Fachhochschule Amberg-Weiden

Fachbereich Elektro- und Informationstechnik

Siemens VDO Regensburg

Prof. Dr.-Ing. Josef Pösl

Studiengang Software-Systemtechnik

Diplomarbeit von Andreas S p i t z k o p f

Entwicklung und Umsetzung eines über CAN-Bus angesteuerten Flashloaders

Fachhochschule Amberg-Weiden

Fachbereich Elektro- und Informationstechnik

Siemens VDO Regensburg

Prof. Dr.-Ing. Josef Pösl

Studiengang Software-Systemtechnik

Diplomarbeit von Andreas S p i t z k o p f

Entwicklung und Umsetzung eines über CAN-Bus angesteuerten Flashloaders

Gutachter:	Prof. Dr. -Ing. Josef Pösl
Zweitgutachter:	Prof. Dr. -Ing. Wolfgang Schindler
Bearbeitungsbeginn:	12.04.2007
Bearbeitungsende:	11.01.2008

Fachhochschule Amberg - Weiden
Hochschule für Technik und Wirtschaft

Fachbereich Elektro – und Informationstechnik
Studiengang:

Bestätigung gemäß § 31 Abs. 7 RaPO

Name und Vorname
der Studentin / des Studenten: **Spitzkopf Andreas**

Ich bestätige, daß ich die Diplomarbeit mit dem Titel:

Entwicklung und Umsetzung
eines über CAN-Bus angesteuerten Flashloaders

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: **11.01.2008**

Unterschrift:

Fachhochschule Amberg - Weiden
Hochschule für Technik und Wirtschaft

Fachbereich Elektro – und Informationstechnik
Studiengang:

Diplomarbeit Zusammenfassung

Studentin / Student (Name, Vorname): **Spitzkopf Andreas**
Studiengang / Studienschwerpunkt: **Software / Systemtechnik**

Semester der Anmeldung: **Sommersemester 2007**

Aufgabensteller: **Fa. Siemens VDO**, Regensburg

Prof.: **Prof. Dr. Josef Pösl**

Durchgeführt in (Firma / Behörde / FH): **FH Amberg-Weiden**

Betreuer in Firma / Behörde: **Dipl. Ing. (FH) Fr. Heike Lepke**

Ausgabedatum: **12.04.2007** Abgabedatum: **11.01.2008**

Semester der Abgabe: **Wintersemester 2007 / 2008**

Thema:

Entwicklung und Umsetzung eines über CAN-Bus angesteuerten Flashloaders

---

Zusammenfassung:

Im Rahmen des AUTOSAFE-Projekts werden sicherheitsrelevante Applikationen für den Automotive-Bereich entwickelt. Das hierbei zugrunde liegende Betriebssystem OSEK OS. Entwicklungssprache ist C/C++.

Die in einem Steuergerät vorhandenen Softwaremodule sollen durch einen Flashloader beim Neustart neu programmiert werden können. Die Prozesskommunikation wird dabei über den CAN-Bus gesteuert. Dazu wird eine PC-Applikation benötigt, die das CAN-Protokoll mit dem Steuergerät verbindet und den Flashvorgang durch einen entsprechenden Bootloader aktiviert. Die Steuerung des Flashvorgangs soll weiterhin über einen Webserver realisiert werden, so dass es möglich ist, das Steuergerät über einen Datei-Upload per Webbrowser zu flashen.

In einem weiteren Schritt soll untersucht werden, ob dynamisches Laden von Softwaremodulen auf der OSEK Plattform durchführbar ist.

Schlüsselworte: Flashloader, CAN-Bus, Flash over Web

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation 1

1.2 Stand der Technik 2

1.2.1 Der HIS-Flashloader Standard 2

1.2.2 Das AUTOSAFE-Projekt 3

1.2.3 Stand im Rahmen des AUTOSAFE-Projekts 3

2 Aufgabenstellung **5**

2.1 Allgemeine Beschreibung des Aufbaus 5

2.2 Ablaufvarianten des Flash-Vorgangs 6

2.3 Optionaler Bearbeitungsteil 6

3 Zugrunde liegende Hard- und Software **7**

3.1 Übersicht über die verwendeten Komponenten 7

3.1.1 Technische Merkmale der ECU 7

3.1.2 Der Hardware-Debugger 9

3.1.3 Das CAN-Interface von *Softing* 10

3.1.4 Die Entwicklungsumgebung *CodeWarrior* 10

3.1.5 GPRS Modem 11

3.1.6 Debuggen über die RS232 Schnittstelle 11

3.2 Das Übertragungsmedium CAN 13

3.2.1 Allgemeines 13

3.2.2 Aufbau 13

3.2.3 Nachrichtenformate 14

3.3 Der Flash-Speicher 17

3.3.1 Funktionsweise . . . 18
3.3.2 Vor- und Nachteile . . . 19
3.4 Das Betriebsystem OSEK/VDX-OS . . . 20
3.4.1 Beschreibung des OSEK/OS Standards . . . 20
3.4.2 Inter-Prozess Komunikation . . . 21
3.5 Das SRecord Format . . . 22

4 Entwurf der Systembestandteile 24
4.1 Der Entwurf der Bestandteile im Überblick . . . 24
4.2 Definition des WUF Dateiformats . . . 26
4.3 Aufteilung des Flash-Speicher . . . 29
4.4 Entwicklung des Übertragungsprotokolls . . . 30
4.4.1 Anforderungen an das Protokoll . . . 30
4.4.2 Allgemeines zu den Paketen . . . 31
4.4.3 Entwurf der Sender-Pakete . . . 31
4.4.4 Entwurf der Antwort-Pakete . . . 33
4.4.5 Der Flashvorgang im Detail . . . 36
4.5 Flashen per WinCan . . . 38
4.5.1 Kapselung der Funktionalitäten . . . 39
4.5.2 Vorbereitung der Sendedaten . . . 39
4.6 Entwurf des Windows User-Interface . . . 42
4.7 Flashen über eine zweite ECU . . . 43
4.7.1 Intention und Vorteile . . . 44
4.7.2 Grundlegender Aufbau und Ablauf . . . 44
4.8 Datenverarbeitung auf der Target-Seite . . . 46

5 Implementierung und Test des Gesamtsystems 48
5.1 Klassenaufbau und -verwendung in WinCan . . . 48
5.1.1 Die Klasse SRecord . . . 48
5.1.2 Die Klasse MotReader . . . 49
5.1.3 Die Klasse WUFBuilder . . . 50
5.1.4 Die Klasse *TargetItem* . . . 51
5.1.5 Die Klasse CFlashController . . . 51

5.1.6 Die Klasse *CAN_DLL* 53
5.2 Aufbau der Webserver - Applikation 56
5.3 Implementierung der Target - Applikation 59
5.3.1 Aufbau und Besonderheiten der ISR 59
5.3.2 Datenverarbeitung im Flash-Task 60
5.3.3 Zusatzinformationen im Flash-Speicher 62
5.4 Die Bootloader-Applikation 63
5.5 Aufgetretene Probleme 65

6 Zusammenfassung und Ausblick **66**
6.1 Zusammenfassung 66
6.2 Ausblick 66

Literaturverzeichnis **69**

Anhang **72**

A - Use-Case Ausarbeitungen **72**

B - Flashbaustein Datasheet **78**

Abbildungsverzeichnis

3.1 ECU mit Debug-Addon 7
3.2 ECU-Adapter 8
3.3 Hardware-Debugger BDI2000 9
3.4 Softing CAN-Interface 10
3.5 Die IDE CodeWarrior 11
3.6 Oberfläche des Filterterminals 12
3.7 CAN - Busarbitrierung 15
3.8 CAN - Frameaufbau 17
3.9 SRecord-String Aufbau 22

4.1 Aufbau der PC-seitigen Applikationen 24
4.2 Header des WUF Dateiformats 26
4.3 WUF File im Überblick 28
4.4 Aufteilung des Flash-Speicher 29
4.5 Protokoll: Sendepaket „Header“ 31
4.6 Protokoll: Sendepaket „Daten“ 32
4.7 Protokoll: Sendepaket „Schluss“ 32
4.8 Protokoll: Sendepaket „Get Info“ 33
4.9 Protokoll: Sendepaket „SW-Reset“ 33
4.10 Protokoll: Antwortpaket #1 34
4.11 Protokoll: Antwortpaket #2 34
4.12 Protokoll: Ablauf des Flashvorgangs 36
4.13 Protokoll: Einsatz des Remote-Frame 37
4.14 Aufteilung der Windows-Applikationen 38
4.15 Grobaufbau der Windows-Applikation 39

4.16 Datenerzeugung aus MOT-File 40
4.17 WUF-File Erzeugung durch WUFBuilder 41
4.18 WinCan Benutzer-Obefläche 43
4.19 Webflash 45

5.1 Aufteilung der Zusatzinformationen im Flash-Speicher 63

6.1 WinCan Screenschot 67

Symbolverzeichnis

Abb. Abbildung

API Application Programming Interface

Bsp. Beispiel

bspw. beispielsweise

bzgl. bezüglich

bzw. beziehungsweise

ca. circa

CAN Controller Area Network

DLL Direct Link Library

ECU Engine Control Unit

IDE Integrated Developement Environment

ISR Interrupt Service Routine

JTAG Joint Test Action Group

max. maximal

sog. so genannt

WUF Web Upload File

1 Einleitung

1.1 Motivation

In den letzten Jahrzehnten konnte man eine immer schneller voranschreitende Entwicklung in allen technischen Bereichen feststellen. Auch vor dem Automobilsektor machte dieser Trend nicht halt. In immer kürzeren Abständen wurden neue Auto-Modelle auf den Markt gebracht. Der vermehrte Einsatz von computergestützten Steuergeräten ist dabei nicht mehr wegzudenken. Klimaanlagen, Airbags, Soundanlage und Motorensteuerung sind nur einige Beispiele für die zunehmende Technisierung der Fahrzeugkomponenten. Dabei spielt die Entwicklungszeit dieser Steuergeräte und speziell der darauf laufenden Applikationen eine große Rolle. Ein Problem bei der Entwicklung ist die zum Teil noch sehr umständliche Handhabung dieser Steuergeräte. Durch ihre bauartbedingte Kapselung in ein abgeschlossenes System ist es sehr umständlich, darauf laufende Software zu ersetzen. Dazu muss das Gerät ausgebaut und an sog. Hardware-Debugger angeschlossen werden, um Zugriff auf den internen, persistenten Speicher zu erlangen. Während der Hauptentwicklungsphase der Applikation ist dies noch kein Nachteil, da die Software-Entwickler immer über einen Testaufbau verfügen, an dem sie die Software testen können.

Im Umfeld eines Testfahrzeugs kann dies jedoch schnell zum Nachteil werden, wenn bei jeder Code-Änderung das Steuergerät ausgebaut und an etwaige Hardware angeschlossen werden muss, um eine neue verbesserte Version der Applikation zu laden. Deshalb wäre es in dieser Entwicklungsphase ein großer Vorteil, wenn eine Möglichkeit vorhanden wäre, neue Applikations-Versionen in ein Steuergerät zu übertragen, ohne dieses aus dem Testfahrzeug entfernen zu müssen.

Aus diesem Grund beschäftigt sich diese Arbeit mit dem Entwurf und der Erstellung einer geeigneten Möglichkeit, diese Funktionalität in ein Steuergerät zu integrieren. Dabei soll soweit als möglich ausschließlich bereits vorhandene Hard- und Software verwendet werden. Ein großer

Vorteil dabei ist das bereits integrierte und ausgereifte Kommunikations-System, über das die einzelnen Steuergeräte in einem Fahrzeug miteinander verbunden sind und miteinander kommunizieren. Dieses Bus-System heißt CAN[1] und ist in Fahrzeugen sehr verbreitet. Hier lässt sich leicht ein Windows-PC mit Hilfe eines CAN-Interfaces an den Bus anbinden, eine Kommunikation mit dem benötigten Steuergerät aufbauen und eine neue Applikation laden.

Zu diesem Zweck wird Zusatz-Software sowohl auf der Ziel-Hardware als auch auf dem PC benötigt, mit der dieser Flash-Vorgang[2] durchgeführt werden kann.

In den folgenden Kapitel wird das Umfeld dieser Arbeit etwas näher erläutert und die technischen Gegebenheiten, welche die Grundlage für diese Arbeit darstellen. Zudem wird ein Standard vorgestellt, welcher das Prinzip für das Austauschen von Applikationen in Steuergeräten schon näher beschreibt und als Rahmen dienen soll. Aus diesen Gegebenheiten wird dann die genaue Aufgabenstellung für diese Arbeit definiert.

1.2 Stand der Technik

Im Hinblick auf die Entwicklung einer einfachen Methode zum Flashen von Steuergeräten wurden bereits vor einiger Zeit schon im Umfeld der Hersteller-Initiative Software (HIS[3]) Standards geschaffen, die den Aufwand für Hersteller und Zulieferer beim Erstellen von Applikationen im automobilen Umfeld minimieren sollen.

1.2.1 Der HIS-Flashloader Standard

Im HIS-Standard wurde der Begriff des Flashloaders geprägt. Hierbei handelt es sich um eine eigenständiges Software, die unabhängig von einer Applikation auf einem Steuergerät vorhanden ist. Er wurde der eigentlichen Applikation vorgeschalten und hat die Aufgabe, beim Starten des Steuergeräts bei Bedarf die vorhandene Applikation durch eine andere zu ersetzen.

Als Datenschnittstelle kommt der CAN-Bus als Standard-Kommunikationsschnittstelle des Steuer-

[1] Controller Area Network

[2] Das Austauschen der Software in einem eingebetteten System wird Flashen genannt, da meistens als persistenter Speicher sog. Flash-Speicher verwendet wird

[3] Im HIS sind die Automobilhersteller Audi, BMX, Daimler-Chrysler, Porsche und VW vertreten. Weiterführende Informationen sind unter [12] zu finden

geräts zum Einsatz. Dabei muss das Steuergerät nicht aus dem Fahrzeug ausgebaut werden, was den Aufwand für das Aufspielen einer neuen Software verringert und so die Entwicklung neuer Applikationen sehr vereinfacht.

Der HIS-Standard definiert eine Reihe von Eigenschaften für den Flashloader, die einen reibungslosen Ablauf gewährleisten sollen. Besonderer Wert wird dabei auf eine hohe Robustheit gelegt, vor allem nach dem Abbruch eines Flashvorgangs oder bei einer korrupten Anwendung. Dies soll durch umfangreiche Maßnahmen innerhalb des Flashloaders erreicht werden, indem die Gültigkeit der übertragenen Anwendung vor dem Flashvorgang geprüft wird.

Weitere Details zum HIS-Flashloader Standard und dessen Aufbau befinden sich auf der offiziellen Homepage [12], wo auch andere Arbeitsgruppen im Rahmen der Herstellerinitiative Software ihre Arbeiten vorstellen.

1.2.2 Das AUTOSAFE-Projekt

Das Projekt AUTOSAFE [7] wurde Anfang Oktober 2005 vom Bundesministerium für Bildung und Forschung (BMBF) ins Leben gerufen. In dem ursprünglich auf 3 Jahre ausgelegten Verbund-Projekt, an dem Hersteller wie Infineon München, Porsche, Siemens-VDO (jetzt Continental) und Bereiche der Siemens AG beteiligt sind, hatte es sich die FH Amberg-Weiden in enger Zusammenarbeit mit Siemens VDO zur Aufgabe gemacht, ein integrales Sicherheitssystem zur Erhöhung der Verkehrssicherheit von Fahrzeugen zu schaffen. Schlüsselanforderungen sind dabei hohe Performanz, ein starker Kundennutzen, die Wiederverwendbarkeit von Modulen, Skalierbarkeit und eine geeignete Sicherheits-Systemarchitektur.

Die Hauptergebnisse der Arbeit in dem Projekt sollen dabei in 2 Fahrzeugen getestet werden, die mit dem integralen Sicherheitssystem ausgerüstet werden.

1.2.3 Stand im Rahmen des AUTOSAFE-Projekts

Wie im vorherigen Kapitel schon beschrieben, werden im Rahmen des AUTOSAFE-Projekts sicherheitsrelevante Applikationen für den Automotive-Bereich entwickelt. Die Grundlage für diese Applikationen stellt dabei das Echtzeitbetriebssystem OSEK OS [11] dar.

Die Entwicklung neuer Applikationen erfolgt dabei unter Zuhilfenahme von etablierten Ent-

wicklungstools wie einem Hardwaredebugger[4]. Hierbei ist es notwendig, dass das Steuergerät in ausgebautem Zustand dem Entwickler zur Verfügung steht. Auf diese Weise hat der Entwickler die Möglichkeit, während des Entwickelns die Software direkt auf dem Steuergerät zu testen und zu debuggen. Die Software wird zu diesem Zweck über den Hardware-Debugger in den RAM des Steuergeräts geladen und dann gestartet. Nachdem die Software in dieser Testumgebung ausgiebigen Tests unterzogen worden ist, kann sie direkt in den Festspeicher des Geräts geschrieben werden. Dies geschieht über ein selbst entwickeltes Tool, welches ebenfalls über den Hardware-Debugger die erzeugte Software in den Flashspeicher des Steuergeräts lädt. Nachdem die Software geladen wurde, kann der Hardware-Debugger entfernt werden und beim nächsten Reset wird die geflashte Software aus dem Flashspeicher geladen und ausgeführt.

Die Entwicklung an sich findet in einer integrierten Entwicklungsumgebung von Freescale statt [9]. Das „CodeWarrior“ genannte Produkt bietet neben dem typischen Hilfsmittel auch eine direkte Anbindung an den Hardware-Debugger. So ist es ein relativ einfach, eine neu erzeugte Applikation auf dem Steuergerät zu testen.

[4]Wird direkt an den Prozessor gekoppelt und kann diesen komplett steuern und zum Beispiel anhalten

2 Aufgabenstellung

Der bisherige Weg, neue Applikationen über einen Hardware-Debugger auf das Steuergerät zu flashen, soll zukünftig durch eine einfachere Möglichkeit abgelöst werden. Angelehnt an den HIS-Flashloader soll eine reduzierte Variante des HIS-Flashloaders entwickelt werden, der nur während des Entwicklungsprozesses zum Einsatz kommt und das Laden einer neuen Applikation auch in eingebautem Zustand ermöglichen soll.

In den folgenden Kapitel soll die Vorgehensweise etwas näher beschrieben und die Anforderungen an das System definiert werden.

2.1 Allgemeine Beschreibung des Aufbaus

Als Übertragungsmedium kommt das Standard-Übertragungsmedium des Steuergeräts zum Einsatz: der CAN-Bus. Dieser ist auf die Übertragung sehr kleiner Pakete von max. 8 Byte Länge beschränkt und nicht auf die Übertragung von etwas größeren Datenmengen ausgelegt, wo hingegen Applikationen in diesem Umfeld meistens eine Größe von 200 - 600 kB besitzen. Jedoch kommt es beim Einsatz dieser Anwendung nicht auf einen hohen Datendurchsatz an, weshalb eine langsame Datenrate in Kauf genommen werden kann.

Für die Umsetzung dieses Flashloaders ist eine Zusatzapplikation angedacht, die als Bootloader für die eigentliche Applikation fungieren soll, falls eine solche installiert ist. Die bisherige Aufgabe des bereits verwendeten Bootloaders besteht darin, die im Flash-Speicher liegende Applikation in den RAM der ECU zu kopieren und diese zu starten. Zukünftig wird der Begriff „Bootloader“ erweitert auf eine komplette Applikation, die zuerst geladen wird und bei Bedarf einen Flashvorgang durchführen kann. Erst nach einer kurzen Wartezeit soll dann die eigentliche Applikation gestartet werden.

2.2 Ablaufvarianten des Flash-Vorgangs

Zur Ansteuerung sollen zwei Möglichkeiten zur Verfügung gestellt werden. Zum einen ein Win32 Programm, welches über ein am PC angeschlossenes CAN-Interface mit dem Bus verbunden werden kann, um so mit dem Steuergerät zu kommunizieren. Da diese Methode einen PC im Auto oder in direkter Umgebung voraussetzen würde, wurde noch eine weitere Möglichkeit der Ansteuerung angedacht: Ein in einem schon früher gestarteten Teilprojekt entwickelter Webserver, der in eine Applikation auf einem anderen Steuergerät integriert wurde und über ein GPRS-Modem mit dem Internet verbunden ist, soll als Gegenstück zum Flashloader in normalen Steuergeräten fungieren. Dabei soll die bestehende Weboberfläche um eine File-Upload Funktionalität erweitert werden, über die eine Applikation auf den Webserver geladen werden kann. In einem weiteren Schritt soll es möglich sein, diese hochgeladene Applikation dann über den CAN-Bus auf ein anderes Steuergerät zu flashen.

2.3 Optionaler Bearbeitungsteil

Im Zuge der relativ langsamen Übertragungsrate und immer größer werdenden Applikationen soll in dieser Arbeit noch die Machbarkeit einer Modularisierung untersucht werden, mit deren Hilfe es möglich wäre, nur bestimmte Teile der Gesamtapplikation auszutauschen und so die Größe der zu übertragenden Datenmengen auf die viel kleineren Teilmodule zu reduzieren. Ein weiteres Ziel wäre das dynamische Austauschen solcher Module während des Betriebs, um etwaige Ausfallzeiten, die durch einen Reset des Geräts entstehen würden, zu vermeiden.

3 Zugrunde liegende Hard- und Software

3.1 Übersicht über die verwendeten Komponenten

Wie in der Einleitung schon beschrieben wurde, besteht das System, das bei dieser Arbeit Verwendung findet, aus verschiedenen Komponenten. Neben einer speziellen Entwicklungsumgebung und der eigentlichen ECU kommt dabei ein Zusatzgerät zum Einsatz, durch dessen Hilfe das Entwickeln und Testen der Software auf der ECU ermöglicht wird.

3.1.1 Technische Merkmale der ECU

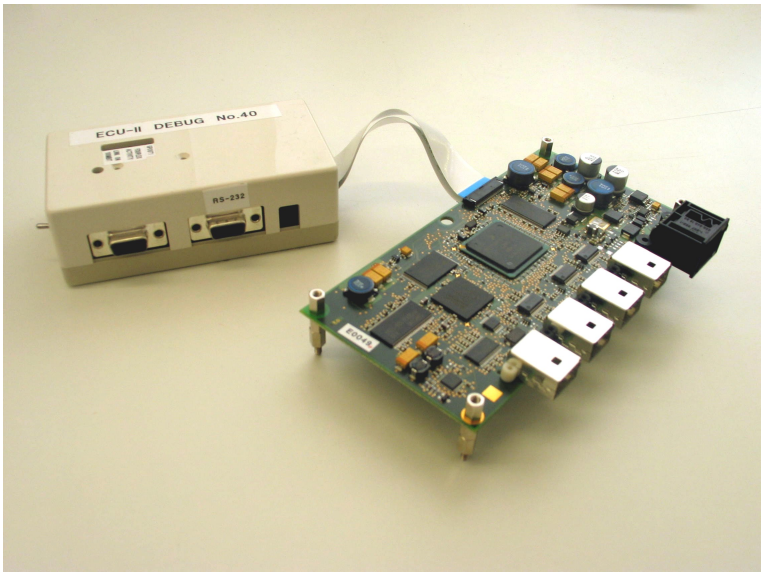

Abbildung 3.1: ECU mit Debug-Addon für JTAG und Debug-RS232 Schnittstelle

Bei dem in der ECU (Abbildung 3.1, rechts) verbauten Prozessor handelt es sich um den von

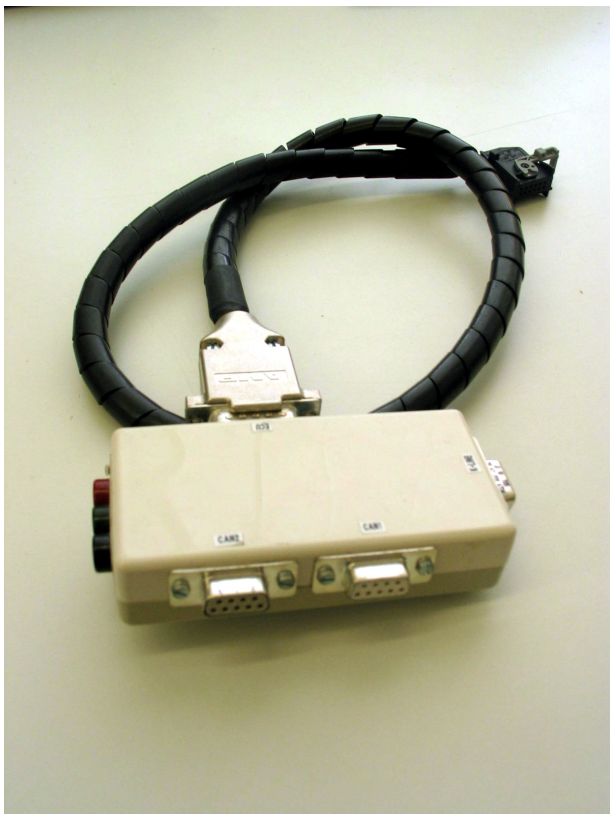

Abbildung 3.2: ECU-Adapter für Stromversorgung und Anschlussmöglichkeiten für die 2 CAN-Schnittstellen

Freescale stammenden PowerPC Prozessor *MCP5200*. Er kann mit max. 400Mhz betrieben werden und wurde hauptsächlich wegen seiner breiten Palette an integrierter Peripherie gewählt. Neben sechs RS232 Schnittstellen bietet er noch eine Ethernet-Schnittstelle, USB 1.1 und zwei CAN-Bus Schnittstellen. Verbaut wurde der Prozessor auf einem eigens von Siemens VDO entwickeltem Board.

Als Speichermöglichkeiten stehen neben 8 MB RAM auch ein auf der Platine integrierter Flash-Baustein mit einer Kapazität von 2 MB zur Verfügung. Der Flash-Speicher dient dabei der dauerhaften Speicherung der Applikation und zusätzlich benötigten Parametern. Nach dem Starten der ECU kann die Applikation dann in den RAM kopiert werden.

Nicht die komplette Peripherie wird wird für den späteren Betrieb benötigt, weshalb nicht alles mit einem Anschluss auf der Platine versehen wurde. Neben vier RS232 Schnittstellen für den Anschluss von Kameras wurden nur noch die beiden CAN-Interfaces zur Verfügung gestellt

(Siehe Abbildung 3.2). Zu Debug-Zwecken wurde jedoch ein Zusatzgerät (Abbildung 3.1, links) entwickelt, welches über 2 Flachbandkabel mit der ECU verbunden werden kann und neben einer weiteren RS232 Schnittstelle noch das JTAG-Interface[1] des Prozessors bereitstellt.

3.1.2 Der Hardware-Debugger

Abbildung 3.3: Hardware-Debugger „BDI2000“ der Abatron AG

Bei dem Hardware-Debugger handelt es sich um ein Gerät der Abatron AG [1] mit der Bezeichnung BDI2000. Wie in der Einleitung schon erwähnt, wird dieses Gerät dafür verwendet, die für embedded Geräte geschriebene Software auf dem Gerät selbst zu debuggen. Dabei wird dem Entwickler über diesen Hardware-Debugger die vollkommene Kontrolle über den Prozessor gegeben. Er kann ihn anhalten, Breakpoints setzen, dessen Register lesen und schreiben und den erstellten Code Schritt für Schritt ausführen lassen (Durch-Steppen). Auf diese Weise können wie bei der Entwicklung einer PC-Anwendung Programmierfehler entdeckt und behoben werden.

[1] „Joint Test Action Group mit Hilfe dieser Schnittstelle ist es möglich, die komplette Software, die auf dem Prozessor aufsetzt, zu debuggen; Eine genauere Beschreibung befindet sich unter [13]

Die Anbindung des Geräts an den Entwickler-PC erfolgt dabei wahlweise über eine RS232 Schnittstelle oder über Ethernet. Für die Ansteuerung des Prozessors wird eine speziell zugeschnittene Firmware verwendet, die in diesem Fall nur mit Prozessoren der Reihe MPC52xx zusammenarbeitet. Der Hersteller stellt jedoch auch für viele weitere Familien Firmware-Versionen bereit, welche auf das BDI aufgespielt werden können. Die Ansteuerung von Seiten des PCs erfolgt direkt aus der IDE[2] heraus.

3.1.3 Das CAN-Interface von *Softing*

Abbildung 3.4: Softing CAN-Interface für die USB-Schnittstelle

Das CAN-Interface, das während dieser Arbeit an einem Windows-Rechner betrieben wird, heisst *CANusb* und stammt von *Softing* [10]. Dieses Interface verbindet einen Windows-Rechner mit einem CAN 2.0 Netzwerk und stellt einer Anwendung einen Port zur Verfügung. Die Ansteuerung erfolgt dabei über eine vom Hersteller mitgelieferte Standard-API in Form einer C++ - DLL. Diese kann in jegliche Programme eingebunden und verwendet werden.

3.1.4 Die Entwicklungsumgebung *CodeWarrior*

Die Entwicklungsumgebung stammt wie der Prozessor von Freescale. Neben den üblichen Fähigkeiten einer IDE besitzt der CodeWarrior auch einen eigenen Compiler, der auf den eingesetzten Prozessor optimal abgestimmt ist.

[2]Integrated Developement Environment

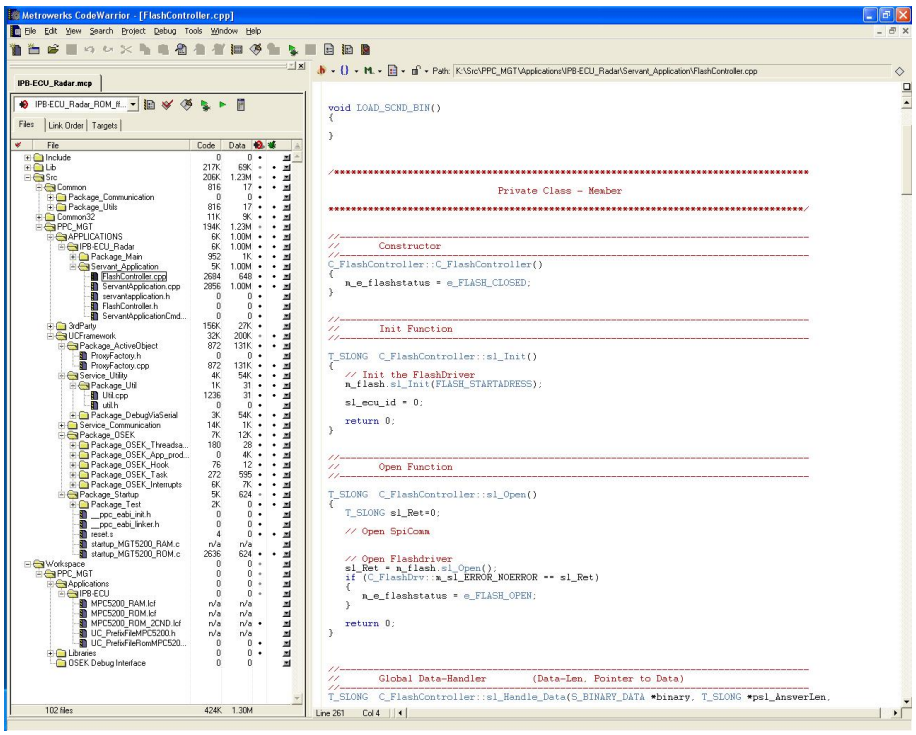

Abbildung 3.5: Oberfläche der IDE *CodeWarrior*

3.1.5 GPRS Modem

Um die ECU, die als Webserver fungiert, an das Internet anzubinden, wird ein zusätzliches Modem verwendet, über das sich die ECU in das Internet einwählen kann. Dabei wird ein GPRS-Modem verwendet, welches eine normale SIM-Karte für ein Handy nutzt.

3.1.6 Debuggen über die RS232 Schnittstelle

Ein großes Problem bei einem embedded System ist es, aufgetretene Fehler während des Betriebs mitzubekommen, da kein Monitor oder Ähnliches angeschlossen ist, auf dem man Fehlermeldungen ausgeben könnte. Abhilfe wurde hier geschaffen, indem einer der freien RS232 Ports für Debug-Ausgaben reserviert wurde. Über einen speziellen in die Applikation integrierten Debug-Treiber können nun Debug-Ausgaben auf diesem Port ausgegeben und auf einem PC angezeigt werden.

Im Umfeld dieser Arbeit wird hierfür das Programm *Filterterminal* verwendet, das nicht nur die

Ausgaben auf dem COM-Port mitloggen kann, sondern auch noch Filter beherrscht. So können Fehlerausgaben und sonstige Ausgaben getrennt angezeigt werden.

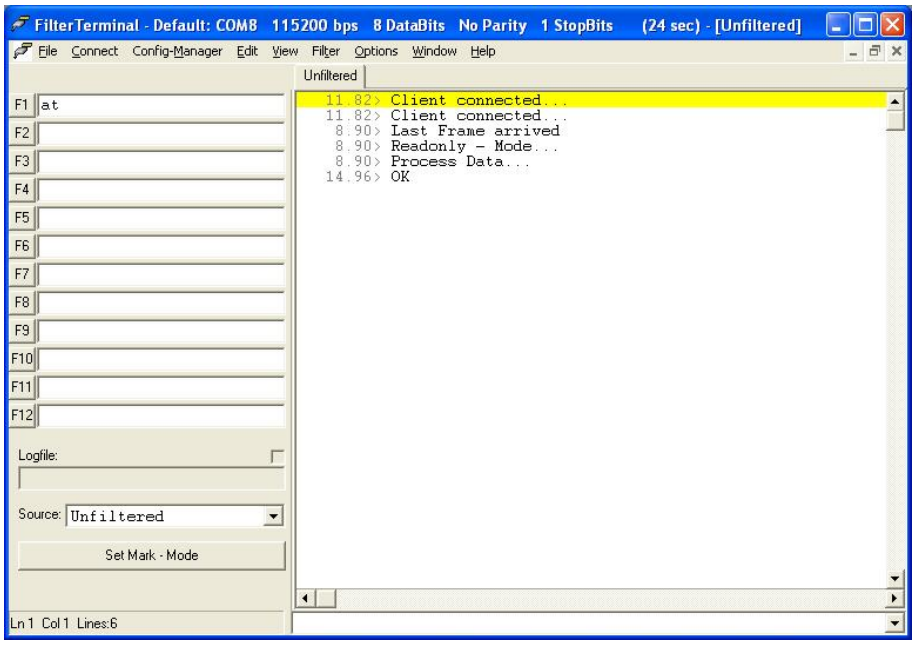

Abbildung 3.6: Oberfläche des Filterterminals

3.2 Das Übertragungsmedium CAN

3.2.1 Allgemeines

Dieses Kapitel befasst sich mit einem der wichtigsten Teile dieser Arbeit, dem CAN-Bus[3]. Er ist bis dato das verbreitetste Bus-System in der Automobilbranche und bis dato in fast jedem Auto vertreten. Grund dafür sind die relativ kostengünstigen Protokoll-Bausteine und integrierten Protokoll-Schnittstellen für so ziemlich alle wichtigen Microcontroller-Familien (vgl. [4] Seite 34). Diese Argumente und die daraus resultierenden hohen Stückzahlen in der Automobilindustrie sorgten dafür, dass sich dieses System auch auf anderen Anwendungsgebieten ausbreitete.

Aber nicht nur der günstige Preis allein war dafür verantwortlich, dass der CAN Bus so erfolgreich wurde. Auch seine Leistungsmerkmale und Eigenschaften waren maßgeblich daran beteiligt, was im folgenden Kapitel näher erläutert werden soll.

3.2.2 Aufbau

Im CAN-Bus finden sich sich viele Aspekte, die den Anforderungen an Kommunikationssysteme für mobile Systeme Rechnung tragen. In den folgenden Kapiteln soll jedoch nur auf diejenigen Eigenschaften eingegangen werden, die maßgeblich zum Verständnis der restlichen Arbeit beitragen werden. Weiterführende Angaben, die den CAN-Bus betreffen, können unter [4] in sehr ausführlicher FOrm nachgelesen werden.

Das wohl wichtigste Merkmal des CAN-Protokolls ist die nachrichtenorintierte Adressierung. Nicht Sender und Empfänger besitzen eine Adresse, sondern die Datenpakete (sog. CAN-Identifier[4]). Dies ermöglicht es, bestimmte Informationen durch lediglich ein Paket allen interessierten Busteilnehmern zukommen zu lassen. Besonders in Verbindung mit der Multi-Master Fähigkeit, bei der jeder Busteilnehmer gleichberechtigt ist, lässt sich die Buslast sehr gut reduzieren, da jeder Teilnehmer seine Daten bei Bedarf senden kann und ein Polling-Betrieb[5] vermieden wird. Das Anfordern von Daten ist hier auch durch einen sog. Remote-Frame möglich siehe Kapitel 3.2.3.

Im Multi-Master Betrieb stellt sich jedoch die Frage, wie der Zugriff der Teilnehmer geregelt wird,

[3] Controller - Area - Network
[4] CAN IDs; Länge entweder 11 Bit im Standardformat und 29 Bit im erweiterten Format
[5] Geziehltes, wiederholtes Anfordern von Daten durch einen Teilnehmer

wenn mehrere Teilnehmer gleichzeitig Zugriff benötigen. Zunächst einmal muss geklärt werden, zu welchem Zeitpunkt überhaupt ein neues Telegramm gesendet werden kann. Ein sendebereiter Teilnehmer erkennt dies an einer definierten Ruhezeit, die zwischen zwei Paketen immer eingehalten werden muss. Diese Ruhezeit beträgt 11 Bits und gewährleistet, dass der Bus bereit für ein neues Telegramm ist. Ist nun die Ruhephase vorbei, haben theoretisch alle Teilnehmer das Recht, mit dem Senden ihrer Nachricht zu beginnen. Um eine Kollision zu verhindern, gewährleistet eine Arbitrierungsphase[6] zu Beginn der Nachricht, dass nur ein Teilnehmer am Ende seine Nachricht sendet. Während dieser Phase werden Bit für Bit der Nachricht, beginnend mit dem Nachrichten-Identifier, mit dem Pegel verglichen, der tatsächlich am Bus anliegt. Legt der Teilnehmer einen rezessiven Pegel am Bus an und stellt dann beim Gegenlesen einen dominanten Pegel fest, weil ein anderer Teilnehmer eine Nachricht mit höherer Priorität als der eigenen sendet, stellt dieser das Senden sofort ein. Dies geschieht mit allen Teilnehmern, die senden wollen, bis am Ende nur noch einer übrig bleibt, der dann das Recht erhält, seine Nachricht zu schicken. Eine Kurzübersicht eines Arbitrierungsvorgangs zwischen 3 sendebereiten Teilnehmern ist auch in Abbildung 3.7 noch einmal zu sehen. Ein großer Vorteil dabei ist es, dass durch dieses Arbitrierungsverfahren keine Nachrichten zerstört werden. Bei anderen Buszugriffsverfahren besteht das Problem, dass im Bedarfsfall eine Nachricht noch einmal gesendet werden muss, wenn ein anderer Teilnehmer versucht, seine Nachricht zu senden. Eine weitere Eigenschaft des CAN Bus ist die definierte maximale Länge des Datenblocks. Dieser darf höchstens 8 Byte betragen und garantiert den Teilnehmern kurze Wartezeiten. Die Länge wurde im Hinblick auf die meist relativ kurz gefassten Nachrichten im Automobilbereich festgelegt, wo es vielmehr auf eine hohe Nachrichtenrate ankommt als auf die Übertragung großer Datenblöcke.

3.2.3 Nachrichtenformate

Wie im Kapitel 3.2.2 schon einmal angesprochen, gibt es mehrere Nachrichtenformate, die unterschiedliche Ziele verfolgen. Die in dieser Arbeit benötigten Formate sind zum einen das Standard-Datentelegramm (Data-Frame) und das Datenanforderungsformat (Remote-Frame).

Ein Data Frame ist für das Übertragen von Datenblöcken zuständig und somit das am meisten genutzte Nachrichtenformat. Bei diesem Nachrichtentyp sendet die Datenquelle eine Nachricht an einen oder mehrere Empfänger, wobei die Nutzdaten eine Länge von 0 - 8 Byte haben dürfen.

[6] Phase, in welcher der sendeberechtigte Teilnehmer ermittelt wird

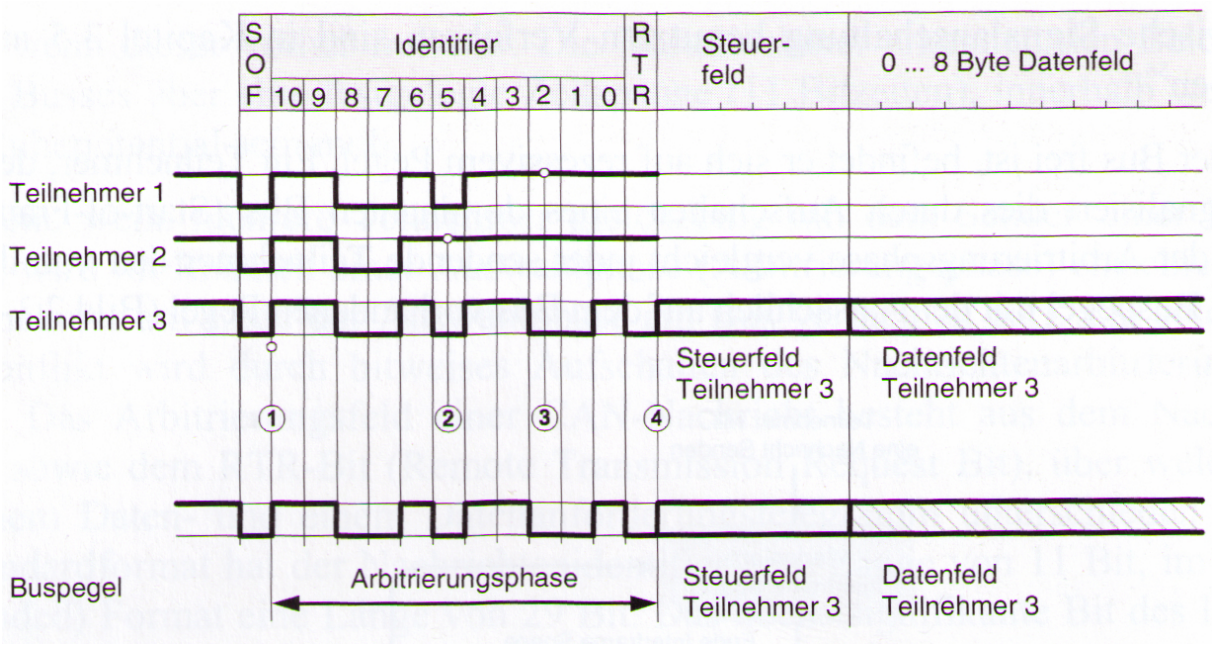

Abbildung 3.7: Beispiel eines Arbitrierungsvorgangs; Die Teilnehmer 1, 2 und 3 beginnen gleichzeitig einen Arbitrierungsversuch (1). Teilnehmer 2 verliert zum Zeitpunkt (2), Teilnehmer 1 zum Zeitpunkt (3) das Buszugriffsrecht. Beide Teilnehmer gehen in den Empfangsmodus. Am Ende (4) besitzt nur noch Teilnehmer 3 das Buszugriffsrecht (Siehe [4] Seite 56)

Neben einem Startbit[7], einem Stopbit[8], einer Checksum und einem Identifier gibt es noch ein Anforderungsbit (RTR-Bit[9]), welches direkt nach dem Identifier folgt und bei der Busarbitrierung mit berücksichtigt wird. Dieses Bit ist bei einem Datentelegramm dominant, wodurch es einem Datenanforderungstelegramm mit gleichem Identifier vorgezogen wird.

Ein Remote Frame dient einem Teilnehmer zum gezielten Anfordern von Informationen von einem oder mehreren Teilnehmern. Dabei fungiert der Identifier als Information, welche Nachricht benötigt wird. Der Aufbau dieses Telegramms entspricht im großen und ganzen dem des Data Frames mit dem Unterschied, dass das RTR-Bit nicht dominant[10] ist, sondern rezessiv und die

[7] Start-of-Frame, SOF
[8] End-of-Frame, EOF
[9] Remote Transmition Request
[10] Ein dominanter Pegel wird durch eine logische „1“ repräsentiert, ein rezessiver Pegel durch eine logische „0“ auf dem Datenbus

Länge des Nutzdatenbereichs 0 beträgt. Ein Remote Frame kann also nie Informationen übertragen sondern nur anfordern. Abbildung 3.8 auf Seite 17 veranschaulicht den Aufbau des Daten- und Datenanforderungstelegramms noch einmal ausführlich am Beispiel eines Standardframes mit 11-Bit Identifier.

Zusätzlich zu diesen Telegrammtypen gibt es noch zwei weitere, die an dieser Stelle der Vollständigkeit halber genannt werden sollen. Zum einen das Fehlertelegramm (Error Frame) und das Überlastungstelegramm (Overload Frame).

Ein Error Frame wird immer dann gesendet, wenn ein Teilnehmer einen Fehler bei einer durch einen anderen Teilnehmer gesendeten Nachricht entdeckt hat. Hierzu dient eine Bitsequenz von 6 Bits mit gleicher Polarität[11], die so in dieser Form im normalen Betrieb nicht auftreten kann. Erkennt ein Teilnehmer nach dem Senden seiner Nachricht einen solchen Error Frame, hat er die Möglichkeit, den Sendevorgang zu wiederholen. Kommt der Fehler jedoch mehrmals hintereinander, hat der sendende Teilnehmer die Möglichkeit, sich selbst vom Bus zu trennen und so den Betrieb nicht weiter zu stören.

Ein Overload Frame dient dazu, eine Verzögerung des nächsten Datentelegramms beim Sender anzufordern.

[11] Jeweils nur dominante ode rezessive Bits

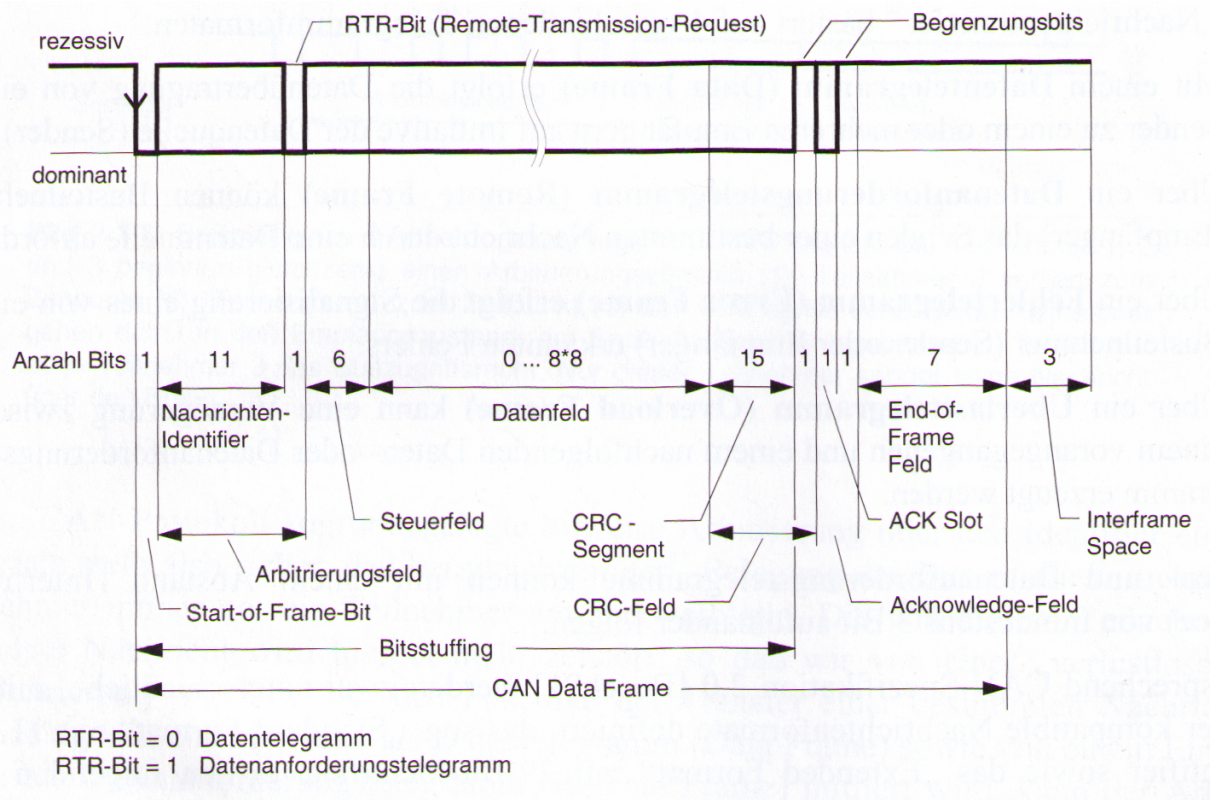

Abbildung 3.8: Aufbau des Data Frames bzw. des Remote Frames mit Standard-Identifier ([4] Seite 58)

3.3 Der Flash-Speicher

Der Flash-Speicher ist ein elektronisch lösch- und programmierbarer Speicher, der sich in den letzten Jahren in immer mehr Einsatzgebieten etabliert hat. So wird er vor allem in mobilen Geräten eingesetzt, um dauerhaft Informationen zu speichern. Die Anwendungspalette reicht dabei von USB-Sticks über MP3-Playern bis zum Einsatz als Festplattenersatz bei PCs. Vor allem die benutzerfreundliche Anbindung an bestehende Systeme im PC-Bereich erleichtert die Handhabung, da der Speicher wie eine Festplatte oder Diskette angesprochen werden kann.

Aber auch im Automotive-Bereich wird der Flash-Speicher bereits umfangreich genutzt und dient als Festspeicher für Applikationen und deren Zusatzdaten wie Konfigurationen.

Die folgenden Kapitel beschäftigen sich mit den wichtigsten Eigenschaften des Flashspeichers und zeigen die Vor- und Nachteile.

3.3.1 Funktionsweise

Aufgrund seiner technischen Gegebenheiten unterliegt der Flash-Speicher im Vergleich zu anderen Dauerspeichern einigen Begrenzungen, die besondere Anforderungen an den Flashtreiber stellen.

Einordnen lässt sich der Flash-Speicher als eine Spezialform von EEPROM[12] und verwendet hauptsächlich zwei verschiedene interne Organisationsstrukturen: NOR und NAND. Der Unterschied der beiden Typen besteht in der Anschlussart der Speicherzellen an die Bitleitungen. Während Speicherzellen bei NOR-Speicher parallel an die Bitleitung angeschlossen sind, werden bei NAND-Speicher mehrere Zellen hintereinander an eine Bitleitung angeschlossen. Der Vorteil der NOR-Technik ist der schnelle und wahlfreie Zugriff auf bestimmte Speicherzellen, während das Löschen und Schreiben sehr langsam ist. Bei der NAND-Technik geschieht das Löschen und Schreiben sehr zügig, während der Lesezugriff sequentiell und nicht direkt erfolgt. Daher sind die beiden Speicher-Typen in verschiedenen Einsatzgebieten angesiedelt. NAND-Speicher wird hauptsächlich als reiner Datenspeicher verwendet (z.B. USB-Sticks oder Ähnliches) und NOR-Speicher als Programmspeicher.

Der in dieser Arbeit verwendete NAND-Speichertyp ist wegen der Anordnung seiner Speicherzellen in sog. Blöcke aufgeteilt, die jeweils 64kb groß sind. Nur jeweils der erste oder letzte Block ist noch einmal in kleinere Teile unterteilt, in jeweils einen 32kb Block, zwei 8kb Blöcken und einen 16kb Block, was man jeweils als Bottom- bzw. Top-Flash bezeichnet. In dieser Arbeit wird ausschließlich Top-Flash-Speicher verwendet. Im Anhang B auf Seite 78 ist für diesen Flash-Typ ein kurzes Datenblatt zu finden mit einer genauen Beschreibung der Sektoren-Aufteilung.

Das Schreib-, Lese- und Löschverhalten des Flash-Speichers unterscheidet sich zwischen herkömmlichen Speichermedien. Bevor man an eine Speicherstelle schreiben kann, muss der Block, in dem sich die Adresse befindet, gelöscht werden. Hierbei werden die Speicherzellen in diesem Block auf eine logische 1 gesetzt. Wenn dann Daten geschrieben werden, kann nur der Übergang von 1 auf 0 erfolgen, nicht umgekehrt. Wenn also nur ein kleiner Teil eines Blocks aktualisiert werden muss, ist es Aufgabe des Flash-Treibers, die anderen Speicherzellen, die sich in diesem Block befinden, zu sichern und nach dem Löschvorgang wieder zurück zuschreiben.

[12] Electrical Eraseable Programmable ROM

3.3.2 Vor- und Nachteile

Die größten Nachteile des Flash-Speicher sind die aus seinem Aufbau resultierenden Gegebenheiten. Nur durch vorheriges Löschen von ganzen Blöcken lassen sich einzelne Adressen beschreiben und sinnvoll nutzen. Dabei muss immer darauf geachtet werden, dass andere Daten im gleichen Block nicht verloren gehen. Ebenfalls ein Problem dabei ist der auftretende Verschleiß von Speicherzellen. Sie besitzen nur eine maximal garantierte Lebensdauer und können nicht beliebig oft gelöscht werden. Ist diese Grenze erreicht, sind sie nicht mehr in der Lage, die gespeicherten Daten zuverlässig zu halten. So ist eine ausgeklügelte Verwaltung der Speicherzellen notwendig, um ausgefallene Zellen durch ein Mapping-Verfahren zu ersetzen. Auch das übermäßige Beanspruchen von einigen wenigen Speicherzellen sollte vom Treiber unterbunden werden. Stattdessen sollte auf andere Speicherzellen ausgewichen werden, um so eine ungleichmäßige Abnutzung zu vermeiden.

Auf der anderen Seite gibt es auch eine Reihe von Vorteilen, die letztendlich den Einsatz von Flash-Speicher rechtfertigen. So lassen sich große Datenmengen in sehr kompakter Bauweise unterbringen. 2GB und mehr sind z.B. in USB-Speichersticks keine Seltenheit. Daher ist es auch möglich, solche Speicherchips direkt auf Platinen mit zu verlöten und auf diese Weise erheblich Platz einzusparen. Ein weiterer Vorteil ist, dass Flash-Speicher ohne bewegliche mechanische Teile auskommt, wie es z.B. in Festplatten der Fall ist. Hier kommen sich drehende Scheiben zur Datenspeicherung zum Einsatz, welche von einem mechanischen Arm abgetastet werden um Daten zu lesen bzw. zu schreiben. Dies ist beim Flash-Speicher nicht der Fall und macht ihn deshalb vollkommen unempfindlich gegen Stöße jeglicher Art. Insbesondere in Fahrzeugen ist dies ein großer Vorteil, da die Steuergeräte nicht speziell gesichert werden müssen.

3.4 Das Betriebsystem OSEK/VDX-OS

Der Begriff OSEK/VDX-OS steht für ein ganzes Paket an Spezifikationen, welche 1995 erstmals von einem Gremium geschaffen wurde, um Standards für die Elektronik im Kraftfahrzeug zu schaffen. Dabei steht die Abkürzung OSEK für „Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug“ und VDX für „Vehicle distributed executive“. In diesem Zusammenhang wird jedoch der Einfachheit wegen nur von „OSEK/OS“ gesprochen.

In den folgenden Kapitel wird das eingesetzte OSEK-konforme Echtzeitbetriebssystem und dessen Erweiterungen etwas genauer erklärt.

3.4.1 Beschreibung des OSEK/OS Standards

Allgemein beschreibt der OSEK/OS Standard ein statisches Echtzeitbetriebssystem, bei dem alle wesentlichen Bestandteile bereits vor der Übersetzungszeit bekannt sind. So ist es nicht möglich, dynamisch während der Laufzeit Speicher anzufordern und zu nutzen. Dabei werden auch schon im Vorfeld die benötigten Tasks definiert mit all ihren Eigenschaften wie der Task-Priorität oder den benötigten Stack-Größen. Neben den Tasks, in denen die eigentlichen Applikationen ablaufen, werden auch betriebssystemeigene Teile bei Bedarf eingebunden wie spezielle Hardware-Treiber oder sonstige Addons.

Nach dieser Konfiguration, die üblicherweise über ein grafisches Tool erledigt wird, werden Kernel und die selbst erzeugten Sourcen zu einer Applikation zusammengebaut. Dies geschieht üblicherweise während eines iterativen Prozesses, da teilweise die Änderungen im eigenen Code eine Änderung am Kernel erfordern, beispielsweise um den Stack eines Tasks zu vergrößern oder um Task-Prioritäten zu verändern.

Auf diese Art ist eine optimale Anpassung des Betriebssystems an seine Umgebung gewährleistet. Es werden nur absolut notwendige Subsysteme in den Kernel eingebunden, die gewährleisten, dass der erzeugte Programmcode in relativ kurzer Zeit einen stabilen Zustand erreicht.

3.4.2 Inter-Prozess Komunikation

Da der OSEK/OS Standard für die bei Siemens VDO benötigten Zwecke ein nur unzureichendes Mittel zur Kommunikation zwischen den einzelnen Prozessen vorsieht, wurde für diesen Zweck ein Addon unter dem Namen „Active Object“ in das Betriebssystem integriert, welche die Restriktionen durch die schon integrierten Kommunikationsmittel aufhebt und den Austausch von Nachrichten und Daten zwischen den Prozessen zulässt. Zu diesem Zweck wurden Standard-Schnittstellen geschaffen, welche die schon vorhandenen Schnittstellen erweitern.

3.5 Das SRecord Format

Das SRecord-Format wurde vor einigen Jahren von Motorola entwickelt. Es dient zur Übertragung von Binärdateien von einem Host-Rechner zu einem Empfänger. In diesem Kontext wird es vorzugsweise dazu verwendet, um Applikationen auf ein eingebettetes System zu übertragen (z.B. Steuergeräte).

Hierzu werden die Binärdaten einer Applikation in sog. SRecords umgewandelt. SRecords sind speziell formatierte ASCII Strings, die in einer Datei zeilenweise eingetragen werden. Die Darstellung von Zahlenwerten erfolgt im HEX-Format, wobei jede Stelle der HEX-Zahl als eigenes ASCII Zeichen dargestellt wird.

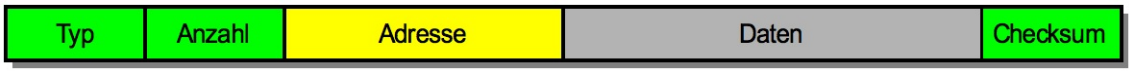

Abbildung 3.9: Aufbau eines SRecord Strings

Abbildung 3.9 verdeutlicht den Aufbau eines solchen Strings. Das erste Element stellt dabei den Typ dieses SRecords dar, wovon es insgesamt acht verschiedene gibt. Bezeichnet werden sie durch ein „S“ und eine nachfolgende Nummer, womit dieses Feld eine Länge von 2 Byte besitzt. Das zweite Feld enthält die Länge des SRecord Strings, ebenfalls in einer Länge von 2 Byte. Das Feld „Adresse“ enthält die Startadresse, ab der die folgenden Bytes auf dem Empfängersystem geschrieben werden sollen. Dieses Feld ist, je nach SRecord-Typ 4, 6 oder 8 Byte groß. Das Datenfeld darf eine maximale Länge von 64 Byte haben und dient der Übertragung der eigentlichen Nutzdaten. Aufgrund der Darstellungsweise der Zahlenwerte können pro SRecord maximal 32 Byte Nutzdaten übertragen werden. Das letzte Feld enthält eine 8bit Checksum des kompletten SRecord-Strings und belegt, wie Typ und Länge 2 Byte.

Für jedes SRecord ist eine Startadresse definierte, die auf Empfängerseite ausgewertet wird. So ist die Einhaltung einer bestimmten Reihenfolge in einer Datei nicht nötig. Ein weiterer Vorteil dieser Art der Übertragung von Applikationen ist es, dass bestimmte Bereiche davon, die keinen Programmcode oder sonstige Nutzdaten enthalten, von der Erzeugung der SRecords ausgenommen werden können. Beispiel hierfür ist der Bereich einer Applikation, der vom Linker für das Speichern von Variablen angedacht ist. Sie enthalten anfänglich keine Daten und müssen auf

Empfängerseite auch nicht geschrieben werden. So kann die Übertragung auf die notwendigsten Daten beschränkt werden.

Im Kontext dieser Arbeit werden SRecords genutzt, um aus der übersetzten Applikation ein sog. MOT-File[13] zu erzeugen, das aus SRecord-Einträgen besteht und durch ein Tool über die JTAG-Schnittstelle auf die ECU geflasht wird. Die Erzeugung wird dabei automatisch nach dem Übersetzungsvorgang von der IDE[14] übernommen. Mit dieser Technik ist es aber auch möglich, gezielt weitere Informationen in den Flash-Speicher abzulegen, wie z.B. Einstellungen, die die Hardware betreffen. Diese Informationen müssen zuerst mittels eines HEX-Editors in eine Binärdatei geschrieben und dann durch ein kleines Tool in ein MOT-File konvertiert werden. Danach können sie gewohnt wie eine Applikation auf die ECU übertragen werden.

[13] Ein Mot-File besteht aus SRecords, erzeugt aus einer Applikation
[14] Freescale CodeWarrior

4 Entwurf der Systembestandteile

4.1 Der Entwurf der Bestandteile im Überblick

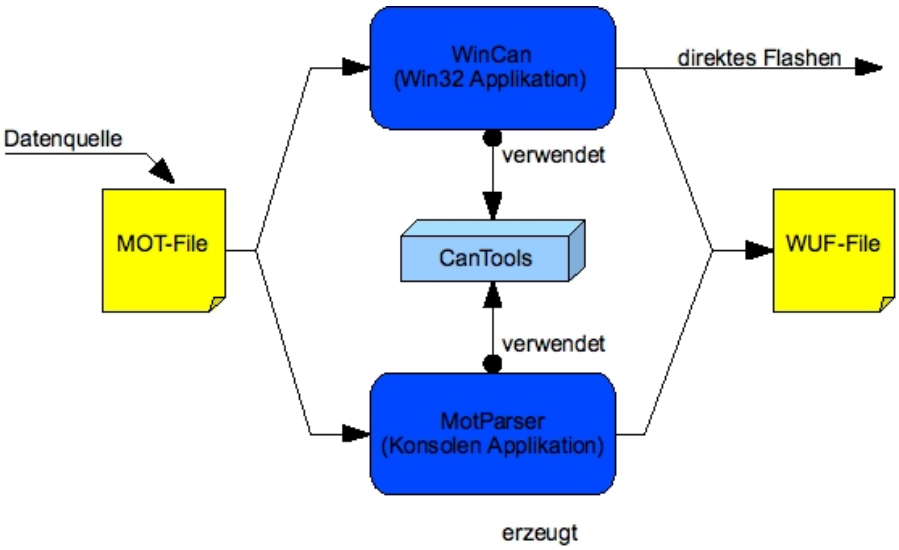

Abbildung 4.1: Aufbau der PC-seitigen Applikationen

Wie in Abbildung 4.1 zu sehen ist, wurden 2 PC-Applikationen entworfen. Dabei stellt die Anwendung ***WinCan*** die eigentliche Windows-Anwendung dar, die in der Aufgabenstellung gefordert war. Die zweite Anwendung ***MotParser*** ist ein Kommandozeilen-Programm, welches sich auf das Erstellen von ***WUF***[1]-Files beschränkt und für den automatisierten Batch-Betrieb entwickelt wurde.

Um die Funktionalitäten, die in beiden Applikationen benötigt werden, nicht doppelt implementieren zu müssen, wurden diese in ein eigenes Modul (DLL) ausgelagert, welches von beiden

[1]WUF: Web Upload File

Programmen benutzt werden kann.

Das verwendete ***WUF***-Dateiformat wurde im Kontext dieser Arbeit entworfen und dient dem Zweck der Kapselung aller zum Flashen benötigten Daten. Es wurde hauptsächlich zur Verwendung des Uploads auf eine Webserver-ECU eingeführt, da hier sonst die für den Flashvorgang benötigten Zusatzinformationen per Hand in ein Webseiten-Formular eingegeben werden müssten, was eine große Fehlerquelle darstellt und auch sehr unpraktisch ist.

4.2 Definition des WUF Dateiformats

Wie in Kapitel 4.1 schon erwähnt, soll das WUF-Dateiformat in erster Linie der Kapselung der Informationen dienen, die sowohl auf der Empfänger-ECU als auch auf der Webserver-ECU benötigt werden. So enthält es neben der zu flashenden Applikation noch einen 16 Byte langen Header. Die Abbildung 4.2 zeigt einen Überblick über den genauen Aufbau des Headers.

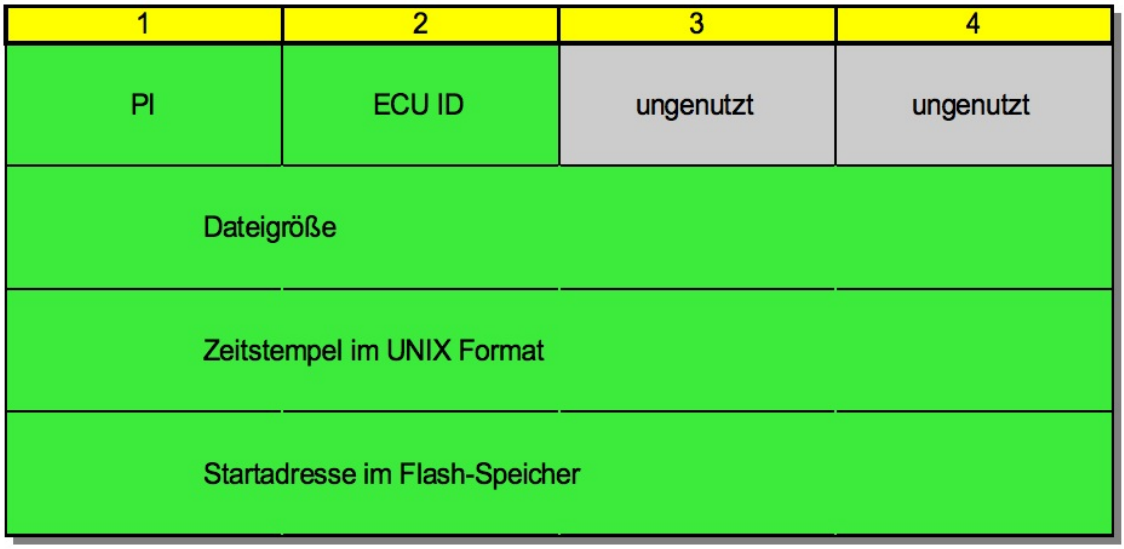

Abbildung 4.2: Header des WUF Dateiformats

Paket - Identifier

Der erste Parameter ist ein 1-Byte großer PI (Paket-Identifier), der den Typ der im WUF-File untergebrachten Binärdaten angibt. So wird es möglich, nicht nur Applikationen zu flashen, sondern auch andere Dateien, wie zum Beispiel Bilder oder Ähnliches. Zum Zeitpunkt der Fertigstellung der Arbeit gibt es 2 verschiedene Typen:

PI 0 Definiert eine Standard-Applikation. Es werden alle Zusatzinformationen berücksichtigt und auf Empfängerseite in den dafür vorgesehenen Flags-Bereich in den Flash-Speicher geschrieben

PI 1 Definiert ein anderes Dateiformat. Es wird nur die Startadresse berücksichtigt, an welche die Nutzdaten geschrieben werden sollen. Über den genauen Inhalt der Nutzdaten gibt es keine näheren Informationen

Eine spätere Erweiterung dieser Liste ist somit gegeben und ermöglicht es, die Sendedaten noch weiter zu differenzieren. Die Einführung dieser Typisierung war in erster Linie für eine spätere Weiterentwicklung dieser Arbeit vorgesehen und nicht zwingend erforderlich, wodurch sich der Entwurf auf die Unterscheidung zwischen einer Applikation und einer Nicht-Applikation beschränkt.

ECU - ID

Der zweite Parameter, ebenfalls 1-Byte groß, definiert den Identifier der Empfänger-ECU. Er wird hauptsächlich zur Verarbeitung im Webserver verwendet (Die Verarbeitung der Daten sowohl in Webserver-Applikation als auch auf der Empfänger-ECU werden in den folgenden Kapitel (4.7, 4.8) noch näher beschrieben).

Ungenutzter Platz

Die folgenden 2 Byte sind bis dato ungenutzt und für eventuell zukünftig benötigte Parameter reserviert. Als nächstes folgen jeweils drei 32bit Parameter. Zuerst die Dateigröße des eingelesenen Binaries, dann ein Zeitstempel im Unix Format aus der Erstellungszeit und zuletzt die gewünschte Startadresse im Flash-Speicher, ab der die Nutzdaten geschrieben werden sollen.

Dateigröße

Die Größe im Header dient neben der Checksum zur zusätzlichen Validierung der Datei auf Empfängerseite. Sie wird außerdem noch in den Flash-Speicher geschrieben, um anderen Tasks, die den Flash-Speicher beschreiben, die Möglichkeit zu geben, den noch freien Speicherplatz zu berechnen.

UNIX Zeitstempel

Der Zeitstempel wird, wie die Dateigröße, in den Flash-Speicher geschrieben. Er dient nur als Zusatzinformation für den Entwickler. An Hand dieses Werts kann genau festgestellt werden, wann die installierte Applikation erstellt wurde und verhindert ein unnötiges Flashen von zwei gleichen Applikationen.

Startadresse

Die Startadresse wird aus zwei Gründen mit übertragen. Zum einen gibt es dem Entwickler eine einfache Möglichkeit zu bestimmen, an welche Stelle die Applikation (oder sonstige Binärdaten) in den Flash-Speicher geschrieben werden sollen. So wird eine feste Codierung im Quellcode vermieden. Zum anderen wird die Adresse vom Bootloader zum Starten der Applikation nach einem Reset benutzt. Dazu wird sie, wie schon die vorigen zwei Parameter, in den Flash-Speicher geschrieben, wo sie auch ohne die Nutzung eines Flashtreibers gelesen werden kann.

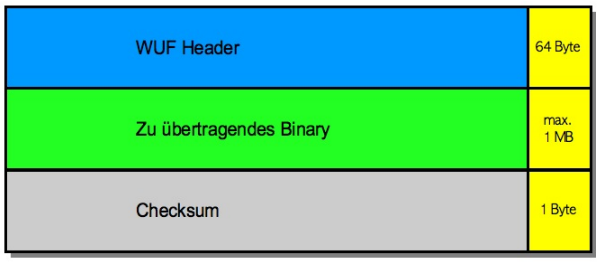

Abbildung 4.3: Das WUF File-Format im Überblick

Direkt im Anschluss an den Header folgen die eigentlichen Nutzdaten welche zur Zeit nur eine maximale Länge von 1024 kB haben dürfen, und eine 8 Bit Checksum. Abbildung 4.3 zeigt den Aufbau noch einmal im Überblick.

4.3 Aufteilung des Flash-Speicher

Neben der Speicherung der eigentlichen Applikation wird der Flash-Speicher auch noch zusätzlich für das Speichern von benötigten Parametern und das Ablegen von Debug-Informationen genutzt.

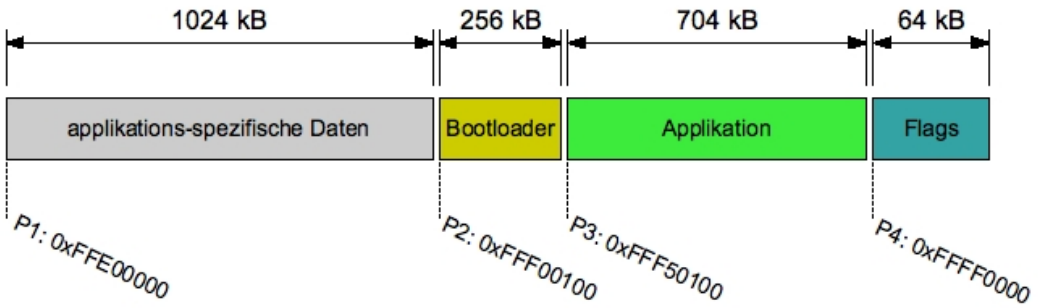

Abbildung 4.4: Aufteilung des Flash-Speicher

Abbildung 4.4 zeigt im Überblick, wie der 2MB Flash-Baustein aufgeteilt und genutzt wird. Eine Besonderheit ist der Einsprung-Punkt für den Prozessor, wenn die ECU gestartet wird. Er befindet sich nicht am Anfang des Flash-Speicher, sondern in der Mitte an Punkt „P2“ in der Grafik. D.h. an dieser Position muss sich der Startcode befinden. In diesem Fall befindet sich an dieser Position die Bootloader-Applikation für welche ein maximaler Platz von 256 kB vorgesehen ist, was genau 4 Sektoren entspricht.

Der Bereich vom Anfang bis zum Einsprung-Punkt in der Mitte des Speichers wird für applikations-spezifische Daten genutzt. Dies können Parameterwerte sein, Grafiken oder sonstige Daten, die von der jeweiligen Applikation benutzt werden. Die letzten 4 Sektoren mit einer Gesamtgröße von 64 kB sind ebenfalls für applikations-spezifische Daten vorgesehen wie CAN-Parameter oder den Bootloader-Informationen.

Somit bleibt für die eigentliche Applikation maximal ein Platz von 704 kB. Dieser Platz kann nur vergrößert werden, indem die Bootloader-Applikation weiter verkleinert wird oder der 1024 kB große Platz vor dem Bootloader genutzt wird.

4.4 Entwicklung des Übertragungsprotokolls

4.4.1 Anforderungen an das Protokoll

Die grundlegende Anforderung aus der Aufgabenstellung ist die Übertragung einer Datei von einem Sender (Windows-Programm oder eine weitere ECU) zu einer Empfänger-ECU. Dabei sollten sicherheitsrelevante Vorschriften, welche im HIS-Standard festgelegt sind, nur zum Teil beachtet werden. So wird zum soll zum Beispiel auf eine Validierung der gesendeten Daten oder auf aufwändige Verschlüsselungs- und Prüfverfahren verzichtet werden, da diese Technik vorerst nur in Testfahrzeugen zur Anwendung kommen werden. Lediglich eine einfache 8-Bit Checksum soll eine gewisse Sicherheit der Datenübertragung gewährleisten.

Nach anfänglichen Tests der Datenübertragung kamen noch weitere Anforderungen hinzu. So ist es sinnvoll, eine feste ID für jede ECU im CAN-Netzwerk zu vergeben, die eine ECU eindeutig identifiziert. Wie im Kapitel 4.2 schon beschrieben, wurde für die Identifizierung eine 8-Bit Nummer eingeführt, die sog. ECU-ID. Dabei sind nur Zahlen von 1 bis einschließlich 254 gültig. Die 0 und die 255 sind reserviert für bestimmte Zwecke wie zum Beispiel ein Broadcast oder Ähnlichem.

Mit der Einführung einer ECU-ID wird es auch möglich, dass sich an den CAN-Bus angeschlossene ECUs von selbst mit ihrer ID auf einen Broadcast hin melden. So kann kontrolliert werden, welche ECUs wirklich laufen und welche nicht.

Weiterhin ist es sinnvoll, Erstellungsdatum und Größe einer Applikation mit zu übertragen. Die Informationen werden auf der Empfänger-ECU nach erfolgreichem Flashen ebenfalls in den Flash-Speicher geschrieben. So wird es möglich, zusätzlich zur Überprüfung der Checksum die Größe der Applikation zu verifizieren und festzustellen, ob die zu übertragende Applikation wirklich neuer ist als die schon vorhandene. Auf diese Weise wird ein unnötiges Übertragen von Daten vermieden.

Um die neue Applikation nach einem Flash-Vorgang dann auch zu laden, ist es nötig, die ECU einem Neustart zu unterziehen. Daher wird zusätzlich noch eine Möglichkeit benötigt, einen Reset durchzuführen, ohne dafür am Gerät selbst einen Hardware-Reset auslösen zu müssen.

4.4.2 Allgemeines zu den Paketen

Um die Anforderungen umsetzen zu können, werden zur eigentlichen Datenübertragung verschiedene Pakete benötigt. Dabei wird unterschieden zwischen Paketen, die auf Senderseite generiert werden und solchen, die von den ECUs erzeugt werden. Dabei unterliegen die Pakete einigen Einschränkungen, die sich aus den technischen Gegebenheiten ergeben. So stehen, wie im Kapitel 3.2 schon beschrieben, lediglich 8 Bytes an Daten pro CAN-Paket zur Verfügung. Um die Übertragungsrate nicht unnötig zu vermindern, wurde darauf geachtet, dass möglichst wenig Zusatzinformationen pro CAN-Paket übertragen werden.

Dabei entstand eine Typisierung von Paketen, die jeweils eine bestimmte Aufgabe erfüllen und einen definierten Inhalt besitzen.

Die folgenden Kapitel beschäftigen sich mit diesen Paketen und zeigen detailliert deren Aufbau und Verwendungszweck.

4.4.3 Entwurf der Sender-Pakete

Die Pakete lassen sich in sechs verschiedene Typen einteilen, welche jeweils eine bestimmte Aufgabe übernehmen. Um die Pakete unterscheiden zu können, wurde das erste der 8 zur Verfügung stehenden Bytes mit einer Typ-ID belegt. Das einzige Paket, das dieser Konvention nicht folgt, ist der Remote-Frame, der nach der CAN-Beschreibung keine Daten enthält. Dieser Frame wurde dafür angedacht, als Broadcast zu fungieren und die ECUs dazu aufzufordern, sich selbst mit ihrer ID zu melden. Auf diese Weise ist es möglich, auf einfachem Wege festzustellen, welche ECUs verfügbar bzw. in Betrieb sind (Hauptsächlich von der Win32 Applikation genutzt).

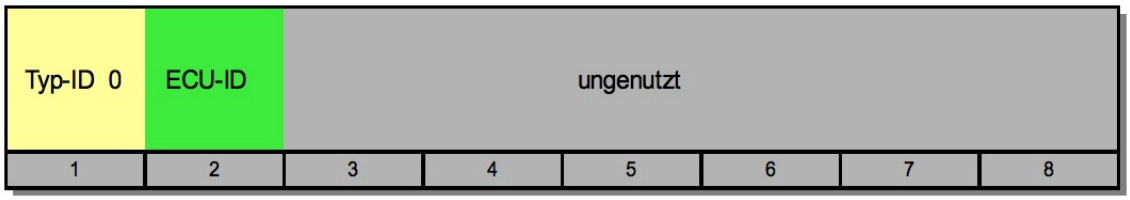

Abbildung 4.5: Header - Paket

Abbildung 4.5 zeigt den Header-Frame. Dieses Paket ist das erste, welches bei einer Datenüber-

tragung gesendet wird. Der dabei übertragene Parameter stellt die ID der zu flashenden ECU dar. Durch den Empfang dieses Pakets wird die entsprechende ECU auf die nun folgende Datenübertragung aufmerksam gemacht, worauf sie sich in einen aktiven Empfangsmodus versetzt und nun kommende Pakete auch verarbeitet. ECUs, auf die diese ID nicht zutrifft, ignorieren weiterhin folgende Pakete und warten auf ein Header-Paket mit ihrer ID.

Der nächste Paket-Typ ist für die eigentliche Nutzdatenübertragung gedacht. Wie in der Abbildung 4.6 vermerkt, werden pro Datenpaket 7 Bytes an Nutzdaten übertragen. Auf eine Nummerierung der Pakete wurde zugunsten einer schnelleren Datenrate verzichtet.

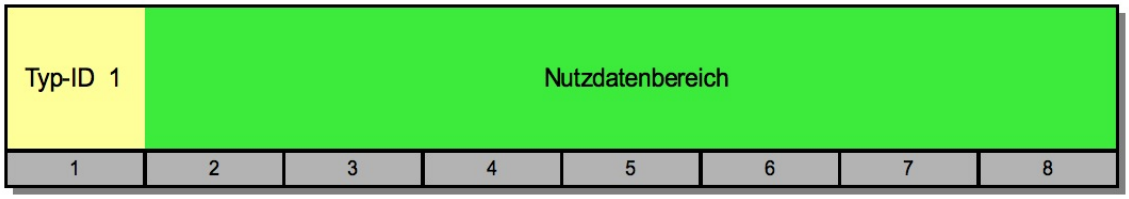

Abbildung 4.6: Nutzdaten - Paket

Das Paket mit der Typ-ID 3 stellt das Schluss-Paket der Datenübertragung dar. Wenn dieses Paket bei der Empfänger-ECU eintrifft, leitet diese die Verarbeitung der empfangenen Daten ein.

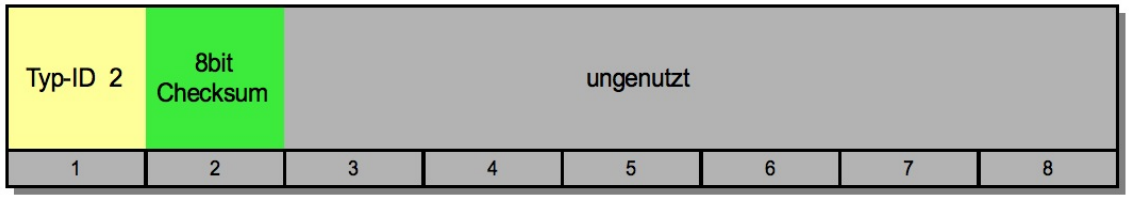

Abbildung 4.7: Schluss - Paket

Zusätzlich wird, wie in Abbildung 4.7 zu sehen, die auf Sender-Seite generierte 8-Bit Checksum mit übertragen, welche auf Empfängerseite noch einmal berechnet und dann verglichen wird.

Die folgenden Pakete dienen, wie der Remote-Frame, nur der Datenaquirierung und dem Auslösen von Aktionen auf der ECU.

Zum einen das in Abbildung 4.8 beschriebene Paket, um Informationen über eine etwaige installierte Applikation abzurufen.

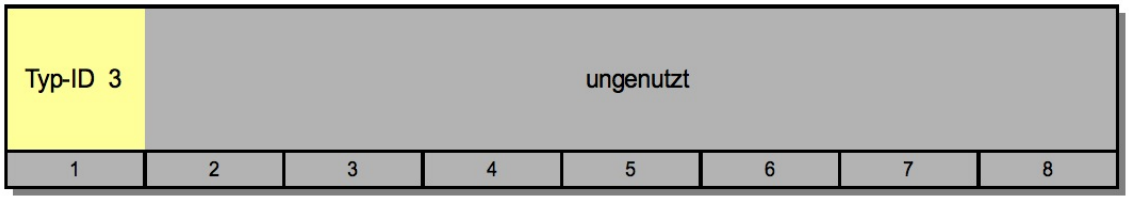

Abbildung 4.8: Paket zum Abrufen der Applikationsinformationen

Der Typ dieses Pakets wird mit einer „3“ gekennzeichnet und besitzt keine Parameter. Deshalb ist es nur möglich, von einer durch ein Header-Paket aktivierten ECU direkt Informationen abzurufen. Als Antwort bekommt man dann ein Paket mit den benötigten Informationen wie den UNIX-Zeitstempel und das Erstellungsdatum der zuletzt geflashten Applikation.

Zum anderen gibt es noch ein Reset-Paket, gekennzeichnet durch die ID 4 (Abbildung 4.9).

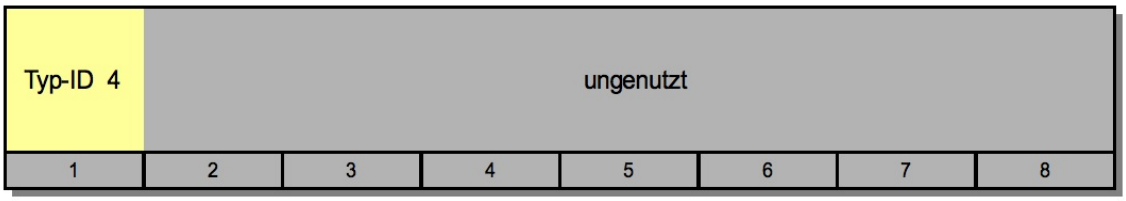

Abbildung 4.9: Paket zum Auslösen eines Software-Resets auf der gerade aktiven ECU

Dieses Paket dient der Auslösung eines Software-Resets. Wichtig bei diesem Paket ist es, dass ein Reset nicht zufällig ausgelöst werden kann. Daher wird es, wie auch beim vorherigen Paket, nur von einer aktiv geschalteten ECU verarbeitet

4.4.4 Entwurf der Antwort-Pakete

Anders als die Pakete auf der Sender-Seite besitzen die Antwort-Pakete keine Kennung in Form eines Paket-Typs. Stattdessen wird im ersten Byte immer die ID der ECU eingetragen, welche die Daten erhalten und verarbeitet hat. Da es hier bis dato nur zwei verschiedene Paketarten

gibt, kann auf eine weitere Typisierung verzichtet werden.

Das erste Antwort-Paket enthält neben der ECU-ID noch die Informationen über die im Flash-Speicher abgelegten Applikation. Abbildung 4.10 zeigt den genauen Aufbau dieses Pakets.

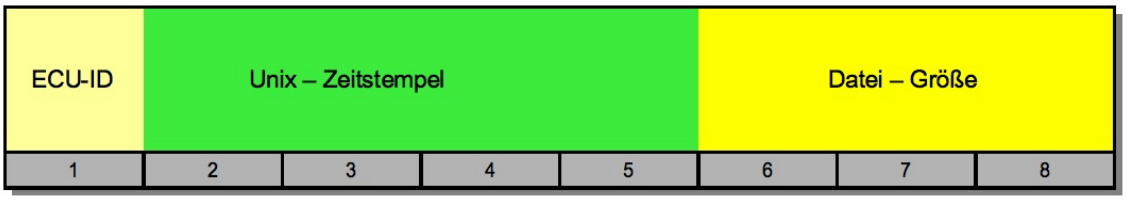

Abbildung 4.10: Antwort-Paket mit Informationen über die Applikation, die auf der ECU mit der angegebenen ID angelegt ist. Falls keine Applikation installiert wurde, sind diese Felder mit 0 belegt

Dieser Paket-Typ wird sowohl als Antwort auf einen Broadcast als auch auf ein Header-Paket verwendet. So werden automatisch die auf Sender-Seite benötigten Informationen gesendet und müssen nicht noch zusätzlich angefordert werden.

Das zweite Antwort-Paket ist in Abbildung 4.11 dargestellt.

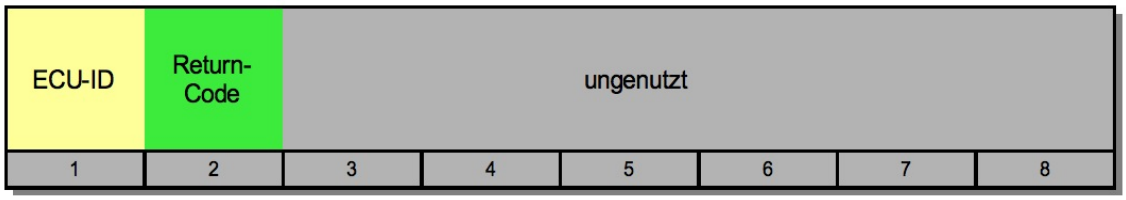

Abbildung 4.11: Antwort-Paket mit Return - Code vom Flashvorgang auf der Target-ECU

Dieses Paket enthält den Rückgabewert, der nach dem Flashvorgang auf der Empfänger-ECU vorliegt. Im Idealfall sollte dieser Wert 0 sein, was den Erfolg des Flashvorgangs symbolisiert. Ist dieser Wert größer 0, liegt ein Fehler vor.

Die nachfolgende Liste zeigt Fehler, die während des Fashvorgangs auftreten können:

0 Kein Fehler ist aufgetreten und der Flashvorgang war erfolgreich

1 Hier nicht von Relevanz

2 Hier nicht von Relevanz

3 Die angegebene Startadresse befindet sich nicht im Adressbereich des Flash-Speichers

4 Die zu flashende Applikation ist zu groß und würde über das Ende des Flash-Speichers hinausreichen

5 Der Flash-Treiber wurde nicht geöffnet

6 Ein Fehler während des Lesens vom Flash-Speicher ist aufgetreten

7 Ein Fehler während des Schreibens in den Flash-Speicher ist aufgetreten

8 Ein Fehler während des Löschens eines oder mehrerer Sektoren im Flash-Speicher ist aufgetreten

9 Es sind keine Informationen über eine Applikation verfügbar

10 Die gesendete Prüfsumme stimmt nicht mit der lokal berechneten überein

11 Die gesendete Dateigröße stimmt nicht mit der tatsächlichen Größe überein

Die Fehlercodes finden sich auch im Quellcode des Flash-Task wieder und können dort nachgeschlagen werden.

4.4.5 Der Flashvorgang im Detail

Der Flashvorgang lässt sich in drei Teilbereiche gliedern. Im ersten Teil wird die ECU, die geflasht werden soll, durch ein Headerpaket aktiviert. Dies passiert, indem die ECU die ID, welche im Header-Paket mitgeschickt wird, mit der eigenen vergleicht. Der Sinn darin besteht in der Minderung von unnötiger Arbeit in den unbeteiligten ECUs. Sie bleiben solange in einem passiven Zustand, bis sie selbst durch ein Headerpaket mit der richtigen ID aktiviert werden. Bis dahin ist die Datenverarbeitung auf ein Minimum reduziert. In der zweiten Phase werden dann die eigentlichen Nutzdaten übertragen. Dabei wird das im Vorfeld in kleine Pakete zerlegte WUF-File Stück für Stück gesendet. Wurden alle Pakete erfolgreich übertragen, folgt das Schluss-Paket, welches die auf Senderseite berechnete Checksum enthält. Dieses letzte Paket signalisiert dem Empfänger das Ende der Übertragung, woraufhin dort die Datenverarbeitung gestartet wird.

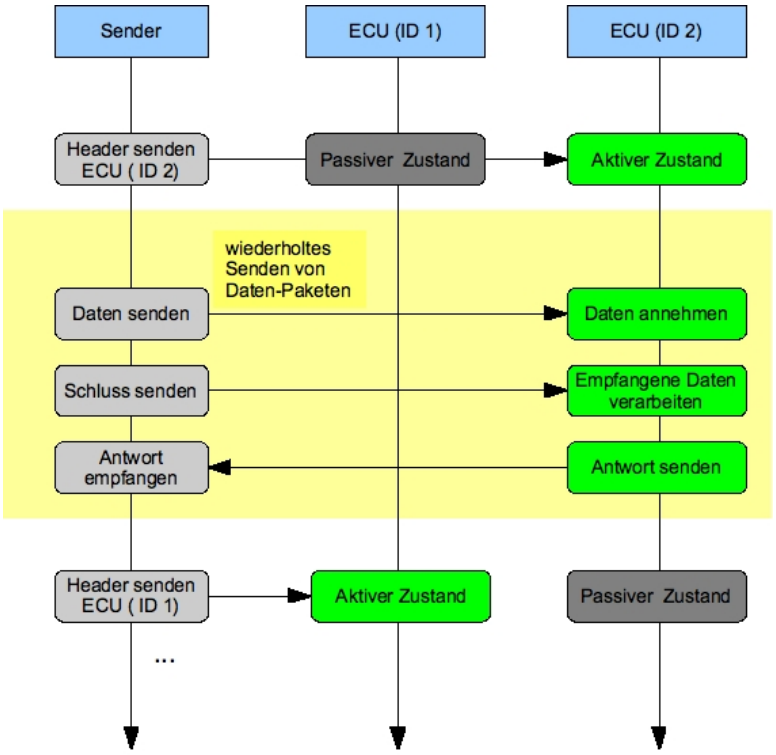

Abbildung 4.12: In dieser Grafik ist der prinzipielle Ablauf eines Flashvorgangs an Hand eines Senders und 2 Empfänger-ECUs dargestellt. Dabei ist sehr gut erkennbar, dass nur die aktive ECU Daten empfangen und verarbeiten kann

Nach der Übertragung aller Pakete wartet der Sender auf eine Antwort des Empfängers. Ab-

bildung 4.12 zeigt den Ablauf noch einmal an Hand eines Beispiels mit zwei ECUs und einem Sender.

Neben dem einfachen Ablauf gibt es noch eine erweiterte Ablaufvariante. Hierbei wird nach dem erfolgreichen Laden und Verarbeiten eines Binaries ein Reset-Paket gesendet, worauf die Empfänger-ECU neu gestartet und die neu installierte Applikation geladen wird. Die Trennung dieses Pakets vom normalen Ablauf wurde deshalb gemacht, um ein versehentlich falsch hochgeladenes Binary noch korrigieren zu können.

Ein Zusatz, der die Handhabung des Windows-Tools vereinfachen soll, ist der Einsatz des Remote-Frames. Mit ihm lassen sich die verfügbaren ECUs leichter handhaben. Abbildung 4.13 verdeutlicht den Einsatz des Remote-Frames noch einmal in grafischer Form.

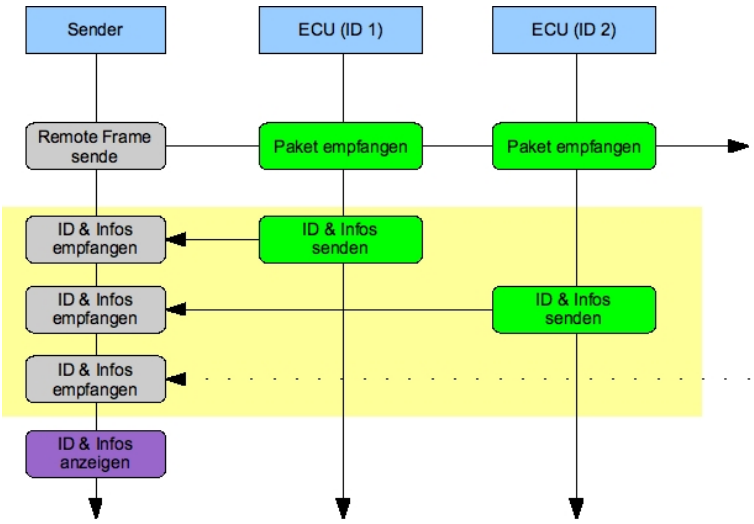

Abbildung 4.13: Der Einsatz des Remote-Frame zum Sammeln von Informationen über verfügbare ECUs im CAN-Netzwerk

4.5 Flashen per WinCan

Die Aufgabenstellung für diese Arbeit sieht ein Windows-Tool vor, über welches der Flashvorgang abgewickelt werden kann. Dabei unbeachtet blieb jedoch der genaue Ablauf des Flashvorgang, vor allem im Hinblick auf die zusätzliche Möglichkeit, den Flashvorgang über die Webserver-ECU durchführen zu können. Bei dem Entwurf des WUF-Formats, der das Flashen über den Webserver vereinfachen soll, musste eine Möglichkeit geschaffen werden, ein WUF-File zu erzeugen und abzuspeichern. Dies kann in erster Linie im WinCan erledigt werden. Es macht den ganzen Vorgang aber wieder umständlicher und birgt weitere Fehlerquellen, da nach jedem Übersetzungsvorgang das WUF-File per Hand neu erzeugt werden muss. Vergisst man dies, könnte aus Versehen eine nicht mehr aktuelle Version für den Flashvorgang verwenden.

Nach einigen Recherchen stellte sich heraus, dass die benutzte Entwicklungsumgebung *CodeWarrior* eine Funktionalität bereitstellt, sowohl vor als auch nach einem Übersetzungsvorgang automatisch ein Batch-File ausführen zu lassen. Dabei steht auch die Option von Parametern seitens der IDE zur Verfügung.

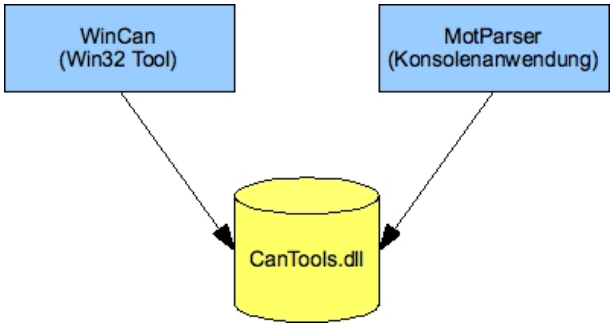

Abbildung 4.14: Aufteilung der beiden Applikationen und die Auslagerung der Funktionalitäten in eine DLL

Daher liegt es nahe, neben dem normalen Windows-Tool zum Erzeugen und Senden von WUF-Files auch ein Kommandozeilen-Tool zu entwerfen, welches sich der Funktionalitäten des Windows-Tools bedient und so in der Lage ist, im Zuge einer Batch-Verarbeitung automatisch nach jedem Übersetzungsvorgang ein WUF-File zu erzeugen. Dadurch ist gewährleistet, dass immer eine aktuelle Version vorliegt, die jetzt auf einfachem Wege geflasht werden kann.

Der Entwurf der windows-seitigen Software stützt sich nunmehr auf drei Teile. Zum einen die

WinCan Applikation mit ihren angepassten Oberflächen-Elementen, zum anderen die Konsolenanwendung *MotParser* und zuletzt eine DLL, die von beiden Applikationen verwendet wird und alle benötigten, funktionalen Bestandteile enthält.

4.5.1 Kapselung der Funktionalitäten

Die Trennung von Funktionalität und Anzeige ist grundlegender Bestandteil eines jeden Software-Entwurf. Mit der Klasse *CFlashController* wurde eine Schnittstelle geschaffen, die einer Oberflächen-Applikation alle benötigten Werkzeuge in die Hand gibt, um einen Flashvorgang durchzuführen.

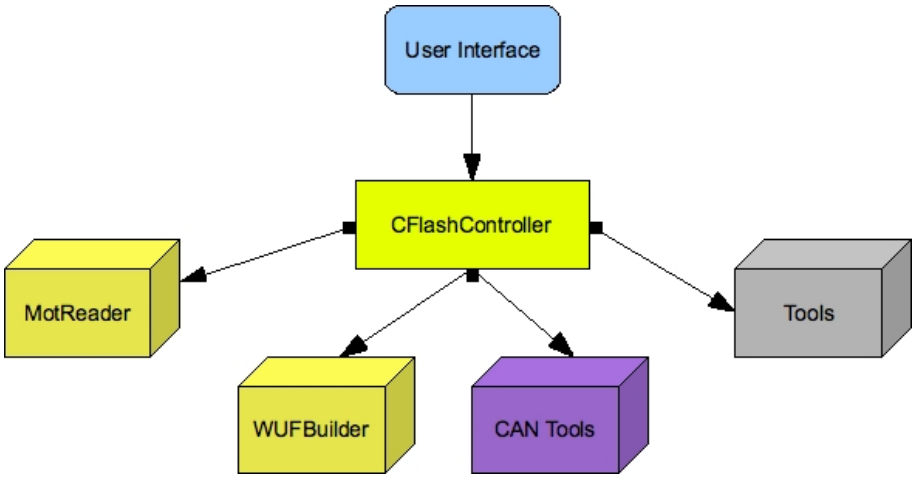

Abbildung 4.15: Gliederung der einzelnen Applikationsteile

Bei den einzelnen Funktionalitäten bietet es sich an, wie in Abb. 4.15 gezeigt, Expertenklassen einzuführen, welche spezielle Anforderungen kapseln und eine Wiederverwendbarkeit an anderer Stelle garantieren. So wurden Funktionalitäten, die den CAN-Bus betreffen, ebenso ausgegliedert, wie das Auslesen von MOT-Files und das Erzeugen von WUF-Files.

4.5.2 Vorbereitung der Sendedaten

Neben einer schnellen und sicheren Datenübertragung gehört auch eine gute Vorbereitung der zu sendenden Daten zu den Kernpunkten dieser Arbeit. Nach mehreren Vorentwürfen, die sich mit

verschiedenen Wegen der Datenvorbereitung beschäftigten, hat sich der im Folgenden vorgestellte Weg als der beste erwiesen.

Die Generierung der WUF-Datei erfolgt dabei in mehreren Schritten, die jeweils bestimmte Aspekte schon vorhandener Techniken wiederspiegeln. So werden statt der vom Compiler erzeugten Binärdatei MOT-Dateien verwendet, welche auch von dem vorhandenen FlashProgrammer - Tool verarbeitet werden. Zusätzlich werden die aus der MOT-Datei ausgelesenen Daten in eine speziell vorbereitete Datenstruktur kopiert.

Einlesen der Rohdaten

Wie in der Einleitung schon beschrieben wurde, werden die Rohdaten des Binaries aus den von der IDE erzeugten MOT-Files gewonnen. Zu diesem Zweck wurden Klassen erstellt, welche diese Verarbeitung übernehmen und die Struktur eines MOT-Files nachbilden.

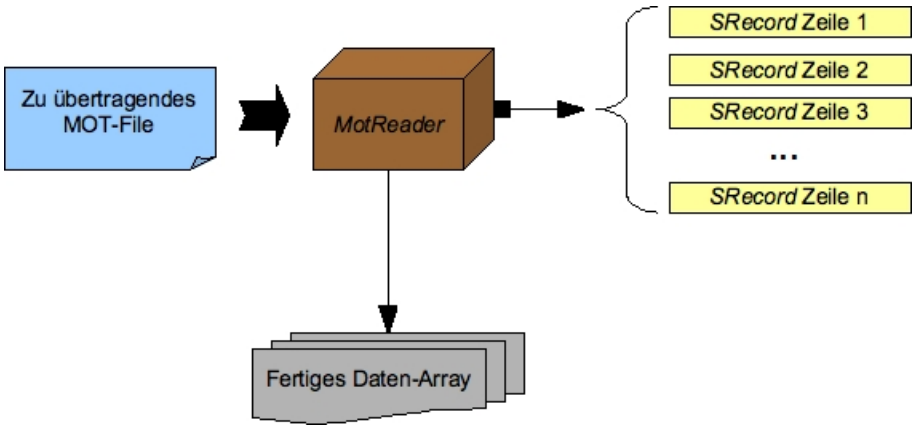

Abbildung 4.16: Vorgang der Datenerzeugung aus einem vorliegenden MOT-File mit Hilfe der MotReader-Klasse und der SRecord-Klasse

Die Klasse *SRecord* verkörpert eine Zeile eines MOT-Files. Sie extrahiert die dort beschriebenen Daten wie Startadresse und die Nutzdaten, die dann wieder weiterverarbeitet werden können. Um die einzelnen SRecords wiederum zu verwalten und ein komplettes Binary zu erzeugen, wurde die Klasse *MotReader* angelegt. Sie übernimmt das Erzeugen der SRecords und das Zusammensetzen der dadurch gewonnenen Nutzdaten. Abbildung 4.16 veranschaulicht den Zusammenhang noch einmal grafisch.

Erzeugung des WUF-Files

Die Erzeugung von WUF-Dateien kann auf 2 Wege erfolgen. Entweder werden als Quelldateien die schon erwähnten MOT-Files verwendet. Der Vorteil dieser Methode liegt darin, dass die Startadresse, die auf der ECU benutzt werden soll, bereits im MOT-File enthalten ist und verwendet werden kann. Es können aber auch alle anderen Dateien als Quelldatei verwendet werden - in diesem Fall muss zusätzlich nur die Startadresse angegeben werden.

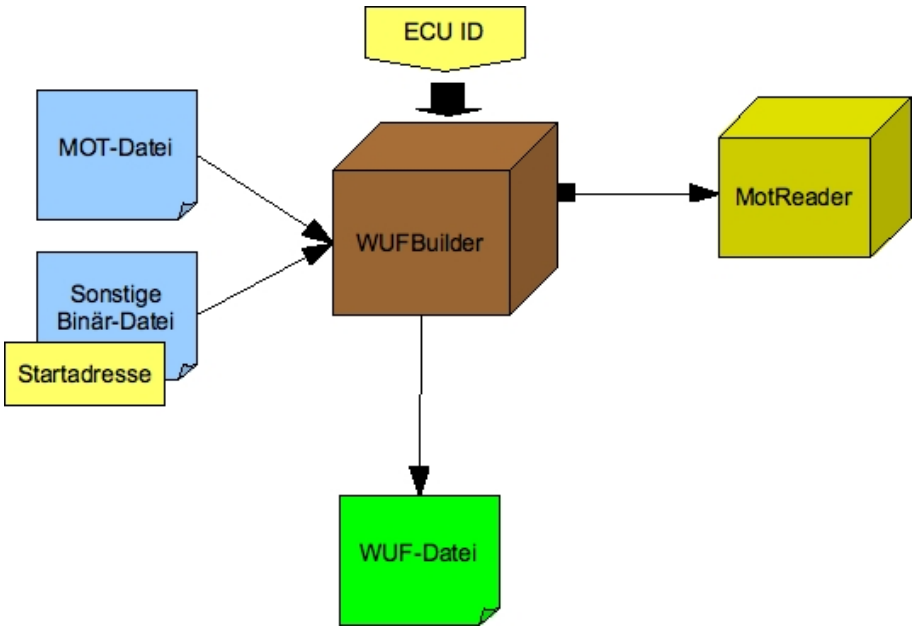

Abbildung 4.17: Darstellung der Komponenten, die bei der Erzeugung eines WUF-Files beteiligt sind und deren Abhängigkeiten

Wie in der Abbildung 4.18 zu erkennen ist, stellt Die zentrale Klasse, die alle Arbeitsabläufe kapselt, ist der *WUFBuilder*. Diese Klasse verwendet die in Kapitel 4.5.2 beschriebenen Klassen, um die Rohdaten vorzubereiten und generiert aus diesen und den Header-Daten das eigentliche WUF-File.

4.6 Entwurf des Windows User-Interface

Der wichtigste Punkt bei der Gestaltung der Oberfläche des Windows-Programms war eine einfache, intuitive Bedienung. Deshalb wurde das Programm-Interface so klein wie nur möglich gehalten und auf die Anzeige der verfügbaren ECUs beschränkt.

Die gängige Palette an Anzeigemöglichkeiten in C# ist für den angedachten Zweck der Anzeige von mehreren Informationen nicht oder nur bedingt geeignet. Aus diesem Grund wurde auf ein gängiges Pattern bei der Entwicklung neuer Oberflächenelemente unter C# zurückgegriffen und ein Extender für ein schon vorhandenes Listen-Steuerelement entworfen. Bei einem Extender handelt es sich um eine Klasse, die das originale Steuerelement um zusätzliche Funktionen erweitert und für einen einfachen Gebrauch kaspselt. In diesem Fall wird das Steuerelement *System.Windows.Forms.ListBox* durch den Extender *ListBoxTargetExtender* um eine CustomDraw Funktionalität[2] erweitert.

Die durch den Extender bereitgestellten Listenelemente informieren den Benutzer jetzt über sämtliche wissenswerten Informationen bezüglich der ECU und der dort installierten Applikation.

Diese übersichtlichen Informationen und die sehr einfach gehaltenen Funktionen auf der Oberfläche sollten die Benutzung sehr vereinfachen und mögliche Fehlerquellen im Voraus beseitigen.

[2] Statt der eigentlichen Zeichenroutine für die ListBox Elemente wird eine an die eigenen Bedürfnisse angepasste Version verwendet. Diese angepasste Version wird selbst implmentiert

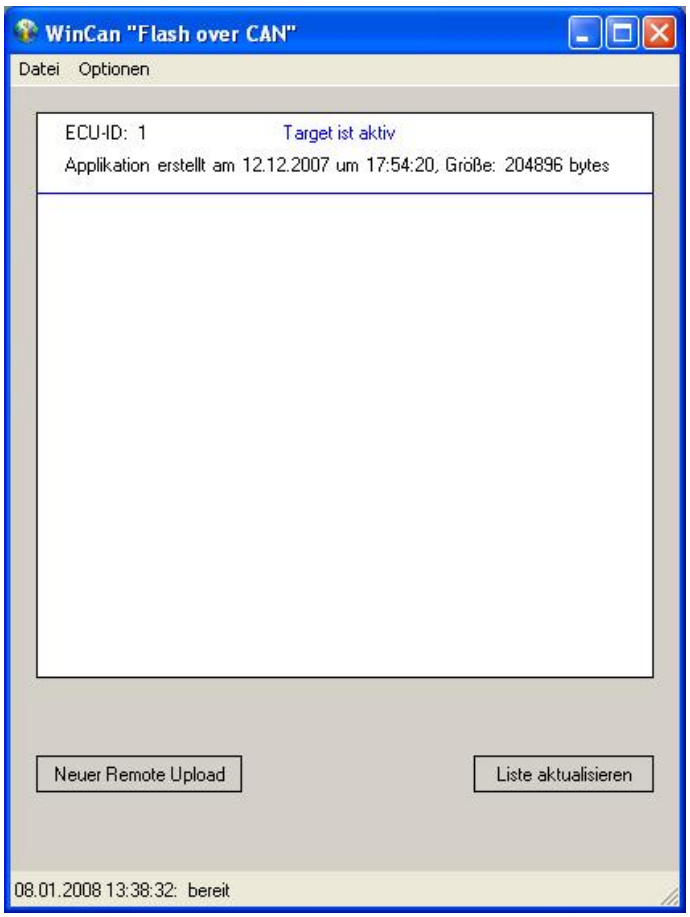

Abbildung 4.18: WinCan Benutzer-Oberfläche; Zu sehen eine im CAN-Bus vorhandene ECU mit ihren Applikationsdaten

4.7 Flashen über eine zweite ECU

In einem früheren Teilprojekt im Rahmen des AUTOSAFE-Projekts wurde eine Webserver Anwendung erstellt, welche in der Lage ist sich über ein angebundenes GPRS-Modem mit dem Internet zu verbinden. Nach dem Verbindungsaufbau legt es seine IP-Adresse auf einem zentralen Internetserver ab, wo sie für AUTOSAFE Mitarbeiter zugänglich ist.

Der Webserver wurde ursprünglich entworfen, um auf einfache Weise Fahrzeugparameter, die während einer Testfahrt anliegen, abzurufen und auszuwerten. So können zum Beispiel Kamerabilder, Radar-Sensordaten und GPS-Koordinaten abgerufen werden.

Dabei ist aber nicht nur das Abrufen von Daten möglich, auch das hochladen von Applikationen,

die für eine andere ECU bestimmt sind, wäre machbar.

4.7.1 Intention und Vorteile

Das Flashen mit Hilfe eines Win32 Tools, das über ein CAN-Interface an den CAN-Bus angebunden ist, ist nur in bestimmten Situationen sinnvoll. Zum Beispiel während der Haupt-Entwicklungsphase, während dessen die ECU noch in kein Fahrzeug verbaut wurde. Sobald dies aber der Fall ist, kann das Flashen von neuen Applikationen wieder sehr umständlich und zeitintensiv werden, da sich immer ein PC oder Laptop im direkten Umfeld befinden muss, um diesen mit dem fahrzeuginternen CAN-Bus zu verbinden.

Als praktikabler Lösungsansatz war der Einsatz des schon vorhandenen Webservers als Flashtool. An der Struktur des Servers selbst waren keine Änderungen nötig, da bereits alle Funktionalitäten fast vollständig integriert waren, die für das Hochladen von Dateien nötig sind. Auch die Anbindung an den CAN-Bus ist bereits vollständig vorhanden, so dass nur noch die Logik für den Flashvorgang integriert werden musste.

Vorteil dieser Lösung ist es, dass das Aufspielen einer neuen Applikation auf eines der sich im Fahrzeug befindlichen ECUs jetzt sehr einfach realisiert werden kann. Es bedarf lediglich einer Internetverbindung und eines Browsers. Nachdem eine Applikation auf den Server hochgeladen wurde, kann sie auf Knopfdruck auf die Ziel-ECU geflasht werden. Vor allem während Testfahrten kann sich diese Methode sehr bezahlt machen, da die Fahrt bei einem Fehler nicht unterbrochen werden muss, sondern einfach eine neue Applikation über das Internet aufgespielt wird.

4.7.2 Grundlegender Aufbau und Ablauf

Der Aufbau der Teile der Webserver-Applikation, die für den Flashvorgang wichtig sind, lässt sich grundlegend in zwei Teile aufteilen. Zum einen der Webserver selbst, der die Verbindung mit dem Internet herstellt und die zu flashende Applikation entgegen nimmt. Zum anderen ein separater Task, der die fertig übertragene Applikation analysiert und den Flashvorgang durchführt.

Nachdem die Applikation auf den Server geladen wurde, werden die Informationen aus dem Header verarbeitet und bei Bedarf auf einer gesonderten Webseite dem Benutzer angezeigt. So können die Daten wie ECU-ID, Startadresse etc. noch einmal verifiziert werden und ggf.

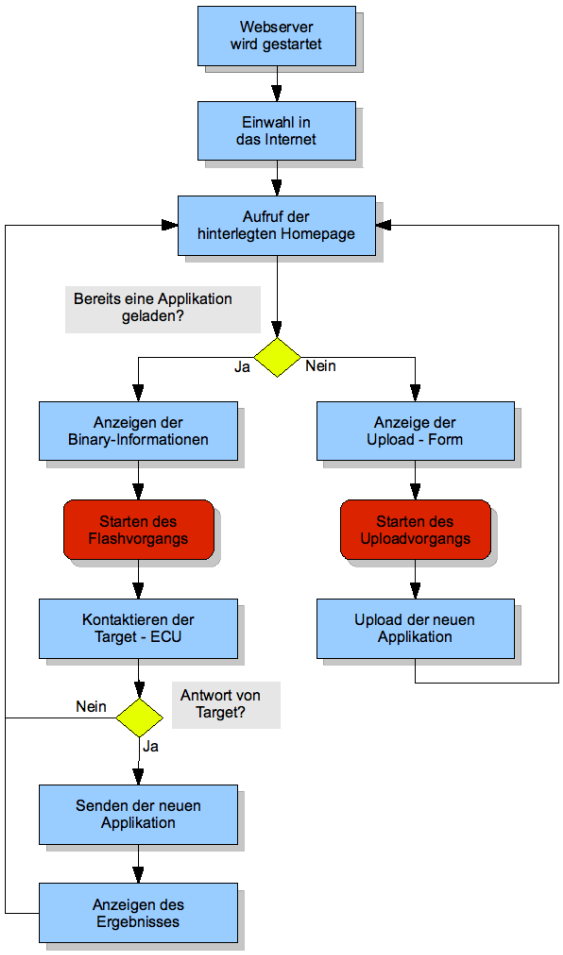

Abbildung 4.19: Web - Flashvorgang

eine neue Applikation geladen werden. Sind die ausgelesenen Informationen korrekt, kann der Flashvorgang auf der gleichen Seite veranlasst werden. Über eine ADAS-Message[3] wird der Flash-Task informiert, den Flashvorgang zu starten. Der Ablauf des Flashvorgangs selbst unterscheidet sich nicht von dem des Win32 Tools. Nach dem Senden des Header-Pakets mit der ECU-ID wird die Applikation gesendet und im Anschluss das Schluss-Paket mit Checksum. Nach Abschluss des Flashvorgangs informiert der Flash-Task den Webserver über das Ergebnis des Flashvorgangs.

Die Abbildung 4.19 zeigt noch einmal den generellen Ablauf eines Flashvorgangs über den Webserver.

[3]Teil des sog. ADAS-Frameworks, entwickelt von Siemens VOD als Erweiterung für den OSEK/OS Standard

4.8 Datenverarbeitung auf der Target-Seite

Die Verarbeitung der Daten auf der ECU lässt sich grob in zwei Teilbereiche gliedern. Zum einen die direkte Annahme der Daten per ISR[4] und zum anderen die Datenverarbeitung in einem gesonderten Task.

Die Entscheidung, diese Trennung durchzuführen, hat 2 Gründe. Zum einen gibt es auf Grund der technischen Gegebenheiten keine andere Möglichkeit, an die CAN-Daten heranzukommen als durch eine ISR. Der lokale CAN-Treiber bietet hierfür eine API[5], mit der man in der Lage ist, eigene Callback-Handler[6] zu registrieren für bestimmte CAN-IDs. Eine Vorab-Verarbeitung der eingehenden Daten wird dabei vom Treiber nicht vorgenommen. Auch der Puffer für eingehende Daten ist nicht sehr groß, was der zweite Grund für die Trennung der Verarbeitung von der Datenannahme ist. Eine um die Verarbeitung der Daten erweiterte ISR wäre zu langsam, was den Verlust von Paketen bedeuten würde.

Um zu gewährleisten, dass eine ECU nur soviel Daten verarbeitet wie nötig ist, wurde eine Technik entworfen, die eine zu flashende ECU in einen aktiven Zustand versetzt. Dabei werden Nutzdaten-Pakete und Schluss-Paket nur von einer aktiv geschalteten ECU weiterverarbeitet, während die restlichen ECUs diese Pakete ignorieren und verwerfen. Der Schlüssel, der eine ECU aktiv schaltet, besteht dabei aus einem Header-Paket, das die jeweilige ECU-ID enthält. Trifft ein solches Paket ein, wird die erweiterte Behandlung freigeschaltet und es kann eine neue Applikation auf dieses Gerät übertragen werden. Dabei werden auch Daten einer vorigen, abgebrochenen Datenübertragung zurückgesetzt und gelöscht. Somit ist sichergestellt, dass nach einem Abbruch eines Uploads sofort ein neuer Versuch gestartet werden kann.

Der einzige Paket-Typ, der normal nicht ignoriert wird, ist der Remote-Frame. Wie in Kapitel 4.4.3 schon gezeigt, dient er dazu, Informationen über die laufende Applikation einer ECU und deren eigene ID an den Sender zurückzugeben. Nur während der Datenverarbeitung durch den Flash-Task wird auch dieser Paket-Typ außer Kraft gesetzt.

Sollte ein Flashvorgang unterbrochen werden auf Grund eines aufgetretenen Fehlers, werden

[4] Interrupt Service Routine

[5] Applikation Programming Interface

[6] Funktion, die bei Auftreten eines Ereignisses durch jemand anderes aufgerufen wird, in diesem Fall der CAN-Treiber

alle nachfolgenden noch anstehenden Aktionen im Rahmen der Datenverarbeitung abgebrochen und ein Fehlercode an den Sender zurückgegeben. Sofern der Fehler behebbar ist, kann der Flashvorgang jederzeit wiederholt werden. Erst nachdem ein Reset der ECU veranlasst wird, ist die neu geflashte Applikation aktiv. Dieser Reset wurde bewusst nicht so in den Flash-Ablauf integriert, dass er automatisch ausgeführt wird. So kann der Benutzer selbst entscheiden, ob die übertragene Applikation in Ordnung ist und ein Reset des Geräts veranlasst werden kann.

5 Implementierung und Test des Gesamtsystems

In diesem Teil der Arbeit wird sich mit der Umsetzung des Entwurfs aus Kapitel 4 beschäftigt. Dabei wird vor allem auf die Besonderheiten eingegangen, die während der Implementierungs-Phase aufgetreten sind.

5.1 Klassenaufbau und -verwendung in WinCan

Wie im Kapitel 4.5 schon erläutert, gibt es einige Expertenklassen, welche sich um bestimmte Aufgaben kümmern. In den folgenden Kapitel werden deren Aufgabe näher beschrieben und anschließend die Klassen, in denen sie verwendet werden. Abgespeckte Codebeispiele sollen dabei den Aufbau bzw. deren Gebrauch aufzeigen.

5.1.1 Die Klasse SRecord

Die Klasse *SRecord* dient nur dem Zweck, eine Zeile eines MOT-Files im Speicher abzubilden. Beim Anlegen einer neuen Instanz wird dem Konstruktor eine Zeile als Parameter übergeben:

```
public SRecord(string line)
```

Dabei wird selbstständig erkannt, ob es sich um einen gültigen SRecord-Ausdruck handelt. Nach der Instanzierung können die benötigten Werte wie Startadresse oder die Rohdaten in Form eines Byte-Array über die Eigenschaften der Klasse ausgelesen und weiterverarbeitet werden.

5.1.2 Die Klasse MotReader

Die Klasse *MotReader* dient, wie in Kapitel 4.5.2 schon beschrieben, dem Auslesen von Mot-Files und dem anschließenden Erstellen eines Binary-Arrays. Für das Konvertieren der einzelnen Zeilen bedient es sich der schon beschriebenen Klasse *SRecord*.

Listing 5.1: MotReader ReaFile Funktion

```
1  /// <summary>
2  /// Reads a Motfile and creates a binary from it
3  /// </summary>
4  /// <param name="file">Filepath</param>
5  /// <returns>byte - Array of generated binary</returns>
6  public byte[] ReadFile(string file)
7  {
8    // Ergebnis-Array
9    byte[] bin;
10   // Fileinformationen abrufen
11   FileInfo fi = new FileInfo(file);
12   // Datei zum lesen öffnen
13   StreamReader sr = fi.OpenText();
14
15   // SRecords erzeugen
16   while(true)
17   {
18     // Nächste Zeile lesen
19     string line = sr.ReadLine();
20
21     // ...
22
23     // Erzeugtes SRecord speichern
24     lines.Add(SRecord(line));
25   }
26
27   // Erstellung des Array
28   // ...
29
30   // Fertiges Array zurückgeben
31   return bin;
32 }
```

Listing 5.1 zeigt den Rumpf der Funktion, die das gewünschte Array erstellt. Hierbei werden zuerst alle SRecords erstellt und anschließend deren Nutzdaten in ein mit 0xFF vorinitialisiertes Array kopiert. Dieses Array bildet den Flash-Speicher nach, nachdem er gelöscht, also komplett mit 0xFF vorinitialisiert wurde. Vorteil bei dieser Vorgehensweise ist die Möglichkeit, die Daten

nach der Übertragung auf eine ECU noch zusätzlich validieren zu können. Dabei wird ein Verfahren angewendet, welches sich aus den technischen Gegebenheiten des Flash-Speichers ergibt. Diese Gegebenheit besagt, dass der Flash-Speicher ohne vorheriges Löschen fehlerfrei beschrieben werden kann, wenn sich an den Daten nichts ändert. So kann man nach der Übertragung über den CAN-Bus die Daten erneut mit Hilfe des FlashProgrammers flashen. Wenn hier kein Fehler auftritt, wurden die Daten sauber übertragen.

Nach der Verarbeitung der Daten wird das Ergebnis-Array an die aufrufende Funktion zurückgegeben.

5.1.3 Die Klasse WUFBuilder

Die *WUFBuilder* - Klasse ist die zentrale Instanz für die Erzeugung des benötigten WUF-Files. Sie ist in der Lage, diese sowohl aus MOT-Files als auch aus anderen Binaries zu erzeugen. Listing 5.2 zeigt die API noch einmal im Überblick.

Listing 5.2: WUFBuilder API

```
1  /// <summary>
2  /// Generates a WUF File from an mot-File
3  /// </summary>
4  /// <param name="file">Filepath</param>
5  /// <returns>WUF File as a Bytearray</returns>
6  public byte[] CreateFromMot(string file, int ecu_id)
7  {
8    //...
9  }
10
11
12 /// <summary>
13 /// Generates a WUF File from an given bin, except a mot-file
14 /// </summary>
15 /// <param name="file">Filepath</param>
16 /// <param name="startadress">Startadress in Flash</param>
17 /// <returns>WUF File as a Bytearray</returns>
18 public byte[] CreateFromFile(string file, int startadress, int ecu_id)
19 {
20   //...
21 }
```

Rückgabewert ist jeweils das erzeugte WUF-File in Array Form, welches entweder sofort weiterverarbeitet kann oder als Datei abgespeichert werden kann.

5.1.4 Die Klasse *TargetItem*

Diese Klasse wurde eingeführt, um eine ECU mit all ihren Attributen in ein Objekt zu kapseln. Die Erstellung der Instanzen diesen Typs erfolgt dabei bei Eintreffen von Antwort-Paketen der ECUs auf ein gesendeten Headerframe. Das *TargetItem* findet dabei in der kompletten Anwendung Verwendung und ermöglicht so ein einfaches Erweitern um weitere Attribute im Bedarfsfall.

5.1.5 Die Klasse CFlashController

Wie im Kapitel 4.5 schon beschrieben, stellt diese Klasse die Verbindung zwischen Programm-Oberfläche und eigentlichen Produktiv-Code her. Bei der Implementierung der verschiedenen Funktionalitäten wird dabei mehrmals auf sog. Threads zurückgegriffen, was den Vorteil hat dass das Programm an sich nicht in einen Zustand verfällt, in dem es auf Benutzereingaben nicht mehr reagiert, bevor eine Operation durchgeführt worden ist. Das .NET Framework bietet eine sehr einfache Möglichkeit, diese Technik auf den eigenen Code umzusetzen, zu finden im Namespace *System.Threading* unter der Klasse *Thread*.

Das Beispiel des Read-Threads im Listing 5.3 demonstriert den Einsatz der Thread-Klasse.

Listing 5.3: Implementierung des Read-Thread

```
1
2  // Startet den Thread
3  private void StartRead()
4  {
5    if (read_status == ThreadStatus.Running)
6      return;
7
8    // Reset Stop-Flag
9    read_stop = false;
10
11   // Create new Worker-Thread
12   Thread th = new Thread(new ThreadStart(ReadCAN));
13   th.IsBackground = true;
14   th.Start();
15 }
16
17
18 // Stopt denThread
19 private void StopRead()
20 {
```

```
21    if (read_status == ThreadStatus.Stopped)
22      return;
23
24    // Set Stop-Flag
25    read_stop = true;
26
27    // Wait until the Thread has quit
28    while(read_status == ThreadStatus.Running);
29  }
30
31
32  // Eigentliche Thread-Funktion
33  private void ReadCAN()
34  {
35    read_status = ThreadStatus.Running;
36
37    while (!read_stop)
38    {
39      // thread-related Code
40    }
41
42    read_status = ThreadStatus.Stopped;
43  }
```

Wichtig an der ganzen Sache ist, dass man einer Funktion, die in einem Thread ausgeführt werden soll, keine Parameter besitzen darf. Das Konstrukt besteht aus drei Funktionen zur Steuerung und Ausführung des Threads und zwei Status-Variablen, mit welchen zum einen der aktuelle Status des Threads ermittelt werden kann (*read_status*) und zum anderen einer Bool-Variable (*read_stop*), mit der der Thread gestoppt werden kann. Letzteres ist möglich, da der Lesevorgang im Polling-Betrieb läuft, also zyklisch ausgeführt wird. Dadurch kann diese Bool-Variable bei jedem Zyklus geprüft und der Thread bei Bedarf gestoppt werden. Das Setzen dieser Variable wird dabei durch die Start- bzw. Stopp-Funktion übernommen.

Der Einsatz von Threads kommt hier nicht nur bei der Lese-Funktion zum Zuge, sondern auch für die Übertragung der Daten an die Target-ECU. So wird nicht nur eine saubere Aktualisierung des Fortschrittsbalken auf der Oberfläche garantiert, sondern auch ein einfaches Abbrechen des Flash-Vorgangs.

Eine weitere hier verwendete Technik ist der Einsatz von Delegaten. Dadurch steht der Oberflächen-Anwendung die Möglichkeit zur Verfügung, sich an diversen Events zu registrieren und so über benötigte Informationen informiert zu werden anstatt diese selbst abzufragen. Die Klasse *CFlas-*

hController bietet zu diesem Zweck eine Reihe von Events an. Listing 5.4 zeigt diese Events und deren Delegaten im Überblick;

Listing 5.4: Events der Klasse CFlashController

```
1  // Delegaten
2  public delegate void DataTransmit
3              (TargetItem item, int size, int transmitted);
4  public delegate void TransmitionEnd   (TargetItem item, int answer);
5  public delegate void TargetItemUpdate (TargetItem item);
6
7
8  public class CFlashController
9  {
10   // Klassen-Events
11   public event DataTransmit OnTransmit;
12   public event TransmitionEnd OnTransmitionEnd;
13   public event TargetItemUpdate OnTargetItemUpdate;
14
15
16   public CFlashController()
17   {}
18
19   // ...
20 }
```

Bei dem Delegaten *DataTransmit* handelt es sich um ein Event, welches jeweils nach dem Senden eines Paketes während eines Upload-Vorgangs ausgelöst wird und als Parameter neben dem *TargetItem* der ECU, die geflasht wird, auch die Gesamtgröße der zu übertragenden Daten und die Anzahl der eben übertragenen Daten in Bytes besitzt. Der Delegat *TransmitionEnd* wird ausgelöst, nachdem der Upload-Vorgang abgeschlossen wurde und übergibt neben einem *TargetItem* die Antwort der Empfänger-ECU. Das Event, das durch den Delegaten *TargetItemUpdate* repräsentiert wird, tritt immer dann auf, wenn nach dem Senden eines Header-Frames eine ECU ein Antwort-Paket mit ECU-ID und Target Informationen sendet. Beim Eintreffen des Antwort-Pakets wird ein *TargetItem* erstellt und dieses dann als Parameter mit übergeben.

5.1.6 Die Klasse *CAN_DLL*

Das CAN - Interface (siehe Kapitel 3.1.3, das bei dieser Arbeit Verwendung findet für die Datenübertragung auf Windows-Seite wird von Seiten des Herstellers mit einer Minimal-Software ausgelfiefert in Form einer C++ - DLL, mit der es möglich ist, die rudimentärsten Hardwarefunktio-

nen anzusprechen und zu nutzen. Da das Windowsprogramm *WinCan* in C# entwickelt wurde, wird an dieser Stelle eine Wrapperklasse benötigt, welche die Funktionalitäten aus der C++ - DLL in das C# Programm portiert. Für diesen Zweck wurde die Klasse *CAN_DLL* entworfen und implementiert. Sie bietet der Windows-Applikation in mehreren überladenen Methoden die Möglichkeit, Datenpakete zu senden und zu empfangen, zu sehen in Listing 5.5.

Listing 5.5: API der Klasse CAN_DLL

```
1  public class CAN_DLL
2  {
3    // Konstruktor
4    public CAN_DLL();
5
6    // Send - Funktionen
7    public int  SendMessage(int id, string data);
8    public int SendMessage(int id, byte data);
9    public int SendMessage(int id, byte[] data, int len);
10   public int SendMessage(int id, char[] data);
11
12   // SendRemote - Funktion
13   public int SendRemote(int id);
14
15   // Read - Funtkion
16   public byte[] ReadMessage(int id, int timeout);
17 }
```

Bei allen Funktionen wird zuerst der wichtigste Parameter übergeben, und zwar die CAN-ID, die beim Senden oder Empfangen benutzt werden soll. Die übrigen Parameter der Sende-Funtkionen erlauben dabei ein breites Spektrum an Datentypen, die verwendet werden können, was die API sehr flexibel gestaltet. Bei der Read-Funktion kommt zusätzlich noch ein Timeout-Parameter zum Einsatz, der es ermöglicht, eine Leseoperation nach definierter Zeit abzubrechen. Ein Wert von „0“ setzt diesen Timeout dabei außer Kraft und lässt die Funktion blockieren, bis Daten anliegen. Für die Entscheidung, eine pollende und blockierende Lese-Funktion zu implementieren hatte mehrere Gründe. Zum einen ist es nicht möglich, etwaige Callbacks, die von der originalen Hersteller-DLL zur Verfügung gestellt werden, in einem C# Programm zu verwenden. Man ist also auf ein Polling - Betrieb eingeschränkt. Das Setzen eines Timeouts dient dazu, dass die WinCan Applikation dynamischer auf bestimmte Gegebenheiten reagieren kann. So muss nach einem Upload-Vorgang länger auf eine Antwort von Seiten der Empfänger-ECU gewartet werden, da das Flashen der Daten in den Flash-Speicher einige Zeit in Anspruch nimmt. Auf der anderen Seite reicht ein sehr kurzer Timeout bei dem Read-Thread innerhalb der *CFlashController* Klasse

aus, da die Lese-Funktion sowieso zyklisch aufgerufen wird und durch einen kurzen Timeout ein schnelles Beenden des Threads gewährleistet wird.

5.2 Aufbau der Webserver - Applikation

Wie im Kapitel 4.7.2 schon beschrieben, wurde die Funktionalität im Webserver, die für den Flash-Vorgang benötigt wird, in einem eigenständigen Task („ServantFlashOverCan“) gekapselt. Dabei läuft die Kommunikation zwischen diesem Task und der Webserver-Anwendung über ADAS-Messages.

Sobald über eine spezielle Homepage ein WUF-File hochgeladen wurde, wird es per ADAS-Message an den *ServantFlashOverCan*-Task gesendet. Dort gibt es zwei Bearbeitungsstufen.

In einer ersten Stufe wird aus den Informationen, die dem WUF-Header entnommen wurden, ein String zusammengestellt, welcher an den Webserver-Task übermittelt wird. Dieser verwendet diesen String, um ihn dem Benutzer nach dem Upload auf einer gesonderten Seite über den Erfolg des Uploads zu informieren.

In einer zweiten Stufe wird dann der eigentiche Flash-Vorgang veranlasst. Dabei folgt dieser Vorgang im Wesentlichen dem der *WinCan*-Applikation. Beim Aufbau der Kommunikation über den CAN-Bus war bestand jedoch ein kleines Problem: Mit den Mitteln des zur Verfügung stehenden CAN-Treibers ist das Empfangen von CAN-Nachrichten nur über einen Callback-Handler realisierbar. Dieser läuft im Kontext des CAN-Interrupts und somit nicht in dem task, in dem der Flash-Vorgang abläuft. Aus diesem Grund musste hier über einen kleinen Umweg eine Lese-Funktion zu implementieren, die auch (wie in der WinCan-Applikation) über einen Timeout verfügt. Entstanden ist ein Konstrukt aus Callback-Handler, einer Status-Variablen und boardeigenen Mitteln zur Zeitmessung.

Das Setzen von Callback-Handlern für eine bestimmte CAN-Nachricht erlaubt die Übergabe eines void-Pointers, in welchem normal ein Pointer auf die Klasse übergeben wird, die den Callback setzt. Diesen Pointer bekommt man beim Aufruf des Callbacks wieder als Parameter übergeben. Auf diesem Weg ist es möglich, API-Funktionen dieser Klasse aufzurufen. Aus diesem Grund wurde folgende Funktion in das Klasseninterface aufgenommen:

Listing 5.6: Rumpf der Datenübergabe-Funktion für den CAN-Callback

```
1
2 T_VOID  sl_SetTargetAnswer(const T_UBYTE *answer, T_SLONG len)
```

Sie kann im CAN-Callback aufgerufen werden und schafft so eine Möglichkeit, empfangene Daten

der Klasse, welche die Daten benötigt, zugänglich zu machen. In der Klasse *ServantFlashOverCan* selbst wird zusätzlich eine Statusvariable verwaltet, mit der die Ankunft einer neuen CAN-Nachricht angezeigt wird. Diese Status-Variable kann in einer temporären Lese-Schleife in der Klasse selbst geprüft werden.

Listing 5.7: Konstruierte Lese-Schleife mit Timeout

```
1
2  T_SLONG C_ServantFlashOverCan::sl_WaitForAnswer(T_ULONG ul_WaitTime)
3  {
4     T_SLONG sl_Ret = e_FLASHOVERCAN_NO_TARGET_RESPONSE;
5     LARGE_INTEGER  li_TimeStart;
6     LARGE_INTEGER  li_TimeNow;
7     T_ULONG        ul_duration=0;
8
9     // Get Time
10    C_Util::pc_GetInstance()->sl_GetTime(li_TimeStart);
11
12    do
13    {
14       // Get act. Time
15       C_Util::pc_GetInstance()->sl_GetTime(li_TimeNow);
16
17
18       if (b_answer_arrived == true)
19       {
20          // Reset State
21          b_answer_arrived = false;
22          sl_Ret = e_NOERROR;
23          break;
24       }
25
26       // Get Miliseconds
27       C_Util::pc_GetInstance()->sl_GetDurationInMsec
28                 (li_TimeStart, li_TimeNow, ul_duration);
29    }
30    while (ul_duration < ul_WaitTime);
31
32
33    return sl_Ret;
34 }
```

Der CAN-Callback wird dabei nur bei Bedarf gesetzt und nach Erhalt der benötigten Informationen wieder gelöscht. Auf diese Weise entsteht ein Konstrukt, das eine blockierende Lese-Funktion nachbildet.

Nachdem der Flash-Vorgang abgeschlossen wurde, wird das Ergebnis ausgewertet und eine Debug-Ausgabe auf der RS232-Schnittstelle getätigt. Eine Möglichkeit, das Ergebnis auf einer Webseite darzustellen, ist zur Zeit leider noch nicht möglich, da der Flash-Vorgang einige Sekunden in Anspruch nimmt.

5.3 Implementierung der Target - Applikation

Wie im Kapitel „Entwurf“ schon gezeigt wurde, besteht die Implementierung auf der Target-Seite aus 2 Teilen. Der erste Teil, die ISR bei ankommenden CAN-Nachrichten, trifft bei Bedarf eine Vorauswahl und entscheidet, ob und wie die angekommenen Daten verarbeitet werden.

5.3.1 Aufbau und Besonderheiten der ISR

Zentraler Punkt bei der Verarbeitung der Daten auf Target-Seite ist ein struct, das es ermöglicht, eingehende Daten zu speichern und an einen Task zur weiteren Verarbeitung zu übergeben, zu sehen in Listing 5.8

Listing 5.8: Definition des Daten - Structs

```
1
2 #define  ARRAY_SIZE           0x00100000        // 1 MB
3
4 struct S_BINARY_DATA
5 {
6    T_UBYTE pub_RawData[ARRAY_SIZE];
7    T_SLONG sl_Position;
8    T_SLONG sl_checksum;
9    T_SLONG sl_ReadOnly;
10 };
```

Das Struct hat dabei nicht nur die Möglichkeit, die Rohdaten an sich zu speichern, sondern auch noch zusätzliche Werte wie einen Indexer[1] für die aktuelle Schreibposition im Array *pub_RawData* oder Platz für die zum Schluss gesendete Checksum. Zusätzlich wurde noch ein ReadOnly - Flag integriert, das ein versehentliches Überschreiben von fälschlicherweise gesendeten Daten vermeiden soll. Das setzen dieses Flags geschieht dabei nach dem Eintreffen des Schlusspakets, worauf die Daten-Verarbeitung angestoßen wird. Die Überprüfung des Flags geschieht schon zu Begin der ISR - so wird unnötiger Overhead vermieden.

Wie in Kapitel 4.8 schon beschrieben wurde, werden außer dem Remote-Frame keine weiteren Pakete verarbeitet. Erst nach einer Aktiv-Schaltung durch einen Header-Frame mit der richtigen ECU-ID wird die erweiterte Bearbeitung freigeschalten und das Flashen einer Applikation möglich.

[1] Variable, in der die aktuelle Schreib-Position eines Arrays darstellt und als Offset benutzt wird

Der erweiterte Bereich der ISR besteht dabei aus einer großen Switch-Case Anweisung, die in Verbindung mit der Bereitstellung aller vorhandenen Informationen über Paket-Typ und Paketdaten eine einfache und übersichtliche Vorverarbeitung der Daten ermöglichen. Auch ist hier ein einfaches Erweitern der Paket-Typen gewährleistet, was der Wartbarkeit sehr zu Gute kommt.

Listing 5.9: Erweiterte Datenbehandlung der ISR

```
1
2 // ...
3
4 // Frame-Typ speichern
5 sl_frametype = (T_SLONG)*ub_data++;
6
7 if (sl_frametype == e_HEADER)
8 {
9   // Stimmt die ID? Ja: ECU aktiv schalten
10 }
11
12 switch (sl_frametype)
13 {
14   case e_HEADER:     // Daten-struct zurücksetzen
15   case e_DATA:     // Daten zwischenspeichern
16   case e_TAIL:     // Checksum speichern, Message an Task
17   case e_INFO_GET:   // Targetinfos senden
18   case e_RESET:    // SW-Reset durchführen
19
20   default:       // nicht unterstützter Pakettyp!
21 }
```

5.3.2 Datenverarbeitung im Flash-Task

Nachdem das komplette Binary an die Empfänger-ECU übertragen worden ist, werden die Daten über das ADAS-Message System an den für die Verarbeitung vorgesehenen Task übergeben.

Die dafür verantwortliche Funktion besitzt folgende Signatur:

```
1
2   T_SLONG    sl_Handle_Data( S_BINARY_DATA *binary,
3                              T_SLONG *psl_AnswerLen,
4                              T_UBYTE *pub_Answer);
```

Neben der *S_BINARY_DATA* Structur werden noch 2 weitere Parameter übergeben, die für die Antwort vorgesehen sind. Dabei hat der Aufrufer der Funktion dafür zu sorgen, dass der Pointer korrekt mit genügend Speicher vorinitialisiert wurde, mindestens 8 Bytes. Diese Funktion

ist Teil einer zusätzlichen Klasse *C_FlashController*, in der die komplette Binary-Verarbeitung gekapselt wurde. Auf die Integration einer Möglichkeit, Daten über den CAN-Bus zu senden, wurde aber verzichtet. Lediglich der Zugriff auf den Flash-Speicher über einen Treiber wurde realisiert. Auf diese Weise bleibt die Klasse unabhängig von einer speziellen Kommunikationsform und kann theoretisch auch an ein anderes Bussystem angebunden werden. Die Kommunikation selbst verbleibt im laufenden Task. Er kümmert sich um das Datenhandling zwischen ISR, CAN-Bus und der *C_FlashController* Klasse.

Der Ablauf des Flashvorgangs wird in Listing 5.10 verdeutlicht.

Listing 5.10: Verarbeitung der Binary-Daten

```
1
2  // WUF - Header
3  S_BINARY_HEADER *header = (S_BINARY_HEADER*)binary->pub_RawData;
4
5  // Checksum erzzeugen und vergleichen
6  sl_Ret = sl_FileCheck(binary);
7
8  // Eigentliches Binary extrahieren
9  T_UBYTE *bin = binary->pub_RawData+sl_headersize;
10
11 // Binary in den Flash-SPeicher schreiben
12 sl_Ret = sl_Write_Flash(sl_FileStart, header->sl_filesize, bin);
13
14 // Weitere Verabreitung nur bei einem Binary
15 if (header->ub_type == 0)
16 {
17   // Dateigröße schreiben
18   sl_Ret = sl_SetFileSize(header->sl_filesize);
19
20   // Unix Zeitstempel schreiben
21   sl_Ret = sl_SetFileDate(header->sl_filetime);
22
23   // Startadresse schreiben
24   sl_Ret = sl_SetStartadress(header->sl_startadress);
25 }
26
27 // CAN-Antwort zusammenbauen
```

Dabei wird unterschieden, ob es sich um eine Applikation handelt, oder ein anderes Binary. Bei letzterem wird nur das Binary selbst an gegebene Stelle in den Speicher geschrieben, während die anderen Parameter komplett ignoriert werden.

Das im Listing verwendete Struct *S_BINARY_HEADER* ist wie folgt definiert:

```
1  struct S_BINARY_HEADER
2  {
3     T_UBYTE ub_type;
4     T_UBYTE ub_ecu_id;
5
6     T_UBYTE ub_reserved2;
7     T_UBYTE ub_reserved3;
8
9     T_SLONG sl_filesize;
10    T_SLONG sl_filetime;
11    T_SLONG sl_startadress;
12 };
```

Es beschreibt in seinem Aufbau exakt den WUF-Header. Durch einfaches Casten des *S_BINARY_DATA*-Pointers auf diese Struktur erhält man sehr einfach diesen Header. Durch das Addieren der Struct-Größe auf den Pointer erhält man dann das eigentliche Binary, dessen Größe im WUF-Header steht und mit der tatsächlichen Größe abzüglich Header und Checksum verglichen werden kann.

5.3.3 Zusatzinformationen im Flash-Speicher

Wie im vorherigen Kapitel schon erläutert wurde, werden die Zusatzinformationen einer Applikation in den Flash-Speicher abgelegt. Der Bereich, der dafür genutzt wird, befindet sich dabei im Ende des Flash-Speichers im Sektor 31. Wie sie der Tabelle im Anhang B auf Seite 78 entnehmen können, bestehen die letzten 4 Sektoren nicht aus 64 kByte wie die ersten 30 Sektoren, sondern sind kleiner. Sektor 31 ist 32 kByte groß, Sektor 32 und 33 sind 8 kByte groß und Sektor 34 16 kByte. Somit ergibt sich eine Startadresse von 0x001F8000 relativ zum Anfang des Flash-Speichers. Der Aufbau des Informationsbereichs im Flash-Speicher ist in Abbildung 5.1 zu sehen.

Die Zusatzinformationen dienen dabei nicht nur, der Windows-Applikation als Informationsquelle zu dienen, sondern werden auch vom Startup-Code der Bootloader-Applikation genutzt, da ein Lesen des Flash-Speichers auch ohne das Nutzen des Flash-Treibers möglich ist. So kann die Bootloader-Applikation an Hand des Auslesens der Startadresse feststellen, ob eine Applikation installiert ist oder nicht.

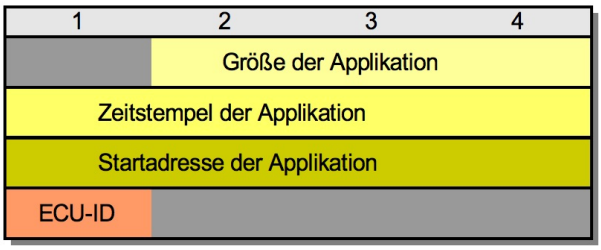

Abbildung 5.1: Aufteilung der Zusatzinformationen im Flash-Speicher

5.4 Die Bootloader-Applikation

Die Implementation, wie sie in Kapitel 5.3 beschrieben wurde, findet sich derzeit sowohl in normalen Applikationen als auch in der sog. Bootloader-Applikation. Dies ist im Prinzip eine vollwertige Applikation, jedoch reduziert auf nur den einen Task, der für das Flashen notwendig ist. Auch stehen hier nur Treiber für CAN-Bus und Flash-Speicher zur Verfügung.

Um ein zweites Binary laden zu können, werden mehrere Informationen benötigt. Die wichtigste ist die Startadresse der Applikation. Diese Information kann, wie in Kapitel 5.3.3 schon beschrieben wurde, aus dem Flash-Speicher ausgelesen werden, wobei für den Lesebetrieb kein Flash-Treiber notwendig ist. Deshalb kann das Auslesen bereits im Startup-Code auf Assembler-Ebene geschehen, wo noch keine Treiber zur Verfügung stehen.

Wenn eine gültige Startadresse gefunden wurde, wird die Applikation sofort geladen, indem seine Startfunktion angesprungen wird, zu sehen in Listing 5.11

Listing 5.11: Startup-Code zum Starten der Applikation

```
1
2 #ifdef ECU_BOOTLOADER
3 /*
4  * Read Binary-Adress of second Binary
5  * 0x0 means, no bin installed --> load bootloader
6  */
7
8 lis     r3,FLASH_STARTADRESS@ha    // Load Hi-part of address
9 addi    r3,r3,FLASH_STARTADRESS@l  // load Low-part of address
10 lis    r4,FLAGS_APP_START@ha
11 addi   r4,r4,FLAGS_APP_START@l
12
13 add    r3,r3,r4
```

```
14
15 lwz     r4,0(r3)
16
17 cmpwi   0,r4,0
18 beq     no_branch
19
20 mtlr    r4  // Load Address to Link Register
21 blr         // Branch to the Address the Link Register points to
22
23 no_branch:
24
25 #endif
```

Der Ablauf der Anweisungen ist wie folgt definiert:

* Laden des High-Byte der Flash-Startadresse nach r3
* Laden des Low-byte der Flash-Startadresse nach r3
* Laden des High-Byte der Adresse der gespeicherten Binary-Startadresse nach r4
* Laden des Low-Byte der Adresse der gespeicherten Binary-Startadresse nach r4
* Addieren der beiden Adressen zu einer absoluten Adresse in r3
* Auflösen der eigentlichen Startadresse des installierten Binaries nach r4
* Prüfen der Adresse auf 0

Die Prüfung de Startadresse entscheidet, ob eine ladbare Applikation installiert ist. Im Falle, dass die Adresse 0 ist, wurde keine Applikation gefunden und der Branch wird durch eine goto-Anweisung umgangen. Ist die Adresse ungleich 0, wird die Adresse durch den Befehl *mtlr* („move to link register“) in das Linkregister des Prozessors geladen und durch den Befehl *blr* („branch to link register“) angesprungen.

5.5 Aufgetretene Probleme

Die ursprüngliche Lösung, die ECU zu flashen, konnte im gesetzten Zeitrahmen leider nicht ganz verwirklicht werden. Zwar lässt sich der Flashvorgang problemlos durchführen, jedoch falls ein fehlerhaftes Binary geflasht wird, besteht keine Möglichkeit, die zusätzlich installierte Bootoader-Applikation zu starten, ohne vorher die Zusatzinformationen auf dem Flash-Speicher zu entfernen. Das Problem liegt dabei im Aufruf der eigentlichen Applikation aus der Bootloader-Applikation heraus. Es war angedacht, dass die Bootloader-Applikation in jedem Fall gestartet wird und nach einer kurzen Delay-Zeit die eigentliche Applikation geladen wird, indem dessen Startfunktion angesprungen wird. Während des Delays besteht dann die Möglichkeit, mit Hilfe der WinCan Anwendung das Booten zu stoppen und eine neue Applikation zu flashen. Somit kann garantiert werden, dass das Gerät in jeder Situation nach einem Reset neu geflasht werden kann.

Während der Implementierung dieser Methode traten jedoch Probleme beim Anspringen der Start-Funktion der Applikation auf - während es im Debug-Modus der IDE problemlos funktionierte, machte die Umsetzung in der Release-Variante, welche in den Flash-Speicher kommt, Probleme, indem es die Applikation nicht oder nur fehlerhaft lädt.

Leider ist es nicht möglich, eine im Flash-Speicher liegende Applikation zu debuggen, was eine Fehlerfindung sehr erschwert. Lediglich über Debug-Ausgaben, welche auf einen der lokalen COM Ports geschrieben werden, lässt sich grob feststellen, in wie weit die Applikation läuft und funktioniert.

Die Softwareteile, welche für diese Abwicklung benötigt werden, sind sowohl auf Windows- als auch auf Target-Seite bereits fertig implementiert und unter Debug-Bedingungen getestet.

6 Zusammenfassung und Ausblick

6.1 Zusammenfassung

In dieser Arbeit wurde ein System entworfen und erstellt, mit dem es möglich ist, neue Applikationen auf ein Steuergerät zu flashen. Als Übertragungsmedium wird dabei der CAN-Bus verwendet, die das Steuergerät sowieso schon als Standard-Schnittstelle zu anderen Geräten nutzt. Durchgeführt wird der Flash-Vorgang dabei durch eine spezielle Applikation, die als Bootloader der normalen Applikation vorgeschaltet wurde und immer als erste gestartet wird. Während einer kurzen Delay-Zeit von 50ms kann über ein Windows-Tool namens *WinCan* (Abb. 6.1), das über ein CAN-Interface verfügt, der Flash-Vorgang durchgeführt werden. Dies geschieht in der Regel über einen in *WinCan* integrierten automatischen Ablauf, der den Bootloader der Applikation anhält. Der Teil der Bootloader-Applikation, der das Flashen durchführt, kann auch in eine normale Applikation eingebaut werden, wodurch es möglich wird, eine Applikation auch während des Normalbetriebs auszutauschen. Dadurch spart man sich einen sonst benötigten Reset. Nach Abschluss der Entwicklung kann der als eigenständiger Task implementierter Flash-Teil wieder aus der Applikation entfernt werden.

Zusätzlich zum Windows-Tool wurde auch die Möglichkeit geschaffen, die zu flashende Applikation über eine zweite ECU auf das Ziel-Steuergerät zu bringen. Hierzu wurde diese ECU zu einem vollwertigen Webserver aufgebaut, der über ein GPRS-Modem an das Internet angebunden ist. Das Hochladen der Daten geht bei dieser Methode über einen HTTP-Upload von statten.

6.2 Ausblick

Trotz der hohen Skalierbarkeit des OSEK/OS werden Applikationen in Zukunft in ihrer Größe noch weiter wachsen. Dadurch werden die Datenmengen noch größer und die Übertragungszeiten

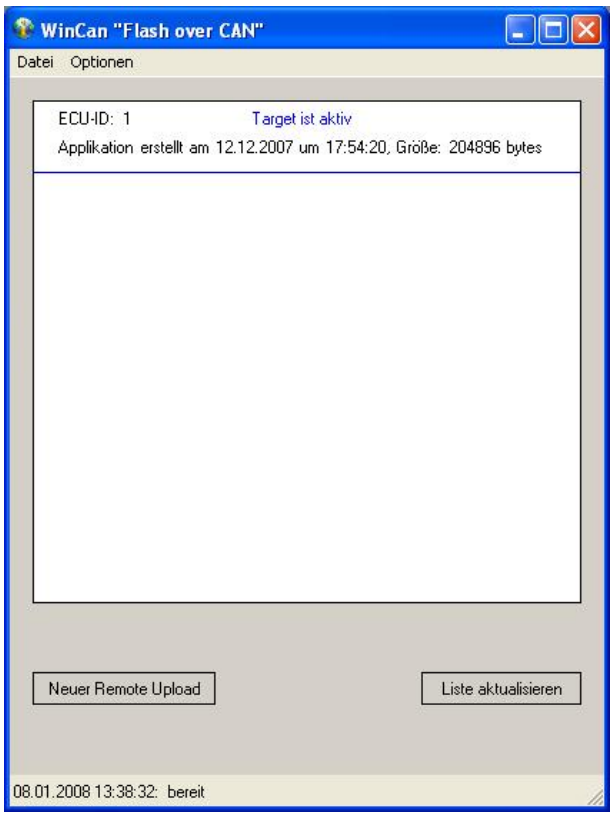

Abbildung 6.1: Screenshot des Windows-Tools *WinCan*

länger.

Um diesem Trend entgegen wirken zu können, wird es ab einer gewissen Applikations-Größe sinnvoll, sich Gedanken zu machen über eine Modularisierung. Auf diese Weise können Teile der Anwendung allein ausgetauscht werden, ohne die ganze Applikation laden zu müssen.

Das Einführen einer solchen Technik sollte jedoch gut bedacht werden, da dies einen großen Eingriff in die Struktur des OSEK OS bedeuten würde. Angelehnt an die Modul-Techniken im Linux-Kernel, müssten ebenfalls Komponenten hinzugefügt werden, die ein dynamisches Laden und Entladen von Modulen durchführen.

Wie in [11] erklärt, wurde das OSEK OS als ein statisches Betriebssystem entworfen, das über keinerlei dynamische Techniken verfügt. So muss benötigter Speicher schon im Vorfeld genau bestimmt werden. Vorteil daran ist die dadurch entstehende Berechenbarkeit der Anwendung, da während der Laufzeit keine Speicherbereich dynamisch verwaltet werden müssen.

Ein weiteres Problem ist es, dass im Zuge einer Modularisierung umfangreiche Zusatzsoftware in das Betriebssystem integriert werden muss, um ein vernünftiges Modul-Handling zu gewährleisten. Auch die Kommunikation der Module untereinander ist dabei ein Thema.

Aus diesen Gründen muss ein derartiges Vorgehen genau erruiert werden, da dies einen tiefen Eingriff in den Aufbau des Betriebssystem darstellt. Aufschluß über die nötigen Techniken könnten hierfür aus geeigneter Linux-Literatur oder den Kernel-Sourcen genommen werden.

Literaturverzeichnis

Verwendete Literatur

Die Literaturangaben in diesem Verzeichnis sind alphabetisch nach den Namen der Autoren sortiert.

[1] AG, Abatron. *Produktbeschreibung der BDI2000.*
http://www.abatron.ch/products/bdi-family/bdi1000-bdi2000.html
Letzter Besuch: 07.01.2008

[2] Automotive, Elektronik: Gehirn-Wäsche fürs STeuergerät. In: *Elektronik Automotive* (2005), S. 53–57

[3] Dalheimer, Matthias K.: *LATEX kurz & gut.* O'Reilly Verlag, 2005. – ISBN 3–89721–500–4

[4] Etschberger, Conrad: *Controller-Area-Network.* Carl Hanser Verlag München Wien, 2000. – ISBN 3–446–19431–2

[5] Max Kleiner, Berhard A.: *Patterns konkret.* Software & Support Verlag GmbH, 2003. – ISBN 3–935042–46–9

[6] Michael Niedermair, Elke und: *LATEX das Praxishandbuch.* Franzis Verlag GmbH, 2005. – ISBN 3–7723–6434–9

[7] Projekt, AUTOSAFE. *Offizielle Homepage des AUTOSAFE-Projekts.*
http://www.autosafe-online.de/
Letzter Besuch: 28.12.2007

[8] s.c., AM E. *Ausführliche Beschreibung des Motorola SRecord Formats.*

http://www.amelek.gda.pl/avr/uisp/srecord.htm
Letzer Besuch: 02.01.2008

[9] Semiconductor, Freescale. *CodeWarrior Development Tools.*
http://www.freescale.com/webapp/sps/site/homepage.jsp?nodeId=012726
Letzter Besuch: 07.01.2008

[10] Softing. *Produktbeschreibung des CANusb Interface.*
http://www.softing.com/home/de/automotive-electronics/products/hardware/bus-interfaces/can-bus-usb.php?navanchor=4010394
Letzter Besuch: 09.01.2008

[11] VDX, OSEK. *Offizielles OSEK-VDX Portal.*
http://www.osek-vdx.org/
Letzter Besuch: 19.11.2007

[12] Vereinigung, HIS. *Offizielle Homepage der Herstellerinitiative für Software (HIS).*
http://www.automotive-his.de/
Letzter Besuch: 03.01.2008

[13] Wikipedia. *Joint Test Action Group.*
http://de.wikipedia.org/wiki/Joint_Test_Action_Group
Letzter Besuch: 07.01.2008

ANHANG

Anhang A - Use-Case Ausarbeitungen

In diesem Teil des Anhangs finden sich Ausarbeitung zu den wichtigsten Use-Cases, welche im Lauf dieser Arbeit für die Windows-Applikation erarbeitet wurden.

Use Case „Programm starten“

Beschreibung

Das Programm wird gestartet und die Oberfläche wird angezeigt. Dabei wird die Liste der verfügbaren ECU's sofort aktualisiert.

Vorbedingungen

Ein CAN-Interface wurde angeschlossen

Nachbedingungen

- Das Programm ist gestartet
- Die Liste wurde erfolgreich aktualisiert

Normaler Ablauf

1. Der Use Case beginnt, wenn der Benutzer das Programm startet
2. Das System initialisiert die Hardware und baut die Oberfläche auf
3. Die Liste der verfügbaren ECU's wird aktualisiert
4. Der Use Case ist beendet

Ablauf-Varianten

Keine

Use Case „Programm beenden“

Beschreibung

Das Programm wird durch den Benutzer beendet.

Vorbedingungen

- Das Programm läuft gerade

Nachbedingungen

Keine

Normaler Ablauf

1. Der Use Case beginnt, wenn der Benutzer „Datei“ -> „Beenden“ auswählt
2. Die Software wird deinitialisiert und die Hardware abgekoppelt
3. Das Programm wird beenden
4. Der Use Case ist hier zu Ende

Ablauf-Varianten

2. a) Ein Upload-Vorgang wurde vorher durch einen Benutzer gestartet
    1. Der Upload-Vorgang wurde bereits abgeschlossen
        (1) Weiter mit Punkt 3 im normalen Ablauf

    2. Der Upload-Vorgang wurde noch nicht abgeschlossen
        (1) Der Benutzer wird gefragt, ob er den Upload-Vorgang abbrechen möchte
            1. Der Benutzer beendet den Dialog mit einem Nein
                (a) Der Upload-Vorgang wird fortgesetzt
                (b) Der Use Case ist hier zu Ende
            2. Der Benutzer beendet den Dialog mit einem Ja
                (a) Der Upoad-Vorgang wird beendet
                (b) Weiter mit Punkt 3 im normalen Ablauf

Use Case „ECU-Liste aktualisieren“

Beschreibung

Der Benutzer veranlasst einen Neu-Aufbau der Liste mit den verfügbaren ECU's

Vorbedingungen

- Das Programm ist gestartet

Nachbedingungen

Keine

Normaler Ablauf

1. Der Use Case beginnt, wenn der Benutzer den Knopf „Aktualisieren“ auf der Oberfläche betätigt
2. Bestehende Listeneinträge werden vom System entfernt
3. Das System schickt eine neue Remote Message an die ECU's
4. Die ECU's antworten mit ihrer ID und ihren Informationen
5. Das System baut an Hand der neuen Daten die Liste erneut auf
6. Der Use Case ist beendet, wenn alle verfügbaren ECU's ihre Informationen gesendet haben

Ablauf-Varianten

2. a) Die Liste ist bereits leer
    1. Das System muss keine Einträge entfernen
    2. weiter mit Punkt 3 im normalen Ablauf

3. a) Es wurde zwischenzeitlich das CAN-Interface entfernt
    1. Es können keine Nachrichten verschickt werden
    2. Das System informiert den Benutzer über den Fehler
    3. Der Use-Case endet hier

4. a) Es sind keine ECU's verfügbar zur Zeit
    1. Es werden keine neuen Listeneinträge angelegt
    2. Der Use Case endet hier

Use Case „Binary Upload starten“

Beschreibung

Der Benutzer starten den Upload eines Binaries auf eine vorher ausgewählte ECU

Vorbedingungen

- Das Programm wurde gestartet
- Die Liste der verfügbaren ECU's wurde aktualisiert
- Eine ECU wurde in der Liste durch einen Klick ausgewählt

Nachbedingungen

- Der Upload-Vorgang wurde beendet

Normaler Ablauf

1. Der Use Case beginnt, wenn der Benutzer den Upload-Vorgang durch „Rechtsklick auf einen ECU-Listeneintrag“ -> „Upload starten“
2. Der Benutzer wählt das Binary in einem Datei-Dialog aus, das er hochladen möchte
3. Der Upload-Vorgang wird gestartet, sobald der Benutzer den „OK“ Button des Datei-Dialogs ausgewählt hat
4. Der Fortschritt des Upload-Vorgangs wird in einem Fortschrittsbalken angezeigt
5. Sobald der Upload erfolgt ist, wird der Benutzer über den Erfolg der Aktion informiert
6. Der Use Case endet hier

Ablauf-Varianten

3. a) Der Benutzer betätigt den „Abbrechen“ Knopf im Datei-Auswählen Dialog
   1. Der Use Case endet hier

4. a) Der Benutzer betätigt während des Upload-Vorgangs den „Abbrechen“ Knopf
   1. Der Upload-Vorgang beendet
   2. Der Use Case endet hier

Use Case „WUF – Datei erstellen“

Beschreibung

Der Benutzer erstellt aus einer beliebigen Datei (Applikation oder sonstige Dateien) ein WUF-File zur weiteren Bearbeitung, z.B. Upload über die Webseite

Vorbedingungen

- Binary muss vorhanden sein

Nachbedingungen

- Ein gültiges WUF-File wurde erzeugt

Normaler Ablauf

1. Der Use Case beginnt, wenn der Benutzer den Menüpunkt „Datei“ -> „WUF erstellen“ auswählt
2. Der Benutzer wählt im folgenden Datei-Auswahl-Dialog das gewünschte Binary (Eine beliebige Datei)
3. Der Benutzer wählt die Schaltfläche „OK“ des Datei-Auswahl-Dialogs aus
4. Der Benutzer gibt die nötigen Zusatzinformationen an (Startadresse, ECU-ID)
5. Der Benutzer startet den Erzeugungsvorgang durch das Auswählen der „OK“ Schaltfläche innerhalb des Dialogs
6. Das System erzeugt aus den Informationen und dem selektierten Binary ein gleichnamiges WUF-File
7. Das System informiert den Benutzer über den Ausgang der Aktion
8. Der Use Case ist beendet, wenn das System mit dem Erstellen der Datei fertig ist

Ablauf-Varianten

2. a) Der Benutzer wählt die Schaltfläche „Abbrechen“ in diesem Dialog
    1. Der Use case ist an dieser Stelle beendet

4. a) Der Benutzer wählt sofort die Schaltfläche „OK“ des Dialogs
    1. Das System benachrichtigt den Benutzer über die fehlenden Informationen
        1. Der Benutzer gibt die fehlenden Informationen an
        2. Der Use Case wird in Punkt 5 im normalen Ablauf fortgesetzt

    2. Der Benutzer bricht das Erstellen durch Auswahl der Schaltfläche „Abbechen“ ab
        1. Das Erstellen des WUF-Files wird beendet
        2. Der Use Case endet hier

Anhang B - Flashbaustein Datasheet

In diesem Anhang befindet sich ein Auszug aus dem offiziellen AMD Flashbaustein Datasheet. Die Liste zeigt die Sektorverteilung und die zugehörigen Adressbereiche eines Top-Flash Bausteins, wie er in dieser Arbeit auf der ECU verwendet wird.

Table 2. Sector Address Tables (Am29LV160DT)

Sector	A19	A18	A17	A16	A15	A14	A13	A12	Sector Size (Kbytes/ Kwords)	Address Range (in hexadecimal)	
										Byte Mode (x8)	Word Mode (x16)
SA0	0	0	0	0	0	X	X	X	64/32	000000–00FFFF	00000–07FFF
SA1	0	0	0	0	1	X	X	X	64/32	010000–01FFFF	08000–0FFFF
SA2	0	0	0	1	0	X	X	X	64/32	020000–02FFFF	10000–17FFF
SA3	0	0	0	1	1	X	X	X	64/32	030000–03FFFF	18000–1FFFF
SA4	0	0	1	0	0	X	X	X	64/32	040000–04FFFF	20000–27FFF
SA5	0	0	1	0	1	X	X	X	64/32	050000–05FFFF	28000–2FFFF
SA6	0	0	1	1	0	X	X	X	64/32	060000–06FFFF	30000–37FFF
SA7	0	0	1	1	1	X	X	X	64/32	070000–07FFFF	38000–3FFFF
SA8	0	1	0	0	0	X	X	X	64/32	080000–08FFFF	40000–47FFF
SA9	0	1	0	0	1	X	X	X	64/32	090000–09FFFF	48000–4FFFF
SA10	0	1	0	1	0	X	X	X	64/32	0A0000–0AFFFF	50000–57FFF
SA11	0	1	0	1	1	X	X	X	64/32	0B0000–0BFFFF	58000–5FFFF
SA12	0	1	1	0	0	X	X	X	64/32	0C0000–0CFFFF	60000–67FFF
SA13	0	1	1	0	1	X	X	X	64/32	0D0000–0DFFFF	68000–6FFFF
SA14	0	1	1	1	0	X	X	X	64/32	0E0000–0EFFFF	70000–77FFF
SA15	0	1	1	1	1	X	X	X	64/32	0F0000–0FFFFF	78000–7FFFF
SA16	1	0	0	0	0	X	X	X	64/32	100000–10FFFF	80000–87FFF
SA17	1	0	0	0	1	X	X	X	64/32	110000–11FFFF	88000–8FFFF
SA18	1	0	0	1	0	X	X	X	64/32	120000–12FFFF	90000–97FFF
SA19	1	0	0	1	1	X	X	X	64/32	130000–13FFFF	98000–9FFFF
SA20	1	0	1	0	0	X	X	X	64/32	140000–14FFFF	A0000–A7FFF
SA21	1	0	1	0	1	X	X	X	64/32	150000–15FFFF	A8000–AFFFF
SA22	1	0	1	1	0	X	X	X	64/32	160000–16FFFF	B0000–B7FFF
SA23	1	0	1	1	1	X	X	X	64/32	170000–17FFFF	B8000–BFFFF
SA24	1	1	0	0	0	X	X	X	64/32	180000–18FFFF	C0000–C7FFF
SA25	1	1	0	0	1	X	X	X	64/32	190000–19FFFF	C8000–CFFFF
SA26	1	1	0	1	0	X	X	X	64/32	1A0000–1AFFFF	D0000–D7FFF
SA27	1	1	0	1	1	X	X	X	64/32	1B0000–1BFFFF	D8000–DFFFF
SA28	1	1	1	0	0	X	X	X	64/32	1C0000–1CFFFF	E0000–E7FFF
SA29	1	1	1	0	1	X	X	X	64/32	1D0000–1DFFFF	E8000–EFFFF
SA30	1	1	1	1	0	X	X	X	64/32	1E0000–1EFFFF	F0000–F7FFF
SA31	1	1	1	1	1	0	X	X	32/16	1F0000–1F7FFF	F8000–FBFFF
SA32	1	1	1	1	1	1	0	0	8/4	1F8000–1F9FFF	FC000–FCFFF
SA33	1	1	1	1	1	1	0	1	8/4	1FA000–1FBFFF	FD000–FDFFF
SA34	1	1	1	1	1	1	1	X	16/8	1FC000–1FFFFF	FE000–FFFFF

Note: *Address range is A19:A-1 in byte mode and A19:A0 in word mode. See "Word/Byte Configuration" section.*