IALB

Studienarbeit

Entwicklung eines Kommunikationsinterfaces für das SSI des VeCon-Controllers auf eine RS232 und CAN-Bus Schnittstelle

angefertigt am Institut für elektrische Antriebe,
Leistungselektronik und Bauelemente
der Universität Bremen

September 1999

Bearbeiter: Thomas Wedemeyer
Betreuer: Prof. Dr.-Ing. B. Orlik
Dipl.-Ing. U. Schumacher

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

In der Automatisierungstechnik werden in den letzten Jahren vermehrt sogenannte Feldbussysteme eingesetzt. Feldbussysteme sind spezielle Netzwerke mit denen die einzelnen Komponenten eines Automatisierungssystems (Prozeßrechner, Sensoren & Aktoren) untereinander vernetzt sind. Durch die Vernetzung der Komponenten ergeben sich verschiedene Vorteile beim Entwurf und der Realisierung von Automatisierungssystemen, z.B wird die Komplexität der Verkabelung der Einheiten untereinander reduziert. Der größte Vorteil, der sich durch den Einsatz von Feldbussystemen ergibt, ist, daß die Parametrisierung der einzelnen Komponenten von jedem Ort im Netzwerk aus durchgeführt werden kann. Außerdem können die Informationen aller Komponenten an einer zentralen Stelle im Netzwerk gesammelt, verarbeitet und visualisiert werden.

Ein sehr verbreitetes Feldbussystem ist der CAN-Bus. Bei dem CAN-Bus handelt es sich um einen herstelllerunabhängigen Standard, der in der ISO-Norm 11519-2 und 11898 definiert ist. Mit dem CAN-Bus steht ein universelles und sehr flexibles Feldbussystem zur Verfügung, mit dem jede Komponente eines Automatisierungssystems vernetzt werden kann. Z.B. ist bei den am Institut für elektrische Antriebe, Leistungselektronik und Bauelemente (IALB) vorhandenen Stromrichtern der Hersteller Lenze, Lust und Baumüller eine CAN-Schnittstelle vorhanden. Damit wäre es z.B. möglich ganze Motorprüfstände über den CAN-Bus zu vernetzen.

Allerdings ist es bisher nicht möglich das am Institut verwendete VeCon-Prototypenboard, daß z.B. zur feldorientierten sensorlosen Drehzahlregelung von umrichtergespeisten Drehstrom-Asynchronmaschinen eingesetzt wird, über CAN zu vernetzen. Bei dem VeCon-Chipsatz handelt es sich um einen speziell für Stromrichteranwendungen entwickeltes Prozessorsystem, das aus einem analogen und einem digitalen Chip besteht. Der digitale Chip enthält einen C165 Mikrocontrollerkern und den VeCon-DSP. Das besondere daran ist, daß der Mikrocontrollerkern und der DSP vollkommen unabhängig arbeiten und die Daten untereinander über ein Dual-Port-RAM austauschen. Für die Kommunikation mit externen Komponenten stehen unter anderem eine asynchrone RS232 und eine synchrone serielle Schnittstelle zur Verfügung.

Über die RS232-Schnittstelle wird das VeCon-Board an einen PC angeschlossen. Diese Schnittstelle wird zur Kommmunikation mit dem Monitorprogramm des C165 benutzt. Das Monitorprogramm ist ein einfaches Betriebssystem mit dessen Hilfe die Programme, die auf dem VeCon ablaufen sollen, heruntergeladen und Daten ausgelesen werden können. Der Zugriff auf den DSP-Kern erfolgt ebenfalls über das Monitorprogramm des C165-Kerns. Dadurch ist unter anderem die Geschwindigkeit mit der auf Daten des DSP-Programms zugegriffen werden kann sehr begrenzt. Ein Zugriff auf die DSP-Daten mit konstanter Zykluszeit erfordert neben dem DSP-Programm ein angepaßtes Programm, das auf dem C165-Kern abläuft. Damit ist diese Schnittstelle gar nicht oder nur eingeschränkt geeignet um Prozeßdaten direkt in das DSP-Programm einzubinden.

Um Prozeßdaten auf einfache Weise direkt in DSP-Programme einzubinden ist die SSI-Schnittstelle des VeCon-Chipsatzes besser geeignet. Der VeCon-DSP ist in der Lage direkt ohne den C165-Kern auf die SSI-Schnittstelle zu zugreifen. Läuft auf dem VeCon-DSP ein entsprechendes Programm ist es also möglich direkt Prozeßdaten mit dem DSP auszutauschen, ohne ein zusätzliches Programm auf dem C165 Prozeßdaten in das DSP-Programm einzubinden.

In dieser Studienarbeit soll ein Kommunikationsinterface entwickelt werden, das an die SSI-Schnittstelle des VeCon-Boards angeschlossen wird. Mit Hilfe dieses Interfaces soll es möglich sein Prozeßdaten über den CAN-Bus oder eine asynchrone RS232-Schnittstelle über die SSI-Schnittstelle direkt zum VeCon-DSP zu übertragen. Darüberhinaus soll es auch möglich sein, daß der DSP Prozeßdaten über diese Schnittstellen ausgibt.

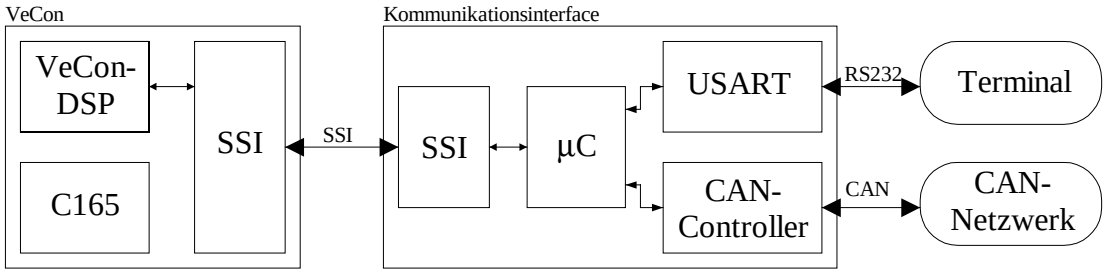


Abbildung 1: Aufbau der Kommunikationswege

Auf dem VeCon-DSP können auf Grund des begrenzten Speicherplatzes und der für die Kommunikation zur Verfügung stehenden Rechenleistung keine aufwendigen Kommunikationsroutinen realisiert werden. Aus diesem Grund ist es notwendig ein »intelligentes« Kommunikationsinterface zu entwickeln, das den VeCon-DSP bei der Kommunikation entlastet. Die Aufgaben des DSP-Programms müssen darauf begrenzt werden, Datentelegramme in einem bestimmten Format zu übertragen und die Daten in das Regelungsprogramm einzubinden. Alle weiteren Aufgaben, wie z.B. Initialisierung der Schnittstellen, muß von dem Interface selbst übernommen werden.

2 Aufgabenstellung

Im Rahmen dieser Studienarbeit soll ein geeignetes Kommunikationsprotokoll für die SSI-Schnittstelle definiert werden. Bei der Definition dieses Kommunikationsprotokolls muß die begrenzte Rechenzeit und der begrenzte Programmspeicher im VeCon-DSP berücksichtigt werden. Weiterhin sollen für die Kommunikation, z.B. für Handshake-Signale, möglichst wenige I/O-Pins benötigt werden. Außerdem muß ein geeignetes Protokoll definiert werden mit denen die Daten vom Interface über die RS232-Schnittstelle übertragen werden.

Im nächsten Schritt soll die Hardware des Kommunikationsinterfaces entwickelt werden. Im Rahmen der Hardwareentwicklung soll ein Schaltplan und ein Platinenlayout für das Interface erstellt werden. Bei dem Entwurf des Platinenlayouts soll auf einen möglichst kompakten Aufbau der Platine geachtet werden. Dadurch soll es möglich sein, das Interface nachträglich in oder an bestehende Reglerkassetten mit dem VeCon-Prototypenboard anzuschließen.

Eine weitere Teilaufgabe ist die Entwicklung der Firmware des Interfaces. Die Entwicklung dieser Software soll in Assembler erfolgen.

Um die Funktion der Interface Hard- und Software zu verifizieren soll außerdem ein Demonstrationsprogramm für den VeCon-DSP entwickelt werden. Dieses Programm soll die Kommunikation mit dem Interface über die SSI-Schnittstelle und die Verarbeitung der Prozeßdaten beim Senden und Empfang enthalten. Die Entwicklung dieser Software soll ebenfalls in Assembler erfolgen.

Um die über die RS232-Schnittstelle empfangenen Daten auf dem PC, der als Terminal dient, interpretieren zu können, soll auch ein einfaches Terminalprogramm geschrieben werden. Dieses Programm soll unter dem Betriebssystem MSDOS lauffähig sein.

3 Schnittstellen

3.1 Die synchrone serielle Schnittstelle (SSI)

Die synchrone serielle Schnittstelle (SSI bzw. SPI) wurde entwickelt um auf einfachem Weg Peripherie (A/D-Wandler, EEPROMs, Display-Treiber usw.) an Mikrocontroller anzuschließen. Für die serielle Übertragung der Daten sind nur

drei Leitungen notwendig, um eine Kommunikation zwischen zwei Teilnehmern in beide Richtungen zu ermöglichen. Der Verdrahtungsaufwand reduziert sich also erheblich gegenüber einer parallelen Übertragung.

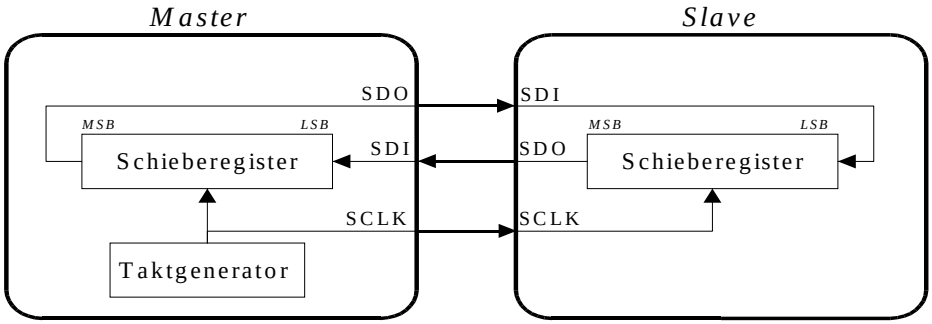


Abbildung 2: Funktionsschema der SSI-Schnittstelle

Bei den drei Leitungen handelt es sich neben den zwei Datenleitungen (SDI:=Serial Data Input und SDO:=Serial Data Output) um eine Taktleitung (SCLK). Durch dieses Taktsignal wird die Übertragung der Daten synchronisiert. Einzelheiten zur Übernahme der Daten können [Eim99, Mic1] entnommen werden. Wie man in der Abbildung 2 erkennen kann wird bei der synchronen seriellen Übertragung der Inhalt eines Schieberegisters bitweise übertragen. Bei dieser Übertragung wird gleichzeitig gesendet und empfangen. D.h., daß z.B. der Master beim Empfang eines Bytes gleichzeitig den Inhalt seines Schieberegisters sendet. In einem Schiebezyklus wird also der Inhalt der Schieberegister des Masters und des Slaves ausgetauscht.

Die SSI-Schnittstelle arbeitet mit Standard-TTL-Pegeln, d.h., eine Spannung von über +2V wird als Eins und eine Spannung unter +0,7V als Null interpretiert. Auf Grund des geringen Spannungsunterschiedes zwischen dem 1- und dem 0-Pegel ist diese Schnittstelle nicht geeignet, um längere Distanzen zu überbrücken. Der Hauptanwendungsbereich dieser Schnittstelle liegt in der Verdrahtung von einzelnen Bausteinen auf einer Platine und der Verdrahtung von Baugruppen in einem Gerät. Die Übertragungsgeschwindigkeit kann dabei bis zu mehrere Mbit/sec betragen.

3.2 Die asynchrone serielle Schnittstelle (RS232)

Im Gegensatz zur SSI-Schnittstelle ist bei der asynchronen seriellen Schnittstelle keine zusätzliche Taktleitung zur Synchronisation der Datenübertragung vorhanden. Stattdessen wird zur Synchronisation jedem übertragenen Byte ein sogenanntes Startbit vorangestellt, mit dem sich der Empfänger auf die dann folgenden Datenbits synchronisiert. Auf Grund der unterschiedlichen Synchronisation sind die SSI- und die asynchrone Schnittstelle zueinander nicht kompatibel. Weitere Einzelheiten zur asynchronen seriellen Übertragung können [Eim99] entnommen werden.

Dieses Verfahren zur Datenübertragung zwischen zwei Teilnehmern wird auch bei der RS232-Schnittstelle verwendet, die in den meisten Personal Computern standardmäßig vorhanden ist.

Im Gegensatz zur SSI-Schnittstelle erfolgt die Übertragung der Daten bei der RS232-Schnittstelle nicht mit TTL-Pegeln, sondern mit höheren Spannungsdifferenzen, um den Störabstand zwischen den Pegeln zu verbessern. Dabei wird eine Spannung unter -3V bis -15V als Eins und eine Spannung von +3V bis +15V als Null interpretiert. Durch diese Maßnahme sind Kabellängen von mehreren Metern bei der Datenübertragung kein Problem.

Neben den Spannungspegeln sind bei der RS232-Schnittstelle auch Verfahren zur Abwicklung von Handshakes und zum Aufbau von Verbindungen definiert. Durch das Signal RTS (Request To Send) kann ein Teilnehmer anzeigen, daß

er sendebereit ist. Das Gegenstück dazu ist das Signal CTS (Clear To Send) mit dem ein Teilnehmer anzeigen kann, daß er bereit ist Daten zu empfangen. Weitere Steuersignale sind DSR (Data Set Ready) und DTR (Data Terminal Ready). Mit DSR kann ein Teilnehmer anzeigen, daß er bereit ist Daten zu verarbeiten. Über das DTR-Signal wird dem Teilnehmer mitgeteilt, daß die Gegenstelle bereit ist Daten zu verarbeiten.

Außerdem sind noch zwei weitere Steuersignale definiert, die für die Kommunikation mit dem Modem verwendet werden. Das RI-Signal (Ring Indicator) zeigt an, daß das Modem angerufen wird. Mit dem DCD-Signal (Data Carrier Detect) zeigt das Modem an, daß eine Verbindung hergestellt wurde.

3.3 Der CAN-Bus

3.3.1 Grundlagen des CAN-Bus

Beim CAN-Protokoll handelt es sich um ein Bus-Protokoll, das seit Anfang der 80'er Jahre auf betreiben der Automobilindustrie entwickelt wurde. Ziel der Entwicklung war es durch die Einführung eines Bussystems im Kraftfahrzeug die Verkabelung der einzelnen Systeme, Sensoren und Aktoren zu vereinfachen und Kosten zu sparen. Für den Einsatz im KFZ-Bereich wurden deshalb die folgenden Anforderungen an das Kommunikationsprotokoll formuliert und später im CAN-Protokoll realisiert:

Der CAN-Bus muß für sogenannte Klasse-C Applikationen geeignet sein. D.h., der Bus muß in der Lage sein zeitkritische Informationen mit Zykluszeiten von 1 bis 10ms und Botschafts-Latenzzeiten von unter 1ms zu übertragen. Die Länge der einzelnen Botschaften umfaßt dabei nur wenige Byte. Die zu erwartende Datenrate bei dieser Anwendung liegt im Bereich von 250kBit/sec bis 1MBit/sec. Beispiele für typische Klasse-C Applikationen sind der Bereich des Motormanagements und der Stabilitätskontrolle.

Darüber hinaus sollte der CAN-Bus aber auch für sogenannte Klasse-A Applikationen geeignet sein. In dieser Klasse sind Applikationen zusammengefaßt, die nur geringe Datenraten benötigen und eine große Zykluszeit zwischen den Datentransfers besitzen. Die Datenrate liegt bei dieser Applikation unterhalb von 10kBit/sec. Im KFZ-Bereich ist ein Beispiel für diese Applikationsklasse die gesamte Chassis-Elektronik & Elektrik im Auto, also Zentralverriegelung, Bremsleuchten, Blinker usw.

Eine Anforderung, die sich aus Sicherheitsgründen ergibt, ist die Multimaster-Fähigkeit des CAN-Busses. Bei einem Multimaster-System ist jeder einzelne Knoten in der Lage eine Kommunikation einzuleiten. Dadurch wird vermieden, das z.B. durch einen Defekt in einem Knoten die Funktion des Gesamtsystems gefährdet ist. Für den Fall, das mehrere Knoten gleichzeitig versuchen auf dem Bus Daten zu übertragen, müssen entsprechende Vorkehrungen getroffen werden, um diese Zugriffskonflikte zu erkennen und aufzulösen. Der CAN-Bus bedient sich der Zugriffstechnik CSMA/CD+CR (Carrier Sense, Multiple Access/ Collision Detection + Collision Resolution). Einzelheiten zur Kollisionserkennung und Busarbitierung können dem Kapitel 2.3 entnommen werden.

Ein weiterer Vorteil ist, daß durch die Multimaster-Fähigkeit eine ereignisgesteuerte Kommunikation ermöglicht wird. D.h., jeder Teilnehmer ist in der Lage die Übertragung von Informationen, die durch ein Ereignis erzeugt wurden, selbstständig auszulösen. Dabei werden die Informationen mit einer sogenannten Broadcast-Kommunikation übertragen. Broadcast-Kommunikation heißt, daß die gesendete Information von allen Teilnehmern empfangen wird. Die Auswertung der empfangenen Daten in den Teilnehmern erfolgt anhand des gesendeten inhaltsbezogenen Identifiers der Nachricht. Aus diesem Grund wird jedem Ereignis ein eindeutiger Identifier zugeordnet.

Weiterhin ist beim CAN-Protokoll auch die Möglichkeit vorgesehen durch Anforderung das Versenden von Informationen auszulösen. Diese Abfrage von Informationen wird Remote Request genannt. Bei einem Remote Request wird ein spezieller Remote Frame gesendet, der keine Daten sondern nur den Identifier der gewünschten Nachricht enthält. Dieser Remote Frame wird dann von einem Teilnehmer mit einem normalen Datenframe beantwortet. Die Nachricht wird wiederum von allen Teilnehmer empfangen und die Datenkonsistenz im ganzen System ist sichergestellt.

Bei der Entwicklung des CAN-Protokolls wurde auch darauf geachtet, daß Möglichkeiten zur Erkennung von Übertragungsfehlern vorhanden sind. Dazu wurde eine Fehlererkennung in mehreren Ebenen vorgesehen. Auf Botschaftsebene ist eine Fehlererkennung mittels einer im Telegramm übertragenen 15-Bit CRC-Prüfsumme (CRC:=Cross Redundancy Check) implementiert. Darüber hinaus ist auch eine Fehlererkennung auf der physikalischen Übertragungsebene vorgesehen. Weitere Einzelheiten zur Fehlererkennung können den nachfolgenden Kapiteln entnommen werden.

Durch die oben beschriebenen Eigenschaften des Protokolls setzt sich der CAN-Bus in der Autoindustrie seit 1990 immer mehr durch. Inzwischen wird von fast allen Herstellern dieses Protokoll in ihren Fahrzeugen verwendet. Darüber hinaus setzt sich der CAN-Bus auch im Bereich der Industriesteuerungen und der Automatisierungstechnik durch. Die weitere Verbreitung des CAN-Busses auch außerhalb der Automobilindustrie führte dazu, daß der Wunsch nach dem CAN-Protokoll als »offenes« Kommunikationsprotokoll immer größer wurde. 1992 wurde dann von der CiA (CAN in Automation) damit begonnen die OSI Layer 1,2 & 7 zu definieren.

Layer		***Aufgabe***
7	*Application*	Funktionen zum Zugriff auf die Daten bereitstellen
6	*Presentation*	Konvertierung & Formatanpassung von Daten
5	*Session*	Dienste zum Auf- & Abbau von Sitzungen
4	*Transport*	Verbindung zwischen Prozessen herstellen
3	*Network*	Vermittlung der Daten zwischen verschiedenen Quellen & Zielen
2	*Data Link*	Sichere Übertragung der Telegramme sicherstellen
1	*Physical*	Definiert die Übertragung der einzelnen Bits auf dem Übertragungsmedium

Abbildung 3: Aufbau des OSI-Modells

Im OSI Layer 1 wurden detaillierte Spezifikationen für die physikalische Ebene der Kommunikation festgelegt. In diesem Layer sind z.B. Empfehlungen für Kabel, Stecker und Leistungstreiber zusammengefaßt. Der OSI Layer 2, der Data Link Layer, steuert und schützt die Kommunikation auf der Ebene eines Frames. Die OSI Layer 3 bis 6 sind für CAN nicht notwendig. Die Spezifikation auf der Ebene des Application-Layers (Layer 7) ist noch nicht abgeschlossen. Für diesen Layer sind aber bereits verschiedene Protokolle definiert worden (z.B. SDS, DeviceNET, usw.), aber es gibt keine bindende Spezifikation.

3.3.2 Aufbau des CAN-Frames

Für den inhaltlichen Aufbau eines CAN-Frames, also eines CAN-Telegramms, existieren im Augenblick zwei Spezifikationen, die sich hauptsächlich in der Anzahl der Identifierbits unterscheiden. In einem Standard CAN-Frame gemäß der CAN-Spezifikation 2.0A wird der Identifier durch 11-Bit festgelegt. Damit lassen sich 2032 verschiedene logische Adressen kodieren. Das heißt, daß in einem Netzwerk maximal 2032 verschiedene Informationen verschickt

werden können. Da dies für viele Anwendungen nicht ausreicht ist die Anzahl der Identifier in der Spezifikation 2.0B auf 29 Bit erweitert worden. Durch diese Erweiterung ist es möglich 536.870.912 verschiedene Informationen zu übertragen. Telegramme, die gemäß der Spezifikation 2.0B aufgebaut sind, werden als Extended-CAN-Frames bezeichnet. Der genaue Aufbau der Frames ist in der Abbildung 4 & Abbildung 5 dargestellt.

Dataframe CAN 2.0A (11 Bit Identifier)

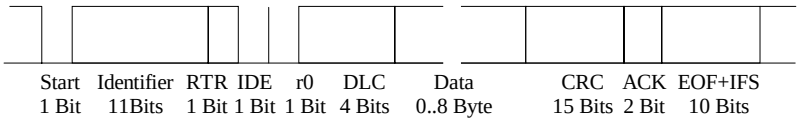


Abbildung 4: Aufbau des Standard-CAN-Telegramms

Dataframe CAN 2.0B (29 Bit Identifier)

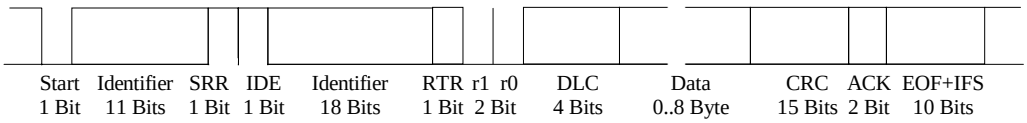


Abbildung 5: Aufbau des Extended-CAN-Telegramms

Eingeleitet wird ein CAN-Frame durch ein Startbit, das Low-Pegel führt (dominantes Bit), durch dessen fallende Flanke die einzelnen Teilnehmer synchronisiert werden. Nach dem Startbit folgen 11 Bit, die den Identifier bilden. Der Identifier kennzeichnet die logische Adresse und die Priorität der Nachricht. Je kleiner die logische Adresse der Nachricht ist, desto höher ist die Priorität der Nachricht. Bei der Übertragung des Identifiers wird das MSB zuerst übertragen.

Nun folgt bei einem Standard CAN-Frame die Übertragung des Remote Transmission Request Bits (RTR-Bit). Mit diesem Bit wird ein Remote-Frame gekennzeichnet. Dieser Frame-Typ enthält keine Daten, sondern fordert eine Nachricht von einem anderen Teilnehmer an. Die Übertragung der angeforderten Nachricht erfolgt direkt im Anschluß an die Übertragung des Remote Frames, vorausgesetzt der Bus wird nicht vorher durch eine Nachricht mit höherer Priorität belegt.

Als nächstes wird das Control Feld übertragen, das aus 6 Bits besteht. Bei einem Standard-Frame ist das erste Bit immer low. Dieses Bit wird IDE (Identifier Extension) genannt. Mit diesem Bit wird zwischen Standard und Extended-Frames unterschieden. Das Bit »r0« ist für zukünftige Erweiterungen reserviert. In den nächsten 4 Bit des Controlfeldes ist die Anzahl der Datenbytes in dem Telegramm kodiert.

In dem folgenden Datenfeld werden die Nutzdaten übertragen. Es können bis zu 8 Datenbyte pro Telegramm übertragen werden. Um Übertragungsfehler in den vorangegangenen Feldern zu erkennen, wird dann eine 15 Bit lange CRC-Prüfsumme übertragen. Zur Bestätigung, daß die übertragene Nachricht fehlerfrei empfangen wurde, stehen zwei ACK-Bits (Acknowledge) zur Verfügung. Mit diesen Bits zeigen die anderen Teilnehmer dem Sender, daß die Nachricht richtig empfangen wurde. Dazu legt der Sender einen rezessiven Pegel (1) auf den Bus und wartet darauf, daß die anderen Teilnehmer diesen Pegel mit einem dominanten Pegel (0) überschreiben. Durch dieses Acknowledge-Verfahren ist sichergestellt, daß mindestens ein Teilnehmer des Netzwerks die Nachricht richtig empfangen hat.

Beendet wird die Übertragung eines Frames durch sieben rezessive Bits, die die Bit Stuffing Kodierung verletzt (Siehe auch Kap. 2.3.3). Dieses Kennzeichen wird EOF (=End Of Frame) genannt. Nach dem EOF werden noch drei weitere

rezessive Bits (IFS=Inter Frame Space) eingefügt, mit denen sichergestellt werden soll, daß die Teilnehmer genügend Zeit haben, um die empfangene Nachricht zu verarbeiten.

Die Unterscheidung zwischen einem Standard und einem Extended CAN-Frame erfolgt anhand des IDE-Bits. Führt das IDE-Bit einen rezessiven Pegel handelt es sich bei dem übertragenen Telegramm um einen Extended Frame. In diesem Fall hat das bei einem Standard Frame vor dem IDE-Bit gesendete RTR-Bit keine Bedeutung. Deshalb wird dieser »Platzhalter« bei Extended Frames als Substitute Remote Request Bit bezeichnet. Das IDE-Bit wird gefolgt von den restlichen 18 Bit des Identifiers, dem RTR-Bit und dem 6.Bit langen Control Feld. Die Bits r1 & r0 in dem Control Feld sind für zukünftige Entwicklungen reserviert.

Wie man an dem Aufbau der CAN-Frames gemäß der Spezifikation 2.0A & 2.0B sieht, ist es für einen CAN-Controller nicht schwierig das Format der einzelnen Nachrichten-Telegramme zu unterscheiden. Dadurch ist es möglich gleichzeitig Standard & Extended CAN-Frames über ein Netzwerk zu übertragen. Voraussetzung dafür ist allerdings, daß die Controller die beiden Standards erkennen können. Es gibt viele CAN-Controller, die nur die CAN Spezifikation 2.0A aktiv unterstützen. Im Allgemeinen verhalten sich die neueren dieser Typen aber gegenüber CAN 2.0B passiv,d.h. sie ignorieren Telegramme, die nach CAN 2.0B Spezifikation übertragen werden.

3.3.3 Bit-Kodierung und Arbitrierung

Die einzelnen Bits eines Frames werden über das Netzwerk im NRZ-Verfahren (Non Return to Zero) mit Bitstuffing übertragen. Bei NRZ-Verfahren werden die Daten digital übertragen, wobei die Dauer jedes Bits gleich lang ist und die Bits direkt aufeinanderfolgen. Der High-Pegel wird dabei als logisch Null und der Low-Pegel als logisch Eins interpretiert. Durch das NRZ-Verfahren wird die zur Verfügung stehende Bandbreite optimal genutzt. Allerdings enthält dieses Verfahren keine Informationen, die zur Synchronisation bei der Übertragung benutzt werden können. Um bei längeren Datentelegrammen eine sichere Synchronisation auf die einzelnen Bits sicherzustellen, werden die Flankenwechsel im Bitstrom zur Nachsynchronisation des Bittakts benutzt.

Um sicherzustellen, daß dieses Verfahren auch funktioniert, wenn in einem Bitstrom keine Flankenwechsel auftreten, wird in den Bitstrom nach genau 5 gleichen Bits ein sogenanntes komplementäres Stuff-Bit zusätzlich eingefügt (to stuff:=hineinstopfen). Auf der Empfängerseite wird dieser Stuff-Bit automatisch vom Controller wieder aus dem Bitstrom herausgefiltert.

Der Buszugriff erfolgt beim CAN-Bus mit Hilfe einer zerstörungsfreien, bitweisen Arbitrierung. Die Voraussetzung für dieses Buszugriffsverfahren sind Bustreiber, die in der Lage sind einen rezessiven 1-Pegel und einen dominanten 0-Pegel zur Verfügung zu stellen. D.h. der rezessive 1-Pegel kann durch den dominanten 0-Pegel auf dem Bus überschrieben werden. (Siehe dazu auch Kapitel 2.3.5)

Bei diesen Arbitrierungsverfahren wird verhindert, daß zwei Teilnehmer gleichzeitig Nachrichten auf dem Bus übertragen nachdem beide Teilnehmer einen freien Bus erkannt haben, in dem der Identifier bitweise auf den Bus gelegt wird. Während sie dies tun lesen die Teilnehmer gleichzeitig den Bitstrom wieder ein und vergleichen die Bitwerte mit dem gesendeten Wert. Wird dabei festgestellt, daß ein rezessiver Pegel von einem anderen Teilnehmer mit einem dominanten Pegel überschrieben wurde stellt der Teilnehmer, dessen Bit überschrieben wurde, sofort seine Übertragung ein und wartet bis der Bus wieder frei ist. Durch dieses Verfahren ist sichergestellt, daß immer die Nachricht mit der höchsten Priorität den Zugriff auf den Bus erhält. Dieses Arbitrierungsverfahren wird zerstörungsfrei genannt, weil der Gewinner der Arbitrierung sein Telegramm nicht erneut von vorn zu senden beginnen muß.

3.3.4 Ausnahmebehandlung

Durch den Aufbau der CAN-Telegramme und das Bit-Stuffing-Verfahren sind CAN-Controller in der Lage Fehler während der Übertragung zu erkennen. Festgestellt werden können Bit-Fehler, Bit-Stuffing-Fehler, CRC-Fehler, Formatfehler im Telegramm und auch Acknowledgement-Fehler.

Wird einer dieser Fehler von einem Teilnehmer erkannt informiert er sofort die anderen Teilnehmer und den Absender des Telegramms, indem er einen Error-Frame sendet. Durch den Empfang eines Error-Frames verwerfen alle Teilnehmer die empfangene Nachricht und beginnen ebenfalls einen Error-Frame zu senden. Wenn der Bus wieder frei ist, beginnt dann der Absender der Nachricht mit einer Wiederholung der Sendung. Dadurch ist sichergestellt, daß durch Übertragungsfehler keine Informationen verloren gehen.

Ein Error-Frame besteht aus einer bewußten Verletzung der Kodierungsvorschrift. In einem Error-Frame wird einfach eine Folge von 6 oder mehr aufeinanderfolgenden dominanten Bits gesendet. Damit ist ein Error-Frame eine Verletzung der Bit-Stuffing-Regel und der Error-Frame wird als Bit-Stuffing-Fehler von den anderen Teilnehmern erkannt.

Um sicherzustellen, daß ein defekter Teilnehmer, der nicht in der Lage ist Nachrichten richtig zu empfangen, das gesamte Netzwerk blockiert, ist in jedem Controller ein spezieller Algorithmus implementiert. Mit Hilfe dieses Algorithmus zieht sich der Controller im Fehlerfall schrittweise vom Busgeschehen zurück.

Stellt der CAN-Controller fest, daß er als erster Netzteilnehmer einen Error-Frame gesendet hat, erhöht er einen internen Fehlerzähler. Solange der Wert des Fehlerzählers unter 126 liegt sendet der Controller im Fehlerfall die oben beschriebenen Error-Frames. Dieser Fehlerbehandlungsmodus wird »Active Error« genannt. Liegt der Wert des Fehlerzählers aber über 126 schaltet der Controller in den sogenannten »Error Passiv«-Modus. In diesem Modus sendet der Controller einen 6 Bit Error-Frame mit einem rezessiven Pegel. Wenn der Fehlerzähler einen Stand von 255 erreicht hat wird der Controller in den »Bus Off«-Modus geschaltet und nimmt nicht mehr an der Kommunikation auf dem Bus teil. Aus dem »Bus Off«-Modus kann der CAN-Controller nur noch durch einen Reset befreit werden. Aus dem »Error Passiv«-Modus kann sich der Controller selbst befreien, da der Fehlerzähler um eins dekrementiert wird, wenn ein anderer Teilnehmer einen Fehler zuerst erkannt hat.

Die Bearbeitung des Fehlerhandlings erfolgt vollständig im CAN-Controller. Ein an den CAN-Controller angeschlossener Mikroprozessor wird nur über Flags über den aktuellen Fehlerbehandlungsmodus informiert.

3.3.5 Die physikalische CAN-Schnittstelle

Im Prinzip kann die physikalische Schnittstelle des CAN-Busses auf unterschiedliche Weise realisiert werden. Wichtig ist dabei nur, daß es möglich ist auf dem Bus rezessive und dominante Bits darzustellen. Neben Konzepten zur Datenübertragung über optische Eindrahtverbindungen gibt es verschiedene Konzepte, die eine Übertragung der Daten mit Hilfe von Differenzsignalen vorsehen.

Im Allgemeinen wird die physikalische Schnittstelle des CAN-Busses gemäß der ISO 11898 realisiert. Bei dieser Verbindung handelt es sich um eine Zweidraht-Verbindung mit Differenzsignalen. Die maximale Übertragungsgeschwindigkeit beträgt 1MBit/sec bei einer Kabellänge von 40m, wobei maximal 30 Busknoten installiert sein dürfen. Bei geringeren Datenraten sind größere Kabellängen möglich, z.B. bei einer Übertragungsrate von 125kBit/sec beträgt die zulässige Kabellänge 500m.

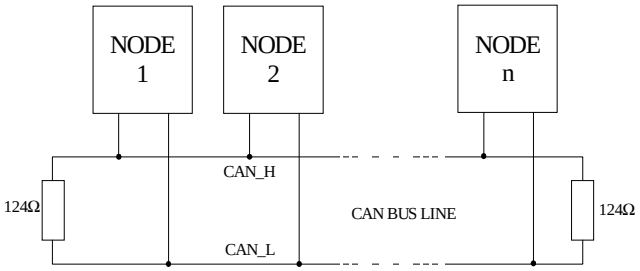


Abbildung 6: Aufbau des CAN-Netzwerks

Die Nennimpedanz des Buskabels wird mit 120Ω angegeben. An den beiden Enden der zwei CAN-Signalleitungen, die CAN_H und CAN_L genannt werden, ist ein Leitungsabschluß von jeweils 124Ω bei 1% Toleranz und 200mW Verlustleistung vorgesehen.

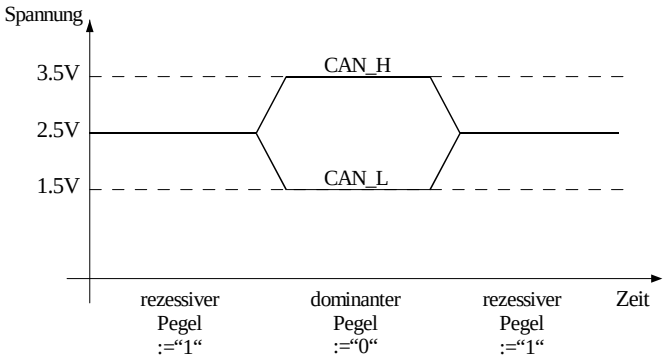


Abbildung 7: Signalpegel des CAN-Bus

Laut Spezifikation liegt auf dem Bus ein rezessiver Pegel an, wenn das Signal des CAN_H-Signals nicht höher liegt als das Signal CAN_L+0,5V. Der dominante Pegel wird durch eine Spannung an CAN_H definiert, die mindestens 0,9V höher liegt als die der CAN_L Leitung.

In dem vereinfachten Blockschaltbild in Abbildung 8 ist die interne Beschaltung eines CAN-Transreceivers dargestellt. Soll ein rezessiver Pegel (TxD:=1) auf den Bus gelegt werden sperren die beiden Transistoren im Transreceiver. Da über die Transistoren und den Terminierungswiderstand kein Strom fließt fällt auch keine Spannung über dem Widerstand ab. D.h., die Spannung, die an den beiden Eingängen des Receiver anliegt, ist gleich. Es wird also ein rezessiver Pegel über den Receiver zurückgelesen.

Durch einen 0-Pegel am TxD-Eingang wird auf dem CAN-Bus ein dominater Pegel erzeugt. In diesem Fall schalten die beiden Transistoren im Transreceiver durch. Damit fließt ein Strom über die Transistoren und den Terminierungwiderstand. An dem Terminierungswiderstand fällt dabei eine Spannung ab. Dadurch liegt am CAN_H-Eingang des Receivers eine höhere Spannung an als an dem CAN_L-Eingang. Diese Spannungsdifferenz wird von dem Receiver als dominanter Pegel zurückgelesen.

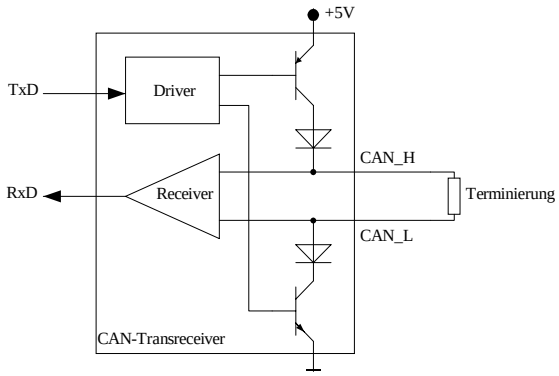


Abbildung 8: Vereinfachtes Blockschaltbild eines CAN-Transreceivers

Bisher gibt es für die Steckverbindungen zwischen einzelnen CAN-Komponenten keine Normierung im OSI 1 Layer. Allerdings gibt es eine Empfehlung, die von der Gruppe »CAN in Automation« (CiA) entwickelt wurde. Bei dem Stecker handelt es sich um eine 9-polige Sub-D Steckverbindung. Die Pinbelegung kann der Tabelle 1 entnommen werden.

PIN	***Signal***	***Beschreibung***
1	-	(reserviert)
2	CAN_L	Signal CAN_L
3	CAN_GND	Signal GND
4	-	(reserviert)
5	-	(reserviert)
6	CAN_SHLD	Signalabschirmung
7	CAN_H	Signal CAN_H
8	-	(reserviert)
9	CAN_V+	Optionales Power Supply V+

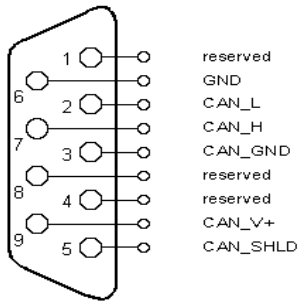


Abbildung 9: Pinbelegung des CAN-Steckers

Tabelle 1: Pinbelegung gemäß CiA Draft Standard 102 V2.0

4 Die Hardware des Interfaces

4.1 Das Konzept des Interface

Wie bereits in der Einleitung beschrieben soll durch das Kommunikationsinterface die Übertragung von Prozeßdaten direkt in den VeCon-DSP über den CAN-Bus und die asynchrone RS232-Schnittstelle ermöglicht werden. Auf Grund des begrenzten Programmspeichers und um möglichst viel Rechenzeit für die Regelalgorithmen zur Verfügung zu haben muß das Kommunikationsinterface die Daten bzw. die Datentelegramme eigenständig verarbeiten können. Der VeCon-DSP soll also nur noch den Empfang und die Sendung von Daten übernehmen, alle anderen Aufgaben sollen durch das Kommunikationsinterface übernommen werden. Dieses Ziel kann nur durch den Einsatz eines Mikrocontrollers in dem Kommunikationsinterface erreicht werden.

Wie in dem Blockschaltbild (Abbildung 10) gezeigt, wird in dem Kommunikationsinterface ein PIC-Mikrocontroller eingesetzt. Bei dem Controller handelt es sich um einen PIC16C73A von Microchip. Die Wahl fiel auf diesen Controller, weil bereits eine SSI/SPI- und asynchrone Schnittstelle (USART) auf dem Chip realisiert sind. Weiterhin

besitzt der Controller 256 Bytes RAM, die zum Zwischenspeichern der empfangenen Nachrichtentelegramme vollkommen ausreichen. Da auch der Programmspeicher in dem Chip realisiert ist, wird zum Betrieb des Mikrocontrollers kein externer Bus, an den RAM oder ROM-Bausteine angeschlossen werden, benötigt. Dadurch vereinfacht sich das Platinenlayout des Interfaces erheblich und die Platinengröße wird reduziert.

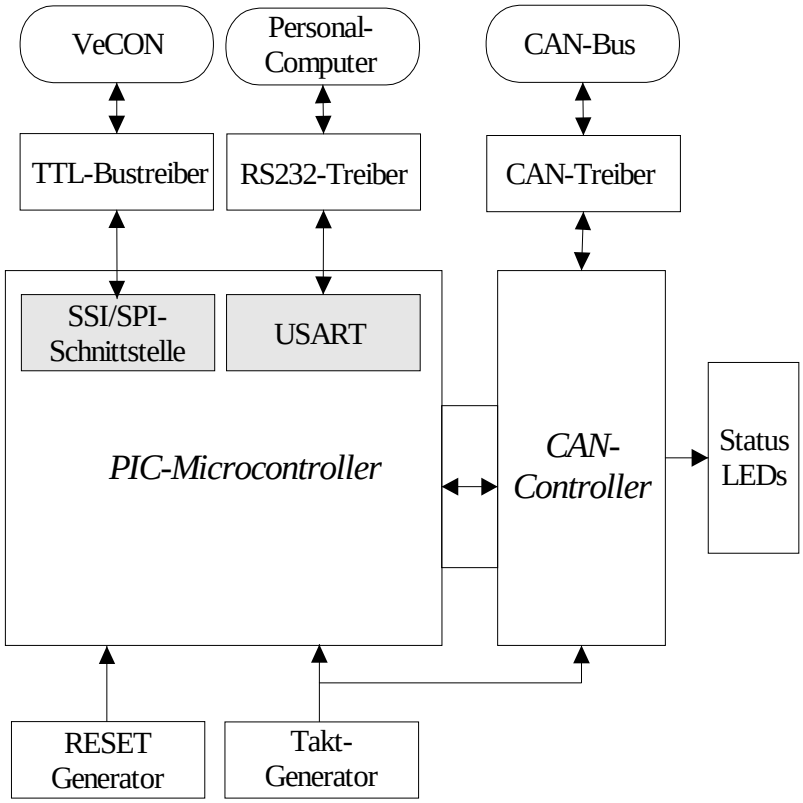


Abbildung 10: Blockschaltbild des Kommunikationsinterface

Da der PIC-Mikrocontroller keinen CAN-Controller enthält muß diese Schnittstelle mit einem externen CAN-Controller realisiert werden. In dem Interface kommt ein CAN-Controller vom Typ SAE81C90 der Firma Siemens zum Einsatz. Bei diesem CAN-Controller handelt es sich um einen sogenannten Stand-alone-Full-CAN-Controller. Diese Art CAN-Controller ist in der Lage die Kommunikation über den CAN-Bus fast vollkommen eigenständig durchzuführen. D.h. der PIC-Mikrocontroller muß nur den Datentransfer anstoßen bzw. eine empfangene CAN-Nachricht auslesen. Die Umsetzung der Daten in CAN-Telegramme und das Error-Handling übernimmt der CAN-Controller selbst. Dabei arbeitet der Controller gemäß der Spezifikation 2.0A und verhält sich gegenüber CAN-Telegrammen der Spezifikation 2.0B passiv.

Der CAN-Controller besitzt außerdem zwei 8 Bit I/O-Ports, die über den PIC angesteuert werden können. Vier dieser I/O-Leitungen werden benutzt um damit Status-LEDs anzusteuern. Es ist für jede Schnittstelle jeweils eine LED vorgesehen. D.h., bei der Datenübertragung über eine der Schnittstellen leuchtet die entsprechende LED. Die letzte LED ist eine sogenannte »Alive«-Anzeige. Diese LED zeigt durch ihr Blinken an, daß der Controller arbeitet.

Zum Betrieb der beiden Controller ist noch ein Taktgenerator notwendig, von dem z.B. der CAN-Controller die Übertragungsgeschwindigkeit auf dem CAN-Bus ableitet. Die maximale Frequenz, mit der die Controller betrieben werden dürfen, beträgt 20MHz.

Durch eine hochintegrierte Spannungsüberwachung und einen RESET-Generator der Firma Maxim vom Typ MAX811 wird sichergestellt, daß nach Anlegen der Versorgungsspannung der PIC-Controller richtig zurückgesetzt wird und das Programm zuverlässig anläuft. Liegt die Versorgungsspannung unter 4.63V wird der PIC-Controller automatisch durch das RESET-Signal gesperrt. Weiterhin besteht auch die Möglichkeit manuell durch einen Taster einen RESET auszulösen.

Die physikalische Anpassung der Signalpegel erfolgt über entsprechende Treiberschaltungen. Einzelheiten über die Schaltungsteile können den nachfolgenden Kapiteln entnommen werden.

Soweit möglich wurde versucht durch die Verwendung von SMD-Bauteilen die notwendige Platinenfläche zu reduzieren. Die Größe der doppelseitigen Platine beträgt 7,5 x 5,5 cm. Der vollständige Schaltplan und die Platinenlayouts können dem Anhang entnommen werden.

4.2 Der Aufbau der CAN-Schnittstelle

Die zentralen Bauelemente der CAN-Schnittstelle sind der CAN-Controller vom Typ SAE81C90 und der CAN-Treiber vom Typ PCA82C250.

Der CAN-Controller SAE81C90 enthält alle Funktionen, die notwendig sind, um autonom CAN-Telegramme zu senden und zu empfangen. Dabei wird die CAN Spezifikation 2.0A mit 11-Bit Identifiern unterstützt. Nachrichten gemäß der Spezifikation 2.0B mit 29-Bit Identifiern werden passiv behandelt. Dabei verwaltet der Controller 16 verschiedene CAN Nachrichtenobjekte. In einem Nachrichtenobjekt sind acht Byte für die Nutzdaten und zwei Byte, in denen die Anzahl der Nutzdatenbytes und der Identifier gespeichert werden, zusammengefaßt. Für die Nachrichtenobjekte 0 bis 7 sind weiterhin zwei Bytes reserviert, in denen ein sogenannter Timestamp gespeichert wird. Bei dem Timestamp handelt es sich um einen Zeitstempel, der den Zeitpunkt an dem die Nachricht empfangen wurde enthält.

Die Ansteuerung des Controllers kann entweder über eine SSI/SPI-Schnittstelle oder einen parallelen Interface erfolgen. Da die SSI/SPI-Schnittstelle des PIC-Controllers bereits für die Kommunikation mit dem VeCON-Chip benötigt wird muß in dem Kommunikationsinterface der parallele Interface benutzt werden. Der parallele Interface wird durch einen Low-Pegel am MS-Pin aktiviert. Da der PIC-Controller keinen externen Datenbus besitzt werden die I/O-Ports des PICs für diesen Zweck benutzt. Der gemultiplexte Adress/Datenbus des SAE81C90 wird dazu mit dem 8-Bit breiten Port B des PICs verbunden. Die Steuersignale des Controllers sind an Port A des PICs angeschlossen. Der externe Bus wird über diese Ports per Software emuliert.

Eingeleitet wird der Buszugriff durch einen Low-Pegel des CS-Signals (Chip Select). Jetzt ist der Businterface des CAN-Controllers aktiviert. Als nächstes wird das ALE-Signal (Adress Latch Enable) auf High-Pegel gesetzt und die Adresse des Registers, das angesprochen werden soll, über Port A ausgegeben. Durch die fallende Flanke des ALE-Signals wird die Adresse in den CAN-Controller übernommen (Zeitpunkt I in Abbildung 11 & Abbildung 12). Um einen Schreibzugriff durchzuführen wird nun das WR-Signal auf Low-Pegel gelegt und die Daten auf Port A ausgegeben. Durch die steigende Flanke des WR-Signals werden die Daten in das interne Register des CAN-Controllers übernommen (Zeitpunkt II in Abbildung 12). Bei einem Lesezugriff wird das RD-Signal auf Low-Pegel gelegt und der Registerinhalt über Port A eingelesen (Zeitpunkt II in Abbildung 11).

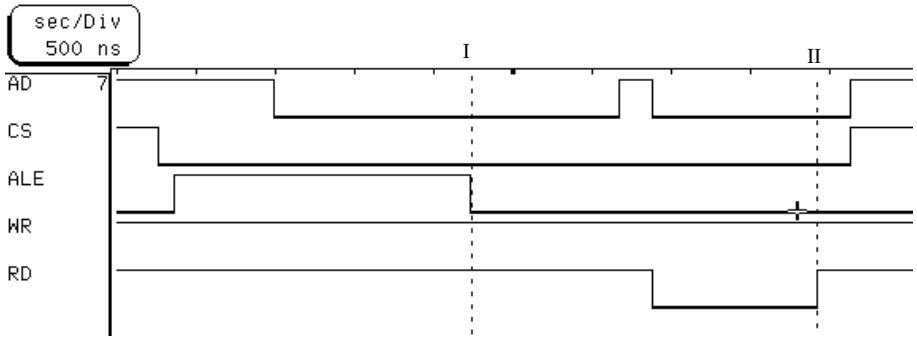


Abbildung 11: Timingdiagramm beim Lesezugriff

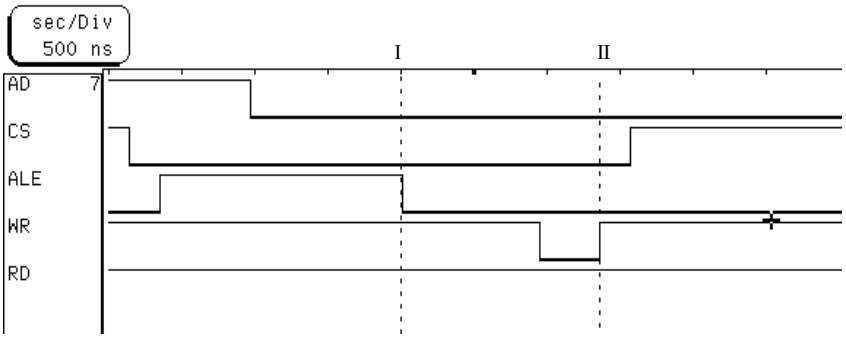


Abbildung 12: Timingdiagramm des Schreibzugriffs

Der CAN-Controller besitzt noch zwei weitere Steuerleitungen. Mit der RESET-Leitung kann der CAN-Controller zurückgesetzt werden. Diese Leitung ist mit dem PIN RA5 des PICs verbunden. Damit kann der CAN-Controller über die Software des PICs zurückgesetzt werden. Mit dem INT-Signal zeigt der CAN-Controller an, daß er Daten empfangen hat oder ein Fehler aufgetreten ist. Dieses Signal wird über den Pin RA4 des PICs ausgewertet. Über den Eingang X2 wird der CAN-Controller mit dem Taktsignal des Quarzoszillators versorgt. Die maximale Taktfrequenz, mit der der CAN-Controller betrieben werden kann, beträgt 20MHz.

Da der Bit-Stream-Prozessor im CAN-Controller sehr empfindlich auf Störungen, die über die Versorgungsleitung eingekoppelt werden, reagiert, sind Abblockkondensatoren in der Versorgungsspannung vorgesehen. Wichtig ist dabei, daß diese Kondensatoren so dicht wie möglich an den entsprechenden Pins des CAN-Controllers plaziert sind. Durch den 10nF Kondensator C7 wird die Empfindlichkeit des Bit-Stream-Prozessors gegenüber Störungen weiter reduziert.

Der CAN-Controller enthält keinen Treiber für den physikalischen Layer des CAN-Protokolls. Dieser Treiber ist extern durch den CAN-Transreceiver PCM82C250 von Philips realisiert. Dieser Transreceiver ist kompatibel zum ISO11898 Standard, bei dem die Daten mit Hilfe von differentiellen Signalen übertragen werden. Das TxD-Signal des Transreceivers wird mit dem digitalen Ausgang TXD0 des CAN-Controllers verbunden. Das vom CAN-Bus empfangene Signal wird über das RXD-Signal des Transreceivers an den RXD0-Eingang des CAN-Controllers geleitet.

Über den RS-Anschluß des Transreceivers kann die Anstiegsgeschwindigkeit der Signalflanken auf dem CAN-Bus begrenzt werden. Dadurch kann die elektromagnetische Abstrahlung von hochfrequenten Störungen, die durch steile Signalflanken erzeugt werden, reduziert werden. Die Anstiegszeit wird durch den Wert des Widerstands R1 festgelegt.

Die Anstiegsgeschwindigkeit der Flanke wird mit der Formel

$$T_s = \frac{1.5ns \cdot R_1}{(k\Omega)}$$ berechnet.

In der Schaltung wird ein Wert von 20kΩ für R1 verbaut, wodurch eine Anstiegszeit von 30ns realisiert wird. Mit dem Jumper JP1 kann der Widerstand R1 überbrückt werden, um den Transreceiver in den High-Speed-Modus zu schalten. Die CAN-H und CAN-L Signale des Transreceivers werden auf einen 9-poligen Sub-D-Stecker geführt, dessen Pinbelegung gemäß CiA Draft Standard 102 V2.0 ausgeführt ist.

An den I/O-Port 1 des CAN-Controllers werden die vier Status-LEDs angeschlossen. Ansteuerung der LEDs erfolgt dabei über invertivierende TTL-Treiber von Typ 74HCT06.

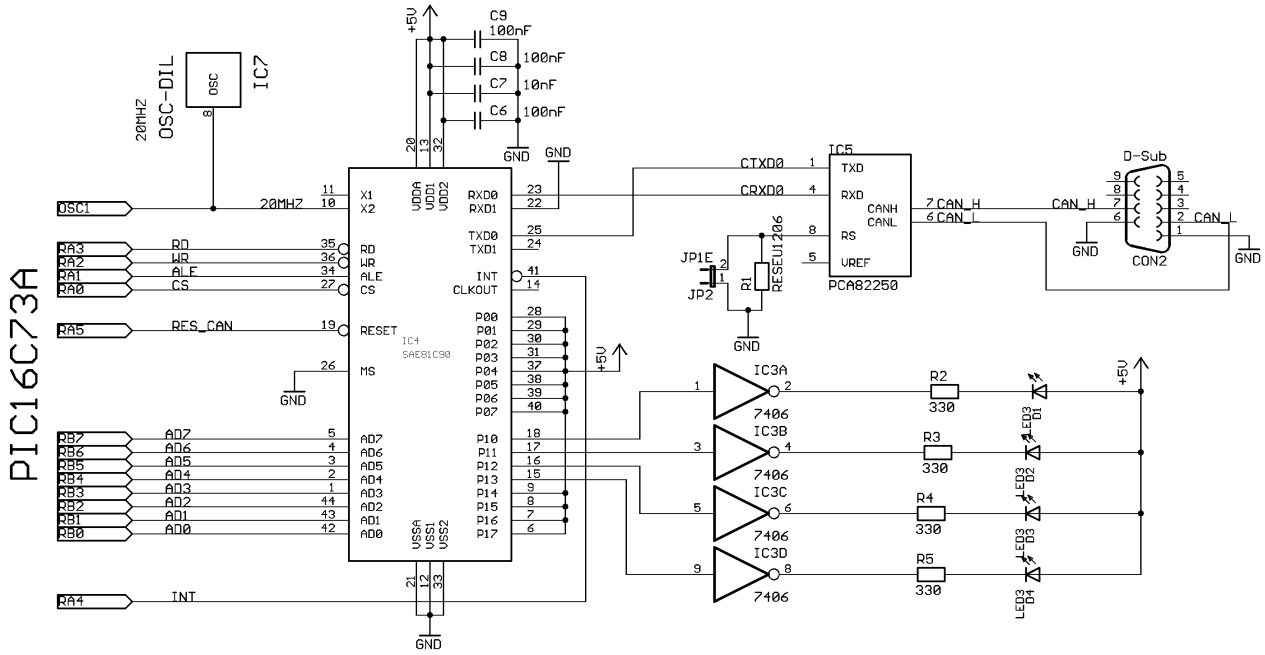


Abbildung 13: Schaltplan des CAN-Interfaces

4.3 Der Aufbau der RS232-Schnittstelle

Die RS232-Schnittstelle besteht aus einem asynchronen seriellen Interface (USART) und einem Pegelwandler. Die USART ist bereits auf dem PIC16C73 integriert. Die Initialisierung der USART wird per Software durchgeführt. Die Baudrate der asynchronen Schnittstelle wird dabei von der Taktfrequenz des PIC's abgeleitet. Bei einer Taktfrequenz von 20MHz kann die Baudrate zwischen 1.2Kbaud und 1250kbaud eingestellt werden. Einzelheiten zur Initialisierung der asynchronen Schnittstelle des PIC können dem Kapitel 5.1.2 entnommen werden.

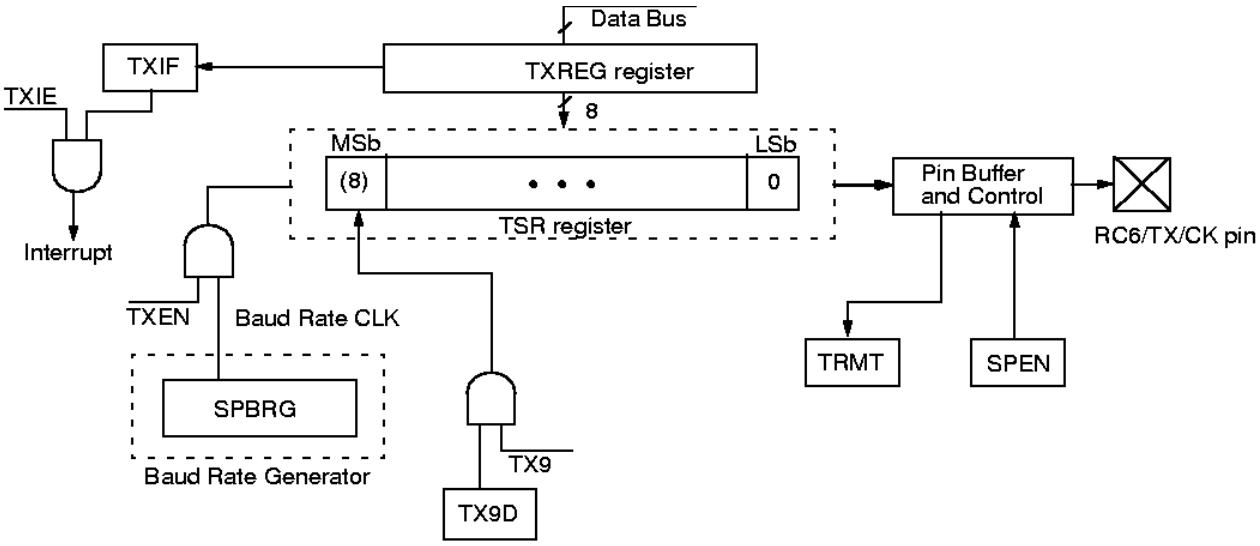


Abbildung 14: Blockschaltbild des USART-Transmitters des PIC

Der interne Aufbau des asynchronen Transmitters im PIC ist in Abbildung 14 dargestellt. Das zentrale Element des Transmitters ist das Transmitter Shift Register (TSR). Dieses Register wird über das TXREG-Register geladen. Wenn das TSR-Register leer ist, wird bei einem Schreibzugriff auf das TXREG-Register, der Wert des TXREG automatisch in das TSR-Register kopiert. Über dieses TSR-Register werden die Daten bitweise über den Pin RC6 herausgeschoben. Begonnen wird dabei mit dem LS-Bit. Die Länge des gesendeten Werts kann 8 oder 9 Bit betragen. Das optionale 9.Bit der Übertragung muß vor dem Schreibzugriff auf TXREG gesetzt werden. Nachdem der Wert „herausgeschoben“ wurde wird das TSR-Register automatisch mit einem neuen Wert aus dem TXREG geladen, wenn dort bereits ein neuer Wert abgelegt wurde.

Mit dem TXIF-Bit kann überprüft werden, ob der aktuelle Wert im TXREG bereits in das TSR-Register kopiert wurde oder nicht. Ist das TXIF-Bit gesetzt kann ein neuer Wert in das TXREG kopiert werden. Die Aktualisierung dieses Bits erfolgt automatisch durch die Hardware. Dieses Flag kann auch als Interrupt-Quelle konfiguriert werden. Außerdem kann über das TRMT-Bit überprüft werden, ob das Senden aus dem TSR-Register abgeschlossen wurde. Ist dieses Bit gesetzt, ist das Senden aus dem TSR-Register beendet.

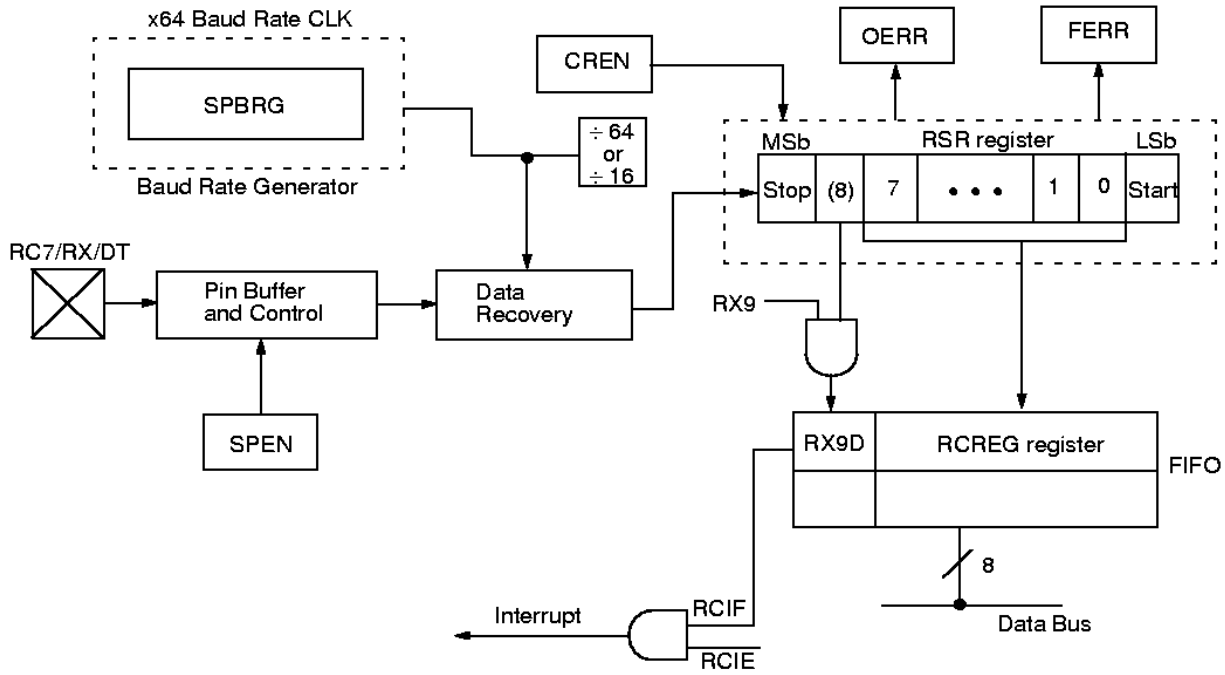


Abbildung 15: Blockschaltbild des USART-Receivers des PIC

Wie in der Abbildung 15 dargestellt, werden beim Receiver die Daten über den Eingang RC7 bitweise eingelesen und im RSR-Register (Receiver Shift Register) zwischengespeichert. Wurde ein Wert vollständig empfangen wird er automatisch in das RCREG kopiert. Hinter dem RCREG „versteckt“ sich ein FIFO-Speicher mit einer Speichertiefe von zwei Werten. Wenn bereits 2 Werte empfangen und nicht aus dem FIFO ausgelesen wurden und es wird ein weiter Wert empfangen wird über das OERR-Bit (Overrun Error Bit) angezeigt, das Daten beim Empfang verloren gegangen sind. Durch ein gesetztes RCIF-Bit wird angezeigt, das neue Daten empfangen wurden. Durch die Trennung der Sende- & Empfangslogik kann der PIC Daten Full-Duplex übertragen. D.h., die asynchrone Schnittstelle kann gleichzeitig Daten empfangen und senden.

Allerdings unterstützt die USART des PIC's kein Handshaking. Deshalb wird das RTS-Signal, mit dem der Kommunikationsinterface seine Bereitschaft zum Empfang von Daten anzeigt, über den I/O-Pin RC1 erzeugt. Ein CTS-Signal, mit dem der andere Teilnehmer seine Empfangsbereitsschaft anzeigen kann, ist nicht vorgesehen, weil davon ausgegangen wird, daß dieser Teilnehmer (Terminal) in der Lage ist „jederzeit“ Daten zu empfangen.

Der TTL-Pegel des Ein- und Ausgangssignals der USART wird durch einen invertierenden Pegelwandler vom Typ MAX232 an den RS232-Standard von +/- 10V angepaßt. Bei dem MAX232 handelt es sich um einen Spannungswandler, der aus einer Versorgungsspannung von +5V die notwendigen Spannungspegel für die RS232-Schnittstelle erzeugt. Die Beschaltung des Bausteins kann der Abbildung 16 entnommen werden.

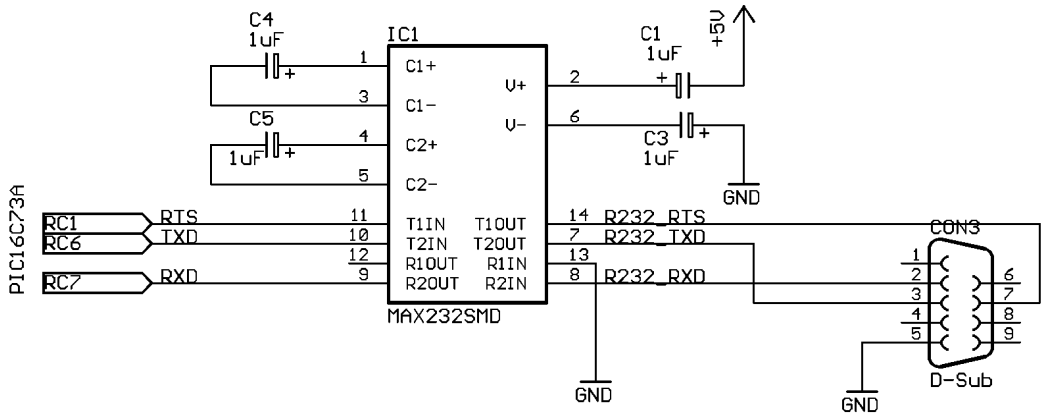


Abbildung 16: Beschaltung des MAX232

4.4 Der Aufbau der synchronen seriellen Schnittstelle

Das zentrale Element des synchronen seriellen Interface zum VeCon-Prototypenboard ist das SSI-Interface in dem PIC. Das vereinfachte Blockschaltbild des SSI-Interfaces im Slavemodus ist in der Abbildung 17 dargestellt. Im Slavemodus wird die Geschwindigkeit mit der die Daten übertragen werden durch den Takt am SCK-Eingang (RC3) bestimmt. Dieser Takt wird vom Master erzeugt, in diesem Fall also von der SSI-Schnittstelle des VeCon-Chipsatzes.

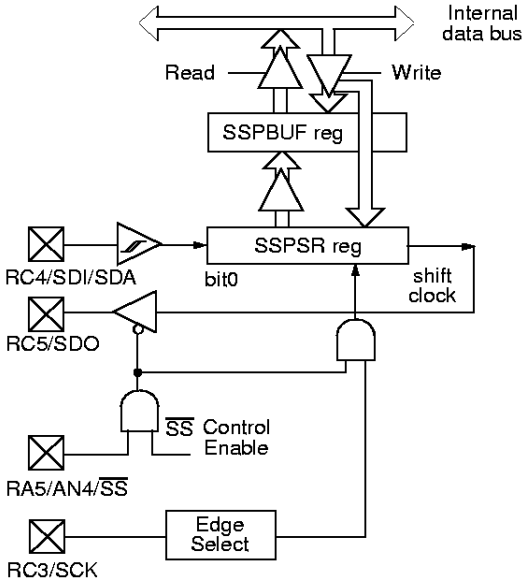


Abbildung 17: Vereinfachtes Blockschaltbild des SSI-Interfaces

Wie bereits in Kapitel 2.2 erwähnt, wird bei der SSI-Schnittstelle immer gleichzeitig gesendet und empfangen. Bei einem Schreibzugriff auf das SSPBUF-Register werden die Daten direkt in das SSPSR-Register kopiert. Wenn der Master eine Übertragung einleitet, wird der Inhalt dieses Registers bitweise an den Master gesendet. Gleichzeitig wird das bitweise vom Master empfangene Byte in dem Register gespeichert. Ist die Übertragung beendet wird das empfangene Byte im SSPSR-Register in das SSPBUF-Register kopiert. Von dort kann es ausgelesen und weiterverarbeitet werden. Außerdem wird das BF-Bit (Buffer Full) gesetzt, mit dem festgestellt werden kann, ob die Übertragung beendet ist.

Der im Blockschaltbild dargestellte Eingang an Pin RA5 wird bei dem Kommunikationsinterface nicht zur Steuerung des SSI-Interface eingesetzt, sondern als I/O-Pin. Die ursprüngliche Funktion des Pin RA5 für das SSI-Interface ist die Freigabe der SSI-Schnittstelle von außen.

Zusätzlich zu den Signalen SDI, SDO und SCK werden für die Kommunikation zwischen VeCon und PIC noch zwei Handshake-Signale benötigt. Eine Beschreibung der Funktion der beiden Handshake-Signale Received und Busy kann dem Kapitel 4 entnommen werden. Der Anschluß des Busy-Signals erfolgt an Pin RC2 und das Received-Signal ist an den Pin RC0 angeschlossen.

Die Signale der synchronen seriellen Schnittstelle und die Handshake-Signale werden über einen TTL-Bustreiber vom Typ 74HCT244 auf eine 10-polige Stiftleiste geführt. Durch den TTL-Bustreiber wird der Signalpegel der Schnittstellensignale angehoben. Durch Stecken des Jumpers JP5 kann das Kommunikationsinterface auch über diesen Stecker mit Spannung versorgt werden. Wird dieser Jumper nicht gesteckt, muß die Schaltung durch eine eigene Leitung an Stecker JP4 mit Spannung versorgt werden.

Nicht ganz unproblematisch an der SSI-Schnittstelle des PICs ist der SCK-Eingang. Schon sehr kurze Impulse und Spikes auf diesem Eingang werden von dem PIC als gültiges Taktsignal gewertet und das interne Schieberegister um ein Bit verschoben. Das führt dazu, daß alle nachfolgend übertragenen Datenbits in ihrer Position verschoben sind, es also zu einer fehlerhaften Übertragung kommt. Leider kann dieser Fehler nicht durch Status-Bits des SSI-Interface abgefragt und abgefangen werden. Aber selbst wenn der Fehler erkannt werden könnte, wäre es nur möglich ihn durch einen Reset des ganzen PIC-Controllers zu beheben. Es ist keine Möglichkeit vorgesehen das SSI-Interface allein zurückzusetzen.Es bleibt also nur die Möglichkeit Störungen und Peaks auf dem SCK-Signal zu vermeiden und ggf. zu unterdrücken. Um die Einkopplung von Störungen in das SCK-Signal zu verhindern wird empfohlen, ein möglichst kurzes abgeschirmtes Kabel für die Verbindung zwischen dem PIC und dem VeCon zu verwenden. Als zusätzliche Maßnahme ist vor dem Bustreiber im SCK-Signal ein Widerstand und ein Kondensator plaziert. Die Aufgabe dieser beiden Bauteile ist es die Energie von Störimpulsen abzubauen und so den Pegel von Störimpulsen zu dämpfen. Die Bauteilwerte sind so gewählt worden, daß zwar Störpegel gedämpft werden, aber die steigenden Flanken des SCK-Signals nicht verschliffen werden.

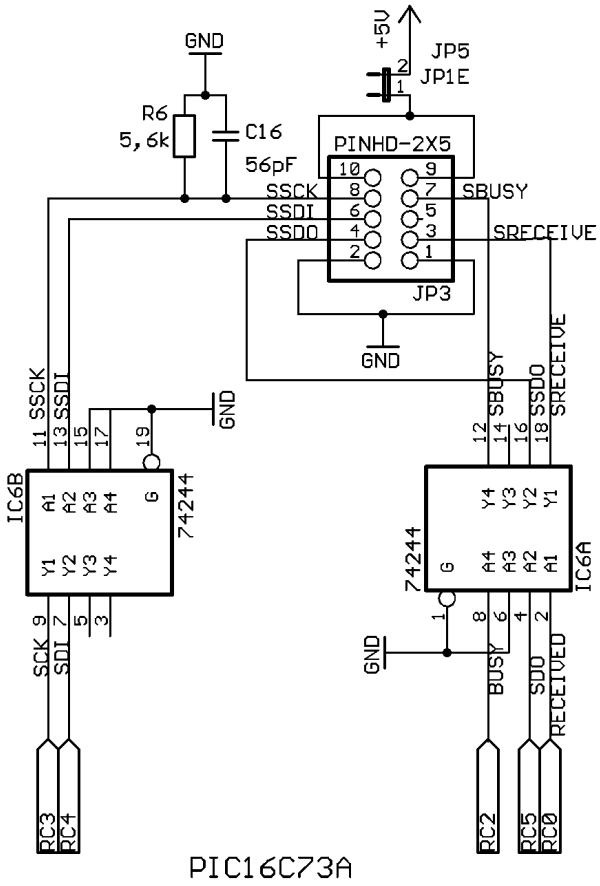


Abbildung 18: Beschaltung der SSI-Schnittstelle

5 Definition des Übertragungsprotokolls

Eine Voraussetzung für die Übertragung von Daten zwischen Teilnehmern ist, daß die übertragenen Daten von beiden Teilnehmern richtig interpretiert werden. Aus diesem Grund wird die Formatierung der Daten zu Telegrammen in diesem Kapitel beschrieben. Die Definition umfaßt dabei nicht nur den logischen Aufbau der übertragenen Daten, sondern auch die Definition von Hardware-Signalen mit denen die Übertragung koordiniert wird.

Wie bereits in Kapitel 2.1 beschrieben handelt es sich bei der Kommunikation über die synchrone serielle Schnittstelle um eine sogenannte Master/Slave-Übertragung. D.h., daß die Kommunikation immer von dem Master, in diesem Fall also dem VeCon-Prozessor, eingeleitet wird. Wenn der PIC-Controller also Daten an den VeCon senden möchte muß er dies dem VeCon-Prozessor signalisieren, damit dieser die Kommunikation beginnen kann. Für diesen Zweck ist das Received-Signal vorgesehen.

Mit dem Received-Signal wird dem VeCon-Board signalisiert, daß ein Datenbyte über die synchrone serielle Schnittstelle ausgelesen werden kann. Liegt ein neues Byte vor, wird dieses Signal logisch High gesetzt. Ist die Übertragung des Byte beendet wird das Signal Low. Jetzt wird das nächste Byte der Nachricht in das Senderegister kopiert und das Received-Signal wieder High gesetzt. Dieser Vorgang wird so oft wiederholt bis das ganze Datentelegramm übertragen wurde. Auf der VeCon-Seite wird dieses Signal über den Input-Pin CAP ausgelesen.

Neben diesem Handshake-Signal sind noch zwei weitere Handshake-Signale für eine sichere Übertragung von Daten notwendig. Da das Kommunikationsinterface drei voneinander unabhängige Schnittstellen besitzt muß ein Verfahren

sicherstellen, daß sich die Schnittstellen nicht gegenseitig blockieren. Dieser Fall könnte z.B. auftreten, wenn gleichzeitig an zwei Schnittstellen Daten empfangen werden. Aus diesem Grund werden, wenn an einer der Schnittstellen Daten empfangen werden, sofort die anderen Schnittstellen über ein Handshake-Signal für die anderen Teilnehmer gesperrt.

Die synchrone serielle Schnittstelle wird vom PIC durch einen High-Pegel des Busy-Signals blockiert. Solange am Busy-Signal ein High-Pegel anliegt, wartet der VeCon, der dieses Signal am MP1_IN-Pin einliest, mit dem Senden von Daten bis der PIC bereit ist.

Die asynchrone Schnittstelle wird durch das RTS-Signal gesperrt. Durch einen High-Pegel des RTS-Signals wird dem PC, der als Quelle der Datenübertragung dient, signalisiert, daß der PIC im Augenblick nicht bereit ist Daten zu empfangen. Erst wenn das RTS-Signal wieder Low-Pegel führt beginnt der PC mit dem Senden von Daten.

Für die CAN-Schnittstelle ist kein Handshake-Signal vorgesehen, weil diese Schnittstelle durch den CAN-Controller, der in der Lage ist in jeder Messagebox jeweils eine vollständige Nachricht zu speichern, bereits ausreichend von der restlichen Schaltung des Kommunikationsinterfaces entkoppelt ist. Dadurch ist der CAN-Controller in der Lage die Nachrichten zwischen zu speichern bis der PIC die empfangenen Daten verarbeiten kann, ohne daß Nachrichtentelegramme verloren gehen.

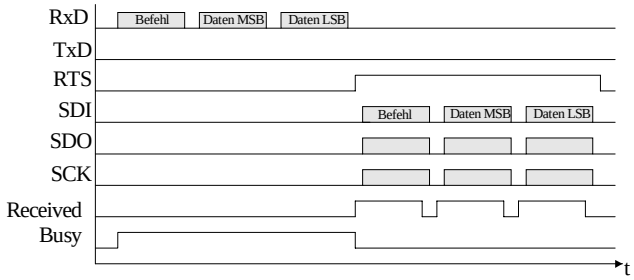


Abbildung 19: Datenübertragung RS232 auf SSI

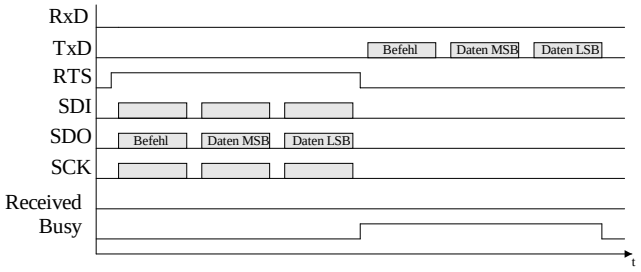


Abbildung 20: Datenübertragung SSI auf RS232

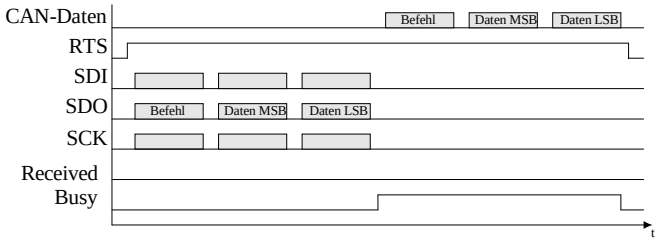


Abbildung 21: Datenübertragung SSI auf CAN

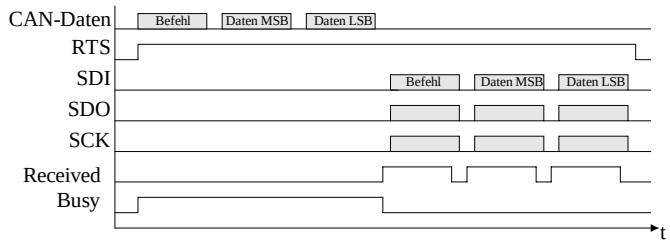


Abbildung 22: Datenübertragung CAN auf SSI

In den Abbildungen 19 bis 22 sind die Signalverläufe der Handshake-Signale für verschiedene Übertragungen noch mal zusammenfassend dargestellt. In den Beispielen wird jeweils ein Befehlsbyte und zwei Datenbyte übertragen.

Der Aufbau der einzelnen Datentelegramme ist unabhängig davon, ob Daten über die synchrone oder die asychrone Schnittstelle übertragen werden. Jedes Telegramm besteht aus einem Befehlsbyte am Anfang, das von 0 bis 8 Datenbytes gefolgt wird. Bei den Datenbytes wird immer das MSB (Most Significant Byte) als erstes und das LSB (Least Significant Byte) als letztes gesendet. Der genaue Aufbau des Befehlsbyte kann der Tabelle 2 entnommen werden.

In den Datentelegrammen wurden keine Methoden zur Erkennung und Korrektur von Übertragungsfehlern, wie z.B. Checksummen, implementiert. Der Grund hierfür ist der sehr begrenzte Programmspeicher des VeCon-DSPs.

Bit 7.	***Bit 6.***	***Bit 5.***	***Bit 4.***	***Bit 3.***	***Bit 2.***	***Bit 1.***	***Bit 0.***
RS232/CAN	REMOTE CAN	Anzahl Datenbyte		Messagebox-Nr.			

Tabelle 2: Aufbau des Befehlsbyte

- Bit 7. RS232/CAN: Mit diesem Bit wird das Ziel bzw. die Quelle einer Datenübertragung über die SSI-Schnittstelle angegeben. D.h. daß der VeCon beim Senden mit diesem Bit angibt, ob das Telegramm auf den CAN-Bus oder die RS232 gesendet werden soll. Beim Empfang kann der VeCon an diesem Bit feststellen, ob das Telegramm über den CAN-Bus oder die RS232-Schnittstelle empfangen wurde. Bei Übertragungen über die RS232-Schnittstelle hat dieses Bit keine Funktion.

RS232/CAN	***Ziel/Quelle der Übertragung***
0	RS232
1	CAN-Bus

Tabelle 3: Definition des RS232/CAN-Bits

- Bit 6. REMOTE CAN: Mit dem REMOTE CAN-Bit kann ein Remote Request auf dem CAN-Bus ausgelöst werden.

REMOTE CAN	***Funktion***
0	Standard Übertragung
1	Remote Request

Tabelle 4: Definition des REMOTE CAN-Bits

- Bit 5.& 4. Anzahl Datenbyte: Mit diesen zwei Bit wird die Anzahl der Datenbytes in dem Telegramm festgelegt.

Bit 5.	*Bit 4.*	*Anzahl Datenbyte*
0	0	1 Byte
0	1	2 Byte (INT-Wert)
1	0	4 Byte (LONG-Wert)
1	1	8 Byte

Tabelle 5: Definition der Anzahl Datenbyte

- Bit 3.-0. Messagebox-Nr.: Durch die vier Bit wird die Ziel bzw. Quell Messagebox der Übertragung definiert. Bei einer Übertragung zum bzw. vom CAN-Bus ist durch die Messagebox-Nr. das Nachrichtentelegramm fest mit der Messagebox verknüpft. Da für jede Messagebox der Identifier, die Länge der CAN-Nachricht und die Art der Nachricht (Remote oder Standard) in der Firmware des Kommunikationsinterface festgelegt ist, ist über die Messagebox-Nr. ein CAN-Telegramm vollständig definiert. Bei einer Übertragung über die RS232-Schnittstelle wird die Messagebox-Nr. nur dazu benutzt verschiedene Datensätze auseinander zu halten und zu sortieren. Z.B. kann jeder Messagebox-Nr. eine Variable im VeCon-Programm zugeordnet werden. Gültige Messagebox-Nummern sind die Zahlenwerte 1 bis 15. Die Messagebox 0 wird zur Übermittlung von Fehlercodes benötigt.

Durch das Befehlsbyte sind drei verschiedene Datentelegrammtypen definiert, das „normale“ Datentelegramm, das Remote Request-Telegramm und das Fehlermeldungstelegramm.

Mit einem „normalen“ Datentelegramm werden Daten zwischen den Teilnehmern über die SSI- oder RS232-Schnitsstelle ausgetauscht. Bei einer Übertragung über die SSI-Schnittstelle wird das Ziel der Übertragung mit dem RS232/CAN-Bit festgelegt. Der Aufbau eines „normalen“ Datentelegramms ist in der Abbildung 23 dargestellt. Das Beispiel in Abbildung 23 zeigt ein Datentelegramm mit zwei Datenbytes, das vom VeCon über die SSI-Schnittstelle auf den CAN-Bus übertragen werden soll. Der Identifier für die Übertragung auf dem CAN-Bus wird dabei über die Messagebox-Nr. ermittelt. In diesem Beispiel soll die Nachricht auf den CAN-Bus über die Messagebox 4 gesendet werden.

Befehl	*Daten 0*	*Daten 1*
1001 0100	MSB	LSB

Abbildung 23: Aufbau eines "normalen" Datentelgramms

In Abbildung 24 ist der Aufbau eines Remote Request-Telegramms dargestellt. Dieses Telegramm besteht nur aus dem Befehlsbyte. In dem Beispiel wird ein Remote-Request auf dem CAN-Bus mit den Identifier der Messagebox-Nr. 13 ausgelöst. Die Bits, mit denen die Anzahl der Datenbytes definiert sind, werden bei diesem Telegrammtyp nicht ausgewertet.

Befehl
11XX 1101

Abbildung 24: Aufbau eines Remote Requests

Ein Beispiel für ein Fehlertelegramm ist in der Abbildung 25 dargestellt. Das Fehlertelegramm wird durch die Messagebox-Nr. 0 gekennzeichnet. Es besteht neben dem Befehlsbyte aus einem Datenbyte, das den Fehlercode enthält. In diesem Beispiel wird ein Fehlertelegramm vom Kommunikationsinterface an den VeCon geschickt.

Befehl	*Fehlercode*
1000 0000	0000 0XXX

Abbildung 25: Aufbau eines Fehlertelegramms

Es sind die folgenden Fehlercodes definiert:

Fehlercode	*Funktion*
0	Undefinierter Fehler
1	Error Warning Level beim CAN-Controller überschritten
2	CAN-Controller im Error Passiv-Modus
4	CAN-Controller im Bus Off-Modus

Tabelle 6: Definition der Fehlercodes

6 Die Software

6.1 Die Interfacefirmware

6.1.1 Konzept & Entwicklungsziel

Die Firmware des Kommunikationsinterfaces hat die Aufgabe die Schnittstellen zu initialisieren und die Übertragung zwischen den drei Schnittstellen sicherzustellen. Dazu muß die Software die empfangenen Daten gemäß dem Protokoll aus Kapitel 4 interpretieren und weitersenden. Im Verlauf der Kommunikation ist die Software auch für die Ansteuerung der Handshake-Leitungen zuständig. Eine weitere Aufgabe der Software ist es Fehler, die im CAN-Controller auftreten, an das VeCon-Board weiterzuleiten.

Ein besonderes Augenmerk wurde bei der Entwicklung der Firmware auf den modularen Aufbau der Software gelegt. Im Gegensatz zu Hochsprachen, wie C oder Pascal, ist es in Assembler nicht möglich Module in Form von Prozeduren und Funktionen zu definieren, denen Variablen übergeben werden können. Statt dessen können in Assembler einzelne Funktionen in Unterprogrammen verpackt werden, in die dann verzweigt werden kann. Bei dieser Entwicklung werden z.B. die Funktionen für den Empfang und Versand von Daten für die einzelnen Schnittstellen in eigenen Dateien zusammengefaßt. Diese Files können dann in das Hauptprogramm importiert werden.

Für diese Art der modularen Softwareentwicklung bietet das MPLAB-Entwicklungssystem von Microchip zwei Konstrukte, Macros & Include-Files. In Macros können kleine Programmteile zusammengefaßt werden. Außerdem besteht die Möglichkeit bei der Definition von Marco's Platzhalter für die verwendeten Variablen zu benutzen. In einem Programm werden die Macro-Aufrufe dann direkt durch den Source-Code des Macros und die Platzhalter durch die Variabelnamen ersetzt. Daraus folgt, daß ein Programm länger wird, wenn ein Macro mehrfach in einem Programm verwendet wird. Die zweite Möglichkeit Source-Code aus verschiedenen Files zu importieren ist das Include-Konstrukt. Der Source-Code des in der Include-Anweisung angegebenen Files wird bei der Assemblierung direkt an der Stelle des Aufrufs eingefügt. Im Gegensatz zu Macro's besteht bei Include-Files nicht die Möglichkeit für Variablennamen Platzhalter zu verwenden. In diesem Projekt wurden z.B. die Funktionen für die einzelnen Schnittstellen in Include-Files zusammengefaßt und aus dem Hauptprogramm mit dem CALL-Befehl aufgerufen.

Durch den modularen Aufbau der Firmware ergeben sich mehrere Vorteile bei der Entwicklung der Software und bei nachträglichen Erweiterungen, Modifikationen usw. Durch die Aufteilung der Software in einzelne Teile ist es z.B.

möglich schon während der Entwicklung einzelne Module zu testen und ihre Funktionalität zu verifizieren. Ein weiterer Vorteil, der sich durch die Modularisierung der Software ergibt ist, daß sich die Module durch ihre funktionale Abgeschlossenheit auch für zukünftige Projekte weiter verwenden lassen.

Im Rahmen dieser Studienarbeit hat sich das Konzept des modularen Softwareentwurf in der oben beschriebenen Form sehr gut bewährt. Denn durch die Nutzung dieses Entwurfskonzepts war es möglich neben der Software für das RS232/CAN auf SSI-Interface auch eine Software für ein RS232 auf CAN-Interface zu entwickeln. Für dieses RS232 auf CAN-Interface wird die gleiche Hardware und die gleichen Software-Module wie im RS232/CAN auf SSI-Interface verwendet. Einzig die Verknüpfungen der einzelnen Module im Hauptprogramm und die Initialisierungsdaten der Messageboxen unterscheiden sich bei den beiden Programmen.

Insgesamt besteht die Firmware des Kommunikationsinterfaces aus den folgenden Sourcecode-Files:

- *CAN.MAC:* Dieser Macro enthält die Funktionen zum Lesen und Schreiben der Register des CAN-Controllers, Ein- & Ausschalten der Status-LEDs und den Macro zur Initialisierung der Messageboxen (Siehe Kap. 5.1.4)
- *CANINIT.INC:* In diesem Include-File wird die Initialisierung des CAN-Controllers vorgenommen. (Siehe Kap. 5.1.4)
- *CAN_BOX.INC:* Die Konfigurationsdaten der Messageboxen sind in diesem Include-File abgelegt. (Siehe Kap. 5.1.4)
- *CAN_COM.INC:* Dieses Include-File enthält die Funktionen mit denen Daten über den CAN-Bus empfangen bzw. gesendet werden. (Siehe Kap. 5.1.7)
- *INITIAL3.INC:* In diesem File wird die Initialisierung der PIC internen Hardware vorgenommen. (Siehe Kap. 5.1.3)
- MAIN.ASM: MAIN.ASM enthält das Hauptprogramm von dem aus die ganzen Unterprogramme aufgerufen werden. (Siehe Kap. 5.1.2)
- RS232.MAC: In diesem Marco sind die Funktionen zum Senden & Empfangen über die RS232-Schnittstelle definiert. (Siehe Kap. 5.1.5)
- RS_VAR.INC: Die Definition von Variablen und die Zuordnung von Speicherplätzen wird in diesem File vorgenommen. (Siehe Kap. 5.1.2)
- SPI.MAC: Dieses Macro enthält die Funktionen zum Senden & Empfang über die SSI-Schnittstelle. (Siehe Kap. 5.1.6)

6.1.2 Die Funktion des Hauptprogramms

6.1.2.1 Die Initialisierung und die Struktur der Abfrageroutine

Die Aufgaben des Hauptprogramms, das in dem Sourcefile „MAIN.ASM“ unterbracht ist, lassen sich in zwei Bereiche unterteilen: die Initialisierungs- und die Betriebsphase.

Die Initialisierungsphase wird durch das Anlegen der Betriebsspannung oder das Betätigen des Reset-Knopfes ausgelöst. Der PIC beginnt dann mit der Abarbeitung des Hauptprogramms.

Als erstes werden die Variablen deklariert und initialisiert. Dazu wird vom Hauptprogramm das File „RS_VAR.INC“ eingebunden, in dem die eigentliche Deklaration vorgenommen wird.

Nachdem dies erfolgt ist, wird die PIC interne Hardware und der CAN-Controller initialisiert. Dazu werden die Files „INITIAL3.INC“ und „CANINIT.INC“ eingebunden. Einzelheiten zur Initialisierung der PIC-Hardware und des CAN-Controllers können den nachfolgenden Kapiteln entnommen werden.

Ist die Initialisierung der Variablen und der Hardware erfolgt wird die RS232-Schnittstelle freigeschaltet. Damit ist die Initialisierungsphase abgeschlossen und das Kommunikationsinterface ist jetzt betriebsbereit.

In der Betriebsphase durchläuft das Hauptprogramm, wie in Abbildung 43 dargestellt, immer wieder eine Schleife. In dieser Schleife werden drei ineinander verschachtelte Zähler dekrementiert, die als Verzögerungsglied dienen. Nachdem die Zähler durchlaufen sind überprüft das Programm, ob ein im Errata Sheet zum SAE80C90 aufgeführter Fehler im Chip (Step D13) aufgetreten ist. Sollte dies der Fall sein, wird der ErrorCode auf den Wert 4 gesetzt und über die SSI-Schnittstelle an das VeCon-Board

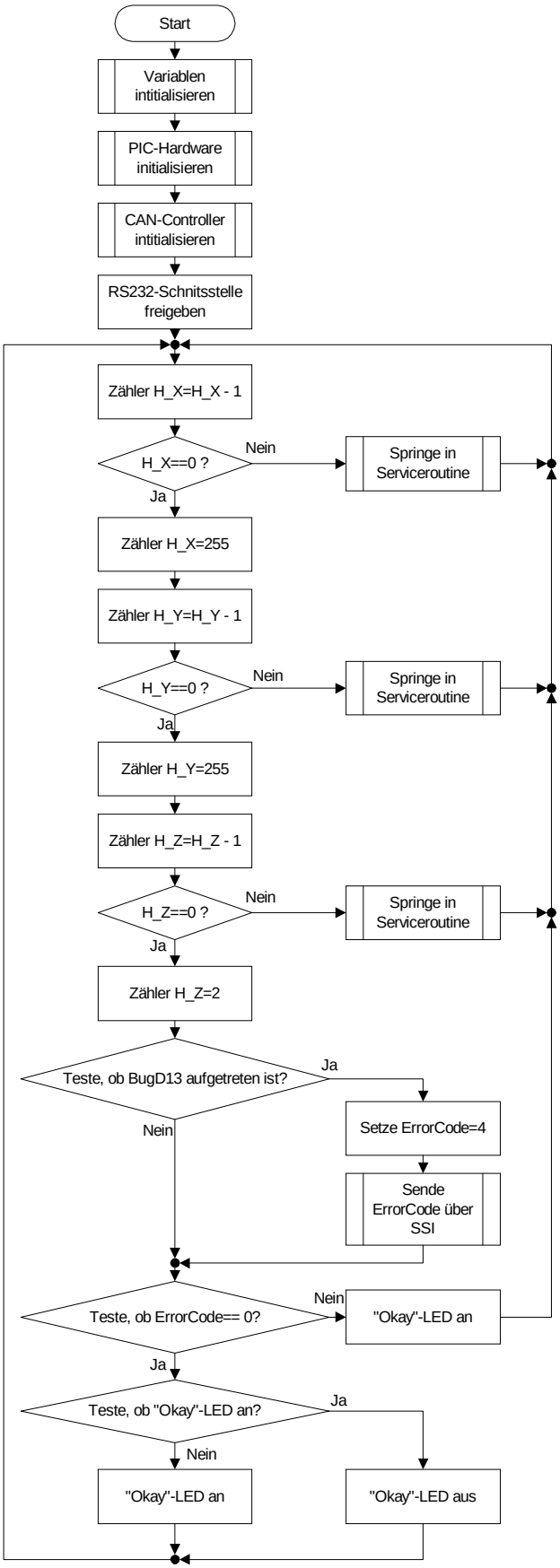


Abbildung 26: Flußdiagramm des Hauptprogramms

gesendet. Wie in Kapitel 4. beschrieben bedeutet dieser ErrorCode, daß der CAN-Controller sich in den Bus_Off-Modus geschaltet hat.

Am Ende der Schleife wird der aktuelle Zustand der Okay-LED, deren blinken signalisiert das das Kommunikationsinterface ordnungsgemäß funktioniert, invertivert. Diese Invertierung erfolgt aber nur, wenn kein Fehler aufgetreten ist, also der Wert der Variablen ErrorCode=0 ist. Ist dies nicht der Fall wird die Okay-LED in jedem Fall eingeschaltet. D.h. also, das ein dauerndes Leuchten der Okay-LED einen Fehler anzeigt.

Nach jeder Dekrementierung der Zählervariablen in der Hauptschleife springt das Programm in die sogenannte Service-Routine.

Wie in Kapitel 5.1 beschrieben gibt es zwei Versionen der Software, die RS232/CAN auf SSI und die RS232 auf CAN-Version. Diese beiden Versionen der Software unterscheiden sich nur durch die Serviceroutinen.

6.1.2.2 Die Serviceroutine der RS232/CAN auf SSI-Version

Das Flußdiagramm der Serviceroutine für die RS232/CAN auf SSI-Version ist in Abbildung 27 dargestellt.

Die Aufgabe dieses Unterprogramms ist es zu überprüfen, ob an einer der Schnittstellen neue Daten eingetroffen sind. Ist dies der Fall leitet sie den Empfang und den späteren Versand des Telegramms über die richtige Schnittstelle ein.

Nachdem in diese Routine verzweigt wurde wird als erstes der Watchdog-Timer zurückgesetzt. Der Watchdog-Timer muß alle 288ms zurückgesetzt werden, ansonsten löst er einen Reset aus. Damit ist sichergestellt, das das Kommunikationsinterface bei auftreten eines schweren Fehlers, der zum Absturz der Firmware führt, nach spätestens 288ms wieder voll funktioniert.

Danach wird überprüft, ob eine der Schnittstellen Daten empfangen hat. Wurde ein Byte über die SSI-Schnittstelle empfangen wird sofort die RS232-Schnittstelle durch einen Low-Pegel des CTS-Signals gesperrt. Erst danach wird das Unterprogramm „SPI_Load“ aus dem File „SPI.INC“ aufgerufen, die das ganze Telegramm über die SSI-Schnittstelle einliest. Während die Daten über die SSI-Schnittstelle gelesen werden ist die SSI-Led eingeschaltet.

Nachdem das Telegramm vollständig empfangen wurde, überprüft das Programm wohin die empfangenen Daten gesendet werden sollen. Dazu überprüft die Routine, ob das RS232/CAN-Bit in dem Befehlsbyte gesetzt ist. Ist dieses Bit =“1“ wird das empfangene Telegramm durch das Unterprogramm „CAN_PUT“ aus dem File „CANCOM.INC“ in den CAN-Controller kopiert und von dort aus auf den CAN-Bus gesendet. Ist aber das RS232/CAN-Bit =“0“ wird das Telegramm mit der Funktion „RS232_SAVE“ aus dem File „RS232.MAC“ über die RS232-Schnittstelle an das Terminal gesendet. Danach wird die Schnittstelle wieder freigeschaltet und aus den Unterprogramm wieder zurück in die Schleife des Hauptprogramms gesprungen.

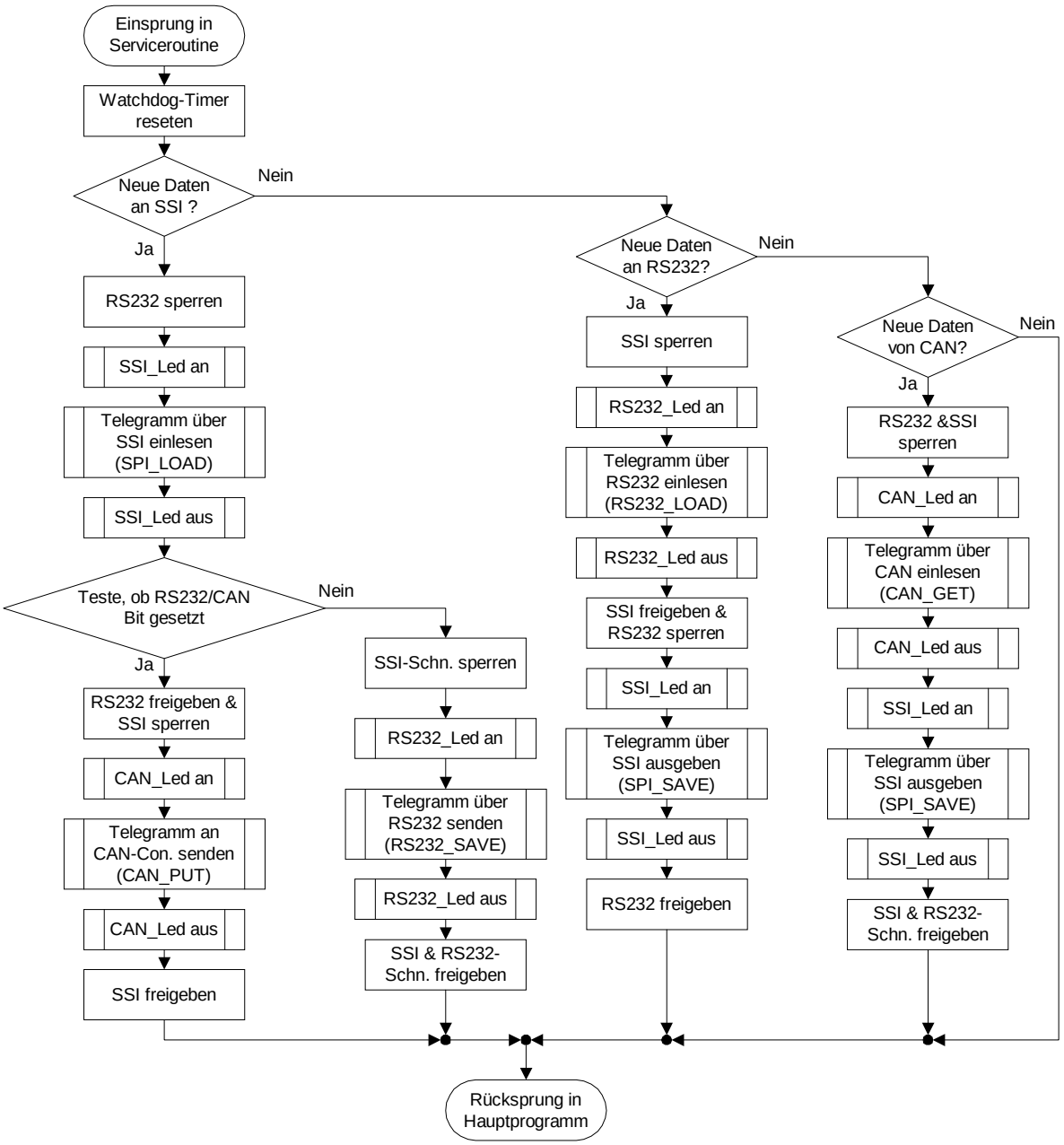


Abbildung 27: Flußdiagramm der Service-Routine

Wurde ein Byte an der RS232-Schnittstelle empfangen, wird mit dem Unterprogramm „RS232_Load“ das ganze Telegramm empfangen und danach mit der Funktion „SPI_SAVE“ zum VeCon gesendet. Sind Daten über dem CAN-Bus empfangen worden, werden die Daten mit der Funktion „CAN_GET“ aus dem CAN-Controller ausgelesen und wieder mit der Funktion „SPI_SAVE“ zum VeCon gesendet.

6.1.2.3 Die Serviceroutine der „Spezialversion“ RS232 auf CAN-Version

Der Aufbau der Initialisierungs- und Abfrageroutinen dieser Firmware-Version ist identisch mit der Version, die in Abbildung 43 dargestellt ist. Einzig im Aufbau der Service-Routine, wie in Abbildung 28 dargestellt, unterscheiden sich diese beiden Programme.

Bei dieser Serviceroutine werden die Telegramme nur zwischen der RS232-Schnittstelle und dem CAN-Controller ausgetauscht, aus diesem Grund sind alle Funktionen, die für die SSI-Schnittstelle benötigt werden nicht notwendig. Bei dem RS232 auf CAN-Interface werden die Daten, die mit der Funktion „RS232_Load“ empfangen werden direkt mit der Funktion „CAN_PUT“ in den CAN-Controller kopiert. Über den CAN-Bus empfangene Daten werden mit der Funktion „CAN-GET“ in den Mikrocontroller kopiert und von dort aus mit der Funktion „RS232_SAVE“ über die RS232-Schnittstelle an das Teminal gesendet. Die Unterprogramme dieser Programmversion unterscheiden sich nicht von denen des RS232/CAN auf SSI Kommunikations-interfaces.

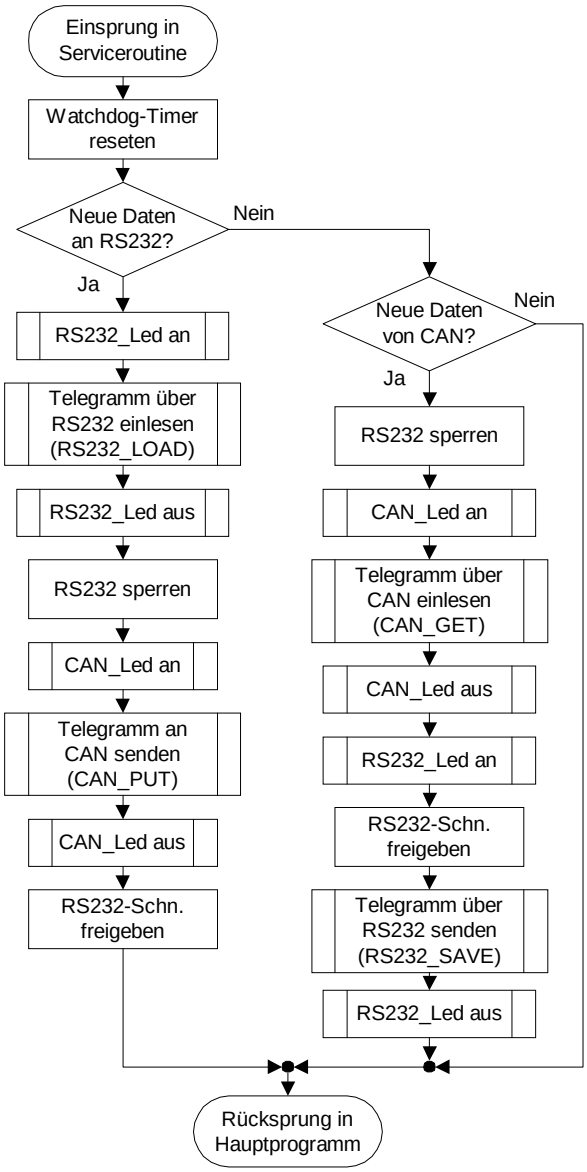


Abbildung 28: Flußdiagramm der Serviceroutine des RS232 auf CAN-Interfaces

6.1.3 Initialisierung des PIC's

In den PIC-Mikrocontrollern der 16C7X-Familie sind bereits verschiedene Funktionseinheiten wie A/D-Wandler, serielle Schnittstellen, Timer usw. auf den Chip integriert. Für die Vielzahl von Funktionen stehen aber nicht ausreichend viele I/O-Pins zur Verfügung. Unter anderem aus diesem Grund müssen die Hardwarefunktionen des PIC's konfiguriert und initialisiert werden. Die Konfiguration erfolgt durch das Schreiben in speziellen Peripherieregistern mit denen die einzelnen Funktionen an bzw. aus geschaltet werden können. Eine vollständige Beschreibung der Peripherieregister im PIC und deren Funktion kann [Mic1] entnommen werden.

Die Konfigurations- und Initialisierungsdaten sind in dem File „INITIAL3.INC“ untergebracht, das vom Hauptprogramm aus eingebunden wird. Neben den Konfigurationsdaten wird in diesem File auch jedem I/O-Pin ein Name zugeordnet. Dieser Name entspricht den im Schaltplan angegebenen Signalnamen. So ist es möglich in den Source-Files direkt mit den Namen zu arbeiten, wodurch der Source-Code besser lesbar wird.

Für die Konfiguration der asynchronen Schnittstelle sind das RCSTA- (Receiver Status) und das TXSTA-Register (Transmitter Status) zuständig. Mit Hilfe dieser beiden Register wird die asynchrone Schnittstelle in den 8-Bit

Übertragungsmodus geschaltet und freigegeben. Die Einstellung der Baudrate erfolgt mit Hilfe des BRGH-Bit im Register TXSTA und dem Wert des SPBRG-Registers (Baud Rate Generator Register). Bei einer Taktfrequenz von 20 MHz kann die Baudrate prinzipiell auf Werte zwischen 1,2 kBaud und 1250 kBaud eingestellt werden. Allerdings kann nicht jede gewünschte Baudrate eingestellt werden, weil der Baudratenteiler mit diskreten 8 Bit langen Teilerwerten arbeitet. Die Berechnung der Baudrate erfolgt je nach Modus mit den folgenden Formeln:

$$Baud\ Rate = F_{OSC} / (64 * (SPBRG + 1))\ für\ BRGH = 0$$

$$Baud\ Rate = F_{OSC} / (4 * (SPBRG + 1))\ für\ BRGH = 1$$

F_{osc}: Oszillatorfrequenz (hier 20MHz)

In der folgenden Tabelle sind verschiedene Baudraten für die beiden Betriebsmodi zusammengefaßt. Da die gewünschten Übertragungsraten nicht exakt auf den gewünschten Wert eingestellt werden können, ist für jeden Wert die Abweichung in Prozent angegeben.

	BRGH=0			***BRGH=1***		
Baudrate	KBaud	Abweichung %	SPBRG (dezimal)	KBaud	Abweichung %	SPBRG (dezimal)
1,2	1,221	1,73	255	-	-	-
2,4	2,404	0,16	129	-	-	-
9,6	9,469	-1,36	32	9,615	0,16	129
19,2	19,530	1,73	15	19,230	0,16	64
38,4	39,062	1,72	7	37,878	-1,36	32
57,6	-	-	-	56,818	-1,36	21
115,2	-	-	-	113,636	-1,36	10

Tabelle 7: Auswahl von Baudraten bei 20 MHz Oszillatorfrequenz

Prinzipiell eignet sich der Betriebsmodus BRGH=1 besser für die Baudratenerzeugung, weil dabei höhere Übertragungsraten möglich sind. Im Daten- und im Errata-Sheet zum PIC16C73A wird aber davon abgeraten diesen Modus bei der aktuellen Chip-Revision zu verwenden. Laut Datenblatt muß in diesem Modus mit Fehlern beim Empfang gerechnet werden. Aus diesem Grund wird die Baudrate im Modus BRGH=0 auf die höchste zum PC-kompatible Übertragungsrate von 38,4 KBaud eingestellt.

Die Konfiguration der synchronen Schnittstelle erfolgt über die Register SSPCON und SSPSTAT. Das Register SSPCON (Synchronous Serial Port Control) enthält alle Konfigurationsbits, die zur Steuerung der SSI-Schnittstelle dienen. In dem SSPSTAT (Synchronous Serial Port Status) sind die Status-Bits, die über den Zustand der synchronen seriellen Schnittstelle Auskunft geben, zusammengefaßt. Mit den Bits 0-3 des SSPCON Registers kann der Modus der Schnittstelle eingestellt werden. In der Initialisierungsroutine wird die Schnittstelle als SSI/SPI-Slave konfiguriert. Mit dem Bit 4 kann festgelegt werden mit welcher Flanke die Übernahme der Daten erfolgt. Durch das Löschen dieses Bits erfolgt die Übernahme der Daten beim Empfang über die SSI-Schnittstelle bei der steigenden Flanke des SCK-Signals. Die Bits, die vom PIC gesendet werden, werden bei der fallenden Flanke des SCK-Signals übergeben. Durch das Setzen von Bit 5 wird die Schnittstelle freigegeben.

Bevor die seriellen Schnittstellen betriebsbereit sind müssen die Ausgabetreiber der Ports konfiguriert werden. Bei der Konfiguration wird festgelegt, ob ein Port-Pin als Aus- oder als Eingang benutzt wird. Die Signale der beiden seriellen

Schnittstellen liegen an Port C an. Dieser Port wird so konfiguriert, daß die Pins, über die Daten empfangen werden sollen also SCK, SDI und RxD, als Input programmiert werden. Dazu werden im TRISC-Register die entsprechenden Bits „High“ gesetzt. Die anderen Pins dieses Ports dienen als Ausgang.

Auch beim Port A und B muß die Datenrichtung konfiguriert werden. Die I/O-Pins des Port A, die als Steuerleitungen für den Datentransfer zum CAN-Controller benutzt werden, werden fast alle als Ausgang programmiert. Nur der INT-Pin, mit dem der CAN-Controller signalisiert, daß er Daten empfangen hat, ist als Eingang konfiguriert. Die Pegel der Steuerleitungen werden auch in der Initialisierungsroutine bereits gesetzt. D.h., daß Signale, die „Low“-aktiv sind, mit einem „High“-Pegel initialisiert werden, während die „High“-aktiven Signale mit einem „Low“-Pegel initialisiert werden. Durch diese Maßnahme ist sichergestellt, daß nach der Initialisierung die Signale an Port A keinen aktiven Zustand im Zugriffzyklus darstellen. Der Port B, der als Adress-/Datenbus dient, wird während der Initialisierung als Eingang programmiert. Später wird dieser Port bei jedem Schreib-/Lesezugriff entsprechend konfiguriert.

Die anderen auf dem PIC untergebrachten Funktionseinheiten werden von der Firmware des Kommunikationsinterface nicht benutzt und werden deshalb während der Initialisierung abgeschaltet.

6.1.4 Initialisierung des CAN-Controllers

Die Initialisierung des CAN-Controllers erfolgt durch den Aufruf des Unterprogramms „CANINIT“, das sich in der Datei „CANINIT.INC“ befindet. Bei der Initialisierung werden die Übertragungsparameter für den CAN-Bus, als auch die Parameter einzelner Messageboxen, konfiguriert.

Weil die Zugriffe auf den CAN-Controller über die I/O-Ports des PICs erfolgen wurden die Zugriffsverfahren [Siehe Kapitel 3.2] in Unterprogrammen zusammengefaßt. Der Schreib- und Lesezugriff während der Initialisierung des CAN-Controllers erfolgt durch den Aufruf des Unterprogramms CWRI_REG und CREAD_REG. Diese beiden Unterprogramme sind in dem File „CAN_RW.INC“ untergebracht. Diese beiden Funktionen sind auch noch mal als Macro im dem File „CAN.MAC“ implementiert. Als Macros heißen die beiden Funktionen WRI_REG und READ_REG. Der Grund für die unterschiedliche Implementierung der Zugriffsroutinen auf die Register des CAN-Controllers liegt in der unterschiedlichen Ausführungsgeschwindigkeit und dem notwendigen Speicherplatz für Macros und Unterprogramme begründet. Unterprogramme sind nur einmal im Programmspeicher vorhanden und es wird durch einen CALL-Befehl in diese Programmteile verzweigt. Für diese Verzweigung werden aber insgesamt vier Takte benötigt. Dadurch wird die Abarbeitung des Programms langsamer. Bei den Macros wird die entsprechende Code-Passage bei der Assemblierung direkt an die Stelle des Macro-Aufrufs kopiert. Es sind also keine Sprunganweisungen notwendig. Diese Code-Passage wird aber jedes Mal neu eingesetzt. D.h., daß der Aufruf einer Macro-Funktion Speicherplatz kostet. Aus diesem Grund werden in den Routinen, die schnell abgearbeitet werden sollen, die Macro-Funktionen zum Zugriff auf die Register des CAN-Controllers verwendet. Bei der Initialisierung, die nicht Geschwindigkeits optimiert ablaufen muß, werden stattdessen die Unterprogramme zum Zugriff verwendet, um Speicherplatz zu sparen.

Zu Beginn der Initialisierungsroutine wird der CAN-Controller hardwaremäßig resetet. Danach wird der Controller in den Initialisierungsmodus geschaltet. Dies erfolgt durch das Setzen des RES- & IM-Bits im Mode/Status-Register MOD.

Bevor mit der Konfiguration der Übertragungsgeschwindigkeit des CAN-Bus begonnen wird, werden das *Controlregister* (CTRL) und das Interruptregister (INT) in CAN-Controller auf 00h gesetzt.

Jetzt werden die Übertragungsgeschwindigkeit und die Bit-Timing Parameter konfiguriert. Die Übertragungsrate wird dabei nicht direkt angegeben, sondern berechnet sich aus dem eingestellten Baud Rate Prescalar und den Bit-Timing.
Die Abtastung jedes übertragenen Bits setzt sich, wie in Abbildung 29 dargestellt, aus drei Abtastphasen zusammen, der Synchronisationsphase (SYNC_SEG), der ersten Abtastphase (PHASE_SEG1) und der zweiten Abtastphase (PHASE_SEG2). Zwischen der ersten und zweiten Abtastphase wird das Bit gesampled, also abgetastet. Die Synchronisationsphase dauert eine Zeiteinheit T_q . Die Anzahl der möglichen T_q-Zyklen kann für die erste Abtastphase auf Werte zwischen 0 und 15 Zyklen eingestellt werden. Für die zweite Abtastphase beträgt der Einstellbereich 0 bis 7 Zyklen.

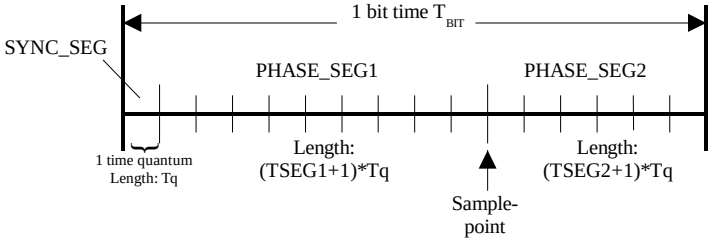


Abbildung 29: Abtastphasen des CAN-Bittimings

Die Dauer eines T_q-Zyklus wird durch die Taktfrequenz des CAN-Controllers und den Wert des Baud Rate Prescaler Registers festgelegt. Dabei gilt:

$$T_{CANClock} = 2 * T_{Osc} = \frac{2}{f_{Osc}} = \frac{2}{20\,MHz} = 100ns$$

$$T_q = (BRP + 1) * T_{CANClock}$$

Die Übertragungsrate wird durch die Dauer von T_q und die Summe der T_q-Zyklen festgesetzt. Dabei gilt:

$$T_{Bit} = T_{SYNCH\ SEG} + T_{PHASE\ SEG1} + T_{PHASE\ SEG2}$$

$$T_{SYNC\ SEG} = 1 * T_q$$

$$T_{PHASE\ SEG1} = (TSEG1 + 1) * T_q$$

$$T_{PHASE\ SEG2} = (TSEG2 + 1) * T_q$$

$$T_{SJW} = (SJW + 1) * T_q$$

Bei einer gewünschten Übertragungsrate von 250kBit/s gilt für $T_{Bit} = \frac{1}{baudrate} = 4\,\mu s$. Wird das Baud Rate Prescaler Register auf den Wert 1 gesetzt, ergibt sich für T_q=200ns. D.h., daß jedes Bit genau 20 T_q-Zyklen dauert. Die Abtastung des Bits erfolgt im allgemeinen nachdem etwa 60-70% der gesamten Bitdauer verstrichen ist. Bei 20 T_q -Zyklen für die gesamte Bitdauer sollte das Bit also im 14 T_q-Zyklus abgetastet werden. Da bereits ein T_q-Zyklus für die Synchronisation notwendig ist bleiben 13 T_q-Zyklen für die erste Abtast-Phase übrig. Damit folgt aus den oben angegebenen Formeln für TSEG1=12 und TSEG2=5.

Als letztes muß der Wert für die Maximum Synchronization Jump Width (SJW) angegeben werden. Für diesen Wert gilt: $0 \leqslant SJW \leqslant 3$ und $SJW \leqslant TSEG2$. Im allgemeinen ist SJW=1. Ein größer Wert für SJW ist nur notwendig, wenn keine stabile Taktfrequenz am CAN-Controller anliegt.

Wird die maximal mögliche Übertragungsrate von 1MBit/s gwünscht, gilt für $T_{Bit} = \frac{1}{baudrate} = 1\,\mu s$.

Wird das Baud Rate Prescaler Register auf den Wert 0 gesetzt, entspricht $T_q = T_{CANClock} = 100ns$. D.h., daß jedes Bit genau 10 T_q-Zyklen dauert. Soll das Bit abgetastet werden nachdem bereits 70% der gesamten Bitdauer verstrichen sind, so muß die Abtastung im 7 T_q-Zyklus erfolgen. Nach Abzug eines T_q-Zyklus zur Synchronisation bleiben 6 T_q-Zyklen für die erste Abtast-Phase übrig. Damit folgt für TSEG1=5 und TSEG2=2. Der Wert für die Maximum Synchronization Jump Width (SJW) wird wiederum auf einen Wert gleich 1 gesetzt. Außer den Timing Parametern ergibt sich keine Änderung bei der Initialisierung für diese Übertragungsgeschwindigkeit.
Hier die Konfiguration für die Übertragungsraten von 250kBit/sec und 1MBit/Sec nochmal zusammengefaßt:

	250kbit/sec	***1MBit/sec***
Baud Rate Prescaler	1	0
TSEG1	12	5
TSEG2	5	2
SJW	1	1

Tabelle 8: Konfigurationsdaten für das CAN-Bittiming

Die berechneten Werte für eine der Übertragungsgeschwindigkeiten werden dann in das *Baud Rate Prescaler Register (BRPR)* und das *Bit-Length Register 1 und 2 (BL1 und BL2)* geschrieben. Damit ist die Konfiguration der Übertragungsgeschwindigkeit abgeschlossen.
Als nächstes wird mit der Konfiguration des Interrupt-Mask *Registers* fortgefahren. In diesem Register können Interrupts, die beim Auftreten von verschiedenen Betriebs- und Fehlerzuständen ausgelöst werden, freigegeben werden. Tritt im CAN-Controller bei einer der freigegeben Interruptquellen ein Ereignis ein, dann wird der PIC über einen „Low“-Pegel am I/O-PIN RA4 davon informiert. Der PIC ließt daraufhin das INT-Register im CAN-Controller aus in dem die Interrupt-Quelle gespeichert ist und dekodiert den entsprechenden Interrupt. In der Initialisierungsroutine werden der Receive Interrupt, der Warning Level, Error Passive und der Bus Off Interrupt im CAN-Controller freigegeben. (Einzelheiten zu den Fehlermodi siehe Kap. 2.3.4)
Weil der CAN-Controller SAE80C90 keine physikalische CAN-Schnittstelle auf dem Chip besitzt und eine Vielzahl verschiedener Bustreiber dafür in Frage kommen, stehen verschiedene Konfigurationsmöglichkeiten für die Ausgangslogik zur Verfügung. Mit dem Output Control Register wird diese Verbindung mit dem physikalischen Übertragungslayer konfiguriert. Dabei kann festgelegt werden, wie ein Bit-Wert auf dem physikalischen Layer übertragen wird. Es kann sowohl die Polarität als auch die Dominanz des Bits in diesem Register programmiert werden. Da der im Kommunikationsinterface verwendete CAN-Bustreiber zum CAN-Controller hin TTL-kompatible Signale verwendet, wird die Ausgangslogik des Controller entsprechend konfiguriert. Einzelheiten zu der Konfiguration der Ausgangslogik können [Sie1] entnommen werden.

Im nächsten Schritt müssen die Message-Boxen konfiguriert werden. Die Konfigurationsdaten für die Messageboxen werden in dem File „CAN_BOX.INC“ zusammengefaßt. Weil in dem PIC-Mikrocontroller insgesamt nur 192 RAM-Speicherzellen vorhanden sind, werden die Konfigurationsdaten in diesem Include-File mit dem DEFINE-Befehl initialisiert. D.h. der Variablenname dient im Sourcecode nur als Platzhalter für den eigentlichen Zahlenwert, der bei der Assemblierung direkt in das Programm eingefügt wird. Dadurch ist es möglich, die Konfigurationsdaten übersichtlich in einem File zusammenzufassen ohne das Speicherzellen in dem PIC benötigt werden.

Die Konfigurationsdaten jeder Messagebox sind in den vier Definitionen MBxx_DLC, MBxx_RTR, MBxx_IDH und MBxx_IDL untergebracht. Die Definition MBxx_DLC enthält die Länge der Nachrichtentelegramme für die Messagebox xx. Um zum Übertragungsprotokoll zur SSI-Schnittstelle kompatibel zu bleiben, sind bei dieser „Variable“ die Werte 1,2,4 & 8 zugelassen. Mit der Definition MBxx_RTR wird festgelegt, ob die Daten in dieser Messagebox als „normale“ oder „Remote-Frames“ übertragen werden. Der Wert „00“ kennzeichnet eine „normale“ Datenübertragung, während bei einem Wert von „10“ die Daten erst bei dem Empfang eines Remote-Requests mit dem entsprechenden Identifier gesendet werden. Der Identifier der Messagebox wird in den beiden Definitionen MBxx_IDH und MBxx_IDL gespeichert. In der Definition MBxx_IDH werden die oberen 8 Bit des insgesamt 11-Bit langen Identifiers gespeichert. Die unteren drei Bit des Identifiers sind in der Definition MBxx_IDL gespeichert. Prinzipiell kann für den Identifier jeder beliebige 11-Bit lange Wert benutzt werden. Allerdings ist es auf Grund eines Fehlers in dem CAN-Controller SAE80C90 Step D13 ratsam Werte für den Identifier zu benutzen, die kleiner als 0x400h sind.(Siehe dazu auch [Sie2] Unterpunkt.9)

Freigegeben werden die Messageboxen mit der Definition RIMRx_ENABLE. Durch den definierten Wert werden die Receiver-Interrupts der einzelnen Messageboxen freigeschaltet. D.h. erst wenn der Receiver-Interrupt einer Messagebox freigeschaltet ist, wird eine in der Messagebox empfangene Nachricht verarbeitet. Mit der Definition RIMR1_ENABLE werden die Messageboxen 1 bis 7 freigeschaltet. Die Messagebox 7 wird dabei durch das MSB und die Messagebox 0 durch das LSB repräsentiert. Der Interrupt einer Messagebox wird durch das Setzen des entsprechenden Bits freigeschaltet. Die Receiver-Interrupts für die Messageboxen 8 bis 15 wird durch die Definition RIMR2_Enable freigeschaltet.

Eine weitere Definition, die das File „CAN_BOX.INC“ enthält, ist DATAx_Frame. Mit dieser Definition wird gekennzeichnet, ob es sich bei den Messageboxen um „normale“ oder Remote-Request“-Boxen handelt. Diese Definition wird nur für die CAN-Senderoutinen benutzt. „Normale“ Messageboxen werden durch das Setzen des entsprechenden Bits in der DATAx_Frame Definition gekennzeichnet.

Die Übertragung der Messagebox-Konfigurationsdaten in den CAN-Controller erfolgt durch den Aufruf des „SetDR“-Macros, der in dem File „CAN.MAC“ definiert ist. Der Macro wird jeweils für jede Messagebox einmal ausgeführt. Dabei werden dem Macro die Definition der Registeradressen der Konfigurationsregister im CAN-Controller und die Konfigurationsdaten für die Messagebox übergeben.

Nachdem die Konfigurationsdaten für jede Messagebox in den CAN-Controller übertragen wurden, werden die Receiver-Interrupts für die einzelnen Messageboxen im CAN-Controller freigeschaltet. Dazu werden die RIMRx_ENABLE Definitionen in die RIMRx-Register im CAN-Controller kopiert.

Zum Abschluß der Initialisierung wird das IM- und das RES-Bit in MOD-Register zurückgesetzt. Damit ist die eigentliche Initialisierung des CAN-Controllers abgeschlossen und der Controller ist betriebsbereit.

Am Ende der Initialisierungsroutine werden aber noch zwei Tabellen im RAM-Speicher des PIC's angelegt. In der ersten Tabelle werden die Nachrichtenlängen der einzelnen Messageboxen gespeichert. Mit Hilfe dieser Routine wird in der CAN-Empfangsroutine die Nachrichtenlänge bestimmt. Die zweite Tabelle enthält die Adressen des ersten Byte jeder Messagebox. Mit Hilfe dieser Tabelle werden die Adressen zum Schreiben und Lesen der Messagebox-Daten berechnet.

6.1.5 Die Funktionen zur Kommunikation über die asynchrone Schnittstelle

Die Funktionen zum Senden und zum Empfang über die USART des PICs sind in dem Macro-File „RS232.MAC“ definiert. Der Macro zum Senden von einem Datentelegramm heißt „RS232_SAVE“. Der Empfang von Datentelegrammen erfolgt durch den Macro „RS232_LOAD“.

In Abbildung 30 ist das Flußdiagramm des Macro „RS232_SAVE“ dargestellt. Nachdem dieser Macro aufgerufen wurde wird als erstes das Befehlsbyte gesendet. Dies erfolgt indem der Wert der Variablen H_Control, in der das Befehlsbyte gespeichert ist, in das Senderegister TXREG der USART geschrieben wird. Nun wartet das Programm bis das TXIF-Bit der USART den Wert 1 angenommen hat. Mit diesem Bit signalisiert die USART, daß das Byte vollständig gesendet wurde.

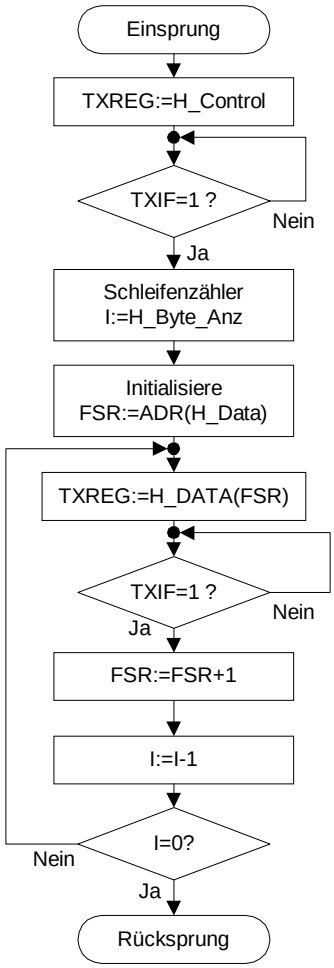


Abbildung 30: Flußdiagramm des Macro "RS232_SAVE"

Jetzt werden die Vorbereitungen zum Senden der Datenbytes getroffen. Dabei wird der Schleifenzähler I mit der Anzahl der zu sendenen Datenbytes initialisiert. Außerdem wird in das Register FSR, das den Adresszeiger für den indirekten Zugriff auf den RAM-Speicher enthält, mit der Adresse des ersten Datenbyte „H_Data“ initialisiert.

Das Senden der Datenbyte erfolgt in einer Schleife. Am Anfang dieser Schleife wird ein Datenbyte in das TXREG der USART kopiert. Nachdem gewartet wurde bis das Byte vollständig gesendet wurde, wird der Adresszeiger auf die Adresse des nächsten Datenbyte gesetzt und der Schleifenzähler „I“ um eins verringert. Am Ende der Schleife wird überprüft, ob der Schleifenzähler=0 ist. Ist dies nicht der Fall wird das nächste Datenbyte gesendet. Ist der Schleifenzähler „I“=0 wird aus dem Macro wieder zurück in die Serviceroutine gesprungen.

In Abbildung 31 ist das Flußdiagramm des „RS232_LOAD“-Marcos dargestellt. Dieser Marco wird von der Serviceroutine erst dann aufgerufen, wenn bereits das erste Byte eines Telegramms, also das Befehlsbyte, von der USART des PICs empfangen wurde. Aus diesem Grund wird direkt nach dem Einsprung in den Macro das Empfangsregister des USART ausgelesen und der Inhalt in der Variablen H_Control gespeichert. Nachdem dies erfolgt ist, wird das Befehlsbyte als Echo zurück an das Terminal gesendet. Das Terminalprogramm benutzt das Echo um zu kontrollieren, ob das gesendete Byte richtig empfangen wurde. Wie bereits in dem „RS232_SAVE“-Macro wird dazu erst das Byte in das Senderegister TXREG kopiert und dann gewartet bis die Übertragung beendet ist.

In der Firmware der „Spezialversion“ des RS232 auf CAN-Interface wird danach kontrolliert, ob es sich bei dem empfangenen Befehlsbyte um einen Remote Request handelt. Dazu wird das 6 Bit des Befehlsbit ausgewertet. Ist dieses Bit gesetzt handelt es sich um Remote Request, d. h., es werden keine weiteren Datenbytes in diesem Telegramm übertragen. Deshalb verzweigt das Programm zum Ende des „RS232_LOAD“-Macros.

In der Firmware des RS232/CAN auf SSI-Interfaces ist diese Abfrage im „RS232-Load“-Macro nicht implementiert, weil die Anforderung von Remote Requests in dieser Programmversion über die SSI-Schnittstelle ausgelöst wird. Ansonsten sind die Macros der beiden Firmwareversionen identisch.

Bevor die Datenbytes des Nachrichtentelegramms empfangen werden können, muß die Länge der Nachricht bestimmt werden. Bei der Entschlüsselung werden die Bits 4 & 5 ausmaskiert und die oberen 4 Bit mit dem unteren 4 Bit getauscht. Damit steht jetzt ein Wert zwischen 0 und 3 in diesem Byte. Dieser Wert wird in die „wirkliche“ Telegrammlänge umgerechnet und in der Variablen „H_Byte_Anz“ gespeichert.

Das Einlesen der Datenbytes erfolgt durch eine Schleife. Vor der Abarbeitung der Empfangsschleife wird der Schleifenzähler I mit der Anzahl der Nachrichtenbyte initialisiert. Außerdem wird, wie bereits in dem „RS232_SAVE“-Macro, das FSR-Register mit der Adresse des ersten Datenbyte im RAM initialisiert.

Am Anfang der Empfangsschleife wartet der Controller bis das RXIF-Bit gesetzt ist, also ein Byte vollständig empfangen wurde. Das empfangene Byte wird an die Adresse im RAM geschrieben, auf die das FSR-Register zeigt. Danach wird ein Echo des empfangenen Bytes gesendet. Wurde das Echo vollständig gesendet, wird der Zeiger im FSR-Register auf die Adresse des nächsten Datenbytes gesetzt und der Schleifenzähler um eins verringert. Am Ende der Schleife wird überprüft, ob der Schleifenzähler I=0 ist, also ob alle Datenbyte empfangen wurden. Ist dies der Fall erfolgt der Rücksprung ins Hauptprogramm. Ist der Wert des Schleifenzähler ungleich Null, wird die Schleife noch einmal durchlaufen.

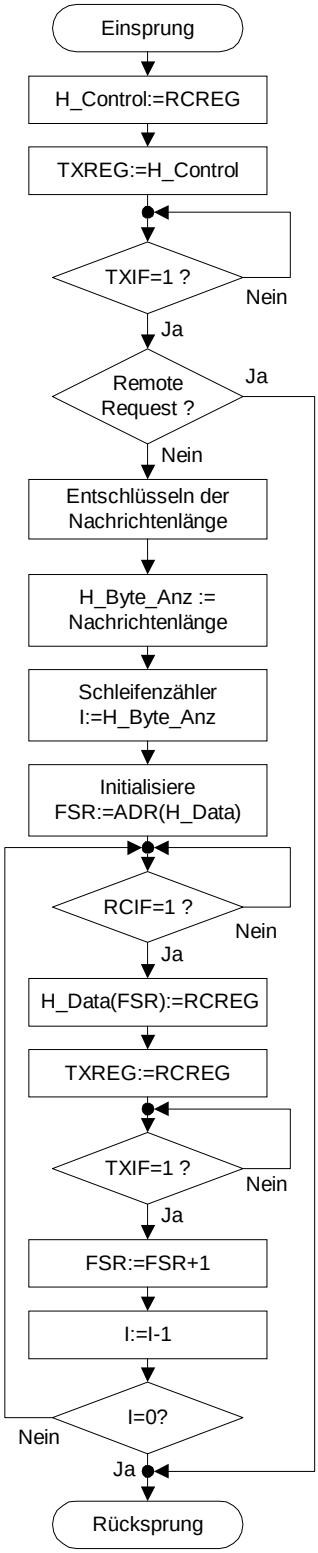


Abbildung 31: Flußdiagramm des "RS232_LOAD"-Macros

6.1.6 Die Funktionen zur Kommunikation über die synchrone Schnittstelle

Für die Kommunikation über die SSI-Schnittstelle stehen zwei Funktionen zur Verfügung. Mit dem Macro „SPI_LOAD“ erfolgt der Empfang der Datentelegramme. Zum Senden der Telegramme über die SSI-Schnittstelle steht der „SPI_SAVE“-Macro zur Verfügung. Diese beiden Macros sind in dem File „SPI.MAC“ zusammengefaßt.

Der Aufbau des „SPI_SAVE“-Macros, der in Abbildung 32 dargestellt ist, unterscheidet sich kaum vom Aufbau des „RS232_SAVE“-Marcos. Allerdings ergeben sich durch die Eigenschaften der SSI-Schnittstelle einige Besonderheiten.

Nach dem Einsprung in den Macro wird das Befehlsbyte in das Senderegister der SSI-Schnittstelle (SSPBUF) kopiert. Weil die SSI-Schnittstelle als Slave konfiguriert ist, muß der VeCon-Prozessor die Übertragung der Daten einleiten. Deshalb wird der VeCon durch das Setzen des Received-Signals aufgefordert die Übertragung zu starten. Durch die Abfrage des SSPIF-Bit ist sichergestellt, daß das Programm erst dann weiter abgearbeitet wird, wenn das Byte gesendet wurde.

Nun wird das Received-Signal wieder gelöscht und die Schleife, in der die Datenbyte gesendet werden, initialisiert. In der Schleife wird bei jedem Durchlauf, wie in dem „RS232_SAVE“-Macro, ein Datenbyte übertragen. Ist die Übertragung abgeschlossen wird wieder zurück ins Hauptprogramm gesprungen.

Der Empfang der Telegramme über die SSI-Schnittstelle erfolgt mit Hilfe des „SPI_LOAD“-Macros, dessen Flußdiagramm in der Abbildung 33 dargestellt ist. Dieser Macro ist von der Funktion und dem Aufbau her dem „RS232_LOAD“_Macro sehr ähnlich.

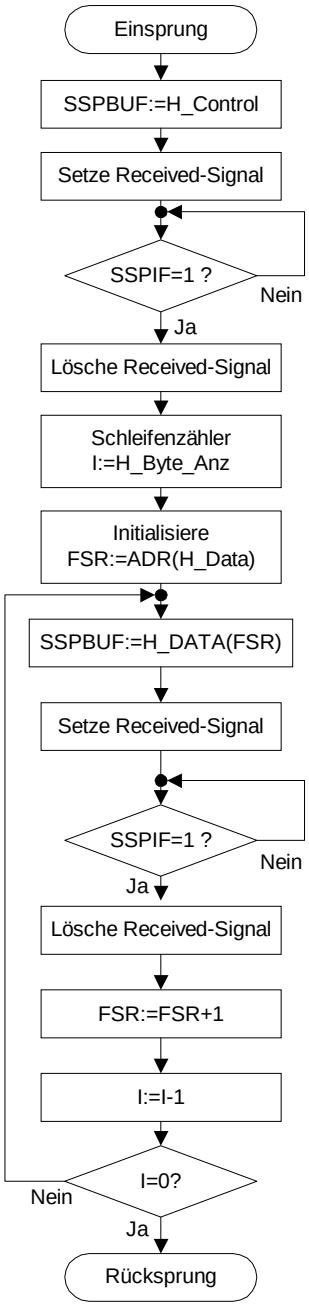


Abbildung 32: Flußdiagramm des "SPI_SAVE" Macros

Wie bei dem „RS232_LOAD“-Macro wird auch beim „SPI_LOAD“-Macro direkt nach dem Aufruf das im Empfangsregister der SSI-Schnittstelle (SSPBUF) empfangene Befehlsbyte in der Variable H_Control gespeichert.

Nachdem dies erfolgt ist, wird das BF-Bit (Buffer Full), mit dem das SSI-Interface signalisiert das ein Byte vollständig empfangen wurde, gelöscht. Laut Handbuch des PICs sollte das Löschen des BF-Bits eigentlich von der Hardware des PIC [Mic1] durchgeführt werden, nachdem ein Lesezugriff auf das SSPBUF-Register erfolgt ist. In der in dieser Studienarbeit verwendeten Chip-Revision arbeitet diese Funktion aber nicht wie im Handbuch beschrieben.

Im nächsten Verarbeitungsschritt wird dann überprüft, ob es sich um einen Remote Request handelt. Ist dies der Fall folgen keine weiteren Datenbyte und es wird an das Ende des Macros gesprungen. Vor dort aus erfolgt dann der Rücksprung ins Hauptprogramm.

Handelt es sich um ein normales Datentelegramm wird die Länge des Datentelegramms aus dem Befehlsbyte berechnet und der Schleifenzähler für die Empfangsschleife initialisiert. Der Empfang der Datenbytes erfolgt dann wie in dem „RS232_LOAD“-Macro in einer Schleife.Ist das Telegramm vollständig empfangen worden, erfolgt der Rücksprung ins Hauptprogramm.

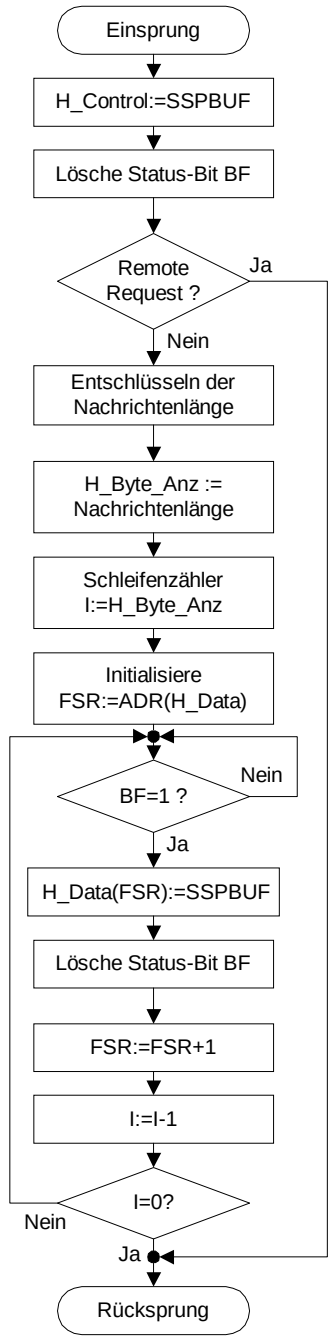


Abbildung 33: Flußdiagramm des "SPI_LOAD"-Macro

6.1.7 Die Funktionen zur Kommunikation über den CAN-Bus

Die Kommunikationsroutinen für den Zugriff auf den CAN-Bus sind in dem File „CAN_COM.INC“ zusammengefaßt. Dieses File stellt die Unterprogramme „CAN_PUT“ zum Senden und „CAN_GET“ für den Empfang von Nachrichten über den CAN-Bus zur Verfügung.

Das Flußdiagramm des Unterprogramms „CAN_PUT“ ist in der Abbildung 34 dargestellt.

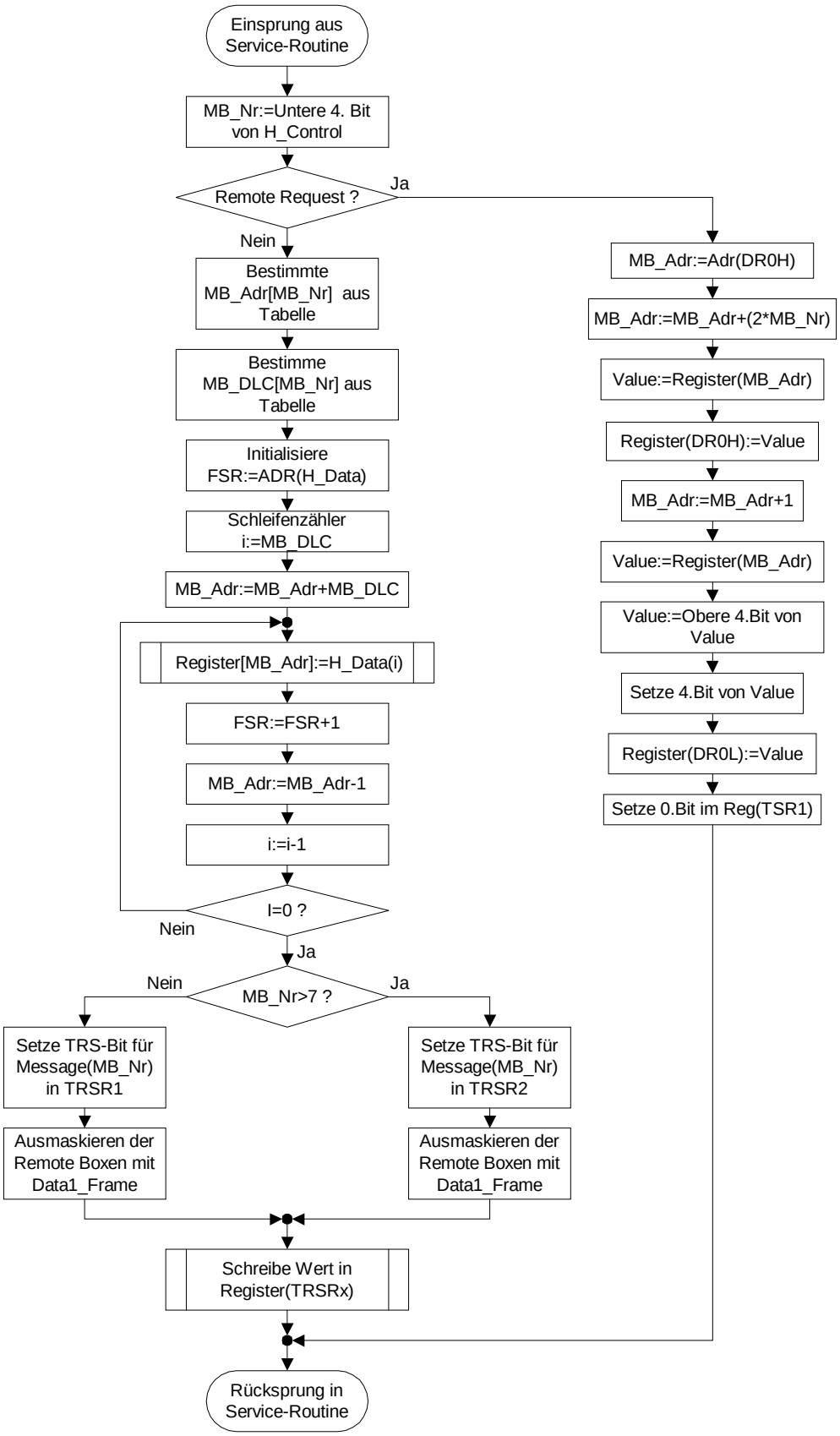


Abbildung 34: Flußdiagramm des Unterprogramms "CAN_PUT"

Nachdem das „CAN_PUT“ Unterprogramm mit einem CALL-Befehl aus der Service-Routine heraus aufgerufen wurde wird die MessageBox-Nummer, über die die Daten gesendet werden sollen, aus dem Befehlsbyte ermittelt. Dazu werden die oberen 4 Bit des Befehlsbyte ausmaskiert und das Ergebnis dieser Operation in der Variable MB_Nr gespeichert.

Als nächstes wird überprüft, ob es sich bei der Übertragung um das Senden eines normale Telegramms oder eines Remote Requests handelt. Dazu wird das Bit 6 des Befehlsbytes ausgewertet. Wenn es sich um einen Remote Request handelt wird in eine spezielle Routine verzweigt, die einen Remote Request mit dem durch die Messagebox Nummer definierten Identifier über die Messagebox 0 sendet.

Dies erfolgt in dem die beiden Identifier Descriptor Register der angesprochenen Messagebox in die Identifier Descriptor Register der Messagebox 0 kopiert werden. Dazu wird die Adresse des High Byte Descriptor Register der Messagebox 0 (DR0H) als Basisadresse geladen und die Registeradresse des HighByte Descriptor (DRxH) der durch die Messagebox Nummer definierten Messagebox berechnet. In diesem Register befinden sich die oberen 8 Bit des Identifiers. Der Inhalt des DRxH- Registers wird in das DR0H-Register der Messagebox 0 kopiert. Dann wird die Registeradresse um eins erhöht, wodurch der Adresszeiger auf das Low Byte Descriptor Register (DRxL) der im Befehlsbyte definierten Messagebox zeigt. In dem DRxL-Register werden die unteren 3 Bit des Identifiers, das Remote Request Bit und die Länge des Datentelegrammes definiert. Der Inhalt dieses Registers wird gelesen und die oberen 4 Bit in der Variablen Value zwischengespeichert. In diesen 4 Bit sind die unteren 3 Bit des Identifiers und das RTR-Bit gespeichert. Dann wird das RTR-Bit gesetzt und der Inhalt der Variable Value in das DR0L-Register der Messagebox 0 geschrieben. Nun wird das TRS-Bit für die Messagebox 0 gesetzt, wodurch das Senden des Remote Requests ausgelöst wird.

Durch diese Vorgehensweise können zwar nur 15 Nachrichtenboxen zur Übertragung von „normalen“ Telegrammen benutzt werden, aber dafür ist sichergestellt, daß die durch den Remote Request angeforderten Daten wirklich empfangen werden. Denn Remote Messageboxen können keine Datentelegramme empfangen. D.h., daß die Messagebox direkt nach dem Senden eines Remote Requests umkonfiguriert werden müßte, wenn das angeforderte Datentelegramm in der gleichen Messagebox empfangen werden soll. Wenn der andere Teilnehmer bereits während der Umkonfiguration beginnt das angeforderte Datentelegramm zu senden, würde das Telegramm verloren gehen.

Ergibt die Überprüfung des Remote Request Bits im Befehlsbyte, daß ein normales Datentelegramm gesendet werden soll, wird in eine Routine verzweigt, die die Datenbytes vom RAM in die Register der Messagebox kopiert. Im Anschluß daran wird, wenn es sich um eine „normale“ Messagebox handelt, das Transmission Request Set- Bit (TRSxx_Bit) der entsprechenden Messagebox gesetzt, wodurch die Übertragung der Daten gestartet wird.

Bevor die Datenbytes in die Register der gewählten Messagebox kopiert werden können, muß die Basisadresse des Messagebox-Registers berechnet werden. Dazu wird die Basisadresse des 0-ten Bytes der Messagebox mit Hilfe der Messagebox-Nummer aus einer Tabelle bestimmt. Die Zugriffe auf die Messagebox-Register erfolgen immer über sogenannte Shadow-Register, deren Inhalt in die Messagebox-Register kopiert wird, wenn ein Zugriff auf das 0.Byte der Messagebox erfolgt. Aus diesem Grund muß das Schreiben der Daten in die Messagebox-Register immer mit dem MSB beginnen. Deshalb wird zu der ermittelten Registeradresse noch ein Offset-Wert, der aus der Nachrichtenlänge gebildet wird, hinzu addiert.

Im nächsten Schritt wird der Adresszeiger im FSR-Register auf das erste Datenbyte (MSB) im RAM gesetzt und der Schleifenzähler initialisiert. Nun werden die Datenbytes in einer Schleife nacheinander in den CAN-Controller kopiert. Nachdem das Datenbyte aus dem RAM des PIC mit dem „WRITE_REG“-Macro in das Messagebox-Register des CAN-

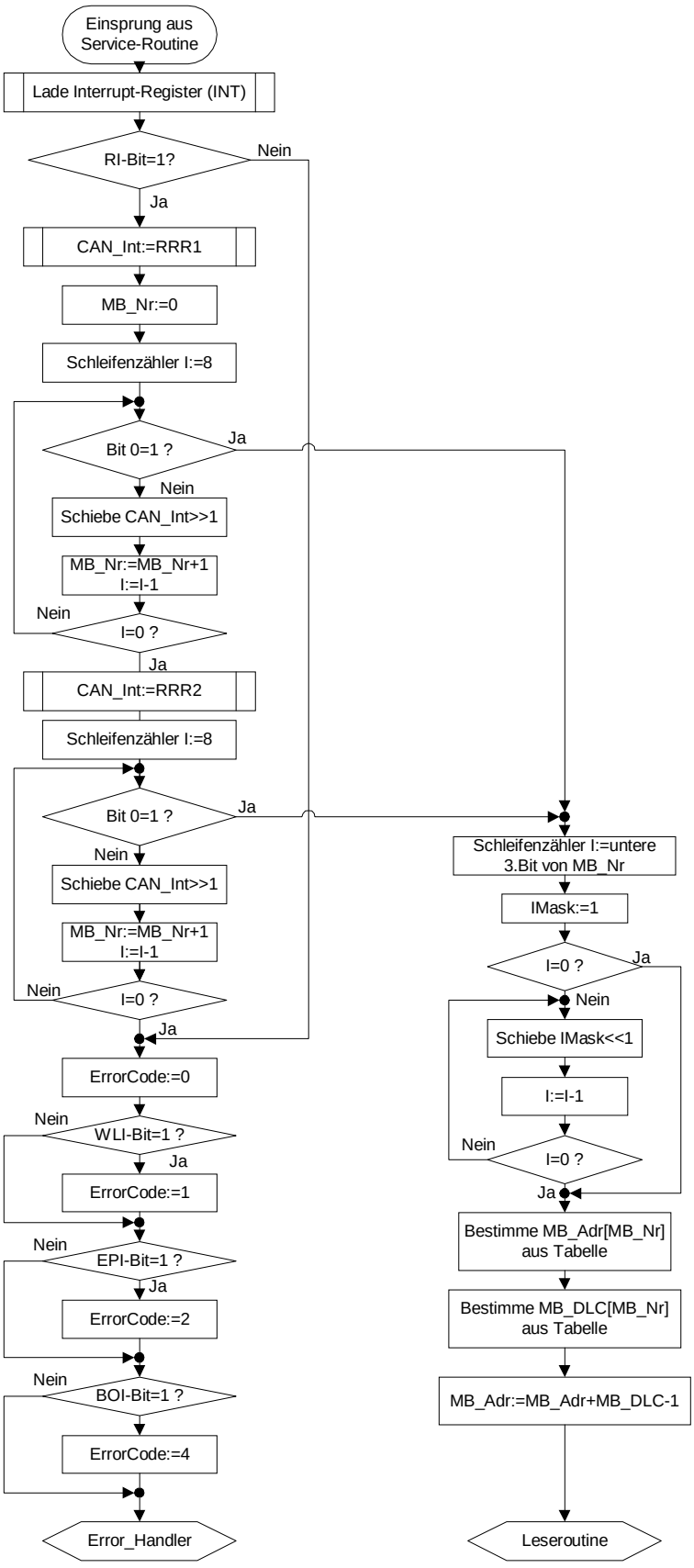
Controllers kopiert wurde, wird der Adresszeiger im FSR-Register auf die nächste Speicherzelle weitergestellt und die Registeradresse der Messagebox um eins verringert. Am Ende der Schleife wird der Schleifenzähler I um eins verringert und überprüft, ob der Schleifenzähler gleich null ist. Wenn der Wert des Schleifenzählers größer als Null ist wird das nächste Byte übertragen.

Ist der Schleifenzähler gleich null sind alle Datenbytes in die Messagebox-Register des CAN-Controller kopiert worden. Nun muß das entsprechende Transmission Request Set-Bit (TRS-Bits) gesetzt werden, um die Nachricht zu senden. Die TRS-Bits für die Messageboxen 0 bis 7 sind in dem TRSR1-Register und die für die Messageboxen 8 bis 15 im dem TRSR2-Register untergebracht. Wenn die Messagebox-Nummer nicht größer als 7 ist, wird die Adresse des TRSR1-Registers geladen und das Bit der entsprechenden Messagebox in der Variablen Value zwischengespeichert. Da bei Remote Boxen der Inhalt der Messagebox erst nach dem Empfang eines Remote Requests gesendet werden soll, darf das TRS-Bit nicht bei Remote Boxen gesetzt werden. Dies wird verhindert in dem gesetzte TRS-Bits bei Remote Box ausmaskiert werden bevor der Inhalt der Variablen Value in das entsprechende TRSR-Register kopiert wird. Dazu wird der Inhalt der Variablen Value über eine Und-Operation mit der Definition Data1_Frame verknüpft. Ist die Messagebox-Nummer größer als 7 wird das gleiche Verfahren mit dem TRSR2-Register und der Data2_Frame durchgeführt. Die Definition von DataX_Frame erfolgt in dem Messagebox-Konfigurationsfile „CAN_BOX.INC“. [Siehe dazu auch Kap. 5.1.4].

Am Ende des Unterprogramms wird der Wert der Variablen Value an die vorher bestimmte Adresse im CAN-Controller geschrieben. Damit ist das TRS-Bit gesetzt und der Controller beginnt mit der Übertragung der Daten.

Für das Auslesen von Nachrichten aus dem CAN-Controller wird das „CAN_GET“ Unterprogramm benutzt. Das Flußdiagramm dieses Unterprogramms ist in Abbildung 35 und Abbildung 36 dargestellt.

In diesem Unterprogramm werden alle Meldungen, die vom CAN-Controller an den PIC übertragen werden sollen verarbeitet. D.h. in dieser Routine wird sowohl der Empfang von Telegrammen als auch die Verarbeitung und Auswertung von Fehlermeldungen durchgeführt. Aus diesem Grund wird als erstes, nachdem in dieses Unterprogramm verzweigt wurde, das Interrupt-Register des CAN-Controllers (INT) eingelesen und ausgewertet. Die Auswertung erfolgt in zwei Teilen. Zuerst wird überprüft, ob es das Receiver Interrupt-Bit (RI-Bit) in dem INT-Register gesetzt ist. Ist das RI-Bit nicht gesetzt, wurde keine Nachricht empfangen, sondern der Interrupt durch eine Fehlermeldung des CAN-Controllers ausgelöst. Aus diesem Grund wird für den Fall das das RI-Bit nicht gesetzt ist zur Fehlerverarbeitung verzweigt.



Fortsetzung siehe nächste Seite

Abbildung 35: Flußdiagramm des Unterprogramms "CAN_GET" (Teil A)

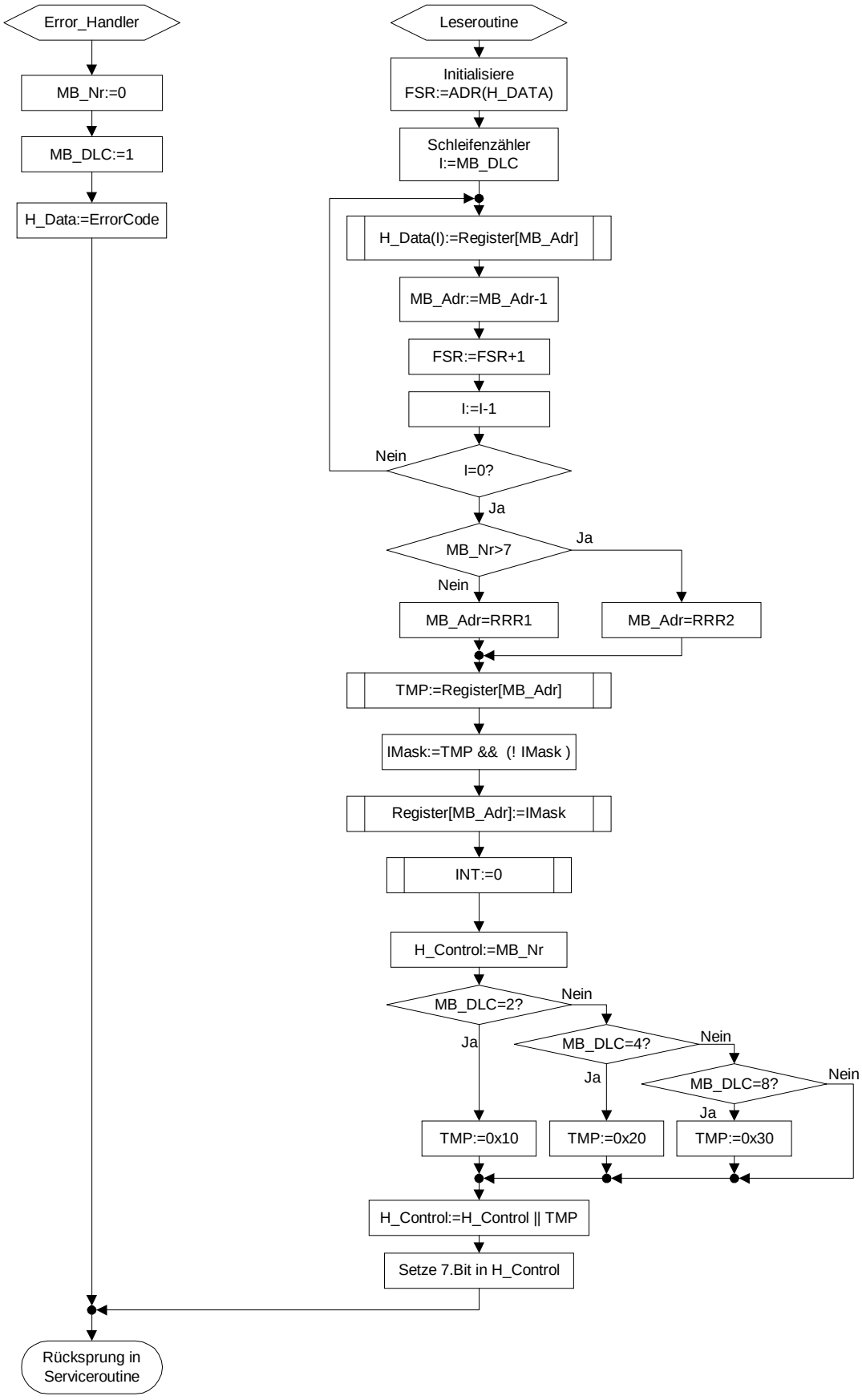


Abbildung 36: Flußdiagramm des Unterprogramms "CAN_GET" (Teil B)

In der Fehlerverarbeitungsroutine werden nacheinander die Interrupt-Bits der verschiedenen Fehlerzustände abgefragt und wenn das entsprechende Bit gleich eins ist der ErrorCode gesetzt. Der CAN-Controller besitzt drei Fehlerinterrupts, die sich in der Schwere der Fehler unterscheiden. Die erste Fehlerstufe ist der Warning Level Interrupt (WLI). Die zweite Stufe ist der Error Passive Interrupt (EPI). Der schwerste Fehler ist der Bus Off-Interrupt (BOI) nach dessen auftreten überhaupt keine Kommunikation über den CAN-Bus mehr möglich ist. Wenn mehrere Fehlerinterrupts bei der Abarbeitung der Fehlerbehandlungsroutine gesetzt sind, dann enthält der ErrorCode den Wert des schwersten Fehlers. Dieses Verhalten wird erreicht, in dem die Interruptbits nacheinander gemäß der schwere der Fehler bearbeitet werden und der ErrorCode bei einem gesetzten Interruptbit mit den neuen ErrorCode überschrieben wird.

Fehlercode	***Name***	***Funktion***
0	Undefinierter Fehler	Undefiniert
1	Warning Level Interrupt (WLI)	Warnung, daß Fehlerzähler über 96 Fehler registiert hat
2	Error Passiv Interrupt (EPI)	Fehlerzähler über 126 -> CAN-Controller verhält sich ErrorPassiv
4	Bus Off Interrupt (BOI)	Fehlerzähler über 255 -> CAN-Controller im BusOff-Mode

Tabelle 9: Übersicht der Fehlercodes

In der Tabelle 9 sind die ErrorCodes und ihre Bedeutung noch einmal zusammengefaßt. Einzelheiten zu den Fehlermeldungen des CAN-Controllers können den Kapitel 2.3.4 und 4. entnommen werden.

Nachdem der ErrorCode bestimmt wurde, werden die Informationen zur Übertragung der Fehlermeldung initialisiert. Dazu wird die Messagebox Nummer auf 0 und die Nachrichtenlänge für die nachfolgende Übertragung auf 1 gesetzt. Der ErrorCode selbst wird den nachfolgenden Routinen in der Variable H_Data übergeben. Damit sind alle Variablen initialisiert und es erfolgt der Rücksprung in die Serviceroutine. Von dort aus wird dann bei dem RS232/CAN auf SSI-Interface die „SPI_SAVE“ Routine aufgerufen, die die Fehlermeldung weiterleitet.

Wurde bei der Überprüfung des RI-Bits festgestellt, daß dieses Bit gesetzt ist, dann erfolgt die Auswertung in welcher Messagebox eine Nachricht empfangen wurde. Dazu wird zuerst das Receiver Ready Register 1 (RRR1), das die Receive Ready Bit für die Messageboxen 0 bis 7 enthält, eingelesen und in der Variablen „CAN_Int“ gespeichert. In einer Schleife wird kontrolliert, ob ein Receive Ready Bit in der Variablen CAN_Int gesetzt ist. In dieser Auswertungs-Schleife wird jeweils überprüft, ob das 0.Bit gleich eins ist. Wenn das Bit gesetzt ist wird in die Einleseroutine verzweigt. Ist das Bit nicht gesetzt, dann wird der Inhalt der Variablen CAN_Int um ein Bit nach rechts geschoben und dann die Messagebox Nr. um eins erhöht. Außerdem wird der Schleifenzähler I um eins verringert. Ist der Schleifenzähler nicht gleich Null wird das nächste Bit ausgewertet. Ist der Schleifenzähler Null heißt das, daß in den Messageboxen 0 bis 7 keine Nachricht empfangen wurde und es wird mit der Auswertung der Receive Bits der Messagboxen 9 bis 15 begonnen. Dazu wird das Receive Ready Register 2 (RRR2) in der Variablen CAN_Int zwischengespeichert. Die Auswertung der gesetzten Receive Bit erfolgt wieder in einer Schleife. Die Funktion dieser Auswertungsschleife unterscheidet sich nicht, von der oben bereits beschriebenen. Ist in diesem Register ebenfalls kein Receive Ready Bit gesetzt durchläuft das Programm die Fehlerauswertung. So ist sichergestellt, daß die „CAN_GET“ Routine auch bei einer Fehlfunktion ein definiertes Ergebnis liefert mit dem das Programm weiterarbeiten kann. Im Allgemeinen würde in diesem Fall eine Fehlermeldung mit dem ErrorCode Null gesendet werden.

Auf Grund der sequentiellen Abarbeitung der Receive Ready Bits für die 16 Messageboxen ergibt sich eine Unterschiedliche Abarbeitungspriorität bei den Messageboxen. Je kleiner die Messagebox Nr. ist, desto großer ist die Priorität mit der die Nachricht abgearbeitet wird. D.h. wenn mehrere Receive Ready Bits gesetzt wird wird immer erst

die Nachricht mit der kleinsten Messagebox Nr. ausgelesen. Die anderen Nachrichten werden dann erst beim nächsten Durchlauf der Polling-Schleife abgearbeitet. Dieses Verfahren ist deshalb möglich, weil das RI-Bit im INT-Register und damit der INT-Ausgang des CAN-Controllers erst nachdem alle RRR-Bit gelöscht wurden zurückgesetzt werden kann.

Wurde ein Receive Ready Bit, das auf eins gesetzt ist, gefunden wird in die Leseroutine verzweigt. Am Anfang der Leseroutine wird die Interruptmaske erzeugt, mit der das Receive Ready Bit der entsprechenden Messagebox am Ende der Routine zurückgesetzt wird. Dazu wird der Schleifenzähler I mit den unteren 3 Bit der Messagebox Nr. (MB_Nr) initialisiert. Die Variable IMask, in der die Interrupt-Maske gespeichert wird, wird gleich eins gesetzt. Wenn der Schleifenzähler gleich null ist wird die nachfolgende Schleife übersprungen. In dieser Schleife wird der Wert des IMask-Registers I-mal nach links geschoben.

Im nächsten Schritt wird die Registeradresse des Byte 0 der Messagebox Nr., die in der Variablen MB_Nr definiert ist, mit Hilfe der Adresstabelle bestimmt. Mit dem gleichen Verfahren wird die Nachrichtenlänge der in MB_Nur definierten Messagebox ermittelt. Das Auslesen der Messagebox aus dem CAN-Controller beginnt mit dem „most significant Byte“ (MSB), weil so sichergestellt ist, daß die ausgelesenen Bytes zu einer Nachricht gehören. Das Auslesen der Nachrichten erfolgt, wie bei Schreibzugriff, über ein sogenanntes Shadow-Register, in dem der Inhalt der entsprechenden Messagebox zwischengespeichert wird. Die Übernahme der Messageboxregister in die Shadowregister erfolgt durch einen Zugriff auf das MSB. Zur Berechnung der Registeradresse des MSB im CAN-Controller wird die Messageboxlänge (MB_DLC-1) zu der bereits ermittelten Registeradresse des 0-ten Bytes addiert.

Im nächsten Schritt wird der Adresszeiger im FSR Register des PIC auf die Adresse des H_Data-Bytes gesetzt und der Schleifenzähler I mit der Messagelänge (MB_DLC) initialisiert. Nun beginnt die Ausleseschleife. In dieser Schleife wird der Inhalt des Registers mit der Adresse[MB_Adr] in den Speicher auf den das FSR Register zeigt kopiert. Außerdem wird bei jedem Schleifendurchlauf die Registeradresse MB_Adr um eins verringert und der Adresszeiger im FSR Register um eins erhöht. Weiterhin wird der Schleifenzähler I verringert. Die Nachricht ist vollständig in den PIC kopiert, wenn der Schleifenzähler gleich null ist.

Danach wird das Receive Ready Bit der ausgelesenen Messagebox wieder zurückgesetzt. Dazu wird in Abhängigkeit von der Messagebox Nr. die Adresse des RRR1 oder RRR2 Registers eingelesen. Ist die Messagebox Nr. größer als sieben wird die Variable MB_Adr auf die Adresse der Registers RRR2 gesetzt. Bei einer Messagebox Nr. kleiner gleich sieben wird die Adresse des Register RRR1 geladen. Nun wird das entsprechende Receiver Ready Register eingelesen und der Inhalt in der Variablen TMP zwischengespeichert. Der Inhalt von TMP (also der Inhalt des RRR Registers) wird mit der invertierten Interruptmaske (IMask) über eine UND-Operation verknüpft. Das Ergebnis dieser Verknüftung wird dann in das entsprechende RRR Register geschrieben. Durch diese Operation wird nur das Receiver Ready Bit der ausgelesenen Messagebox zurückgesetzt. Die Receive Ready Bit der anderen Messageboxen bleiben erhalten. Danach wird das INT-Register im CAN-Controller zurückgesetzt.

Damit ist das Kopieren der Nachricht aus dem CAN-Controller in den PIC abgeschlossen. Nun muß nur noch das H_Control-Byte erzeugt werden. Dazu wird als erstes die Messagebox Nr. in das H_Control-Byte kopiert. Dann werden die Bits, mit denen die Nachrichtenlänge kodiert ist, gesetzt. Dazu wird der Inhalt der Variablen MB_DLC überprüft und die Bits zur Kodierung der Nachrichtenlänge in der Variablen TMP zwischengespeichert. Das Ergebnis in dieser Variable wird dann über eine bitweise Oder-Operation mit der Variablen H_Control verknüpft. Nun wird zum Abschluß das CAN/RS232-Bit in der Variablen H_Control gesetzt. Dadurch ist der CAN-Bus als Quelle des

Datentelegramms im Speicher des PICs eindeutig definiert. Damit wurde die Nachricht vollständig aus dem CAN-Controller ausgelesen und das Unterprogramm durch einen Rücksprung in die Serviceroutine beendet.

6.2 Das Terminalprogramm

6.2.1 Die Aufgaben und Funktion des Terminalprogramms

Die Aufgabe des Terminalprogramms ist es Nachrichtentelegramme über die RS232-Schnittstelle des PC's, der als Terminal dient, zu senden und zu empfangen. Beim Senden soll das Programm, nachdem die Messagebox-Nr. und ein Byte, Integer oder Long Wert eingegeben wurde, die Daten automatisch in ein entsprechendes Nachrichtentelegramm mit Befehls- und Datenbytes umwandeln. Beim Empfang hat das Programm die Aufgabe das Befehlsbyte zu interpretieren und aus den nachfolgenden Datenbytes des Nachrichtentelegrammes wieder einen Byte, Integer oder Long Wert zu bilden. Außerdem bietet das Terminalprogramm noch die Möglichkeit eine Logbuch-Datei mit den gesendeten und empfangenen Nachrichtentelegrammen anzulegen.

Das Programm ist dabei sowohl für die Kommunikation mit dem RS232/CAN auf SSI-Interface als auch für die spezielle RS232 auf CAN-Version geeignet. Das Terminalprogramm setzt als Rechner einen PC mit MS-DOS als Betriebssystem voraus. Es ist auch in der DOS-Box unter Win95 lauffähig. Programmiert wurde das Programm mit Turbo Pascal 6.0.

Nachdem das Terminalprogramm „TWTERM.EXE“ gestartet wurde liest es automatisch die Konfigurationsdatei „TWTERM.INI“ für RS232-Schnitstelle ein. In der ersten Zeile dieser Datei wird die Baudrate der Übertragung angegeben und in der zweiten der COM-Port, der für die Übertragung genutzt werden soll. Die Übertragungsrate wird nicht direkt, sondern durch den Faktor 100 geteilt angegeben. Für das Terminalprogramm sind die in Tabelle 10 zusammengefaßten Übertragungsraten möglich.

Einstellung	***Baudrate***
48	4800
96	9600
192	19,2k
384	38,4k

Tabelle 10: Einstellung der Baudrate in "TWTERM.INI"

Die Angabe des gewünschten COM-Ports erfolgt durch eine Zahl zwischen 1 und 4. Dabei wird von der in Tabelle 11 aufgeführten Standard Hardwarekonfiguration ausgegangen.

COM-Port Nr.	***IRQ***	***I/O-Base***
1	4	$3f8
2	3	$2f8
3	4	$3e8
4	3	$2e8

Tabelle 11: Konfiguration der COM-Ports

Nachdem die RS232-Schnittstelle konfiguriert wurde, wird nun die Länge der Nachrichtenbox initialisiert. Dazu fragt das Terminalprogramm nach dem Namen des „Messageboxen-Konfigurationsfiles“. In diesem File ist der Reihe nach

für jede Messagebox die Anzahl Datenbytes pro Nachrichtentelegramm gespeichert. In der Abbildung 36 ist das Beispielskonfigurationsfile „CAN.CFG“ dargestellt.

```
0=0;  MessageBox0  => 0 Byte
1=1;  MessageBox1  => 1 Byte
2=1;  MessageBox2  => 1 Byte
3=2;  MessageBox3  => 2 Byte
4=2;  MessageBox4  => 2 Byte
5=4;  MessageBox5  => 4 Byte
6=4;  MessageBox6  => 4 Byte
7=8;  MessageBox7  => 8 Byte
8=4;  MessageBox8  => 4 Byte
9=0;  MessageBox9  => 0 Byte REMOTEBOX !
10=0; MessageBox10 => 0 Byte REMOTEBOX !
11=0; MessageBox11 => 0 Byte REMOTEBOX !
12=0; MessageBox12 => 0 Byte REMOTEBOX !
13=0; MessageBox13 => 0 Byte REMOTEBOX !
14=0; MessageBox14 => 0 Byte REMOTEBOX !
15=0; MessageBox15 => 0 Byte REMOTEBOX !
```

Abbildung 37: Beispiel Konfigurationsfile "CAN.CFG"

Der erste Wert in der Zeile enthält die Messagebox Nummer. Nach dem Gleich-Zeichen folgt die Messageboxlänge. Für die Messagebox-Länge sind die Werte 0, 1,2,4 & 8 zulässig. Der Wert 0 wird benutzt um eine Messagebox als Remote-Box zu kennzeichnen. Der Text nach dem Semikolon ist Kommentar.

Das Konfigurationsfile dient nicht zur Konfiguration der Messageboxen in dem Interface, sondern die Daten dieses Files werden nur von dem Terminalprogramm benutzt. Beim Senden eines Nachrichtentelegramms wird an Hand der Daten aus dem Konfigurationsfile entschieden, ob ein Remote Request oder ein 1-, 2-, 4- oder 8-Byte langes Datentelegramm gesendet werden soll. Dadurch ist sichergestellt, daß jede Messagebox mit dem richtigen Datentyp geladen wird. Aus diesem Grund sollte das Messagebox-Konfigurationsfile immer mit der aktuellen Konfiguration des Interfaces übereinstimmen.

Nun erfolgt eine Abfrage, ob eine Protokoll-Datei angelegt werden soll. Wenn diese Abfrage mit „ja“ beantwortet wird erfolgt die Eingabe eines Filenames. Nun wird eine Datei mit dem entsprechenden Namen in dem angegebenen Pfad angelegt, in der alle Übertragungen aufgezeichnet werden. Jede Zeile in der Datei setzt sich aus drei Spalten zusammen: der Messagebox-Nr. , den Daten (ein Zahlenwert vom Typ Byte, Int, Long oder acht Zahlen (byte)) und einem Zeitstempel. Dieser gibt den Zeitpunkt der Ausgabe des Datensatzes relativ zum Startzeitpunkt des Terminalprogramms an. Berechnet wird diese Zeit über den vier Byte langen Zähler, der im PC vom Systemtakt abgeleitet wird. Pro Stunde zählt dieser Zähler 65543 Takte. D.h. pro Sekunde werden 18,206 Takte gezählt. Der Zeitstempel bildet ein grobes Zeitraster und ist zur Bestimmung von nicht periodischen zeitlichen Abläufen gedacht.

6.2.2 Die Realisierung des Terminalprogramms

Der Sourcecode des Terminalprogramms ist auf zwei Files verteilt. Alle Funktionen, die für den Zugriff auf die serielle Schnittstelle benutzt werden, sind in einer Turbo-Pascal Unit mit dem Namen „twcom.pas“ zusammengefaßt. In dem anderen File mit dem Namen „twterm.pas“ ist das eigentliche Empfangsprogramm mit dem die Daten gelesen und verarbeitet werden untergebracht.

Die TWCOM-Unit wurde 1995 von Wayne Hoxsie erstellt und von mir modifiziert bzw. erweitert. Die Unit enthält neben Funktionen zur Initialisierung der Schnittstelle auch eine Interrupt-Serviceroutine, die für den Empfang der Daten

über die RS232-Schittstelle zuständig ist. In diese Serviceroutine wird hardwaremäßig verzweigt, nachdem ein Byte im Empfangsregister der seriellen Schnittstelle empfangen wurde und dann das empfangene Byte in einen 1024 Byte großen Ringbuffer gespeichert. Aus diesen Ringbuffer können die empfangenen Daten dann mit einem normalen Funktionsaufruf der Funktion „AsyncReceive“ über eine Polling-Schleife ausgelesen werden. Auf die weiteren Funktionen dieser Unit werde ich in dieser Arbeit nicht weiter eingehen, da Sie für das Verständnis der Verarbeitung der Daten beim Senden und Empfang nicht von Bedeutung ist.

Stattdessen werden in den folgenden drei Kapiteln die Funktionsweise der Sende- und Empfangsroutine sowie des Hauptprogramms ausführlich dargestellt.

6.2.3 Das Hauptprogramm des Terminalprogramms

Vom Hauptprogramm des Terminalprogramms werden alle Funktionen zur Initialisierung und Steuerung der Datenübertragung aufgerufen. Durch das Hauptprogramm werden also die logischen Verknüpfungen der einzelnen Funktionen untereinander gebildet. Das Flußdiagramm des Hauptprogramms ist in der Abbildung 38 dagestellt.

Nachdem das Programm „TWTERM.EXE“ gestartet wurde wird eine Startmeldung von der Hauptroutine auf dem Bildschrim ausgegeben. Nun soll das Konfigurationsfile „TWTERM.INI“ gelesen und die Daten im File entschlüsselt werden. Dazu ruft das Hauptprogramm die Funktion „Read_Konfig“ auf, die für diese Aufgabe zuständig ist. Wenn das Konfigurationsfile nicht geladen wurde wird von der „Read_Konfig“-Funktion eine Fehlermeldung ausgegeben und es werden die Defaultwerte COM-Port=1 und Baud=9600 an das Hauptprogramm übergeben.

In nächsten Schritt wird vom Hauptprogramm der Filename des Messagebox-Konfigurationsfiles abgefragt und die Messageboxen in dem Programm mit der Funktion „mb_konfig“ konfiguriert. In der „mb_konfig“-Funktion werden die Daten aus dem Konfigurationsfile eingelesen und interpretiert. Das Ergebnis wird in einem 16 Byte großem Array mit dem Namen „mb_lenght“ gespeichert. Nachdem die Konfiguration erfolgt ist, wird jetzt der aktuelle Zählerstand des PC internen Counters in der Variablen „start-time“ gespeichert. Dieser Zählerwert wird als Referenzwert zur Berechnung des Zeitstempels benutzt.

Nun erfolgt die Abfrage, ob ein LOG-File erstellt werden soll. Wird diese Abfrage mit Ja beantwortet wird der Name des Files, das angelegt werden soll abgefragt und versucht ein entsprechendes File anzulegen. Ist dies aber z.B. weil der Datenträger schreibgeschützt ist nicht möglich wird eine Fehlermeldung ausgegeben und das Programm beendet.

Nun wird die serielle Schnittstelle durch einen Aufrufe der Funktion „asyncinit“ aus der „twcom“-Unit initialisiert und die Variablen „NMR“ (New Message Received) und „done“ auf den boolean Wert „false“ sowie die Variable „CmdByte“ auf den Wert „true“ gesetzt. Mit der Variablen „NMR“ wird angezeigt, daß eine Nachricht vollständig empfangen wurde. Der Wert „false“ bedeutet, daß noch keine neue Nachricht vollständig empfangen wurde. Die Variable „done“ enthält die Abbruchbedingung für die Polling-Schleife. Die Aufgabe der Variablen „CmdByte“ ist es Befehls und Datenbyte auseinander zu halten. Ist diese Variable „true“ handelt es sich bei dem empfangenen Byte um ein Befehlsbyte. Damit ist die Initialisierungsphase des Hauptprogramms abgeschlossen und die Polling-Schleife, in der die Daten empfangen und gesendet werden, beginnt.

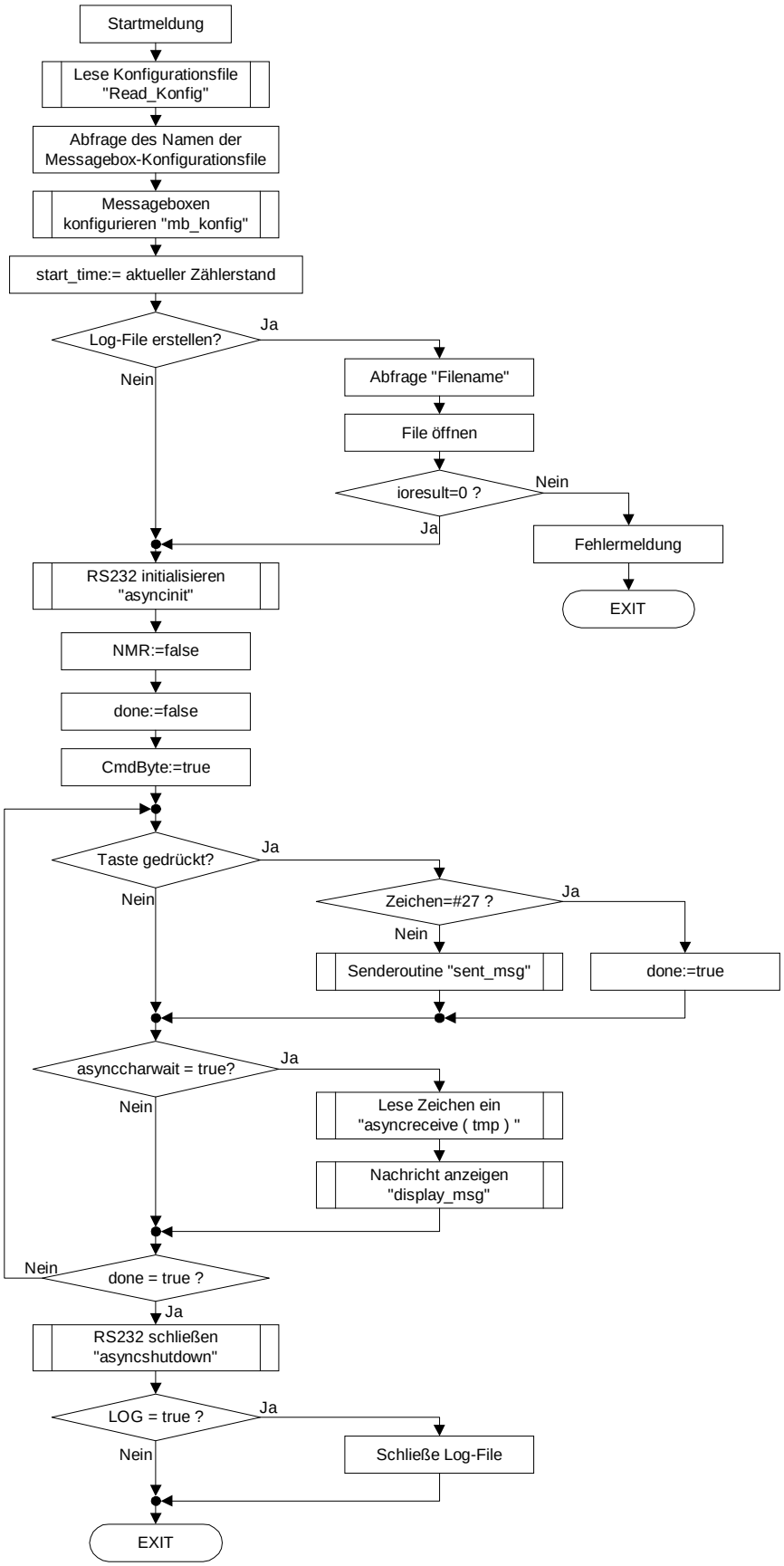


Abbildung 38: Flußdiagramm des Hauptprogramms von „TWTERM.PAS“

In der Polling-Schleife wird als erstes abgefragt, ob eine Taste gedrückt wurde. Ist dies der Fall wird das Zeichen eingelesen. Wenn es sich bei dem Zeichen um die Escape-Taste (Wert:=#27) handelt wird die Variable „done“ auf der Wert „true“ gesetzt. Damit ist die Abbruchbedingung am Ende der Polling-Schleife erfüllt und die Polling-Routine wird am Ende dieses Durchlaufs verlassen. Handelt es sich bei der gedrückten Taste nicht um die ESC-Taste wird in die Senderoutine „sent_msg“ verzweigt. Den Ablauf in dieser Funktion kann dem nächsten Kapitel entnommen werden. Nachdem diese Funktion abgearbeitet ist, wird das Programm normal fortgesetzt.

Als nächstes wird überprüft, ob neue Daten empfangen wurden. Dazu wird Überprüft, ob die Funktion „asynccharwait“ den Wert „true“ zurück liefert. Ist dies der Fall wird das Zeichen mit der Funktion „asyncreceive“ eingelesen und in der „tmp“-Variablen zwischengespeichert. Dann wird die Funktion „display_msg“ aufgerufen, in der die Verarbeitung der empfangenen Daten erfolgt. Der genaue Ablauf in der Funktion „display_msg“ kann dem Kapitel 5.2.5 entnommen werden.

Am Ende der Polling-Schleife erfolgt, wie bereits oben erwähnt, die Abfrage, ob die Polling-Schleife beendet werden soll. Dazu wird überprüft, ob die Variable „done“ gleich „true“ gesetzt ist. Ist dies nicht der Fall wird die Schleife ein weiteres mal durchlaufen. Wenn diese Bedingung aber erfüllt ist wird als nächstes die RS232-Schnittstelle mit der Funktion „asyncshutdown“ geschlossen sowie ein ggf. geöffnetes LOG-File geschlossen. Dazu wird die LOG-Variable auf den Wert „true“ hin überprüft. Ist diese Bedingung erfüllt wird das LOG-File geschlossen. Ist dies nicht der Fall wird das Programm direkt beendet.

6.2.4 Die Sendefunktion des Terminalprogramms

Die Senderoutine des Terminalprogramms setzt sich aus der eigentlichen Senderoutine und den Eingabefunktionen für die verschiedenen Datentypen zusammen. Ein Flußdiagramm der Funktion „send_msg“, die die Eingaberoutinen aufruft und das Senden der Daten durchführt, ist in der Abbildung 39 dargestellt.

Nachdem die Funktion „send_msg“ aus der Polling-Schleife des Hauptprogramms heraus aufgerufen wurde, wird als erstes die Messagebox Nr. abgefragt. Dann wird die Nachrichtenlänge mit Hilfe der Messagebox Nr. aus dem Konfigurationsarray „mb_lenght“ ermittelt. Je nach Länge des Datentelegramms wird nun in die entsprechende Eingaberoutine „SentX“verzweigt.

In den „SentX“-Funktionen wird eine zum Datentyp passende Eingabemaske aufgebaut und die Eingabe des Zahlenwertes, der gesendet werden soll, abgefragt. Danach wird dieser Zahlenwert, wenn notwendig, in einzelne Bytes zerlegt und das Befehlsbyte erzeugt. Das Befehls- und die Datenbytes werden dabei im einem Array (sd_byte) zwischengespeichert.

Bevor der eingegebene Zahlenwert jetzt gesendet wird, muß sichergestellt sein, daß das Kommunikationsinterface bereit ist Daten von der RS232-Schnittstelle zu empfangen. Deshalb wird die CTS-Leitung der RS232-Schnittstelle mit der Funktion „asyncCTS“ auf den Wert „true“ hin überprüft. Ist der Rückgabewert der Funktion nicht „true“ heißt dies, daß das Kommunikationsinterface nicht bereit ist und es wird solange gewartet bis dies der Fall ist. Während dieser Zeit ist der Empfang von Daten durch die Interruptserviceroutine weiterhin sichergestellt.

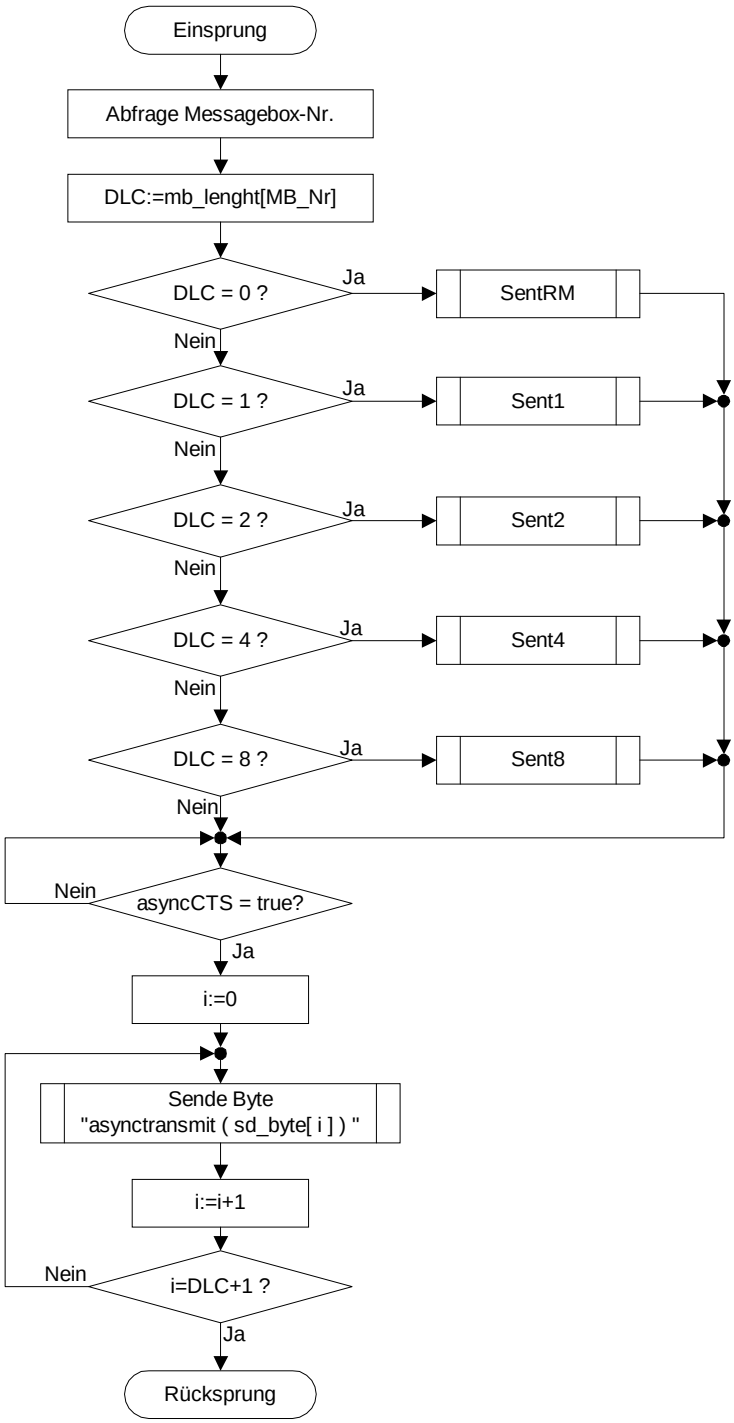


Abbildung 39: Flußdiagramm der Funktion "send_msg" des Programms "TWTERM.PAS"

Ist das Kommunikationsinterface bereit beginnt die eigentliche Sendeschleife in der die Daten des „sd_byte“-Arrays byteweise übertragen werden. Die Übertragung der Daten erfolgt mit der Funktion „asynctransmit“, der bei jedem Durchlauf der Sendeschleife jeweils ein Byte des „sd_byte“-Arrays zum Senden übergeben wird. Wenn der Stand des Schleifenzähler gleich der Anzahl der Datenbyte plus dem einem Befehlsbyte ist, ist die Nachricht vollständig über die RS232-Schnittstelle gesendet worden. Damit kann nun der Rücksprung in die Polling-Schleife erfolgen.

6.2.5 Die Empfangsroutine des Terminalprogramms

Der Displayroutine „display_msg“, deren Flußdiagramm in der Abbildung 40 dargestellt ist, wird bei jedem Aufruf aus der Polling-Schleife heraus die Variable „tmp“ mit einem empfangenen Byte übergeben. Die Aufgabe der „display_msg“-Routine ist es nun die einzelnen Bytes eines Nachrichtentelegramms zusammenzufassen und aus den Datenbytes einen Zahlenwert zu bilden. Aus diesem Grund ist es wichtig, daß in der Routine das Befehlsbyte, das als erstes in einem Nachrichtentelegramm gesendet wird, erkannt und interpretiert wird. Die Erkennung des ersten Bytes eines Telegramms, also des Befehlsbytes, erfolgt mit Hilfe der Variablen „CmdByte“ vom Typ boolean. Ist diese Variable „true“ handelt es sich um das Befehlsbyte; bei einem false“-Wert ist das empfangene Byte ein Datenbyte.

Die Unterscheidung zwischen Befehls- und Datenbyte erfolgt direkt nach dem Einsprung in die „display_msg“-Funktion. Ist die Variablen „CmdByte“ gleich „true“ wird in eine Unterroutine verzweigt, die das Befehlsbyte auswertet. Die Messagebox Nr. wird ermittelt, indem die unteren vier Bit des Befehlsbyte ausmaskiert werden und das Ergebnis in der Variable „MB_Nr“ gespeichert wird.

Nun erfolgt die Dekodierung der Nachrichtenlänge. Hierfür werden die Bits 4 & 5 des empfangenen Bytes ausmaskiert und um 4 Bit nach rechts geschoben. Das Ergebnis dieser Operation wird in der Variablen „btmp“ gespeichert. An Hand des in „btmp“ gespeicherten Wertes erfolgt nun die Initialisierung der Variablen „MB_DLC“ in der die Anzahl der Datenbytes des empfangenen Telegramms gespeichert wird.

Als nächstes wird das Remote Request Bit (Bit 6) aus dem empfangenen Byte ausmaskiert und das Ergebnis wieder in der Variablen „btmp“ gespeichert. Ist dieses Bit gesetzt handelt es sich um einen Remote Request, der besonders behandelt wird, denn der Remote Request besteht nur aus dem Befehlsbyte. Deshalb wird in diesem Fall die Nachrichtenlänge „MB_DLC“ auf den Wert null gesetzt. Weil der Remote Request damit auch vollständig empfangen wurde werden die Variablen „NMR“ und „CmdByte“ auf den Wert „true“ gesetzt. D.h. die nachfolgende Ausgaberoutine kann die Nachricht ausgeben und das nächste empfangene Byte ist wiederum ein Befehlsbyte.

Ist das 6. Bit in der Variablen „btmp“ nicht gesetzt, handelt es sich um ein „normales“ Datentelegramm, dem weitere Datenbytes folgen. Aus diesem Grund wird die Variable „CmdByte“ in diesem Fall auf den Wert „false“ gesetzt. Dadurch wird beim nächsten Aufruf der „display_msg“-Funktion bei der Abfrage am Anfang der Funktion in die Unterroutine verzweigt, die für den Empfang der Daten zuständig ist.

In dieser Unterroutine wird das empfangene Byte in dem Array „D_Byte“ gespeichert. Die Position, wo das Byte im Array gespeichert wird, ist durch die Variable „byte_cnt“ festgelegt. Diese Variable wird bei jedem Schleifendurchlauf um eins erhöht. Wenn alle Datenbytes einer Nachricht empfangen wurden ist der Wert der Variablen „byte_cnt“ gleich der Variablen „MB_DLC“. In diesem Fall wird der Zähler „byte_cnt“ auf Null zurückgesetzt und die Variablen „NMR“ und „CmdByte“ auf „true“ gesetzt. Damit wird signalisiert, daß eine Nachricht vollständig empfangen wurde und das es sich bei dem nächsten Byte um ein Befehlsbyte handelt.

Nachdem das empfangene Byte verarbeitet wurde, wird jetzt überprüft, ob eine vollständige Nachricht empfangen wurde. Dazu wird die Variable „NMR“ auf den Wert „true“ hin überprüft. Ist diese Bedingung erfüllt wird je nach empfangenen Datentyp in die entsprechende DisplayX-Routine verzweigt.

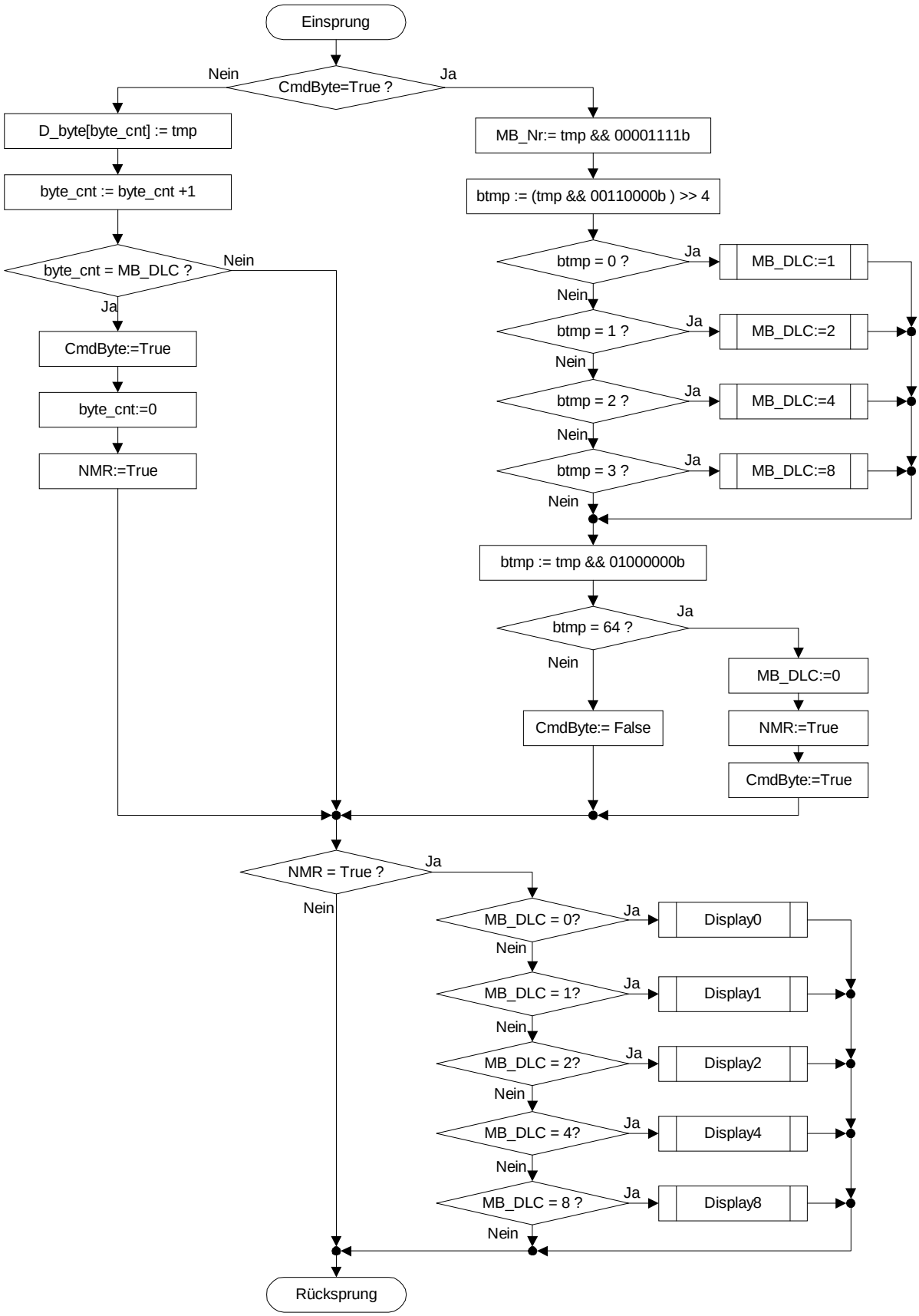


Abbildung 40: Flußdiagramm der Funktion "display_msg"

In den DisplayX-Routinen werden die Datenbytes aus dem Array „D_Byte“ wieder zu einem Zahlenwert zusammengefügt und in einer Ausgabemaske dargestellt. Außerdem werden in diesen Routinen auch die Daten in das LOG-File geschrieben. In der Display1-Routine ist neben der Ausgabe des empfangenen Nachrichtentelegramms auch die Auswertung und Anzeige von Fehlermeldungen untergebracht. Die Fehlermeldungen des CAN-Controllers, die nur bei der in Kapitel 5.1.2.3 erwähnten RS232 auf CAN-Version des Kommunikationsinterfaces über die RS232-Schnittstelle ausgegeben werden, werden in dem Terminalprogramm in Klartext dargestellt.
Wenn die Abfrage ergibt, daß die Bedingung „true“ für die Variable „ NMR“ nicht erfüllt ist, erfolgt der Rücksprung aus der „display_msg“ in die Polling-Schleife des Hauptprogramms.

6.3 Das VeCon-Programm

6.3.1 Die Aufgabe und Funktion des VeCon-Programms

Das Ziel des VeCon-Programms ist es mit Hilfe von Code-Beispielen das Senden und Empfangen von Nachrichten, als auch die spätere Verarbeitung der Nachrichten, zu demonstrieren und damit gleichzeitig die Funktionalität des Kommunikationsinterfaces zu verifizieren. Dabei sollte das Programm so programmiert werden, daß die entwickelten Routinen auch in Regelerprogrammen eingesetzt werden können. D.h. sie dürfen nicht die gesamte Rechenzeit benötigen.
Für das Beispielprogramm wurde die folgende Aufgabenstellung entwickelt: In dem Programm soll ein 32-Bit Zähler (Long) hochgezählt werden, der alle 62,5µs um eins erhöht wird. Der Inhalt dieses Zählers soll alle x-Zyklen über die SSI-Schnittstelle an das Kommunikationsinterface gesendet werden. Dabei wird der Wert des Zählers als 32-Bit Wert über die Messagebox 5 des CAN-Busses und die unteren 16-Bit des Zählers über die Messagebox 3 der RS232-Schnittstelle ausgeben. Nach wie vielen Durchläufen die Übertragung des Zählerstandes erfolgt läßt sich für die beiden Ausgabekanäle getrennt angeben. Nach dem Start des Programms sind für beide Routinen bereits Standardwerte gesetzt.
Um mit dem Programm aber auch den Empfang von Datentelegrammen zu demonstrieren, läßt sich die Anzahl der nötigen Durchläufe von außen über die Kommunikationskanäle einstellen. Dazu wird ein integer Wert an die Messageboxen 3 oder 4 geschickt. Der Inhalt der Messagebox 3 ändert den Wert für die Übertragung auf dem CAN-Bus, während die Messagebox 4 für die Übertragung über die RS232-Schnittstelle zuständig ist. Wenn über eine der anderen Messageboxen eine Nachricht an den VeCon geschickt wird wird der Inhalt der Nachricht je nach Datentyp in der Variablen „TESTVAR8“, „TESTVAR16“ oder „TESTVAR32“ gespeichert. Der Inhalt dieser Variablen kann dann mit dem Monitorprogramm des VeCon-Boards angezeigt werden. Von welchem Kommunikationskanal die Daten an den VeCon übertragen werden wird dabei nicht ausgewertet.

6.3.2 Die Realisierung des VeCon-Programms

Auf Grund des anwendungsorientierten Designs des VeCon-Chipsatzes unterscheidet sich die Programmierung dieses Bausteins erheblich von dem herhömmlicher RISC oder CISC-Prozessoren. So bietet der VeCon-DSP z.B. Platz für 1024 Befehle und 256 Byte RAM-Speicher. Neben dem nur begrenzt zur Verfügung stehenden Speicherplatz unterscheidet sich der VeCon-DSP aber auch in der Programmausführung von herkömmlichen Prozessoren. Bei der Programmierung von Regelungsalgorithmen auf herkömmlichen Prozessoren werden Regelungsroutinen zwar auch mit einer konstanten Zykluszeit ausgeführt, aber diese festen Zykluszeiten werden im allgemeinen durch spezielle Softwareroutinen (z.B. Interrupt gesteuert) realisiert. Außerhalb dieser Routinen wird der Programmablauf durch

Sprünge und Verzweigungen gesteuert. Der VeCon-DSP arbeitet im Gegensatz dazu mit einer festen Zykluszeiten von genau 62,5µs, die festverdrahtet in der Hardware realisiert ist. D.h. die Ausführung des Programms wird alle 62,5µs neu gestartet. Daraus folgt, daß alle Funktionen in dem Programm innerhalb dieser Zeit abgearbeitet worden sein müssen. Hinzu kommt, daß um die Kommunikationsroutinen später auch in Reglerprogramme integrieren zu können es wichtig ist so wenig dieser Zykluszeit wie möglich für die Kommunikation zu verwenden.
Um dies zu erreichen, wurde der Empfang und das Senden von Nachrichtentelegrammen in dem VeCon-Programm in mehrere kleine Unterroutinen zerlegt, von denen im allgemeinen immer nur eine pro Zyklus abgearbeitet wird. Die Koordination der einzelnen Routinen untereinander erfolgt mit Hilfe von zwei Statusregistern, in denen der aktuelle Stand der Verarbeitung gespeichert wird. Das Register „RSTATUS“ enthält dabei die Statusinformationen für die Empfangsroutinen und das Register „TSTATUS“ die der Senderoutinen. Die Aufteilung der beiden Register und die Funktion der einzelnen Statusbits können der Tabelle 12 und Tabelle 13 und den nachfolgenden Funktionsbeschreibungen entnommen werden.

	MSB															***LSB***
BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	NMR						JM64	JM32	JM16	JM8	JMR	RSENT	RBF	RDATA	RA	

Tabelle 12: Aufteilung des Register "RSTATUS"

Funktion der einzelnen Statusbts im „RSTATUS“-Register:

NMR_Bit (NewMessageReceived_Bit): Dieses Bit ist gesetzt, wenn ein neues Nachrichtentelegramm vollständig empfangen wurde.

JM64 (JoinMessage64): Wenn dieses Bit gesetzt ist wird ein acht Byte langes Datentelegramm zu einem Zahlenwert zusammengefügt. Diese Funktion ist in der aktuellen Version nicht implementiert.

JM32 (JoinMessage32): Wenn dieses Bit gesetzt ist handelt es sich um ein vier Datenbyte langes Nachrichtentelegramm, das nachdem es empfangen wurde zu einem LONG-Wert zusammengefügt wird.

JM16 (JoinMessage16): Ist dieses Bit gesetzt handelt es sich um ein 2 Datenbyte langes Nachrichtentelegramm, daß zu einem INT-Wert zusammengesetzt wird.

JM8 (JoinMessage8): Ist dieses Bit gesetzt wurde nur ein Datenbyte empfangen und der Inhalt dieses Byte wird als Int-Wert gespeichert.

JMR (JoinMessageReady): Nachdem aus den Daten des Nachrichtentelegramms ein Zahlenwert gebildet wurde ist dieses Bit gesetzt um damit zu signalisieren, daß dieser Wert jetzt in einer Variablen gespeichert werden kann.

RSENT_Bit (ReceiveSent-Bit): Durch das gesetzte RSENT_Bit wird signalisiert, daß die Übertragung eines Byte über die SSI-Schnittstelle eingeleitet wurde.

RBF_Bit (ReceiveBufferFull_Bit): Ein gesetztes RBF_Bit gibt an, daß ein Byte empfangen wurde.

RDATA_Bit: Durch dieses Bit wird gekennzeichnet, ob gerade ein Befehls- oder Datenbyte empfangen wurde. Ist dieses Bit gesetzt handelt es sich um ein Datenbyte.

RA_Bit (Receiver_Aktiv_Bit): Durch das Setzen diese Bits wird angezeigt, daß die Receiverroutine noch nicht vollständig abgearbeitet ist.

	MSB															***LSB***
BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name		NMT	Dk64	Dk32	Dk16	Dk8					TSENT	TF	TBF	TDATA		TA

Tabelle 13: Aufteilung des Register "TSTATUS"

NMT_Bit (NewMessageTransmisson_Bit): Dieses Bit wird gesetzt, wenn eine Nachricht gesendet werden soll.

Dk64 (Dekode 64): Durch das Setzen dieses Bits wird ein Zahlenwert in acht Datenbytes zerlegt. Diese Funktion ist in der aktuellen Version nicht implementiert.

Dk32 (Dekode 32): Ist dieses Bit gesetzt, wird die Routine zur Zerlegung eines LONG-Werts in vier Datenbytes aufgerufen.

Dk16 (Dekode 16): Wenn dieses Bit gesetzt ist wird ein INT-Wert in zwei Datenbytes zerlegt.

Dk8 (Dekode 8): Ein ein Byte langer Wert, der in einem 16-Bit Register gespeichert ist, wird in ein Datenbyte kopiert.

TSENT_Bit (Transmitter Sent): Dieses Bit wird gesetzt, wenn ein Byte gesendet wurde.

TF_Bit: Das „Transmission finished"-Bit wird beim Senden des letzten Datenbyte gesetzt.

TBF_Bit (BufferFull_Bit): Dieses Bit gibt an, daß ein Byte empfangen wurde. D.h. das Senden eines Bytes ist beendet.

TDATA_Bit: Diese Bit legt fest, ob gerade ein Befehls- oder ein Datenbyte bearbeitet wird. Ist dieses Bit gesetzt wird in der Empfangsroutine gerade ein Datenbyte verarbeitet.

TA_Bit (Transmitter_Aktiv_Bit): Solange dieses Bit gesetzt ist, ist die Transmitterroutine nicht vollständig abgearbeitet.

In dem Hauptprogramm wird jetzt jeweils an Hand der Statusbits die notwendige Bearbeitungsroutine für den nächsten Arbeitsschritt ausgewählt. Das Flußdiagramm dieser Routine ist in der Abbildung 41 dargestellt. Der Übersichtlichkeit halber sind die Funktionsaufrufe und Abfragen noch mal in Funktionsblöcke zusammengefaßt, die durch die gepunkteten Linien angedeutet sind. Außerdem sind unter den Funktionsnamen in eckigen Klammern die veränderten Steuerbits aufgeführt um den Ablauf des Programms besser nachvollziehen zu können.

In dem ersten Funktionsblock, der in dem Hauptprogramm abgearbeitet wird, werden die Testdaten erzeugt. In diesem Programmteil werden die Zählerstände der drei Variablen „Loop_CNT", „RS_CNT" und „CAN_CNT" um eins erhöht. Die Variable LOOP_CNT enthält dabei den 32-Bit langen Zähler, dessen Inhalt alle X-Perioden ausgegeben wird. Die Variablen „RS_CTN" und „CAN_CNT" dienen als Periodenzähler. Wenn der Inhalt dieser Variablen größer oder gleich dem eingestellten Samplewert ist, wird der Inhalt der Variablen „Loop_CNT" über die entsprechende Schnittstelle übertragen und der Periodenzähler zurückgesetzt.

In dem nächsten Funktionsblock sind die Routinen zum Empfang und zur Verarbeitung der empfangenen Daten untergebracht. In diesem Block wird erst abgefragt, ob die empfangenen Daten verarbeitet werden sollen, und erst dann die Abfragen, ob ein neues Nachrichtentelegramm empfangen werden soll. Diese Abfragereihenfolge erscheint auf dem ersten Blick unlogisch. Ist sie aber nicht, wenn man bedenkt, daß die Hauptroutine alle 62,5µs neu aufgerufen wird und

im Allgemeinen immer nur ein Unterprogramm zur Kommunikation in dieser Zeit aufgerufen wird. Durch die Anordnung der Verarbeitungsroutinen vor den Empfangsroutinen ist sichergestellt, daß die Daten eines empfangenen Telegramms auf jeden Fall verarbeitet und gespeichert werden. Erst, wenn die Verarbeitung der Daten erfolgt ist, kann die Abfrageroutine bis zu den Empfangsroutinen durchlaufen.

Als erstes erfolgt in dem Empfangs-Funktionsblock, wie bereits oben erwähnt, die Verarbeitung der vollständig empfangenen Nachtrichtentelegramme. Dazu wird im ersten Schritt überprüft, ob das NMR-Bit gesetzt ist. Wenn dieses Bit gesetzt ist, wurde ein Nachrichtentelegramm vollständig empfangen und die einzelnen Bytes des Telegramm müssen nun wieder zu einem Zahlenwert zusammengesetzt werden. Diese Aufgabe erfüllt das Unterprogramm „JMSelect“. In dieser Routine wird in Abhängigkeit von der Nachrichtenlänge die passende Verarbeitungsroutine aufgerufen und das Ergebnis zwischengespeichert. Wenn die Verarbeitung der empfangenen Nachricht beendet wurde, wird von dieser Routine das „JMR“-Bit gesetzt.

Dieses Bit wird wiederum beim nächsten Durchlauf des Programms abgefragt und wenn es gesetzt ist in das Unterprogramm „VarSave“ gesprungen. In diesem Unterprogramm wird das Ergebnis des vorherigen Unterprogramms in einer Variablen gespeichert. In welcher Variablen die Nachricht bzw. die Daten der Nachricht gespeichert werden, wird an Hand der Message-Nr. entschieden. In dieser Unterroutine kann jeder Messagebox-Nr. eine Zielvariablen zugeordnet werden. Nachdem die Speicherung erfolgt ist wird das „JMR“-Bit gelöscht. Damit ist die Verarbeitung des empfangenen Telegramms abgeschlossen.

Vom nächsten Teil der Empfangsroutine wird der eigentliche Empfang von der Nachrichtentelegrammen gesteuert. Am Anfang dieser Routinen wird überprüft, ob das „TA“-Bit gesetzt ist. Wenn dieses Bit gesetzt ist, ist die Übertragung eines Nachrichtentelegramms noch nicht abgeschlossen. Aus diesem Grund darf in diesem Fall die Empfangsroutine nicht damit beginnen Daten zu empfangen. Deshalb wird in diesem Fall direkt zu den Senderoutinen gesprungen. Als nächstes erfolgt die Abfrage des „RBF“-Bits. Ist dieses Bit gesetzt befindet sich ein empfangenes Byte in dem Zwischenspeicher. Der Inhalt dieses Zwischenspeichers wird nun in von der Funktion „Storage“ in den Nachrichtenbuffer einsortiert. Außerdem kontrolliert diese Funktion, ob ein Nachrichtentelegramm vollständig empfangen wurde und setzt dann ggf. das „NMR“-Bit. In nächsten Schritt wird das „RSENT“-Bit abgefragt. Mit diesem Bit wird angezeigt, daß eine Übertragung auf der SSI-Schnittstelle vom Master eingeleitet wurde. D.h. konkret, daß ein Byte über die SSI-Schnittstelle gesendet wird und gleichzeitig ein Byte, das die Daten enthält empfangen wurde. Aus diesem Grund wird die „Take“-Routine aufgerufen, die das empfangene Byte aus dem Empfangsregister der SSI-Schnittstelle ausließt und in einem Zwischenspeicher ablegt. Das „RSENT“-Bit wurde von der „SDUMMY“-Routine gesetzt, die immer dann aufgerufen wird, wenn ein High-Pegel an dem Receive-PIN des PICs anliegt. In dieser Routine wird die Übertragung eines Bytes eingeleitet.

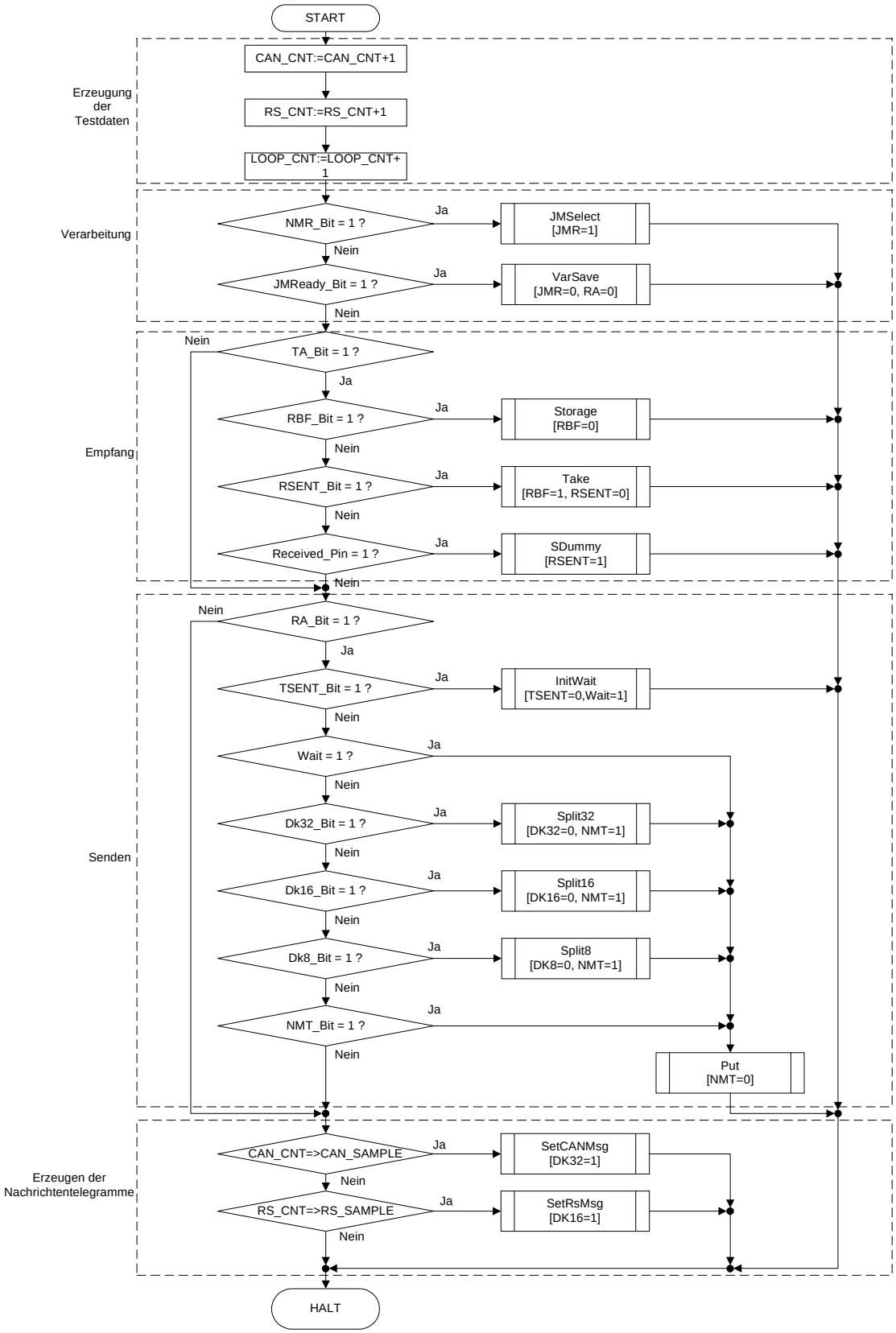


Abbildung 41: Flußdiagramm der Hauptroutine des VeCon-Programms

Die Empfangsroutinen benötigen zum Empfang eines Bytes drei Programmzyklen. Hinzukommen noch weitere zwei Zyklen, die für die Verarbeitung der empfangenen Daten notwendig sind. Damit ergeben sich die in der Tabelle 14 angegebenen Programmzyklen pro empfangene Nachricht.

Nachrichtenlänge (Anzahl Datenbyte)	***Anzahl Programmzyklen***
1	8
2	12
4	17
8	29

Tabelle 14: Anzahl der Programmzyklen pro empfangene Nachricht

Am Anfang der Senderoutinen wird an Hand des „RA“-Bit überprüft, ob die Empfangsroutine noch aktiv ist. Ist dieses Bit gesetzt ist die Empfangsroutine noch aktiv und die Senderoutine ist damit gesperrt, bis die aktuelle Nachricht vollständig empfangen wurde. Als nächstes in der Senderoutine das „TSENT“-Bit abgefragt. Dieses Bit ist gesetzt, wenn ein Byte gesendet wurde. In diesem Fall wird das „InitWait“-Unterprogramm aufgerufen, mit dem ein Wartezyklus eingeleitet wird. D.h. nach dem ein Byte zum PIC-Controller gesendet wurde, wird ein Wartezyklus eingelegt. Der Grund dafür ist, daß der PIC keine Möglichkeit hat mittels einem Handshake-Signal das Senden des VeCon zu steuern. Deshalb wird mit diesem Wartezyklus sichergestellt, daß der PIC die Daten verarbeiten kann. Da in dem Unterprogramm „InitWait“ die Variable „Wait“ auf den Wert 1 gesetzt wird, wird dann im nächsten Durchlauf das nächste Byte gesendet. Die Abfrage der „DKx“-Bits in der Senderoutine dient der Aufbereitung der Daten zu einem Nachrichtentelegramm. Wenn eines dieser „DKx“-Bits gesetzt ist, wird in die entsprechende Dekodierungsroutine verzweigt. In dieser Routine wird dann der in einem Zwischenspeicher übergebene Zahlenwert in einzelne Byte zerlegt und ein Befehlsbyte generiert. In Anschluß daran wird das „NMT“-Bit gesetzt und dann gleich damit begonnen das erste Byte der Nachricht zu senden. Der Grund dafür ist, daß damit der Zwischenspeicher im folgenden Programmzyklus nicht mehr benötigt wird und wieder für andere Aufgaben zur Verfügung steht. Als letztes wird in der Senderoutine überprüft, ob daß „NMT“-Bit gesetzt ist. Dieses Bit ist solange gesetzt bis das ganze Nachrichtentelegramm mit dem „PUT“-Unterprogramm übertragen wurde.

Als letzter Funktionsblock werden die Routinen, die die Abfrage der Zählerstände vornehmen, ausgeführt. Wenn der Zählerstand gleich oder größer dem angegebenen Wert ist, wird die „SetxxMsg“-Funktion aufgerufen, die die Daten in einen Zwischenspeicher schreibt und die „DKx“-Bits setzt, damit die Daten im nächsten Zyklus übertragen werden können.

Für die Aufbereitung eines Zahlenwertes und die Initialisierung des Befehlsbytes benötigt das Programm einen Programmzyklus. Für die Übertragung der einzelnen Byte einer Nachricht werden jeweils zwei Programmzyklen benötigt. In der Tabelle 15 ist die notwendige Anzahl Programmzyklen beim Senden für die verschiedenen Nachrichtenlängen zusammengefaßt.

Nachrichtenlänge (Anzahl Datenbyte)	*Anzahl Programmzyklen*
1	5
2	7
4	11
8	19

Tabelle 15: Anzahl der Programmzyklen pro gesendete Nachricht

In der jetzigen Form unterstützt das Demonstrationsprogramm für den VeCon-DSP sowohl das Senden und als auch das Empfangen von allen Nachrichtenlängen. Dadurch ist das Programm für den VeCon-DSP verhältnismäßig groß, es belegt etwa die Hälfte des gesamten zur Verfügung stehenden Programmspeicherplatzes. Der notwendige Speicherplatz läßt sich aber z.B. dadurch reduzieren, daß man nur einen Datentyp unterstützt.

7 Abschluß & Ausblick

Die Entwicklung des Kommunikationsinterfaces für das VeCon-Board bestand aus der Lösung mehrerer kleiner Teilaufgaben. Die erste Teilaufgabe war die Entwicklung eines Übertragungsprotokolls, das an die speziellen Anforderungen des VeCon-DSPs und die Aufgabe der Prozeßdatenübertragung angepaßt ist. Auf Grund der begrenzten Resourcen auf Seiten des VeCon-DSPs war es wichtig, daß das Übertragungsprotokoll möglichst einfach gehalten ist . Da beim VeCon-Board auch nur wenige freie I/O-Leitungen zur Verfügung stehen, konnten für das Protokoll keine aufwendigen Handshake Verfahren verwendet werden. Weil das Kommunikationsinterface zur Übertragung von Prozeßdaten verwendet werden soll, die im Allgemeinen als eine zeitliche Folge von Meß- und Sollwerten aufgefaßt werden können, wurde ein Übertragungsprotokoll entwickelt, bei dem die einzelnen Nachrichtentelegramme jeweils einen Datenwert enthalten. Die Nachrichtenlänge der Telegramme wurde dabei so gewählt, daß es möglich ist verschiedene Datentypen effizient zu übertragen. D.h. es ist möglich jeweils eine 8-, 16- oder 32-Bit lange Zahl direkt in einem Nachrichtentelegramm zu übertragen. Darüber hinaus ist es aber auch möglich die maximal mögliche Nachrichtenlänge von CAN-Telegrammen mit 64-Bit auszunutzen. Um die einzelnen Nachrichtentelegramme verarbeiten zu können besteht jedes Nachrichtentelegramm aus einem Befehlsbyte, daß die Längen und das Ziel der Übertragung enthält, und den eigentlichen Daten. Gesteuert wird die Übertragung durch nur zwei Handshake-Leitungen mit denen das Kommunikationsinterface die Übertragung von Daten sperren oder das Auslesen von Daten einleiten kann.

Die nächste Teilaufgabe bestand aus der Entwicklung der Hardware des Kommunikationsinterfaces auf der das Übertragungsprotokoll implementiert werden sollte. Nach ein paar Vorüberlegungen wurde die Hardware des Kommunikationsinterfaces um einen Mikrocontroller vom Typ PIC16C73A und einem Stand-Alone CAN-Controller vom Typ SAB80C90 herum entwickelt. Im Rahmen der Realisierung der Schaltung wurde auch ein Platinenlayout erstellt. Um die Platine des Kommunikationsinterfaces möglichst kompakt aufbauen zu können, wurden dafür soweit möglich SMD-Bauteile verwendet. Dadurch konnte die gesamte Schaltung auf einer 7,5cm * 5,5cm großen doppelseitigen Platine untergebracht werden.

Bei der Entwicklung der Software für das Kommunikationsinterfaces wurde versucht die Software modular aufzubauen. Durch den modularen Aufbau, in dem jeweils ganze Funktionsbereiche in einzelnen Files zusammengefaßt sind, ist es

sehr leicht die Software nachträglich zu verändern oder zu erweitern. Dies wurde auch schon durch die Entwicklung der RS232 auf CAN-Version des Kommunikationsinterfaces unter Beweis gestellt.

Eine weitere Teilaufgabe dieser Studienarbeit war die Entwicklung von Demonstrationsprogrammen für den PC und das VeCon-Board mit denen die Funktion des Kommunikationsinterfaces verifiziert werden konnte. Dazu wurde auf der PC-Seite ein einfaches Terminalprogramm entwickelt, das unter MS-DOS lauffähig ist. Das Programm ist in der Lange aus eingegebenen Daten direkt entsprechende Nachrichtentelegramme zu erzeugen, die dann über das Kommunikationsinterface an das VeCon-Board oder über die spezielle RS232 auf CAN-Version über den CAN-Bus gesendet werden. Außerdem ist das Programm in der Lage Nachrichtentelegramme zu empfangen und die empfangenen Daten dem Benutzer im Klartext darzustellen bzw. auf einem Datenträger zu speichern.

Das Demonstrationsprogramm für den VeCon-DSP zeigt, wie es möglich ist Daten direkt vom DSP aus über die SSI-Schnittstelle zu senden. Das Programm ist dabei in der Lage alle vom Kommunikationsinterface unterstützen Datenlängen zu verarbeiten. Dies gilt sowohl für die Empfangs- als auch für die Senderoutinen.

Die im Rahmen dieser Studienarbeit entwickelte Hard- und Software stellt eine funktionierende Lösung zur Übertragung von Prozeßdaten vom oder in den VeCon-DSP auf den CAN-Bus und die RS232-Schnittstelle eines Terminalrechners zur Verfügung. Darüber hinaus stellt das Kommunikationsinterface aber auch eine Basis für weitere Entwicklungen zur Verfügung. Auf Grund der großen Verbreitung der synchronen seriellen Schnittstelle ist es nicht nur möglich das VeCon-Board an dieses Interface anzuschließen, sondern das Kommunikationsinterface stellt eine universelle Möglichkeit vorhandene Mikrocontrollerboards mit einer CAN-Schnittstelle nachzurüsten zur Verfügung. Weiterhin besteht die Möglichkeit andere Komponenten, die über eine synchrone serielle Schnittstelle verfügen wie z.B. A/D-Wandler, über den CAN-Bus zu vernetzen. Hierfür wäre nur eine Anpassung der Software des Kommunikationsinterfaces notwendig, wobei allerdings ein Großteil der Kommunikations- und Initialisierungsroutinen übernommen werden könnten. Dadurch ist es möglich ganze Versuchsaufbauten über den CAN-Bus zu vernetzen und zu steuern.

8 Literaturverzeichnis

[Eim99] Eimer, Cord
Projekt:“Kommunikation über die synchrone serielle Schnittstelle des VeCon-Prototypenboards mit einem RS232-Teilnehmer“, IALB Feb. 1999

[Kai90] Kainka, B.
Messen, Steuern und Regeln über die RS232-Schnittstelle, 2.Auflage, Franzis-Verlag 1990

[Law98] Lawrenz,W.
CAN Controller Area Network-Grundlagen und Praxis, Hüthig-Verlag 98

[Max1] Maxim
Datenblatt MAX811, www.maxim-ic.com

[Mic1] Microchip
Datenblatt PIC16C7X-Familie, www.microchip.com, 1996

[Mic2] Microchip
Errata Sheet PIC16C73A, www.microchip.com, Nov. 1996

[Phi] Philips

Datenblatt PCA82C250,www.philips.com, Oct 1997

[Ris97] Ristedt, Jens
Projekt:“Schaltungs & Layoutentwicklungder VeCon-Grundig Adapterplatine“, IALB Dez. 1997

[Sie1] Siemens
Datenblatt SAE80C90/91, www.siemens.de, Jan. 1997

[Sie2] Siemens
Errata Sheet SAE80C90/91, Siemens München, May 1997 Rev.1.4

[Sie3] Siemens
Application Note AP2900: Connecting C166 and C500 Mikrocontrollers to CAN, www.siemens.de, Jun 1997

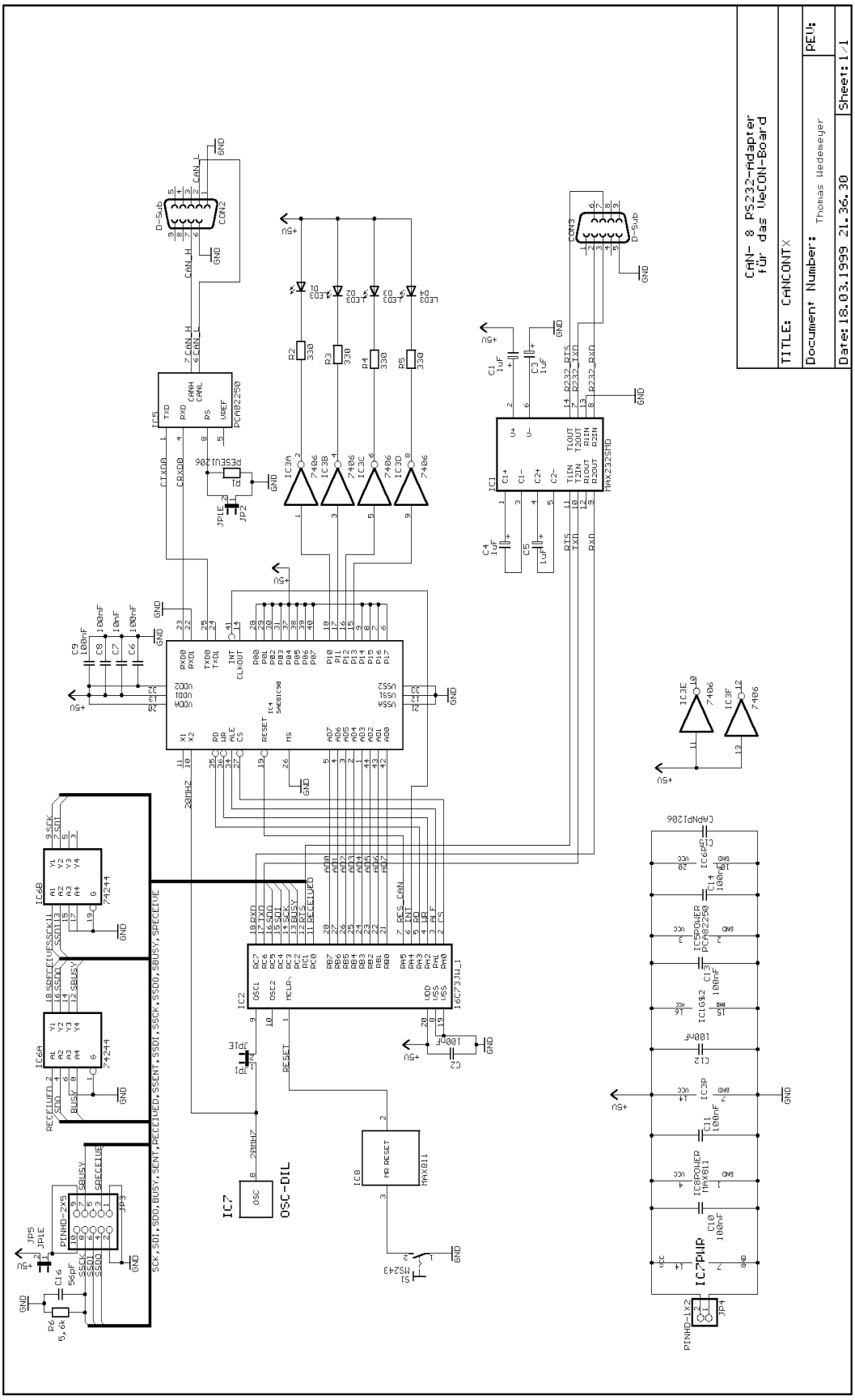
9 Anhang

9.1 Hardwareunterlagen

9.1.1 Zusammenfassung der technischen Daten des Kommunikationsinterfaces

Betriebspannung	
Versorgungspannung:	+5V
Stromaufnahme:	ca. 60mA
RS232-Schnittstelle:	
Übertragungsgeschwindigkeit:	1200kBaud bis 115,2kBaud (Eingestellt 38,4 kBaud)
Signalpegel:	V.24 konform +/- 10V
SSI-Schnittstelle:	
Übertragungsgeschwindigkeit:	bis 5MBit/sec (Eingestellt als Slave)
Signalpegel:	TTL
CAN-Bus:	
Übertragsgeschwindigkeit:	250kbit/sec bis 1MBit/sec (Eingestellt 250kBit/sec)
Übertragungsmedium:	Differentielle Übertragung gemäß ISO11898
Unterstützte Datentypen:	Byte, Int, Long, 8 * Byte

9.1.2 Schaltplan des Kommunikationsinterface



9.1.3 Layout des Kommunikationsinterface

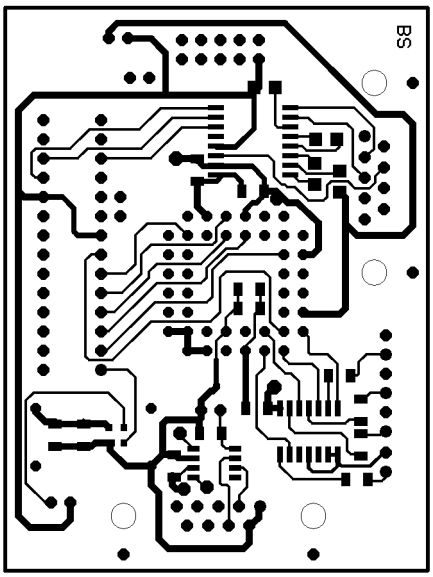


Abbildung 43: Bestückungsseite

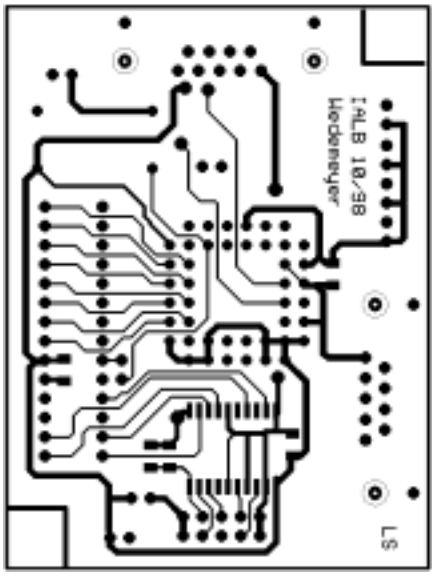


Abbildung 42: Lötseite

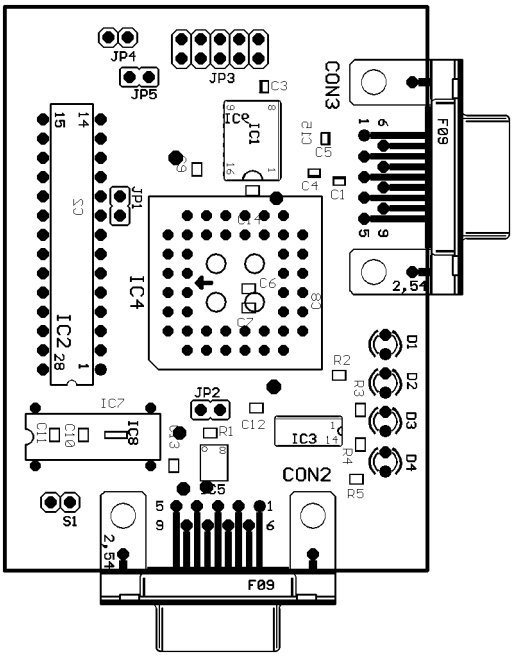


Abbildung 44: Bestückung oben

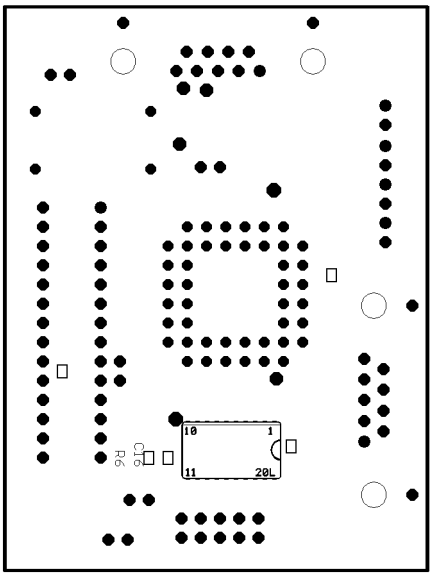


Abbildung 45: Bestückung unten

9.1.4 Bauteilliste

Bez.	Bauteilwert	Bezeichnung
C1	1uF / 16V	Elko SMD Bauform A
C2	100nF	Kondensator Bauform A
C3	1uF / 16V	Elko SMD Bauform A
C4	1uF / 16V	Elko SMD Bauform A
C5	1uF / 16V	Elko SMD Bauform A
C6	100nF	Kondensator SMD 1206
C7	100nF	Kondensator SMD 1206
C8	100nF	Kondensator SMD 1206
C9	100nF	Kondensator SMD 1206
C10	100nF	Kondensator SMD 1206
C11	100nF	Kondensator SMD 1206
C12	100nF	Kondensator SMD 1206
C13	100nF	Kondensator SMD 1206
C14	100nF	Kondensator SMD 1206
C15	100nF	Kondensator SMD 1206
C16	56pF	Kondensator SMD 1206
CON2	D-Sub	D-Sub Buchse 9-pol.
CON3	D-Sub	D-Sub Buchse 9-pol.
D1	LED3R	Led 3MM rot Low-Current
D2	LED3G	Led 3MM grün Low-Current
D3	LED3G	Led 3MM grün Low-Current
D4	LED3G	Led 3MM grün Low-Current
IC1	MAX232ECWE	MAX232 WSO16P
IC2	16C73JW	PIC-Mikrocontroller 20MHz
IC3	74HCT06	TTL SO14
IC4	SAE81C90	CAN-Controller
IC5	PCA82250	CAN-Transreceiver SO-8
IC6	74HCT244	TTL SO20L
IC7	20MHz OSC	20 MHz Quarzossilator
IC8	MAX811	SOT-143
JP1	2x1 Pins	Stiftleiste
JP2	2x1 Pins	Stiftleiste
JP3	2x5 Pins	Stiftleiste
JP4	2x1 Pins	Stiftleiste
JP5	2x1 Pins	Stiftleiste
R1	20kΩ	Widerstand 1206
R2	1,5kΩ	Widerstand 1206
R3	1,5kΩ	Widerstand 1206
R4	1,5kΩ	Widerstand 1206
R5	1,5kΩ	Widerstand 1206
R6	5,6kΩ	Widerstand 1206
S1	Taster	Schließer

9.1.5 Pinbelegungen am Kommunikationsinterface

CON2	
PIN	Name
1	GND
2	CAN_L
6	GND
7	CAN_H

Tabelle 16: Stecker CON2

CON3	
PIN	Name
2	RxD
3	TxD
5	GND
7	RTS

Tabelle 17: Stecker CON3

JP3	
PIN	Name
1	GND
2	GND
3	Receive
4	SDO
5	NC
6	SDI
7	SCK
8	Busy
9	Vcc
10	Vcc

Tabelle 18: Stecker JP3

JP4	
PIN	Name
1	GND
2	VCC

Tabelle 19: Stecker JP4

9.1.6 Modifikationen an der Grundig-Adapterplatine

Um das Kommunikationsinterface an die Grundig-Reglerbox anzuschließen sind ein paar Modifikationen an der Schaltung der Grundig-Adapterplatine notwendig. Auf der ursprünglichen Grundig-Adapterplatine sind nicht alle notwendigen Signale über einen Stecker nach außen geführt. Außerdem sind die Signale der synchronen seriellen Schnittstelle über einen RS232-Pegelwandler vom Typ MAX211 geführt. Das Kommunikationsinterface benötigt aber TTL-Pegel.

Die Modifikation besteht aus dem Entfernen und Überbrücken des RS232-Pegelwandlers und dem zusätzlichen Verschalten der notwendigen Signale an den Stecker RS232_3. Insgesamt besteht die Modifikation aus acht Arbeitsschritten:

- MAX221 (IC MAX1) auslöten
- Das Signal VH2 (=>SSDI) wird auf Pin 2 der RS232_3 Buchse gelegt. Der MAX1 Pin 3 wird dazu mit MAX1 Pin 6 verbunden. Außerdem muß der SW_DIP3 (PIN3/4) auf AN geschaltet sein.
- Das VSDAT-Signal (<=SSDO) auf Pin 3 der Buchse RS232_3 legen. Dazu wird der Pin 5 des MAX1 mit Pin 4 des gleichen Bausteins verbunden. Der Schalter SW15 muß in Down- und der SW14 in Up-Position geschaltet sein.
- Pin 13 & Pin 15 des Steckers ST-6 werden mit GND verbunden. Dazu wird der Pin 13 des Steckers ST-6, der schon auf GND liegt, mit dem Pin 15 verbunden.
- Das VSCLK-Signal (=>SSCK) wird auf den Pin 17 des Steckers ST-6 geführt. Dazu wird eine Verbindung zwischen dem Stecker VS11 Pin15 mit dem Stecker ST-6 Pin 17 gelegt. Außerdem muß der Schalter SW26 in die UP-Position geschaltet werden.
- Das CAP_IN-Signal (<=SReceived) von Stecker VS11 Pin12 wird auf Pin19 des Steckers ST-6 geführt. Zusätzlich müssen die Schalter Sw9 &SW14 in die Up-Position geschaltet werden.
- Durch eine Verbindung des Pins 19 des Steckers VS11 mit dem Pin 21 des ST-6 Steckers wird das MP1_IN-Signal (<=SBusy) nach außen geführt. Der Schalter SW13 ist in die Up-Position geschaltet.
- Als letztes wird die Versorgungsspannung von +5V auf die Pins 23 & 25 des Steckers ST-6 gelegt.

Mit der oben beschriebenen Modifikation ergibt sich die folgende Steckerbelegung für den Stecker RS232_3 und das Verbindungskabel:

RS232_3-Stecker			***Kommunikations-interface***	
PIN	***Signal***		***PIN***	***Signal***
2	VH2		6	SSDI
3	VSDAT		4	SSDO
7	GND		1	GND
8	GND		2	GND
9	VSCLK		8	SSCK
10	CAP_IN		3	SRECEIVED
11	MP1_IN		7	SBUSY
12	+5V		9	+5V
13	+5V		10	+5V