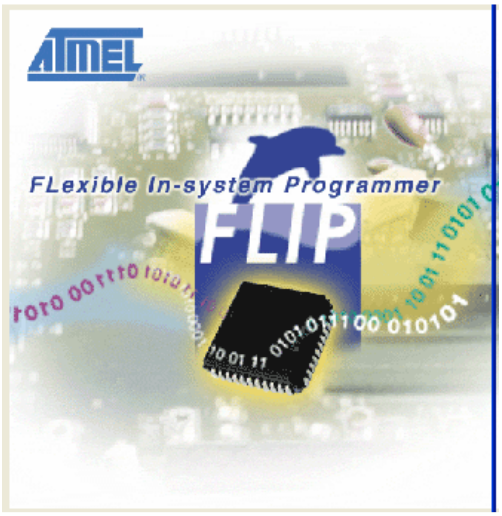
Mikrocontroller-Programmierung mit dem neuen Board „EST-ATM1“ der Elektronikschule Tettnang unter

und



Band 3: CAN-Projekte

(mit der neuen Zusatzplatine und dem neuen Controller AT89C51CC3 im ATM1-Board)

Version 1.0

von Gunthard Kraus, Oberstudienrat a.D.,
Elektronikschule Tettnang

08. Oktober 2009

Änderungshinweise:

==========

Unser ATM1-Board wird nur noch mit dem ATMEL - Controller AT89C51CC3 geliefert und bestückt. Dieser Controller ist pinkompatibel mit dem vorher verwendeten AT89C51AC3, enthält aber eine komplette CAN-Maschine auf dem Chip. Die bisherigen Programme (Siehe Band 2 des Tutorials) können jedoch ohne jede Änderung weiterverwendet werden, ohne dass Probleme zu befürchten sind.

Für die CAN-Anwendungen und Übungen wurde eine neue Zusatzplatine entwickelt. Sie ist in diesem Tutorial ebenfalls genauer beschrieben.

Die Sache mit dem CAN-Bus ist bei diesem Controller recht aufwendig und kompliziert, deshalb sollte man sich auch als erfahrener CAN-Anwender nicht scheuen, das Tutorial von Anfang an zu lesen. Vielleicht enthalten die Anfangskapitel mit den Grundlagen und Beschreibungen doch neue und interessante Details.

Aber jetzt geht es wirklich los….

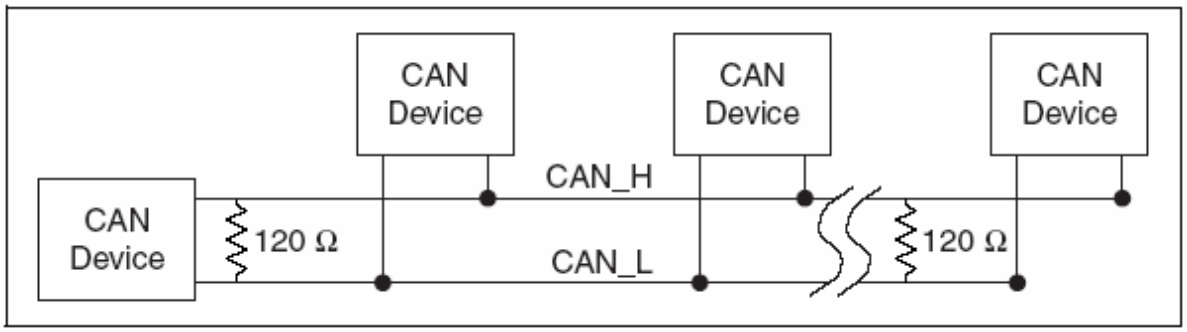
Inhaltsverzeichnis Seite

1. Warum CAN?

In der modernen Technik besteht ein sehr hoher Bedarf an Kommunikation. Messwerte müssen übermittelt werden, Steuerkommandos werden erteilt, Anfragen werden gestartet und beantwortet, Daten werden gesammelt und archiviert….all das läuft heute immer über ein „Bus-System“, an das alle beteiligten Baugruppen und Bausteine (Fachausdruck: „Teilnehmer“) gleichzeitig angeschlossen sind.
Es existieren deshalb viele konkurrierende Systeme (I2C-Bus, Feld-Bus, CAN-Bus usw.), aber in der Automobilindustrie und in den Fahrzeugen hat sich der CAN-Bus völlig durchgesetzt und dominiert die Szene als Standard **(CAN steht übrigens für „Controller Area Network**“ und wurde 1983 von der Firma Bosch entwickelt). Er weist geradezu astronomische Zuwachsraten in allen möglichen Bereichen der Elektronik und Automatisierung auf und deshalb kommt Keiner mehr an ihm vorbei.

2. CAN - Hardware (= Physical Layer des ISO-Schichtenmodells)

2.1. Bus-Topologie und Bus-Ankopplung

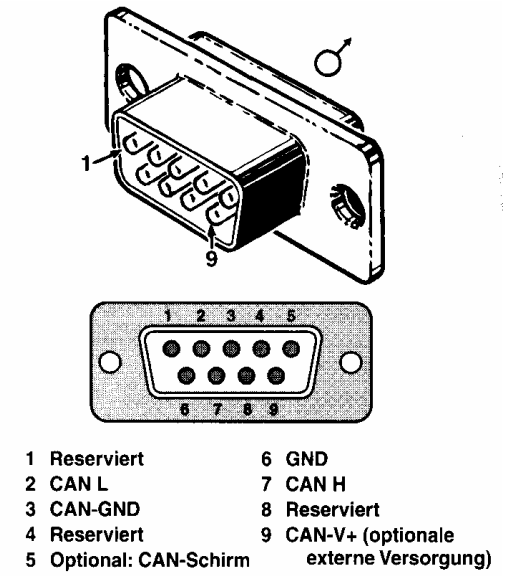


Es handelt sich um einen 2-Leiter-Bus (mit zusätzlicher Ground-Leitung), an den alle Teilnehmer völlig gleichberechtigt und parallel angeschlossen sind. Dieses symmetrische System (das nur auf den **Spannungs*unterschied*** zwischen beiden Leitungen reagiert!) zeichnet sich besonders durch hohe Störsicherheit aus und ermöglicht die Kommunikation entweder bis zu einer Entfernung von 1km mit max. 50 Kilobit / sec oder bis zu 40m mit max. 1Megabit / sec.

Bitrate	Kabellänge
10 kbits/s	6,7 km
20 kbits/s	3,3 km
50 kbits/s	1,3 km
125 kbits/s	530 m
250 kbits/s	270 m
500 kbits/s	130 m
1 Mbits/s	40 m

Genauere Informationen liefern dazu die Spezifikationen für „CAN – Low-Speed“- bzw. „CAN-High-Speed“-Betrieb in der ISO 11898 – Norm. Hier die Übersicht über die Zuordnung von Bitrate und maximaler Kabellänge,

Die beiden Leiter (**CAN_H = CAN High** und **CAN_L = CAN Low**) sind miteinander verdrillt (= „**Twisted Pair**“) und bilden auf diese Weise eine HF-Leitung mit einem Wellenwiderstand von ca. 120 Ohm (..genau genommen sollten es 124 Ohm sein…). Eine zusätzliche geerdete Abschirmung dieser verdrillten Leitung ist zulässig und wird in der Praxis oft zur Erhöhung der Störsicherheit eingesetzt.
Ein Signal, das von einem Teilnehmer (im obigen Bild heißt er „CAN-Device“) losgeschickt wird, breitet sich nach links **und** nach rechts auf der Leitung aus. Deshalb müssen beide Leitungsenden korrekt mit dem Wellenwiderstand von 120 Ohm abgeschlossen werden, um Probleme durch Reflexionen zu vermeiden (= „Bus – Termination“).



1 Reserviert	6 GND
2 CAN L	7 CAN H
3 CAN-GND	8 Reserviert
4 Reserviert	9 CAN-V+ (optionale externe Versorgung)
5 Optional: CAN-Schirm	

Als CAN-Anschlussstecker wurde der bekannte **neunpolige Sub-D-Stecker** genormt. Bei ihm gilt die nebenstehende Stiftbelegung.

(Hinweis: wie vorhin erwähnt, steht Pin 5 als „0ptionaler CAN-Schirm“ zur Verfügung.)

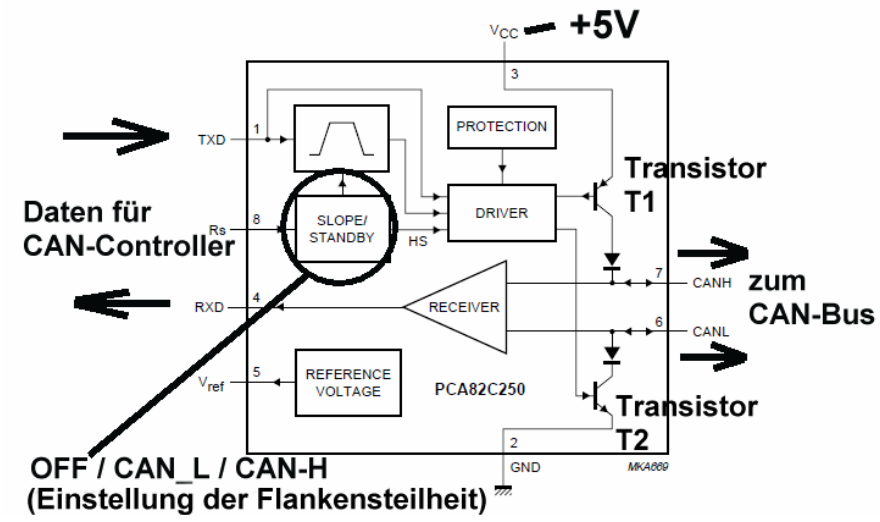
2.2. Bus-Pegel und Bustreiber

Wie besprochen, sind bei diesem symmetrischen Prinzip die **Spannungsunterschiede** zwischen der CAN_H und der CAN_L – Leitung entscheidend. Bevor wir jedoch einen Blick auf die Hardware des „Bustreibers“ werfen, prägen wir uns ein:

a) **Gleiche Pegel** gegen Masse (…üblich sind +2,5V…) an der CAN_H und der CAN_L – Leitung ergeben **keine Spannungsdifferenz** zwischen ihnen. Dies ist der **rezessive Zustand** und er entspricht der „**logischen Eins**“

b) Steigt der Pegel auf der CAN_H – Leitung auf min. +3,5V an und sinkt gleichzeitig der Pegel auf der CAN_L – Leitung auf max. +1,5V ab, dann erhält man eine **Spannungsdifferenz von mindestens 2V**. Das stellt den **dominanten Zustand** dar und ergibt die **„logische Null**“.

Sehen wir uns nun beim üblichen Bustreiber-Standardbaustein PCA82C250 mal genauer an, wie das abläuft:



Links wird der nötige CAN-Controller angeschlossen -- er liefert die zu übertragenden Daten als **TTL-Signal** (0V = logische Null, +5V = logische Eins).

Von Pin 8 gegen Masse wird ein Widerstand Rs geschaltet, über dessen Widerstandswert sich die Flankensteilheit des Ausgangssignals verändern lässt. Dabei gilt:

Wird die Spannung an Pin 8 kleiner als 30% der Versorgungsspannung (= +1,5V), dann arbeiten wir mit steilen Flanken und damit im Highspeed-Mode.

Wird die Spannung durch Vergrößern von Rs immer weiter bis 75% der Versorgungsspannung (= +3,75V) gesteigert, dann werden die Signalflanken am Ausgang immer „müder“ (= slope control im Lowspeed-Mode) und schließlich geht der Baustein in den Standby-Mode über (= kein CAN-Betrieb möglich)

Bei den Ausgängen CAN_L und CAN_H sieht es so aus:

Logischer Pegel	Zustand	CAN_H	CAN_L	Pegeldifferenz
0	dominant	Transistor T1 leitet, deshalb Pegel = +5V	Transistor T2 leitet, deshalb Pegel = 0V	+5V
1	rezessiv	Transistor T1 leitet, deshalb Pegel = +5V	Transistor T2 sperrt, deshalb Pegel eben-falls = +5V	Null
Tristate	floating	Transistor T1 sperrt	Transistor T2 sperrt	

Beim Studium der Tabelle wird auch eine wichtige Eigenschaft des CAN-Busses klar:

Sobald ein Teilnehmer eine logische Null (= dominanter Zustand) auf den Bus legt, wird automatisch ein gerade herrschender rezessiver Zustand „überschrieben“ (…der dann eingeschaltete Transistor T2 zieht nämlich die komplette CAN_L –Leitung des Systems auf Null Volt und daran können andere Teilnehmer nichts ändern. Die andere Leitung „CAN_H“ liegt sowieso schon auf +5V…..).

3. CAN – Protokoll (= Data Link Layer des ISO – Schichtenmodells)

3.1. CAN Data Frame

CAN stellt einen „Objekt-orientieren Nachrichtenaustausch“ dar. Das bedeutet, dass **jede auf dem Bus verschickte Nachricht mit einer eindeutigen Kennzeichnung des Inhalts versehen ist** (…während die Adressen der einzelnen Teilnehmer unwichtig sind). Diese Nachrichtenmarkierung heißt

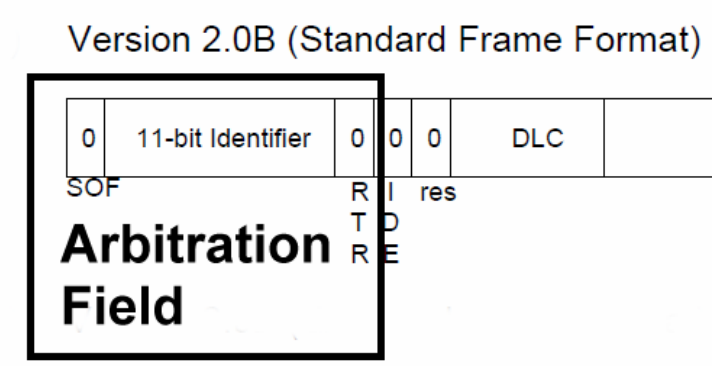
Identifier

und jeder Busteilnehmer entscheidet dann individuell, ob diese Nachricht für ihn interessant ist.
Dieser **Identifier** hat im Normalfall **eine Länge von 11 Bit** und steht immer am **Anfang eines „CAN Data Frames**“. Das ergibt theoretisch 2048 verschiedene „Nachrichten-Nummern“, von denen aber nur 2032 verwendet werden dürfen (..der Rest ist für Sonderfunktionen reserviert).

Es hat sich aber gezeigt, dass das in der Praxis zu knapp ist und deshalb kann dieser 11Bit - Identifier(= „**CAN 2.0A**“) durch einen **29 Bit – Identifier ersetzt werden.**

Das läuft als „CAN 2.0B“, wobei hier ebenfalls die Möglichkeit besteht, nur mit 11 Bit zu arbeiten. Aber das muss man dann wieder extra über ein bestimmtes Bit im Frame mitteilen……aber das kriegen wir gleich.

Sehen wir uns jetzt mal den Aufbau eines Datenrahmens **(Data Frame) nach CAN 2.0B mit 11-Bit – Identifier** an. Er beginnt IMMER mit dem **Arbitration Field**, das aus **13 Bit** besteht.

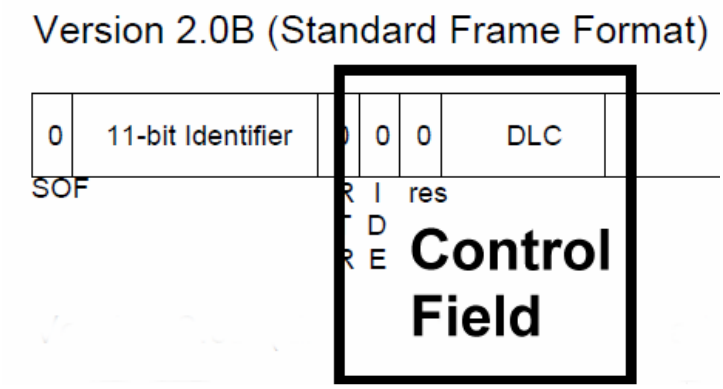


Erläuterung:

a) **SOF bedeutet „Start of Frame-Bit**“ und ist immer **dominant = logisch Null.**

b) Nun folgt der **11 Bit-Identifier**, mit dem die verschickte Nachricht gekennzeichnet wird.

c) Abgeschlossen wird die Bus-Arbitrierung mit **RTR = „Remote Transmission Request Bit“**. Damit kann eine bestimmte Nachricht „angefordert“ werden. Bei einem normalen Data Frame ist dieses Bit = logisch Null, also dominant.



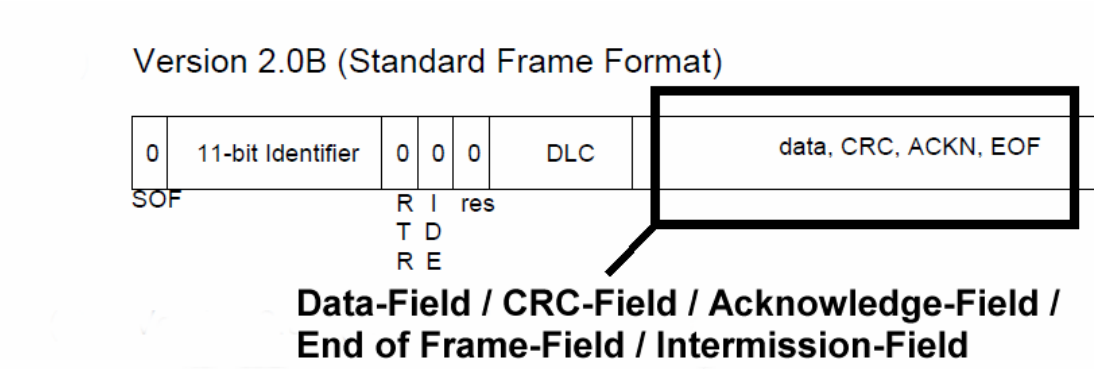
Jetzt folgt das **Control Field (= 6 Bit) .**

Es beginnt mit dem **IDE – Bit** (= Identifier Extension Bit). Eine **logische Null (= dominanter Zustand**) bedeutet, dass wir mit einem **11 Bit – Identifier** arbeiten.

Eine **logische Eins (= rezessiv**) steht folglich für die Verwendung eines **29 Bit – Identifiers.**

Nach einem Reservebit (= Zukunftsmusik) folgt schließlich mit **4 Bit** der **DLC = Data Length Code**. Die CAN-Spezifikationen lassen 0….8 Datenbytes in einem Rahmen (= Frame) zu -- die gesendete Anzahl muss also hier eingetragen werden.

Nun folgt der Rest des Data-Frames:



Im **Data-Field** stecken **zwischen Null und 8 Nutz-Datenbytes** -- um die geht es eigentlich!

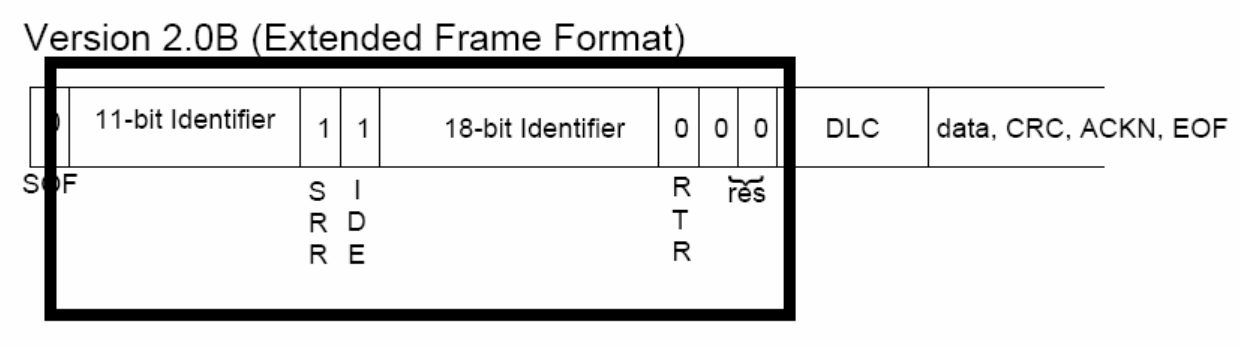
Das **CRC-Field (= 16 Bit)** dient zur Erhöhung der Übertragungssicherheit. Darin wird eine **15 Bit – Prüfsumme** übermittelt (**CRC = Cyclic Redundance Code**). Es wird von einem **CRC-Delimiter-Bit (rezessiv)** abgeschlossen und so vom nachfolgenden Ackowledge-Teil getrennt.

Das **Acknowledge-Field (= 2 Bit)** beginnt mit einem **rezessiv gesendeten Bit namens „Acknowledge Slot**". Die Empfänger reagieren darauf -- wenn sie die Nachricht korrekt empfangen haben! -- blitzschnell mit einer **Umschaltung dieses Bits auf „dominant = logisch Null**". Dadurch weiß der Sender, dass mindestens ein Teilnehmer ihn verstanden hat….
Zur Trennung vom letzten Teil des Frames folgt dann noch ein **ACK-Delimiter-Bit (rezessiv**).

Den Abschluss bildet das **End of Frame – Field (= 7 rezessiv gesendete Bits)**.

Um den anderen Bus-Teilnehmern Zeit zum Abspeichern oder zur Verarbeitung zu lassen, wird nun eine **Pause von mindestens 3 (rezessiven) Bits eingeschaltet**, bevor ein neuer Rahmen gesendet werden darf (…er beginnt dann wieder mit dem dominanten SOF = Start of Frame – Bit).
Diese Pause heißt **Intermission –Field (3 Bit).**

3.2. Kurze Information: Das Extended Frame Format mit 29 Bit -Identifier



Da muss nun natürlich Einiges anders sein:

a) Das **IDE – Bit (= Identifier Extension Bit) ist nun auf logisch 1 (rezessiv),** um die Verwendung des Extended Frame – Formats zu signalisieren.

b) Das SDR**-Bit (= Subsitute Remote Request)** wird neu eingeführt und sitzt an der Stelle, wo vorher das RTR – Bit angeordnet war.

c) Der **11 Bit – Identifier bleibt und wird um einen zweiten Block mit 18 Bit ergänzt**. Erst hinter diesem zweiten Block folgt nun das **RTR – Bit (= Remote Transmission Request).**

3.3. CAN Remote Request Frame

Damit kann eine **bestimmte Information** von einem Teilnehmer **angefordert** werden.

Dazu wird ein Frame auf dem Bus verschickt, der folgende Unterschiede aufweist:

a) Im **Identifier - Feld** muss nun der **Indentfier des Nachrichtenobjektes** stehen, das von einer anderen Stelle angefordert wird.

b) Im **Data Length Code – Feld (= DLC)** muss nun die **Anzahl der Bytes** eingetragen werden, die diese Nachricht aufweisen wird.

c) Das bisher auf „**dominant**“ gesetzte **RTR-Bit** (Remote Transmission Request) muss jetzt auf **rezessiv“ (= logische Eins) umgeschaltet** werden.

d) Das **Datenfeld muss leer bleiben**, es dürfen also keine Datenbytes ausgesendet werden.

3.4. Fehlererkennung und Fehlerbehandlung

Dieser Punkt hat bei CAN eine zentrale Bedeutung (….ist auch logisch, wenn man sich so die meisten Anwendungen des CAN-Busses in der Praxis überlegt). Deshalb folgen hier einige Informationen dazu.

CAN setzt dabei 5 „Werkzeuge“ ein:

1) **Die Bit- Fehlererkennung**
Wenn eine Station sendet, liest sie gleichzeitig die auf den Bus geschickten Bits mit. Sobald nur ein einziges Bit auf dem Bus nicht mit der „Vorgabe“ (durch den im CAN-Controller zusammengestellten Datenrahmen) übereinstimmt, wird die Sendung abgebrochen.

2) **Die Stuffbit – Fehlererkennung**
Die CAN-Norm legt fest, dass innerhalb eines Frames höchstens fünf Bits desselben Zustandes aufeinander folgen dürfen (Ausnahme am Rahmenende beim „EOF“). Muss das jedoch trotzdem sein, dann wird ein komplementäres Bit dazwischen geschoben (…das der Empfänger der Nachricht dann wieder rauswirft). Sobald also hintereinander mehr als 5 gleiche Bits auf dem Bus beobachtet werden, klingeln die Alarmglocken und eine Fehlerbehandlungsroutine muss sehen, was sie retten kann.

3) Die CRC – Fehlererkennung
Das ist der bekannte Prüfsummencheck, wie er auch oft bei anderen Übertragungsarten verwendet wird.

4) Acknowledgement – Fehlererkennung
Bei der Besprechung des Datenrahmens haben wir das Prinzip schon angedeutet. Noch während der Sender das Acknowledge-Bit rezessiv ausstrahlt, reagieren die angeschlossenen Empfänger darauf mit einer Umschaltung dieses Bits auf „dominant“. Fehlt diese Reaktion, dann weiß der Sender, dass da etwas auf dem Bus nicht stimmt.

5) Format-Fehlererkennung
Bestimmte Teile im Datenrahmen (CRC-Delimiter, Acknowledge-Delimiter, EOF) bestehen **immer aus rezessiven Bits** und wenn da plötzlich etwas Anderes ankommt, merkt das auch der Dümmste….

Wer sich für die genauen Fehlerbehandlungs-Routinen interessiert, möge in der ISO 11898 nachsehen. Es gilt jedoch dieses eiserne Prinzip:

Entweder Alles oder Nix!

(= entweder empfangen ALLE Stationen korrekte Daten und sind zufrieden ODER Keiner darf etwas verwenden, solange nicht alles stimmt und auf die Reihe gebracht ist).

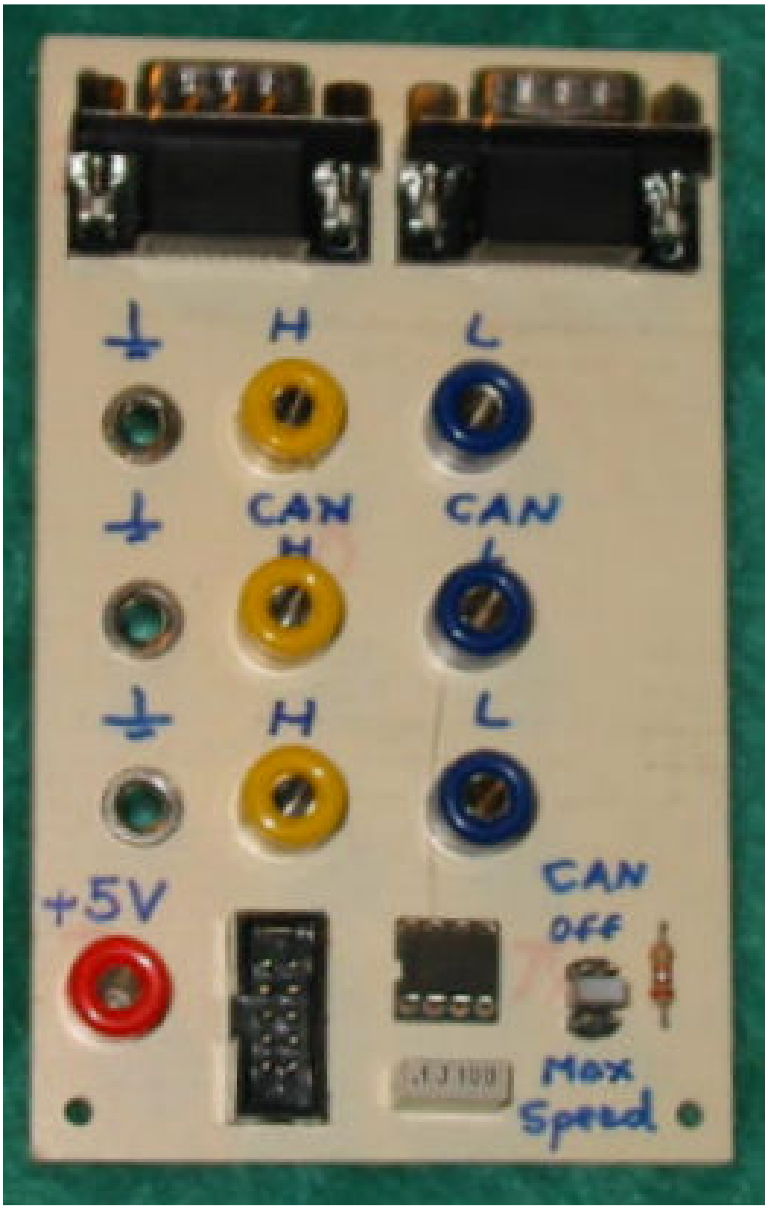
3.5. Bus-Arbitrierung zum Vermeiden von Konflikten

An diesem Punkt wollen wir uns klarmachen, weshalb die Sache mit den rezessiven und dominanten Bits so ein toller Trick ist.
Wir wissen, dass jede Station, die sendet, automatisch jedes auf den Bus gelegte Bit kontrolliert und mit der „Vorgabe“ vergleicht. Taucht nun auf dem Bus plötzlich ein dominantes Bit auf -- obwohl der Sender ein rezessives Bit von sich gegeben hat! -- dann kann das vom „Sender“ nicht mehr zurückgesetzt werden. Also MUSS jetzt geprüft werden, was da los ist und was zu geschehen ist.

Sehr schön sehen wir das, wenn wir annehmen, das zwei Stationen genau zum gleichen Zeitpunkt anfangen zu senden. Jeder Frame geht los mit dem SOF-Bit -- das ist bei beiden identisch und es passiert nichts. Dann folgt der 11 Bit – Identifier und darin geht es solange gut, bis sich die Nachrichten-Nummern zu unterscheiden beginnen (… ist ja logisch, dass beide Stationen kaum dieselbe Nachricht mit derselben Nummer senden wollen).

Sobald nun im Identifier bei der einen Nachricht eine Stelle mit einer logischen Eins („rezessiv“) und bei der anderen Nachricht an derselben Stelle ein logische Null („dominant“) kommt, gewinnt die logische Null und der Teilnehmer mit der logischen Eins muss ab jetzt den Mund halten!

4. Die neue CAN-Zusatzplatine für unser ATMEL ATM1-Controllerboard



Sie ist im üblichen Zusatzplatinen-Format unseres Boards gestaltet und enthält im oberen Teil die Anschlüsse für den CAN – Bus (wir brauchen zwei, wenn wir mehrere CAN-Teilnehmer am Bus haben wollen, nämlich „Zugang“ und „Abgang“.

Dafür stehen ein Sub D9 – Stecker und eine Sub – D9 – Buchse an der Stirnseite der Platine zur Verfügung. Für die Verbindung zum nächsten Teilnehmer ist dann ein ungekreuztes („Through“ -) Kabel mit einer Buchse und einem Stecker am Ende erforderlich.

Zusätzlich und als Alternative sind noch drei EIN-AUSGÄNGE in Form von Telefonbuchsen vorgesehen. Der dritte „Telefonbuchsen-Ausgang“ ermöglicht die Beobachtung der Vorgänge auf den Busleitungen mit dem Oszilloskop.

Der Abstand auf der Platine zwischen einer CAN_H – Buchse und ihrer zugehörigen CAN_L – Buchse beträgt 19 mm, damit kann man auch entsprechende käufliche Steckverbinder mit diesem Stiftabstand einsetzen.

Dient die Platine als „letzte Station“ am CAN-Bus, dann muss ein 120 Ohm Widerstand zwischen CAN_L und CAN_H angebracht werden (= Bus Termination).

Rechts unten erkennt man ein kleines Jumperfeld mit drei Stellungen. Damit wird die Spannung an Pin 8 des Treiberbausteines über den Zusatzwiderstand Rs verändert. Es stehen folgende 3 Möglichkeiten zur Verfügung:

a**) CAN OFF (Baustein in Standby-Betrieb)**

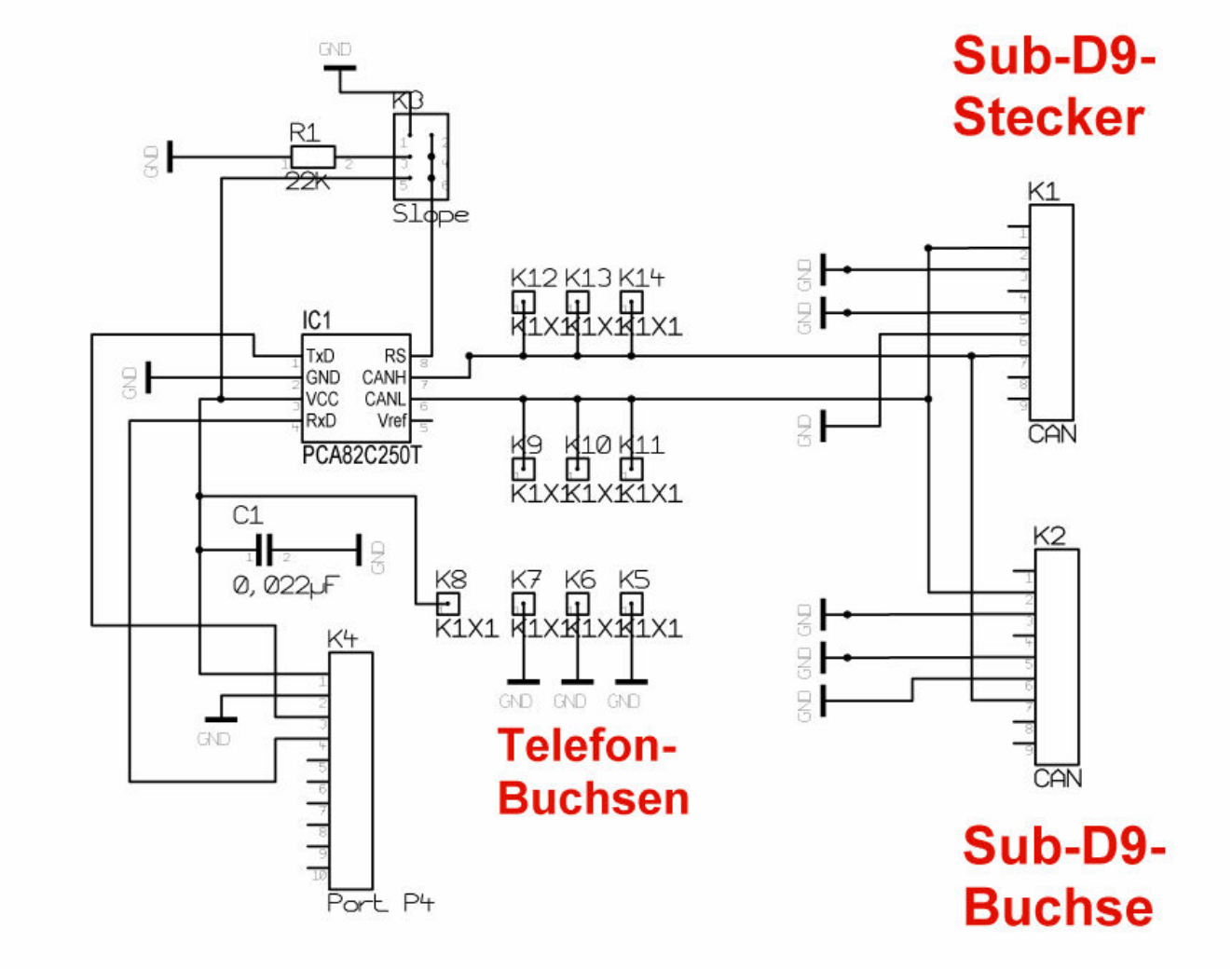
b) **Mäßige Flankensteilheit für den Betrieb mit maximal 50 Kilobit / Sekunde**

c) **Große Flankensteilheit für Highspeed-Betrieb (1 Megabit / Sekunde)**

Unten links folgen noch eine +5V – Buchse (…mit der die über das Flachbandkabel vom Controller angelieferte Versorgungsspannung zugänglich wird) und der bekannte Pfostenfeld-Stecker, mit dem über das Flachbandkabel die Verbindung zum Controllerport P4 hergestellt wird. Dabei gilt folgende Stiftbelegung:

Portpin P4.0 liefert das TxD - Signal und liegt an Pin 3 des Pfostenfeldsteckers
Portpin P4.1 liefert das RxD – Signal und liegt an Pin 4 des Pfostenfeldsteckers

Auf der nächsten Seite findet sich der Stromlaufplan des Boards mit dem Treiberbaustein PCA82C250.


Sub-D9-
Stecker
GND
K3
R1
22K
Slope
K1
K12 K13 K14
K1X1 K1X1 K1X1
IC1
TxD
GND
VCC
RxD
RS
CANH
CANL
Vref
PCA82C250T
CAN
K9 K10 K11
K1X1 K1X1 K1X1
C1
0,022µF
K2
K8 K7 K6 K5
K1X1 K1X1 K1X1 K1X1
K4
Telefon-
Buchsen
Port P4
Sub-D9-
Buchse

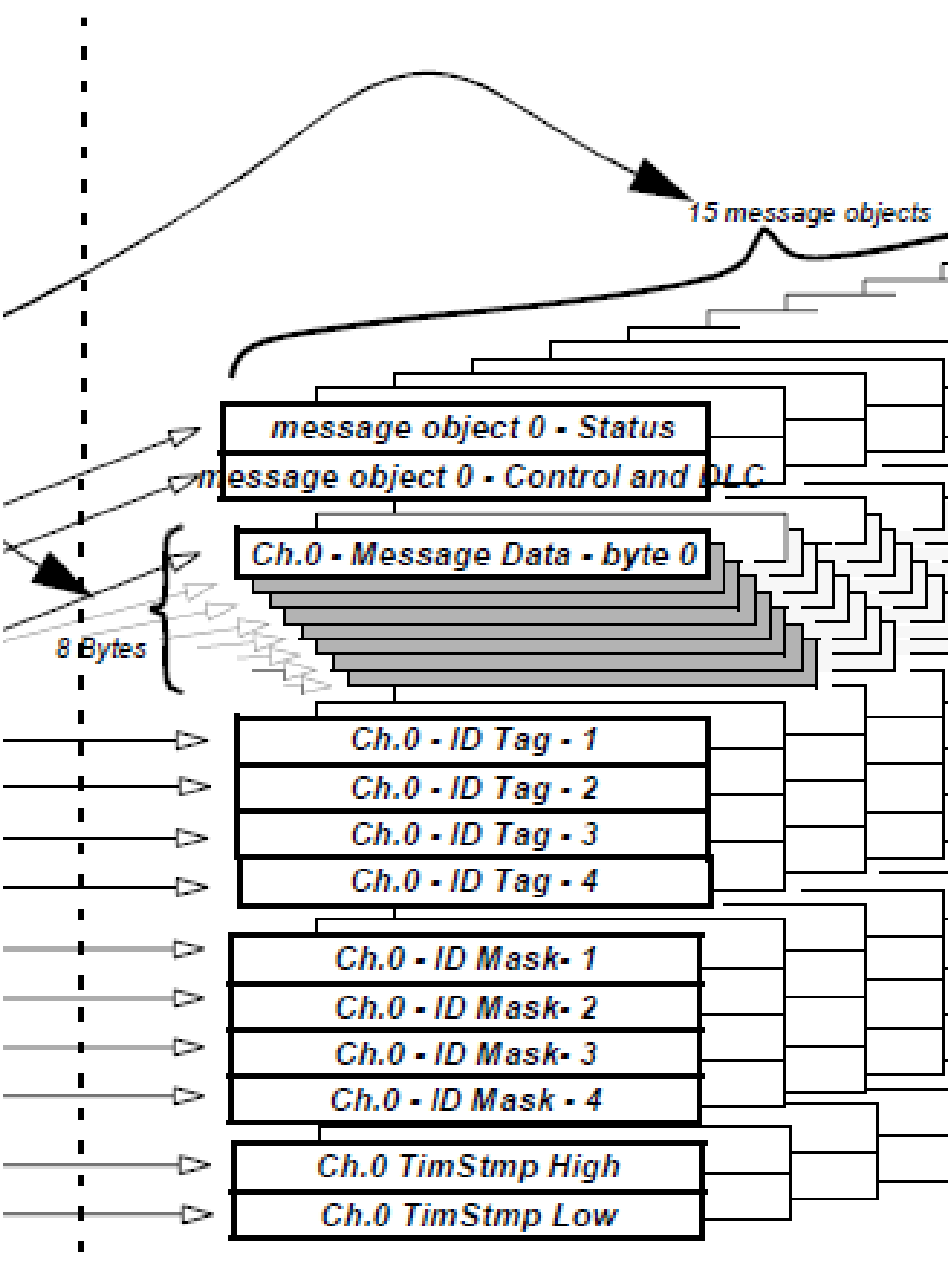
5. CAN-Betrieb mit dem ATMEL – Controller AT89C51CC03

5.1. Organisation der CAN-Maschine auf dem Chip

Da geht es wirklich wild zu, denn wir brauchen dazu 321 Arbeitsregister zu je 8 Bit und 34 Spezial- Funktions-Register (SFR's). Jede Menge Zeug, um den Überblick zu verlieren und deshalb hilft dabei nur eine strenge Systematik. Die sieht so aus:

Es wird mit 15 „Message Objects" gearbeitet. Jedes dieser Objekte kann eine CAN-Nachricht mit maximal 8 Datenbytes aufnehmen und wird von entsprechenden SFR's verwaltet und gesteuert.

So muss man sich das vorstellen:



Jetzt schon sind die Bereiche für den

„Data Length Code" (= DLC),

die 8 Message Data Bytes und -- am Ende! --

die beiden Bytes für den „Zeitstempel" (= Time Stamp) zur Kennzeichnung jeder Nachricht

gut zu erkennen.

Für diesen „Zeitstempel" ist ein eigener CAN-Timer auf dem Chip integriert, der nach dem Start des CAN - Betriebs

(gleich merken: durch Setzen des ENA – Bits im CANGCON – Register)

automatisch gestartet wird.

Dazu kommen **4 Register (= ID Tag-1 …ID Tag 4) zum Umgang mit dem Indentifier (…dran denken: bei CAN 2.0B haben wir nämlich einen 29 Bit -Identifier, deshalb brauchen wir 4 x 8 Bits…..).**

Und als krönenden Abschluss finden wir **4 Register mit der Bezeichnung „ID – Mask".** Damit können wir gezielt durch logische UND-Verknüpfung mit der Identifier- Nummer nach bestimmten Nachrichten fahnden.

Wer es ganz genau wissen will: das Ganze geht los mit dem „**Message Object Status**". Damit können wir zwischen ***„Transmit Message Object", „Receive Message Object", „Receive Message Buffer Object" und „Disable" umschalten.*** Details kriegen wir später….