2. Technische Grundlagen

2.1. SPI

Das Serial Peripheral Interface (SPI) ist eine Serielle Schnittstelle, welche meist zur Kommunikation zwischen einem Mikrocontroller und Peripheriegeräten eingesetzt wird. SPI kommt zum Einsatz, da es sich um eine sehr schnelle und zuverlässige Kommunikationsart handelt, so dass bis zu 20 MByte/s erreicht werden können [1, S. 213]. SPI arbeitet dabei als Master-Slave-Kommunikation. Das bedeutet, das immer genau ein Mikrocontroller als Master die Kommunikation steuert. Als Slave werden die Peripheriegeräte bezeichnet, mit welchen der Master kommuniziert. Die Anzahl an Slaves ist dabei variabel. Der Master übernimmt die Ansteuerung der Slaves, den Aufbau der Kommunikation oder die Generierung des Signaltakts [2, S. 158]. Um die Kommunikation zwischen dem Master und den Peripheriebausteinen zu ermöglich existieren vier Signalleitungen zwischen den Geräten. Dabei handelt es sich um die Signale Slave Select (SS), Serial Clock (CLK), Master-In, Slave-Out (MISO) und Master-Out, Slave-In (MOSI) [3, S. 220], wie in Abbildung 1 dargestellt. Während SCK, MISO und MOSI einmal pro Master vorgesehen sind und alle Slaves über dieselben Leitungen kommunizieren, muss jeder Peripheriechip über einen eigenen Chip Select verfügen. Diese Leitung dient zur Adressierung des Chips [1, S. 214]. Um mit dem Chip kommunizieren zu können, muss das Bauteil über die SS-Leitung angesprochen werden. Standardmäßig handelt es sich um ein lowaktives Signal, sodass der Pin gegen Masse gezogen werden muss, um den Chip zu aktivieren. Erst wenn der Slave Select aktiv ist, achtet der Chip auf den Takt und beginnt die Kommunikation. Alles, was außerhalb eines aktiven SS-Pins passiert, wird vom Chip nicht wahrgenommen [2, S. 160]. Um Kollisionen in der Kommunikation zu vermeiden, darf immer maximal ein Chip angesprochen werden. Wenn es nur einen Peripheriebaustein gibt, welcher mit dem Mikrocontroller verbunden ist, kann die Chip Select Leitung auch entfallen und der Pin am Slave dauerhaft gegen Ground gezogen werden. Dadurch wird das System jedoch störanfälliger [3, S. 160]. Da es sich bei SPI um eine synchrone Schnittstelle handelt, müssen alle Geräte mit demselben Takt versorgt werden. Dieser wird vom Master generiert [4, S. 359]. Die Konfiguration dieses Signals findet im Master statt. So wird über den Parameter Clock Polarity (CPOL) der Ruhezustand des Takts festgelegt. Für CPOL = 0 ist der Takt Highaktiv, ansonsten lowaktiv [1, S. 216]. Die Kommunikation zwischen den

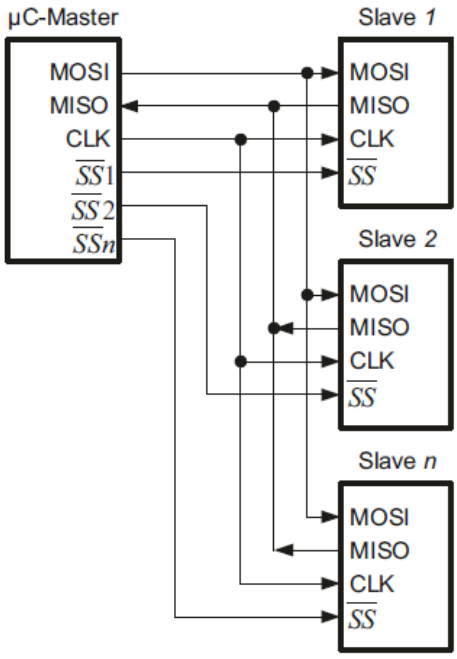

Abbildung 1:Schematische Darstellung der Kommunikation mehrerer SPI-Slaves, aus [1, S. 214]

beiden Teilnehmern findet über die Signalleitungen MISO und MOSI statt. Die Verwendung von zwei Leitungen zur Kommunikation ermöglich Full-Duplex-Übertragung, das bedeutet, dass bei Kommunikation beide Teilnehmer gleichzeitig senden und empfangen [2, S. 159]. SPI funktioniert dabei wie ein Ringpuffer. Jeder SPI-Teilnehmer besitzt ein Schieberegister mit üblicherweise 8bit Breite. Der Slave sendet seine Daten bitweise über den MISO-Pins an den Master, während der Master über die MOSI-Leitung seine Daten gleichzeitig an die Peripherie sendet [1, S. 215]. Die Empfangenen Daten werden ihrerseits wieder in das Schiebregister geschoben. Nach 8 Takten, entsprechend der Breite des Registers, haben beide Teilnehmer ihre Daten vollständig ausgetauscht [3, S. 220]. Dieser Ablauf ist schematisch in Abbildung 2 dargestellt.

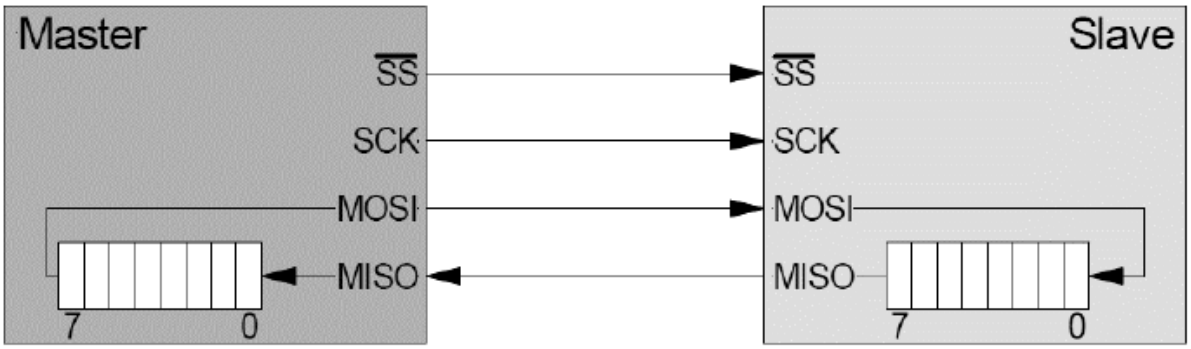

Abbildung 2: Schiebemechanismus der SPI-Kommunikation zwischen zwei Schieberegistern, aus [1, S.216]

Damit diese Kommunikation funktionieren kann, müssen sich Master und Slave auf gleiches Verhalten einigen. Es gibt 4 unterschiedliche Übertragungsmodi, welche sich durch Polarität des Taktes und die Taktphase unterscheiden [1, S. 216]. Die Polarität des Takts wurde bereits erläutert. Die Taktphase definiert gibt an auf welche Flanke die Teilnehmer ihre Kommunikation beginnen. Die Einstellungen nimmt der Master vor, indem er das CPHA-Register beschreibt. Mit dem Wert 0 beginnen Master und Slave ihre Kommunikation auf der ersten fallenden Flanke, nachdem Chip Select gegen Masse gezogen wurde. Mit der folgenden steigenden Flanke wird die Busleitung abgetastet, bis zur nächsten Fallenden Flanke das nächste Bit gesendet wird [3, S. 223].

2.2. I^2C

Neben SPI kommt als serieller Peripheriebus der Inter Integrated Circuit Bus (I^2C-Bus) vor. Genau wie bei SPI handelt es sich hierbei um einen seriellen Bus für synchrone Kommunikation mit Peripheriekomponenten wie Speicherchips. Im Gegensatz zu SPI ist I^2C deutlich langsamer. So kommuniziert der Bus im Standardmodus mit 100 kBit/s. Später kam noch ein schnellerer Fast-Mode hinzu, welcher bis zu 400 kBit Datenrate unterstützt [1, S. 228f]. Dafür ist der I^2C Bus deutlich sparsamer im Umgang mit benötigten Mikrocontroller Pins. Während SPI 3 Leitungen und zusätzlich für jeden Slave noch eine weitere benötigt, kommt I^2C unabhängig von der Anzahl der Busteilnehmer mit zwei Leitungen aus. Diese sind zum einen eine Taktleitung (SCL) und zum anderen die

bidirektionale Datenleitung (SDA) [2, S. 253]. Ähnlich zu SPI funktioniert auch I^2C nach dem Master-Slave-Prinzip, unterstützt dabei jedoch mehrere Master als Busteilnehmer [4, S. 385]. Als Master fungiert auch hier die Komponente, welche den Takt vorgibt und die Kommunikation initiiert und beendet [3, S. 242]. Da es bei I^2C keine Chip Select Leitungen wie bei SPI gibt, werden die Slaves je über eine eigene definierte Adresse angesprochen. Wird ein Slave erfolgreich adressiert, antwortet dieser mit einem Acknowledge-Bit. So erkennt der Master, dass er mit dem Slave eine Kommunikation starten kann [1, S. 230]. Die Signale für I^2C werden über eine Open-Drain Ausgangsstufe realisiert. Dabei wird ein n-kanal MOSFET als Schalter verwendet, welcher die Signalleitung gegen Masse zieht. Damit wird ein logischer Low-Pegel erzielt [4, S. 385]. Durch die verketteten Open-Drain-Transistorschaltungen wirkt diese wie ein logisches UND, da eine 1 nur dann vorliegen kann, wenn niemand auf den Bus zugreift. Dadurch wird ein dominanter Low-Pegel erzeugt [1, S. 230f]. Durch die Open-Drain Schaltung ist der Bus nicht in der Lage selbst einen High-Pegel zu erzeugen. Daher müssen die Daten- und die Taktleitungen mit Pull-Up Widerständen gegen VCC verbunden werden. Üblicherweise werden dafür 4.7 kΩ vorgesehen [5, S.350f], wie in Abbildung 3 gezeigt.

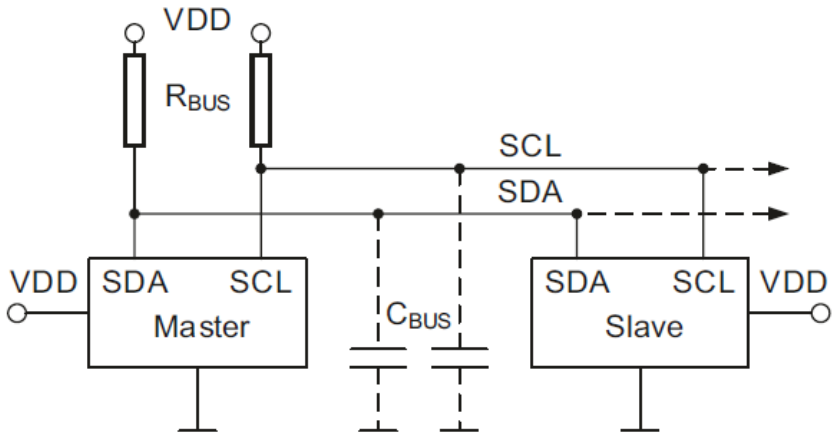

Abbildung 3: Schematischer Aufbau der Signalleitung bei I^2C-Kommunikation, aus [3, S. 243]

Die Kommunikation an sich wird immer durch den Master initialisiert. Findet keine Kommunikation statt, sind beide Leitungen high. Zur Initiierung einer Kommunikation sendet der Master eine Startbedingung. Dabei wird während eines High-Pegels der Taktleitung, eine fallende Flanke auf der Datenleitung erzeugt [2, S. 255]. Danach sendet der Master die Adresse des Chips mit einem Read/Write Bit und nach Erhalt eines Acknowledge-Bits vom Slave die entsprechenden Daten. Dabei ist das Timing entscheidend, damit eine stabile Kommunikation stattfindet. Anders als bei der Startbedingung ist während der Kommunikation kein Pegelwechsel auf SDA während eines High-Pegels auf der Taktleitung erlaubt. Die Daten werden immer während des High-Pegels auf SCL abgetastet und müssen in diesem Bereich Stabil sein [4, S. 387]. Zum Abschluss der Übertragung sendet der Master eine Stopbedingung. Diese funktioniert ähnlich wie die Startbedingung, nur erzeugt der Mikrocontroller eine

steigende statt einer fallenden Flanke während des SCL-High [2, S. 255].
Dadurch, dass es sich bei I^2C um einen Multimaster-Bus handelt, kann es auch passieren, dass zwei Master zeitgleich versuchen auf den Bus zuzugreifen. In diesem Fall würde es zu Kollisionen kommen, welcher durch Arbitrierung verhindert werden [3, S. 243]. Dafür liest jeder Master, während er selbst sendet, die Daten vom Bus. Stellt er fest, dass auf der Datenleitung eine 0 anliegt, obwohl er selbst eine 1 gesendet hat, erkennt er, dass neben ihm noch jemand versucht auf dem Bus zu senden und beendet seine Kommunikation. Dadurch ergibt sich eine priorisierte Verteilung der Adressen. Je kleiner die Adresse, desto mehr nullen werden zu Beginn der Adresse gesendet, sodass die niedrigste Adresse Vorrang gegenüber allen anderen erhält [1, S. 233].

2.3. CAN

Neben der nur für sehr kurze Strecken geeigneten I^2C-Kommunikation, kommt für Datenaustausch zwischen Mastern über lange Distanzen häufig ein Controller Area Network Bus (CAN) zum Einsatz. Anders als I^2C und SPI ist CAN nicht für den Austausch von Daten zwischen Mikrocontroller und Peripherie auf einer Platine gedacht, sondern kommt vor allem in der Fahrzeugtechnik oder Automatisierungstechnik zum Austausch von Daten zwischen Steuergeräten zum Einsatz. CAN funktioniert dabei nach dem Multi-Master-Prinzip [6, S. 16]. CAN überzeugt dabei mit sehr hohen Datenraten von bis zu 1Mbit/s auf einer Distanz von bis zu 40m. Mit niedrigeren Frequenzen sind sogar Reichweiten bis zu mehreren 100 Metern möglich. Auch CAN kommt dabei mit wenigen Signalleitungen aus. Lediglich zwei Leitungen für die Signalpegel CAN-High (CANH) und CAN-Low CANL) werden benötigt. Das hängt mit der Beschaffenheit von CAN als differenzielles Non-Return-To-Zero (NRZ) Signal zusammen [6, S. 16]. Das bedeutet, dass ein ruhendes Signal nicht auf Masse oder VCC liegt, sondern im Fall von CAN auf 2.5V. Wird eine logische 0 auf den CAN gesendet, wird die Spannung auf der CANH-Leitung auf 3.5V erhöht und auf CANL auf 1.5V verringert. Ein Empfänger wertet nun die Spannungsdifferenz zwischen High und Low aus und erhält so einen Bitwert. Zum einfacheren Verständnis ist das in Abbildung 4 dargestellt. Das macht CAN zu einem sehr störungssicheren Signal, da eingehende Störungen ignoriert werden, da sie auf beide Signalleitungen wirken und die Spannungsdifferenz trotz Störung identisch bleibt [7, S. 125]. Zusätzlich gibt es weitere Maßnahmen, welche für CAN standardmäßig angedacht

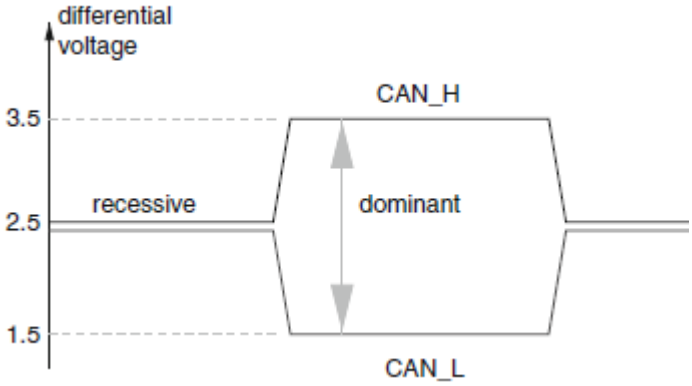

Abbildung 4: Darstellung der Spannungspegel von CANH und CANL, aus [8, S. 10]

sind. So wird die Leitungsimpedanz durch Abschlusswiderstände zwischen CANH und CANL auf 120Ω definiert.

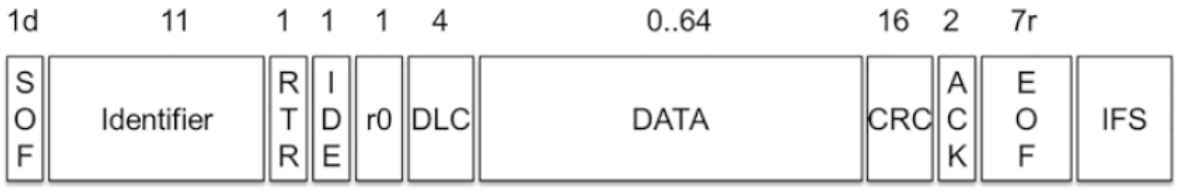

Abbildung 5: Aufbau eines CAN-Paket mit Unterteilung in einzelne logische Bereiche, aus [3, S. 265]

Eine CAN-Kommunikation besteht dabei aus definierten immer gleich aufgebauten Paketen und folgt dabei der Darstellung in Abbildung 5. Ein Master beginnt die Kommunikation mit einem Start of Frame Bit. Dabei wird ein dominantes Bit nach mindestens 11 rezessiven gesendet, wodurch der Beginn einer Nachricht signalisiert wird [3, S. 265]. Darauf folgt der Identifier. Über diesen kann die Nachricht vom Empfänger ausgewertet werden, da es sich bei CAN um eine nachrichtenorientiertes Bussystem handelt. Jede Nachricht besitzt eine eigene, eindeutige ID [6, S. 16]. Das RTR-Bit ein Remote Transmission Request. Dieses wird gesetzt, wenn ein Knoten eine bestimmte Nachricht erwartet. Auf die gesendete Nachricht reagiert der eigentliche Absender des Pakets mit dem Senden des echten Pakets. Danach folgen 6 Kontrollbits. Die DLC ist der Data Length Code, welcher die Anzahl an Datenbytes übermittelt, welche das Paket enthält [8, S. 15]. Die Daten sind der nächste Abschnitt innerhalb des Pakets. Ein Paket kann bis zu 8 Byte an Daten übermitteln [3, S. 265]. Die folgenden 16 Bit sind eine Checksumme für die bisher übertragene Nachricht. Der Sender berechnet seine Checksumme aus den gesendeten Daten und übermittelt diese mit dem Paket. Der Empfänger wiederum berechnet seine eigene Checksumme und vergleicht diese mit der gesendeten. So können Fehler innerhalb der ID, den Kontrollbits oder den Daten erkannt werden. Zusätzlich wird ein Delimiter für das CRC-Feld gesendet. Auf das CRC folgt ein Acknowledge-Bit. Dieses wird von einem Empfänger auf einen Dominaten Pegel gesetzt, sobald das Paket empfangen wurde [8, S. 16f]. Beendet wird die Nachricht durch eine End-of-Frame Sequenz, welche aus 7 rezessiven Bits besteht. Diese bildet zusammen mit dem rezessiven, ungenutzten Bus die Voraussetzung für ein neues SOF [7, S. 139]. Genau wie bei I^2C besteht auch bei CAN als Multi-Master-Bus die Möglichkeit, dass mehrere Steuergeräte gleichzeitig versuchen Daten zu senden. Damit das nicht passiert, gibt es Arbitrierungsverfahren. Diese funktionieren wie bei I^2C. Ein Sender liest parallel die auf dem Bus geschriebenen Bits. Unterscheiden sich die Pegel zwischen gesendet und gelesen, beendet der Master seine Übertragung und wartet auf einen freien Bus. Auch bei CAN ist der Dominante Pegel eine logische 0. Somit gewinnt der Knoten, welcher die Nachricht mit dem niedrigsten Identifier sendet, den Bus [7, S. 141]. Das ist auch aus Abbildung 6 zu entnehmen.

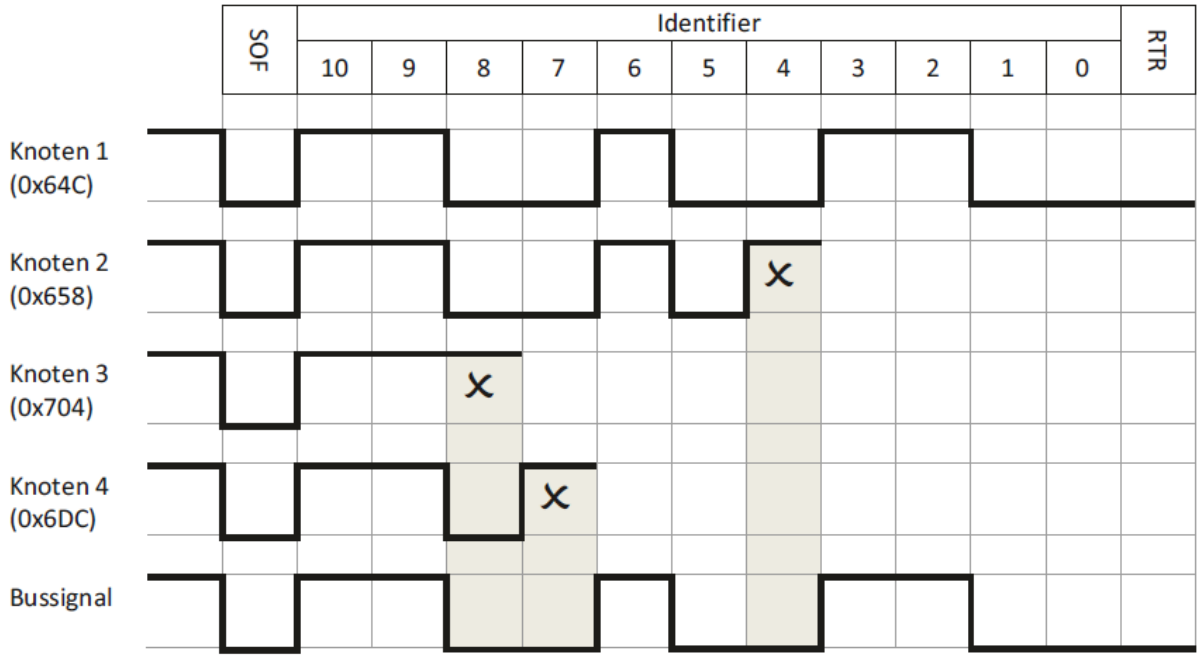

Abbildung 6:Darstellung des Verhaltens bei gleichzeitigem Zugriff mehrerer Master auf den Bus, aus [3, S. 264]

Neben dem Standardframe wie er gerade beschrieben wurde gibt es noch 3 weitere mögliche Paketaufbauten. Einer davon wurde bereits kurz genannt und das ist der Remote frame. Dieser wird genutzt, um eine bestimmte Nachricht anzufordern. Dabei entspricht die Message ID der der angeforderten Nachricht. Daten enthält das Paket keine, die DLC ist aber ungleich 0 und übermittelt die erwartete Anzahl an Bytes. Zusätzlich muss das RTR-Bit rezessiv sein [8, S. 17]. Die zweite alternative Paketvariante ist ein Errorframe. Dieser wird gesendet, wenn ein Busfehlerauftritt und besteht aus 6 aufeinanderfolgenden identischen Bits. Er sorgt damit dafür, dass alle Knoten das Senden einstellen und erzwingt dadurch quasi einen Reset des Busses [6, S. 18]. Zuletzt gibt es einen Overload Frame. Dieser wird von überlasteten CAN-Controllern am Ende eines Standardframes gesendet und besteht aus 6 dominanten Bits. Der Controller erzwingt so eine Warteperiode, bevor ein neuer SOF gesendet werden kann und hat Zeit, seine übermäßigen Nachrichten abzuarbeiten [8, S. 17]. Ein neues Feature bei CAN, welches es vorher bei SPI oder I^2C nicht gab, ist, dass die Knoten selbst in der Lage sind, Fehler zu erkennen und darauf zu reagieren, ohne dass der Programmierer dies groß im Mikrocontroller abfangen muss. Das CAN-Protokoll kennt 5 unterschiedliche Fehler [6, S. 19]. Bereits angesprochen wurde der CRC-Fehler. Der Empfänger berechnet dabei eine Checksumme und vergleicht diese mit der gesendeten. Damit können Fehler von SOF bis inklusive der Daten erkannt werden [8, S. 21]. Eine weitere Fehlererkennung bietet der Check auf korrekte Stuffing Bits. Diese werden zur Synchronisation eingesetzt. Nach fünf identischen Bitpegeln auf dem Bus muss ein invertierter Buspegel gesendet werden, da der Empfänger sonst nicht mehr in der Lage ist Grenzen zwischen Bits zu erkennen und unter Umständen Bits doppelt liest oder überspringt [7, S. 140]. Wird dieser erzwungene Pegelwechsel nach 5 Bits nicht erkannt, liegt ein Stuffing-Fehler vor, welcher vom Controller erkannt und mit einem Error Frame quittiert wird. Dieses besteht aus 6

aufeinanderfolgenden identischen Bits, sodass jeder Knoten einen Stuffing-Fehler erkennt [6, S. 19].

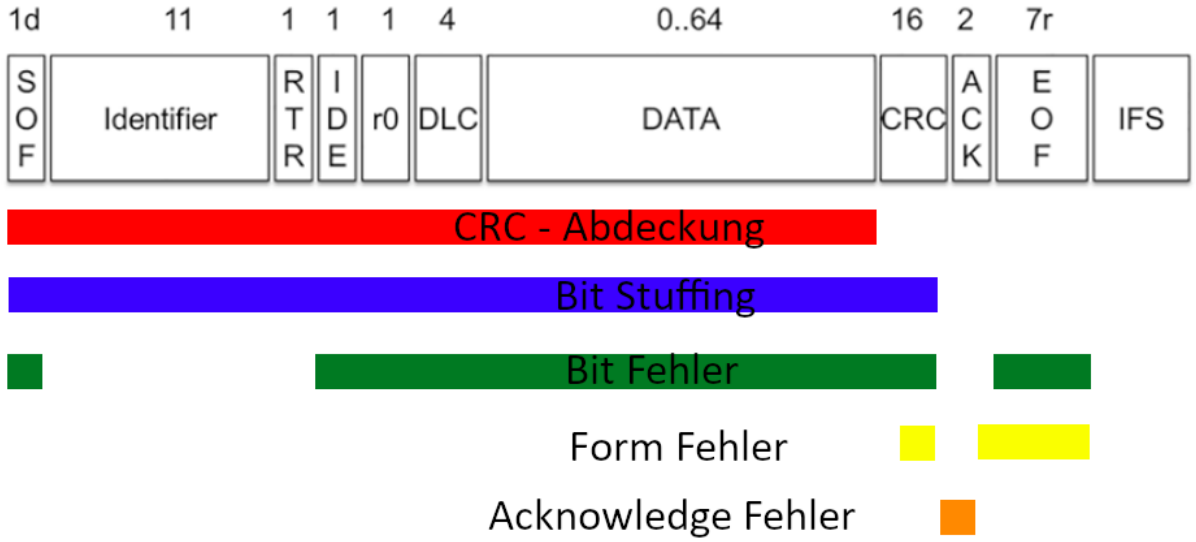

Abbildung 7: Bereichsabdeckung unterschiedlicher Fehler im CAN-Paket, nach

Auch bereits gehört haben wir vom Bit Error. Dieser tritt zum Beispiel auf, wenn ein Knoten die Arbitrierung verliert. Jeder Sendende Knoten hört selbst auf den Bus und erkennt einen Bitfehler, sobald seine gesendeten Bits nicht mehr den Bits auf dem Bus entsprechen [8, S. 21]. Auch einen Fehler erkennt der Controller, wenn niemand den korrekten Erhalt seiner gesendeten Nachricht bestätigt. Das erkennt er durch ein rezessives Acknowledge Bit. Normalerweise würde ein Sender das rezessive Bit vom Sender überschreiben [7, S. 143]. Der letzte Fehler ist ein Form-Fehler. Dieser tritt auf, wenn eine Nachricht ein anderes Format hat als der Empfänger erwarten würde, zum Beispiel durch falsche oder fehlende EOF-Bits oder fehlende Delimiter nach CRC oder Acknowledge [6, S. 18]. Die Abdeckung der einzelnen Bereiche des Pakets ist aus Abbildung 7 erkennbar. Man erkennt dort, dass die Fehlererkennung redundant ist, da jeder Bereich bis auf ACK von mindestens 2 verschiedenen Kontrollmechanismen überwacht wird.
Zusätzlich zu den Error Frames, welche dafür sorgen, dass Knoten aufhören zu senden, besitzt jeder knoten einen Fehlerzähler. Dieser wird erhöht, sobald ein Fehler während seiner Kommunikation auftritt und verringert, wenn der Knoten erfolgreich Nachrichten empfängt. Sobald ein Knoten einen Fehlerzähler > 127 besitzt, darf dieser keine dominanten Error Frames mehr senden, um zu verhindern, dass der Knoten selbst das Problem ist. Ein Knoten mit mehr als 255 Fehlerpunkten schaltet sich ab und nimmt nicht mehr an der Kommunikation teil [3, S. 266].

2.4. UART

Universal Asynchronous Receiver/Transmitter (UART) Kommunikation ist die erste Betrachtete Kommunikationsart, welche kein Bussystem abbildet. Stattdessen stellt UART eine Kommunikation zwischen genau 2 Teilnehmern zu Verfügung. Die Daten werden dabei seriell und asynchron übertragen, das heißt es gibt wie bei CAN keinen Signaltakt, sondern alle Teilnehmer müssen sich vorher auf eine bestimmte Datenrate einigen [9, S. 549]. Wie SPI arbeitet UART mit einer Leitung für Empfangssignale und einer für empfangene Signale. Das ermöglicht Vollduplex, also gleichzeitige Kommunikation. Auch UART arbeitet dabei mit Schieberegistern, welche ihre Daten zwischen Empfänger und Sender bitweise austauschen [3, S. 197]. Zur Kommunikation verwendet UART einen High und einen Lowpegel, welche jeweils eine logische 1 und 0 abbilden. Der Bus ist standardmäßig auf High-Pegel, wodurch der Idle Zustand definiert wird. Die Kommunikation wird über ein Startbit eingeleitet. Als Startbedingung wird die erste Fallende Flanke nach dem Idle Zustand betrachtet [4, S. 329]. Danach beginnt die Übertragung der Daten. Die Anzahl der Übertragenen Daten muss dabei vorher definiert werden. Am Ende wird die Nachricht durch ein Stoppbit eingerahmt. Als solche wird die letzte Flanke von Low zu High betrachtet [9, S. 547]. Dieser Aufbau ist auch in Abbildung 8 dargestellt. Viele Controller fügen noch ein Parity Bit hinzu. Dieses wird nach dem letzten Datenbit und vor dem Stoppbit platziert. Dieses dient als Validierung der Daten. Zur Bildung des Parity Wertes wird der Wert der übertragenen Daten gelesen. Ist dieser Wert gerade, wird das Parity Bit als 0 gesendet, ist der wert ungerade, als logische 1. Der Sender schick sein berechnetes Parity mit und der Empfänger ermittelt es aus den Übertragenen Daten und vergleicht es mit dem gesendeten [10, S. 589].

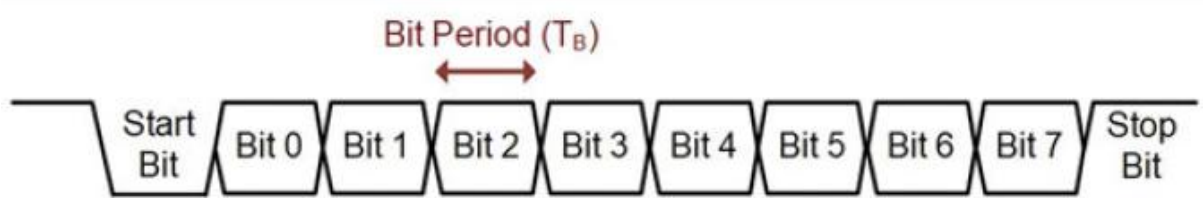

Abbildung 8: Aufbau einer UART-Kommunikation [4, S. 329]

Dadurch ist es dem Empfänger möglich Datenfehler zu erkennen und darauf zu reagieren. Dasselbe gilt auch für Formfehler in der Übertragung. Dadurch, dass die Anzahl der Datenbits, Parity und Stoppbits vorher definiert sein müssen, um eine gültige Kommunikation einzurichten, kann der Empfänger Formfehler erkennen, sollten Bits fehlen oder falsche Stopp oder Startbits empfangen werden [4, S. 330].