H_
#define _SENSOR_DEFINITION_H_

#include "global.h"

#define MAX_NOF_SENSORS  6

const T_SLONG    SENSOR_IDS_LRR[]        = {0,1};        //Receiver IDs
const T_SLONG    SENSOR_IDS_MRR_l[]      = {4,5};        //Receiver IDs
const T_SLONG    SENSOR_IDS_MRR_r[]      = {6,7};        //Receiver IDs

enum SensorType
{
        SENS_FRONT_LRR          = 0,
        SENS_FRONT_MRR_l        = 1,
        SENS_FRONT_MRR_r        = 2,

        SENS_EMPTY              = 90,
        SENS_ALL                = 99
};

struct SensorData
{
        T_UBYTE ub_DataStorage[2160];   //SSC-Datenspeicher
        T_BOOL  b_DataStorageOccupied;  //Daten im Speicher ausgewertet?
        SensorType      Sensor;
        T_ULONG ul_TimeStamp;           //Zeitpunkt der Aufnahme der Radardaten
};

#endif //_SENSOR_DEFINITION_H_
```

ANHANG D

Sensor Parameter

In der Header-Datei SensorConfigData.h werden alle wichtige Parameter für die Sensoren festgelegt. Diese Daten werden von der SensorParametrisation Klasse verwendet, teilweise konvertiert und an die Sensoren übertragen

Listing D.1: Sensor Parameter

```
const T_SLONG   SIGNAL_LENGTH    = 256;
const T_REAL    DRO_FREQUENCY    = 22e9;
const T_REAL    LS_VACUUM        = 299792458.0;

//<VAR_PARAMS_0_FMCW>
const T_SLONG   REC0_FMCW_SENSOR_HF_CLOCK_MODE                  = 0;
const T_REAL    REC0_FMCW_SENSOR_HF_CLOCK_PERIOD                = 1.2e-6;
const T_REAL    REC0_FMCW_SENSOR_HF_GATE_WIDTH                  = 1.2e-6;
const T_SLONG   REC0_FMCW_SENSOR_START_DELAY                    = 0;
const T_REAL    REC0_FMCW_SENSOR_TAKT_DELAY                     = 10e-6;
const T_REAL    REC0_FMCW_SENSOR_PWM_DELAY                      = 400e-6;
const T_REAL    REC0_FMCW_SENSOR_SAMPLE_DELAY                   = 0;
const T_SLONG   REC0_FMCW_SENSOR_ADC_CHANNEL                    = 16;
const T_REAL    REC0_FMCW_SENSOR_ADC_SAMPLE_TIME                = 12.4e-6;
const T_UBYTE   REC0_FMCW_SENSOR_DEBUG_VARIABLES[5]             = {0, 1, 75, 10, 0};
const T_SLONG   REC0_FMCW_SENSOR_LIN_MODE                       = 0;
const T_SLONG   REC0_FMCW_SENSOR_VCO_START_FREQUENCY            = 0;
const T_SLONG   REC0_FMCW_SENSOR_VCO_STOP_FREQUENCY             = 200;
// <\VAR_PARAMS_0_FMCW>

//<VAR_PARAMS_0_CW>
const T_SLONG   REC0_CW_SENSOR_HF_CLOCK_MODE                    = 0;
const T_REAL    REC0_CW_SENSOR_HF_CLOCK_PERIOD                  = 1.2e-6;
const T_REAL    REC0_CW_SENSOR_HF_GATE_WIDTH                    = 1.2e-6;
const T_SLONG   REC0_CW_SENSOR_START_DELAY                      = 0;
const T_REAL    REC0_CW_SENSOR_TAKT_DELAY                       = 10e-6;
const T_REAL    REC0_CW_SENSOR_PWM_DELAY                        = 400e-6;
const T_REAL    REC0_CW_SENSOR_SAMPLE_DELAY                     = 0;
const T_SLONG   REC0_CW_SENSOR_ADC_CHANNEL                      = 18;
```

```
const T_REAL    REC0_CW_SENSOR_ADC_SAMPLE_TIME                = 4.0e-6;
const T_UBYTE   REC0_CW_SENSOR_DEBUG_VARIABLES[5]             = {0, 0, 0, 0, 0};
const T_SLONG   REC0_CW_SENSOR_LIN_MODE                       = 4;
const T_SLONG   REC0_CW_SENSOR_VCO_START_FREQUENCY            = 0;
const T_SLONG   REC0_CW_SENSOR_VCO_STOP_FREQUENCY             = 0;
// <\VAR_PARAMS_0_CW>

//<VAR_PARAMS_0_RES>
const T_SLONG   REC0_RES_SENSOR_HF_CLOCK_MODE                 = 0;
const T_REAL    REC0_RES_SENSOR_HF_CLOCK_PERIOD               = 550e-9;
const T_REAL    REC0_RES_SENSOR_HF_GATE_WIDTH                 = 400e-9;
const T_SLONG   REC0_RES_SENSOR_START_DELAY                   = 0;
const T_REAL    REC0_RES_SENSOR_TAKT_DELAY                    = 10e-6;
const T_REAL    REC0_RES_SENSOR_PWM_DELAY                     = 400e-6;
const T_REAL    REC0_RES_SENSOR_SAMPLE_DELAY                  = 0;
const T_SLONG   REC0_RES_SENSOR_ADC_CHANNEL                   = 0;
const T_REAL    REC0_RES_SENSOR_ADC_SAMPLE_TIME               = 12.4e-6;
const T_UBYTE   REC0_RES_SENSOR_DEBUG_VARIABLES[5]            = {0, 0, 0, 0, 0};
const T_SLONG   REC0_RES_SENSOR_LIN_MODE                      = 0;
const T_SLONG   REC0_RES_SENSOR_VCO_START_FREQUENCY           = 0;
const T_SLONG   REC0_RES_SENSOR_VCO_STOP_FREQUENCY            = 0;
// <\VAR_PARAMS_0_RES>



//<VAR_PARAMS_1_FMCW>
const T_SLONG   REC1_FMCW_SENSOR_HF_CLOCK_MODE                = 0;
const T_REAL    REC1_FMCW_SENSOR_HF_CLOCK_PERIOD              = 1.2e-6;
const T_REAL    REC1_FMCW_SENSOR_HF_GATE_WIDTH                = 1.2e-6;
const T_SLONG   REC1_FMCW_SENSOR_START_DELAY                  = 0;
const T_REAL    REC1_FMCW_SENSOR_TAKT_DELAY                   = 10e-6;
const T_REAL    REC1_FMCW_SENSOR_PWM_DELAY                    = 400e-6;
const T_REAL    REC1_FMCW_SENSOR_SAMPLE_DELAY                 = 0;
const T_UBYTE   REC1_FMCW_SENSOR_ADC_CHANNEL                  = 17;
const T_REAL    REC1_FMCW_SENSOR_ADC_SAMPLE_TIME              = 12.4e-6;
const T_UBYTE   REC1_FMCW_SENSOR_DEBUG_VARIABLES[5]           = {0, 1, 75, 10, 0};
const T_SLONG   REC1_FMCW_SENSOR_LIN_MODE                     = 0;
const T_SLONG   REC1_FMCW_SENSOR_VCO_START_FREQUENCY          = 0;
const T_SLONG   REC1_FMCW_SENSOR_VCO_STOP_FREQUENCY           = 200;
// <\VAR_PARAMS_1_FMCW>

//<VAR_PARAMS_1_CW>
const T_SLONG   REC1_CW_SENSOR_HF_CLOCK_MODE                  = 0;
const T_REAL    REC1_CW_SENSOR_HF_CLOCK_PERIOD                = 1.2e-6;
const T_REAL    REC1_CW_SENSOR_HF_GATE_WIDTH                  = 1.2e-6;
const T_SLONG   REC1_CW_SENSOR_START_DELAY                    = 0;
const T_REAL    REC1_CW_SENSOR_TAKT_DELAY                     = 10e-6;
const T_REAL    REC1_CW_SENSOR_PWM_DELAY                      = 400e-6;
const T_REAL    REC1_CW_SENSOR_SAMPLE_DELAY                   = 0;
const T_UBYTE   REC1_CW_SENSOR_ADC_CHANNEL                    = 19;
const T_REAL    REC1_CW_SENSOR_ADC_SAMPLE_TIME                = 4.0e-6;
const T_UBYTE   REC1_CW_SENSOR_DEBUG_VARIABLES[5]             = {0, 0, 0, 0, 0};
const T_SLONG   REC1_CW_SENSOR_LIN_MODE                       = 4;
const T_SLONG   REC1_CW_SENSOR_VCO_START_FREQUENCY            = 0;
const T_SLONG   REC1_CW_SENSOR_VCO_STOP_FREQUENCY             = 0;
// <\VAR_PARAMS_1_CW>

//<VAR_PARAMS_1_RES>
const T_SLONG   REC1_RES_SENSOR_HF_CLOCK_MODE                 = 0;
const T_REAL    REC1_RES_SENSOR_HF_CLOCK_PERIOD               = 550e-9;
const T_REAL    REC1_RES_SENSOR_HF_GATE_WIDTH                 = 400e-9;
const T_SLONG   REC1_RES_SENSOR_START_DELAY                   = 0;
const T_REAL    REC1_RES_SENSOR_TAKT_DELAY                    = 10e-6;
const T_REAL    REC1_RES_SENSOR_PWM_DELAY                     = 400e-6;
const T_REAL    REC1_RES_SENSOR_SAMPLE_DELAY                  = 0;
```

```
const T_SLONG    REC1_RES_SENSOR_ADC_CHANNEL                = 0;
const T_REAL     REC1_RES_SENSOR_ADC_SAMPLE_TIME            = 12.4e-6;
const T_UBYTE    REC1_RES_SENSOR_DEBUG_VARIABLES[5]         = {0, 0, 0, 0, 0};
const T_SLONG    REC1_RES_SENSOR_LIN_MODE                   = 0;
const T_SLONG    REC1_RES_SENSOR_VCO_START_FREQUENCY        = 0;
const T_SLONG    REC1_RES_SENSOR_VCO_STOP_FREQUENCY         = 0;
// <\VAR_PARAMS_1_RES>


//<VAR_PARAMS_4_FMCW>
const T_SLONG    REC4_FMCW_SENSOR_HF_CLOCK_MODE             = 0;
const T_REAL     REC4_FMCW_SENSOR_HF_CLOCK_PERIOD           = 1.2e-6;
const T_REAL     REC4_FMCW_SENSOR_HF_GATE_WIDTH             = 1.2e-6;
const T_SLONG    REC4_FMCW_SENSOR_START_DELAY               = 0;
const T_REAL     REC4_FMCW_SENSOR_TAKT_DELAY                = 10e-6;
const T_REAL     REC4_FMCW_SENSOR_PWM_DELAY                 = 400e-6;
const T_REAL     REC4_FMCW_SENSOR_SAMPLE_DELAY              = 0;
const T_SLONG    REC4_FMCW_SENSOR_ADC_CHANNEL               = 16;
const T_REAL     REC4_FMCW_SENSOR_ADC_SAMPLE_TIME           = 12.4e-6;
const T_UBYTE    REC4_FMCW_SENSOR_DEBUG_VARIABLES[5]        = {0, 1, 105, 25, 0};
const T_SLONG    REC4_FMCW_SENSOR_LIN_MODE                  = 0;
const T_SLONG    REC4_FMCW_SENSOR_VCO_START_FREQUENCY       = 0;
const T_SLONG    REC4_FMCW_SENSOR_VCO_STOP_FREQUENCY        = 500;
// <\VAR_PARAMS_4_FMCW>

//<VAR_PARAMS_4_CW>
const T_SLONG    REC4_CW_SENSOR_HF_CLOCK_MODE               = 0;
const T_REAL     REC4_CW_SENSOR_HF_CLOCK_PERIOD             = 1.2e-6;
const T_REAL     REC4_CW_SENSOR_HF_GATE_WIDTH               = 1.2e-6;
const T_SLONG    REC4_CW_SENSOR_START_DELAY                 = 0;
const T_REAL     REC4_CW_SENSOR_TAKT_DELAY                  = 10e-6;
const T_REAL     REC4_CW_SENSOR_PWM_DELAY                   = 400e-6;
const T_REAL     REC4_CW_SENSOR_SAMPLE_DELAY                = 0;
const T_SLONG    REC4_CW_SENSOR_ADC_CHANNEL                 = 18;
const T_REAL     REC4_CW_SENSOR_ADC_SAMPLE_TIME             = 4.0e-6;
const T_UBYTE    REC4_CW_SENSOR_DEBUG_VARIABLES[5]          = {0, 0, 0, 0, 0};
const T_SLONG    REC4_CW_SENSOR_LIN_MODE                    = 4;
const T_SLONG    REC4_CW_SENSOR_VCO_START_FREQUENCY         = 0;
const T_SLONG    REC4_CW_SENSOR_VCO_STOP_FREQUENCY          = 0;
// <\VAR_PARAMS_4_CW>

//<VAR_PARAMS_4_RES>
const T_SLONG    REC4_RES_SENSOR_HF_CLOCK_MODE              = 0;
const T_REAL     REC4_RES_SENSOR_HF_CLOCK_PERIOD            = 550e-9;
const T_REAL     REC4_RES_SENSOR_HF_GATE_WIDTH              = 400e-9;
const T_SLONG    REC4_RES_SENSOR_START_DELAY                = 0;
const T_REAL     REC4_RES_SENSOR_TAKT_DELAY                 = 10e-6;
const T_REAL     REC4_RES_SENSOR_PWM_DELAY                  = 400e-6;
const T_REAL     REC4_RES_SENSOR_SAMPLE_DELAY               = 0;
const T_SLONG    REC4_RES_SENSOR_ADC_CHANNEL                = 0;
const T_REAL     REC4_RES_SENSOR_ADC_SAMPLE_TIME            = 12.4e-6;
const T_UBYTE    REC4_RES_SENSOR_DEBUG_VARIABLES[5]         = {0, 0, 0, 0, 0};
const T_SLONG    REC4_RES_SENSOR_LIN_MODE                   = 0;
const T_SLONG    REC4_RES_SENSOR_VCO_START_FREQUENCY        = 0;
const T_SLONG    REC4_RES_SENSOR_VCO_STOP_FREQUENCY         = 0;
// <\VAR_PARAMS_4_RES>


//<VAR_PARAMS_5_FMCW>
const T_SLONG    REC5_FMCW_SENSOR_HF_CLOCK_MODE             = 0;
const T_REAL     REC5_FMCW_SENSOR_HF_CLOCK_PERIOD           = 1.2e-6;
const T_REAL     REC5_FMCW_SENSOR_HF_GATE_WIDTH             = 1.2e-6;
const T_SLONG    REC5_FMCW_SENSOR_START_DELAY               = 0;
const T_REAL     REC5_FMCW_SENSOR_TAKT_DELAY                = 10e-6;
const T_REAL     REC5_FMCW_SENSOR_PWM_DELAY                 = 400e-6;
```

```
const T_REAL    REC5_FMCW_SENSOR_SAMPLE_DELAY                = 0;
const T_SLONG   REC5_FMCW_SENSOR_ADC_CHANNEL                 = 17;
const T_REAL    REC5_FMCW_SENSOR_ADC_SAMPLE_TIME             = 12.4e-6;
const T_UBYTE   REC5_FMCW_SENSOR_DEBUG_VARIABLES[5]          = {0, 1, 105, 25, 0};
const T_SLONG   REC5_FMCW_SENSOR_LIN_MODE                    = 0;
const T_SLONG   REC5_FMCW_SENSOR_VCO_START_FREQUENCY         = 0;
const T_SLONG   REC5_FMCW_SENSOR_VCO_STOP_FREQUENCY          = 500;
// <\VAR_PARAMS_5_FMCW>

//<VAR_PARAMS_5_CW>
const T_SLONG   REC5_CW_SENSOR_HF_CLOCK_MODE                 = 0;
const T_REAL    REC5_CW_SENSOR_HF_CLOCK_PERIOD               = 1.2e-6;
const T_REAL    REC5_CW_SENSOR_HF_GATE_WIDTH                 = 1.2e-6;
const T_SLONG   REC5_CW_SENSOR_START_DELAY                   = 0;
const T_REAL    REC5_CW_SENSOR_TAKT_DELAY                    = 10e-6;
const T_REAL    REC5_CW_SENSOR_PWM_DELAY                     = 400e-6;
const T_REAL    REC5_CW_SENSOR_SAMPLE_DELAY                  = 0;
const T_SLONG   REC5_CW_SENSOR_ADC_CHANNEL                   = 19;
const T_REAL    REC5_CW_SENSOR_ADC_SAMPLE_TIME               = 4.0e-6;
const T_UBYTE   REC5_CW_SENSOR_DEBUG_VARIABLES[5]            = {0, 0, 0, 0, 0};
const T_SLONG   REC5_CW_SENSOR_LIN_MODE                      = 4;
const T_SLONG   REC5_CW_SENSOR_VCO_START_FREQUENCY           = 0;
const T_SLONG   REC5_CW_SENSOR_VCO_STOP_FREQUENCY            = 0;
// <\VAR_PARAMS_5_CW>

//<VAR_PARAMS_5_RES>
const T_SLONG   REC5_RES_SENSOR_HF_CLOCK_MODE                = 0;
const T_REAL    REC5_RES_SENSOR_HF_CLOCK_PERIOD              = 550e-9;
const T_REAL    REC5_RES_SENSOR_HF_GATE_WIDTH                = 400e-9;
const T_SLONG   REC5_RES_SENSOR_START_DELAY                  = 0;
const T_REAL    REC5_RES_SENSOR_TAKT_DELAY                   = 10e-6;
const T_REAL    REC5_RES_SENSOR_PWM_DELAY                    = 400e-6;
const T_REAL    REC5_RES_SENSOR_SAMPLE_DELAY                 = 0;
const T_SLONG   REC5_RES_SENSOR_ADC_CHANNEL                  = 0;
const T_REAL    REC5_RES_SENSOR_ADC_SAMPLE_TIME              = 12.4e-6;
const T_UBYTE   REC5_RES_SENSOR_DEBUG_VARIABLES[5]           = {0, 0, 0, 0, 0};
const T_SLONG   REC5_RES_SENSOR_LIN_MODE                     = 0;
const T_SLONG   REC5_RES_SENSOR_VCO_START_FREQUENCY          = 0;
const T_SLONG   REC5_RES_SENSOR_VCO_STOP_FREQUENCY           = 0;
// <\VAR_PARAMS_5_RES>



//<VAR_PARAMS_6_FMCW>
const T_SLONG   REC6_FMCW_SENSOR_HF_CLOCK_MODE               = 0;
const T_REAL    REC6_FMCW_SENSOR_HF_CLOCK_PERIOD             = 1.2e-6;
const T_REAL    REC6_FMCW_SENSOR_HF_GATE_WIDTH               = 1.2e-6;
const T_SLONG   REC6_FMCW_SENSOR_START_DELAY                 = 0;
const T_REAL    REC6_FMCW_SENSOR_TAKT_DELAY                  = 10e-6;
const T_REAL    REC6_FMCW_SENSOR_PWM_DELAY                   = 400e-6;
const T_REAL    REC6_FMCW_SENSOR_SAMPLE_DELAY                = 0;
const T_SLONG   REC6_FMCW_SENSOR_ADC_CHANNEL                 = 17;
const T_REAL    REC6_FMCW_SENSOR_ADC_SAMPLE_TIME             = 12.4e-6;
const T_UBYTE   REC6_FMCW_SENSOR_DEBUG_VARIABLES[5]          = {0, 1, 105, 25, 0};
const T_SLONG   REC6_FMCW_SENSOR_LIN_MODE                    = 0;
const T_SLONG   REC6_FMCW_SENSOR_VCO_START_FREQUENCY         = 0;
const T_SLONG   REC6_FMCW_SENSOR_VCO_STOP_FREQUENCY          = 500;
// <\VAR_PARAMS_6_FMCW>

//<VAR_PARAMS_6_CW>
const T_SLONG   REC6_CW_SENSOR_HF_CLOCK_MODE                 = 0;
const T_REAL    REC6_CW_SENSOR_HF_CLOCK_PERIOD               = 1.2e-6;
const T_REAL    REC6_CW_SENSOR_HF_GATE_WIDTH                 = 1.2e-6;
const T_SLONG   REC6_CW_SENSOR_START_DELAY                   = 0;
const T_REAL    REC6_CW_SENSOR_TAKT_DELAY                    = 10e-6;
```

```
const T_REAL    REC6_CW_SENSOR_PWM_DELAY                    = 400e-6;
const T_REAL    REC6_CW_SENSOR_SAMPLE_DELAY                 = 0;
const T_SLONG   REC6_CW_SENSOR_ADC_CHANNEL                  = 19;
const T_REAL    REC6_CW_SENSOR_ADC_SAMPLE_TIME              = 4.0e-6;
const T_UBYTE   REC6_CW_SENSOR_DEBUG_VARIABLES[5]           = {0, 0, 0, 0, 0};
const T_SLONG   REC6_CW_SENSOR_LIN_MODE                     = 4;
const T_SLONG   REC6_CW_SENSOR_VCO_START_FREQUENCY          = 0;
const T_SLONG   REC6_CW_SENSOR_VCO_STOP_FREQUENCY           = 0;
// <\VAR_PARAMS_6_CW>

//<VAR_PARAMS_6_RES>
const T_SLONG   REC6_RES_SENSOR_HF_CLOCK_MODE               = 0;
const T_REAL    REC6_RES_SENSOR_HF_CLOCK_PERIOD             = 550e-9;
const T_REAL    REC6_RES_SENSOR_HF_GATE_WIDTH               = 400e-9;
const T_SLONG   REC6_RES_SENSOR_START_DELAY                 = 0;
const T_REAL    REC6_RES_SENSOR_TAKT_DELAY                  = 10e-6;
const T_REAL    REC6_RES_SENSOR_PWM_DELAY                   = 400e-6;
const T_REAL    REC6_RES_SENSOR_SAMPLE_DELAY                = 0;
const T_SLONG   REC6_RES_SENSOR_ADC_CHANNEL                 = 0;
const T_REAL    REC6_RES_SENSOR_ADC_SAMPLE_TIME             = 12.4e-6;
const T_UBYTE   REC6_RES_SENSOR_DEBUG_VARIABLES[5]          = {0, 0, 0, 0, 0};
const T_SLONG   REC6_RES_SENSOR_LIN_MODE                    = 0;
const T_SLONG   REC6_RES_SENSOR_VCO_START_FREQUENCY         = 0;
const T_SLONG   REC6_RES_SENSOR_VCO_STOP_FREQUENCY          = 0;
// <\VAR_PARAMS_6_RES>

//<VAR_PARAMS_7_FMCW>
const T_SLONG   REC7_FMCW_SENSOR_HF_CLOCK_MODE              = 0;
const T_REAL    REC7_FMCW_SENSOR_HF_CLOCK_PERIOD            = 1.2e-6;
const T_REAL    REC7_FMCW_SENSOR_HF_GATE_WIDTH              = 1.2e-6;
const T_SLONG   REC7_FMCW_SENSOR_START_DELAY                = 0;
const T_REAL    REC7_FMCW_SENSOR_TAKT_DELAY                 = 10e-6;
const T_REAL    REC7_FMCW_SENSOR_PWM_DELAY                  = 400e-6;
const T_REAL    REC7_FMCW_SENSOR_SAMPLE_DELAY               = 0;
const T_SLONG   REC7_FMCW_SENSOR_ADC_CHANNEL                = 17;
const T_REAL    REC7_FMCW_SENSOR_ADC_SAMPLE_TIME            = 12.4e-6;
const T_UBYTE   REC7_FMCW_SENSOR_DEBUG_VARIABLES[5]         = {0, 1, 105, 25, 0};
const T_SLONG   REC7_FMCW_SENSOR_LIN_MODE                   = 0;
const T_SLONG   REC7_FMCW_SENSOR_VCO_START_FREQUENCY        = 0;
const T_SLONG   REC7_FMCW_SENSOR_VCO_STOP_FREQUENCY         = 500;
// <\VAR_PARAMS_7_FMCW>

//<VAR_PARAMS_7_CW>
const T_SLONG   REC7_CW_SENSOR_HF_CLOCK_MODE                = 0;
const T_REAL    REC7_CW_SENSOR_HF_CLOCK_PERIOD              = 1.2e-6;
const T_REAL    REC7_CW_SENSOR_HF_GATE_WIDTH                = 1.2e-6;
const T_SLONG   REC7_CW_SENSOR_START_DELAY                  = 0;
const T_REAL    REC7_CW_SENSOR_TAKT_DELAY                   = 10e-6;
const T_REAL    REC7_CW_SENSOR_PWM_DELAY                    = 400e-6;
const T_REAL    REC7_CW_SENSOR_SAMPLE_DELAY                 = 0;
const T_SLONG   REC7_CW_SENSOR_ADC_CHANNEL                  = 19;
const T_REAL    REC7_CW_SENSOR_ADC_SAMPLE_TIME              = 4.0e-6;
const T_UBYTE   REC7_CW_SENSOR_DEBUG_VARIABLES[5]           = {0, 0, 0, 0, 0};
const T_SLONG   REC7_CW_SENSOR_LIN_MODE                     = 4;
const T_SLONG   REC7_CW_SENSOR_VCO_START_FREQUENCY          = 0;
const T_SLONG   REC7_CW_SENSOR_VCO_STOP_FREQUENCY           = 0;
// <\VAR_PARAMS_7_CW>

//<VAR_PARAMS_7_RES>
const T_SLONG   REC7_RES_SENSOR_HF_CLOCK_MODE               = 0;
const T_REAL    REC7_RES_SENSOR_HF_CLOCK_PERIOD             = 550e-9;
const T_REAL    REC7_RES_SENSOR_HF_GATE_WIDTH               = 400e-9;
const T_SLONG   REC7_RES_SENSOR_START_DELAY                 = 0;
const T_REAL    REC7_RES_SENSOR_TAKT_DELAY                  = 10e-6;
const T_REAL    REC7_RES_SENSOR_PWM_DELAY                   = 400e-6;
```

```
const T_REAL    REC7_RES_SENSOR_SAMPLE_DELAY                    = 0;
const T_SLONG   REC7_RES_SENSOR_ADC_CHANNEL                     = 0;
const T_REAL    REC7_RES_SENSOR_ADC_SAMPLE_TIME                 = 12.4e-6;
const T_UBYTE   REC7_RES_SENSOR_DEBUG_VARIABLES[5]              = {0, 0, 0, 0, 0};
const T_SLONG   REC7_RES_SENSOR_LIN_MODE                        = 0;
const T_SLONG   REC7_RES_SENSOR_VCO_START_FREQUENCY             = 0;
const T_SLONG   REC7_RES_SENSOR_VCO_STOP_FREQUENCY              = 500;
// <\VAR_PARAMS_7_RES>
```

ANHANG E

Klassendiagramm Input-Task

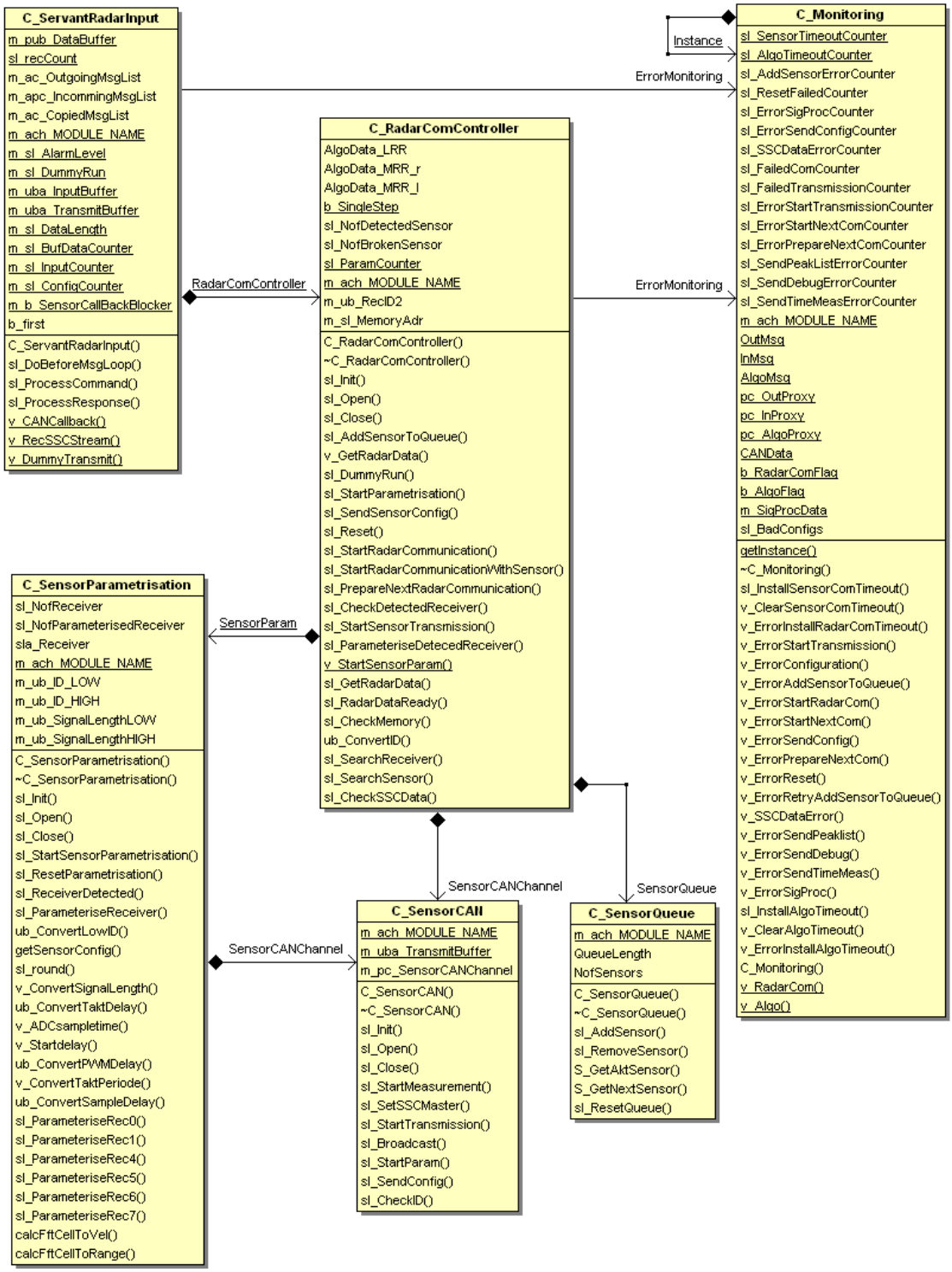

Abbildung E.1.: Klassendiagram des Input-Task