CAN-Bus:
Fehlererzeugung und Analyse

Diplomarbeit von
Stephan Pahlke und Sven Herzfeld

Fachhochschule Hannover
Fachbereich Elektro- und Informationstechnik
Fachgebiet Industrieelektronik und Digitaltechnik

Erstprüfer: Prof. Dr.-Ing. Zeggert
Zweitprüfer: Prof. Dr.-Ing. Brümmer

Sommersemester 2000

Zusammenfassung

CAN – Controller Area Network – ist ein von Bosch entwickelter Feldbus, der inzwischen zu den am weitesten verbreiteten Feldbussen in der Automobil- und Automatisierungstechnik gehört.

Im Rahmen einer Diplomarbeit im Sommersemester 2000 ist an der Fachhochschule Hannover ein CAN-Bus als Testsystem für weitere Versuche entstanden. Zur Busanbindung der im Industrieelektroniklabor entwickelten DIDO537-Platine wurde ein neues Aufsatzmodul konstruiert. Hierfür und für den Mikrocontroller Infineon 80C515C mit integrierter CAN-Schnittstelle liegen außerdem C-Programmbibliotheken vor, die in eigene Entwicklungen eingebunden werden können.

Eine Schaltung zur gezielten Störung der Busaktivitäten und ein einfaches Gerät zur Lokalisierung von Fehlermeldungen wurden ebenfalls entworfen.

CAN – Controller Area Network – is a fieldbus developed by Bosch which meanwhile belongs to the most frequently used fieldbusses in automotive and automation technic.

At Fachhochschule Hannover, a CAN bus system for testing future works was developed as a degree during summer 2000. A new ”piggyback” board connects the bus to the DIDO537-board developed in industrial electronics laboratory. For this and the microcontroller Infineon 80C515C with on-chip CAN-interface, libraries for the C language were written that can be used for own developments.

Bus activities can be disturbed by a special board, and a simple device to locate error frames was also created.

Eigenständigkeitserklärung

Wir versichern, diese Diplomarbeit selbständig bearbeitet und außer den genannten keine weiteren Hilfsmittel genutzt zu haben.

Die Abschnitte werden wie folgt zugeordnet:

Stephan Pahlke: 3, 4, 6, 7, 9, 10, 12.1, 12.2, 13

Sven Herzfeld: 2, 5, 7, 11, 12.3, 14 und alle Programmierarbeiten

Sven Herzfeld	Stephan Pahlke
Mat.-Nr. 86 52 90	Mat.-Nr. 88 04 08

Aufgabenstellung

Die Aufgabenstellung umfasst die folgenden Punkte:

1. Installation und Konfiguration des Vector CANanlyzer-Paketes
2. Aufbau eines CAN-Busses mit zwei selbständigen Teilnehmern
3. Entwurf eines Platinenaufsatzes zum Anschluss der im Labor für Industrieelektronik und Digitaltechnik entwickelten Mikrocontrollerplatine DIDO537 an den CAN-Bus
4. Design einer Platine zur Erzeugung eines Kurzschlusses zwischen CAN_H und CAN_L sowie zur Unterbrechung des CAN-Busses als auch zur definierten Zerstörung von Datenrahmen
5. Schaltungskonstruktion zur Erkennung von aktiven Errorframes in Abhängigkeit von der Richtung des auslösenden Teilnehmers

Inhaltsverzeichnis

Anhang 71

Abbildungsverzeichnis

Tabellenverzeichnis

Teil I

Allgemeines

1 Anmerkung zur Dokumentation

Die Dokumentation zu dieser Diplomarbeit liegt neben der gedruckten auch in zwei verschiedenen elektronischen Versionen im PDF-Format vor. Eine davon entspricht der gedruckten Fassung, die andere ist speziell für die Bildschirmausgabe entwickelt.

Dateien im von Adobe entwickelten PDF-Format können mit dem für verschiedene Betriebssysteme kostenlos erhältlichen *Acrobat Reader* gelesen und gedruckt werden. Mehr Informationen und die Möglichkeit zum Herunterladen finden Sie im Internet unter www.adobe.com/acrobat.

Im Anhang sind die benötigten Datenblätter und Applikationen als separate Dateien verfügbar.

2 Einführung in den CAN-Bus

2.1 Einsatz von Feldbussen

Die steigende Anzahl von Rechnersystemen erfordert deren Vernetzung in unterschiedlichen Einsatzgebieten. Neben den bekannten Weitverkehrsnetzen (WAN = Wide Area Network) wie dem Internet und den lokalen Netzen (LAN = Local Area Network) wie Ethernet gibt es auch weniger beachtete Netzwerke, die sich mit der Übertragung von Daten zur Maschinen- und Anlagensteuerung beschäftigen.

Ist die Technik in der Leitebene einzelner Industrieunternehmen beziehungsweise deren Abteilungen ist in der Regel noch mit den LANs identisch, doch die Anforderungen der Zellenebenen, die einzelne Produktionszellen innerhalb einer Abteilung verbinden, weichen hiervon ab. Obwohl die Datenvolumina in den Produktionszellen meist kleiner sind, wird hier aus Kostengründen oftmals auf die bewährte LAN-Technik zurückgegriffen.

In der untersten Ebene der Vernetzung sind die sogenannten Feldbusse anzutreffen. Sie transportieren überwiegend geringe Datenmengen, haben aber erhöhte Anforderungen an das Zeitverhalten. Deterministisches Zeitverhalten, also die Vorhersagbarkeit der Zeit, die eine Nachricht maximal braucht, um zum Empfänger zu gelangen, ist ihr herausragendes Merkmal.

Daten wie zum Beispiel Messwerte müssen sehr schnell zur Auswertung gelangen. Zu den anschaulichsten Beispielen gehören moderne PKW: Messwerte von Lambda-Sonde und Klopfsensoren sollen möglichst schnell dafür sorgen, dass der Verbrennungsvorgang optimiert wird. Noch dringlicher ist diese Anforderung bei der Bremsanlage, wo eine Verzögerung bei der Datenauswertung gleichbedeutend mit der Verlängerung der Reaktionszeit des Systems ist.

Im Gegensatz zu diesen „harten Echtzeitanforderungen“, wo eine vorher definierte Zeit nicht überschritten werden darf, gibt es auch „weiche Echtzeitanfoderungen“. Hierbei sind die Aufgaben weniger zeitkritisch, so dass auch Techniken wie das Ethernet zum Einsatz kommen können. Beispielsweise ist ein sehr schwach belastetes Ethernet durchaus in der Lage Nachrichten schnell zu übertragen, dieses ändert sich aber, im Gegensatz zu Feldbussen, sobald die Netzauslastung größer wird.

Eine weitere Anforderung an die Feldbusse ist eine hohe Störsicherheit. Typischerweise werden Feldbusse in Maschinenhallen, Anlagen oder Fahrzeugen eingesetzt in denen starke Störfelder vorhanden sind.

In der Regel werden nur technische Einrichtungen verbunden, daher implementieren Feldbusse keine höheren Schichten des ISO/OSI-Schichtenmodells, siehe Anhang C. Von den sieben Schichten dieses Modells bilden Feldbusse – wie auch CAN – normalerweise nur die Layer 1 (Physical Layer = Bitübertragungsschicht) und 2 (Data Link Layer = Verbindungsschicht) ab, es gibt aber auch verschiedene Definitionen für die Anwendungsschicht (Layer-7 Application Layer).

2.2 Besonderheiten von CAN

CAN wurde ab 1983 von Bosch auf Anforderung von Daimler-Benz und BMW als Automobilbus entwickelt. Grundlage hierfür war die Tatsache, dass ein Mittelklasse-PKW über 600 verschiedene Kabelbaumtypen mit mehr als 2000 Metern Kabellänge und über 100 kg Gewicht gehabt hat.

Konventionelle UARTs (Universal Asynchronous Receiver/Transmitter), wie sie auch in PCs verwendet werden, sind nur für Punkt-zu-Punkt-Verbindungen geeignet und deshalb nicht für die Kraftfahrzeugindustrie einsetzbar.

1987 führte Intel den ersten CAN-Chip ein. Seit 1992 wird CAN in der Mercedes-S-Klasse eingesetzt, später folgten auch andere Automobilhersteller. Ebenfalls seit 1992 findet CAN Verwendung in der allgemeinen Automatisierungstechnik. Inzwischen ist dieser Feldbus mit großem Abstand der mit den meisten installierten Netzknoten.

Während der überwiegende Teil der Automobilhersteller eigene Entwicklungen zugunsten der CAN-Technik aufgab, hat DaimlerChrysler kürzlich mit der Arbeit an einem eigenen Bussystem begonnen.

Bustopologie

CAN arbeitet standardmäßig mit einer linienförmigen Topologie, siehe Abbildung 1. Dabei ist die Netzausdehnung durch die Signallaufzeit auf etwa 40 m bei 1 MBaud oder 1 km bei 80 kBaud eingeschränkt. Es gibt keine im Protokoll festgelegte Höchstgrenze für die Anzahl der Busteilnehmer, in der Praxis wird diese allerdings durch die Leistungsfähig-

keit der Bustreiber beschränkt. Je nach Bustyp gibt es für die Anzahl der Busteilnehmer verschieden definierte Mindestwerte.

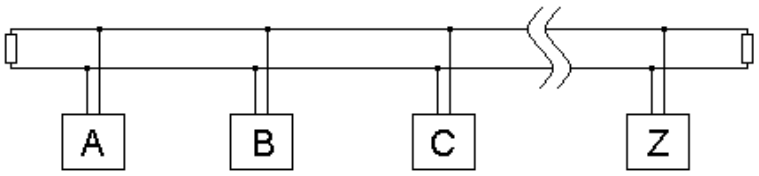


Abbildung 1: CAN-Bustopologie

Multimaster/Multislave

Im CAN-Bus sind alle Teilnehmer gleichberechtigt, das bedeutet, dass bei einen zufälligen Zugriff auf den Bus jeder Teilnehmer sobald der Bus frei ist senden kann.

Nachrichtenorientiertes Protokoll

Ein Teilnehmer am CAN-Bus hat nicht, wie sonst üblich, eine Adresse. Zur Identifizierung von Nachrichten wird ihre Kennung, der Identifier gesendet. Damit verbunden ist die Tatsache, dass jede Nachricht von jedem Teilnehmer empfangen werden kann (Broadcast-Betrieb).

Zerstörungsfreie Arbitrierung

Da der Zugriff auf den Bus zufällig erfolgt, kann es vorkommen, dass mehrere Teilnehmer gleichzeitig den Sendevorgang beginnen. Übliche Protokolle wie CSMA/CD (Carrier Sense Multiple Access/Collision Detection, Ethernet) lassen die Beteiligten eine Kollision erkennen und nach einer gewissen Zeit einen neuen Sendeversuch starten. Dabei steigt natürlich die Wahrscheinlichkeit weiterer Kollisionen. CAN nutzt eine sogenannte zerstörungsfreie Arbitrierung[1] mittels des Identifiers. Dieses Zugriffsverfahren enthält eine automatische Priorisierung aufgrund des Identifiers der einzelnen Teilnehmer. Derjenige Teilnehmer mit dem kleineren Identifier hat hierbei die höhere Priorität. Eine höherpriore Nachricht setzt sich gegenüber einer niedrigprioren bei der Arbitrierung durch. Die Nachricht, die sich durchgesetzt hat, wird bei der Arbitrierung nicht zerstört sondern vollständig übertragen. Eine Kollision erkennen nur die Teilnehmer mit Nachrichten niederer Priorität, die ihre Botschaften anschließend erneut senden.

[1] Von „arbitrage“, französisch für Schlichtung, Schiedsspruch

Die Bezeichnung eines solchen Vorganges ist je nach Quelle unterschiedlich: Neben *CSMA/CA* für „Carrier Sense Multiple Access/Collision Avoidance“ ist ebenso *CSMA/CD NDBA* für „Carrier Sense Multiple Access/Collision Detection with Non-Destructive Bit Arbitration“ sowie *CSMA/CD-CR* für „ . . . /Collision Resolution“ in Gebrauch.

Sicherheit

Die kurze Blocklänge der Nachrichten verringert die Wahrscheinlichkeit, dass innerhalb einer Nachricht ein Fehler auftritt. Damit ist die Gefahr, dass eine Nachricht durch eine Störung beschädigt wird geringer als bei längeren Nachrichten. Wird eine Nachricht als gestört erkannt, so sendet der Teilnehmer, der dieses festgestellt hat, eine Fehlermeldung, den Errorframe. Alle anderen Teilnehmer senden daraufhin ihrerseits einen Errorframe. Nach Ablauf dieser Botschaften überträgt der Sender der ursprünglichen Nachricht diese erneut, somit bleibt die Datenkonsistenz im Netz gewährleistet. In jedem Controller sind Mechanismen vorhandenen, die dafür sorgen, dass ein Teilnehmer der ständig Fehler erzeugt sich selbst abschaltet. CAN war damit der erste fehlertolerante Feldbus im Auto!

2.3 Verschiedene Versionen von CAN

Unterschiedliche Anforderungen an die Bustechnologie haben zu verschiedenen Arten der CAN-Implementationen geführt. Unterschieden werden CAN 2.0A und 2.0B, BasicCAN und FullCAN, sowie Low-Speed- und High-Speed-Netze.

CAN 2.0A und 2.0B

Der Wunsch nach einer über die Standarddefinition von CAN hinausgehenden Version wurde geäußert als der 11-Bit-Identifier nach CAN 2.0A in einigen Anwendungen nicht mehr ausreichte. CAN 2.0B erweitert die Nachrichten um ein zweites Identifier-Feld mit 18 weiteren Bits, so dass nun insgesamt 29 Bits für die Identifizierung einer Nachricht zur Verfügung stehen. Diese erweiterten Telegramme werden als Extended Frames bezeichnet.

CAN 2.0B mit seinen Extended Frames ist nicht in allen CAN-Controllern implementiert. Wenn Nachrichten mit einem 29-Bit-Identifier von diesen Controllern empfangen werden kommt es zu Fehlermeldungen, da in einigen Punkten gegen die Definition des Standard-Datenpaketes verstoßen wird.

Neuere Controller für CAN 2.0A verhalten sich CAN 2.0B-passiv, das heißt, sie erkennen den Extended Frame und ignorieren ihn. In Netzen die vollständig mit Controllern nach CAN 2.0B oder zumindest mit 2.0B-passiven Controllern ausgestattet sind können problemlos sowohl Nachrichten mit 11-Bit- als auch mit 29-Bit-Identifier genutzt werden. Ein 29-Bit-Identifiert ist mit keinem 11-Bit-Identifier identisch, es können gleichzeitig 2^{29} Identifier für Extended Frames und zusätzlich 2^{11} Identifier für Standard Frames verwendet werden. Sind die 11 höchstwertigen Bits bei beiden gleich, wird bei gleichzeitigem Buszugriff der Standard Frame übertragen, siehe Abschnitt 2.5.2.

BasicCAN und FullCAN

Die Unterscheidung von BasicCAN und FullCAN bezieht sich auf die Controller. BasicCAN-Controller empfangen jedes auf dem Bus vorhandene Datenpaket und leiten es an den angeschlossenen Rechner weiter. Ist der Empfangspuffer für die eingehenden Nachrichten voll, so gehen *neue* Nachrichten verloren.

Controller für FullCAN besitzen einen Filter, der nur Nachrichten mit bestimmten Identifiern durchlässt. Dabei kann für jedes einzelne Bit des Identifiers gewählt werden, ob dieses high oder low sein soll. Bei den meisten FullCAN-Controllern gibt es zusätzlich eine Filtermaske. Durch ihre Programmierung kann für jedes Bit separat angegeben werden, ob es für den Filtervorgang genutzt oder sein Wert ignoriert werden soll.

Zusätzlich beantworten FullCAN-Controller die Anforderung von Daten, das sogenannte Remoteframe, durch eigenständiges Senden des angeforderten Datentelegrammes. Diese automatische Antwort ist bei einigen Controllern abschaltbar.

Bei Überlast der Applikation gehen bei FullCAN die *alten* Telegramme im Empfangspuffer verloren, der CPU wird ständig das aktuelle Telegramm zur Verfügung gestellt.

FullCAN-Controller haben mehrere Register, die auf jeweils einen Identifier konfiguriert werden müssen, sobald das Register als Sende- oder Empfangsregister genutzt wird. Die Anzahl der Nachrichten, die empfangen werden können, ist somit auf die Zahl der für den Empfang vorgesehenen Register beschränkt. Durch eine Maskierung wird die Anzahl der Nachrichten erhöht, die von den einzelnen Registern empfangen werden können. Eine Maskierung ist nur für alle Register gemeinsam möglich, einige Controller besitzen jedoch ein spezielles Register, das nur für den Empfang genutzt werden kann und eine eigene Filtermaske besitzt.

BasicCAN-Controller sind einfacher im Aufbau und preiswerter, belasten aber die angeschlossene CPU stärker, da diese für das Auswählen der nutzbaren Nachrichten und das Beantworten von Remoteframes zuständig ist. Ein FullCAN-Controller erledigt diese Aufgaben intern. In vielen Fällen ist es sinnvoller, die alten Nachrichten zu verlieren, um die aktuelle zu verarbeiten.

Einige Controller bieten eine Mischung aus BasicCAN und FullCAN. So gibt es BasicCAN-Controller, die einen Eingangsfilter besitzen. Dieser ist überwiegend auf ein Byte beschränkt, so dass die drei niederwertigsten Bits weiterhin von der CPU ausgewertet werden müssen. Durch geschickte Wahl der Identifier kann man diesen Nachteil umgehen.

Normungen für die physikalische Schnittstelle

Die ISO hat für die Unterscheidung zwischen Nieder- und Hochgeschwindigkeitsnetzen eine simple Festlegung getroffen und die Grenze auf 125 kBit/s festgelegt. Für CAN gibt es mehrere Standards:

Die **ISO 11898** behandelt High-Speed-Netze bis 1 MBaud. Sie sind für mindestens 30 Teilnehmer definiert. Bei hohen Baudraten geschieht die physikalische Busansteuerung schnellstmöglich. Für langsamere Übertragungen ist es bei einigen Controllern möglich, zur Verringerung von Störungen im Hochfrequenzbereich die Flankensteilheit der Bits zu reduzieren (Slope Control).

Der **J1938**-Standard aus den USA für LKW und Omnibusse ist der ISO 11898 ähnlich, er ist allerdings grundsätzlich für 250 kBaud ausgelegt.

Fehlertolerante Konzepte sind von Philips und Texas Instruments für PKW entworfen worden und in der **ISO 11519** für langsamere Verbindungen definiert. Sie sind für bis zu 20 Knoten spezifiziert.

Die vierte und eine etwas exotische Variante ist die **ISO/WD 11992** für Punkt-zu-Punkt-Verbindungen zwischen landwirtschaftlichen Fahrzeugen und ihren Arbeitsgeräten.

2.4 Grundlegende Eigenschaften der Übertragung

2.4.1 Bitcodierung

CAN nutzt eine Non-return-to-zero-Codierung (NRZ). Dabei wird einem Bit genau ein Zustand des Logikpegels zugeordnet, low = 0 und high = 1. Mehrere gleiche Bits hintereinander erscheinen auf dem Bus also als ein entsprechend längerer gleichbleibender Pegel. Dieses ermöglicht gegenüber einem RZ-Code, der in jedem Bit auf Null zurückgeht, bei gleicher Baudrate die doppelte Datenrate. Da bei CAN kein Takt übertragen wird, muss auf eine andere Art gewährleistet werden, dass die Busteilnehmer die Bits korrekt synchronisieren, deshalb wird grundsätzlich nach fünf gleichen Bits ein sogenanntes Stopfbit (Stuffbit) eingefügt. Ein Stopfbit beginnt mit einem Pegelwechsel und hält diesen neuen Pegel für die Dauer eines Bits. Der CAN-Controller übernimmt das Einfügen des Stopfbit beim Senden und entfernt es bei einer empfangenen Nachricht automatisch wieder aus dem Datentelegramm.

Der Low-Pegel ist auf den CAN-Bus dominant. Wenn auf den Bus gleichzeitig rezessive High-Pegel und dominante Low-Pegel gesendet werden, erscheint auf dem Bus der Low-Pegel.

2.4.2 Arbitrierung

Wie bereits in Abschnitt 2.2 erwähnt geschieht die Arbitrierung verlustfrei anhand des Identifiers auf der Grundlage der Dominanz des Low-Pegels. Die Teilnehmer senden den Identifier ihrer Nachricht, bis diese sich in einem Bit unterscheiden. Der Teilnehmer, der in diesem Augenblick einen rezessiven High-Pegel sendet, bemerkt, dass das von ihm gesendete Signal nicht dem entspricht, was nun auf dem Bus anliegt. Er beendet den Sendevorgang, bis der andere Teilnehmer das Übertragen seiner Nachricht abgeschlossen hat.

Der Identifier ist bei Standard-CAN 11 Bits lang, ermöglicht also 2048 verschiedene Nachrichten. Jeder Identifier darf nur einfach verwendet, das heißt von nur einem Teilnehmer gesendet, werden. Wenn ein Sender, der sich in der Arbitrierungsphase durchgesetzt hat, am Bus einen Pegel feststellt, den er nicht gesendet hat, erkennt er darin einen Fehler. Die Beschreibung der Fehlerbehandlung folgt im nächsten Kapitel.

2.5 Nachrichtentypen bei CAN

2.5.1 Grundüberlegungen

In jedem Netzwerk wird ein Nachrichtentyp benötigt, der die Nutzdaten enthält. Die Nutzdaten werden mit einem Datenrahmen umgeben, damit sie auf einem Netz versendet werden können. Dieser Rahmen enthält die Adresse – bei CAN den Identifier – und kann in Abhängigkeit von der Aufgabe des Netzes noch wesentlich mehr Informationen beinhalten.

Einige Netzwerke benötigen besondere Nachrichten, die die Konfiguration des Netzes oder den Buszugriff steuern. CAN braucht beides nicht, verfügt aber dennoch über weitere Nachrichtentypen, um die Anforderungen an Sicherheit und Datenkonsistenz so effektiv wie möglich zu gewährleisten.

Die sogenannten Remoteframes enthalten selbst keine Nutzdaten, sondern fordern diese aus dem Netz an. Die Nutzung von Remoteframes hat zwei Vorteile: es ist nicht erforderlich Nachrichten zu definieren die einen Controller dazu veranlassen seine Daten zu senden. Der zweite Vorteil ist, dass durch das festgelegte Format eine automatische Verarbeitung möglich ist, die in FullCAN-Controllern realisiert wird. Bedingt durch die Spezifikationen CAN 2.0A und 2.0B gibt es Daten- und auch Remotetelegramme für beide Standards.

Für die Sicherstellung der Datenkonsistenz im Netz ist es erforderlich einen erkannten Fehler für alle Teilnehmer deutlich zu machen. Gäbe es eine solche Mitteilung nicht, könnte ein Teilnehmer mit Daten arbeiten, die aufgrund einer Störung ungültig sind. Realisiert wird diese Mitteilung mit einem Errorframe, der verschickt wird, wenn ein beliebiger Controller während einer Nachricht einen Fehlerzustand erkannt hat.

Ein weiteres Telegramm, der Overloadframe, fordert die anderen Teilnehmer auf, das Senden neuer Nachrichten hinauszuzögern. Dadurch wird erreicht, dass die vorliegenden Daten vom Buscontroller verarbeitet werden können, bevor neue Nachrichten eintreffen.

2.5.2 Daten- und Remoteframes nach CAN 2.0A

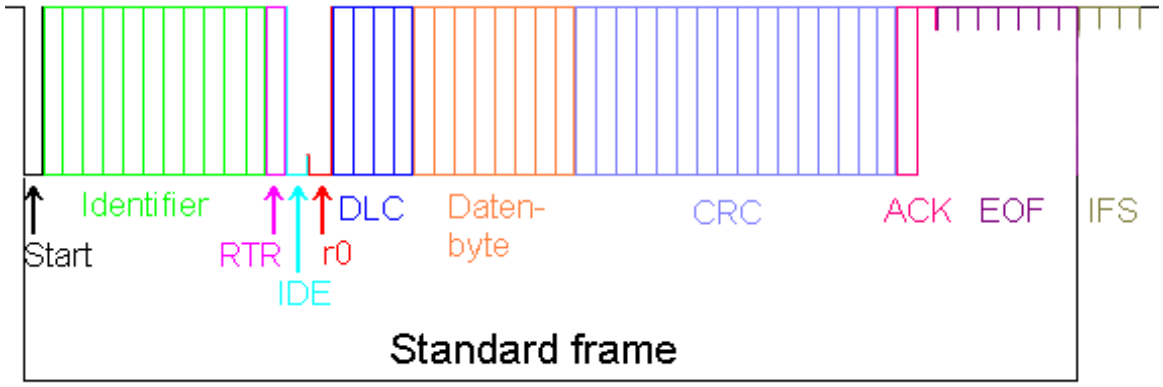


Abbildung 2: Standard frame

Startbit

Das Startbit kennzeichnet den Anfang eines Telegrammes. Es ist immer dominant.

Identifier

Dem Startbit folgen die 11 Bit des Identifiers. Sie geben die Priorität der Botschaft an.

RTR

Es schließt sich das Remote Transmission Request Bit RTR an. Bei Datentelegrammen ist es dominant, bei Remoteframes rezessiv.

IDE

Das nächste Bit wird als Identifier Extension Bit IDE bezeichnet. Es gibt an, ob die Nachricht ein Standard- oder Extended Frame ist. Bei Standard Frames ist dieses Bit dominant.

r0

Das dominante Bit r0 ist reserviert.

DLC

Die Länge des eigentlichen Nutzdatenfeldes in Bytes wird in den nächsten vier Bits übertragen. Obwohl hier Werte von 0 bis 15 möglich wären, sind definitionsgemäß Angaben über acht verboten.

Data

Dem Wert des DLC-Feldes entsprechend folgen 0 bis 8 Bytes, die die Nutzdaten des Telegramms enthalten.

CRC

Die nächsten 16 Bits sind zur Fehlererkennung vorgesehen. 15 Bits enthalten den CRC-Code (Cyclic Redundancy Check), Bit 16 ist ein rezessives Begrenzungsbit. Bei CAN dient die CRC-Prüfung ausschließlich zur Fehlererkennung, nicht zur Fehlerkorrektur.

ACK

Die nächsten zwei Bits sind der Acknowledge Slot ACK. Beide werden vom Sender einer Nachricht rezessiv gesetzt. *Jeder* Busteilnehmer der erfolgreich den CRC-Test ausgeführt hat, sendet während des ersten ACK-Bits ein dominantes Signal auf den Bus. Das zweite Bit bleibt rezessiv, um ein ACK vom Beginn eines Errorframes unterscheiden zu können. Die Bestätigung im ACK-Slot bedeutet nicht, dass die Nachricht von allen Teilnehmern empfangen wurde, die sie bekommen sollten. Es ist möglich, dass Bestätigungen nur von FullCAN-Controllern kommen, die die Nachricht nicht weiterleiten. Die Bedeutung des ACK liegt darin, dass ein Teilnehmer erkennen kann, ob es Controller gibt, die seine Telegramme verstehen.

EOF

Die Nachricht endet mit einer gezielten Verletzung des Bitstuffings. Das End of file EOF wird aus sieben rezessiven Bits gebildet.

IFS

Nicht mehr zur Nachricht gehören drei weitere rezessive Bits, der Inter Frame Space IFS. Dieser kennzeichnet den minimalen Abstand zwischen zwei Telegrammen und läßt den Controllern Zeit die decodierte Nachricht in den Empfangsspeicher zu schreiben.

2.5.3 Daten- und Remoteframes nach CAN 2.0B

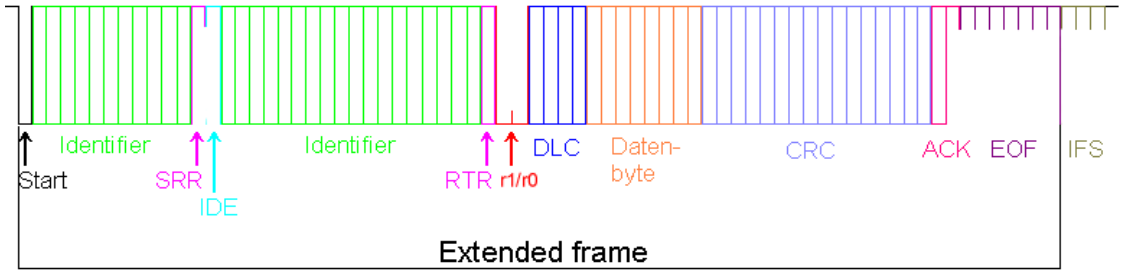


Abbildung 3: Extended Frame

Die Nachrichten nach CAN 2.0B entsprechen in ihrem prinzipiellen Aufbau den Standard Frames. Durch die Einführung der zusätzlichen 18 Bits für den verlängerten Identifier sind jedoch einige Veränderungen erforderlich.

SRR

Das rezessive Substitute Remote Request Bit SRR ersetzt das RTR Bit.

IDE und 18 Bit Identifier

Das IDE-Bit ist hier rezessiv und zeigt an, dass weitere 18 Identifier-Bits folgen.

RTR

Das RTR-Bit hat die gleiche Funktion wie bei Standard Frames, es ist bei Datenframes dominant und bei Remoteframes rezessiv.

r1

Nach dem reservierten dominanten Bit r1 haben die folgenden Bits beginnend mit r0 die gleiche Bedeutung wie bei Standard Frames.

2.5.4 Error- und Overloadframes

Error- und Overloadframe unterscheiden sich im Aufbau nicht. Sie rufen an beliebiger Stelle eine Verletzung der Stopfregel hervor, indem sie sechs dominante Bits in Folge enthalten.

Jeder Teilnehmer, der diese Verletzung erkannt hat, sendet seinerseits diesen Frame. Dadurch ergibt sich auf dem Bus eine Folge von zwölf dominanten Bits. Falls andere Teilnehmer den Fehler ebenfalls erkannt haben, kann sich diese Folge durch Überlappung der Errorframes auf bis zu sechs Bits reduzieren.

Weiterhin gehört zum Errorframe ein EOF aus acht rezessiven Bits. Im Anschluss daran überträgt der sendende Teilnehmer seine Nachricht erneut.

Da ständig wiederkehrende Errorframes den Bus blockieren können, verfügen alle Controller über einen in den CAN-Spezifikationen festgelegten Schutzmechanismus. Die Controller werden zunächst in den fehlerpassiven Status geschaltet und bei weiteren Fehlern ganz vom Busverkehr ausgeschlossen. Ein Controller der error passive ist sendet anstatt eines aktiven einen passiven Errorframe, der aus sechs rezessiven Bits und dem EOF besteht. Ein fehlerpassiver Controller kann aufgrund des rezessiven Pegels keine eingehenden Nachrichten beschädigen, jedoch von ihm selbst gesendete Telegramme für ungültig erklären.

Overloadframes haben eine andere Funktion als Errorframes, sie beginnen im ersten Bit des IFS. Sie sollen die Nachricht nicht für ungültig erklären, sondern die nächste hinauszögern. Der Overloadframe wird von allen anderen Teilnehmern mit einem eigenen Overloadframe beantwortet. Eine weitere Reaktion erfolgt jedoch nicht. Nach [etsch94] gibt es keinen Controller, der eine Verzögerungszeit braucht, so dass die Overloadframes nur theoretisch von Bedeutung sind.

2.5.5 Hierarchie der CAN-Nachrichten

Aus der Arbitrierungsregel und dem Aufbau der CAN-Nachrichten ergibt sich eine Hierarchie. Im folgenden Beispiel soll dieses verdeutlicht werden. Die Identifier mit kleinen Buchstaben enthalten 11 Bit, die mit Großbuchstaben gehören zu Extended Frames. Dabei entspricht a den elf höchstwertigen Bits von A. Identifier a hat höhere Priorität als b, in Zahlenwerten: $a < b$. Beispielsweise ist

$$
\begin{aligned}
A &= 00100100000\ 000000000000000000 \\
a &= 00100100000 \\
B &= 01100100000\ 000000000000000000 \\
b &= 01100100000
\end{aligned}
$$

Es ergibt sich folgende Hierarchie:

1. Standard Dataframe mit Identifier a
2. Standard Remoteframe mit Identifier a
3. Extended Dataframe mit Identifier A
4. Extended Remoteframe mit Identifier A
5. Standard Dataframe mit Identifier b
6. Standard Remoteframe mit Identifier b
7. Extended Dataframe mit Identifier B
8. Extended Remoteframe mit Identifier B

Für die Priorität einer Nachricht sind vor allem die elf höchstwertigen Bits des Identifiers ausschlaggebend. Ist dort keine Entscheidung gefallen, so setzt sich ein Standard Datenframe (RTR dominant) gegen ein Standard Remoteframe oder ein Extended Frame durch. Reicht auch dieses Merkmal nicht, so gewinnt im nächsten Schritt ein Standard Remoteframe (IDE dominant) gegen ein Extended Frame. Sind immer noch mehrere Nachrichten am Bus, so handelt es sich um Extended Frames. Zwischen ihnen entscheiden die unteren 18 Bit des Identifiers. Die letzte Entscheidung fällt zwischen Dataframe und Remoteframe mit gleichem Identifier. Zum Aufbau der Nachrichten siehe Abschnitte 2.5.2 und 2.5.3.

Falls es im gesamten Arbitrierungsbereich nicht zum Rückzug eines Teilnehmers kommt, wurde bei der Konfiguration des Netzwerkes irrtümlich der gleiche Identifier mehrfach vergeben. Sofern diese Teilnehmer nicht zufällig die gleichen Daten senden, wird es im Datenfeld zu einem Bitfehler kommen, weil ein Teilnehmer ein rezessives Bit auf den Bus gibt, aber das dominante Bit des anderen Teilnehmers registriert. Die Nachricht wird als fehlerhaft erkannt und ein Errorframe gesendet.

2.6 CAN-Hardware

2.6.1 Übertragungsmedien

High-Speed-Netze

Als Busmedium ist eine verdrillte oder geschirmte an beiden Enden mit dem Wellenwiderstand (120 Ω) abgeschlossene Zweidrahtleitung vorgesehen. Bei längeren Leitungen wird eine zusätzliche Signalmasse empfohlen. Der rezessive Pegel beträgt 2,5 V auf beiden Leitungen, der dominante Pegel 3,5 V für CAN_H und 1,5 V für CAN_L.

Von der CiA (Anwendervereinigung „CAN in Automation“) gibt es hierzu eine Ergänzung, die Sub-D-Stecker und eine mit dem Wellenwiderstand abgeschlossene Zweidrahtleitung mit gemeinsamer Rückleitung vorsieht. Eine zentrale Stromversorgungsleitung wird empfohlen, um Transceiver ohne DC/DC-Wandler galvanisch von der CPU getrennt anschließen zu können. Als Bitraten sollen 10, 20, 50, 100, 125, 250, 500 und 1000 kBaud vom CAN-Controller verstanden werden.

Low-Speed-Netze

Die Leitungslänge ist „elektrisch kurz“ zu halten, dafür entfallen Abschlusswiderstände und der Zwang zur Linientopologie. Es sind jedoch Leitungsabschlussnetzwerke zu installieren, die CAN_H auf 3,1 ... 3,4 V und CAN_L auf 1,6 ... 1,9 V Leerlaufspannung bei einem Innenwiderstand von $2,2$ kΩ bringen.

Sonstiges

Weitere Möglichkeiten sind die Nutzung von RS-485-Schnittstellen und optischen Medien. Da RS-485 die dominant/rezessive bitweise Arbitrierung nicht direkt unterstützt, muss dabei vom üblichen Verfahren etwas abgewichen werden. Näheres hierzu bei [etsch94].

2.6.2 Busansteuerung

Es gibt grundsätzlich zwei Möglichkeiten, um Hardware an einen CAN-Bus anzuschließen:

- Stand-alone-Controller, die über eine Schnittstelle mit einer beliebigen CPU verbunden werden. Üblich ist der direkte Anschluss an den Adress- und Datenbus, wobei

der CAN-Controller über eine Speicheradresse angesprochen wird (Memory Mapping). Diese Technik nutzt auch der für diesen Versuchsaufbau verwendete Philips SJA1000. Als Schnittstelle kommen außerdem Mikrocomputerbusse wie ISA und PCI in Frage, sowie auch serielle und parallele Schnittstellen und USB. Die für den CANalyzer genutzte CAN-AC2-PCI ist eine PCI-Karte für Personal Computer im Industriestandard, die ebenfalls den SJA1000 benutzt.

- Einige Mikrocontroller verfügen über integrierte CAN-Controller und sind somit für Mess-, Regel- und Steueraufgaben an CAN-Bussen besonders geeignet. Zu dieser Gruppe gehört der auf den Phytec miniMODULen genutzte Infineon (Siemens) 80C515C.

Sollen lediglich einfache Ein- und Ausgabefunktionen ohne Einsatz einer eigenen CPU ermöglicht werden, so ist ein SLIO die günstigste Wahl. Ein Serial Linked Input/Output-Modul verfügt lediglich über die Busansteuerung und Anschlüsse für einige digitale und/oder analoge Sensoren/Aktoren. Sie werden ausschließlich über den CAN-Bus konfiguriert. Es entfällt der Programmieraufwand. SLIOs können billiger sein als Stand-Alone-CAN-Controller.

Der direkte Anschluss der genannten Bausteine an den CAN-Bus ist zwar in vielen Fällen möglich, jedoch ist aus Gründen der Störsicherheit dazu zu raten, einen externen CAN-Transceiver vorzusehen, der den Ausgang des Controllers in die physikalischen Signale des Busses umsetzt und gleichzeitig die Buspegel in Logikpegel umwandelt und an den Controller weiterreicht. Zu dieser Gruppe von Bausteinen gehört der ebenfalls im Versuchsaufbau verwendete Philips 82C250.

2.7 Bestimmung der Bustiming-Parameter

Ein CAN-Bus kann auf beliebige Baudraten konfiguriert werden, die bei High-Speed-Bussen nach ISO 11898 innerhalb von drei Größenordnungen wählbar sind. Ein CAN-Controller kann bei der Konfiguration die Baudrate jedoch nicht direkt entgegennehmen, die erforderliche Einstellung muss in Form von zwei Bytes an die Bus Timing Register BTR übergeben werden. Die BTR sind bei allen Controllern weitgehend ähnlich aufgebaut, es gibt im Detail jedoch Unterschiede, die zu beachten sind. In die Berechnung der Bustiming-Parameter gehen folgende Werte ein:

- die Clock Rate des Controllers
- die Toleranz des Oszillators, der den Controllertakt erzeugt
- die gewünschte Baudrate des CAN-Busses
- die Signalverzögerungen in der Busansteuerung
- die Buslänge
- das Abtastverhalten des CAN-Controllers (einfache oder dreifache Busabtastung)

Eine sehr ausführliche Beschreibung der Berechnung und der technischen Hintergründe wird für den Philips SJA1000 in der Applikation AN97046 vorgestellt. Das Dokument ist im Anhang aufgeführt.

Siemens liefert für den on-Chip-CAN-Controller des 80C515C ein Hilfsprogramm zur Bestimmung der BTR-Bytes. Das Programm CP_515C für Windows ab 3.1 auf MS-DOS 6.22 sowie die erklärende Applikation sind ebenfalls im Anhang wiedergegeben.

Beide Rechenmethoden verlangen unterschiedliche Eingabewerte, weil sie auf verschiedenen Ebenen der Berechnung ansetzen. Philips legt die physikalischen Daten des Busses zugrunde, Siemens geht davon aus, dass der Abtastzeitpunkt bezogen auf die Bitlänge bekannt ist.

SLIO-Module lassen sich nur über den Bus konfigurieren und sind somit nicht auf die beschriebene Art an die Baudrate anzupassen. Sie werden für Übertragungen zwischen 20 und 125 kBit/s mit Telegrammen synchronisiert, die ein quarzgetakteter Teilnehmer regelmäßig senden muss. Andere SLIOs nutzen eine feste Übertragungsrate, die bis 1 MBit/s betragen kann.

3 Das Vector CANalyzer-Paket

3.1 Allgemeines

Der CANalyzer ist ein universelles Entwicklungswerkzeug für CAN-Bussysteme mit dessen Hilfe der Datenverkehr auf der Busleitung beobachtet, analysiert und ergänzt werden kann. Der Einsatzbereich erstreckt sich dabei vom Test einer ersten Businstallation bis zur Fehlersuche bei größeren Anwendungen.

Die auf dem Bus übertragenen Daten können wahlweise im Rohformat (dezimal oder hexadezimal) ausgegeben oder bei zuvor definierten Nachrichten auch in ein physikalisches Format umgerechnet werden.

Das Senden von Nachrichten geschieht in vorgegebenen Zeitintervallen, auf Tastendruck oder als Antwort auf empfangene Nachrichten, wahlweise auch zeitverzögert. Reichen diese Funktionen nicht aus, so besteht die Möglichkeit, mittels einer C-ähnlichen Applikationssprache komplexe Aufgaben zu programmieren.

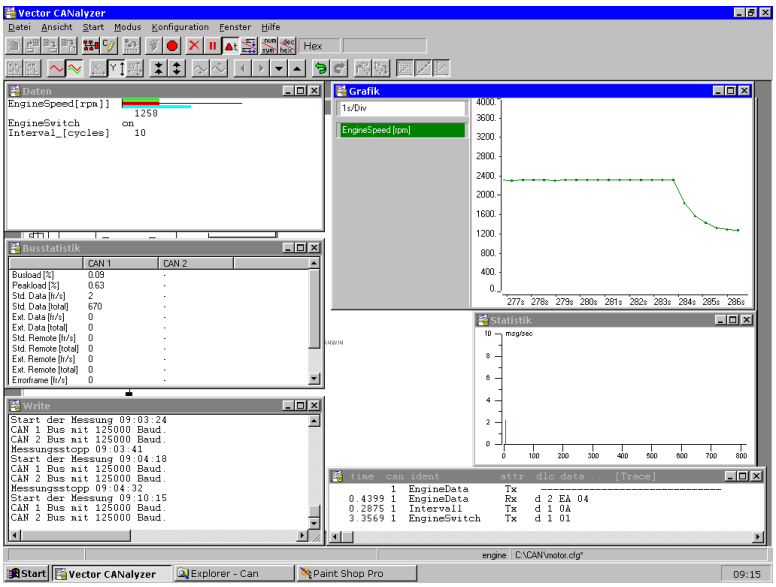


Abbildung 4: Eine Messung mit dem CANalyzer

3.2 Systemvoraussetzungen

- 100% IBM-kompatibel
- Pentium-100-Prozessor oder besser
- 32 MB Arbeitsspeicher oder mehr
- Windows 9x oder NT
- 16 MByte freier Festplattenspeicher
- freier PCI-Slot

3.3 Die CAN-AC2-PCI-Karte zum CANalyzer

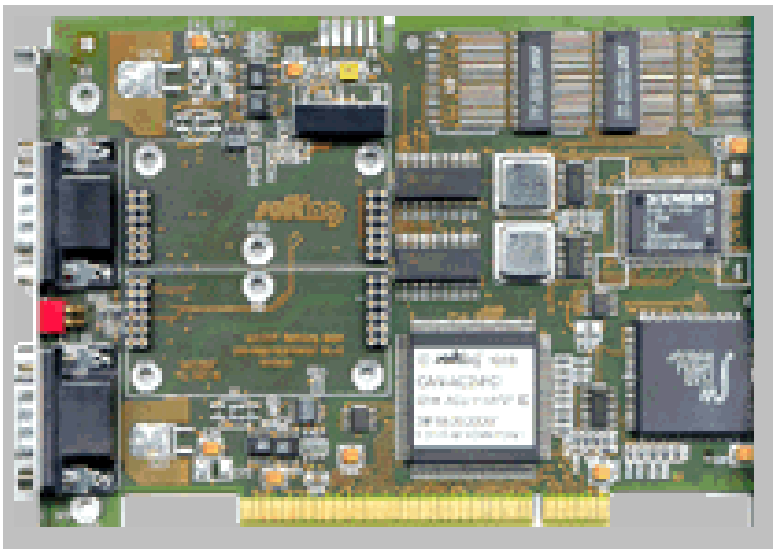


Abbildung 5: Die CAN-AC2-PCI-Karte

3.3.1 Allgemeines

Die CAN-AC2-PCI (Controller Area Network Application Controller 2 for Peripheral Component Interconnection) PC-Steckkarte dient zur Kommunikation des CANalyzers mit dem Bus. Sie muss also in den Computer eingebaut werden, auf dem der CANalyzer als Softwaretool installiert wird. Die CAN-Einsteckkarte besitzt zwei voneinander unabhängige Busteilnehmer, die sowohl an zwei getrennte als auch an nur einem Netzwerk angeschlossen werden können. Es besteht außerdem die Möglichkeit nur einen der Teilnehmer anzuschließen. Diese und weitere Einstellungen für die Karte werden im CANalyzer

konfiguriert. Soll die Karte nicht an einen vorhandenen Bus angeschlossen werden, kann ein CAN-Bus mit dem beiliegenden kurzen Kabel simuliert werden. Dazu werden die beiden Ports der CAN-AC2-PCI-Karte mit diesem verbunden.

Belegung des beigefügten Kabels zur Simulation des Busses:

	D-SUB9 CAN1		D-SUB9 CAN2
CAN_L	Pin 2	———	Pin 2
GND	Pin 3	———	Pin 3
CAN_H	Pin 7	———	Pin 7

Zwischen CAN_H und CAN_L ist an beiden Enden des Kabels ein Abschlusswiderstand eingebaut.

Mit den DIP-Schaltern an der Karte kann für jeden Port ein Abschlusswiderstand von 120 Ω zwischen CAN_H und CAN_L eingeschaltet werden, so dass gegebenenfalls andere Kabel verwendet werden können.

3.3.2 Installation der CAN-AC2-PCI-Treiber/

Installation von Diskette Falls bereits eine frühere CAN-AC2-PCI Version als V4.05 installiert ist, müssen die alten Treiber zunächst entfernt werden. Zur Installation der Software wird von der beiliegenden CAN-AC2-PCI L2 V4.05-Diskette die Datei *CAN-AC2-PCI V4.05Setup.EXE* gestartet. Nach Aufruf dieser Datei erfolgt die Installation menügeführt. Ist die Softwareinstallation abgeschlossen, wird die PCI-Einsteckkarte bei ausgeschaltetem PC in einen freien PCI-Slot gesteckt und der Computer neu gebootet. Falls der Installationsassistent eine Treiberdiskette verlangt, muss die CAN-AC2-PCI L2 V4.05-Diskette eingelegt und deren Pfad angegeben werden. Die Installation verläuft danach automatisch.

Zur Kontrolle der Installation wird der Bus beziehungsweise das beigefügte Kabel an die beiden Ports der Karte angeschlossen und die *Can_test.exe* im Verzeichnis *win32* der aufgespielten Software gestartet und *FIFO* und *Polling* gewählt. Nach erfolgreicher Initialisierung (*Chip is running*) bitte *h* für Hilfe oder *t* für Senden einer CAN-Test-Nachricht eingeben.

Bei erfolgreicher Übertragung der Testnachricht werden folgende Zeilen auf dem Bildschirm ausgegeben:

```
DPRAM chosen in FIFO mode.
Initialization successfull
Chip is running.

XMT     CAN1 Id       1 Dlc7 Data 1  3  5  7  9  b  d  f return Code 0
RCV STD CAN2 Id       1 Dlc7 Data 1  3  5  7  9  b  d  0 T  c80b95 0  c80b95
ACK STD CAN1 Id       1 Dlc7 Data 1  3  5  7  9  b  d  0 T  c80bab 0      16
```

Installation von der CANalyzer-CD Die PCI-Einsteckkarte wird bei ausgeschaltetem PC in einen freien PCI-Slot gesteckt und der Computer neu gebootet. Windows zeigt an, dass eine neue Hardware gefunden wurde. Nach Auswahl von *weiter* und *andere Position* ist die CANalyzer-CD einzulegen und der Pfad *Drivers\CANac2_PCI* anzugeben. Zur Überprüfung der Installation ist unter der Gerätegruppe PCI (*Start/Einstellungen/Systemsteuerung/System*) der Eintrag *CAN-AC2-PCI* doppelt anzuklicken. Nach erfolgreicher Installation ist auf der Karteikarte *Allgemein* der Gerätestatus *Dieses Gerät ist betriebsbereit* angegeben.

Zur Kontrolle der Installation wird der Bus beziehungsweise das beigefügte Kabel an die beiden Ports der Karte angeschlossen und das Programm *Can_test.exe* im Verzeichnis *Drivers\CANac2_PCI\HWTest* gestartet und *FIFO* und *Interrupt* gewählt. Die erfolgreiche Initialisierung der Hardware wird über die Meldung

Initialisation successful

Chip is running

angezeigt. Mit der Taste *t* können CAN-Botschaften versendet werden.

```
DPRAM chosen in FIFO mode.
Initialization successfull
Chip is running.

XMT     CAN1 Id       1 Dlc7 Data 1  3  5  7  9  b  d  f return Code 0
RCV STD CAN2 Id       1 Dlc7 Data 1  3  5  7  9  b  d  0 T 1d2129d D 1d2129d
ACK STD CAN1 Id       1 Dlc7 Data 1  3  5  7  9  b  d  0 T 1d212b2 D      15
```

3.4 Installation des CANalyzers

Zur Installation des CANalyzers muss das Programm setup.exe aus dem Verzeichnis *setup\disk1* der CANalyzer-Installations-CD gestartet werden. Das Installationsprogramm verläuft menügesteuert und ist selbsterklärend.

Es wird empfohlen, die Standardinstallation zu wählen.

Zum Starten des CANalyzer-Paketes muss im CANalyzer-Verzeichnis unter *\Exec32* die *CANW32.EXE* aufgerufen werden.

3.5 Allgemeine Parameter

Für den Betrieb des CANalyzers ist es erforderlich, einige grundlegende Einstellungen vorzunehmen. Der CANalyzer wurde auf die Bedürfnisse dieser Diplomarbeit eingerichtet, die hier beschriebene Konfiguration stellt somit nur eine von mehreren möglichen dar.

In der Menüleiste wird unter *Konfiguration/CAN Kanäle* das *Kanäle*-Fenster geöffnet. Hier ist die Anzahl der zu verwendenden Kanäle im Pulldown-Menü auf zwei zu stellen (*CAN 2*), damit die beiden Ports der eingebauten CAN-AC2-PCI-Karte freigeschaltet werden und Nachrichten gesendet und empfangen werden können.

Unter dem Menüpunkt *Konfiguration/CAN Busparameter* erscheint das Fenster *Kanal Konfiguration*, hier werden Baudrate (Abbildung 6) und Akzeptanzfilter (Abbildung 7) gesetzt. Für die Baudrate wird unter dem Item eines der CAN-Kanäle das Untermenü *Setup* aufgerufen und in das Feld *Baudrate* 125 eingegeben. Das Übernehmen der Einstellung für den anderen Kanal ist zu bestätigen. Selbstverständlich können auch andere Baudraten gewählt werden. Dabei muss beachtet werden, dass die Teilnehmer des gesamten Busses auf die gleiche Geschwindigkeit gestellt werden, da es sonst zu Fehlern kommt.

Da jede eintreffende Nachricht verarbeitet werden soll, wird als *Akzeptanzfilter für Standard Botschaften* XXXXXXXXXXX (11 X) eingegeben. Das X bedeutet, dass der Zustand des jeweiligen Bits nicht überprüft wird.

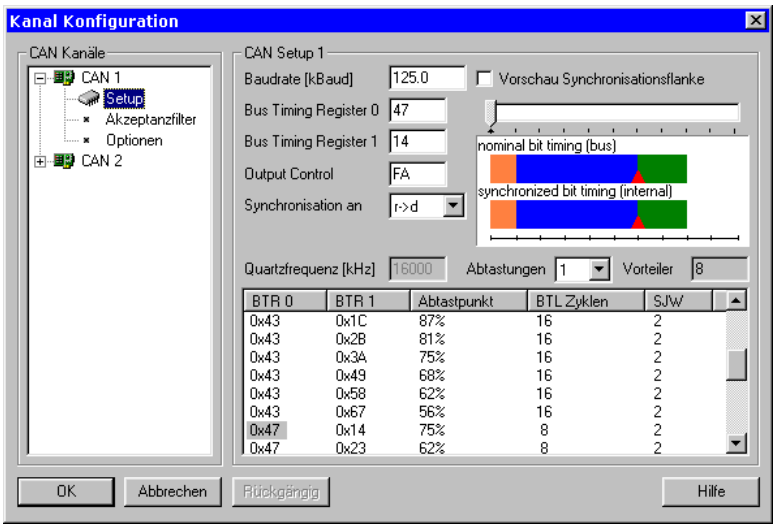


Abbildung 6: Baudrateneinstellung im CANalyzer

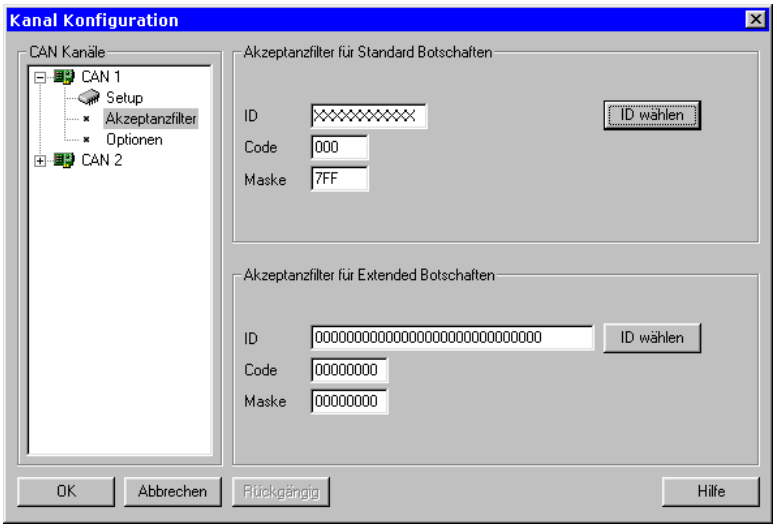


Abbildung 7: Einstellen der Akzeptanzfilter

3.6 Konfiguration des Messaufbaus

Zur Konfiguration des Messaufbaus ist es erforderlich, das Fenster *Meßaufbau* zu öffnen. Falls es nicht sichtbar ist, kann es durch *Ansicht/Meßaufbau* in den Vordergrund geholt werden.

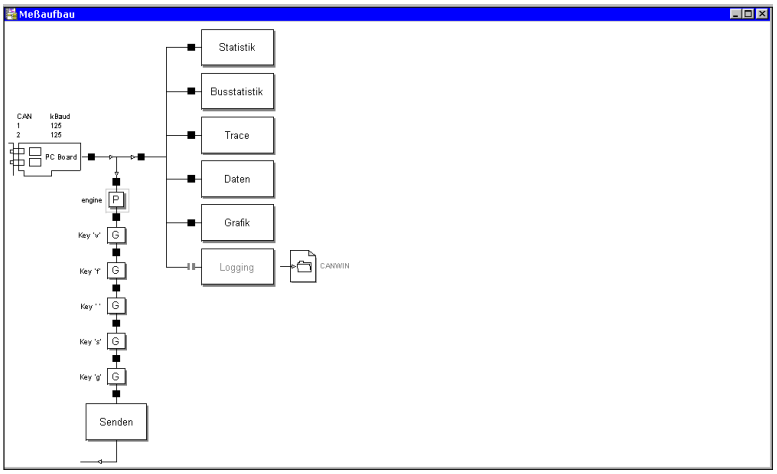


Abbildung 8: Messaufbau im CANalyzer

3.6.1 Konfiguration einer Nachricht

Damit auf Tastendruck eine Nachricht gesendet wird, müssen Generatoren in den Messaufbau eingefügt werden. Durch einen rechten Mausklick auf das Kästchen vor dem Block Senden öffnet sich ein Kontextmenü (Abbildung 9).

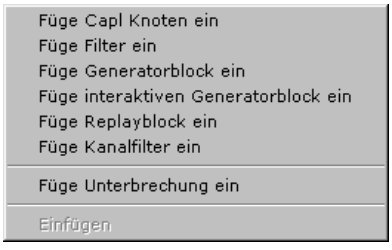


Abbildung 9: Kontextmenü eines Hot-Spots

Auf Anwahl von *Füge Generatorblock* ein wird in den Sendestrang ein Generatorblock eingebunden, der konfiguriert werden muss. Mit einem Rechtsklick auf den Generatorblock öffnet sich ein weiteres Kontextmenü (Abbildung 10).

Abbildung 10: Kontextmenü eines Generatorblockes

Unter *Konfiguration Auslösung* erscheint das Fenster Generator Auslösung. Nach Aktivierung von *Auf Taste* wird der Buchstabe auf *s* geändert.

Abbildung 11: Auslösen einer Nachricht im CANalyzer

Die Taste *s* soll einen Motor starten beziehungsweise stoppen, was ein Teilnehmer, der sich ebenfalls am Bus befindet, übernimmt. Er reagiert auf Nachrichten mit dem Identifier 0x02. Im Datenpaket werden zwei verschiedene Datenblöcke übermittelt, aus denen der Teilnehmer erkennt, ob der Motor an- oder ausgeschaltet werden soll. Die Konfiguration dieses Datentelegramms erfolgt über *Konfiguration Sendeliste* im Kontextmenü des Generatorblocks. Hier wird in die erste Zeile als Identifier (*ID*) die *2*, als *DLC 1* für ein Datenbyte und als 1. Datenbyte (*DATA*) *1* für Start eingegeben. Diese Eingaben wiederholen sich für Zeile 2, mit der Ausnahme, dass im Datenbyte anstatt der 1 eine *0* für Motor Stop eingetragen wird. Die Sendeliste wird nun mit Betätigung der Taste *s* abgearbeitet, dabei wird pro Tastendruck die Nachricht in eine Zeile der Sendeliste übertragen. Wenn alle Zeilen durchlaufen wurden, wird beim nächsten Auslösen wieder mit Zeile eins begonnen.

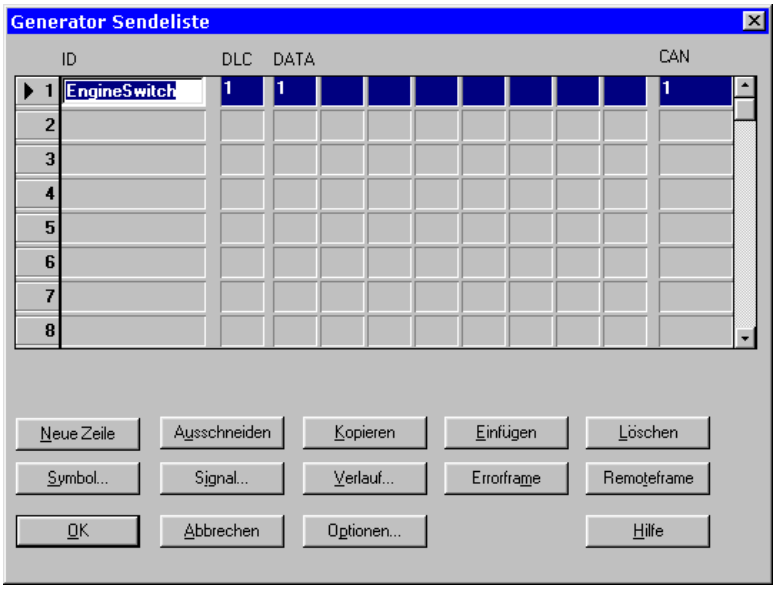


Abbildung 12: Sendeliste im CANalyzer

Nach dem gleichen Prinzip werden weitere Generatoren in den Sendestrang eingefügt deren Inhalte aus Tabelle 1 erkennbar sind. *Remoteframe* und *Errorframe* sind über Buttons auszuwählen.

Andere Arten die Generatoren auszulösen sind der periodische Aufruf (Intervall) oder die Reaktion auf eine empfangene Botschaft, die in diesem Beispiel nicht zum Einsatz kommen.

Taste	Funktion	ID	DLC	Data
s	Start/Stop	2 2	1 1	0 1
Leertaste	Remoteframe	10	R2	–
e	Errorframe	–	–	–

Tabelle 1: Generatoreinträge

3.6.2 Definition der Netzknoten und Botschaften

Zur Vervollständigung des Messaufbaus wird ein Programm eingefügt, das die Auflösung eines Busoff-Zustandes sowie die Intervallregelung der Datenübertragung für die Drehzahl des Motors übernimmt.

Im Programm wird die Nachricht *Intervall* benutzt. Da diese nicht bekannt ist, muss sie vorher definiert werden. Unter *Datei/CANdb Editor öffnen* wird der CAN-Datenbasis

Editor gestartet. Mit *Datei/Neu* wird eine neue Datenbasis erstellt. Zunächst werden die Netzknoten definiert, indem mit *Ansicht/Netzknoten* die Liste der Netzknoten aufgerufen wird und mit *Bearbeiten/Neuer Netzknoten* folgende Teilnehmer konfiguriert werden:

CANalyzer	– der PC mit dem CANalyzer
EngineControl	– das miniMODUL zur Motorkontrolle
Display	– das miniModul zur Ansteuerung eines LCD-Displays
Troublemaker	– die DIDO537 mit der Fehlererzeugung

Tabelle 2: Busteilnehmer des Demonstrationsaufbaus

Nach Aufruf von *Ansicht/Botschaften* wird unter *Bearbeiten/Neue Botschaft* das Fenster *Botschafts- und Signalparameter* geöffnet. Die Einstellungen sind den folgenden Abbildungen zu entnehmen und als *engine.dbc* abzuspeichern. Für EngineSwitch sind außerdem die Werte nach Abbildung 16 einzugeben. Der CAN-Datenbasis Editor kann anschließend geschlossen werden.

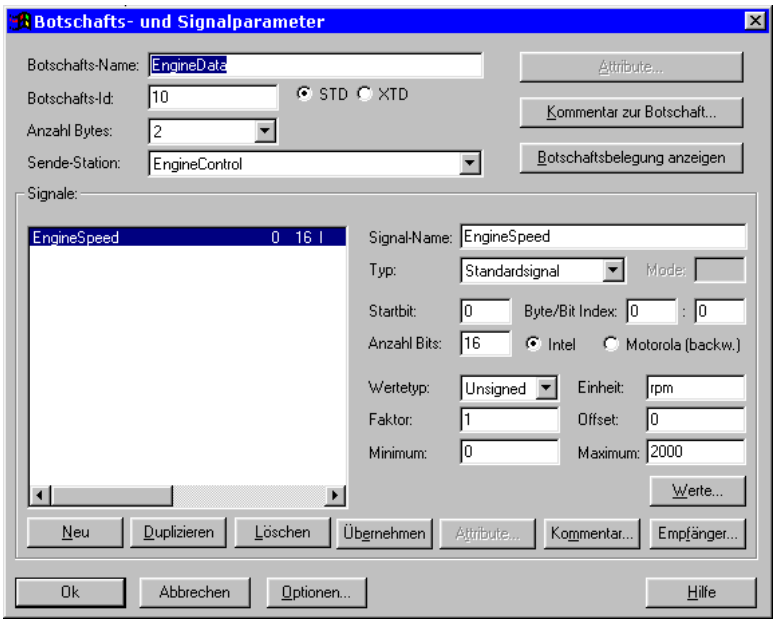


Abbildung 13: Botschafts- und Signalparameter EngineData

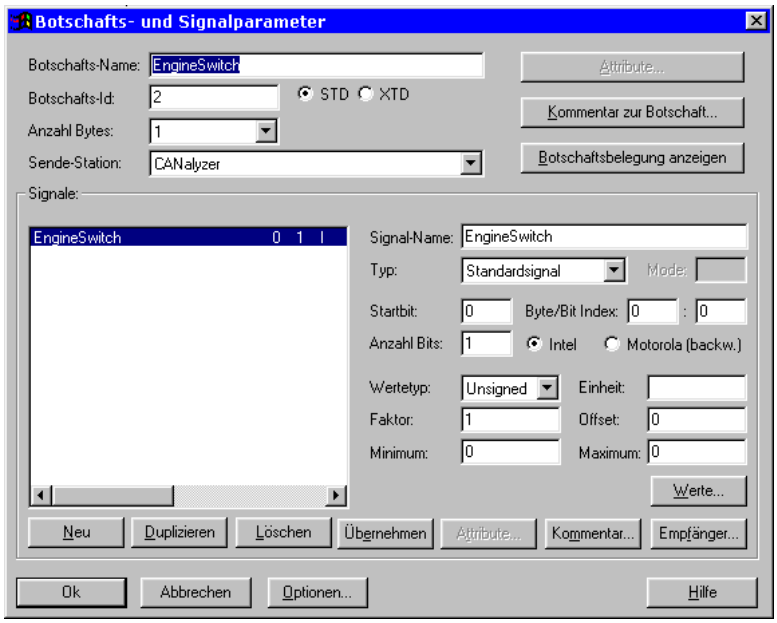


Abbildung 14: Botschafts- und Signalparameter EngineSwitch

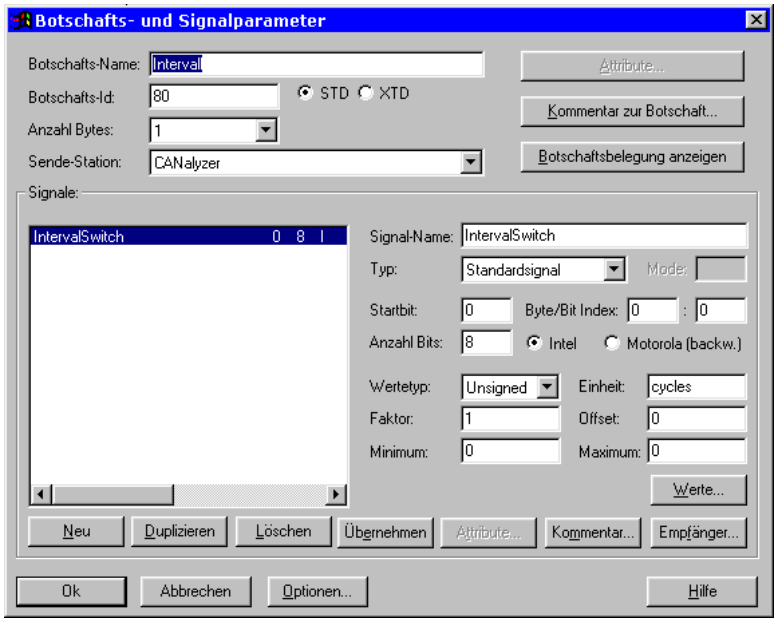


Abbildung 15: Botschafts- und Signalparameter Intervall

Abbildung 16: Wertezuweisung EngineSwitch

3.6.3 Erstellung eines Programmblockes

Im Sendezweig des Messaufbaus wird das Kontextmenü des ersten Knotens aufgerufen und *Füge Capl Knoten ein* gewählt. Die Programmierung erfolgt über einen Doppelklick auf den Programmblock. Es öffnet sich der CAPL Browser, in dem mit *Daten/Neu* oder Auswahl des entsprechenden Symbols ein neues Dokument zu erstellen ist. Im oberen rechten der vier Felder ist folgendes Programm einzugeben:

```
variables {
      int counter;
      message Intervall msg;
}

on key '+' {
      if (counter < 254) counter++;
      msg.BYTE (0) = counter;
      output (msg);
}

on key '*' {
      if (counter < 245) counter += 10;
      msg.BYTE (0) = counter;
      output (msg);
}
```

```
on key '-' {
      if (counter > 0) counter-;
      msg.BYTE (0) = counter;
      output (msg);
}


on key '_' {
      if (counter > 9) counter -= 10;
      msg.BYTE (0) = counter;
      output (msg);
}


on busOff {
      write ("CAN Controller wird neu gestartet");
      resetCAN ();
}
```

Im Anschluss an die Eingabe ist mit *Datei/Datenbasis zuordnen* die *engine.dbc* anzuwählen. Zur Kompilierung des Programms wird *Compiler/Compilieren* aufgerufen, danach kann der CAPL Browser geschlossen werden.

3.7 Konfiguration des Datenfensters

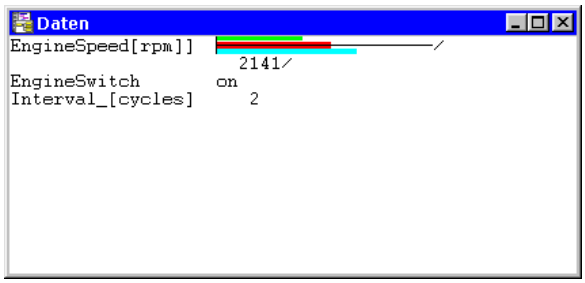


Abbildung 17: Datenfenster im CANalyzer

Im Kontextmenü des Daten-Blocks des Messaufbaus wird *Konfiguration* gewählt. Für die Aufnahme neuer Signale wird der Button *Neues Signal* angeklickt. Es öffnet sich das Fenster *Auswahl von Signalen*. Dort werden die benötigten Signale durch Doppelklicken markiert. Die verwendeten Signale sind der Tabelle zu entnehmen. Bei EngineSpeed wird, wie in Abbildung 18 dargestellt, der *Anzeigetyp* einmal auf Balken geändert.

Signalname	Anzeigetyp	Position Name	Position Werte
EngineSpeed	physikalisch	1,1	2,20
EngineSwitch	symbolisch	3,1	3,20
Intervall	physikalisch	4,1	4,20

Parameter für die anzuzeigenden Signale

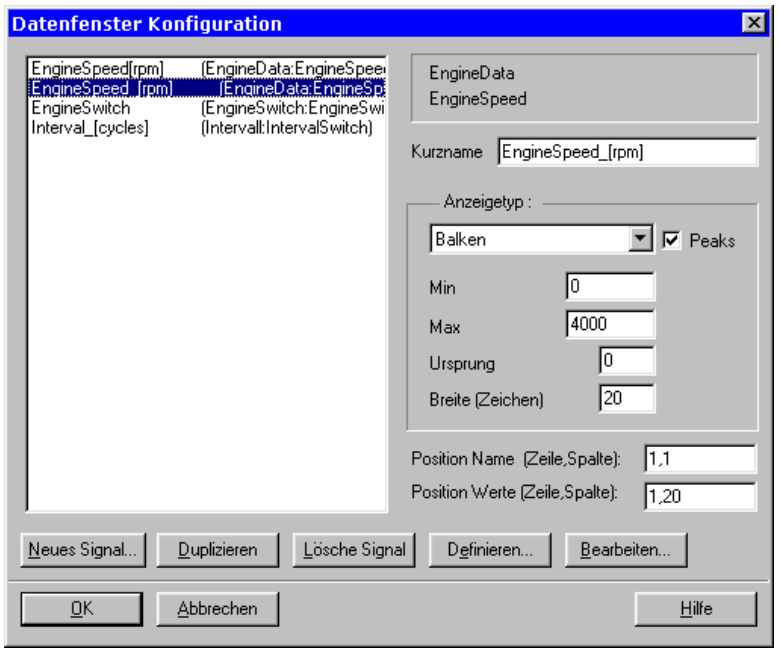


Abbildung 18: Konfiguration des Datenfensters

3.8 Konfiguration des Grafikfensters

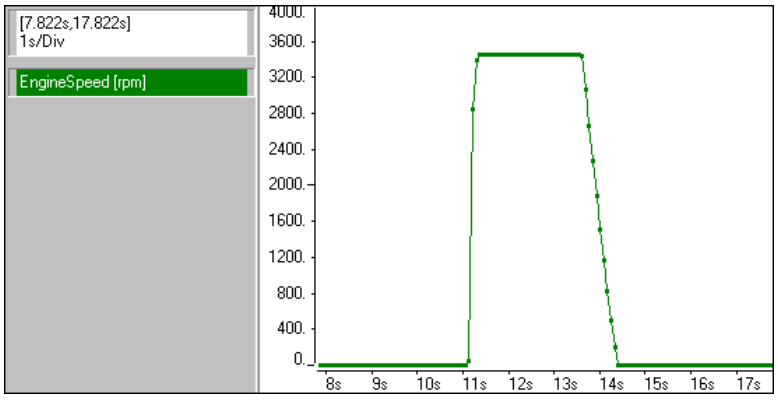


Abbildung 19: Grafikfenster im CANalyzer

Zur Konfiguration des Grafikfensters wird im Kontextmenü des Grafik-Blocks des Messaufbaus *Konfiguration* gewählt. In gleicher Weise wie beim Datenfenster wird hier das

Signal *EngineSpeed* eingefügt. Für die Einstellung des Anzeigebereichs muss im Kontextmenü des Grafikfeldes im Grafikfenster *Einstellungen* angewählt werden. Die zu tätigenden Eingaben sind Abbildung 20 zu entnehmen.

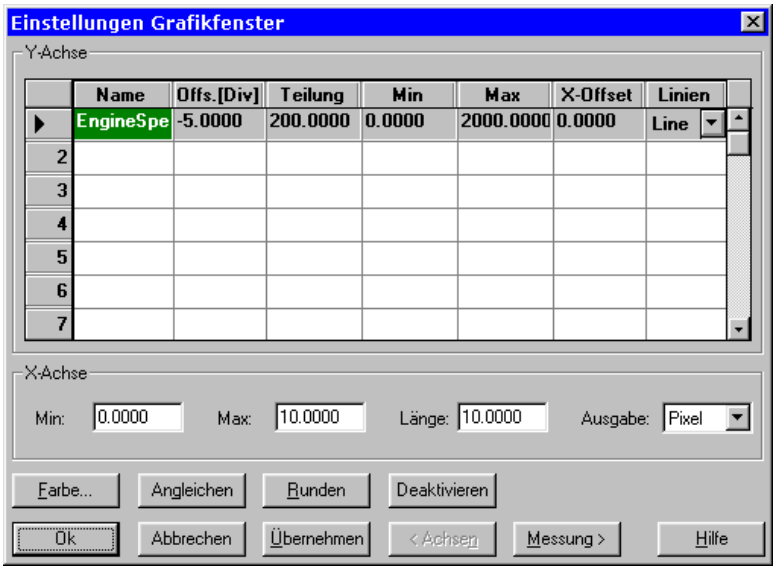


Abbildung 20: Einstellungen für das Grafikfenster

3.9 Das Statistikfenster

Die Konfiguration des Statistikfensters muss für dieses Beispiel nicht verändert werden.

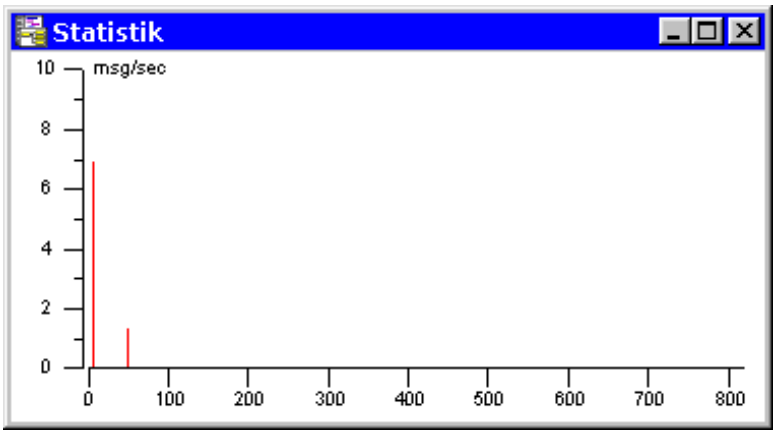


Abbildung 21: Statistikfenster im CANalyzer

3.10 Konfiguration des Tracefensters

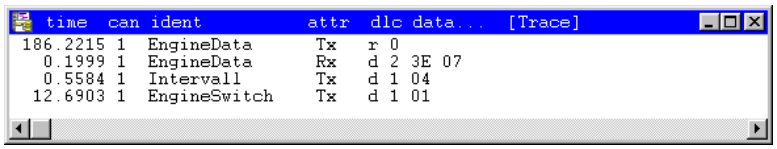


Abbildung 22: Tracefenster im CANalyzer

Im Kontextmenü des Trace-Blocks des Messaufbaus wird *Konfiguration* gewählt und die Einstellungen wie in Abbildung 23 vorgenommen.

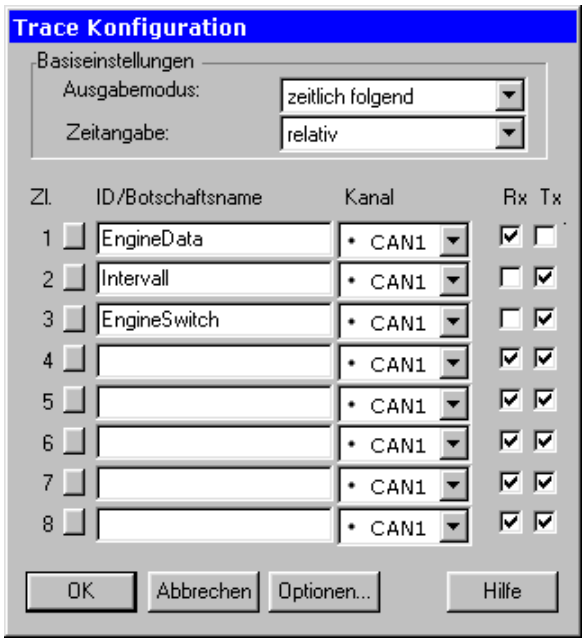


Abbildung 23: Konfiguration des Tracefensters

3.11 Logging

Die Loggingfunktion des CANalyers wird bei diesem Beispiel nicht genutzt, die Daten werden nicht in eine Datei geschrieben, sondern nur direkt auf dem Bildschirm ausgegeben.

3.12 Writefenster und Busstatistikfenster

Für beide Fenster können aus dem Kontextmenü die Schriftarten geändert werden. Beim Busstatistikfenster kann des weiteren die *Refresh Rate* konfiguriert werden, hier wird die Standardeinstellung genutzt.

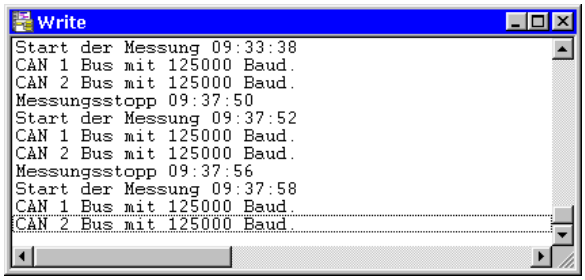


Abbildung 24: Writefenster im CANalyzer

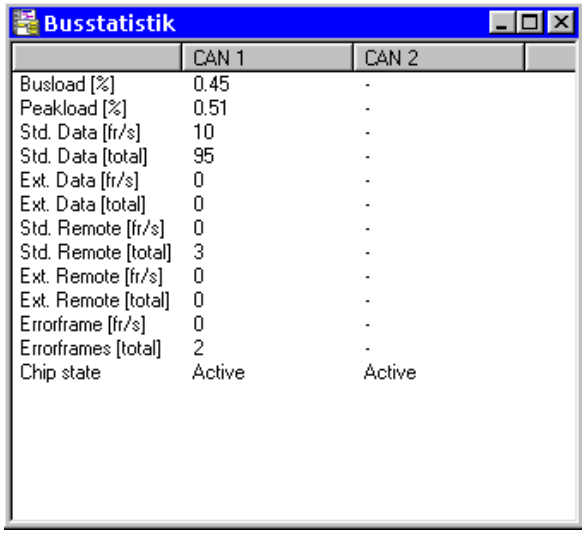


Abbildung 25: Busstatistikfenster im CANalyzer

4 Phytec Rapid Development Kit

4.1 Allgemein

Phytec liefert mit den Rapid Development Kits eine Kombination aus einem Mikrocontrollermodul, einer Basisplatine sowie Spannungsversorgung und Programmiertools. Für diese Diplomarbeit werden zwei dieser Kits mit dem miniMODUL-515C eingesetzt, die über einen Infineon 80C515C mit on-board-CAN-Controller verfügen.

Durch Verwendung eines Flash-ROMs bleibt das geladene Programm erhalten und wird bei jedem Neustart des Controllers erneut ausgeführt.

4.2 Anschluss der Spannungsversorgung

Es gibt prinzipiell zwei verschiedene Möglichkeiten, die Versorgungsspannung an die Basisplatine anzuschließen:

- Anschluss über die VG96-Leiste VG1
- Anschluss über Kleinspannungsbuchse P3

Die erste Möglichkeit ist dabei nur interessant, wenn ein nachgekauftes miniMODUL ohne Basisplatine eingesetzt oder eine Basisplatine in einen 19"-Einschub mit eigener Spannungsversorgung eingebaut werden soll, da die Rapid Development Kits standardmäßig mit einem Netzgerät zum Anschluss an die Kleinspannungsbuchse geliefert werden.

4.2.1 Anschluß über die VG96-Leiste VG1

Für den Anschluß über die VG96-Leiste muss der Jumper JP1 die Stellung 1+2 haben. Hierzu ist eine geregelte Versorgungsspannung von +5 V/500 mA wie folgt anzulegen: Pin 1abc +5 V geregelt, Pin 32abc GND. Die anderen Pins der VG96-Leiste sind nicht verbunden und stehen zur freien Beschaltung zur Verfügung.

> **Labornetzteile dürfen hier nicht verwendet werden, da die Einschaltspitzen das eingesetzte Modul zerstören können.**
>
> **Außerdem darf bei anliegender Spannung weder das Modul gewechselt noch die Jumperbelegung verändert werden!**

4.2.2 Anschluß über Kleinspannungsbuchse P3

Zulässiger Spannungsbereich +8 ... 13 V/500 mA ungeregelt. Für den Anschluß über die Kleinspannungsbuchse P3 muss der Jumper JP1 die Stellung 2+3 haben. Bei Spannungseinspeisung über die Kleinspannungsbuchse liegen keine geregelten +5 V an der VG96-Leiste. Die Spannungsregelung mit dem auf dem Basismodul eingebauten MA7505 stellt einen Strom von 300 mA zur Verfügung. Der erforderliche Stecker für die Kleinspannungsbuchse hat einen Durchmesser von 5,5 mm mit einer Bohrung von 2 mm, Masse liegt außen.

5 Die Mikrocontroller 80C537 und 80C515C

Bei den Siemens 80C537 und 80C515C handelt es sich um Einchip-Mikrocontroller aus der 8-Bit-Familie des 8051. Sie verfügen über 8-Bit-A/D-Wandler, 16-Bit Timer/Zähler sowie mehrere digitale Ein-/Ausgabemöglichkeiten, Watchdogs und ein Interruptsystem. Ihr Speicher ist variabel konfigurierbar.

Der 80C515C hat überdies einen integrierten CAN-Controller, der volle Funktionalität nach CAN 2.0B bietet.

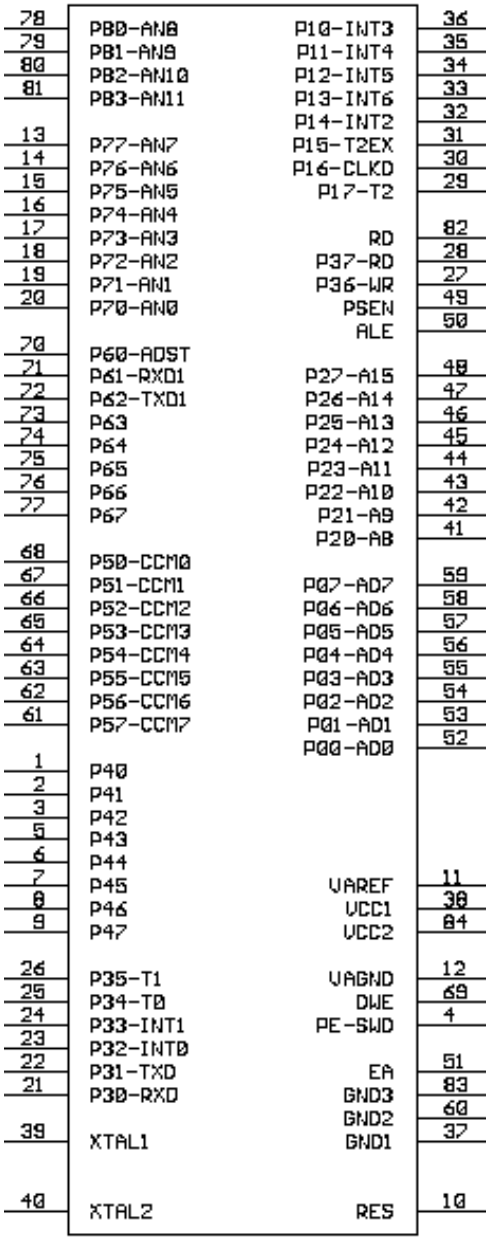


Abbildung 26: Der 80C537

Teil II

CAN-Bus mit Fehlererzeugung und Analyse

6 Buskabel

Die einzelnen Teilnehmer des CAN-Busses werden über eine Leitung verbunden. Diese hat eine Länge von 1,5 m und ist für maximal vier Teilnehmer ausgelegt. An der Busleitung befinden sich jeweils neunpolige SUB-D-Stecker. Die Belegung der Stecker ist wie folgt definiert:

1	NC	
2	CAN_L	grün
3	GND	blau
4	NC	
5	Drain	rot
6	GND	schwarz
7	CAN_H	gelb
8	NC	
9	NC	

Tabelle 3: Steckerbelegung CAN-Kabel

7 Software für den 80C515C

7.1 Zielsetzung

Es sollte eine Möglichkeit gefunden werden, über den CAN-Bus zu kommunizieren, ohne selbst im Einzelnen die nötigen Einstellungen programmieren zu müssen. Die Nutzung des Busses sollte einfach über Aufruf von geeigneten Funktionen die mit *can_[Funktionsname]* bezeichnet sind möglich sein. Als Programmiersprache wird C genutzt, als Compiler der Keil C51.

7.2 Allgemeines

Die erstellte CAN-Funktionsbibliothek ist für viele Aufgabenstellungen verwendbar. In einigen Sonderfällen wird es allerdings trotzdem erforderlich sein, selbst Ergänzungen zu programmieren, insbesondere bei

- Verwendung von CAN-Interrupts,
- Arbeit mit Identifiern nach CAN2.0B (29 Bit),
- Nutzung anderer als der vorgegebenen Baudraten und
- Timingproblemen durch besondere Netztopologie.

Ein Rahmen für die Interruptverarbeitung ist vorhanden, muss bei Bedarf jedoch eingeschaltet und um die gewünschten Funktionen ergänzt werden. Für weitere Baudraten sind die Bus Timing Parameter gemäß Abschnitt 2.7 zu ermitteln. Weiterhin ist eine zusätzliche Abfrage des Übergabewertes zu erstellen, die entsprechend angepasst wird. Für die Verwendung von 29 Bit langen Identifiern nach CAN2.0B können die meisten Funktionen übernommen werden. In *can_db* und *can_rcv* ist das XTD-Bit (extended), in *can_ar* und *can_mlm* sind die „lower“ Teile der Identifier (Bit 0-17) zu setzen. Eine Ausnahme bildet *can_gms*, da die „Global Mask Short“ weiterhin als Eingangsfilter für Nachrichten mit 11–Bit–Identifier arbeitet. Hier sind keine Änderungen erforderlich. Für 29–Bit–Identifier gibt es den besonderen Eingangsfilter „Global Mask Long“.

Bis auf die generellen Einstellungen sind alle Funktionen für die 15 Message Object Register verfügbar. Welches von ihnen angesprochen werden soll, ist mit einem Übergabeparameter anzugeben. Eine Sonderrolle spielt Register 15, das nur zum Empfang konfiguriert werden kann, aber eine eigene Filtermaske besitzt.

Auf die Rückgabe von Fehlermeldungen wurde verzichtet, ersatzweise sind die Routinen weitgehend tolerant gegenüber falschen Parametern. Der Busstatus ist im Bedarfsfall vom Programm abzufragen.

7.3 Funktionsbeschreibungen

void can_init (unsigned short Baudrate)
ist die zunächst aufzurufende Routine. Sie schaltet alle 15 Register aus und setzt die Baudrate, die in kBaud übergeben werden muss. Die Bibliothek kennt die Angaben 100, 125, 250, 500 und 1000. Obwohl diese mit der Testinstallation zuverlässig funktionieren, kann es aufgrund der Netztopologie unter Umständen nötig sein, das Bustiming selbst zu ermitteln, siehe Abschnitt 2.7. Wird ein anderer Wert übergeben, so werden 125 kBaud als Standardwert gesetzt. Dieses hat zur Folge, dass der Teilnehmer einige Errorframes erzeugt und sich schließlich abschaltet.

void can_db (unsigned char mor, unsigned char databyte[7], unsigned char datalength)
schreibt das acht Byte große Datenfeld *databyte* in Message Object Register *mor*, setzt die Datenlänge auf *datalength* und definiert das Register zusätzlich als Senderegister.

mor muss dabei im Bereich von 1 bis 14 liegen, bei unzulässigen Werten wird die Funktion wieder beendet. *datalength* darf nach den CAN Spezifikationen 0–8 betragen, ein Wert größer acht wird auf acht verringert.

struct CAN_STRUCT can_read (unsigned char mor)
liest die Daten aus dem Message Object Register *mor* (1–15). Falls während des Lesens die Daten vom CAN-Controller überschrieben werden, startet der Lesevorgang erneut. So wird gewährleistet, dass alle Datenbytes konsistent sind. Die Daten werden als Struktur CAN_DATA zurückgegeben: Feld *unsigned char can_byte[7]* mit den Datenbytes und einem Byte Datenlänge *unsigned char can_length*.

void can_rcv (unsigned char mor)
definiert das Message Object Register *mor* (1–15) als Empfangsregister.

void can_remote (unsigned char mor, unsigned char datalength)

sendet für Message Object Register *mor* (1–15) einen Remoteframe. Das Register sollte als Empfangsregister konfiguriert sein. Diese Funktion ist praktisch identisch mit *can_send*, allerdings wird hier zusätzlich die Datenlänge (DLC) auf den richtigen Wert gesetzt.

void can_inv (unsigned char mor)

definiert das Message Object Register *mor* (1–15) als ungültig. Dieses Register wird ignoriert, bis es mit *can_db* oder *can_rcv* erneut aktiviert wird.

void can_ar (unsigned char mor, unsigned short identifier)

definiert den Identifier für Message Object Register *mor* (1–15). Dieses gilt sowohl für das Sende- als auch für das Empfangsregister. *Identifier* ist ein 11-Bit-Wert, das heißt die fünf höchstwertigen Bits werden bei der Auswertung ignoriert.

void can_gms (unsigned short identifier)

setzt die Global Mask Short. Mit „0“ belegte Bits eingehender Nachrichten werden nicht mit dem vorgegebenen Identifier verglichen. Wie bei *can_ar* werden nur 11 Bits ausgewertet.

void can_mlm (unsigned short identifier)

setzt die Filtermaske für das Last Message Object (Register 15). Dabei gilt gleichzeitig die globale Maske, das heißt sobald das entsprechende Bit einer der beiden Masken „0“ ist, wird das zugehörige Identifierbit nicht ausgewertet.

void can_send (unsigned char mor)

sendet aus Message Object Register *mor* (1–14). Diese Routine kann sowohl auf Sende- als auch auf Empfangsregister angewendet werden. Die Senderegister übertragen eine Nachricht, Empfangsregister senden ein Remoteframe. Da Remoteframes zur Arbitrierung die korrekte Datenlänge aufweisen müssen, sollte hierzu *can_remote* verwendet werden, das dieses übernimmt. Aus diesem Grund kann die Funktion auch nur auf die Register 1–14 angewendet werden.

unsigned char can_status

gibt den Wert des CAN Status Registers zurück, siehe Tabelle 4.

Bedeutung des Statusregister–Bits:

7	Busoff	Der Controller ist im Busoff Status
6	Fehler	Einer der Fehlerzähler hat die Warnschwelle erreicht
5	unbenutzt	keine Funktion
4	Empfang	Eine Nachricht wurde erfolgreich empfangen
3	Senden	Eine Nachricht wurde erfolgreich übertragen
2	LEC2	Last Error Code Bit 2
1	LEC1	Last Error Code Bit 1
0	LEC0	Last Error Code Bit 0

Bedeutung des Last Error Codes:

0	kein Fehler
1	Stopffehler
2	Formfehler (Fehler im fest definierten Nachrichtenteil)
3	Kein ACK empfangen
4	Bit 1 als dominant empfangen
5	Bit 0 als rezessiv empfangen Im Busoff–Status wurden 11 rezessive Bits empfangen
6	CRC Fehler
7	unbenutzt, siehe *can_state*

Tabelle 4: CAN Status Register des C515C

unsigned char can_state

gibt wie *can_status* das Statusbyte zurück. Zusätzlich werden aber die Bits 3 und 4 auf „0“ und der „Last Error Counter“ auf „7“ gesetzt. Da 7 unbenutzt ist, kann dieser Wert als Kennzeichen genutzt werden, um eine Veränderung festzustellen.

bit can_msglst (unsigned char mor)

gibt eine „1“ zurück, wenn im Message Object Register *mor* (1–15) eine Nachricht verloren ging, weil es nicht ausgelesen wurde, bevor die neue Nachricht eintraf. Diese Funktion ergibt nur bei Empfangsregistern ein sinnvolle Aussage, bei Senderegistern hat der ausgelesene Wert eine andere Bedeutung.

bit can_newdat (unsigned char mor)
gibt eine „1“ zurück, wenn im Message Object Register *mor* (1–15) eine neue Nachricht empfangen wurde. Dabei wird dieser Wert nicht gelöscht. Das Löschen geschieht erst durch Lesen der Nachricht mit *can_read*.

7.4 Einbinden in eigene Programme

Zum Einbinden der CAN-Funktionen sind die Dateien *canctrlr.c* und *canctrlr.h* erforderlich. Die Quelldatei *CANCTRLR.C* ist in das Projekt aufzunehmen (*Projekt/Edit/Add*). Im Kopf des eigenen Programmes muss *#include <canctrlr.h>* angegeben werden.

7.4.1 Beispiel: Senden einer Nachricht

Dieses Beispiel zeigt eine Prozedur, die den Buscontroller auf 125 kBaud einstellt und aus Message Object Register 1 die Nachricht „00000010000“ = 0x0010 mit dem Inhalt „0xABBA“ sendet.

```
void senden (void)                      // Senderoutine
{
    data unsigned char Sendedaten[7];   // definiere Array
    Sendedaten[0] = 0xAB;               // Setze Byte 0
    Sendedaten[1] = 0xBA;               // Setze Byte 1
    can_init (125);                     // initialisiere Bus für 125 kBaud
    can_ar (1, 0x0010);                 // setze Identifier für Message Object
                                        Register 1: 00000010000
    can_db (1, Sendedaten, 2);          // speichere das vorhin definierte Array
                                        in Message Object Register 1 und setze
                                        DLC auf 2
    can_send (1);                       // sende Nachricht aus Message Object
                                        Register 1
}
```

7.4.2 Beispiel: Empfang einer Nachricht

Dieses Beispiel zeigt eine Prozedur, die den Buscontroller auf 125 kBaud einstellt und in Message Object Register 1 auf die Nachricht „00000010000“ = 0x0010 wartet. Danach wird das Register abgeschaltet. Die Speicherinhalte entsprechen dem, was mit dem vorhergehenden Sendebeispiel in den Speicher geschrieben würde. Im Regelfall würde *can_newdat* wahrscheinlich aus einer Schleife heraus aufgerufen werden, in der sich noch weitere Funktionsaufrufe befinden. Aus diesem Grund wäre die Nutzung von Interrupts bei zeitkritischen Anwendungen eine sinnvolle Alternative.

```
void empfangen (void)                  // Senderoutine
{
    struct CAN_DATA Nachricht;         // definiere Struktur für empfangene
                                       Nachricht
    can_init (125);                    // initialisiere Bus für 125 kBaud
    can_ar (1, 0x0010);                // setze Identifier für Message Object
                                       Register 1: 00000010000
    can_rcv (1);                       // erkläre Message Object Register 1
                                       zum Empfangsregister
    while (can_newdat (1) == 0) { }    // warte auf Nachricht
    Nachricht = can_read (1);          // lese Nachricht aus Message Object
                                       Register 1
                               // Nachricht.can_byte[0] = 0xAB
                               // Nachricht.can_byte[1] = 0xBA
                               // Nachricht.can_length = 2
                               // can_newdat (1) = false
    can_inv (1);                       // schalte Message Object Register 1
                                       ab
}
```

8 Aufbau eines eigenständigen Bussystems

Zum Aufbau des Bussystems werden zwei Phytec-miniMODULe 515C als Busteilnehmer verwendet. An einem dieser Module wird über ein Relais ein Motor geschaltet, als Freilaufdiode für das Relais wird eine 1N4007 verwendet. Hierzu wurde eine Zusatzplatine angefertigt, die aus einem Signal des Mikroprozessors die Ansteuerung des Relais realisiert, siehe Abbildung 52 und 53 in Anhang B.4.

Über einen Widerstand wird ein PNP-Transistor bei Low-Pegel durchgeschaltet. Der Vorteil hierbei liegt darin, dass der Prozessor keinen Strom zu liefern braucht.

In dem Motor ist ein Winkelschrittgeber eingebaut, der 660 Impulse pro Umdrehung liefert.

Aus den Impulsen wird die Drehzahl durch den C515C-Prozessor errechnet. Der Teilnehmer sendet die Drehzahl in über den Bus einstellbaren Intervallen mit einem Datentelegramm auf das Medium. Zur Berechnung der Drehzahl werden ein Timer und ein Zähler des 80C515C genutzt. Die Einstellung der Zeitspanne für den Timer geschah mit Hilfe eines Oszilloskopes, da der Motor relativ ungleichmäßig läuft, muss bei Drehzahlen von etwa 1000 pro Minute mit einem Fehler von bis zu 10% gerechnet werden. Auf einen besseren Abgleich wurde für die Demonstration verzichtet.

Mit dem zweiten Teilnehmer kann der Motor über einen Taster an- beziehungsweise ausgeschaltet werden. Ferner wird die Drehzahl des Motors über ein Display ausgegeben. Die Ansteuerung des Displays erfolgt über Ausgänge des Mikrocontrollers. Sie ist über Software realisiert. Das Display zeigt zunächst eine Begrüßungsmeldung, bevor die empfangene Drehzahl erscheint. Werden keine Daten erhalten, so erscheint die Meldung - keine Daten -. Anderenfalls blitzt die auf dem miniMODUL vorhandene LED kurz auf. Um ein Ablesen des Displays auch bei zu häufig wechselnden Werten zu ermöglichen, geschieht die Anzeige neuer Daten nur in bestimmten Abständen. Werden aus diesem Grund neue Messdaten nicht angezeigt, so erscheint hinter dem alten Messwert ein Stern. Anschaltung und Programmierung des Displays wurden [bal92] entnommen und lediglich auf den 80C515C angepasst.

Die Programmlistings für die beiden Teilnehmer sind im Anhang auf Seite 96ff beziehungsweise 98ff aufgeführt.

9 Entwicklungssystem DIDO537

9.1 Hardwaremodul DIDO537

Die DIDO537 ist eine an der Fachhochschule Hannover entwickelte Mikrocontrollerplatine. Als Prozessor wird der 80C537 von der Firma Siemens verwendet. Die DIDO537 verfügt über zwei serielle Schnittstellen nach RS-232 und mehrere frei nutzbare über Steckverbinder zugängliche I/O-Ports. Es stehen 32 kB RAM und 32 kB ROM zur Verfügung, eine flexible Adressdecodierung ist durch den Einsatz eines GAL-Bausteins gewährleistet. Zur Kommunikation mit dem CAN-Control-Board, das im Abschnitt 10 beschrieben wird, müssen zusätzlich die Signale gemäß nachfolgender Tabelle an den Steckverbinder gebracht werden.

Steckverbinder	an
Pin 34	74573 Pin 11
Pin 33	Reset-Taster
Pin 32	GAL Pin 18
Pin 31	Prozessor Pin 23 (INT 0)
Pin 30	GAL Pin 3

Tabelle 5: Zusätzliche Verbindungen der DIDO537

Für die DIDO537 ist kein nichtflüchtiger Programmspeicher vorhanden. Aus diesem Grund muss nach einem Reset oder Neustart das auszuführende Programm erneut eingespielt werden. Alternativ ist auch der Einsatz eines EPROMS möglich, siehe Dokumentation *DIDO537.pdf*.

9.2 Terminalprogramm XT

Das Terminalprogramm XT ermöglicht die Bedienung des Entwicklungssystems DIDO unter Zuhilfenahme eines PCs. Zur Programmierung des Controllers 80C537 wird eine serielle Verbindung zu einem PC hergestellt. Mit Hilfe des Terminalprogramms *XT* werden die Daten für den Mikrocontroller übermittelt. Alle vom Mikrocontroller auf der seriellen Schnittstelle eintreffenden Daten werden am Bildschirm dargestellt, alle Tastatureingaben über die Schnittstelle ausgegeben. Zusätzlich enthält XT weitere Funktionen, um Programme und Daten an den Mikrocontroller zu senden.

Mit dem Aufruf des Programms können Parameter zur Konfiguration angegeben werden.

1	PC-Schnittstelle COM 1 (default)
2	PC-Schnittstelle COM 2
1200	1200 Baud
2400	2400 Baud
4800	4800 Baud
9600	9600 Baud (default)
A	Automatischer Start eines Programms

Tabelle 6: Aufrufparameter von XT

Alle anderen Eingaben werden als Dateiname interpretiert. XT versucht diese Datei zu laden und sendet sie an den Mikrocontroller. Wurde zusätzlich der Parameter A angegeben, so wird danach das Kommando G0 an den Mikrocontroller gesendet. Das Monitorprogramm startet das Programm, das im Codespeicher an der Adresse 0000h steht.

Für die Übertragung ist „keine Parität, 8 Datenbits, 1 Startbit, 1 Stoppbit“ festgelegt.

Zur Kommunikation mit dem Mikrocontroller muss ein Programm im Intel-Hex-Format vorliegen. Zur Programmierung kann der automatische Dateitransfer verwendet oder der Transfer manuell mit der Funktionstaste *F2* gestartet werden.

Das Monitorprogramm bietet folgende Befehle:

*D*x Adresse	Anzeigen
*E*x Adresse	Editieren
*F*x Adresse	Füllen
mit $x = C$	Speicherbereich Code
mit $x = I$	Speicherbereich IDATA
mit $x = X$	Speicherbereich XDATA
*:*Hex-Wert	Lade Intel Hex-Datei
S Bereich	Speichere Intel Hex-Datei
G Adresse [, Stopadresse]	Starte Programm ab *Adresse*
T Schrittzahl	Einzelschritt
U Adresse	Disassemblieren
X Register	Register anzeigen und ändern
M oder *?*	Hilfe
F1	XT beenden
F2	Dateitransfer vom PC an den Mikrocontroller
F3	Bildschirmausgabe in Protokolldatei schreiben

Tabelle 7: Befehle des Terminalprogramms XT

10 Das CAN Control Board

10.1 Allgemeines

Für die Verbindung des Mikrocontrollers 80C537 zum CAN-Bus wurde eine Zusatzplatine als Aufsteckmodul für die DIDO537 entwickelt. Herzstück dieser Schaltung ist der CAN-Controller SJA1000, der die Kommunikation steuert. Die Umwandlung der Daten des SJA1000 in CAN_H- bzw. CAN_L-Pegel wird durch den CAN-Treiber 82C250 realisiert. Zur galvanischen Entkopplung des Bussystems wurden Optokoppler verwendet, jeweils einer für die zu schreibenden und die zu lesenden Daten. Die Trennung der Stromkreise des Mikrocontrollers vom Bustreiber erfolgt über einen DC-DC-Wandler. Für die CAN-Busleitung wurden zwei Anschlüsse mit der Möglichkeit zwei Leitungen anschließen zu können parallel eingebaut. Über einen DIP-Schalter ist ein Widerstand für den Busabschluss zuschaltbar. Der Blockschaltplan des CAN Control Boards ist in Abbildung 27 dargestellt, der Schaltplan und das Platinenlayout befinden sich im Anhang.

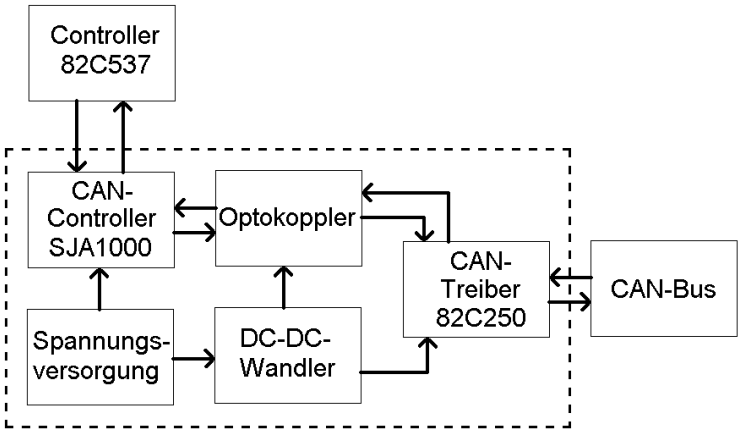


Abbildung 27: Blockschaltbild des CAN Control Boards

10.2 Der CAN-Controller SJA1000

Der Philips SJA1000 ist ein Stand-alone-CAN-Controller zum Anschluss an Mikroprozessoren. Er implementiert die Schichten 1 und 2 des ISO/OSI-Schichtenmodells, das heißt, er wandelt auf dem Adress-/Datenbus empfangene Daten in CAN-Botschaften auf Bitebene um und schreibt in Gegenrichtung die vom Bus gelesenen Daten in Register, die von der CPU, in diesem Fall der 80C537, ausgelesen werden können.

Auf der hier beschriebenen Zusatzplatine ist der Mode-Eingang – Pin 11 – zur Wahl von Intel- bzw. Motorola-Betriebsart des ICs auf High gelegt, so dass der Intel-Modus genutzt wird. Alle anderen erforderlichen Einstellungen für den Betrieb des CAN-Controllers sind per Software gesetzt.

Der Oszillatortakt wird mit einem eigenen 24-MHz-Quarz erzeugt, der SJA1000 könnte alternativ auch vom Takt der CPU versorgt werden, diese versorgen oder mit einem externen Oszillator betrieben werden.

Als Nachfolger des 82C200 hat der SJA1000 im BasicCAN-Modus dessen Eigenschaften und ist zu ihm pinkompatibel. Er verfügt zusätzlich im sogenannten PeliCAN Modus über erweiterte Eigenschaften, die auch die Nutzung von Extended Frames nach CAN 2.0B einschließen.

Die Kommunikation mit der angeschlossenen CPU erfolgt über die Adress- und Datenleitungen AD0-AD7. Der Controller wird als memory-mapped Eingabe-/Ausgabegerät angesprochen. Für den BasicCAN-Modus werden 32 Bytes Adressraum benötigt. Da die höherwertigen Adressbits nicht ausgewertet werden, wiederholen sich die Register mehrfach, Byte 32 entspricht Byte 0 und so weiter.

Der Reseteingang $\overline{\text{RST}}$ des SJA1000 ist mit der Resetleitung der DIDO537 verbunden, so dass bei einem Hardwarereset beide Controller zurückgesetzt werden. Der Chip-Select $\overline{\text{CS}}$ und das Address Latch Enable ALE werden von der DIDO537 gesteuert.

Der SJA1000 kann über eine Verbindung zum 80C537 einen Interrupt auslösen. In weiterführenden Projekten kann der CAN-Controller das Eingehen einer Nachricht, das erfolgreiche Übertragen einer Nachricht und einige andere Ereignisse mit einem Interrupt melden. Details hierzu in der Applikation zum SJA1000.

Der SJA1000 weist eine Anomalie betreffend des Akzeptanzfilters auf.

Diese tritt nur in sendenden Teilnehmern auf, wenn die Arbitrierung nach dem 7. Identifier-Bit verloren geht und gleichzeitig der Akzeptanzfilter aktiv genutzt wird (Akzeptanzmaske ungleich FFhex). In diesem Fall wird der Algorithmus des Akzeptanzfilters derart beeinflusst, dass die höher priore Nachricht

- **nicht empfangen wird, obwohl sie den Akzeptanzfilter passieren sollte oder**
- **empfangen wird, obwohl sie den Akzeptanzfilter nicht passieren sollte**

So könnte eine Nachricht im Empfangspuffer fehlen oder im Empfangspuffer gespeichert werden, obwohl sie nicht hineingeschrieben werden sollte.

Mögliche Umgehung des Problems:

- **Der Akzeptanzfilter wird nicht genutzt: die Akzeptanzmasken-Register auf FFhex setzen[2].**
- **Die Identifier im Netz werden so ausgewählt, dass die Arbitrierung innerhalb der ersten 7 Bits des Identifikationsfeldes abgeschlossen ist**
- **Die Arbitrierung wird durch Softwarekontrolle ganz verhindert (Master-Slave Konzept: Jeder Teilnehmer antwortet nur auf Anforderung eines Masters)[3]**

Hierzu gibt es ein Merkblatt vom Hersteller, das im Anhang aufgeführt ist.

10.3 Die Optokoppler 6N137

Die Optokoppler dienen ausschließlich zur Entkopplung des Busses vom Gesamtsystem. Dabei wird ein Optokoppler zur Übertragung der Daten des Bustreibers zum CAN-Controller, der andere für die umgekehrte Richtung verwendet. Die Optokoppler 6N137 arbeiten mit einer typischen Geschwindigkeit von bis zu 10 MBaud, somit ist eine einwandfreie Funktion auch bei der maximalen CAN-Baudrate von 1 MBaud stets gewährleistet.

[2]Entspricht BasicCAN-Funktionalität.

[3]Widerspricht der Idee der Vernetzung dezentraler Controller.

10.4 Der DC-DC-Wandler TME0505S

Der DC-DC-Wandler unterbricht galvanisch die Verbindung zwischen der Spannung von Mikrocontroller und SJA1000 einerseits und der für die zu trennende Sektion andererseits. Zur Auslegung des DC-DC-Wandlers wurde folgende Berechnung ausgeführt:

LED im Optokoppler	I_F	max.	15 mA
Optokoppler-Versorgung	$I_{CC} + I_O$	max.	13,1 mA
Transceiver		max.	100 mA
Gesamtbedarf		$\approx$	130 mA
Reserve	ca. 50%	$\approx$	70 mA
Summe			200 mA

Tabelle 8: Auslegung des DC-DC-Wandlers

10.5 Der CAN-Transceiver 82C250

Der Transceiverbaustein 82C250 ist für die Umsetzung der Daten aus dem entkoppelten CAN-Controller in einen CAN_H bzw. CAN_L-Pegel verantwortlich, dieses geschieht über die Leitung TxD. Ebenso liefert er dem CAN-Controller einen Logikpegel, der dem Buszustand entspricht, das gilt auch für die Daten, die von ihm selbst gesendet werden.

11 Treibersoftware für das CAN Control Board

11.1 Allgemeines

Die Funktionalität der Routinen für das CAN Control Board entspricht der der Routinen für den 80C515C. Die Unterschiede zwischen dem CAN-Controller des 80C515C und dem SJA1000 bedingen – insbesondere durch die Zahl der Sende- und Empfangsregister – voneinander abweichende Funktionsaufrufe, so dass es nicht möglich ist, Programme durch Wechseln der eingebundenden Routinen auf den jeweils anderen Controller umzuschreiben. Flexibel anwendbare CAN-Bibliotheken müssen Funktionen höherer Schichten enthalten, zum Beispiel die Aufteilung von Nachrichten auf die Message Object Register.

Für die Verwendung von 29-Bit-Identifiern nach CAN2.0B muss der PeliCAN-Modus des SJA1000 genutzt werden. In diesem Fall müssen alle verwendeten Routinen umgeschrieben werden, da die Register eine andere Belegung haben.

11.2 Funktionsbeschreibungen

In weiten Teilen entspricht die Wirkung der Funktionen denen des 80C515C, vollständigkeitshalber sind sie im folgenden trotzdem aufgeführt.

void can_init (unsigned short Bus_speed, unsigned char filter, unsigned char mask)
ist wie beim 80C515C die zunächst aufzurufende Routine.

Weiterhin setzt *can_init* beim SJA1000 die Filterbits und die Bits der Filtermaske für das Empfangsregister. Beide werten lediglich die acht höchstwertigen Bits des Identifiers aus.

void can_db (unsigned char databyte[7], unsigned char datalength)
schreibt das acht Bytes große Datenfeld *databyte* in das Senderegister und setzt die Datenlänge auf *datalength*.

struct CAN_STRUCT can_read (void)
liest die Daten aus dem Empfangsregister. Die Daten werden als Struktur CAN_DATA

zurückgegeben: Feld *unsigned char can_byte[7]* mit den Datenbytes und einem Byte Datenlänge *unsigned char can_length*.

void can_remote (void)

sendet einen Remoteframe. Das **Senderegister** muss vorher auf den richtigen Identifier und die richtige Datenlänge konfiguriert werden. Diese Funktion ist identisch mit *can_send*, allerdings wird hier das RTR-Bit gesetzt.

void can_ar (unsigned short identifier)

definiert den Identifier für das Senderegister. *Identifier* ist ein 11-Bit-Wert, das heißt die fünf höchstwertigen Bits werden bei der Auswertung ignoriert.

void can_send (void)

sendet die Nachricht, die sich im Senderegister befindet.

unsigned char can_status

gibt den Wert des CAN Status Registers zurück, siehe Tabelle 9.

Bedeutung des Statusregister–Bits:

7	Busoff	Der Controller ist im Busoff Status
6	Fehler	Einer der Fehlerzähler hat die Warnschwelle erreicht
5	Transmit	Eine Nachricht wird übertragen
4	Empfang	Eine Nachricht wird empfangen
3	Gesendet	Eine Nachricht wurde erfolgreich übertragen
2	Sendepuffer	Der Sendepuffer darf beschrieben werden
1	Überlauf	Eine Nachricht ging verloren
0	Empfangspuffer	Eine neue Nachricht kann gelesen werden

Tabelle 9: Status Register des SJA1000

bit can_msglst (void)

gibt eine „1“ zurück, wenn im Empfangsregister eine Nachricht verloren ging, weil eine ältere Nachricht noch nicht ausgelesen wurde. Das Überlaufflag wird gelöscht, so dass ein neuer Aufruf dieser Funktion eine 0 ergibt.

bit can_newdat (void)
gibt eine „1“ zurück, wenn eine neue Nachricht empfangen wurde. Dabei wird dieser Wert nicht gelöscht. Das Löschen geschieht erst durch Lesen der Nachricht mit *can_read*.

11.3 Einbinden in eigene Programme

Zum Einbinden der CAN-Funktionen sind die Dateien *canctsja.c* und *canctsja.h* erforderlich. Die Quelldatei *CANCTSJA.C* ist in das Projekt aufzunehmen (*Projekt/Edit/Add*). Im Kopf des eigenen Programmes muss weiterhin *#include <canctsja.h>* angegeben werden.

11.3.1 Beispiel: Senden einer Nachricht

Dieses Beispiel zeigt eine Prozedur, die den Buscontroller auf 125 kBaud einstellt und aus Message Object Register 1 die Nachricht „00000010000“ = 0x0010 mit dem Inhalt „0xABBA“ sendet. Der Akzeptanzcode wird auf 0x00 eingestellt, mit der Akzeptanzmaske 0x00 werden alle Bits geprüft. Ein so konfigurierter Controller schreibt die Nachrichten „00000000XXX“ in den Empfangspuffer.

```
void senden (void)                      // Senderoutine
{
   data unsigned char Sendedaten[7];    // definiere Array
   Sendedaten[0] = 0xAB;                // Setze Byte 0
   Sendedaten[1] = 0xBA;                // Setze Byte 1
   can_init (125,0x00,0xFF);            // initialisiere Bus für 125 kBaud,
                                        // Filter 0x00, Maske 0x00,
   can_ar (0x0010);                     // setze Identifier: 00000010000
   can_db (Sendedaten, 2);              // speichere das vorhin definierte Array
                                        im Senderegister und setze DLC auf 2
   can_send ();                         // sende Nachricht
}
```

11.3.2 Beispiel: Empfang einer Nachricht

Dieses Beispiel zeigt eine Prozedur, die den Buscontroller auf 125 kBaud einstellt und auf die Nachricht „00000010000“ = 0x0010 wartet. Als Akzeptanzcode ist hierbei 0x02 einzugeben, als Akzeptanzmaske 0x00, so dass der Controller nur Nachrichten mit dem Identifier „00000010XXX“ in den Empfangspuffer schreibt. Die abschließend genannten Werte entsprechen denen, die empfangen werden, wenn eine Nachricht wie im vorhergehenden Beispiel gesendet wird. Im Regelfall würde *can_newdat* wahrscheinlich aus einer Schleife heraus aufgerufen werden, in der sich noch weitere Funktionsaufrufe befinden. Aus diesem Grund wäre die Nutzung von Interrupts bei zeitkritischen Anwendungen eine sinnvolle Alternative.

```
void empfangen (void)                          // Senderoutine
{
    struct CAN_DATA Nachricht;                 // definiere Struktur für empfangene
                                               Nachricht
    can_init (125,0x02,0x00);                  // initialisiere Bus für 125 kBaud
                                               // Filter 0x02, Maske 0x00,
    do
    {
      while (can_newdat () == 0) { }           // warte auf Nachricht
      Nachricht = can_read ();                 // lese Nachricht aus Message Ob-
                                               ject Register 1
    } while (Nachricht.can_ID != 0x10);        // prüfe Identifier
                                          // Nachricht.can_byte[0] = 0xAB
                                          // Nachricht.can_byte[1] = 0xBA
                                          // Nachricht.can_length = 2
                                          // Nachricht.can_RTR = 0
                                          // can_newdat () = false
}
```

12 Der Troublemaker Version 1.3

12.1 Allgemeines

Mit einer Zusatzplatine, dem Troublemaker, kann eine vom eigenen Teilnehmer gesendete Nachricht definiert zerstört werden. Dadurch wird ein Errorframe erzeugt. Über DIP-Schalter können im Datenrahmen in den Bereichen Identifier, Daten, CRC und EOF dominante Pegel eingefügt werden. Dieses geschieht über einen Treiber, der separat vom Mikroprozessor der DIDO537 angesteuert wird. Die Position des dominanten Pegels wird über die Software festgelegt. Weitere Fehlererzeugungen sind Kurzschluss der CAN_H-CAN_L-Leitung und deren Unterbrechung ebenfalls über DIP-Schalter.

DIP 1	Troublemaker aktiv
DIP 2	CAN_H unterbrochen
DIP 3	CAN_L unterbrochen
DIP 4	Kurzschluss
DIP 5	Fehlerort
DIP 6	Fehlerort

Tabelle 10: Schalterbelegung des Troublemakers

DIP 5	DIP 6	Fehlerort
off	off	Daten
off	on	End-of-Frame EOF
on	off	bei 125 kBaud: Idle (kurz) sonst: Identifier
on	on	Idle (lang)

Tabelle 11: Auswahl der Fehlerorte

12.2 Funktion der Fehlererzeugung

Der Troublemaker verfügt über einen digital entprellten Taster, der mit einem Eingang des 80C537 verbunden ist. Die Entprellung wird mit der Zusammenschaltung zweier NAND-Gatter als astabile Kippstufe vorgenommen. Durch Auslösen der Taste wird dem CAN Control Board durch den 80C537 der Befehl zum Senden einer Nachricht auf den Bus gegeben. Der Mikrocontroller gibt danach auf einem Port ein Low-Signal aus, das bei Aktivierung der CAN Troublemaker-Platine (DIP 1 *on*) den CAN-Treiber 82C250

ansteuert (TxD, Pin 1), der den gesamten Bus für die Dauer dieses Signals auf einen dominanten Pegel zieht. An welcher Stelle der Nachricht der Pegel dominant wird ist über die DIP-Schalter 5 und 6 wählbar. Die Position der DIP-Schalter wird durch den 80C537 per Software abgefragt. Die Schalterstellungen DIP 2 oder 3 *off* bewirken eine Unterbrechung des CAN_H- beziehungsweise CAN_L-Signales zwischen Ein- und Ausgang des Troublemakers. Ferner wird mit DIP-Schalter 4 CAN_H und CAN_L am Ausgang kurzgeschlossen.

12.3 Software zur Fehlererzeugung

Das Programm *ERROR.C* initialisiert zunächst den Bus. Da es keine Daten empfangen muss, wird der Eingangsfilter auf 11111111XXX gesetzt. Telegramme mit diesen Identifiern werden im Versuchsaufbau nicht genutzt.

Als Datenbytes werden 0x00, 0x55 und 0x00 gewählt, da die Null-Bytes die maximale Zahl von Nullen in einem Datenrahmen enthalten. Auf diese Weise kann die Funktion der Fehlererkennung getestet und das Prinzip des Bitstuffings am Oszilloskop gezeigt werden. 0x55 entspricht binär „01010101“ und ermöglicht das Unterscheiden der einzelnen Bits.

Das Hauptprogramm beginnt zunächst mit der Abfrage des Statusregisters. Der Error Warning Level wird mit einer extern angeschlossenen gelben LED angezeigt. Im Busoff-Zustand bleibt das Programm nach dessen Anzeige mit einer roten LED so lange in einer Warteschleife, bis eine Taste gedrückt wird. Nach einigem Blinken der roten LED erfolgt dann eine erneute Initialisierung des CAN-Controllers.

Nach der Verarbeitung des Statusbytes wird die Stellung der DIP-Schalter eingelesen, die zugeordneten Fehlerorte ermittelt und über grüne LEDs ausgegeben. Falls der Taster betätigt wurde, erfolgt danach die Erzeugung des Fehlers. Liegt dieser in einem Datenrahmen, so wird dieser mit *can_send* übertragen. Die Zeitspanne bis zum Ansteuern des externen Bustreibers 80C250 und danach bis zu dessem Abschalten wird mit Warteschleifen bestimmt, da diese im Gegensatz zu Interrupt-Aufrufen über einen Timer immer exakt die gleiche Taktzahl benötigen. In einigen Fällen ist es erforderlich, die Wartezeit noch genauer einzustellen. Hierzu wird der Port mehrfach gesetzt.

13 Der Errorfinder Version 1.6

13.1 Allgemeines

Der Errorfinder dient zum Erkennen eines Errorframes, der von dem zu prüfenden Busteilnehmer auf den Bus gesendet wurde. Wenn der am Errorfinder angeschlossene Teilnehmer Strom aufnimmt, so wird erkannt, dass dieser Teilnehmer der Auslöser des Errorframes ist. Hierzu sind Einstellungen durch den Bediener nötig. Der Errorfinder wird mit dem 9-poligen Stecker an den zu überwachenden Teilnehmer angeschlossen. Ferner ist es möglich, ihn in den Bus einzufügen und somit mehrere Teilnehmer eines Abschnittes zu überwachen. Eine Leuchtdiode dient hierbei als Erkennungsinstrument. Leuchtet die Diode, wurde ein Errorframe eines zu prüfenden Teilnehmers erkannt. Durch Druck auf einen Taster wird die Schaltung zurückgesetzt.

Eine Messung kann nur an CAN-Bussen mit einer Baudrate von 1000, 500, 250 oder 125 kBaud durchgeführt werden. Die Busgeschwindigkeit wird über die Jumper 2-5 gewählt.

Baudrate		Jumper 2	Jumper 3	Jumper 4	Jumper 5
1	MBaud	geschlossen	offen	offen	offen
500	kBaud	offen	geschlossen	offen	offen
250	kBaud	offen	offen	geschlossen	offen
125	kBaud	offen	offen	offen	geschlossen

Tabelle 12: Einstellung der Baudrate des Errorfinders

13.2 Erkennung der Stromrichtung

Zur Erkennung der Stromrichtung wird ein Komparator mit differentiellem Eingang (Philips LM311) verwendet. Zur Ansteuerung dieses Bausteins wurde ein 10-Ω-Widerstand in die CAN_L-Leitung eingesetzt und darüber die differentiellen Eingänge des Komparators angeschlossen, so dass der Komparator bei Stromfluss in Richtung des Teilnehmers eine Ausgangsspannung liefert.

13.3 Erkennung eines Errorframes

Für die Umwandlung des Signals vom Bus wurde ein Komparator mit jeweils einem differentiellen Eingang an CAN_H und CAN_L angeschlossen. Damit der Komparator einen Ausgangspegel zwischen V_{CC} und Masse liefern kann, muss ein Bezug zum Bus hergestellt werden. Dieses erfolgt über die Verbindung Masse-CAN_H zur Schaltungsmasse. Der Ausgang des Komparators liefert dann den Buspegel als Pegel zwischen V_{CC} (dominant) und 0 (rezessiv).

Die Erkennung eines Errorframes wird über den Abwärtszähler74HCT40103 realisiert. Seine Ansteuerung erfolgt durch den Buspegel. Der Zähler hat die Eigenschaft, bei einem High-Pegel an $\overline{\text{PL}}$ (asynchronous preset enable input) von einer vorher definierten Startwert herunter zu zählen und gibt an seinem Ausgang wenn er 0 erreicht hat kurzzeitig ein Low-Signal aus. Der Startwert ist so gewählt, dass das Ausgangssignal während des 6. dominanten Bits auf dem Bus ausgegeben wird. Bei einem rezessiven Buspegel wird der Zähler auf den Ausgangswert zurückgesetzt. Wenn ein Ausgangssignal vorhanden ist, handelt es sich somit um einen Errorframe. Zur weiteren Auswertung ist es erforderlich ein RS-Flip-Flop des 4044 zu setzen.

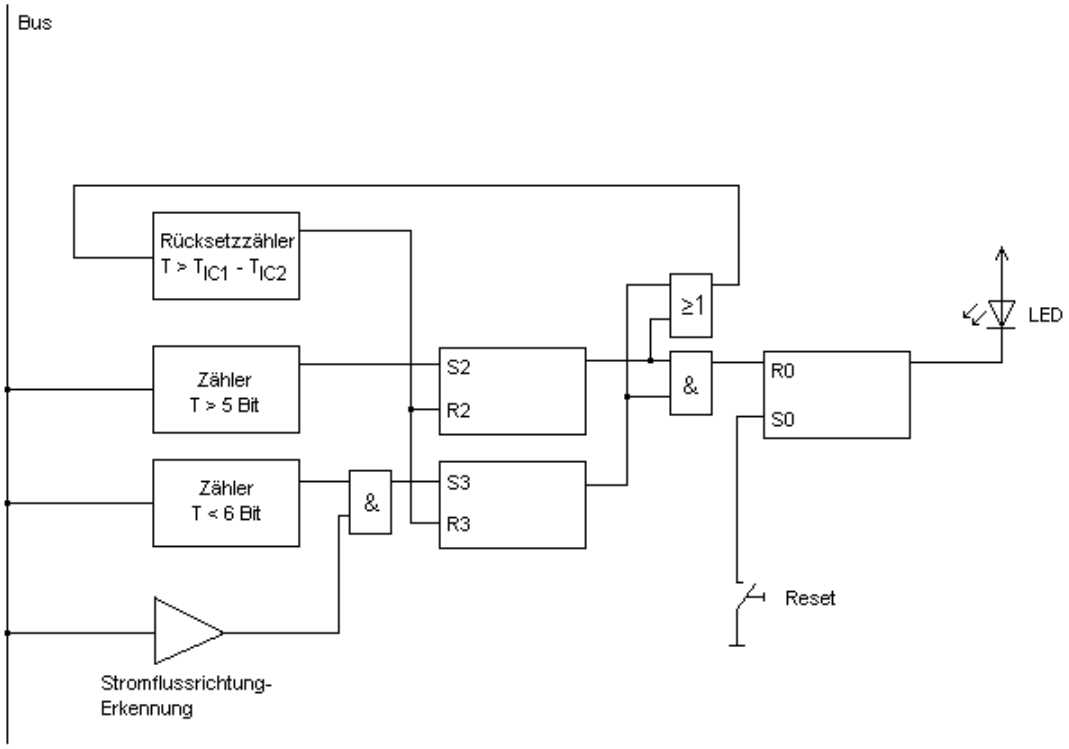
13.4 Signalauswertung

13.4.1 Allgemein

Zur Signalauswertung wird die Stromrichtung innerhalb einer erlaubten Low-Periode ermittelt. Wird ein Errorframe erkannt und die Stromrichtungserkennung liefert ein Signal, handelt es sich um einen Errorframe, der von dem zu überprüfenden Teil des Busses ausgelöst wurde.

13.4.2 Realisierung

Die Realisierung der Signalauswertung ist in dem Blockdiagramm 28 dargestellt.



Zur Verknüpfung der in den vorangegangenen Abschnitten beschriebenen Signale werden zwei weitere Zähler benötigt. Einer dieser Zähler (IC2) gibt innerhalb fünf dominanter Bits des Busses ein kurzzeitiges Low-Signal aus (hier drei Bits). Dieses wird invertiert und mit dem Ausgang der Stromrichtungserkennung (U2) verglichen. Sind beide aktiv, wird RS-Flip-Flop 3 gesetzt. Dieser Ausgang steuert einen Zähler (IC6), der für das Rücksetzen der Flip-Flops 2 und 3 benötigt wird. Das Rücksetzen erfolgt nach einer Zeit, die größer sein muss als der Zeitraum, in dem der Errorframe erkannt wird (in diesem Fall acht Bit). Dieser Zeitraum darf nicht länger sein als ein Errorframe mit nachfolgender Ruhephase (6+8 Bits).

Wie bereits beschrieben gibt die Errorframeerkennung (IC1) im Fehlerfall während des 6. Bits einen High-Pegel aus. Somit wird im gewählten Fall erreicht, dass zwischen dem 6. und dem 8. Bit beide Flip-Flop-Ausgänge einen High-Pegel aufweisen. Über ein NAND-Gatter wird hiermit das RS-Flip-Flop 0 zur Ansteuerung einer LED ausgelöst.

Tritt ein Errorframe auf, ohne dass die Stromrichtungserkennung ein Senden des überwachten Teilnehmers erkannt hat, wird auch hier der Rücksetzzähler aktiviert, so dass während des insgesamt 11. Bits ein Rücksetzen der Errorframeerkennung erfolgt. Hier muss ebenfalls darauf geachtet werden, dass ein Rücksetzen innerhalb der 14 Bit aus Errorframe und Ruhephase stattfindet.

Die nächste Abbildung zeigt das Timingdiagramm des Errorfinders für verschiedene Fehlerrahmen. Der Schaltplan befindet sich im Anhang als Abbildung 47.

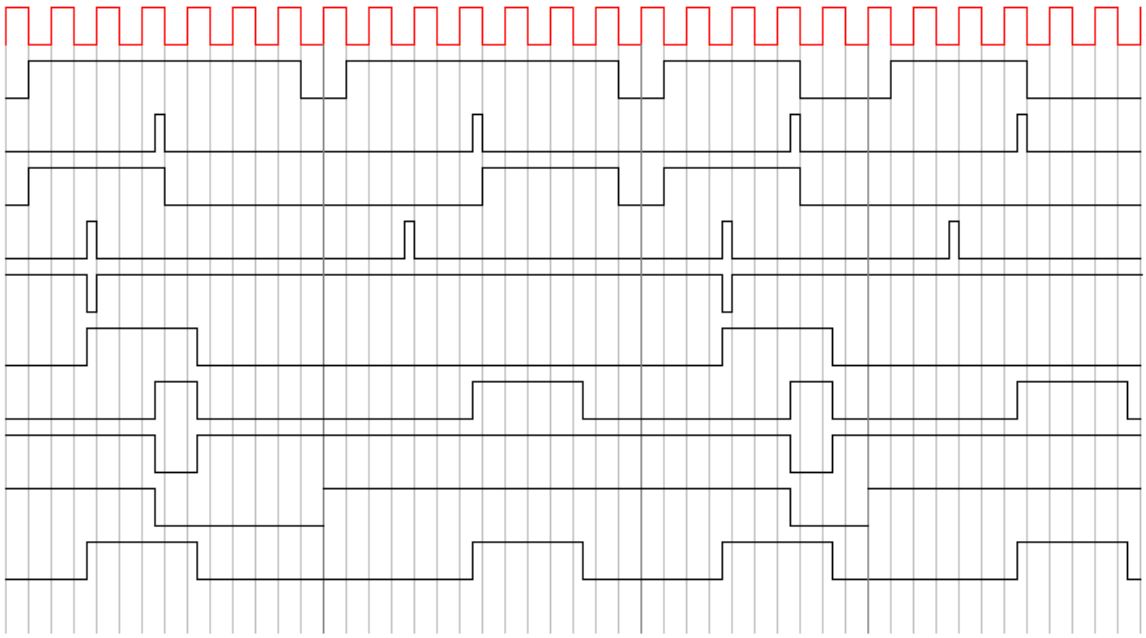


Abbildung 29: Timing-Prinzip des Errorfinders

1 Takt
2 invertierter Bus, Ausgang von Komparator 1
3 Ausgang Timer 1 im 6. Bit
4 Ausgang Komparator 2, Stromrichtungserkennung
5 Ausgang Timer 2 im 3. Bit
6 NAND aus 4 und !5, Strom zu Beginn der Low-Phase
7 Flip-Flop durch 6 gesetzt, eigener Teilnehmer
8 Flip-Flop durch 3 gesetzt, Errorframe
9 NAND aus 7 und 8, Errorframe eigener Teilnehmer
10 Flip-Flop aus 9 gesetzt, LED
gelöscht wird 10 durch Tastendruck
11 7 OR 8, Rücksetzen der Flip-Flops bei Rückkehr nach Low

Eine Neuentwicklung dieser Platine mit einem Prozessor und Auswertung auf Softwaregrundlage wäre aufgrund der bis hierhin erreichten Komplexität sinnvoll. Eine unter allen Umständen sichere Erkennung eines (auch passiven) Errorframes und seiner Herkunft erfordert es, eine Funktionalität zu entwickeln, die einem CAN-Busteilnehmer entspricht. Lediglich die Sendefunktion und das Puffern eingegangener Nachrichten ist nicht erforderlich.

Ein hierzu verwendeter Prozessor muss allerdings die notwendigen Berechnungen in sehr kurzer Zeit durchführen. Insbesondere bei hohen Busgeschwindigkeiten sind einfache Prozessoren wie zum Beispiel PICs nicht mehr geeignet, so dass die hier vorgestellte Schaltung als Einzelstück und für Kleinserien die preiswertere Alternative ist, wenn die Beschränkung auf aktive Errorframes akzeptiert werden kann.

14 Verschiedene Fehler auf dem Bus

Im folgenden werden einige Besonderheiten vorgestellt und erklärt, die bei der Erzeugung von Busfehlern auftreten können. Als Teilnehmer waren neben dem CAN-Controller für die DIDO537 mit der Zusatzplatine zur Fehlererzeugung beide Ports der CAN-AC2-PCI an den Bus angeschlossen. Alle drei SJA1000 sampeln den Bus bei 75% der Bitzeit.

Fehler im Datenfeld

Der Fehler im Datenfeld wird als Beispiel für jene Fehler untersucht, die in einem Teil des Datentelegrammes auftreten können, die ausschließlich dem Sender bekannt sind. Die theoretischen Grundlagen werden in Abschnitt 2.5.4 behandelt. Die zur Demonstration verwendete Nachricht trägt den Identifier 0x555 und die drei Datenbytes 0x00, 0x55 und 0x00. Auf Abbildung 30 ist das ungestörte Telegramm dargestellt. Deutlich zu sehen sind die High-Low-Folgen und die beiden 0-Bytes mit jeweils einem Stuffbit.

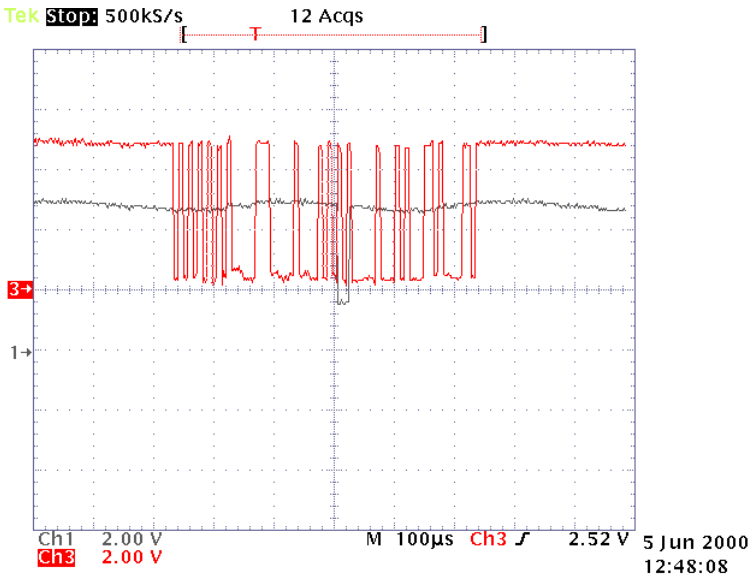


Abbildung 30: Normales Datenfeld

Erkennt ein Busteilnehmer als einziger einen Fehler, so sendet er einen Errorframe, der von allen anderen Teilnehmern mit einem Errorframe beantwortet wird. Bild 31 zeigt jedoch eine low-Phase, die erkennbar kürzer als 12 Bits ist,, da vor dem Beginn des Errorframes bereits ein dominantes Bit auf dem Bus war. Der Peak unmittelbar vor der Störung ist zu kurz, um den Abtastzeitpunkt zu erreichen, dieses Bit wird vom Sender bereits als Fehler erkannt. Der Errorframe beginnt demzufolge nach zwei dominanten Bits und wird

von den anderen Teilnehmern bereits nach vier Bits mit ihren Errorframes beantwortet. Es ergibt sich aus insgesamt zehn Bits für den Fehlerrahmen, dem gestörten High-Bit und dem Low-Bit davor eine zwölf Bitzeiten lange dominante Phase auf dem Bus, da wie erwähnt der kurze High-Pegel nicht ausgewertet wird. Die Länge kann gut mit dem Startbit und dem auf High endenden Identifier verglichen werden.

Beide Ports des CANalyzers protokollieren den Errorframe, bevor die Nachricht im zweiten Versuch empfangen wird.

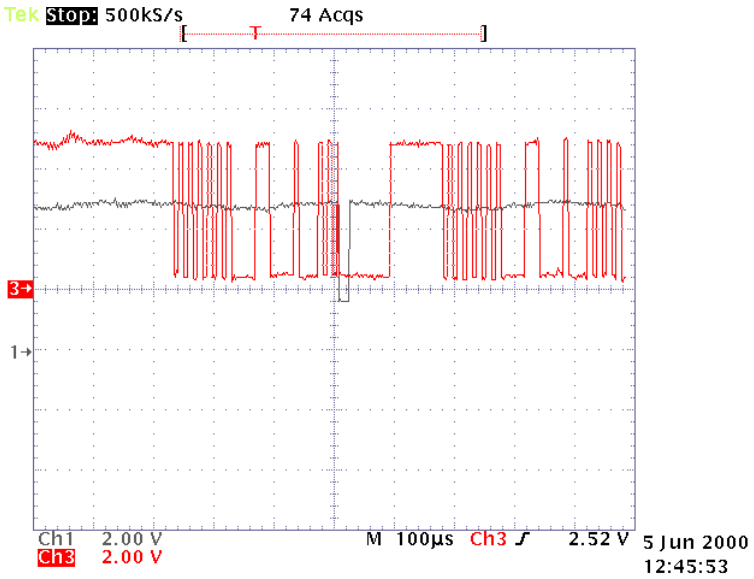


Abbildung 31: Datenfeld nach Störung

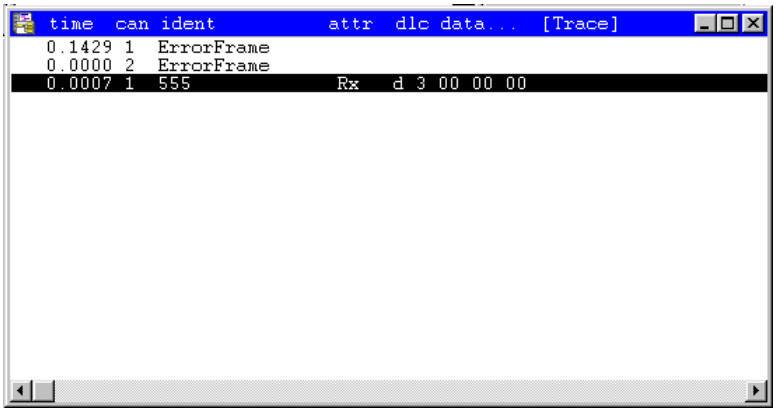


Abbildung 32: CANalyzer-Meldung bei der Fehlererzeugung

Fehler im Identifier

Durch das Überschreiben des Busses mit dominanten Bits im Identifier ist der Fehler direkt nicht zu erkennen, da dieser sich für den sendenden Teilnehmer so darstellt, als hätte er die Arbitrierung verloren. Abbildung 33 zeigt das Datenpaket und die noch nicht aufgeschaltete Störung.

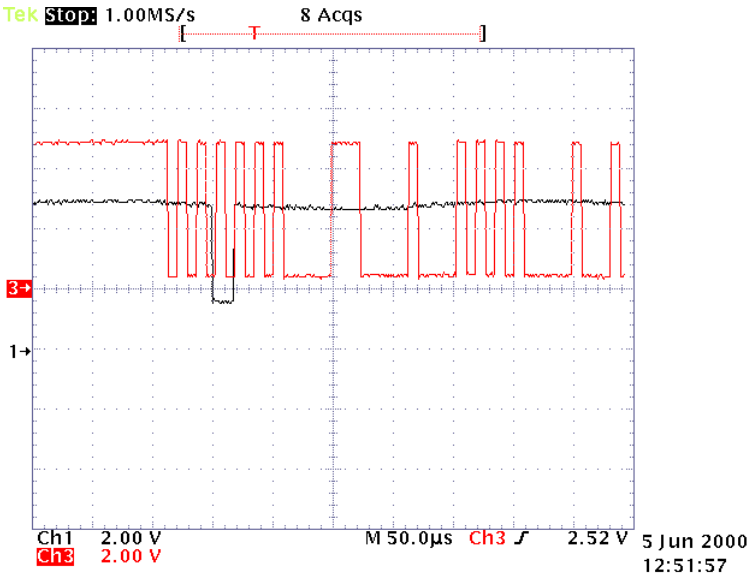


Abbildung 33: Normales Identifier-Feld

Das in Abbildung 34 wiedergegebene Ergebnis mit aufgeschalteter Störung ist wie folgt zu erklären: Der Sender registriert statt des übertragenen High-Bits ein Low-Bit auf dem Bus und stellt seine Aktivität ein. Da der Arbitrierungsverlust künstlich erzeugt wurde und kein anderer Teilnehmer sendet, bleibt der Bus high. Nach sechs Bits registrieren alle Buscontroller eine Verletzung der Stopfregel und senden einen Errorframe. Da der Fehler von allen Teilnehmern gleichzeitig bemerkt wird, ist die Low-Phase auf dem Bus sechs Bits lang. Auch hier beginnt der Teilnehmer nach den acht rezessiven Bits des Errorframe Delimiters erneut mit dem Senden.

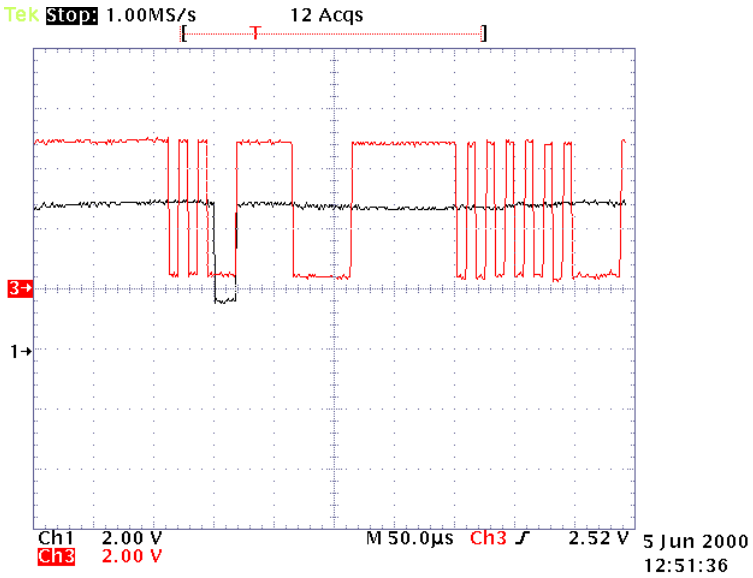


Abbildung 34: Identifier-Feld mit Störung

Aktiver und passiver Errorframe

Abbildung 35 zeigt noch einmal einen Fehler im 2. Datenbyte, der mit einem aktiven Errorframe beantwortet wird. Nachdem weitere Fehler erzeugt wurden, geht der CAN-Controller in den Fehlerpassiven Modus über und sendet nur noch einen passiven Errorframe, siehe Abbildung 36. Da mit Beginn des passiven Fehlerrahmens ein Pegelwechsel verbunden ist, registrieren die anderen Busteilnehmer diesen Errorframe erst nach dessen 6. Bit. Die Summe aus dem ursprünglichen Errorframe und den Antworten darauf ist genau zwölf Bit lang. Wieder beginnt der Sendevorgang nach den rezessiven Bits des Errorframes.

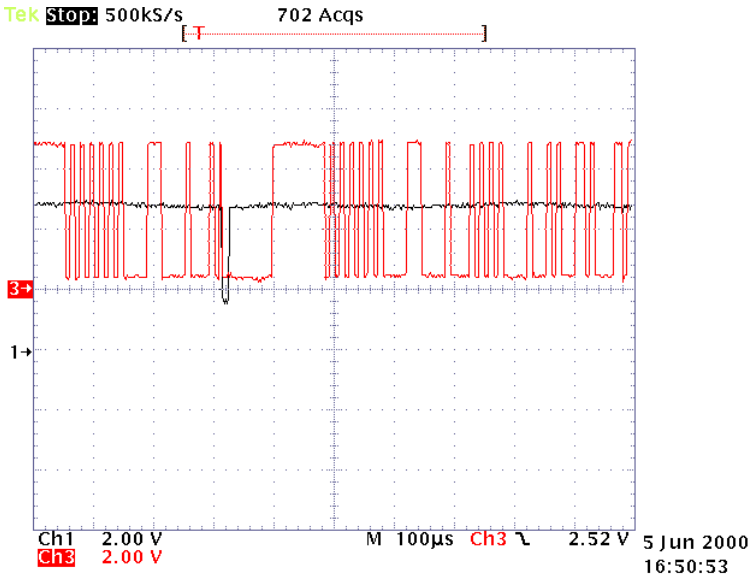


Abbildung 35: Aktiver Errorframe

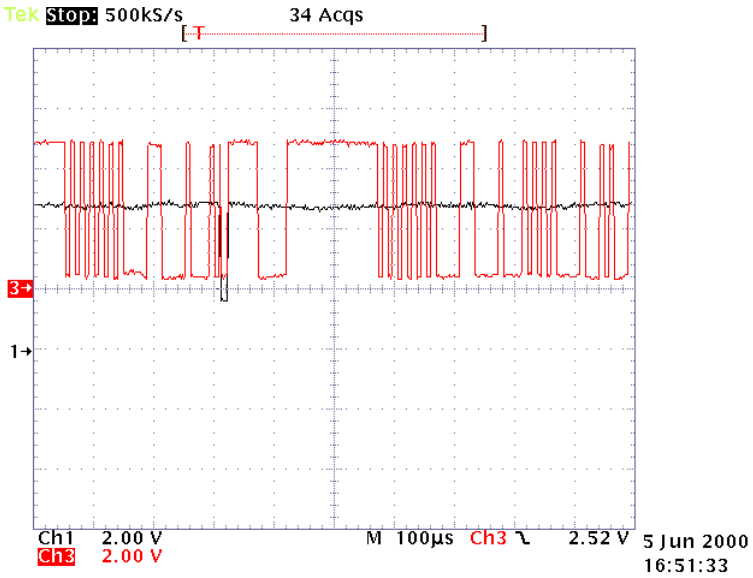


Abbildung 36: Passiver Errorframe

Low-Pegel im Idle-Zustand

Die drei folgenden Bilder zeigen verschiedene Fehler im Idle-Zustand des Busses. Abbildung 37 zeigt eine sehr kurze Low-Phase, auf die der Bus nicht reagiert. Die Busteilnehmer synchronisieren sich auf die fallende Flanke, können aber kein dominantes Bit und damit keinen Telegrammstart feststellen, weil zum Abtastzeitpunkt der Bus bereits wieder high ist.

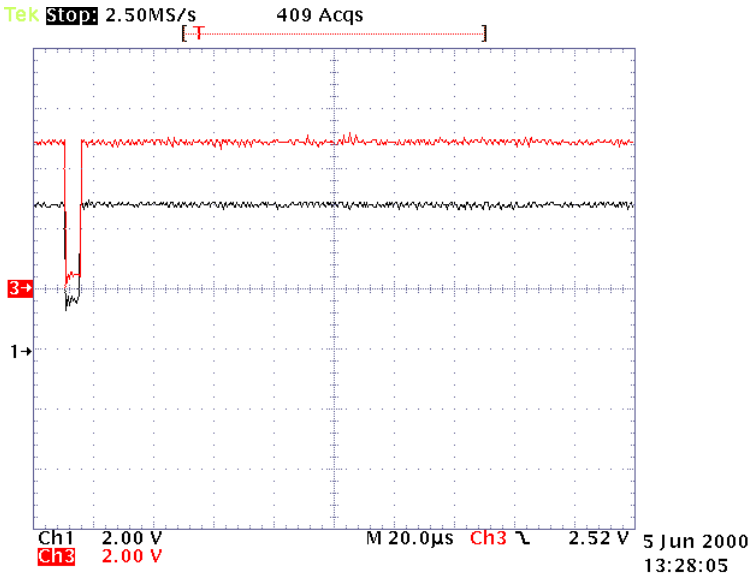


Abbildung 37: Kurze Low-Phase im Idle

Eine längere Low-Phase und ihre Auswirkung zeigt Abbildung 38. Die Buscontroller erkennen ein dominantes Bit und sehen es als Startbit eines Nachrichtentelegrammes an.

Nach sechs rezessiven Bits ist die Stopfregel verletzt, alle Teilnehmer reagieren mit sechs dominanten Bits. Die Erläuterung in [etsch94], dominante Bits im Telegrammzwischenraum würden mit einem Overloadframe beantwortet, ist nach diesem Ergebnis eine reine Definitionsfrage. Das zugrunde liegende Ereignis „Low-Bit im Telegrammzwischenraum“ kann durch Beobachten des Busses nicht vom Ereignis „Stopffehler im Identifier einer Nachricht“ unterschieden werden. Die sechs dominanten Bits können folglich ebenso als Teil eines Error- wie auch eines Overloadframe bezeichnet werden.

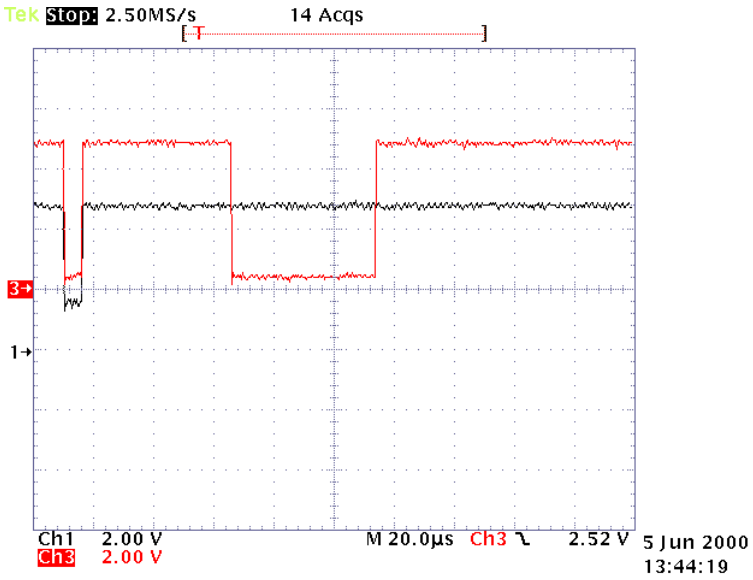


Abbildung 38: Lange Low-Phase im Idle

Abbildung 39 wurde mit einer Low-Phase erzeugt, deren Dauer zwischen denen der beiden vorhergegangenen Bilder lag. Die Aufnahme eines solchen Signalverlaufes gelang nur in einigen Prozent der Versuche, in allen anderen Fällen wurden Bilder nach Abbildung 38 erhalten. Auffällig ist die zwölf Bitzeiten lange Pegelabsenkung. Dieses Ergebnis entsteht aus einer Kombination der oben beschriebenen Abläufe: einige Controller erkennen die Low-Phase als Startbit, andere nicht. Somit gehen einige Teilnehmer davon aus, ein Telegramm zu empfangen, andere halten den Bus für leer. Nur die der 1. Gruppe stellen nach sechs High-Bits eine Codeverletzung fest und senden Errorframes. Die Teilnehmer der 2. Gruppe erkennen darin einen Telegrammanfang und beantworten die Verletzung der Stopfregel nach sechs dominanten Bits ihrerseits mit einem Errorframe.

Dass solche Busfehler nur von einem Teil der Teilnehmer erkannt werden und dieses Ergebnis zudem nicht sicher reproduzierbar ist, kann damit erklärt werden, dass die (heruntergeteilten) Takte der einzelnen Controller nicht völlig gleich sind. Außerdem sind die

Ansteuerungen der Bustreiber bei CAN-AC2-PCI und eigenem Teilnehmer nicht gleich, so dass hier aufgrund unterschiedlicher Flankensteilheiten die eingehenden Signale geringfügig unterschiedliche Dauern haben können.

Das Ergebnis verdeutlicht bereits für einen Bus mit weitgehend identischer Hardware, warum die Einstellung der Baudrate nicht direkt geschieht, sondern die kompliziertere Bestimmung der Bus Timing Parameter vorgezogen wird, die eine flexible Anpassung an den Aufbau des Busses ermöglicht.

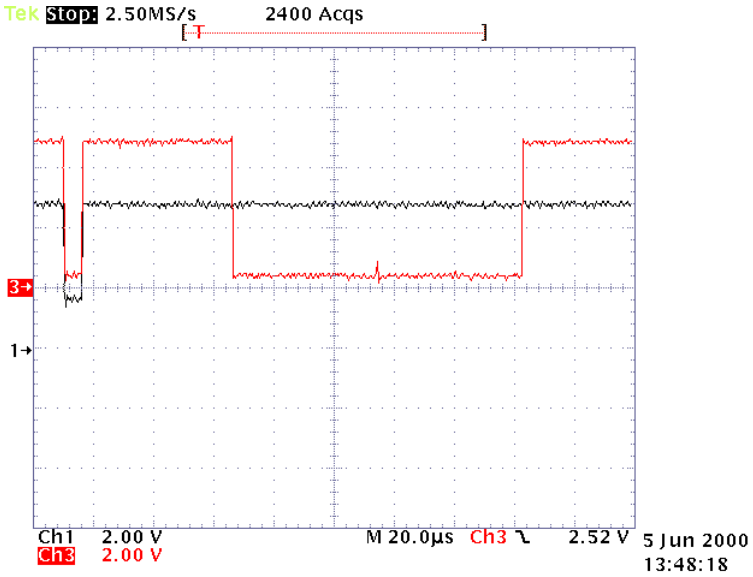


Abbildung 39: Mittlere Low-Phase im Idle

Kurzschluss

Die in diesem Versuch verwendete Schaltung ermöglicht lediglich einen dauerhaften Kurzschluss der beiden Datenleitungen. Ein Teilnehmer, der auf den kurzgeschlossen Bus zu senden versucht, schaltet sich nach kurzer Zeit in den Busoff-Zustand.

Unterbrechung

Wird lediglich eine Datenleitung unterbrochen, so gelingt es der verwendeten Hardware, über die zweite Datenleitung und Masse noch die Daten zu übertragen. Mit den in diesem Versuch eingesetzen Mitteln kann keine Einschränkung der Leistungsfähigkeit des Busses festgestellt werden, ein veränderter Einfluss von Störungen oder vermehrte Störungen bei langen Busleitungen wird nicht untersucht.

Werden beide Datenleitungen unterbrochen, so erscheinen beide Teile der Installation als separate Busse. Das Ergebnis hängt davon ab, ob in diesen Teilen jeweils nur ein

oder mehr als ein Teilnehmer angeschlossen sind. Ein Teilnehmer erkennt aufgrund des ausbleibenden Acknowledge-Bits seine Trennung vom Netzwerk und zieht sich zurück.

Sind mehrere Teilnehmer im Segment vorhanden, so registrieren sie die Störung nicht, da nach wie vor Acknowledge-Signale empfangen werden. Ist es erforderlich, auch eine solche Trennung des Busses zu registrieren, so muss der Anwender dies auf geeignete Art selbst implementieren. Sollen sowieso alle Teilnehmer regelmäßig senden, so muss lediglich überwacht werden, ob die Botschaften eintreffen. Eine andere Möglichkeit ist es, mit Remoteframes direkt einzelne Teilnehmer anzusprechen.

Unabhängig von diesen Ergebnissen kann es bei anderen CAN-Installationen jedoch durch das Fehlen von Abschlusswiderständen zu Störungen kommen. Da die in diesem Versuch verwendeten Bussegmente wenig länger als einen Meter waren, konnte dieses nicht beobachtet werden.

Anhang

A Software

A.1 Zugriff auf CAN-Funktionen des 80C515C

A.1.1 CANCTRLR.H

```
/*                                                                    */
/* CANCTRLR - CAN Controller Routines for use at a Infineon 80C515C ?C */
/* developed for Keil C51-compiler                                    */
/*                                                                    */
/* Stephan Pahlke, Sven Herzfeld                                      */
/* Exam FHH Summer 2000                                               */
/*                                                                    */
/* Version: 05/23/00                                                  */
/*                                                                    */
/* Exported definitions:                                              */
/*                                                                    */
/* CAN_DATA is a structure of can_byte[7] and can_length, both uchar  */
/* (used to return received messages)                                 */
/*                                                                    */
/* Exported routines:                                                 */
/*                                                                    */
/* can_init initializes the bus controller (baudrate in kBaud)        */
/*   currently, possible baud rates are 100, 125, 250, 500 and 1000 (kBaud) */
/*                                                                    */
/* can_db writes data into the registers AND sets DIR to 1 (transmit), as you */
/*   won't write data when you are not going to send them             */
/*                                                                    */
/* can_read sends consistency-checked data (checks, if a new message was */
/*   received while reading, and if yes, re-reads)                    */
/*                                                                    */
/* can_rcv defines message object register as a receive register      */
/*                                                                    */
/* can_inv declares message object register as invalid, so the controller */
/*   won't send anything or react on any received message which would fit into */
/*   the arbitration mask.                                            */
/*                                                                    */
/* can_ar sets the arbitration register (identifier)                  */
/*                                                                    */
/* can_gms sets the global filter mask                                */
/*                                                                    */
/* can_umlm sets the filter mask of message object register 15        */
/*                                                                    */
/* can_send sends previously defined data                             */
/*                                                                    */
/* can_status returns the CAN status byte and deletes pending interrupts */
/*                                                                    */
/* can_state returns the CAN status byte and deletes pending interrupts plus */
/*   resets RXOK und TXOK and sets last error code LEC to 7           */
/*                                                                    */
/* can_msglst returns 1 if a message was lost because the receive register was */
/*   not read before a new message fitting into the arbitration mask came in. */
/*                                                                    */
/* can_newdat checks whether a new message was received or not        */
/*   (Note: NEWDAT is not deleted. To delete NEWDAT, you have to read the */
/*   message using can_rcv)                                           */
/*                                                                    */
```

```
/*                                                                      */
/* Struct CAN_DATA, returned by can_read                                */
/*                                                                      */

extern struct CAN_DATA
 {
  unsigned char can_byte[7];            /* 8 data bytes of which can_length    */
  unsigned char can_length;             /* are valid, others are unspecified   */
};



/*                                                                      */
/* CAN bus initialisation routine, send bus speed (baudrate) as parameter,     */
/* given in kBaud (e. g. 125 = 125 kBaud, 1000 = 1 Mbaud)               */
/*                                                                      */
/* If unknown, 125 is the default                                       */
/*                                                                      */

extern void can_init (unsigned short Bus_speed);


/*                                                                      */
/* Define data of message in message object register mor (standard frame)      */
/*
/* Always include eight data bytes, don't care about the value you give bytes  */
/* which will not be sent.                                              */
/*                                                                      */
/* You'll probably want to send the given data, so the message object register */
/* is configured as a transmitting one.                                 */
/*                                                                      */

extern void can_db (unsigned char mor, unsigned char databyte[7], unsigned char datalength);


/*                                                                      */
/* Returns a consistency-checked eight-byte data array from message object     */
/* register mor plus the value of its data lenght code DLC              */
/*                                                                      */
/* Consistency is checked only in valid bytes (which might be zero), the rest  */
/* is undefined.                                                        */
/*                                                                      */

extern struct CAN_DATA can_read (unsigned char mor);


/*                                                                      */
/* Define message object register mor as receiving register            */
/*                                                                      */

extern void can_rcv (unsigned char mor);


/*                                                                      */
/* Send remote telegram                                                 */
/* This makes only sense when used on a receive message object register        */
/* You need to include the datalength, otherwise bit errors might occur when   */
/* transmit and receive messages are sent the same time.                */

extern void can_remote (unsigned char mor, unsigned char datalength);


/*                                                                      */
/* Define message object register mor as invalid                       */
/* This message object register will neither send or receive anything unless   */
/* you re-activate it using can_db or can_rcv                           */
/*                                                                      */

extern void can_inv (unsigned char mor);
```

```
/*                                                                      */
/* Define arbitration of message mor (standard frame)                   */
/*                                                                      */
/* This will be used as acceptance filter as well as als identifier when a     */
/* message is transmitted from this message object register.            */
/*                                                                      */
/* Note that the arbitration is only 11 bit, the five highest bits are unused. */
/* It's not really checked, but as the identifier ushort variable is shifted   */
/* left fife bit, the unused bits are lost anyway.                      */
/*                                                                      */
/* However, to be on the safe side, you should use 0x07FF = 2047 (decimal) as  */
/* highest identifier.                                                  */

extern void can_ar (unsigned char mor, unsigned short identifier);



/*                                                                      */
/* Define global acceptance filter mask (standard frame)                */
/*                                                                      */
/* Bits set to 0 mean "Do not compare this bit position to the arbitration     */
/* register, when a message comes in." (Standard should be 0x07FF = 2047)      */
/* This is also the default value set by can_init.                      */
/*                                                                      */
/* Setting GMS to 0 and much traffic on the bus is an easy way to keep the     */
/* CPU busy, so use this feature with care!                             */
/*                                                                      */

extern void can_gms (unsigned short identifier)


/*                                                                      */
/* Define acceptance filter mask of last message (15, standard frame)          */
/*                                                                      */
/* Register 15 - which is receive-only - can be configured as a "basic CAN"    */
/* device or any other way by its own acceptance filter mask. When this        */
/* feature is not used, the mask should be set to 0x07FF = 2047 which is also  */
/* the default set by can_init.                                         */
/*                                                                      */
/* For message object register 15, both GMS AND UMLM are valid, this means,    */
/* if one bit of them is 0, the respective arbitration bit is ignored.          */
/*                                                                      */
/* Setting UMLM to 0 and much traffic on the bus is an easy way to keep the    */
/* CPU busy, so use this feature with care!                             */
/*                                                                      */

extern void can_umlm (unsigned short identifier)


/*                                                                      */
/* Send message in message object register mor                          */
/*                                                                      */

extern void can_send (unsigned char mor);


/*                                                                      */
/* Returns CAN status byte                                              */
/*                                                                      */
/* bit 7 : Busoff status    bit 6 : Error warning status                */
/* bit 5 : unused           bit 4 : Received message successfully       */
/* bit 3 : Transmitted message successfully                             */
/* bit 0-2: Last error code                                             */
/* 0 : no error  1 : stuff error  2 : form error (wrong signal in fixed part)  */
/* 3 : ack error (got no acknowledgement)  4 : bit 1 error (dominant detected) */
/* 5 : bit 0 error (recessive detected, also occurs in busoff state after 11   */
/* recessive bits) 6 : CRC error                                        */
/*                                                                      */
/* Note: calling can_status resets a pending interrupt request          */
/*                                                                      */

extern unsigned char can_status (void);

/*                                                                      */
```

```
/* Returns and resets CAN status byte                                        */
/*                                                                           */
/* Nearly the same as before, but also sets RXOK and TXOK to 0 and LEC to    */
/* 7. 7 is unused, but allowed, so this is set to check for updates.         */
/*                                                                           */

unsigned char can_state (void);


/*                                                                           */
/* Returns the state of MSGLST register, if set, a message was lost          */
/*                                                                           */
/* Note: useful only for receive-objects, for transmit-objects, the value is */
/*   CPUUPD                                                                  */
/*                                                                           */

extern unsigned char can_msglst (unsigned char mor);




/*                                                                           */
/* Returns 1 if NEWDAT of message object register mor is set, else 0         */
/*                                                                           */

extern bit can_newdat (unsigned char mor);
```

A.1.2 CANCTRLR.C

```
/*                                                                          */
/* CANCTRLR - CAN Controller Routines for use at a Infineon 80C515C ?C      */
/* developed for Keil C51-compiler                                          */
/*                                                                          */
/* Stephan Pahlke, Sven Herzfeld                                            */
/* Exam FHH Summer 2000                                                     */
/*                                                                          */
/* FOR INFORMATION ABOUT THE USED ROUTINES AND MORE, PLEASE CHECK THE       */
/* CANCTRLR.H HEADER FILE.                                                  */
/*                                                                          */
/* Version: 05/23/00                                                        */
/*                                                                          */
/* History:                                                                 */
/*                                                                          */
/* 4/11/00: start of implementation                                         */
/* 4/13/00: init, db and send functions successfully tested                  */
/* 4/14/00: receive functions successfully tested                           */
/* 4/17/00: "while (MOR > 1)" added to ease use of differents MORs          */
/* 4/18/00: "MCR_address" added to save memory                              */
/* 4/18/00: disabled all CAN interrupts as they are not needed in main program */
/* 4/18/00: added status and global filter functions                        */
/* 5/11/00: can_read: read DLC first, only read this number of bytes. This  */
/*          makes the routine slightly quicker, if less than eight bytes are */
/*          received.                                                       */
/* 5/11/00: corrected can_remote, now setting DLC                           */
/* 5/11/00: improved comments                                               */
/* 5/23/00: tested 100 kBaud                                                */
/* 5/23/00: corrected can_newdat ("&&-error")                               */


#include <canreg.h>                   /* CAN register definitions           */
#include <intc15c.h>                  /* 80C515C interrupts definitions     */
#include <regc515c.h>                 /* 80C515C register definitions       */



// some data definitions
unsigned char pdata canreg[256];      /* array for access to CAN registers  */

unsigned char  IE = 0;                /* all CAN interrupts enabled         */
unsigned char SIE = 0;                /* Status interrupts enabled          */
unsigned char EIE = 0;                /* Error interrupts enabled           */

struct CAN_DATA                       /* definition of structure type       */
{                                     /* can data to return                 */
  unsigned char can_byte[7];          /* eight data bytes                   */
  unsigned char can_length;           /* and one byte DLC                   */
};


//****************************************************************************
/*                                                                          */
/* Calculate the address of the MCR0 of a given message object register     */
/*                                                                          */

unsigned char MCR_address (unsigned char MCR) /* calculate pointer to MCR0_Mx  */

{
  data unsigned char pdata *address;  /* pointer to MCR                     */

  address = &MCR0_M1;                 /* set pointer to first MCR           */
  while (MCR > 1)                     /* go to message object register      */
  {
    address = address + 0x10;         /* the next message object is located */
    MCR--;                            /* at a 16 bytes higher address       */
  }
  return (address);                   /* return value                       */
}
```

```
//*****************************************************************************
/*                                                                           */
/* CAN bus initialisation routine, send bus speed (baudrate) as parameter,   */
/* given in kBaud (e. g. 125 = 125 kBaud, 1000 = 1 Mbaud)                    */
/*                                                                           */

void can_init (unsigned short Bus_speed)

{
  data unsigned char pdata *address;    /* pointer to MCR                    */
  data unsigned char loop;              /* loop counter                      */

  data bit Bus_speed_ok;                /* flag bit                          */
  Bus_speed_ok = 1;

  SYSCON &= 0xFC;                       /* XMAP0=0, XMAP1=0                  */
  XPAGE = 0xF7;                         /* CAN memory space in XRAM          */

  GMS0   = 0xFF;  GMS1   = 0xFF;        /* Global mask short                 */
  UGML0  = 0xFF;  UGML1  = 0xFF;        /* Global mask long                  */
  LGML0  = 0xFF;  LGML1  = 0xFF;
  UMLM0  = 0xFF;  UMLM1  = 0xFF;        /* Last message mask                 */
  LMLM0  = 0xFF;  LMLM1  = 0xFF;

  CR = 0x41;                            /* INIT = 1, CCE = 1 (access baudrate) */


  // WARNING: 100 kBaud is currently untested!
  if (Bus_speed == 100)                 /* 100 kBaud                         */
  {
    BTR0 = 0xC4;                        /* set Bus timing register           */
    BTR1 = 0x5C;
    Bus_speed_ok = 0;                   /* flag OK, we know this bus speed   */
  }

  if (Bus_speed == 250)                 /* 250 kBaud                         */
  {
    BTR0 = 0xC1;                        /* set Bus timing register           */
    BTR1 = 0x68;
    Bus_speed_ok = 0;                   /* flag OK, we know this bus speed   */
  }

  if (Bus_speed == 500)                 /* 500 kBaud                         */
  {
    BTR0 = 0xC0;                        /* set Bus timing register           */
    BTR1 = 0x68;
    Bus_speed_ok = 0;                   /* flag OK, we know this bus speed   */
  }

  if (Bus_speed == 1000)                /* 1 MBaud                           */
  {
    BTR0 = 0x80;                        /* set Bus timing register           */
    BTR1 = 0x25;
    Bus_speed_ok = 0;                   /* flag OK, we know this bus speed   */
  }

  if (Bus_speed_ok != 0)                /* if nothing else is given what we  */
  {                                     /* know, we consider 125 kBaud as    */
    BTR0 = 0xC4;                        /* default (without further annotation)*/
    BTR1 = 0x49;                        /* (after a lot of error frames, the */
  }                                     /* controller will reach busoff state) */

  /* Remark: as we don't use Bus_speed_ok as a return value, it is not       */
  /* necessary to set it even if 125 kBaud was asked                         */

  CR = 0x01;                            /* CCE = 0 (access disable baudrate) */
  SR = 0xE7;                            /* Clear TXOK and RXOK               */


  // Now declare all message object registers as invalid

  address = &MCR0_M1;                   /* set address to MCR0_M1            */
  for (loop = 1; loop < 16; loop++)
  {
```

```
    *address = 0x55;                    /* MCR0_Mx : message not valid         */
    address = address + 0x10;           /* go to next message object           */
  }

  IEN2 = 2;                             /* Enable CAN interrupt (ECAN = 1)     */
  CR = 0;                               /* INIT = 0                            */

  if ( IE) CR |= 0x02;                  /* Enable global int_can               */
  if (SIE) CR |= 0x04;                  /* Enable can_status_int               */
  if (EIE) CR |= 0x08;                  /* Enable can_error_int                */

}


//*****************************************************************************
/*                                                                           */
/* Define data of message standard frame                                     */
/*                                                                           */
/* First byte to transmit is the message object register (variable 'mor'),   */
/* followed by eight data bytes and one byte data lenght which may be in the */
/* range from 0 to 8.                                                        */
/*                                                                           */
/* Always include eight data bytes, don't care about the value you give bytes*/
/* which will not be sent. ("Always include" is obsolete, you have to use the*/
/* eight bytes array, anyway.)                                               */
/*                                                                           */
/* A data lenght of more than eight might bring your bus into some trouble, so*/
/* we strictly limit the value to this maximum.                              */
/*                                                                           */

void can_db (unsigned char mor, unsigned char databyte[7], unsigned char datalength)

{
  data unsigned char pdata *destination;/* pointer to MCR0_Mx                */
  data unsigned char data *source;      /* pointer to structure              */
  data unsigned char loop;              /* loop counter                      */

  if ((mor < 15) && (mor > 0))          /* 1 <= MCR <= 14 (15 is receive only)*/
  {

    destination = MCR_address (mor);    /* calculate pointer to MCR0_Mx      */
    source = &databyte[0];              /* set pointer to given data array   */

    *(destination + 1) = 0xFA;          /* MCR1_Mx, CPUUPD=1                 */
    if (datalength > 8) datalength = 8; /* limit datalenght to allowed maximum*/
    datalength = datalength << 4;       /* shift left four bit               */

    *(destination + 6) = 0x08 | datalength; /* MCFG_Mx, DLC=0, XTD=0, DIR=1  */
                                        /* (transmit), OR DLC as given       */

    for (loop = 1; loop < 9; loop++)    /* loop eight times                  */
    {
      *(destination + 6 + loop) = *source; /* transmit data indirectly,      */
       source++;                        /* eight byte                        */
    }

                                        /* finally, set MCR_Mx:              */
    *destination       = 0xA5;          /* MSGVAL=1, TXIE=1, RXIE=0, INTPND=0*/
    *(destination + 1) = 0x55;          /* RMTPND=0, TXRQ=0, CPUUPD=0, NEWDAT=0*/
  }
}



//*****************************************************************************
/*                                                                           */
/* Returns data bytes of message control register given in argument.         */
/*                                                                           */
/* The answer always includes eight data bytes, if the received message      */
/* contains less, the rest is unspecified.                                   */
/*                                                                           */
/* ATTENTION: if the message control register number is not valid ([1..15]), */
/* the result are eight unspecified bytes and a zero data lenght.            */
/*                                                                           */
```

```
struct CAN_DATA can_read (unsigned char mor)

{
  struct CAN_DATA can_struct;           /* structure to give back              */

  data unsigned char pdata *source;     /* pointer to MCR0_Mx                  */
  data unsigned char data *destination; /* pointer to structure                */
  unsigned char loop;                   /* loop counter                        */

  if ((mor < 16) && (mor > 0))          /* 1 <= MCR <= 15                      */
  {
    source = MCR_address (mor);         /* calculate pointer to MCR0_Mx        */
    destination = &can_struct.can_byte[0]; /* set pointer to structure         */

    do
    {
      *(source + 1) = 0xFD;             /* reset NEWDAT                        */

      can_struct.can_length = *(source + 6) >> 4; /* read DLC and store value  */

      if (can_struct.can_length)        /* if more than zero bytes were recv.  */
      {                                 /* for would work at least once        */
        for (loop = 1; loop <= can_struct.can_length; loop++)
                                        /* loop onve for every data byte       */
        {
          *destination = *(source + 6 + loop); /* transmit data indirectly,    */
          destination++;                /* total number as given before        */
        } /* for (loop ... */
      } /* if (can_struct.can_length) */
    }
    while ( *(source + 1) & 2);         /* re-read, if NEWDAT set, in order to */
  }                                     /* return only consistent data. If it  */
                                        /* was set, some data bytes might have */
                                        /* changed meanwhile.                  */
  else can_struct.can_length = 0;       /* if register unavailable, return 0   */
  return (can_struct);                  /* return complete structure           */
}


//*****************************************************************************
/*                                                                           */
/* Define message object register for receive (standard frame)               */
/*                                                                           */

void can_rcv (unsigned char mor)

{
  data unsigned char pdata *address;    /* pointer to MCR0_Mx                  */

  if ((mor < 16) && (mor > 0))          /* 1 <= MCR <= 15                      */
  {
    address = MCR_address (mor);        /* calculate pointer to MCR0_Mx        */
    *(address + 1) = 0xFA;              /* MCR1, CPUUPD=1                      */
    *(address + 6) = 0x00;              /* MCFR DLC=0, XTD=0, DIR=0 (receive)  */
    *address = 0x98;                    /* MSGVAL=1, TXIE=0, RXIE=1, INTPND=0  */
    *(address + 1) = 0x55;              /* RMTPND=0, TXRQ=0, CPUUPD=0, NEWDAT=0 */
  }
}


//*****************************************************************************
/*                                                                           */
/* Send remote telegram                                                      */
/*                                                                           */

void can_remote (unsigned char mor, unsigned char datalength)

{
  data unsigned char pdata *address;    /* pointer to MCR0_Mx                  */

  if ((mor < 16) && (mor > 0))          /* 1 <= MCR <= 15                      */
  {
    address = MCR_address (mor);        /* calculate pointer to MCR0_Mx        */
```

```
    if (datalength > 8) datalength = 8; /* limit datalenght to allowed maximum */
    datalength = datalength << 4;       /* shift left four bit                 */

    *(address + 6) &= 0x0F;             /* clear DLC                           */
    *(address + 6) |= datalength;       /* set DLC                             */
    *(address + 1) = 0xED;              /* TXRQ=1, NEWDAT=0                    */
  }
}


//*****************************************************************************
/*                                                                           */
/* Define message object register as invalid                                 */
/*                                                                           */
/* If you need it again, use can_db or can_rcv.                              */
/*                                                                           */

void can_inv (unsigned char mor)

{
  data unsigned char pdata *address;    /* pointer to MCR0_Mx                  */

  if ((mor < 16) && (mor > 0))          /* 1 <= MCR <= 15                      */
  {
    address = MCR_address (mor);        /* calculate pointer to MCR0_Mx        */
    *address = 0x7F;                    /* MCR0, MSGVAL=1                      */
  }
}


//*****************************************************************************
/*                                                                           */
/* Define arbitration of message (standard frame)                            */
/*                                                                           */
/* This will be used as acceptance filter as well as identifier when a message */
/* is transmitted from this message object register.                         */
/*                                                                           */
/* Note that the arbitration is only 11 bit, the five highest bits are unused. */
/* It's not really checked, but as the identifier ushort variable is rotated   */
/* left by fife bit, the unused bits are lost anyway.                        */
/*                                                                           */

void can_ar (unsigned char mor, unsigned short identifier)

{
  data unsigned char pdata *address;    /* pointer to MCR0_Mx                  */

  if ((mor < 16) && (mor > 0))          /* 1 <= MCR <= 15                      */
  {
    address = MCR_address (mor);        /* calculate pointer to MCR0_Mx        */
    *address = 0x7F;                    /* MSGVAL=0, Message not valid         */
    identifier = identifier << 5;       /* multiply by 2^5 = 32                */
    *(address + 2) = identifier >> 8;   /* set UAR0 to given bits              */
    *(address + 3) = identifier & 0xFF; /* set UAR1 to given bits              */
    *(address + 4) = 0x0;               /* LAR0 always 0 for standard frames   */
    *(address + 5) = 0x0;               /* we don't use extended frames        */
    *address = 0xBF;                    /* MSGVAL=1, Message valid again       */
  }
}


//*****************************************************************************
/*                                                                           */
/* Define global acceptance filter mask (standard frame)                     */
/*                                                                           */

void can_gms (unsigned short identifier)

{
  identifier = identifier << 5;         /* multiply by 2^5 = 32                */
  GMS0 = identifier >> 8;               /* set UMLM0 to given bits             */
  GMS1 = identifier & 0xFF;             /* set UMLM1 to given bits             */
}
```

```
//***************************************************************************
/*                                                                           */
/* Define acceptance filter mask of last message (15, standard frame)        */
/*                                                                           */

void can_umlm (unsigned short identifier)

{
  MCR0_M15 = 0x7F;                      /* MSGVAL=0, Message not valid         */
  identifier = identifier << 5;         /* multiply by 2^5 = 32                */
  UMLM0 = identifier >> 8;              /* set UMLM0 to given bits             */
  UMLM1 = identifier & 0xFF;            /* set UMLM1 to given bits             */
  LMLM0 = 0x0;                          /* LMLM0 always 0 for standard frames  */
  LMLM1 = 0x0;                          /* we don't use extended frames        */
  MCR0_M15 = 0xBF;                      /* MSGVAL=1, Message valid again       */
}


//***************************************************************************
/*                                                                           */
/* Send message in message object register                                   */
/*                                                                           */

void can_send (unsigned char mor)

{
  data unsigned char pdata *address;    /* pointer to MCR0_Mx                  */

  if ((mor < 15) && (mor > 0))          /* 1 <= MCR <= 14                      */
  {
    address = MCR_address (mor);        /* calculate pointer to MCR0_Mx        */
    SR = SR & 0xF7;                     /* clear TXOK, allow check             */
    *(address + 1) = 0xE7;              /* set NEWDAT, send message            */
  }
}


//***************************************************************************
/*                                                                           */
/* Returns 1 if NEWDAT of message object register is set, else 0             */
/*                                                                           */

bit can_newdat (unsigned char mor)

{
  data unsigned char pdata *address;    /* pointer to MCR0_Mx                  */

  if ((mor < 16) && (mor > 0))          /* 1 <= MCR <= 15                      */
  {
    address = MCR_address (mor);        /* calculate pointer to MCR0_Mx        */
    if ( (*(address + 1) & 3) == 2)     /* check state of NEWDAT               */
      return (1);                       /* return 1, if set, ...               */
    else return (0);                    /* else 0                              */
  }
  else return (0);                      /* return 0, if unknown MCR            */
}


//***************************************************************************
/*                                                                           */
/* Returns CAN status byte                                                   */
/*                                                                           */

unsigned char can_status (void)

{
  return (SR);
}
```

```
//***************************************************************************************
/*                                                                                      */
/* Returns and resets CAN status byte                                                   */
/*                                                                                      */

unsigned char can_state (void)

{
  data unsigned char statusbyte;
  statusbyte = SR;
  SR = 0x07;                            /* set last error code to 7 (unused)   */
  return (statusbyte);
}


//***************************************************************************************
/*                                                                                      */
/* Returns value of MSGLST bit (for transmit-objects: CPUUPD)                          */
/*                                                                                      */

unsigned char can_msglst (unsigned char mor)

{
  data unsigned char pdata *address;    /* pointer to MCR0_Mx                  */

  if ((mor < 16) && (mor > 0))          /* 1 <= MCR <= 15                      */
  {
    address = MCR_address (mor);        /* calculate pointer to MCR0_Mx        */
    if ( (*(address + 1) && 12) == 8)   /* check state of MSGLST               */
        return (1);                     /* return 1, if set, ...               */
    else return (0);                    /* else 0                              */
  }
  else return (0);                      /* return 0, if unknown MCR            */
}


//***************************************************************************************
/*                                                                                      */
/* CAN interrupt routine                                                                */
/*                                                                                      */
/* If CANCTRLR is used as a linked object file in an application, there is     */
/* nothing more to do than to reset the status flags. However, using this file */
/* directly, you can create own interrupt-controlled functions.                */
/*                                                                                      */

void can_interrupt (void) interrupt CANI

{

  unsigned char status, intreg;
  while (IR != 0)
  {
    intreg = IR;
    status = SR; SR = 0;                /* read and reset CAN status           */
    switch (intreg)
    {
      case 1:                           /* status and error interrupt          */
              if (SIE)                  /* status interrupts                   */
              {
                 if (status & 0x08)     /* transmit interrupt                  */
                     {/* insert here whatever you want me to do */}
                                        /* currently unused                    */
                 if (status & 0x10)     /* receive interrupt                   */
                     {}                 /* currently unused                    */
                 if (status & 0x07)     /* erroneous transfer                  */
                     {}                 /* currently unused                    */
              }
              if (EIE)                  /* error interrupts                    */
              {
                 if (status & 0x40)     /* EWRN has changed                    */
                     {}                 /* currently unused                    */
                 if (status & 0x80)     /* BUSOFF has changed                  */
                     {}                 /* currently unused                    */
              }
```

```
            break;
     case 3:                            /* message 1 interrupt                */
            MCRO_M1 = 0xFD;             /* reset INTPND                       */
            if (status & 0x08)          /* transmit interrupt                 */
              {}                        /* currently unused                   */
            if (status & 0x10)          /* receive interrupt                  */
              {}                        /* currently unused                   */
            break;
     case 4:                            /* message 2 interrupt                */
            MCRO_M2 = 0xFD;             /* reset INTPND                       */
            if (status & 0x08)          /* transmit interrupt                 */
                  {}                    /* currently unused                   */
            if (status & 0x10)          /* receive interrupt                  */
                  {}                    /* currently unused                   */
            break;
    /* with these examples, you should be able to determine how to create the  */
    /* calls for message object registers 3-15                                 */
    }
  }
}
```

A.2 Zugriff auf das CAN Control Board der DIDO537

A.2.1 CANCTSJA.H

```
/*                                                                          */
/* CANCTSJA - CAN Controller Routines for SJA1000 at a 80C537 ?C            */
/*                                                                          */
/* Stephan Pahlke, Sven Herzfeld                                            */
/* Exam FHH Summer 2000                                                     */
/*                                                                          */
/* Version: 05/23/00                                                        */
/*                                                                          */
/* Exported definitions:                                                    */
/*                                                                          */
/* CAN_DATA is a structure of can_byte[7] and can_length, both uchar, plus  */
/* can_id (unsigned short) and can_RTR (unsigned char)                      */
/*                                                                          */
/* Exported routines:                                                       */
/*                                                                          */
/* can_init initializes the bus controller (baudrate in kBaud)              */
/*   currently, possible baud rate is currently only 125 (kBaud)            */
/*                                                                          */
/* can_db writes data into the registers (including DLC)                    */
/*                                                                          */
/* can_read returns data                                                    */
/*                                                                          */
/* can_ar sets the arbitration register (identifier)                        */
/*                                                                          */
/* can_send sends previously defined data                                   */
/*                                                                          */
/* can_remote sends a remote telegram (from transmit buffer)                */
/*                                                                          */
/* can_status returns the CAN status byte and deletes pending interrupts    */
/*                                                                          */
/* can_msglst checks whether a message was lost or not and clears data overrun */
/*                                                                          */
/* can_newdat checks whether a new message was received or not              */
/*                                                                          */


/*                                                                          */
/* Structure CAN_DATA, returned by can_read                                 */
/*                                                                          */

extern struct CAN_DATA
 {
  unsigned char can_byte[7];            /* eight data bytes                 */
  unsigned char can_length;             /* one byte telegram length         */
  unsigned short can_ID;                /* identifier of message            */
  unsigned char can_RTR;                /* set to 1 if a remote telegram was
                                           received, otherwise 0.           */
};


/*                                                                          */
/* CAN bus initialisation routine, send bus speed (baudrate) as parameter,  */
/* given in kBaud (e. g. 125 = 125 kBaud, 1000 = 1 Mbaud)                   */
/*                                                                          */
/* If unknown, 125 is the default                                           */
/*                                                                          */
/* Known values are 100, 250, 500 and 1000 (and, of course, 125)            */
/*                                                                          */
/* Filter is the receive buffer acceptance filter, mask the receive buffer  */
/* acceptance mask. The SJA 1000 only filters the eight most significiant bits */
/* of a received message, so you have to decide to operate your CAN bus with */
/* only eight bit identification - or to keep the CPU busy with the rest of */
/* the acceptance filtring with the CPU. In this case, you might get unexpected*/
/* answers from the controller as he also reacts on other remote telegrams  */
/* meeting the eight bit acceptance criteria.                               */
/*                                                                          */
/* A third way is left open for you: write your own CAN library and use the */
/* peliCAN mode of the SJA1000 ...                                          */
/*                                                                          */
```

```
extern void can_init (unsigned short Bus_speed, unsigned char filter, unsigned char mask);


/*                                                                          */
/* Define data of transmit frame                                            */
/*                                                                          */
/* Always include eight data bytes, don't care about the value you give bytes  */
/* which will not be sent.                                                  */
/*                                                                          */
/* As DLC is four bit, you might define a datalength of more than eight bytes, */
/* however, as this may cause difficulties on the bus, the value is limited to */
/* eight by the routine. (The SJA1000 does not worry but send.)  */

extern bit can_db (unsigned char databyte[7], unsigned char datalength);


/*                                                                          */
/* Returns a eight-byte data array from receiving message object plus the      */
/* value of its data lenght code DLC, the complete identifier (-> can_init)    */
/* and one byte RTR. Furthermore, the receive buffer is released so that       */
/* another message may become accessible.                                   */
/*                                                                          */
/* For details about the data format, see documentation to CAN_DATA up in this */
/* file.                                                                    */
/*                                                                          */
/* RTR must be read as the SJA1000 works in basic CAN mode, which does not     */
/* include automatic data frame transmission after a remote frame was          */
/* received.                                                                */

extern struct CAN_DATA can_read (void);


/*                                                                          */
/* Send remote telegram                                                     */
/*                                                                          */
/* It will send from the TRANSMIT buffer, which, in order to avoid bit errors, */
/* must have been set to the correct data length.                           */
/*                                                                          */

extern void can_remote (void);


/*                                                                          */
/* Define arbitration of transmit message                                   */
/*                                                                          */
/* Note that the arbitration is only 11 bit, maximum is 0x07FF = 2047 (decimal)*/
/*                                                                          */

extern void can_ar (unsigned short identifier);


/*                                                                          */
/* Send message                                                             */
/*                                                                          */

extern void can_send (void);


/*                                                                          */
/* Returns CAN status byte                                                  */
/*                                                                          */
/* bit 7 : Busoff status    bit 6 : Error warning status                    */
/* bit 5 : Transmit status  bit 4 : Receive status                          */
/* bit 3 : Transmission complete status                                     */
/* bit 2 : Tranmit buffer status                                            */
/* bit 1 : Data overrun status                                              */
/* bit 0 : Receive buffer status                                            */
/*                                                                          */
/* Note: calling can_status resets a pending interrupt request              */
/* (That's a feature of the controller)                                     */
/*                                                                          */

extern unsigned char can_status (void);
```

```
/*                                                                      */
/* Returns 1 if NEWDAT of message object register 1 is set, else 0      */
/*                                                                      */
/* Does not release receive buffer (which, in fact, means, it does not do */
/* anything to the state of the controller)                             */
/*                                                                      */
/* To release the receive buffer, you have to read the data with can_read */
/*                                                                      */

extern bit can_newdat (void);


/*                                                                      */
/* Returns 1 if incoming messages were lost due to data overrun, else 0 */
/*                                                                      */
/* Clears data overrun status DOS                                       */
/*                                                                      */
/* If for any reason you don't want to reset DOS, you may also read bit 1 from */
/* can_status. Probably you won't need, so this function is more convenient */
/* and self-explaining in the source code.                              */
/*                                                                      */

extern bit can_msglst (void);
```

A.2.2 CANCTSJA.C

```
/*                                                                          */
/* CANCTSJA - CAN Controller Routines for SJA1000 at 80C537                 */
/*                                                                          */
/* Stephan Pahlke, Sven Herzfeld                                            */
/* Exam FHH Summer 2000                                                     */
/*                                                                          */
/* FOR INFORMATION ABOUT THE USED ROUTINES AND MORE, PLEASE CHECK THE       */
/* CANCTOWN.H HEADER FILE.                                                  */
/*                                                                          */
/* Version: 05/23/00                                                        */
/*                                                                          */
/* History:                                                                 */
/*                                                                          */
/* 04/26/00: adapted from CANCTRLR for 80C151C                              */
/* 04/26/00: Receive succesfully tested (125 kBaud)                         */
/* 05/02/00: Sending successfully tested                                    */
/* 05/03/00: Added remote telegram transmission, corrected receive and      */
/*            added handling of arbitration and RTR,                        */
/*           Added msglst                                                   */
/* 05/11/00: Some cosmetic operations, better comments and things like that */
/* 05/12/00: Added limitation of DLC to eight                               */
/* 05/23/00: bus speed 100, 250, 500 and 1000 kBaud available               */



#include <canresja.h>
#include <reg517.h>

// some data definitions

unsigned char pdata canreg[31];


struct CAN_DATA                         /* definition of structure type     */
{                                       /* can data to return               */
  unsigned char can_byte[7];            /* eight data bytes                 */
  unsigned char can_length;             /* and one byte DLC                 */
  unsigned short can_ID;                /* identifier                       */
  unsigned char can_RTR;                /* RTR flag for remote telegram     */
};


//*****************************************************************************
/*                                                                          */
/* CAN bus initialisation routine, send bus speed (baudrate) as parameter,  */
/* given in kBaud (e. g. 125 = 125 kBaud, 1000 = 1 Mbaud), plus acceptance  */
/* mask and filter                                                          */

void can_init (unsigned short Bus_speed, unsigned char filter, unsigned char mask)

{
  data bit Bus_speed_ok;                /* flag bit                         */

  Bus_speed_ok = 1;

  CR = 0x21;                            /* even directly after reset, the   */
                                        /* SJA1000 sometimes refused to be  */
                                        /* programmed as he was not in reset*/
                                        /* mode. So set it manually.        */

  CDR = 0x47;                           /* basic CAN mode, CBP = 1          */
                                        /* RXINTEN = 0, clock off           */

  OCR = 0x1A;                           /* normal output mode, TX1 float, TX0*/
                                        /* Pull-down, OCPOL0 = 0            */

  if (Bus_speed == 100)                 /* 100 kBaud                        */
  {
    BTR0 = 0x4E;                        /* set Bus timing register          */
    BTR1 = 0x14;
    Bus_speed_ok = 0;                   /* flag OK, we know this bus speed  */
  }
```

```
  if (Bus_speed == 250)                /* 250 kBaud                          */
  {
    BTR0 = 0xC2;                       /* set Bus timing register            */
    BTR1 = 0x49;
    Bus_speed_ok = 0;                  /* flag OK, we know this bus speed    */
  }

  if (Bus_speed == 500)                /* 500 kBaud                          */
  {
    BTR0 = 0x41;                       /* set Bus timing register            */
    BTR1 = 0x27;
    Bus_speed_ok = 0;                  /* flag OK, we know this bus speed    */
  }

  if (Bus_speed == 1000)               /* 1 MBaud                            */
  {
    BTR0 = 0x40;                       /* set Bus timing register            */
    BTR1 = 0x27;
    Bus_speed_ok = 0;                  /* flag OK, we know this bus speed    */
  }

  if (Bus_speed_ok != 0)               /* if nothing else is given what we   */
  {                                    /* know, we consider 125 kBaud as     */
    BTR0 = 0xC5;                       /* default (without further annotation)*/
    BTR1 = 0x3A;                       /* (after a lot of error frames, the  */
  }                                    /* controller will reach busoff state) */

  /* Remark: as we don't use Bus_speed_ok as a return value, it is not       */
  /* necessary to set it even if 125 kBaud was asked                         */

  ACR = filter;                        /* set acceptance filter              */
  AMR = mask;                          /* set acceptance mask                */

  CR = 0x20;                           /* disable all interrupts, stop       */
                                       /* reset mode                         */
}

//*****************************************************************************
/*                                                                           */
/* Define data of transmit frame                                             */
/*                                                                           */
/* Always include eight data bytes, don't care about the value you give bytes */
/* which will not be sent.                                                   */
/*                                                                           */
/* can_send sends only the given amount of bytes (0..8)                      */
/*                                                                           */

bit can_db (unsigned char databyte[7], unsigned char datalength)

{
  data bit returnvalue = 0;            /* return value                       */

  if (SR & 4)                          /* if Transmit Buffer Status released */
  {
    if (datalength > 8)                /* in CAN, only eight bytes are allowed*/
    {
      datalength = 8;                  /* limit DLC to 8                     */
    }
    TXB1 = TXB1 & 0xF0;                /* clear DLC bits                     */
    TXB1 = TXB1 | datalength;          /* set DLC                            */
    TXD0 = databyte[0];                /* define data                        */
    TXD1 = databyte[1];                /* define data                        */
    TXD2 = databyte[2];                /* define data                        */
    TXD3 = databyte[3];                /* define data                        */
    TXD4 = databyte[4];                /* define data                        */
    TXD5 = databyte[5];                /* define data                        */
    TXD6 = databyte[6];                /* define data                        */
    TXD7 = databyte[7];                /* define data                        */
    returnvalue = 1;
  }
  return (returnvalue);
}
//*****************************************************************************
```

```
/*                                                                          */
/* Returns data bytes of receiving message object                           */
/*                                                                          */
/* The answer always includes eight data bytes, if the received message     */
/* contains less, the rest is unspecified.                                  */
/*                                                                          */
/* Data length, arbitration and RTR are also returned                       */
/*                                                                          */
/*                                                                          */

struct CAN_DATA can_read (void)

{
  struct CAN_DATA can_struct;           /* structure to give back           */
  data unsigned short identifier;

  can_struct.can_byte[0] = RXD0;        /* set data in structure            */
  can_struct.can_byte[1] = RXD1;        /* set data in structure            */
  can_struct.can_byte[2] = RXD2;        /* set data in structure            */
  can_struct.can_byte[3] = RXD3;        /* set data in structure            */
  can_struct.can_byte[4] = RXD4;        /* set data in structure            */
  can_struct.can_byte[5] = RXD5;        /* set data in structure            */
  can_struct.can_byte[6] = RXD6;        /* set data in structure            */
  can_struct.can_byte[7] = RXD7;        /* set data in structure            */

  can_struct.can_length = RXB1 & 0x0F;  /* read DLC                         */

  identifier = ((unsigned short)RXB0) << 8; /* high byte                    */
  identifier = identifier + (RXB1 & 0xE0); /* 3 MSBs of low byte            */
  identifier = identifier >> 5;         /* divide by 2^5 = 32               */
  can_struct.can_ID = identifier;

  can_struct.can_RTR = 0;               /* delete RTR flag                  */
  if (RXB1 & 0x10)                      /* if RTR was received              */
  {
    can_struct.can_RTR = 1;             /* set RTR flag                     */
  }

  CMR = 0x04;                           /* set RRB, release receive buffer  */
  return (can_struct);                  /* return complete structure        */
}


//****************************************************************************
/*                                                                          */
/* Send remote telegram                                                     */
/*                                                                          */
/* Will be sent from transmit buffer!                                       */
/*                                                                          */

void can_remote (void)

{
  TXB1 = TXB1 | 0x10;                   /* set RTR remote transmission request */
  CMR = 1;                              /* set TR transmission request      */
}


//****************************************************************************
/*                                                                          */
/* Define arbitration of transmit message                                   */
/*                                                                          */
/* Note that the arbitration is only 11 bit, the five lowest bits are unused */
/*                                                                          */
```

```
void can_ar (unsigned short identifier)

{
  TXB0 = 0;                             /* clear ID3-10                       */
  TXB1 = TXB1 & 0x0F;                   /* clear ID0-2, clear RTR             */
  identifier = identifier << 5;         /* multiply by 2^5 = 32               */
  TXB0 = identifier >> 8;               /* set TXB0 to given bits             */
  TXB1 = TXB1 | identifier;             /* set TXB1 to given bits             */
}

//*****************************************************************************
/*                                                                            */
/* Send message in transmit register                                          */
/*                                                                            */

void can_send (void)

{
  TXB1 = TXB1 & 0xEF;                   /* clear RTR remote transmission req. */
  CMR = 1;                              /* set TR transmission request        */
}

//*****************************************************************************
/*                                                                            */
/* Returns 1 if new data received                                             */
/*                                                                            */

bit can_newdat (void)

{
  data bit returnvalue;                 /* return value                       */

  returnvalue = 0;

  if (SR & 0x01)                        /* receive buffer status              */
  {
    returnvalue = 1;
  }
  return (returnvalue);
}


//*****************************************************************************
/*                                                                            */
/* Returns CAN status byte                                                    */
/*                                                                            */

unsigned char can_status (void)

{
  return (SR);
}

//*****************************************************************************
/*                                                                            */
/* Returns 1 if message was lost, otherwise 0                                 */
/*                                                                            */
/* clears data overrun                                                        */
/*                                                                            */

bit can_msglst (void)

{
  data bit returnvalue;                 /* return value                       */

  returnvalue = 0;

  if (SR & 0x02)                        /* status register, DOS               */
  {
    CMR = 0x08;                         /* clear data overrun                 */
    returnvalue = 1;                    /* return 1                           */
  }
  return (returnvalue);
}
```

A.3 Software für den Troublemaker

```
/*                                                                      */
/* ERROR.C - Creates an Error on the CAN Bus                            */
/*                                                                      */
/* For use with the new CAN Control Board, the "Troublemaker" board     */
/* and the DIDO537 board.                                               */
/*                                                                      */
/* Version 07/05/2000                                                   */
/*                                                                      */

// Use of I/O-ports:
// Port 4:
// 0 Bus Error Output
// 1 LED "Busoff State"
// 2 LED "Error Warning Level"
// 3 LED "Error in Identifier"                              Type 0
// 4 LED "Error in Data"                                    Type 1
// 5 LED "Error in EOF"                                     Type 3
// 6 LED "Low between frames short" (only used at 125 kBaud) Type 4
// 7 LED "Low between frames long"                          Type 5
//
// Port 5:
// 0 Switch 6
// 1 Switch 5
// 6 Button


#include <reg517.h>                    /* include 80517 header file        */
#include <canresja.h>                  /* include SJA1000 CAN register     */
#include <canctsja.h>                  /* include header file canctown     */

void main (void)
{

  data unsigned char counter;
  data unsigned char counter2;
  data unsigned char counter3;
  unsigned char errortype;
  unsigned char busstate;
  unsigned char input;
  unsigned short busspeed;
  unsigned char datensatz[7];          /* data bytes for transmission      */
  bit send_request;                    /* send on remote request           */
  struct CAN_DATA telegram;            /* structure for CAN telegram       */

  busspeed = 125;

  // initialize the CAN bus controller
  can_init (busspeed, 0x00, 0x00);     /* init CAN bus for 125 kBaud       */
                                       /* acceptance mask   11111111       */
                                       /* acceptance filter 00000000       */
                                       /* reacts only on 11111111XXX       */


  // store data in transmit register
  can_ar (0x555);                      /* set arbitration                  */
  datensatz[0] = 0x00;                 /* byte 1                           */
  datensatz[1] = 0x55;                 /* byte 2                           */
  datensatz[2] = 0x00;                 /* byte 2                           */
  while (can_db (datensatz, 3) == 0)   /* store message until successful   */
  { }

  P5 = 0xFF;                           /* all high                         */

  // main loop, infinetely
  while (1)
  {
    P4 = 0xFF;                         /* switch off all LED               */

    busstate = can_status ();

    /* bit 7 : Busoff status    bit 6 : Error warning status               */
    /* bit 5 : Transmit status  bit 4 : Receive status                     */
```

```
/* bit 3 : Transmission complete status                                  */
/* bit 2 : Tranmit buffer status                                         */
/* bit 1 : Data overrun status                                           */
/* bit 0 : Receive buffer status                                         */


if (busstate & 64)                  /* error warning status              */
{
  P4 = P4 & 0xFB;                   /* LED 2 on                          */
}

if (busstate & 128)                 /* controller is busoff              */
{
  P4 = P4 & 0xFD;                   /* LED 1 on                          */
  while (!(P5 & 64)) {};            /* until key pressed                 */
  for (counter3 = 10; counter3 > 0; counter3--) /* blink several times    */
  {
    P4 = P4 & 0xFD;                 /* LED 1 on                          */
    for (counter2 = 200; counter2 > 0; counter2--)
    {
      for (counter = 250; counter > 0; counter--);
    }
    for (counter2 = 200; counter2 > 0; counter2--)
    {
      for (counter = 250; counter > 0; counter--);
    }
    for (counter2 = 200; counter2 > 0; counter2--)
    {
      for (counter = 250; counter > 0; counter--);
    }
    P4 = P4 | 0x02;                 /* LED 1 off                         */
    for (counter2 = 200; counter2 > 0; counter2--)
    {
      for (counter = 250; counter > 0; counter--);
    }
    for (counter2 = 200; counter2 > 0; counter2--)
    {
      for (counter = 250; counter > 0; counter--);
    }
    for (counter2 = 200; counter2 > 0; counter2--)
    {
      for (counter = 250; counter > 0; counter--);
    }
  }
  can_init (busspeed, 0x00, 0xFF);  /* reset controller                  */
  while (P5 & 64) {};               /* until key not pressed             */
}

input = P5 & 3;
// Switch         input
//  5     6       value
// off   off        3   databyte  LED 4
// off   on         1   EOF       LED 5
// on    off        2   ID/Short  LED 3/6
// on    on         0   Idle long LED 7

if (input == 0)
{
  errortype = 5;                    /* Low in idle long                  */
  P4 = P4 & 0x7F;                   /* LED 7 on                          */
}

if (input == 1)
{
  errortype = 3;                    /* Error in EOF                      */
  P4 = P4 & 0xDF;                   /* LED 5 on                          */
}

if (input == 2)
{
  if (busspeed == 125)              /* ! alternative meaning !           */
  {
    errortype = 4;                  /* Low in idle short                 */
    P4 = P4 & 0xBF;                 /* LED 6 on                          */
```

```
  }
  else
  {
    errortype = 0;                  /* Error in ID                      */
    P4 = P4 & 0xF7;                 /* LED 3 on                         */
  }
}

if (input == 3)
{
  errortype = 1;                    /* Error in databyte                */
  P4 = P4 & 0xEF;                   /* LED 4 on                         */
}


if (can_newdat () == 0)
{
  telegram = can_read ();           /* read message                     */
  if (telegram.can_ID == 0x7FF)     /* correct identifier               */
  {
    send_request = 1;
  }
  else
  {
    send_request = 0;
  }
}

// delayed output routines
if ((P5 & 64) | send_request)       /* key pressed                      */
{
  // low speed
  if (busspeed == 125)
  {
    if (errortype == 0)             /* Error in identifier              */
    {
      can_send ();                  /* Transmit a message               */
      for (counter = 20; counter > 0; counter--) ; /* wait a while       */
      P4 = 0;                       /* set output low, low on bus       */
      for (counter = 4; counter > 0; counter--) ; /* wait a while        */
      P4 = 0xFF;
    }

    if (errortype == 1)             /* Error in databye                 */
    {
      can_send ();                  /* Transmit a message               */
      for (counter = 130; counter > 0; counter--) ; /* wait a while       */
      P4 = 0;                       /* set output low, low on bus       */
      for (counter = 4; counter > 0; counter--) ; /* wait a while        */
      P4 = 0xFF;
    }

    // Can't be selected by hardware, but as neither bus speed can be
    // switched that way, I left this error here as you might want to use
    // it. Just change one of the "if (input == )" and re-compile.
    if (errortype == 2)             /* Error in CRC                     */
    {
      can_send ();                  /* Transmit a message               */
      for (counter = 220; counter > 0; counter--) ; /* wait a while       */
      P4 = 0;                       /* set output low, low on bus       */
      for (counter = 4; counter > 0; counter--) ; /* wait a while        */
      P4 = 0xFF;
    }

    if (errortype == 3)             /* Error in EOF                     */
    {
      can_send ();                  /* Transmit a message               */
      for (counter = 250; counter > 0; counter--) ; /* wait a while       */
      for (counter = 20; counter > 0; counter--) ; /* wait a while        */
      P4 = 0;                       /* set output low, low on bus       */
      for (counter = 4; counter > 0; counter--) ; /* wait a while        */
      P4 = 0xFF;
    }
```

```
  if (errortype == 4)             /* Low in idle (short)              */
  {
    P4 = 0;                       /* set output low, low on bus       */
    P4 = 0;                       /* set output low, low on bus       */
    P4 = 0;                       /* set output low, low on bus       */
    P4 = 0;                       /* set output low, low on bus       */
    P4 = 0xFF;                    /* approx. 62,5% of a bit time      */
  }

  if (errortype == 5)             /* Low in idle (long)               */
  {
    P4 = 0;                       /* set output low, low on bus       */
    P4 = 0;                       /* set output low, low on bus       */
    P4 = 0;                       /* set output low, low on bus       */
    P4 = 0;                       /* set output low, low on bus       */
    P4 = 0;                       /* set output low, low on bus       */
    P4 = 0xFF;                    /* approx. 75% of a bit time        */
  }
} // 125 kBaud


// 250 kBaud
if (busspeed == 250)
{
  if (errortype == 0)             /* Error in identifier              */
  {
    can_send ();                  /* Transmit a message               */
    for (counter = 10; counter > 0; counter--) ; /* wait a while       */
    P4 = 0;                       /* set output low, low on bus       */
    for (counter = 2; counter > 0; counter--) ; /* wait a while        */
    P4 = 0xFF;
  }

  if (errortype == 1)             /* Error in databyte                */
  {
    can_send ();                  /* Transmit a message               */
    for (counter = 65; counter > 0; counter--) ; /* wait a while      */
    P4 = 0;                       /* set output low, low on bus       */
    for (counter = 2; counter > 0; counter--) ; /* wait a while        */
    P4 = 0xFF;
  }

  if (errortype == 2)             /* Error in CRC                     */
  {
    can_send ();                  /* Transmit a message               */
    for (counter = 100; counter > 0; counter--) ; /* wait a while      */
    P4 = 0;                       /* set output low, low on bus       */
    for (counter = 2; counter > 0; counter--) ; /* wait a while        */
    P4 = 0xFF;
  }

  if (errortype == 3)             /* Error in EOF                     */
  {
    can_send ();                  /* Transmit a message               */
    for (counter = 130; counter > 0; counter--) ; /* wait a while      */
    P4 = 0;                       /* set output low, low on bus       */
    for (counter = 2; counter > 0; counter--) ; /* wait a while        */
    P4 = 0xFF;
  }

  if (errortype == 4)             /* Low in idle (short)              */
  {
    P4 = 0;                       /* set output low, low on bus       */
    P4 = 0xFF;
  }

  if (errortype == 5)             /* Low in idle (long)               */
  {
    P4 = 0;                       /* set output low, low on bus       */
    P4 = 0;                       /* set output low, low on bus       */
    P4 = 0xFF;
  }
} // 250 kBaud
```

```
// 500 kBaud
if (busspeed == 500)
{
  if (errortype == 0)              /* Error in identifier               */
  {
    can_send ();                   /* Transmit a message                */
    for (counter = 5; counter > 0; counter--) ; /* wait a while         */
    P4 = 0;                        /* set output low, low on bus        */
    for (counter = 1; counter > 0; counter--) ; /* wait a while         */
    P4 = 0xFF;
  }

  if (errortype == 1)              /* Error in databyte                 */
  {
    can_send ();                   /* Transmit a message                */
    for (counter = 31; counter > 0; counter--) ; /* wait a while        */
    P4 = 0;                        /* set output low, low on bus        */
    for (counter = 1; counter > 0; counter--) ; /* wait a while         */
    P4 = 0xFF;
  }

  if (errortype == 2)              /* Error in CRC                      */
  {
    can_send ();                   /* Transmit a message                */
    for (counter = 50; counter > 0; counter--) ; /* wait a while        */
    P4 = 0;                        /* set output low, low on bus        */
    for (counter = 1; counter > 0; counter--) ; /* wait a while         */
    P4 = 0xFF;
  }

  if (errortype == 3)              /* Error in EOF                      */
  {
    can_send ();                   /* Transmit a message                */
    for (counter = 65; counter > 0; counter--) ; /* wait a while        */
    P4 = 0;                        /* set output low, low on bus        */
    for (counter = 1; counter > 0; counter--) ; /* wait a while         */
    P4 = 0xFF;
  }

  /* "short" errors are impossible here, the ?C is too slow             */

  if (errortype == 5)              /* Low in idle (long)                */
  {
    P4 = 0;                        /* set output low, low on bus        */
    P4 = 0xFF;
  }
} // 500 kBaud

// 1 MBaud
if (busspeed == 1000)
{
  if (errortype == 0)              /* Error in identifier               */
  {
    can_send ();                   /* Transmit a message                */
    for (counter = 1; counter > 0; counter--) ; /* wait a while         */
    P4 = 0;                        /* set output low, low on bus        */
    P4 = 0xFF;
  }

  if (errortype == 1)              /* Error in databyte                 */
  {
    can_send ();                   /* Transmit a message                */
    for (counter = 14; counter > 0; counter--) ; /* wait a while        */
    P4 = 0;                        /* set output low, low on bus        */
    P4 = 0xFF;
  }


  if (errortype == 2)              /* Error in CRC                      */
  {
    can_send ();                   /* Transmit a message                */
    for (counter = 25; counter > 0; counter--) ; /* wait a while        */
    P4 = 0;                        /* set output low, low on bus        */
```

```
          P4 = 0;                       /* set output low, low on bus        */
          P4 = 0xFF;
        }


        if (errortype == 3)             /* Error in EOF                      */
        {
          can_send ();                  /* Transmit a message                */
          for (counter = 33; counter > 0; counter--) ; /* wait a while        */
          P4 = 0;                       /* set output low, low on bus        */
          P4 = 0xFF;
        }

        /* "short" errors are impossible here, the ?C is far too slow        */

        if (errortype == 5)             /* Low im idle (lang)                */
        {
          P4 = 0;                       /* set output low, low on bus        */
          P4 = 0xFF;
        }
      } // 1 MBaud

      while (P5 & 64) {};               /* until key not pressed             */
    }
  } // while loop
} // main
```

A.4 Software für die Motorsteuerung

```
/*                                                                      */
/* RPM.C - Revolutions per minute measured, calculated and transmitted  */
/*                                                                      */
/*                                                                      */

#include <reg515.h>                     /* include 80515 header file    */
#include <intc15c.h>                    /* include symbolic interrupt names */
#include <canctrlr.h>                   /* include header file canctrlr */

sbit P1_0 = P1^0;                       /* address of port 1.0 (LED)    */
sbit P4_1 = P4^1;                       /* address of port 4.1 (engine) */
data bit newdata;                       /* indicates new data           */


// interrupt routine called by overflow of timer 0

void time_out (void) interrupt TIMER0

{
  TCON = TCON & 0xAF;                   /* clear TR0 and TR1, stop both timer  */
  newdata = 1;
} /* void time_out (void) interrupt TIMER0 */


void main (void)
{
  unsigned char datensatz[7];           /* data bytes for transmission  */
  unsigned char control_value;          /* rpm send every n-times 25 ms */
  data char counter;                    /* counter                      */

  // initialize the CAN bus controller
  can_init (125);                       /* init CAN bus for 125 kBaud   */

  // set the arbitration bits and configure message object register 1 to receive data
  can_ar (1, 0x80);                     /* arbitration 1, rcv. control_value   */
  can_ar (2, 0x10);                     /* arbitration 2, transmit RPM value   */
  can_ar (3, 0x02);                     /* arbitration 3, engine on/off        */
  can_rcv (1);                          /* message obj. reg. 1 will receive    */
  can_rcv (3);                          /* message obj. reg. 3 will receive    */


  // enable interrupt system and timer 0 overflow interrupt
  IEN0 = IEN0 | 0x82;                   /* set EAL and ET0              */

  // set timer modes
  TMOD = 0x51;                          /* timer 0 = timer mode 1       */
                                        /* timer 1 = counter mode 1     */
  // set timer 0 to 0x3CB0
  // this value was found using an oscilloscope
  TL0 = 0xB0;                           /* timer 0 low                  */
  TH0 = 0x3C;                           /* timer 0 high                 */

  TL1 = 0;                              /* set timer 1 to 0             */
  TH1 = 0;

  counter = 0;                          /* start counting with 0        */
  control_value = 0;                    /* set to 0, i. e. don't transmit      */

  TCON = TCON | 0x50;                   /* set TR0 and TR1, start both timer   */


  // main loop, running infinetely
  while (1)                             /* loop forever                 */
  {
    struct CAN_DATA can_data1;          /* structure for received data  */
    struct CAN_DATA can_data3;          /* structure for received data  */
    bit can_new1;                       /* flag for new CAN data object 1      */
    bit can_new3;                       /* flag for new CAN data object 3      */

    can_new1 = can_newdat (1);          /* check for new data           */
    if (can_new1)                       /* if NEWDAT message object 1 is set   */
    {
```

```
      can_data1 = can_read (1);         /* read data from message object 1     */
      control_value = can_data1.can_byte[0]; /* store firste data byte         */
    } /* if (can_new1) */

    can_new3 = can_newdat (3);          /* check for new data                  */
    if (can_new3)                       /* if NEWDAT message object 3 is set   */
    {
      unsigned char engine_switch;      /* the engine switch                   */

      can_data3 = can_read (3);         /* read data from message object 3     */
      engine_switch = can_data3.can_byte[0]; /* store first data byte          */
      if (engine_switch != 0)
      {
        P4_1 = 0;                       /* switch on engine                    */
        P1_0 = 0;                       /* switch on control LED               */
      } /* if (engine_switch != 0) */
      else
      {
        P4_1 = 1;                       /* switch off engine                   */
        P1_0 = 1;                       /* switch off control LED              */
      } /* else if (engine_switch != 0) */
    } /* if (can_new3) */


    if (newdata)                        /* new counter value is ready          */
    {
      data unsigned short signals;      /* counted signals                     */
      data unsigned long spm;           /* signals per minute                  */
      data unsigned short rpm;          /* revolutions per minute              */

      newdata = 0;                      /* reset flag                          */

      signals = 0x0100 * (unsigned short)TH1 + TL1;  /* signals in 25 ms       */
      TL1 = 0;                          /* set timer 1 to 0                    */
      TH1 = 0;

      spm = (unsigned long)signals * 2400; /* signals per minute               */
      rpm = (unsigned long)(spm / 660); /* 660 signals per revolution          */

      datensatz[0] = rpm % 0x0100;      /* store LSB                           */
      datensatz[1] = rpm / 0x0100;      /* store MSB                           */
      can_db (2, datensatz, 2);         /* store data byte in message object2  */

      if (control_value != 0)
      {
        counter++;                      /* increase counter                    */
        if (counter > control_value)
        {
          can_send (2);                 /* send message                        */
          counter = 0;                  /* reset counter                       */
        } /* if (counter > control_value) */
      } /* if (control_value != 0) */

      TCON = TCON | 0x50;               /* set TR0 and TR1, restart both timers*/

    } /* if newdata */
  } /* while - infinetely */
} /* main */
```

A.5 Software für die Displayansteuerung

A.5.1 DISPLAY.H

```
/* display.h */

#define IN 0
#define DA 1

#define CLS  0x01
#define HOME 0x02
#define LF   0x0A

#define ENTRY     0x04
#define INCREMENT 0x02
#define DECREMENT 0x00

#define CONTROL     0x08
#define DISPLAY_ON  0x04
#define DISPLAY_OFF 0x00
#define CURSOR_ON   0x02
#define CURSOR_OFF  0x00
#define BLINK       0x01
#define NO_BLINK    0x00

#define SHIFT       0x10
#define SHI_LEFT    0x00
#define SHI_RIGHT   0x04
#define DIS_LEFT    0x08
#define DIS_RIGHT   0x0C

#define FUNKTION_SET 0x20

#define _8BIT        0x10
#define _4BIT        0x00
#define _2LINE       0x80
#define _1LINE       0x00
#define _10DOTS      0x04
#define _7DOTS       0x00

sbit RS  = P5^0;
sbit R_W = P5^1;
sbit E   = P5^2;
sbit DB4 = P5^4;
sbit DB5 = P5^5;
sbit DB6 = P5^6;
sbit DB7 = P5^7;

extern void disp_init (unsigned char line, unsigned char dots);
extern char put_char (char c, unsigned char lf_adr);
extern void wait (unsigned short zeit);
extern void clrscr (void);
```

A.5.2 DISPLAY.C

```
/*                                                                          */
/* DISPLAY.C - Reads RPM from the bus and displays                          */
/*                                                                          */
/*                                                                          */


#include <intrins.h>
#include <reg515.h>
#include <stdio.h>
#include <display.h>
#include <intc15c.h>                    /* include symbolic interrupt names   */
#include <canctrlr.h>                   /* include header file canctrlr       */

bdata unsigned char split;              /* split byte into bits               */
sbit  SPLIT_0 = split^0;
sbit  SPLIT_1 = split^1;
sbit  SPLIT_2 = split^2;
sbit  SPLIT_3 = split^3;
sbit  SPLIT_4 = split^4;
sbit  SPLIT_5 = split^5;
sbit  SPLIT_6 = split^6;
sbit  SPLIT_7 = split^7;

sbit P1_0 = P1^0;                       /* address of port 1.0 (LED)          */

unsigned short counter;                 /* counter for "no data" warning      */
unsigned short Max_count = 200;         /* 100 = approx 2,5 s                 */
bit timeout;                            /* set if counter is 0 - no data      */
unsigned short r_counter;               /* counter for display refreshment    */
unsigned short Max_r_count = 20;        /* 100 = approx 2,5 s                 */
bit refresh;                            /* the LCD shall be refreshed         */

// Function wait has been obtained from: "Michael Baldischweiler: Der Keil-C51-Compiler"

void wait (unsigned short zeit)         /* Time delays approximately:         */
{                                       /*  time    duration                  */
 while (zeit > 0)                   /*   15 ca. 100 ?s                  */
    zeit--;                         /*  650 ca.   4,1 ms                */
                                        /* 1000 ca.   8 ms                    */
}                                       /* 2000 ca.  15 ms                    */

// Function clrscr has been obtained from: "Michael Baldischweiler: Der Keil-C51-Compiler"

void clrscr (void)
{
  RS = IN;
  putchar ( CLS );                      /* delete LCD                         */
  wait (500);
  putchar ( HOME );                     /* set cursor to first line/column    */
  wait (500);
  RS = DA;
}

// Function disp_init has been obtained from: "Michael Baldischweiler: Der Keil-C51-Compiler"
// delay times have been changed for the 80C515C

void disp_init (unsigned char line, unsigned char dots)
{
  wait(2000);                           /* delay approx. 15 ms                */
  RS = 0;
  R_W = 0;
  DB4 = 1;
  DB5 = 1;
  DB6 = 0;
  DB7 = 0;

  E = 1;                                /* first initialization               */
  _nop_();
  E = 0;
  wait(650);                            /* delay approx 4,1 ms                */

  E = 1;                                /* second initialization              */
```

```
  _nop_();
  E = 0;
  wait(25);                             /* delay approx 100 us               */

  E = 1;                                /* third initialization              */
  _nop_();
  E = 0;
  wait(25);

  DB4 = 0;                              /* four bits data bus width          */
  E = 1;
  _nop_();
  E = 0;
  wait (25);

  RS = IN;                              /* send instructions to the display  */

  putchar( FUNKTION_SET | _4BIT | line | dots ); /* 2-line-display           */
  wait(100);
  putchar( CONTROL | DISPLAY_OFF );     /* display OFF                       */
  wait(100);
  putchar( CONTROL | DISPLAY_ON | CURSOR_OFF | NO_BLINK ); /* Display ON     */
  wait(100);
  putchar( CLS );                       /* clear display                     */
  wait(100);
  putchar( HOME );                      /* cursor home                       */
  wait(100);
  putchar( ENTRY | INCREMENT );         /* move cursor incrementally         */
  wait(100);

  RS = DA;                              /* send data to the display          */
}

// Function putchar has been obtained from: "Michael Baldischweiler: Der Keil-C51-Compiler"

char putchar (unsigned char c)
{
  if (c == 0x0A)  /* \n wird von printf in 0x0A umgesetzt */
  {
// this part SHOULD work, but for unknown reason, it did not ...
//    RS = IN; R_W = 0;
//    split = LF_ADR;                   /* split address byte                */
//    DB7 = 1;                          /* set DD-RAM-address                */

//    DB6 = SPLIT_6; DB5 = SPLIT_5; DB4 = SPLIT_4;
//    E = 1; wait(5) ; E = 0;           /* write to controller               */
//    DB7 = SPLIT_3; DB6 = SPLIT_2; DB5 = SPLIT_1; DB4 = SPLIT_0;
//    E = 1; wait(5) ; E = 0;           /* write to controller               */
//    RS = DA;
//    wait (5000);
  }
  else
  {
    R_W = 0;                            /* write to display                  */
    split = c;
    DB7 = SPLIT_7; DB6 = SPLIT_6; DB5 = SPLIT_5; DB4 = SPLIT_4;
    E = 1; wait(5) ; E = 0;             /* write to display                  */
    DB7 = SPLIT_3; DB6 = SPLIT_2; DB5 = SPLIT_1; DB4 = SPLIT_0;
    E = 1; wait(5) ; E = 0;             /* write to display                  */
  }
  return(0);
}


/* interrupt routine called when timer0 has run out                          */

void time_out (void) interrupt TIMER0

{
  TCON = TCON & 0xEF;                   /* clear TR0, stop timer             */
  counter--;                            /* decrement "no data" counter       */
  r_counter--;                          /* decrement "refresh" counter       */
  if (counter == 0)                     /* maximum wait time for new data has */
  {                                     /* run out - display "no data"       */
```

```
    timeout = 1;
  }
  else
  {                                   /* restart timer                     */
    TL0 = 0xB0;                       /* timer 0 low                       */
    TH0 = 0x3C;                       /* timer 0 high                      */
    TCON = TCON | 0x10;               /* set TR0, start timer              */
  }

  if (r_counter == 0)                 /* wait time for display refreshment  */
  {
    refresh = 1;
  }
} /* void time_out (void) interrupt TIMER0 */



void main (void)
{
  // initialize the CAN bus controller
  can_init (125);                     /* init CAN bus for 125 kBaud        */

  // set the arbitration bits and configure message object register 1 to receive data
  can_ar (1, 0x10);                   /* arbitration 1, transmit RPM value  */
  can_rcv (1);                        /* message obj. reg. 1 will receive   */

  // initialize the LCD
  disp_init (0x08, 0);

  printf ("Diplomarbeit                         CAN-Bus"); /* two lines ... */
  wait (50000);                       /* wait a bit ...                    */
  wait (50000);
  wait (50000);
  wait (50000);
  clrscr ();
  printf ("Sven Herzfeld                        Stephan Pahlke");
  wait (50000);
  wait (50000);
  wait (50000);
  wait (50000);
  clrscr ();
  printf ("Motordrehzahl                        - keine Daten -");
  RS = IN;
  putchar( HOME );                    /* set cursor to home position       */
  wait(500);
  RS = DA;

  // enable interrupt system and timer 0 overflow interrupt
  IEN0 = IEN0 | 0x82;                 /* set EAL and ET0                   */

  // set timer modes
  TMOD = 0x01;                        /* timer 0 = timer mode 1            */

  // time delay
  TL0 = 0xB0;                         /* timer 0 low                       */
  TH0 = 0x3C;                         /* timer 0 high                      */

  counter = Max_count;                /* start counting                    */
  r_counter = Max_r_count;            /* start counting                    */
  TCON = TCON | 0x10;                 /* set TR0, start timer              */

  // main loop, running infinetely
  while (1)                           /* loop forever                      */
  {
    struct CAN_DATA can_data1;        /* structure for received data       */
    bit can_new1;                     /* flag for new CAN data object 1    */
    unsigned short rpm;               /* revolutions per minute            */

    can_new1 = can_newdat (1);        /* check for new data                */
    if (can_new1)                     /* if NEWDAT message object 1 is set  */
    {
      timeout = 0;
      TL0 = 0xB0;                     /* timer 0 low                       */
      TH0 = 0x3C;                     /* timer 0 high                      */
```

```
      counter = Max_count;               /* start counting                    */
      TCON = TCON | 0x10;                /* set TR0, start timer              */

      P1_0 = 0;
      can_data1 = can_read (1);          /* read data from message object 1   */

      if (refresh)
      {
        rpm = can_data1.can_byte[0];     /* store first data byte             */
        rpm = rpm + 256 * can_data1.can_byte[1]; /* store second data byte    */
        printf ("Motordrehzahl                              %4u rpm      ", rpm);
        refresh = 0;
        r_counter = Max_r_count;              /* start counting                  */
      }
      else
      {
        printf ("Motordrehzahl                              %4u rpm *    ", rpm);
      }
      P1_0 = 1;
      RS = IN;
      putchar( HOME );                   /* Cursor first line, first column   */
      wait(500);
      RS = DA;
    } /* if (can_new1) */

    if (timeout)
    {
      printf ("Motordrehzahl                           - keine Daten -");
      timeout = 0;
      RS = IN;
      putchar( HOME );                   /* Cursor first line, first column   */
      wait(500);
      RS = DA;
    }
  } /* while - infinetely */
} /* main */
```

B Hardware

B.1 CAN Control Board für DIDO 537

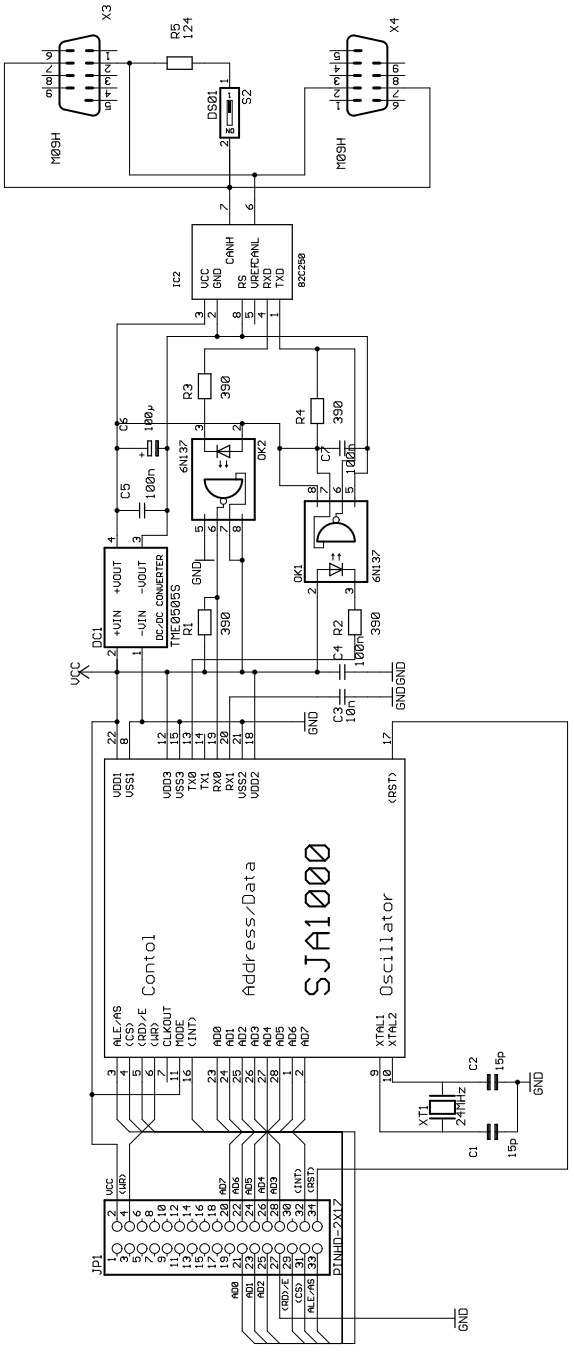


Abbildung 40: Schaltplan des CAN Control Boards

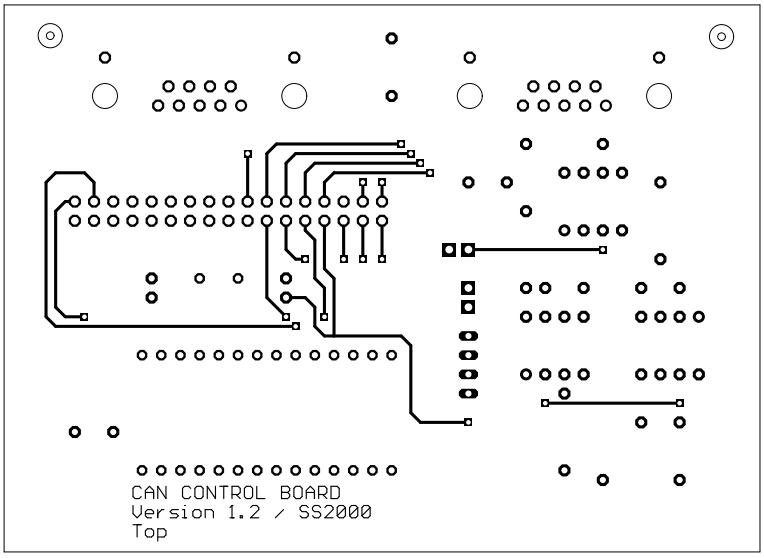


Abbildung 41: Oberseite des CAN Control Boards

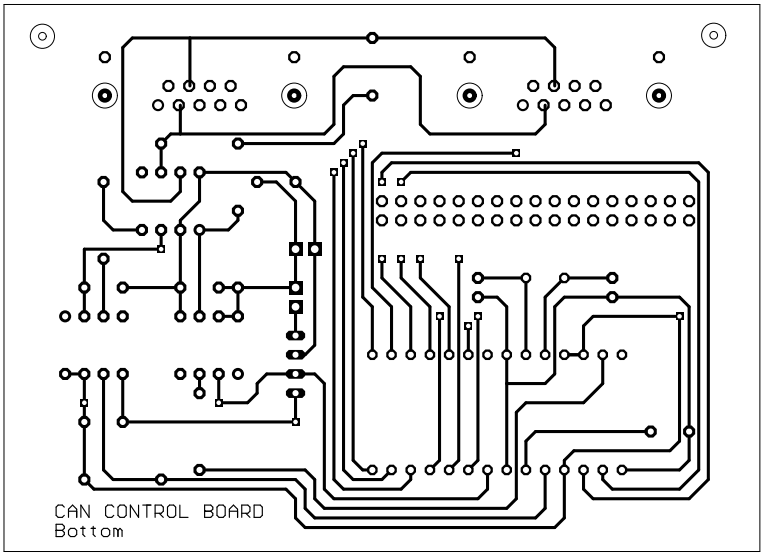


Abbildung 42: Unterseite des CAN Control Boards

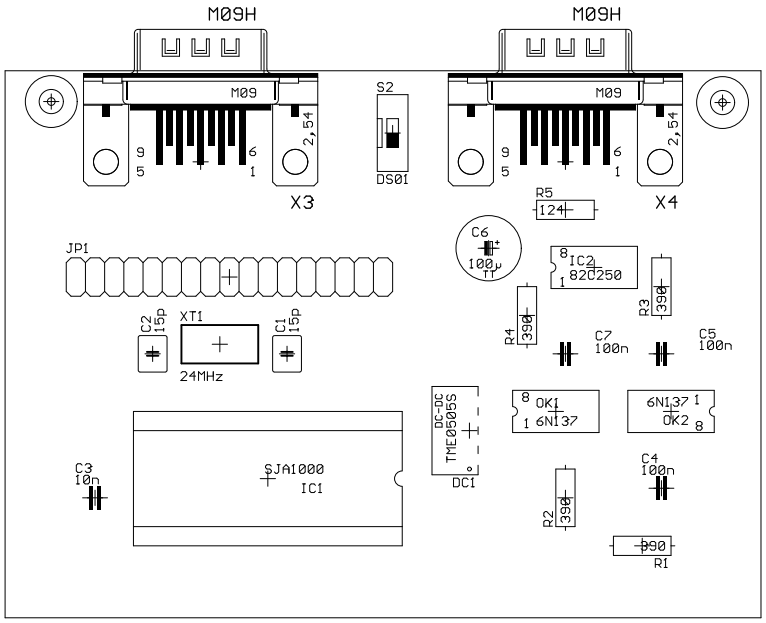


Abbildung 43: Bestückungsplan des CAN Control Boards

Teil	Wert
C1	Kondensator 15 pF
C2	Kondensator 15 pF
C3	Kondensator 10 nF
C4	Kondensator 100 nF
C5	Kondensator 100 nF
C6	Kondensator 100 μF
C7	Kondensator 100 nF
DC1	DC/DC Traco TME0505S
IC1	CAN-Controller Philips SJA1000
IC2	CAN-Treiber Philips 82C250
JP1	Steckerleiste 2x 17-polig
OK1	Optokoppler 6N137
OK2	Optokoppler 6N137
R1	Widerstand 390 Ω
R2	Widerstand 390 Ω
R3	Widerstand 390 Ω
R4	Widerstand 390 Ω
R5	Widerstand 124 Ω
S2	DIL-Schalter 1x
X3	9-poliger SUB-D-Stecker
X4	9-poliger SUB-D-Stecker
XT1	Quarz 24 MHz
Platine	geätzte Platine, zweiseitig

Tabelle 13: Stückliste CAN Control Board

B.2 Troublemaker

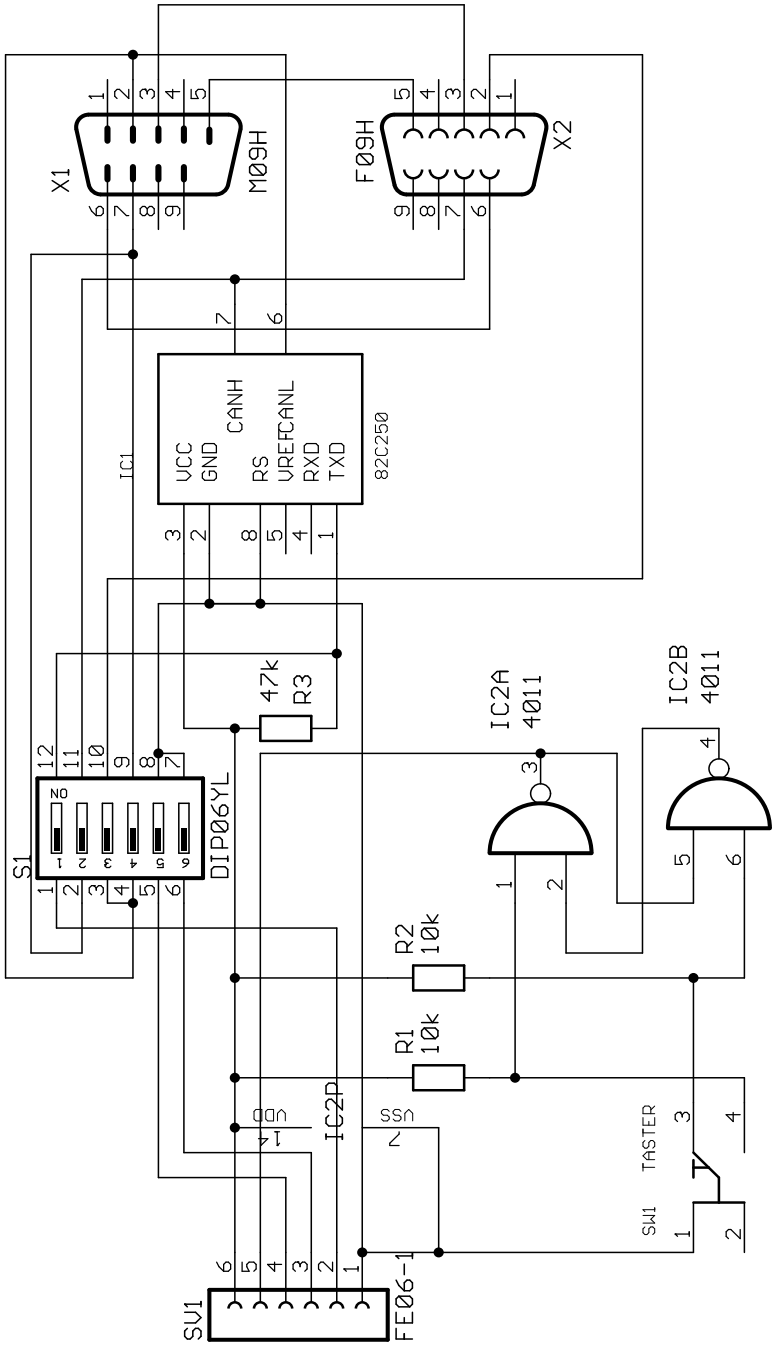


Abbildung 44: Schaltplan des Troublemakers

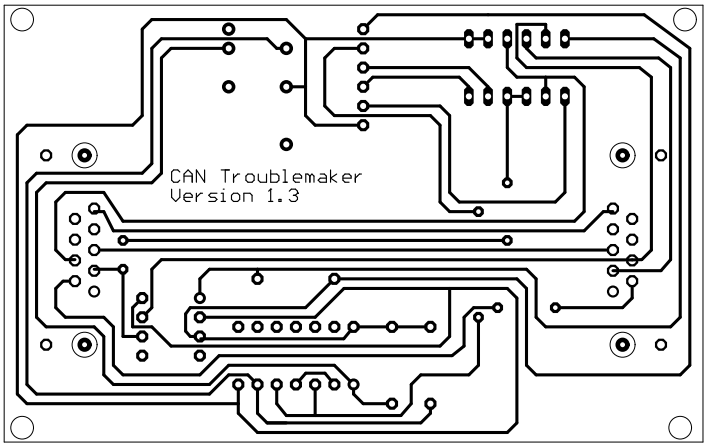


Abbildung 45: Unterseite des Troublemakers

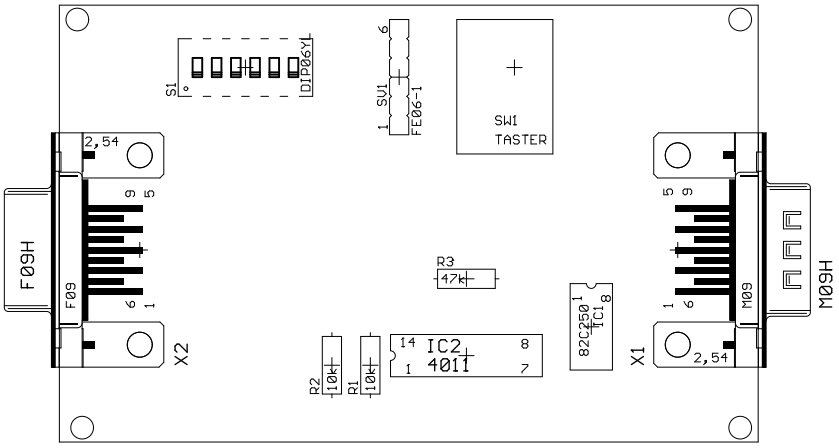


Abbildung 46: Bestückungsplan des Troublemakers

Teil	Wert
IC1	82C250
IC3	4011
R1	Widerstand 10 kΩ
R2	Widerstand 10 kΩ
R3	Widerstand 47 kΩ
S1	DIP-Schalter 6x
SV1	FE06-1
SW1	Taster
X1	9-poliger Sub-D-Stecker
X2	9-polige Sub-D-Buchse
Platine	geätzte Platine

Tabelle 14: Stückliste Troublemaker

B.3 Errorfind

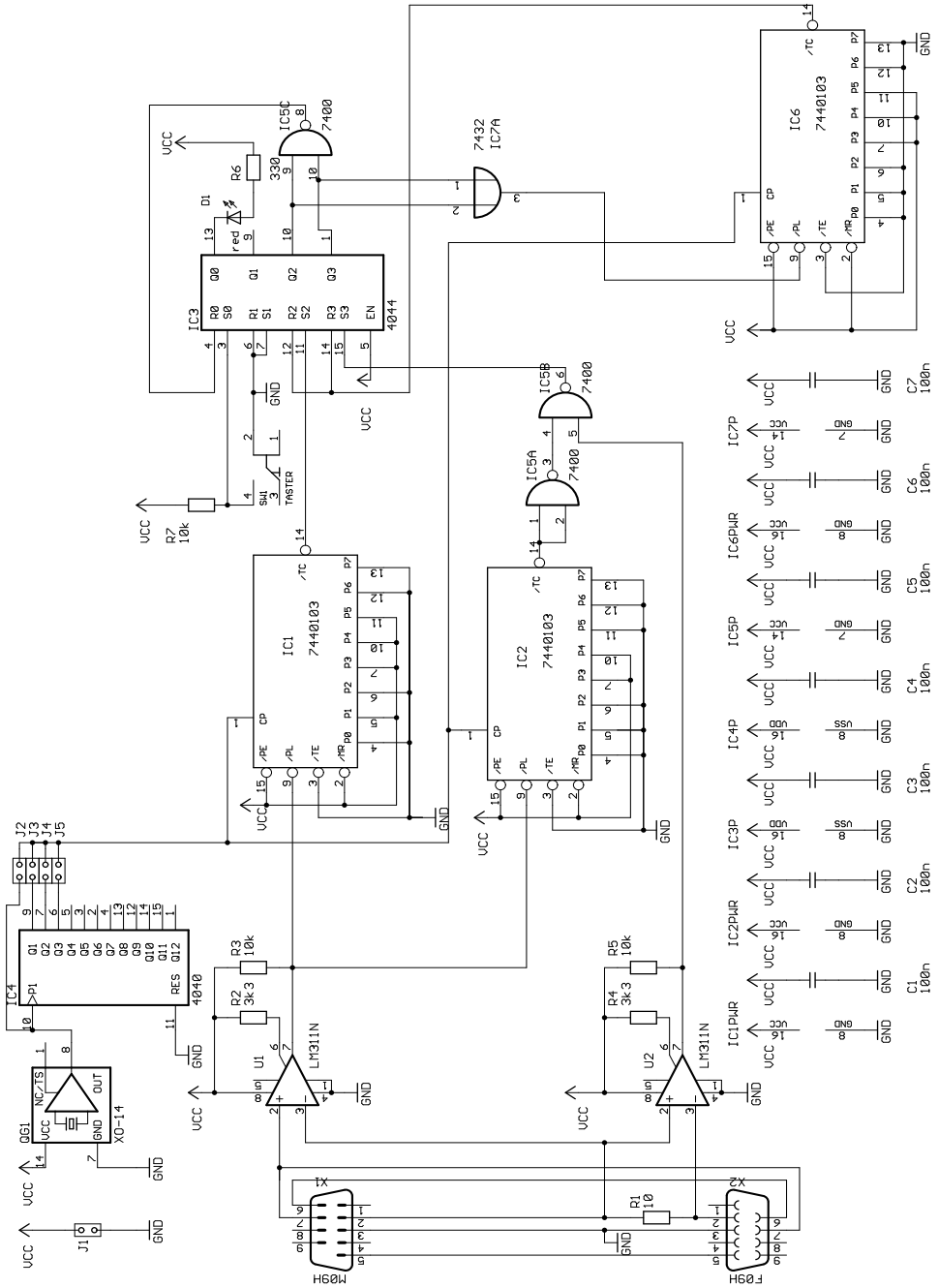


Abbildung 47: Schaltplan des Errorfinders

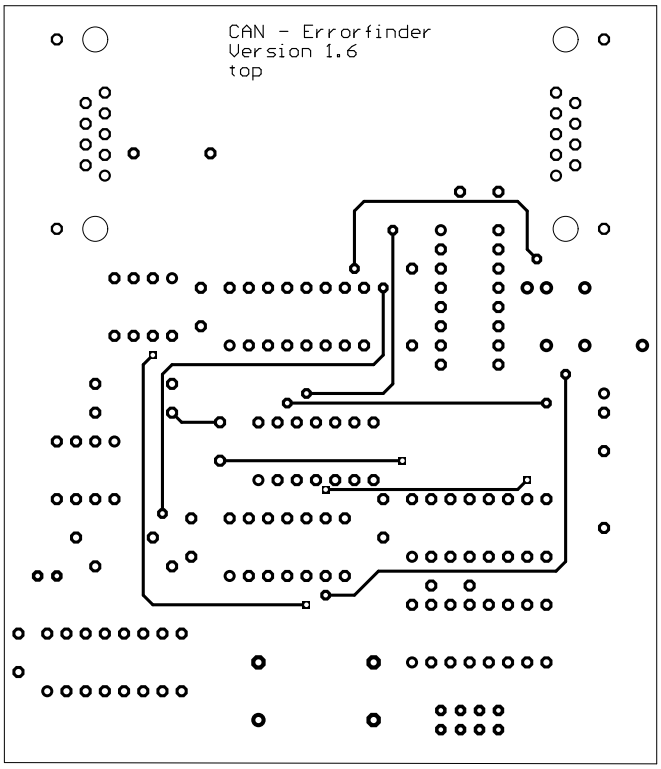


Abbildung 48: Oberseite des Errorfinders

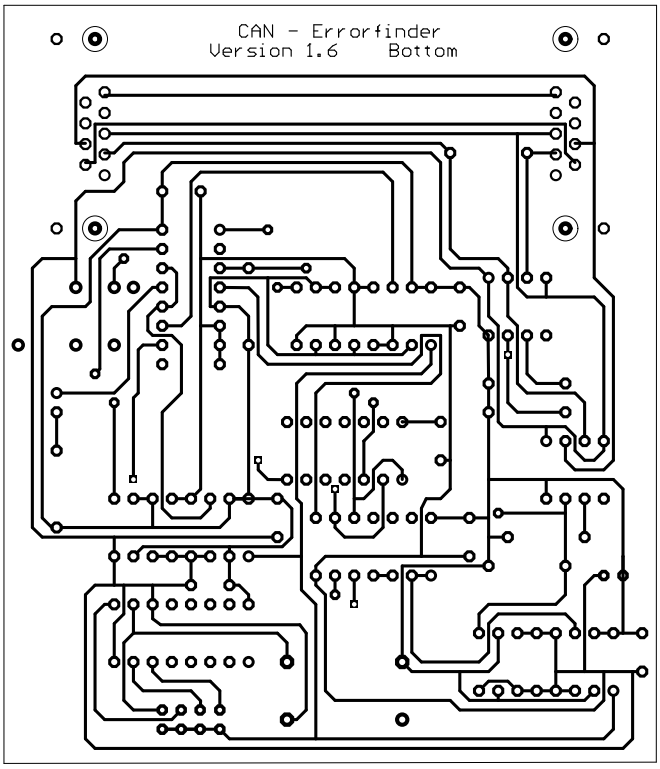


Abbildung 49: Unterseite des Errorfinders

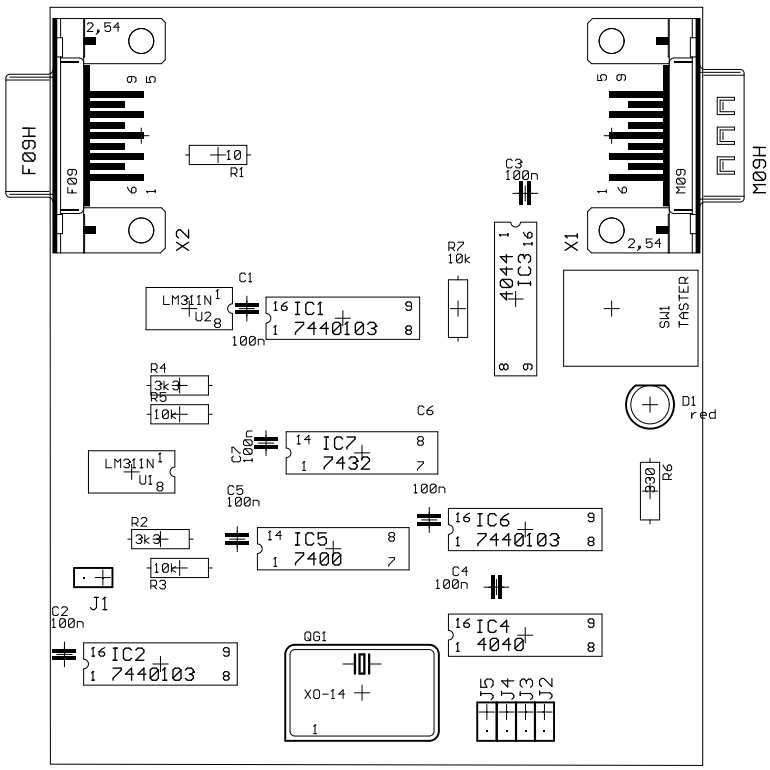


Abbildung 50: Bestückungsplan des Errorfinders

Teil	Wert
C1	Kondensator 100 nF
C2	Kondensator 100 nF
C3	Kondensator 100 nF
C4	Kondensator 100 nF
C5	Kondensator 100 nF
C6	Kondensator 100 nF
C7	Kondensator 100 nF
D1	LED rot
IC1	74HCT40103
IC2	74HCT40103
IC3	4044
IC4	4040
IC5	7400
IC6	74HCT40103
IC7	7432
J1	Jumper
J2	Jumper
J3	Jumper
J4	Jumper
J5	Jumper
QG1	Taktgenerator XO-14 10 MHz
R1	Widerstand 10 Ω
R2	Widerstand 3,3 kΩ
R3	Widerstand 10 kΩ
R4	Widerstand 3,3 kΩ
R5	Widerstand 10 kΩ
R6	Widerstand 330 Ω
R7	Widerstand 10 kΩ
SW1	Taster
U1	Komparator LM311N
U2	Komparator LM311N
X1	9-poliger Sub-D-Stecker
X2	9-polige Sub-D-Buchse
Platine	geätzte Platine, zweiseitig

Tabelle 15: Stückliste Errorfind

B.4 Relaisplatine

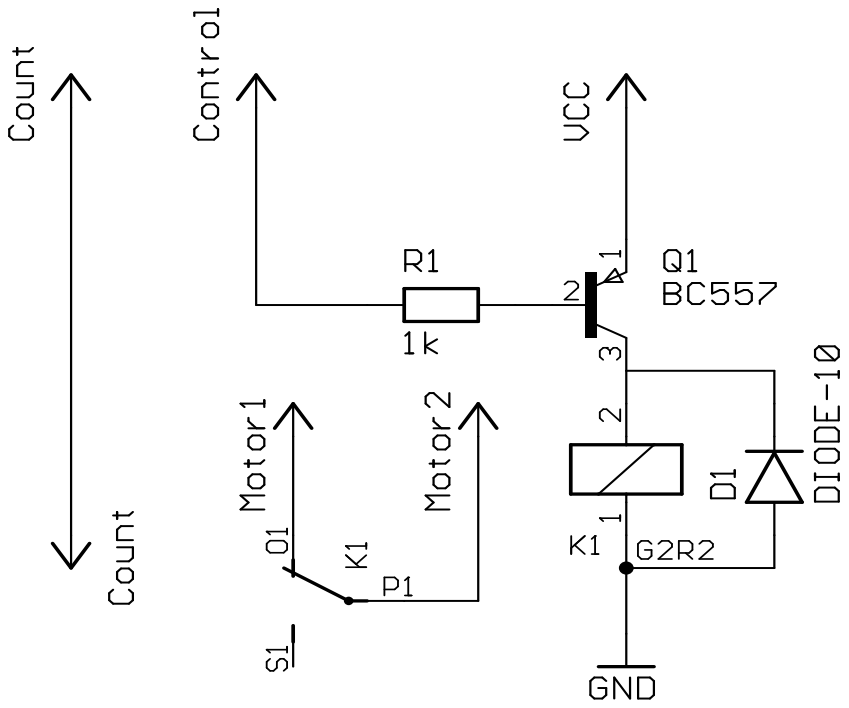


Abbildung 51: Schaltplan der Relaisplatine

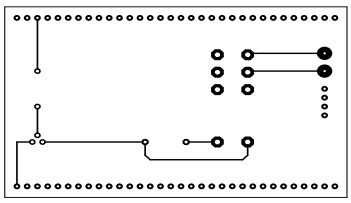


Abbildung 52: Unterseite der Relaisplatine

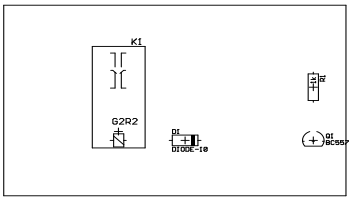


Abbildung 53: Bestückung der Relaisplatine

Teil	Wert
K1	Relais G2R2
Q1	Transistor BC557
R1	Widerstand 1 kΩ
SV1	FE05-2W
D1	Diode 1N4007

Tabelle 16: Stückliste Relaisplatine

C Das ISO/OSI-Schichtenmodell

Die International Standardisation Organisation hat mit dem Open System Interconnect ein Modell vorgestellt, nach dem eine Netzwerkkommunikation hierarchisch beschrieben werden kann.

7	Application Layer	Anwendungsschicht	Interface zur Anwendung
6	Presentation Layer	Darstellungsschicht	Umsetzung der Datenformate, Komprimierung, Verschlüsselung
5	Session Layer	Sitzungsschicht	Verbindungsaufbau und -überwachung
4	Transport Layer	Transportschicht	Aufteilung in Pakete und Rekonstruktion der Daten, Erzeugung der physikalischen Adresse
3	Network Layer	Netzwerkschicht	Steuerung des Transportes, wenn mehrere Übertragungswege vorhanden sind
2	Data LinkLayer	Verbindungsschicht	Fehlererkennung und -korrektur, Eingangsfilter
1	Physical Layer	Bitübertragungsschicht	Elektrische Verbindungen, Bitcodierung

D Literatur

[etsch94] Etschberger, Konrad et. al.:
Controller area network: CAN; Grundlagen, Protokolle, Bausteine, Anwendungen
München, Wien: Hanser 1994
ISBN 3-446-17596-2

[law97] Lawrenz, Wolfhard et. al.:
Controller Area Network: CAN; Grundlagen und Praxis
2., vollständig überarbeitete und erweiterte Auflage
Heidelberg: Hüthig 1997
ISBN 3-7785-2575-1

[bal92] Baldischweiler, Michael:
Der Keil C51-Kompiler: für Entwicklung und Ausbildung
MC-Tools 7
Traunstein: Feger-und-Co.-Hardware-und-Software-Verl. OGH 1992
ISBN 3-928434-10-1

Außerdem wurden die im folgenden Abschnitt erwähnten Handbücher, Datenblätter und Applikationen genutzt.

E Datenblätter und Applikationen

Abschließend sind jene Datenblätter und Applikationen wiedergegeben, die in Zusammenhang mit unserer Arbeit stehen. Dabei muss es sich bei den üblichen Logikbausteinen nicht immer um die Ausgabe des Herstellers handeln, von dem das eingesetzte Exemplar tatsächlich stammt.

CANalyzer

- Handbuch CANalyzer V2.2. Von der genutzten Version 3.0 liegt das Handbuch nur in gedruckter Form vor

Phytec RapidDevelopment Kit

- Handbuch miniModul-515C
- Handbuch Basisplatine

Mikrocontroller mit on-chip CAN–Controller Infineon (Siemens) 80C515C

- Kurzbeschreibung Infineon 80C515C
- Datenblatt Infineon 80C515C
- Handbuch Infineon 80C515C
- Applikation: CAN Controller im 80C515C
- Applikation: Programm zur Bit Timing Berechnung

Microcontrollerplatine DIDO537

- Dokumentation zur DIDO537

Stand-alone CAN-Controller Philips SJA1000

- Datenblatt SJA1000
- Anomalie beim SJA1000
- Applikation SJA1000
- Applikation: Ermitteln der Bit Timing Parameter des SJA1000

CAN-Controller Philips PCA82C250

- Datenblatt PCA82C250
- Applikation PCA82C250

Weitere Hardware des CAN Control Boards

- Datenblatt Optokoppler 6N137
- Datenblatt Optokoppler HCPL7720
- Datenblatt Traco DC/DC (TMA Familie)

Hardware des Errorfinders

- Datenblatt Komparator LM311
- Datenblatt 74HCT4040 12-bit Zähler/Frequenzteiler
- Datenblatt 74HCT40103 8-bit Rückwärtszähler
- Datenblatt 4001 4-fach NOR
- Datenblatt 4011 4-fach NAND
- Datenblatt 4044 4-fach R/S-Flipflop
- Datenbuch C-Max Oszillatoren

Allgemeine Datenblätter

- Datenblatt HEF Familie (Philips)
- Datenblatt HEF Gehäuse (Philips)