Friedrich-Alexander Universität Erlangen-Nürnberg

Lehrstuhl für Elektronische Bauelemente

Prof. Dr. rer. nat. Lothar Frey

Studienarbeit

Entwurf und Entwicklung der Betriebssoftware einer Messplatine für verschiedene Mess- und Regelaufgaben im Hybrid-Fahrzeug

Bearbeiter: Thorsten Spätling
Betreuer: Müsfik Akdere
Abgabe: 14. Mai '12

Selbstständigkeitserklärung

Ich versichere, dass ich diese Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Ort, Datum

Unterzeichner

Zusammenfassung

The thesis' goal was to develop the operating software for an universal and modular measure board. The board's operation area is mainly found in the automotiv sector. Thus a microcontroller with an integrated CAN transceiver was chosen. This allows an easy access to the car's default bus system. Through this interface it's possible to receive data from the board and send data to it. Depending on the application it is also possible to steer the circuit board in this way. But there are much more devices available. There is a real time clock, a digital-to-analog converter, an integrated serial interface and analog-to-digital converter and last but not least a sd card. For all these devices drivers have been programmed and their use is explained in this thesis.

Zweck dieser Arbeit war die Entwicklung der Betriebssoftware für einen universellen, modularen Messadapter, der seinen Einsatz in erster Linie im Automobilsektor finden soll. Aus diesem Grund wurde ein Mikrocontroller mit integriertem CAN-Transceiver gewählt, um einen einfachen Zugriff auf den Standard-Bus im Automobilbereich zu haben. Mittels dieses Busses ist, neben einer Datenausgabe, auch eine Steuerung des Adapters möglich. Weiterhin befindet sich eine Echtzeituhr, ein Digital-Analog-Konverter, eine integrierte serielle Schnittstelle, ein integrierter Analog-Digital-Konverter und eine SD-Karte auf dem Board, die es erlaubt, das Messboard als Datenlogger zu benutzen. Für jedes dieser Geräte wurden Treiber programmiert, die in dieser Arbeit erklärt werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

KAPITEL 1

Einführung

Zu Beginn stellte sich die Frage, ob eine Eigenentwicklung überhaupt von Nöten ist und ob nicht auf eine bereits am Markt befindliche Lösung zurückgegriffen werden kann. Die vorhandenen Angebote haben aber in der Regel den großen Nachteil, dass es sich um spezielle, unflexible Produkte handelt und jedes noch so kleine Extra z.T. teuer zu Buche schlägt. Ziel dieser Arbeit war es deswegen einen universellen, adaptiven Messadapter zu entwickeln, der modular auf die eigenen Bedürfnisse angepasst werden kann, da nicht jedes Projekt die gleichen Anforderungen stellt.
Als Ausgangsbasis wurde der Mikrocontroller AT90CAN128 von Atmel gewählt, da dieser bereits in anderen Projekten Verwendung findet. Dadurch konnte auf einen vorhandenen Erfahrungsschatz zurückgegriffen und eine leichte Einarbeitung sichergestellt werden. Der Mikrocontroller bietet einen integrierten CAN-Transceiver, eine serielle Schnittstelle, einen Analog-Digital-Wandler und unterstützt auf Hardwareebene neben dem CAN-Bus die Bussysteme SPI und TWI, wodurch weitere Komponenten angeschlossen werden können. Dabei handelt es sich um eine Echtzeituhr, einen Digital-Analog-Wandler und eine SD-Karte zur Datenaufzeichnung.
Im nachfolgenden Kapitel wird auf die gewählte Hardware mit ihren elektrischen Eigenschaften eingegangen, ehe im darauf folgenden Kapitel die Befehle zur Benutzung derselben, sowie genutzte Programmierkonzepte beschrieben werden. Dem schließen sich zwei Anwendungsbeispiele an, denen ein abschließendes Fazit folgt.

KAPITEL 2

Grundlagen und Hardware

Wie in der Einleitung bereits angedeutet, widmet sich dieses Kapitel der Messplatine als Ganzes. Dazu gehört das Layout, die eingesetzten Bausteine mit ihren elektrischen Eigenschaften, sowie deren Beschaltung. Zwangsläufig wird auf die verschiedenen Spannungsniveaus und die damit manchmal einhergehende galvanische Trennung eingegangen, ehe zuletzt die unterstützten Schnittstellen betrachtet werden.

2.1 Platinenaufbau

Die Platine auf der alle einzelnen Komponenten aufgebracht sind, wurde mit dem kostenlosen Platinenlayoutprogramm EAGLE von cadsoft[1] entworfen und besteht aus vier Schichten. Die Anordnung der einzelnen Bausteine ist dem Boardlayout in Abbildung 2.1 auf Seite 4 zu entnehmen.

Für die Spannungsversorgung der gesamten Platine wurde der DC/DC Konverter THP 3-2411 von der Firma Tracopower[2] gewählt. Diesem ist ein zweistufiges Tiefpassnetzwerk voraus geschalten, das eventuell auftretende Spannungsspitzen filtert, sowie eine Diode, die beim falschen Anschließen der Versorgungsspannung Schäden verhindert. Der Konverter wandelt eine anliegende Eingangsspannung im Bereich zwischen 9 und 40 V Gleichspannung in eine konstante Ausgangsspannung von 5 V um. Da die Leistungsaufnahme auf 3 W begrenzt ist, ergibt sich ein maximaler Ausgangsstrom von $\frac{3W}{5V} = 0,6A$. Mittels der maximalen Leistungsaufnahme kann nach dem gleichen Schema der maximale Eingangsstrom berechnet werden. Dazu wird ebenfalls der Quotient aus der Leistung und der Eingangsspannung gebildet.

[1] `http://www.cadsoft.de/eagle-pcb-design-software/`
[2] `http://www.tracopower.com/`

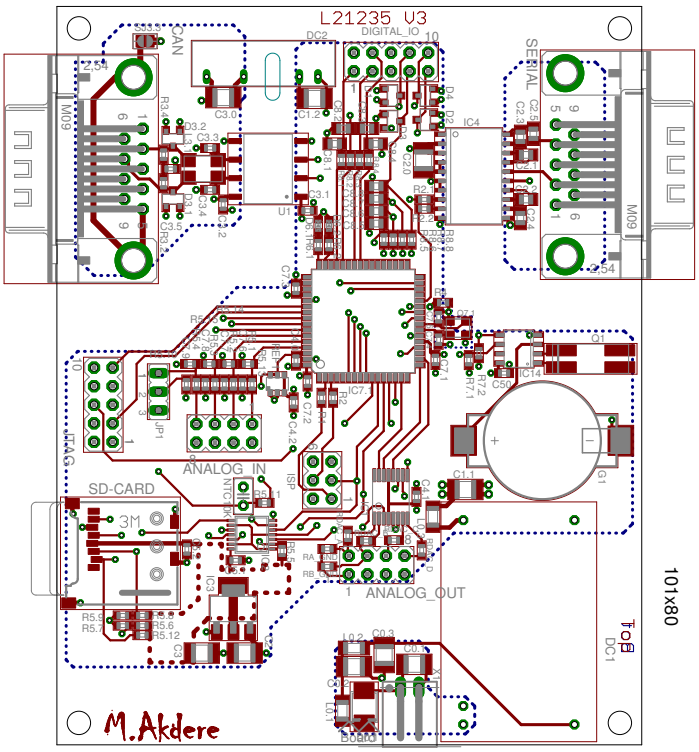

Abb. 2.1: Platineaufbau

Die erzeugte Betriebsspannung von 5 V liegt direkt am Mikrocontroller, dem Digital-Analog-Konverter (*Digital-to-Analog Converter*), der Referenzspannung, der Echtzeituhr und an den Wandlerbausteinen an. Die SD-Karte muss mit einer Spannung von 3,3 V betrieben werden und benötigt deshalb eine Spannungsquelle, die diese Spannung liefert, und einen Pegelwandler (*level shifter*), der die 5 V Ausgangssignale des Mikrocontrollers auf 3,3 V herruntersetzt. Die Wandlung der Pegel wird vom Baustein 74LVX125 übernommen und die Spannungsversorgung durch den Linearregler LM1117-MP3.3. Da über die SD-Karte keine fremden Spannungen eingespeist werden können, ist eine galvanische Trennung nicht nötig. Bei der seriellen und der CAN Schnittstelle ist dies jedoch möglich. Eine galvanische Entkopplung der CAN Schnittstelle ist dabei durch die Verwendung der Bausteine TMV 0505S zur Spannungsversorgung und ISO1050 für die Signalleitungen gegeben. Der Pegelwandler ADM3215E gewährleistet die Trennung für die serielle Schnittstelle.
Neben den bereits erwähnten Bauteilen gibt es noch zwei Quarze und eine Batteriehalterung. Während der Quarzbaustein ABM8 mit 16 MHz schwingt und den Systemtakt bereitstellt, schwingt der Baustein MC-406 mit 32.768 kHz und versorgt die Echtzeituhr aufgrund seiner hohen Güte mit einem sehr genauen Takt. Die Funktion der verwendeten Bausteine ist in Tabelle 2.1 noch einmal zusammengefasst.

Layout-Bez.	**Baustein**	**Funktion**
IC7.1	AT90CAN128	Mikrocontroller
Q7	ABM8	Systemtakt
DC1	THP 3-2411	Spannungsversorgung Platine
DC2	TMV 0505S	Spannungsversorgung CAN
U1	ISO1050	Pegelwandler CAN
IC3	LM1117-MP3.3	Spannungsversorgung SD-Karte
IC2	74LVX125	Pegelwandler SD-Karte
IC14	DS1307	Echtzeituhr
Q1	MC-406	Taktgeber Echtzeituhr
IC4	ADM3251E	Pegelwandler UART
IC1	DAC7564	12-Bit DAC
REF1	ADR3440	Referenzspannung (4,096V)

Tab. 2.1: Auflistung der einzelnen Bauteile

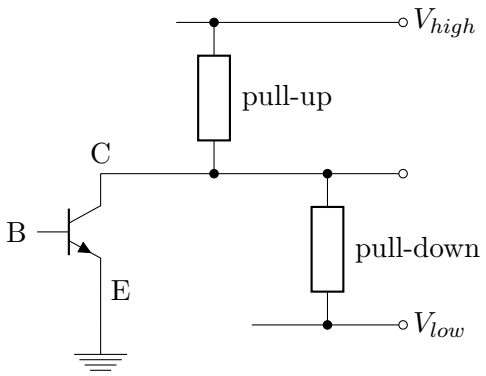

Abb. 2.2: Open-Collector Schaltung

Nachdem nun die aktiven Bauteile genannt wurden, sollen noch einige Worte zu den passiven verloren werden. Wie allgemein bekannt, haben Kondensatoren einen spannungsglättenden Charakter, d.h. sie wirken schnellen Spannungsänderungen entgegen. Deswegen befinden sich an den Versorgungseingängen der aktiven Bausteine Kondensatoren, um kurze Spannungseinbrüche abzufedern. Spulen hingegen wirken stromglättend, d.h. sie unterbinden schnelle zeitliche Stromänderungen und werden deshalb vornehmlich als Eingangsfilter eingesetzt.
Es verbleiben noch die Widerstände mit ihrem rein resistiven Charakter. Die geläufigen Einsatzgebiete sind die der Strombegrenzung, bei den LEDs, oder die Bildung eines RC-Gliedes und damit die Festlegung von dessen Zeitkonstante. Sie werden aber hier auch noch als *Pull-Up*-Widerstände genutzt und da diese Funktion nicht jedem geläufig ist, soll sie im Folgenden erklärt werden.

Die Ausgänge von integrierten Schaltungen (*integrated circuits*) sind i.d.R. als *open*

collector-Schaltung realisiert. Damit ist nichts Anderes gemeint, als dass dieser unbeschaltet ist und quasi „in der Luft hängt“. Ihm ist somit kein festes Potential zugeordnet und die am *Collector* befindliche Spannung *floatet*, d.h. sie variiert zwischen der Versorgungsspannung und Masse. Beschaltet man den *Collector*, wie in Abbildung 2.2 zu sehen ist, mit einem Widerstand, der je nach Anforderung mit der Masseleitung oder der Versorgungsleitung verbunden wird, dann wird über diesen der *Colletor*ausgang entweder auf das Massepotential herunter- (*pull down*) oder auf das Versorgungspotential hochgezogen (*pull up*). Der Widerstand muss in beiden Fällen hinreichend groß (10 kΩ) gewählt werden, was leicht einzusehen ist, denn ein kleiner Widerstand kann als Kurzschluss angesehen werden und würde alle auf der *Collector*leitung liegenden Signale überschreiben. Eine *Pull-Up* Beschaltung befindet sich bei der SD-Karte und bei der Echtzeituhr.
Die Eingänge können über einen programmierbaren Transistor hoch gezogen werden. Dazu muss in den Registern DDxn und PORTxn entsprechende Einstellungen vorgenommen werden. Ein internes Herunterziehen kann nicht eingestellt werden. Das kann nur über externe Widerstände erfolgen.

2.2 Schnittstellen

Um Informationen von A nach B zu übertragen, kann man sich im einfachsten Fall einer Leitung bedienen. Dabei wird z.B. über die Pegeländerung signalisiert, dass ein Prozess beendet wurde. Damit es nicht zu Missverständnissen kommt, müssen Sender und Empfänger zum Einen wissen, welche Information hinter welchem Signal steckt, und zum Anderen erkennen können, ob die empfangenen Informationen valide sind. Nimmt man nun weitere Sender und Empfänger hinzu, so könnte man versuchen, alle Gesprächspartner separat miteinander zu verdrahten. Der Verdrahtungsaufwand ist aber in den meisten Fällen nicht vertretbar. Um diesen auf ein Minimum zu reduzieren, hat man Bussysteme entwickelt. Diese mögen sich hinsichtlich ihrer Topologie unterscheiden, gemeinsam haben sie jedoch, dass sie alle eine geregelte Kommunikation mehrerer Kommunikationspartner gewährleisten. Wie dies vom jeweiligen System sichergestellt wird, kann in den nachfolgenden Abschnitten lediglich angerissen werden. Für detailierte Informationen sei auf die Spezifikationen verwiesen.

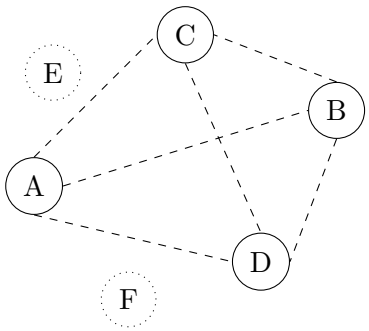

Abb. 2.3: Topologiebeispiel

2.2.1 Two-Wire-Interface

Der Anfang wird mit dem *Two-Wire Interface* (kurz TWI) gemacht. Dabei handelt es sich um einen Bus, der, wie der Name nahelegt, mit lediglich zwei Leitungen betrieben wird. Er ist zum Großteil identisch zum I^2C (*Inter-Integrated-Circuit*) Bus und wurde

von der Firma Atmel eingeführt um eventuellen Rechtsstreitigkeiten vorzubeugen [13].

Die Leitungen an denen die Busteilnehmer angeschlossen werden tragen die Bezeichnungen *serial data line* (SDA) und *serial clock line* (SCL). D.h. auf der einen Leitung werden die Daten und auf der anderen wird der Takt übertragen. Damit erklärt sich, dass Daten nur *half-duplex*, d.h. niemals gleichzeitig in beide Richtungen, übertragen werden können. Prinzipiell gibt es vier verschiedene Betriebsmodi, wie in Tabelle 2.2 aufgeführt, wobei zwei Modi zueinander korrespondieren. Beachtenswert ist, dass der *Master* immer den Takt vorgibt. Damit eine vernünftige Kommunikation zustande kommt, muss der *Master* dem *Slave* genügend Zeit lassen seine Antwort vorzubereiten.

Master Transmission Mode	↔	Slave Receive Mode
Master Receive Mode	↔	Slave Transmission Mode

Tab. 2.2: Betriebsmodi

Die vom Mikrocontroller unterstützten Taktfrequenzen lassen sich mit der Formel (2.1) bestimmen.

$$SCL_{freq} = \frac{16\,\text{MHz}}{16 + 2 \cdot (\text{TWBR}) \cdot 4^{\text{TWPS}}} \tag{2.1}$$

Dabei steht TWBR für das *TWI Bit Rate Register* und TWPS für die Werte der zwei *Prescaler* Bits ((Frequenz-)Teiler) im *TWI Status Register*. Da es sich bei Ersterem um ein acht Bit Register handelt, kann TWBR alle Werte zwischen 0 und $2^8 - 1$ also 255 annehmen. Der Faktor 4^{TWPS} nimmt hingegen die Werte 1, 4, 16 und 64 an.
Die am TWI Bus angeschlossene Echtzeituhr darf laut Datenblatt [9] nur im *Standard Mode* betrieben werden, was einer maximalen Frequenz von 100 kHz entspricht, wie der Tabelle 2.3 entnommen werden kann. Durch Umstellen und Auflösen der angegebenen Gleichung erhält man für die TWPS Bits den Wert 0 und für das TWBR Register den Wert 72.

Modus	**max. Taktrate**
Standard Mode	100 kHz
Fast Mode	400 kHz
Fast Mode Plus	1 MHz
High Speed Mode	3,4 MHz

Tab. 2.3: Taktraten TWI

Um allen Busteilnehmern mitzuteilen, dass eine neue Kommunikation aufgebaut wird, muss zuallererst das START-Symbol gesendet werden. Dieses wird durch das Setzen der Datenleitung von *high* auf *low*, während die Taktleitung *high* ist, generiert. In der Abbildung 2.4 ist dies mit ① markiert. Im An-

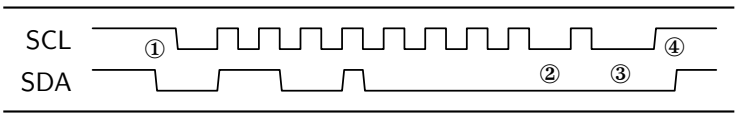

Abb. 2.4: TWI Kommunikation

schluss muss der Master bekanntgeben, mit welchem Busteilnehmer er sprechen möchte

und ob er auf diesen schreiben oder von diesem lesen will. Dazu wird die 7-Bit Adresse des Teilnehmers, gefolgt vom Lese-/Schreib-Bit, auf den Bus gegeben. In Abbildung 2.4 ist dies mit der Adresse der Echtzeituhr (1101 000) und einem Schreibzugriff (0) exemplarisch dargestellt. Ist ein Lesezugriff gewünscht, so muss anstelle einer 0 eine 1 gesendet werden. Nachdem ein Byte (8 Bit) verschickt wurde, muss der Empfänger den Erhalt dieses Bytes quittieren, damit der Sender weiß, dass seine Nachricht angekommen ist. Dazu überlässt der Sender dem Empfänger das elektrische Verhalten der Datenleitung für die Dauer eines Taktes. Wegen der *Pull-up*-Beschaltung würde die Spannung auf das Niveau der Versorgungsspannung steigen und der Sender würde beim nächsten Takt eine Eins (*high*) lesen, die als fehlerhafte Übertragung interpretiert wird. Im Erfolgsfall zieht der Empfänger, wie bei Markierung ③ zu sehen ist, die Datenleitung auf *low*, der Sender liest eine Null und setzt die Kommunikation fort. Das Kommunikationsende wird durch das Senden des STOP-Symbols angezeigt, das durch das Setzen der Datenleitung von *low* auf *high*, während die Taktleitung high ist, generiert wird. Dies ist mit ④ markiert.

So einfach der Kommunikationsverlauf ist, so unsicher ist er. Denn es gibt keinerlei Fehlererkennung, keine Paritätsbits und erst recht keine Prüfsummen. Für sicherheitskritische Anwendungen kommt dieser Bus folglich nicht in Frage. Dafür war er aber auch nie gedacht. Der nächste Bus ist hinsichtlich dieser Punkte schon fortschrittlicher.

2.2.2 Serial Peripheral Interface

Das *Serial Peripheral Interface* (SPI) benötigt im Gegensatz zum TWI mindestens zwei weitere Leitungen. Die Aufteilung der Datenleitung in zwei richtungsbezogene Datenleitungen erklärt die eine zusätzliche Leitung und die pro Busteilnehmer nötige Aktivierungsleitung die andere. Während es beim TWI noch möglich war, einen Teilnehmer mal als *Master* und mal als *Slave* zu betreiben, ist dies beim SPI-Bus aufgrund seiner Architektur nicht mehr möglich, denn hier wird der jeweilige *Slave* durch herunterziehen der Aktivierungsleitung (*Chip Select* oder *Slave Select*) selektiert. Die Topologie ist folglich die einer Sternschaltung mit dem *Master* im Zentrum und ringsherum angeordneten *Slaves*, wie in Abbildung 2.5 verdeutlicht ist. Die Aktivierungsleitungen sind in den Datenblättern entweder mit einem Überstrich oder einem voran gestellten Schrägstrich (/) versehen. Damit wird gekennzeichnet, dass es sich um eine *low active* Leitung handelt.

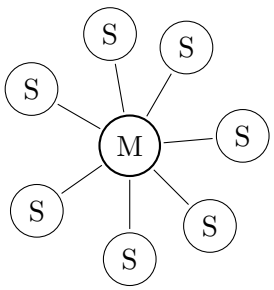

Abb. 2.5: SPI Topologie

high active ist eine ungeläufige Bezeichnung, da es sich hierbei um den Normalzustand handelt und bedeutet, dass das Anlegen eines *high*-Pegels (positive Spannung) den Chip aktiviert.

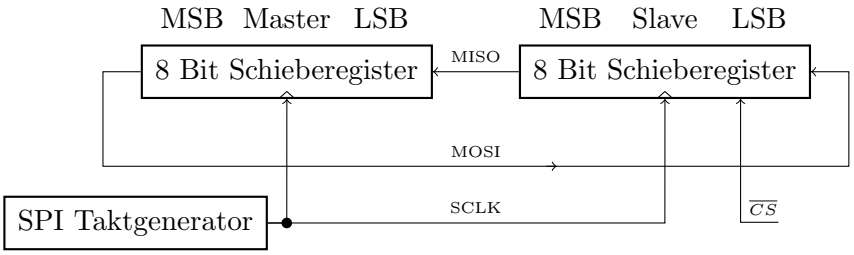

Abb. 2.6: SPI Master-Slave Verbindung

low active ist komplementär zu *high active*. D.h. das Anlegen eines *low*-Pegels (Masse) aktiviert den Chip.

Zu den Aktivierungsleitungen gesellen sich, wie schon erwähnt, zwei Datenleitungen. Auf der Einen werden Daten vom *Master* zum *Slave* übertragen (*Master Output Slave Input* (MOSI)) und auf der Anderen überträgt der *Slave* seine Daten zum *Master* (*Master Input Slave Output* (MISO)). Prinzipiell ist somit eine *full-duplex* Kommunikation möglich. Darstellbar ist der Kommunikationsweg recht anschaulich durch die Verwendung von zusammengeschalteten Schieberegistern wie Abbildung 2.6 zeigt. Was aus dem einen Schieberegister herausfällt, landet im anderen und umgekehrt. Der Takt wird auch hier immer vom Master vorgegeben. Der Mikrocontroller kann den SPI-Bus maximal mit der Hälfte seiner Betriebsfrequenz treiben, was einer Frequenz von 8 MHz entspricht. Da die von den angeschlossenen Geräten unterstützten Taktraten mit 25 MHz für die SD-Karte und 50 MHz für den DAC wesentlich höher liegen, konnte erwartungsgemäß eine fehlerfreie Kommunikation beobachtet werden.

Der Kommunikationsablauf selbst ist in Abbildung 2.7 zu sehen. Vor der eigentlichen Datenübertragung muss die jeweilige *Chip Select*-Leitung des anzusprechenden Geräts von *high* auf *low* gezogen werden, damit dieses weiß, dass die gleich auf den Bus gegebenen Daten und Befehle für es bestimmt sind. Je nachdem, ob ein Befehl gesendet wurde, der eine Antwort erwartet, oder nicht, muss die Aktivierungsleitung weiterhin auf *low* gehalten werden, da der *Slave* nur antwortet, wenn er selektiert ist. Eine Kommunikation in beide Richtungen kann nur mit der SD-Karte betrieben werden, da dem DAC die

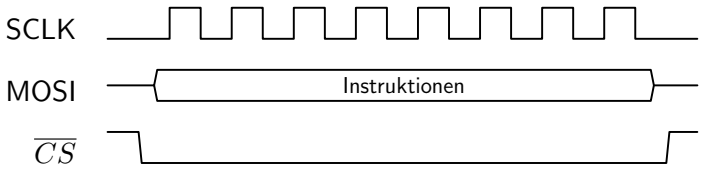

Abb. 2.7: SPI Kommunikation

Rückleitung (MISO) fehlt.

Die Logikpegel sind Masse (*Ground* (GND)) für *low* und die Versorgungsspannung (V_{CC}) für *high*. Da die SD-Karte mit lediglich 3,3 V betrieben werden darf, erfordert dies die Verwendung eines Pegelwandlers, der die zur Verfügung stehende Versorgungsspannung und die Signalpegel von 5 V auf 3,3 V heruntersetzt. Für den DAC ist kein Wandler nötig.

Ein etwas anderer Weg der Kommunikation wurde beim nächsten Bussystem eingeschlagen.

2.2.3 Controller Area Network

Zwischen den bisher besprochenen Bussystem und dem *Controller Area Network*-Bus gibt es ein paar grundlegende Unterschiede. So ist in den bisherigen Systemen eine klare hierarchische Ebene vorhanden, die sich in der Unterteilung in *Master* und *Slave* widerspiegelt. Beim CAN ist diese scharfe Trennung nicht gegeben. Hier ist die Unterscheidung eher fließend und wird durch die Wahl der Adressen festgelegt. Je dominanter die Adresse ist, desto wahrscheinlicher ist es, dass sie auf den Bus zugreifen kann.
Der grundlegenste Unterschied zu anderen Bussen ist jedoch, dass der CAN-Bus nachrichtenbasiert ist. Darunter muss verstanden werden, dass die einzelnen Busteilnehmer den Bus nicht länger als für die Versanddauer einer Nachricht belegen können. Danach ist der Bus wieder frei und das Buszugriffsrecht muss auf ein Neues erworben werden.

Die möglichen Nachrichten (auch Telegramme genannt) haben einen festgelegten aus *Frames* (Rahmen) bestehenden Aufbau für die es insgesamt vier mögliche Aufbauten gibt.

Daten-Frame erlaubt das Übermitteln von bis zu 8 *Byte* langen Daten.

Remote-Frame fordert ein Daten-Frame von einem anderen Teilnehmer an. Damit kann die hierarchische Ordnung umgangen werden.

Error-Frame signalisiert allen Busteilnehmern einen erkannten Fehler.

Overload-Frame hat die Funktion einer Zwangspause zwischen Daten- und Remote-Frames.

Der für die Automobilbranche entwickelte CAN-Bus wird für gewöhnlich als Linienbus aufgebaut. Wie Abbildung 2.8 zeigt, werden dabei die Busteilnehmer einfach nur der Reihe nach mit den Leitungen (CAN_{high} und CAN_{low}) verbunden, die wiederum mit einem Abschlusswiderstand von 120 Ω an jedem Ende versehen sind.

Eingangs wurde bereits erwähnt, dass sich die Priorisierung in der Vergabe der Geräteadresse widerspiegelt. Unter Einbeziehung der Abbildung 2.9 soll diese nun genauer erklärt werden. Wie bei allen gebräuchlichen elektrischen Systemen basiert auch das CAN-Protokoll auf zwei logischen Zuständen: Dem logischen Zustand '1', welcher als rezessiv,

und dem Zustand '0', welcher als dominant bezeichnet wird. Die Namensgebung lässt bereits vermuten, dass dominante Zustände rezessive überschreiben.

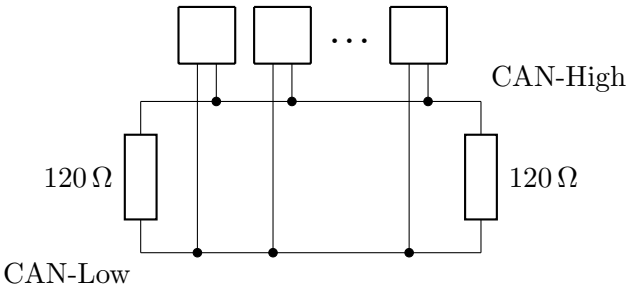

Abb. 2.8: CAN-Bus

Mit dieser Eigenschaft lässt sich nun die Abbildung 2.9 verstehen. Zu Beginn liegt die Busleitung und die Ausgänge aller drei Stationen auf *high*. Dann ändern alle drei gleichzeitig ihren Ausgang von *high* auf *low* und dadurch wird auch die Busleitung auf *low* gezogen. Als Nächstes verändern wieder alle drei Stationen ihren Ausgang, dieses Mal von *low* auf *high*, und die Busleitung verhält sich ebenso. Im nächsten Schritt ziehen nur noch die Stationen 2 und 3 ihre Ausgänge von *high* auf *low* und die Station 1 belässt ihren Ausgang auf *high*. Jetzt kommt es zu zwei Effekten. Zum Einen verhält sich die Busleitung wie die Stationen 2 und 3, da diese eine dominante '0' propagieren und zum Anderen erkennt Station 1, dass die von ihr propagierte '1' nicht auf dem Bus liegt, denn jeder Busteilnehmer, der eine Nachricht auf den Bus schreibt, liest diese zeitgleich wieder aus. Von nun an, hat die Station 1 erkannt, dass sie keine Chance mehr auf den Buszugriff hat und belässt deswegen ihren Ausgang auf *high*. Die beiden übrig gebliebenen Stationen 2 und 3 werden nun ebenfalls nur noch so lange gleichzeitig auf den Bus zugreifen bis einer der beiden ein rezessives Bit propagiert, während der andere ein dominantes Bit propagiert. Im besprochenen Beispiel gewinnt deswegen Station 2 das Rennen um den Buszugriff, was auch Arbitrierung genannt wird.

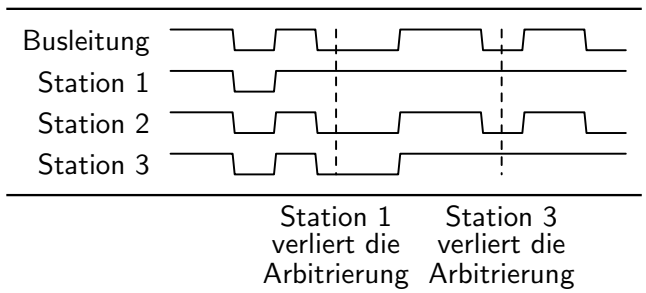

Abb. 2.9: CAN Arbitrierung [4]

Bisher wurde immer von Geräteadressen gesprochen. Der Begriff ist für den CAN nicht treffend, denn wie erwähnt werden beim CAN-Bus Nachrichten verschickt und diese haben keine Geräteadresse sondern eine Identifikationsnummer (ID). Damit ist es auch nicht verwunderlich, wenn ein CAN-Controller mehrere IDs verwaltet. Vorstellen kann man sich das wie bei einem E-Mailserver: Auf diesem sind mehrere Postfächer eingerichtet und ankommende Nachrichten werden anhand ihrer ID dem jeweiligen Postfach zugeordnet. Dabei kann eingestellt werden, ob eine Nachricht eine

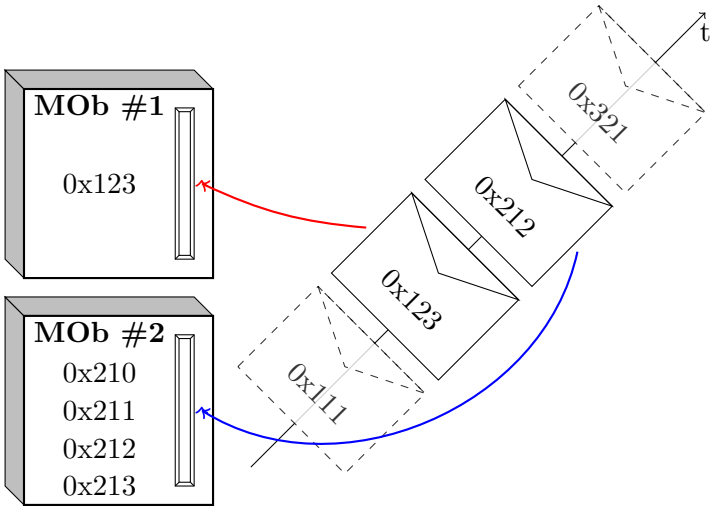

Abb. 2.10: MOb bzw. Mailbox

bestimmte ID haben muss, oder ob auch ähnliche Nachrichten-IDs einem Postfach zugeordnet werden. In Abbildung 2.10 ist dies noch einmal veranschaulicht.

Insgesamt kann der Mikrocontroller bis zu 15 Postfächer verwalten, die richtigerweise als *Message Object* (kurz: MOb) bezeichnet werden. Die MObs sind von 0 bis 14 durchnummeriert, wobei das MOb mit der kleinsten Zahl die höchste Priorität hat. Denn der Mikrocontroller kann im Falle eines Interrupts nicht unterscheiden welches MOb den Interrupt ausgelöst hat und fragt deshalb der Reihe nach die einzelnen MObs ab. Dabei beginnt er beim MOb 0 und damit erklärt sich dessen höchste Priorität.

Eine Identifikationsnummer darf nicht mehrmals vergeben werden und ist je nach Spezifikation zwischen 11 (V2.0 A) und 29 Bit (V2.0 B) lang. Es wurde bereits erwähnt, dass Nullen gegenüber Einsen dominant sind. Wichtige Nachrichten sollten daher eine Identifikationsnummer nahe 0 und unwichtige eine Nummer mit führenden Einsen zugeordnet bekommen. In Abbildung 2.10 ist beispielhaft gezeigt, dass ein MOb nur eine bestimmte oder mehrere Nachrichten-IDs empfangen kann. Der Empfang von Absendern verschiedener IDs macht Sinn, denn wenn eine Nachricht für mehrere Empfänger gedacht ist, kann die Busbelastung gesenkt werden, da die Nachricht nicht mehrmals verschickt werden muss. Außerdem würde sonst ein viel größerer Adressbereich benötigt werden. Festgelegt wird der Empfangsbereich über eine Maske. Diese wird mit der gesetzten MOb-ID logisch verundet und das Ergebnis legt den Empfangsbereich fest. In der Maske gleich Null gesetzte Bits spielen beim Abgleich keine Rolle wohingegen gleich Eins gesetzte Bits eine Übereinstimmung beim betroffenen MOb-ID-Bit und dem Nachrichten-ID-Bit erfordern. Der Sachverhalt wird mit Blick auf die Beispiele in den Tabellen 2.4a und 2.4b klarer.

111	1111	1111	ID-Maske
011	0001	0111	MOb-ID
011	0001	0111	

(a) nur 0x317 ist gültig

111	1111	1000	ID-Maske
011	0001	0xxx	MOb-ID
011	0001	0xxx	

(b) 0x310 bis 0x317 gültig

Tab. 2.4: Maskenberechnung

Damit sind die wesentlichen Grundlagen angesprochen. Wie schon bei den vorherigen Systemen kann auch hier leider nur ein Überblick gegeben werden. Detailierte Informationen lassen sich z.B. der ISO-Spezifikation 11898-1:2003[1] entnehmen. Darin finden sich aber auch keinerlei Angaben zu den Spannungsniveaus der Signale. Denn diese sind bewusst nicht festgelegt, was dem CAN-Bus ein viel breiteres Einsatzspektrum verschafft.

Zuletzt werde noch ein Blick auf die älteste der hier behandelten Schnittstellen geworfen.

2.2.4 Serielle Schnittstelle

Ehe die USB-Schnittstelle (*Universal Serial Bus*) Einzug in die PC Landschaft fand, war die serielle Schnittstelle, damals RS-232 mittlerweile EIA-232 genannt, die Schnittstelle schlechthin, weshalb sie heute noch in vielerlei Geräten anzutreffen ist. Aber im Gegensatz zu den bisher genannten, wurde diese Schnittstelle nicht als Bus konzipiert. Es handelt sich viel mehr nur um eine Punkt zu Punkt Verbindung, die nur einen Datenaustausch zwischen zwei Geräten gestattet.

Prinzipiell erlaubt die Hardware eine synchrone als auch eine asynchrone Übertragung, jedoch spart man sich bei letzterer die Taktleitung, da die Taktinformation bereits in den übertragenen Zeichen enthalten ist. Dabei wird allerdings vorausgesetzt, dass Sender und Empfänger auf die gleiche Taktrate (Baudrate) eingestellt sind. Die zu sendenden Daten werden in 5 bis 9 Bit großen Blöcken übertragen, die von einem Startbit und bis zu zwei Stoppbits flankiert sind. Zur Fehlererkennung kann vor den Stoppbits ein Paritätsbit eingefügt werden, das anzeigt, ob eine gerade oder eine ungerade Anzahl an logischen '1' gesendet wurde. Die nebenstehende Abbildung 2.11 zeigt wie das 7-Bit ASCII-Zeichen

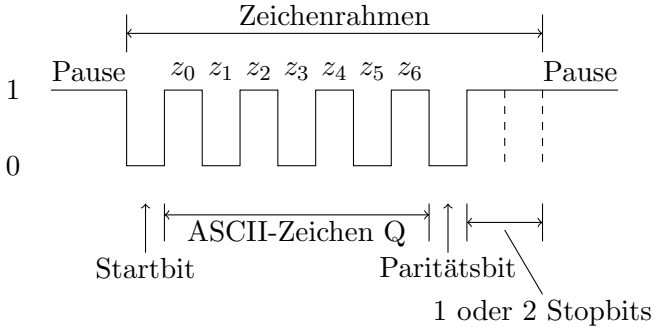

Abb. 2.11: Asynchroner Zeichenrahmen [8, S. 482]

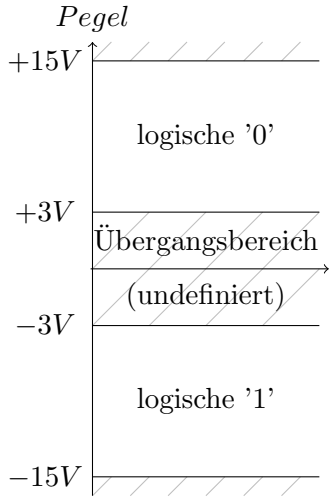

Abb. 2.12: Pegeldefinition bei der RS232-Schnittstelle [8, S. 482]

'Q' übertragen wird.

Wie nun die logische '1' bzw. '0' elektrisch umgesetzt ist, kann Abbildung 2.12 auf Seite 14 entnommen werden. Man erkennt, dass die '0' (auch als *SPACE* bezeichnet) einer positiven Spannung im Bereich zwischen 3 und 12 V entspricht und die '1' (*MARK*) einer negativen Spannung zwischen -3 und -12 V. Da beide Spannungsbereiche genutzt werden, handelt es sich um eine bipolare Schnittstelle. Mikrocontrollern steht im Allgemeinen nur eine positive Versorgungsspannung zur Verfügung. Um auch den negativen Spannungsbereich abbilden zu können, ist es deshalb erforderlich sich eines Pegelwandlers (*level shifter*) zu bedienen. Der dafür gewählte Baustein trägt den Namen ADM3251E und sorgt zusätzlich für eine galvanische Trennung. Diese ist nötig, um ein Überkoppeln der unterschiedlichen Spannungsniveaus zu verhindern, da dies zu Beschädigungen führen kann.

Da jetzt die Funktionen aller vorhandenen Schnittstellen behandelt wurden, widmet sich das nächste Kapitel der Software, die die Nutzung dieser und weiterer Komponenten ermöglicht.

KAPITEL 3

Software

Beim Entwurf der Betriebssoftware wurde viel Wert auf einen modularen und gekapselten Aufbau gelegt. Die Motivation bestand darin ein einfaches Austauschen, Wiederverwenden, Portieren und in erster Linie Warten von Code(-stücken) sicherzustellen. Ein weiterer Gesichtspunkt war es eine möglichst schnelle Verarbeitung zu gewährleisten und soweit wie möglich ein deterministisches Echtzeitverhalten zu erzielen. Prinzipiell bieten sich bei der Mikrocontrollerprogrammierung zwei Methoden an.

Pollingverfahren ermittelt durch zyklisches Abfragen Statusänderungen der abgefragten Hard- oder Software.

Interruptverfahren unterbricht beim Auftreten einer Unterbrechungsanforderung (*interrupt request*) das laufende Programm und führt die Unterbrechungsroutine (*interrupt service routine*) aus. Nach Beendigung dieser wird das Programm an der Unterbrechungsstelle wieder fortgesetzt.

Die erste Methode kennzeichnet i.d.R. eine recht leichte Programmierung und bei einfachen Anwendungen kann sie auch schneller sein, da keine Zeit für das Sichern und Zurückladen des aktuellen Prozesses verloren geht, aber da hier bis zu acht Modulen und mehr gleichzeitig aktiv sein können, kann keine zeitnahe Bearbeitung von eventuell zeitkritischen Daten garantieren werden. Außerdem können Ereignisse die das zeitliche Verhalten eines Moduls beeinflussen, ebenfalls unerwünschte Auswirkungen auf das zeitliche Verhalten anderer Module bzw. des Gesamtsystems haben. Im Extremfall kann ein Modul das Gesamtsystem blockieren (Endlosschleife). Ebenso können selbst minimale Modifikationen am Treiber zu einem gänzlich anderen Zeitverhalten des Gesamtsystems führen. Deswegen wurde die zweite Methode, ein Interrupt-gesteuerter Softwareaufbau, gewählt.
Ehe in den folgenden Abschnitten auf die einzelnen Softwarepakete eingegangen wird,

soll noch schnell ein Wort über die Geltungsbereiche von Variablen verloren werden. Bei der genutzten Programmiersprache C stehen drei verschiedene Gültigkeitsbereiche zur Verfügung.

Globale Variablen sind außerhalb von Funktionen definiert und gelten daher im Modul für alle Funktionen. Wird die Variable zusätzlich in der Header-Datei veröffentlich und mit dem Schlüsselwort *extern* versehen, kann auch außerhalb des Moduls auf sie zugegriffen werden. Diese Variablen werden im Allgemeinen als öffentlich (*public*) bezeichnet.

Lokale Variablen werden innerhalb von Funktionen definiert und sind deswegen auch nur während der Ausführung der Funktion gültig. Mit dem Beenden der Funktion wird der Speicherbereich, den die Variable eingenommen hat, wieder freigegeben.

Statische Variablen sollen nur der Vollständigkeit halber erwähnt werden. Mit dem Schlüsselwort *static* versehene Variablen unterscheiden sich zu den lokalen Variablen dadurch, dass sie, wenn sie innerhalb einer Funktion definiert wurden, nur einmalig initialisiert werden und am Ende eines Funktionsaufrufs ihren Wert beibehalten. Variablen außerhalb von Funktionen verlieren ihren Gültigkeitsbereich nicht, weshalb das Schlüsselwort hier den Sichtbarkeitsbereich der Variable auf die Datei beschränkt in der sie steht. Somit ist es möglich in verschiedenen Dateien gleichlautende globale Variablen zu benutzen. Beide Typen wurde in den Modulen nie benötigt.

Der Geltungsbereich von Funktionen wird in gleicherweise festgelegt werden.

Bei der Nomenklatur (Namensgebung) für die Variablen, Funktionen, Strukturen und anderen Elementen wurde folgendes Schema gewählt.

public Dem in Großbuchstaben geschriebenem Modulnamen (z.B. ADC oder IO) folgt ein Unterstrich und diesem schließt sich ein aussagekräftiger Variablen- bzw. Funktionsname an, in dem kein weiterer Unterstrich enthalten ist und dessen beschreibende (Teil-)Wörter im Rahmen der Lesbarkeit mit einem Großbuchstaben beginnen.

local Lokale Funktionen beginnen ebenfalls mit dem in Großbuchstaben geschriebenem Modulnamen und einem Unterstrich. Dann schließt sich auch hier die Beschreibung an, jedoch sind alle Wörter klein geschrieben und mit einem Unterstrich separiert. Die in der Funktion genutzten Variablen unterliegen keiner Konvention und können beliebige Namen tragen.

In Tabelle 3.1 sind die Bildungsregeln noch einmal kurz zusammengefasst.
Betrachtet man die Module etwas genauer, so fällt auf, dass Funktionen mit Rückgabewert vom Typ TYPE_MODUL_RETURN sind. Dahinter verbirgt sich nichts anderes als ein 8-Bit Integer (Ganzzahl), weshalb bei den kommenden Funktionserklärungen anstelle von TYPE_MODUL_RETURN die Typenbezeichnung U8 verwendet wird. U8 steht

Typ	**Muster**
öffentlich	MODULNAME_FunktionsOderVariablenname
lokal	MODULNAME_funktionsname_mit_beschreibung

Tab. 3.1: Nomenklatur

für einen vorzeichenlosen (*unsigned*) 8-Bit Integer.

In diesem Kapitel wird sich im Wesentlichen auf die öffentlichen (*public*) Funktionen beschränkt, um das Dokument nicht unnötig aufzublähen. Alle weiteren Funktionen und Variablen sind in englischer Sprache in Quellcodedokumentation [12] erläutert.

Bevor in den folgenden Abschnitten die einzelnen Module und deren Selektion behandelt wird, soll anhand der Abbildung 3.1 der Programmablauf veranschaulicht werden. Sobald der Mikrocontroller mit Spannung versorgt ist, werden die einzelnen Module initialisiert. Je nachdem welche und wie viele Module angewählt sind, dauert die Initialisierung zwischen wenigen Millisekunden und einigen Sekunden. Mit dem Abschluss der Initialisierung beginnt die grüne LED im Sekundentakt zu blinken. Von jetzt an ruft das Programm zyklisch die aufgeführten Funktionen (*Tasks*) auf, sobald es sich an der Position der Ausführbedingung befindet und diese erfüllt ist.

3.1 Konfiguration

Nachdem im vorhergehenden Abschnitt der Programmablauf angeschnitten wurde, widmet sich dieser der Konfiguration von selbigem. Die einzelnen Module lassen sich einfach in der Datei setup.h an- und abwählen. Ein Auszug aus dieser zeigt Listing 3.1 und diesem wiederum kann auf den ersten Blick entnommen werden, dass ein Modul angewählt wird, indem das betreffende DEFINE auf den Wert 1 gesetzt wird. Zum Abwählen setzt man es auf 0. Laut dem Auszug sind folglich die Module IO (Ein- und Ausgänge) und DAC (Digital-Analog-Konverter) aktiv.

```
#define MODULE_IO_ENABLE      1   //Ein-/Ausgaenge
#define MODULE_ADC_ENABLE     0   //Analog-Digital-Konverter
#define MODULE_RTC_ENABLE     0   //Echtzeituhr
#define MODULE_SD_ENABLE      0   //SD-Karte
#define MODULE_DAC_ENABLE     1   //Digital-Analog-Konverter
...
```

Listing 3.1: setup.h Ausschnitt

Für die Selektion werden Präprozessoranweisungen genutzt. Dabei handelt es sich um Anweisungen, die vor der eigentlichen Codeübersetzung durch den Compiler abgehandelt werden. Im gerade erwähnten Auszug (Listing 3.1) wird durch #DEFINE eine Variable angelegt, die den ihr nachfolgenden Wert zugewiesen bekommt. Neben der Definition von

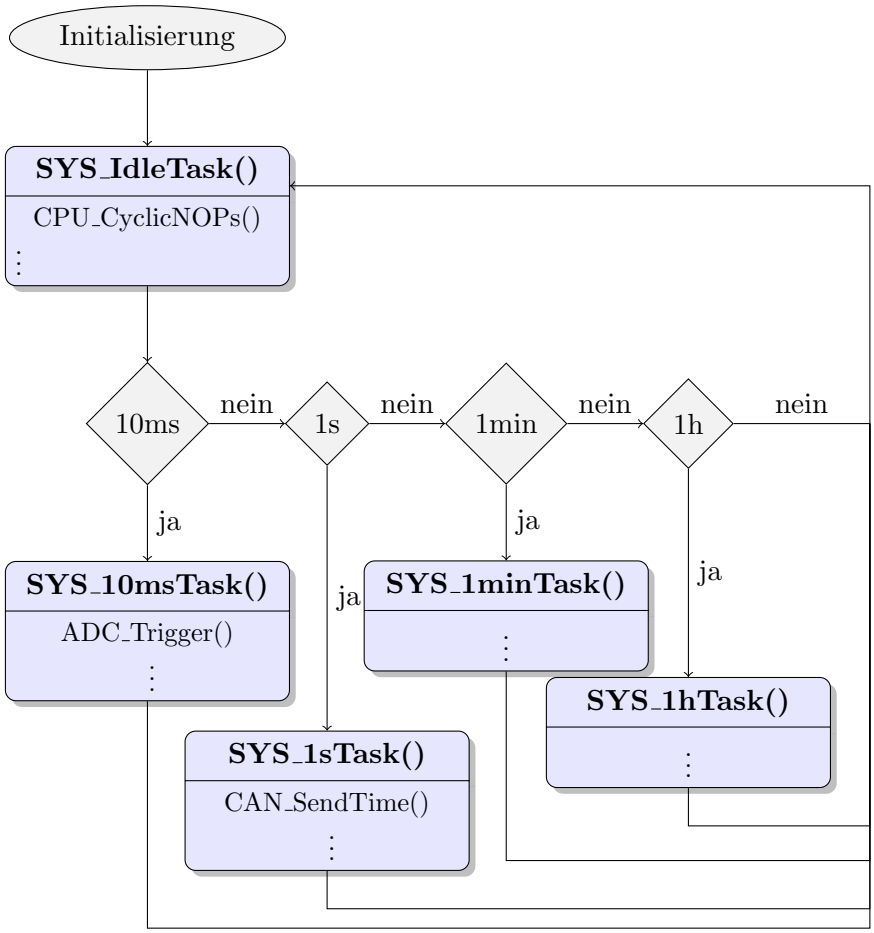

Abb. 3.1: Programmablauf

Variablen lassen sich mittels Präprozessoranweisungen auch Kontrollstrukturen verwirklichen. Z.B. eine if-else-Struktur mit den Befehlen #IF und #ELSE. Dem aufmerksamen Leser mag bereits aufgefallen sein, dass alle bisher genannten Präprozessoranweisungen mit # begonnen haben und die Raute ist in der Tat kennzeichnend für solche einzeiligen Anweisungen. Um nun das Ende einer if-else-Struktur festlegen zu können Bedarf es eines weiteren Befehls namens #ENDIF.
Neben den bisher angeführten Anweisungen gibt es noch weitere. Für die Implementierung der Modulselektion reichen die genannten aber bereits aus. Denn dazu wird am Anfang jeden Moduls einfach abgefragt, ob das Modul selektiert ist oder nicht. Im ersten Fall wird der der Abfrage folgende Code übersetzt, im zweiten nicht. Das Listing 3.2 zeigt dies noch einmal in allgemeiner Form.

```
#include "setup.h"              //Laden der Konfiguration.
#if MODULE_ENABLE_MODULABKUERZUNG //Ueberpruefung, ob das
```

```
                                //Define MODULE_ENABLE_MODULNAME
                                //wahr ist.
...                             //Dann folgt der Code.
#endif                          //Ehe am Ende die Praeprozessor
                                //if-Abfrage geschlossen wird.
```

Listing 3.2: Beispielhafter Aufbau jeder c-Datei

Neben die grobe Konfiguration (an/aus) gesellt sich bei ein paar Paketen noch eine Feinkonfiguration hinzu, die in separaten Dateien zu finden ist. So teilt sich das CAN-Modul in den generellen Treiber (can.c und can.h) und einer darauf aufsetzenden Anwendungsschicht (can_prj.c und can_prj.h) auf und das IO-Modul ebenfalls in einen grundlegenden Treiber (io.c und io.h) und dessen spezifische Konfiguration (io_cfg.c und io_cfg.h). Der Grund für die nochmale Trennung besteht in der einfacheren Fehlersuche, denn der Anwender sollte dadurch keinerlei Veranlassung haben den Treiber selbst anzufassen. Bei einem einzelnen Projekt mag dies kein Problem darstellen. Kommt der Treiber jedoch in mehreren verschiedenen Projekten zum Einsatz, führt dies zu einem schwer überschaubaren Versionswildwuchs.

Auf die Feinjustierung der Module wird in den kommenden Abschnitten eingegangen, doch vorher sollen noch weitere Vorzüge des modularen Softwaredesign dargestellt werden. Denn dieses impliziert nicht nur die Möglichkeit vorhandene Komponenten einfach ab- und anwählen zu können. Vielmehr kann die Parametrierung eines Treibers, wie z.B. die Kalkulation der ADC-Wandler-Frequenz oder die Bitmaskierung für das Aktivieren der ADC-Kanäle, im Präprozessor des Compilers erfolgen, d.h. vor dem eigentlichen Kompilieren, und nicht zur Laufzeit des Programms. Diese sogenannte statische Konfiguration ist für Echtzeit-Systeme der dynamischen Konfiguration grundsätzlich vorzuziehen, da Ressourcen nicht dynamisch zur Laufzeit vergeben werden und wichtige Peripherie-Einstellungen nicht vom Programmablauf oder Ereignissen bestimmt sind. Ein eher beiläufiger Vorteil ergibt sich daraus, dass die Nutzung der Schalter den Entwickler zwingen, „saubere“und schlanke Schnittstellen bereitzustellen.

3.2 State-Machine

Entwirft man Software, die von Interrupts unterbrochen werden kann, ergibt sich zwangsläufig die Problematik, dass irgendwie sichergestellt werden muss, dass der Eingriff von Interrupts in laufende Systemprozesse diese nicht korrumpiert. Im einfachsten Fall bedient man sich sogenannter Semaphoren. Darunter versteht man einfach gesprochen eine Markierung, die aussagt, ob eine Variable in Verwendung ist und (durch den Interrupt) nicht verändert werden darf oder doch.

Erweitert man diesen Gedanken etwas so gelangt man relativ zügig zum Konzept der *State Machine* (Zustandsmaschine). Dabei handelt es sich im Grunde genommen auch nur um eine Variable, aber in dieser wird - wie der Name vermuten lässt - der Zustand

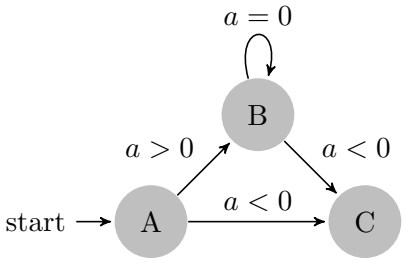

Abb. 3.2: Fehlerhafter Zustandsgraph

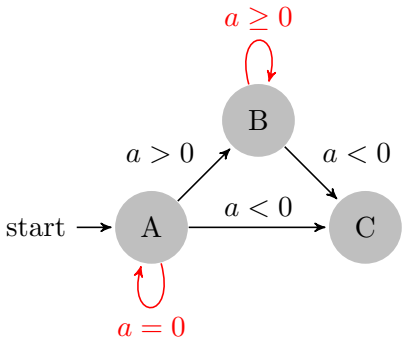

Abb. 3.3: Korrigierter Zustandsgraph

gespeichert. Anhand des Zustands kann selbstverständlich genauso wie bei der Nutzung von Semaphoren entschieden werden, ob ein Variablenzugriff erlaubt ist oder nicht.

State Machines bieten aber noch weitere Vorzüge, die deutlich werden, wenn man einen Blick auf die Abbildung 3.2 wirft. Das Beispiel mag unsinnig erscheinen, aber die Stärken lassen sich trotzdem gut erkennen. Die erste offensichtlichste ist die Möglichkeit der grafischen Darstellung und der damit erkennbaren Ablaufreihenfolge. Dazu gesellt sich noch das Aufzeigen von Zwischenzuständen (z.B. Warten) und das einfache Erkennen von fehlenden, fehlerhaften oder widersprüchlichen Übergängen bzw. Zuständen. In dem gezeichneten Zustandsgraphen sind z.B. zwei Unklarheiten eingebaut, die auch ohne geschultem Blick ausfindig gemacht werden können. Zur Kontrolle werfe man einen Blick in Abbildung 3.3.

Je nach Programmiersprache bieten sich für die Umsetzung eines Zustandsgraphen verschiedene Kontrollstrukturen. Die allgemein bekannteste Struktur dürfte eine IF-ELSE-Struktur darstellen. Für wenige Zustände mag diese auch genügen, jedoch kann man an obigem Beispiel mit nur drei Zuständen und auch nur drei Übergängen bereits feststellen, dass wenige Zustandsänderungen eine Rarität sind. Würde man diesen Graphen in einer

IF-ELSE-Struktur erfassen, stößt man sehr schnell an die Grenzen der Lesbarkeit. Eine wesentlich eleganterer Weg ist die Nutzung der SWITCH-CASE-Struktur, denn bei dieser ist eine lesbare Darstellung auch bei vielen möglichen Zuständen gewährleistet. Sie wartet aber noch mit einem weiteren Vorteil auf. Denn in einer SWITCH-CASE-Struktur kann nur eine Variable behandelt werden, während man bei der Verwendung einer IF-ELSE-Struktur geneigt sein kann mehrere Variablen einzuführen und zu behandeln. Dadurch steigert man aber lediglich die Unleserlichkeit des Codes.

```
switch (var) {
    case (a):
        do_sth();
        var = b;
        break;
    case (b):
        do_sth_else();
        var = a;
        break;
    default:
        wait();
        break;
}
```

Listing 3.3: switch-case Konstruktion

```
if(var == a) {
    do_sth();
    var = b;
} else {
    if(var == b) {
        do_sth_else();
        var = a;
    } else {
        wait();
    }
}
```

Listing 3.4: if-else Konstruktion

In den Listings 3.3 und 3.4 wurde versucht die eben aufgeführten Gründe zu unterstreichen, aber in ihnen ist noch ein entscheidender Punkt zu finden. In beiden Fällen wird die Zustandsvariable (var) innerhalb der jeweiligen Schleife verändert und **nicht** innerhalb einer aufgerufenen Funktion! Das erhöht die Verständlichkeit des Codes wesentlich.

Es ist nicht nötig die bisher geschilderten Konzepte komplett verinnerlicht zu haben, um die in den folgenden Abschnitten erläuterten Funktionen benutzen zu können, für ein leichteres Verständnis des Quelltextes empfiehlt es sich aber.

3.3 Ein- und Ausgänge

Wie im Abschnitt 3.1 des aktuellen Kapitels bereits angedeutet wurde, behandelt nun dieser Abschnitt die genauere Konfiguration der Ein- und Ausgänge. Welche Dateien dieses Modul in Anspruch nimmt, ist Abbildung 3.4 zu entnehmen. Die *Header* Datei basic.h wird wie auch die Datei setup.h von allen Modulen eingebunden, denn in diesen stehen grundlegende Vorschriften sowie die Information, ob das Modul eingebunden werden soll oder nicht. In den anderen beiden Dateien stehen dann neben den Funktionsprototypen noch modulspezifischere Angaben.
In io.h sind unter anderem die Adressen der einzelnen Register definiert, die zum Ansteu-

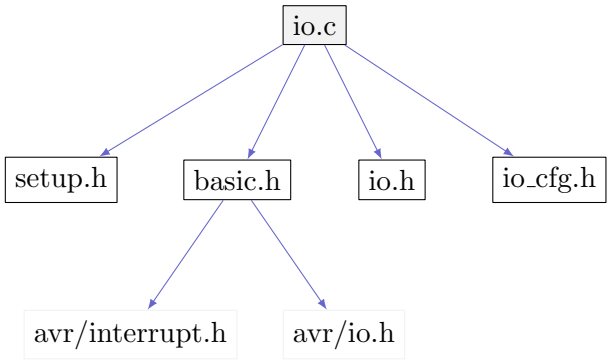

Abb. 3.4: Modulbeziehungen Input/Output

ern eines Pins vonnöten sind, sowie die Struktur PIN_CFG_STRUCT (siehe Listing 3.5) um den Zustand eines Pins zu speichern.

```
typedef struct {
  volatile U8 *DDRx;   //data direction register
  volatile U8 *PORTx;  //port register
  volatile U8 *PINx;   //port input register
  U8 DDxy;             //bit mask of corresponding
                       //pin index; e.g. 0x20 -> A5: PORTA, PIN5
  U8 io;               //input or output
  U8 value;            //allowed values: LOW or HIGH,
                       //INPUT_PULLUP or TRISTATE
} PIN_CFG_struct;
```

Listing 3.5: Pinstruktur in io.h

Der Inhalt von io_cfg.h ist dann projektspezifischer und besteht im Wesentlichen aus *Defines* denen eine Pinnummer zugewiesen wird. Ein kurzer Ausschnitt ist in Listing 3.6 dargestellt. Nimmt man z.B. an, dass bei einer Platine die LEDs grün und rot vertauscht wurden, so kann dieser Fehler einfach dadurch behoben werden, indem man dem *Define* anstelle der Zahl 3 die Zahl 4 zuweist und umgekehrt. Änderungen am Treiber (io.h, io.c) sind somit nicht erforderlich.

```
//Configurated OUTPUTS:
...
#define BOARD_LED_GREEN   3
#define BOARD_LED_RED     4
#define SD_ENABLE        34
...
```

Listing 3.6: Pinbenennung in io_cfg.h

Die vom Modul gestellten Funktionen beschränken sich im Grunde genommen komplett darauf die Pins zu konfigurieren bzw. deren Zustände zu ändern. Dazu werden die folgenden Funktionen bereit gestellt.

void IO_Init(void) Initialisiert die einzelnen Pins je nach Konfiguration als Ein- bzw. Ausgang. Ausgänge können entweder einen *high*-Pegel oder einen *low*-Pegel propagieren und Eingänge können entweder mit einem *pullup*-Widerstand beschalten oder hochohmig geschalten werden. Letzteres wird auch als *tri-state* Schaltung bezeichnet, weil es sich neben den zwei Zuständen 0 und 1 um einen dritten handelt. Die nebenstehende Abbildung 3.5 fasst die Einstellungsmöglichkeiten noch einmal zusammen.

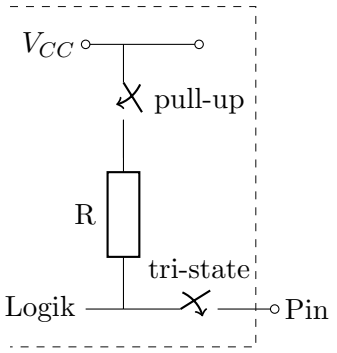

Abb. 3.5: Pinaufbau

void IO_SetPin(U8 channel, U8 level) Setzt den Ausgang eines als Ausgang konfigurierten Pins (channel) auf *high* oder *low* (level).

void IO_TogglePin(U8 channel) Verändert den Pinausgang des gewählten Pins (channel) auf *high*, wenn dieser vorher *low* war und umgekehrt.

void IO_SetPullup(U8 channel, U8 pullup)
Verhält sich analog zu IO_SetPin() nur ändert sich hier das Verhalten des Eingangs. Die möglichen Optionen sind die schon erwähnte *pullup*-Widerstand Beschaltung oder die *tri-state* Schaltung.

U8 IO_GetPin(U8 channel) Zum Auslesen eines anstehenden Pegels ist diese Funktion gedacht. Sieht der Eingang eine Spannung in der Nähe der Versorgungsspannung, so liefert die Funktion eine 1 zurück und befindet sich die Spannung auf Masse, so liefert sie eine 0 zurück.

3.4 Echtzeituhr

Von den einfachen Ein- und Ausgängen geht es nun weiter mit der komplexeren Echtzeituhr (*real time clock*). Prinzipiell erwartet man von einer Uhr zwei Dinge. Man muss sie stellen und lesen können. Bei der Echtzeituhr ist das nicht viel anders, nur bietet sie neben der Zeit noch das Datum. Auch für dieses Modul gibt es einen Abhängigkeitsgraphen, der in Abbildung 3.6 zu sehen ist, und dem entnommen werden kann, dass die Echtzeituhr über den TWI-Bus angeschlossen ist.

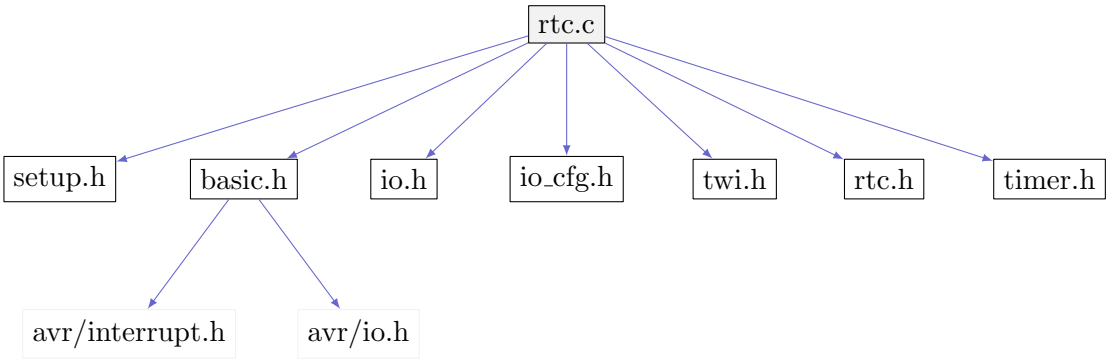

Abb. 3.6: Modulbeziehungen RTC

Die Aufgabe des Moduls besteht in nichts anderem als einen Zugriff auf die Uhr zu ermöglichen. Die Kommunikation selbst wird durch den TWI-Treiber ermöglicht. Somit ist die Aufgabe des RTC-Treibers die Daten für die Uhr aufzubereiten bzw. die Daten von der Uhr nachzubereiten. Wie schon erwähnt speichert die Echtzeituhr neben der Uhrzeit auch das Datum. Es bot sich deshalb an, die Variablen für die Uhrzeit und für das Datum in einer Struktur zu bündeln. Diese findet sich in rtc.h und ist in Listing 3.7 dargestellt.
In rtc.h wird außerdem die Variablenstruktur RTC_CurrentDate veröffentlicht, die die gerade genannte Struktur zu grunde liegen hat. Möchte man ein neues Datum (damit ist auch eine neue Uhrzeit gemeint) setzen, so müssen die einzelnen Variablen in der Struktur mit den jeweiligen Werten beschrieben werden. Von der Uhr wird allerdings vorausgesetzt, dass die an sie gesendeten Daten im BCD-Format vorliegen. Hinter BCD versteckt sich *binary coded decimals* und damit ist gemeint, dass z.B. die Zahl 47 nicht als 0x2F gespeichert wird, sondern als 0x47. In Tabelle 3.2 ist die Abbildungsvorschrift nochmals aufgeführt. Die Konvertierung vom Binärformat ins BCD-Format und umgekehrt wird innerhalb der Funktionen RTC_SetNewDate() und RTC_UpdateDate() ausgeführt. Damit besteht die einzige Aufgabe darin beim Setzen der Uhr darauf zu achten, dass die Zeit richtig in die Struktur geschrieben wird, und beim Lesen besteht sie darin das gelesene Datum möglichst zeitnah weiterzuverarbeiten.

	binär	**BCD**
Dec	Hex	Hex
0	0x00	0x00
1	0x01	0x01
2	0x02	0x02
⋮	⋮	⋮
9	0x09	0x09
10	0x0A	0x10
11	0x0B	0x11
⋮	⋮	⋮
15	0x0F	0x15
16	0x10	0x16
17	0x11	0x17
⋮	⋮	⋮
58	0x3A	0x58
59	0x3B	0x59

Tab. 3.2: Binary Coded Decimals - Tabelle

```
typedef struct {
  U8 seconds;          //seconds 00..59
  U8 minutes;          //minutes 00..59
  U8 hours;            //hours 00..23
  U8 day;              //day can be freely set, e.g. 0=sunday,
                       //1=monday, .., 6=saturday
  U8 date;             //day of the month 00..31
  U8 month;            //month 00..12
  U8 year;             //year 00..99
} RTC_date_struct;
```

Listing 3.7: Datumsstruktur

Theoretisch und praktisch ist es möglich die Zeit sekündlich zu aktualisieren. Dies führt jedoch zu einem dem Nutzen nicht gerechten Aufwand und zur unnötigen Auslastung des Mikrocontrollers. Für die Uhrzeit sollte deshalb auf den Systemtimer zurückgegriffen werden. Dabei handelt es sich um die öffentliche Variablenstruktur SYSTIMER, deren Struktur dem Listing 3.8 entnommen werden kann und die wiederum in der Datei timer.h beheimatet ist.

```
typedef struct{
    U8 time_10ms;      //10ms timer, goes from 0 to 100
    U8 time_1s;        //1s timer, goes from 0 to 59
    U8 time_1min;      //1min timer, goes from 0 to 59
    U8 time_1h;        //1h timer, goes from 0 to 23
} struct_systimer;
```

Listing 3.8: Systemtimer-Struktur

U8 RTC_Init(void) Diese Funktion wird i.d.R. beim Hochfahren des Mikrocontrollers ausgeführt. Sie liest die Zeit von der Echtzeituhr und überprüft, ob die Uhrzeit aktuell ist oder nicht. Das Entscheidungskriterium bildet die Jahreszahl, denn bei einem ungesetzten Datum, ist das Jahr auf 0 gesetzt. Wurde das Jahr 0 ausgelesen, wird RTC_FAILED zurückgeliefert, während sonst RTC_SUCCESS zurückgeliefert wird.

U8 RTC_SetNewDate(void) Um ein neues Datum zu setzen, bedient man sich dieser Funktion. Das vor dem Aufruf in die Variablenstruktur RTC_CURRENTDATE eingetragene Datum wird ins BCD-Format konvertiert und anschließend an die Uhr übertragen. Zu beachten ist, dass zu keinen Zeitpunkt überprüft wird, ob die Werte sinnvoll sind bzw. sich im erlaubten Zahlenbereich bewegen. Bevor die Konvertierung jedoch überhaupt eingeleitet wird, wird nachgesehen, ob der Bus überhaupt zur Verfügung steht. Ist dies nicht der Fall, so liefert die Funktion RTC_FAILED zurück während sie im Falle eines gestatteten Buszugriffs die Umwandlung durchführt, die Übertragung einleitet und RTC_SUCCESS zurückliefert.

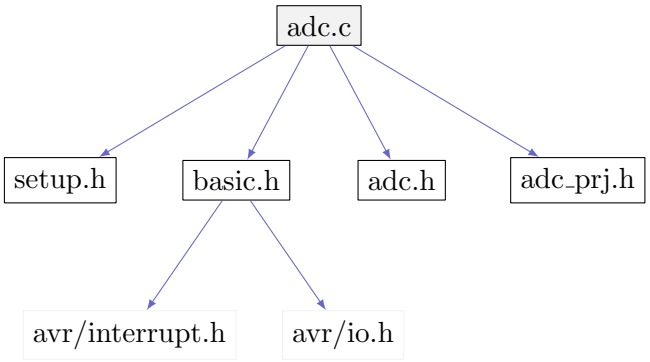

Abb. 3.7: Modulbeziehungen ADC

U8 RTC_UpdateDate(void) Diese Funktion bildet das Gegenstück zur gerade erklärten. Sie holt die aktuelle Uhrzeit ab und schreibt sie, nach der Konvertierung vom BCD-Format ins binäre, in die RTC_CurrentDate-Struktur. Für den Fall das eine Abfrage erfolgreich angestoßen wurde, liefert die Funktion RTC_UPDATE_STARTED zurück. Ist bereits ein Abfrage im Gange, liefert sie RTC_STILL_UPDATING und wenn keine Abfrage losgetreten werden konnte, wird dies durch die Rückgabe von RTC_TWI_BUSY kenntlich gemacht. Um nun das Ende einer Abfrage feststellen zu können, muss man sich der Funktion RTC_IsBusy() bedienen.

U8 RTC_IsBusy(void) Je nachdem ob noch eine Operation ausgeführt wird oder nicht, gibt die Funktion RTC_FAILED bzw. RTC_SUCCESS zurück. Diese Funktion wurde entworfen, um feststellen zu können wann eine Operation beendet ist. Denn wie bereits zu Beginn des Kapitels erwähnt wurde, ist die Kommunikation interruptgesteuert und somit trifft das Ende der Funktion keinerlei Aussage über das Ende der Kommunikation.

U8 RTC_Halt(void) Hiermit kann die Echtzeituhr ausgeschaltet werden, um z.B. Strom zu sparen. Da die Uhr angehalten wird, muss beim Wiedereinschalten logischerweise die Uhrzeit neu gesetzt werden. Wenn der Ausschaltbefehl gesendet werden konnte, wird dies durch den Rückgabewert RTC_SUCCESS kenntlich gemacht. Konnte er nicht übertragen werden, erhält man stattdessen RTC_FAILED.

3.5 Analog-Digital Konverter

Der Analog-Digital-Umsetzer, auch Analog-Digital-Wandler genannt (*Analog-to-Digital Converter*), bildet gemäß seiner Funktion die Schnittstelle zwischen den physikalischen Werten und den leichter weiterverarbeitbaren Digitalwerten. Der verwendete Mikrocontroller AT90CAN128 verfügt über einen internen Wandler. Dementsprechend einfach ist der Abhängigkeitsbaum in Abbildung 3.7 gehalten. Die Trennung zwischen dem zu-

grunde liegenden Treiber und der projektspezifischen Anwendung wurde bereits in Abschnitt 3.1 angerissen. In Abbildung 3.7 ist die Verknüpfung zu erahnen, weil der *Header* adc_prj.h mit eingebunden wird. Dieser enthält die anwendungsspezifischen Einstellungen wie die zu verwendende Abtastfrequenz, die Referenzspannungsquelle und kündigt die für Anwendung benötigten Funktionen an.

Zum Einstellen der Frequenz bedient man sich eines Vorteilers (*Prescaler*), der den Systemtakt für die ADC Komponenten verringert. Bei einer gegebenen Taktrate von 16 MHz ergibt sich für den Vorteiler, unter Berücksichtigung der 10 Bit Auflösung, die nur bis zu einer maximalen Abtastfrequenz von 200 kHz garantiert wird[2, S. 276], und einer schnellstmöglichen Konvertierung, nur ein vernünftiger Wert von 128 der zu einer Abtastfrequenz von 125 kHz führt. Im laufenden Betrieb benötigt eine Wandlung 13 Takte, wobei die anliegende Spannung nach einem Takt mittels eines Abtast-Halte-Gliedes (*Sample&Hold-Circuit*) festgehalten wird. Die festgehaltene Spannung wird dann in den restlichen Takten gewandelt, d.h. selbst wenn sich die eigentlich zu messende Spannung im Anschluss schlagartig ändert, hat dies keinen Einfluss auf die laufende Konvertierung. Damit der ADC weiß wozu er die abzutastende Spannung in Bezug zu setzen hat, muss ihm eine Referenzspannung genannt werden. Zur Wahl stehen die interne Spannung von 2,56 V oder eine externe Spannung. Beim vorliegenden Board liegt das externe Spannungsniveau auf 4,096 V, d.h. ein Bit entspricht 4 mV. In der Software ist die externe Bezugsquelle standardmäßig angewählt.

Je nach Registereinstellungen kann einer von insgesamt acht möglichen Controller-Pins mit dem ADC verbunden werden, wie nebenstehende Abbildung 3.8 zeigt. Der Begriff MUX steht dabei für *Multiplexer* und bedeutet so viel wie Mehrfachkoppler. Es kann somit immer nur eine Spannung nach der anderen konvertiert werden. Dabei ist außerdem zu beachten, dass die Pins vier bis sieben für die Programmierschnittstelle JTAG genutzt werden und daher keine äußere Beschaltung aufweisen. Den Pins null bis drei ist hingegen ein 10 kΩ Widerstand und ein 47 nF Kondensator parallel geschalten.

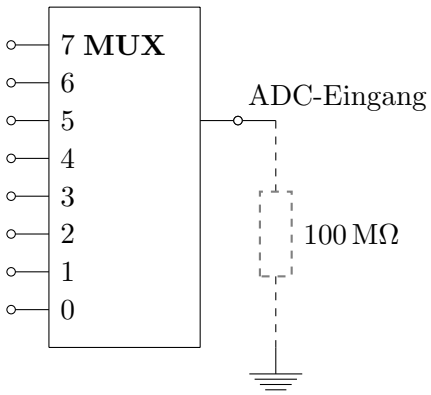

Abb. 3.8: Verdrahtung

Es folgt eine Beschreibung der einzelnen Treiberfunktionen.

void ADC_Init(void) Wählt die Referenzspannungsquelle aus und stellt die abzutastenden Kanäle sowie deren Reihenfolge ein, ehe im Anschluss die Interrupts scharf geschaltet werden.

U8 ADC_AddChannel(U8 channel) Fügt den übergebenen Kanal der Kanalliste hinzu und liefert für den Fall, dass die Liste bereits voll ist ADC_FULL_QUEUE zurück. Der Erfolgsfall wird durch Rückgabe von ADC_SUCCESS kenntlich gemacht.

U8 ADC_DelChannel(U8 channel) Entfernt den angegebenen Kanal wieder aus der Liste wobei nur der erste gefundene Eintrag gelöscht wird, wie in der Tabelle 3.3 verdeutlicht ist. Für den Fall, dass ein Eintrag gefunden wurde, liefert die Funktion ADC_SUCCESS zurück und wenn der Kanal nicht in der Liste gefunden wurde, liefert sie ADC_FAILED zurück. Es sei nochmals darauf hingewiesen, dass nur ein Eintrag entfernt wird. Ist der Kanal mehrmals in der Liste aufgeführt, so muss die Funktion mehrmals aufgerufen werden, um alle Vorkommnisse zu entfernen.

Pos.	Kanal	
	vorher	nachher
0	1	3
1	3	1
2	1	-
3	-	-

Tab. 3.3: Kanallöschung

U8 ADC_Trigger(void) Diese Funktion bildet die Schnittstelle zu den projektspezifischen Funktionen bzw. wird von diesen aufgerufen. Ein konkreter Anwendungsfall wird in Abschnitt 4.2 behandelt. So wird dort vor der Konvertierung ein Pin auf *high* gesetzt, damit ein Kondensator aufgeladen wird und dessen Ladespannung wird zu einem definierten Zeitpunkt abgetastet. Sinnigerweise muss für ein verwertbares Ergebnis ein Rädchen ins andere greifen, d.h. die Konvertierung kann erst angestoßen werden nachdem der Pin gesetzt wurde und für das Anstoßen ist diese Funktion gedacht. Sie hat drei mögliche Rückgabewerte: ADC_SUCCESS wenn alles geklappt hat und die Konvertierung der Kanalliste angestoßen werden konnte, ADC_FAILED wenn der ADC bereits in Verwendung ist und ADC_EMPTY_QUEUE wenn keine Kanäle ausgewählt wurden.

U8 ADC_IsBusy(void) Beantwortet die im Funktionsnamen gestellte Frage, ob der ADC gerade arbeitet oder nicht. Wird die Frage bejaht liefert sie ADC_TRUE und wird sie verneint liefert sie ADC_FALSE zurück.

U8 ADC_GetState(void) Prinzipiell genügt der Aufruf von ADC_IsBusy(void) denn diese liefert bereits die Information, ob der ADC in Verwendung ist (ADC_BUSY) oder nicht (ADC_READY). Über einen weiteren möglichen Zustand kann damit jedoch keine Aussage getroffen werden. Ist dies von Interesse, so muss man sich dieser Funktion bedienen. Neben den schon erwähnten Rückgabewerten ADC_READY und ADC_BUSY gibt es noch den Zustand ADC_QUEUE_FINISHED. Mit diesem Zwischenzustand wird der Anwendungsebene signalisiert, dass die Kanalliste abgearbeitet wurde, alle Kanäle konvertiert wurden und zur weiteren Verarbeitung zur Verfügung stehen. Die Anwendung hat jetzt die Aufgabe die Daten zu sichern bzw. im generellen weiter zu verarbeiten und sich dann im Anschluss der nachfolgenden Funktion zu bedienen, um den ADC wieder freizugeben.

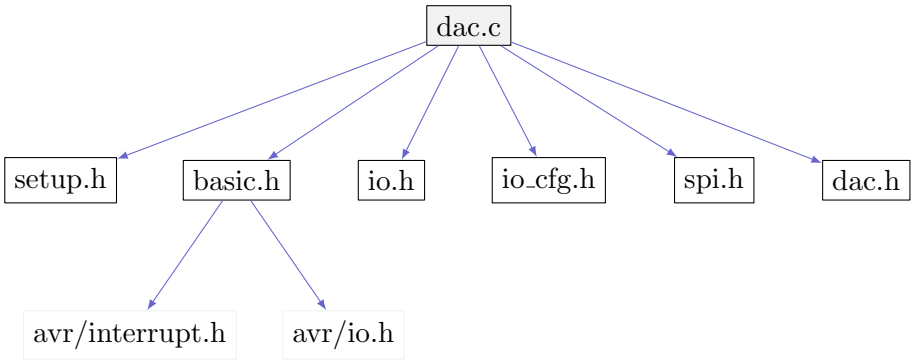

Abb. 3.9: Modulbeziehungen DAC

U8 ADC_SetState(U8 state) Um den Zwischenzustand ADC_QUEUE_FINISHED wieder zu verlassen, ist es nötig diese Funktion aufzurufen, wobei der zu übergebende Status ADC_READY zu lauten hat, da nur dieser den ADC freigibt. Selbstverständlich könnte die Funktion aber genauso benutzt werden, um einen anderen Zustand zu erzwingen.

3.6 Digital-Analog Konverter

Während mit dem Analog-Digital-Konverter physikalische Spannungen in diskrete Werte konvertiert werden, erfüllt ein Digital-Analog-Konverter (*Digital-to-Analog Converter* (DAC)) den gegenteiligen Zweck, die Wandlung eines Digitalwertes in eine Spannung. Ursprünglich war angedacht einen 16-Bit DAC einzusetzen. Aus finanziellen Gründen hat man sich aber dazu entschlossen das baugleiche, kleinere 12-Bit Modell einzusetzen. Diesen könnte man aber problemlos durch die 16-Bit Version ersetzen, da die entworfene Software mit beiden Varianten umgehen kann.

Die Abhängigkeiten sind in Abbildung 3.9 aufgeschlüsselt und es ist zu sehen, dass der DAC über SPI angeschlossen ist. Somit beschränkt sich die Aufgabe dieses Moduls darauf die Daten für den DAC aufzubereiten und an den SPI-Treiber weiterzureichen. Der gewählte DAC hat neben seinen vier Ausgängen, der Versorgungsspannung, den SPI-Leitungen und dem Referenzspannungseingang noch zwei Pins zur Adressierung (A1 und A0). Diese erlauben es bis zu vier DACs an einer gemeinsamen *Chip Select*-Leitung anzuschließen. In Abbildung 3.10 ist dies beispielhaft dargestellt, wobei auf die *Chip Select*-Leitung der Übersichtlichkeit wegen verzichtet wurde. Anstelle die Pins A1 und A0 frei konfigurierbar mit dem Mikrocontroller zu verbinden, können diese auch hart verdrahtet werden, d.h. die Pins A1 und A0 werden direkt mit Masse bzw. der Versorgungsspannung verbunden. Die Pinkonfiguration für A1 und A0 findet sich dann in der Befehlstruktur wieder, die in Tabelle 3.4 zu sehen ist. Der Konverter vergleicht die Bits A1 und A0 mit den Pegeln an den Pins A1 und A0 woraufhin er entscheidet, ob das

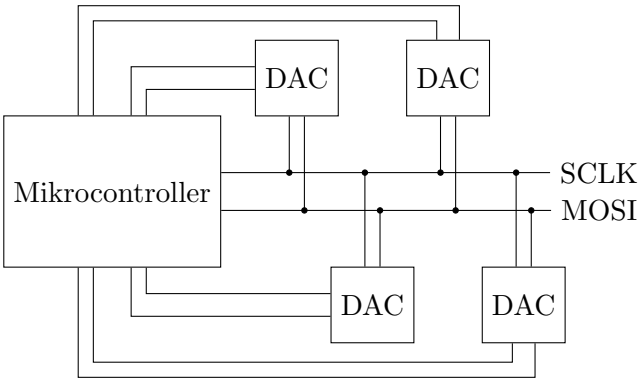

Abb. 3.10: Vier DACs am SPI

DB23	DB22	DB21	DB20	DB19	DB18	DB17	...
A1	A0	LD 1	LD 0	0	DAC Sel 1	DAC Sel 0	

...	DB16	DB15	DB14	DB13-DB4	DB3-DB0
	PD0	MSB	MSB-1	MSB-2...LSB	Don't Care

Tab. 3.4: Befehlsstruktur DAC

Kommando an ihn gerichtet ist oder nicht. Falls ja, werden die weiteren Bits ausgewertet in denen unter anderem steht, um was für einen Befehl es sich handelt (LD 0/1), welcher DAC-Ausgang gemeint ist (DAC Sel 0/1)und ob der Ausgang ausgeschaltet werden soll (PD0 (*Power Down*)) oder nicht.
Dem Befehlsbyte schließen sich noch zwei Wertebytes (16 Bit) an von denen nur 12 Bit ausgewertet werden, schließlich handelt es sich auch um einen 12 Bit DAC. Beim 16 Bit DAC würden logischerweise alle 16 Bit ausgewertet werden. Bei der Treiberimplementierung ergab sich nun anfangs das Problem, dass die Reihenfolge der übertragenen Bytes nicht der gewünschten entsprach. Dazu muss man wissen, dass es zwei Möglichkeiten gibt wie Daten im Speicher abgelegt werden.

big endian speichert das Byte mit den höchstwertigen Bits (d.h. die signifikantesten Stellen) an der kleinsten Speicheradresse. Ein anschauliches Beispiel bietet die Schreibweise der Uhrzeit: Stunde:Minute:Sekunde.

little endian ist das Gegenstück zu *big endian*, d.h. das Byte mit den niederwertigstens Bits wird zuerst gespeichert und nimmt die kleinste Speicheradresse ein. Ein äquivalentes Beispiel hierfür ist die Datumsschreibweise: Tag.Monat.Jahr.

Die Mikrocontroller von Atmel speichern im *little endian* Format, d.h. eine 32 Bit Variable vom Typ Integer mit dem Wert 305.419.896 (0x12345678) wird byteweise von hinten nach vorne im Speicher abgelegt. Wird die Zahl fälschlicherweise von vorne

nach hinten gelesen, so hat diese einen völlig anderen Wert (2.018.915.346). Controllerintern kann so etwas nicht passieren, aber bei der Übergabe von Werten sollte dies beachtet und als mögliche Fehlerquelle angesehen werden.
Der SPI-Treiber wird vereinfacht gesprochen mit zwei Parametern bedient. Der eine Parameter ist die Speicheradresse an der die Daten liegen und der Zweite ist die Anzahl der Bytes, die dort liegen. Letzteres ist bei den DAC-Befehlen folglich immer drei. Das Befehlsbyte und die Wertebytes (16 Bit = 2 Bytes) folgen im Speicher zwar direkt aufeinander, jedoch wird durch die Speichervorschrift *little endian* zuerst das niederwertige Byte abgespeichert und dann das höherwertige abgespeichert. Die Wertebytes müssen deshalb vorher umgespeichert werden. Im DAC-Treiber ist dies selbstverständlich bereits implementiert.

Der DAC-Treiber bietet zwei Schnittstellen über die das Verhalten des DACs beeinflusst werden kann. Für beide muss das gewünschte Verhalten in der Variablenstruktur DAC_OutputValuesStructure, deren Aufbau Listing 3.9 entnommen werden kann, eingestellt werden. Während in den 16 Bit Variablen der Ausgangswert abgelegt wird, wird mittels der 8 Bit Variable OPERATION das Verhalten kontrolliert. Je nach gewählter Schnittstelle muss der zuletzt erwähnten Variable Aufmerksamkeit geschenkt werden oder nicht. Eine genauere Betrachtung folgt in der Beschreibung der Funktionen.

	Speicheradresse			
	0x00	0x01	0x02	0x03
big endian	0x12	0x34	0x56	0x78
little endian	0x78	0x56	0x34	0x12

Tab. 3.5: Endianess

```
typedef struct {
  U8 operation; //control byte for all DAC channels
  U16 valueA;   //channel A's output value
  U16 valueB;   //channel B's output value
  U16 valueC;   //channel C's output value
  U16 valueD;   //channel D's output value
} DAC_short_operation_struct;
```

Listing 3.9: Konfigurationsstruktur

void DAC_Init(void) Initialisiert den SPI-Treiber und anschließend den DAC. Dabei wird im Grunde genommen nur die Taktrate des SPI-Busses festgelegt und die in dac.h selektierte Referenzspannungsquelle ausgewählt. Zur Wahl steht die interne Referenzspannung und eine externe, wobei bei ersterer noch unterschieden werden kann, ob diese durchgehend eingeschaltet sein soll, also selbst wenn die Ausgänge des DACs heruntergefahren sind, oder nicht. Standardmäßig ist die externe Referenzspannungsquelle selektiert, denn sie bietet einen Spannungsbereich bis 4,096 V während die interne nur

maximal 2,5 V bietet.

$$\frac{x\ V}{Referenzspannung\ [V]} = \frac{z\ Bit}{2^{12}\ Bit} \tag{3.1}$$

Mit diesen Informationen und dem Wissen, dass es sich um einen 12 Bit DAC handelt, kann nun der in Gleichung (3.1) angegebene Dreisatz aufgestellt werden. Je nachdem wonach gefragt ist, muss die Gleichung umgestellt werden. Ist z.B. nach der zu einer Digitalzahl korrespondierenden Spannung gesucht, verändert sich die Gleichung zu (3.2)

$$x\ V = \frac{z\ Bit}{2^{12}\ Bit} \cdot Referenzspannung\ [V] \tag{3.2}$$

und wird stattdessen nach der zu einer Spannung gehörigen Digitalzahl verlangt, ändert sich die Gleichung zu (3.3) ab.

$$z\ Bit = \frac{x\ V}{Referenzspannung\ [V]} \cdot 2^{12}\ Bit \tag{3.3}$$

U8 DAC_ChangeOutput(void) Verwendet man diese Funktion ist es erforderlich das bereits erwähnte Konfigurationsbyte genauer zu betrachten. Der DAC hat vier Ausgänge und jeder Ausgang hat wiederum vier verschiedene Konfigurationsmöglichkeiten. Zum Speichern dieser vier Möglichkeiten genügen 2 Bit womit bei vier Ausgängen insgesamt ein Platzbedarf von 8 Bit anfällt, was einem Byte entspricht.

Funktion	Bit 1	Bit 0
ignorieren	0	0
speichern	0	1
ausgeben	1	0
synchrone Ausgabe	1	1

Tab. 3.6: Befehlsbyte

In der nebenstehenden Tabelle 3.6 sind die vier Konfigurationsmöglichkeiten für die einzelnen Kanäle aufgelistet. Um die einzelnen Möglichkeiten leichter zu erfassen, wurden einige Beispielkonfigurationen in Abbildung 3.11 angefertigt. Dabei ist ein *low*-Pegel als ausgeschalteter und eine *high*-Pegel als ein eingeschalteter Ausgang zu verstehen. Über den jeweiligen Spannungswert wird dabei keine Aussage getroffen. Die Kanäle können mit einer Einschränkung nahezu beliebig kombiniert werden. Die Einschränkung wird in Abbilung 3.11b am ersichtlichsten. Die ersten beiden Bit entsprechen dem Kommando 'synchrone Ausgabe', das eventuell im temporären Zwischenregister befindliche Werte ins Hauptregister laden lässt, womit sich der Kanalausgang ändert. Gleichzeitig wird der Ausgang des jeweiligen Kanals, hier Kanal 1, gesetzt. Die Einschränkung besteht nun darin, dass alle nachfolgenden Kommandos ignoriert werden. Stände das Kommando 'synchrone Ausgabe' an letzter Stelle und würde in den anderen drei Kanälen 'speichern' stehen, was folgende Bitfolge ergäbe 01 01 01 11, dann entspräche das dem Verhalten der Funktion DAC_CyclicSimultaneousOutput().
Neben der eben genannten Einschränkung gibt es noch zwei Spezialaufrufe. Ist das Byte komplett mit Nullen gefüllt, wie in Abbildung 3.11e zu sehen ist, dann werden alle DAC-Kanäle abgeschaltet sofern sie das nicht schon sind. Füllt man das Byte hingegen mit lauter Einsen, wie Abbildung 3.11f zeigt, entspricht dies einem *Broadcast*, d.h. alle Kanäle aller DACs fühlen sich angesprochen und ändern ihren Ausgang.

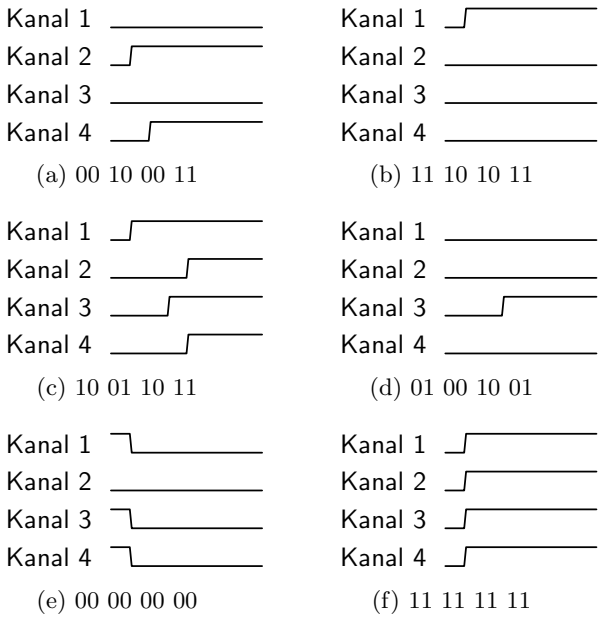

Abb. 3.11: Konfigurationsbeispiele

U8 DAC_CyclicSimultaneousOutput(void) Wenn es nicht nötig ist die einzelnen DAC-Kanäle für sich anzusteuern, bietet sich diese Funktion an. Wie in der vorhergehenden Funktion schon erwähnt, werden die DAC-Ausgänge gleichzeitig geändert. Auch hier wird zurückgegeben, ob die Operation durchgeführt werden konnte (DAC_SUCCESS) oder nicht (DAC_SPI_BUSY).

3.7 SD-Karte

Das SD-Karten-Modul ist als das komplexeste anzusehen. Denn im Gegensatz zu allen anderen Modulen findet hier zum einen Kommunikation in beide Richtungen statt und zum anderen weißt es die meisten Abhängigkeiten auf wie Abbildung 3.12 zu entnehmen ist.

Da der Mikrocontroller nicht gerade mit Arbeitsspeicher gesegnet ist, wurde auf die Implementierung eines allgemein gebräuchlichen Dateisystems verzichtet. Das ursprünglich angedachte Dateisystem FAT (*File Allocation Table*) wurde in anderen Projekten [7] [10] [11] als nicht gerade schlank dokumentiert und man entschloss sich deshalb ein eigenes Dateisystem zu implementieren. Es wurde als noFS bezeichnet, um anzuzeigen, dass es sich um kein allgemein bekanntes Dateisystem handelt. Der Vorteil des gewählten Ansatzes besteht im schlanken Code, jedoch wurde dieser mit einem unhandlichen Auslesen der Daten erkauft. Doch ehe darauf eingegangen wird, wird erst noch erklärt wie die Daten auf der SD-Karte abgelegt werden.

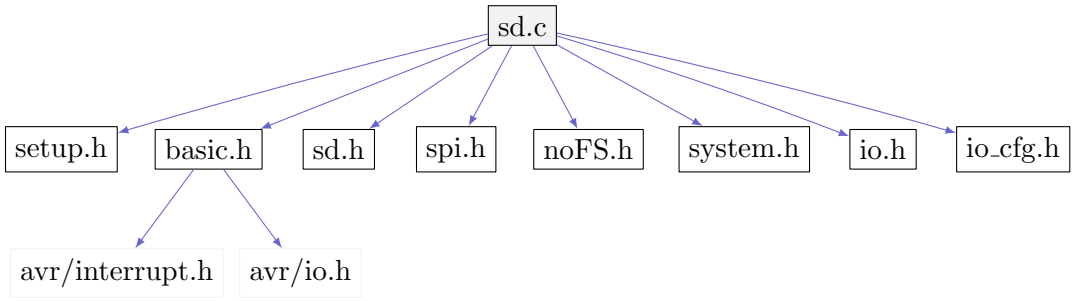

Abb. 3.12: Modulbeziehungen SD

Flashspeicher, wie SD-Karten welche sind, haben nur eine begrenzte Anzahl an Schreibzyklen. Diese Zahl ist mit 100.000 und mehr auf den ersten Blick zwar recht hoch, durch das falsche Beschreiben kann die maximale Anzahl an Schreibzyklen jedoch sehr schnell erreicht werden. Dazu muss man wissen, dass die SD-Karte in 512 Byte große Blöcke unterteilt ist und Operationen nur blockweise ausgeführt werden können. D.h. selbst wenn nur ein Byte verändert werden soll, muss der gesamte Block neugeschrieben werden. Würde nun ein kompletter Block durch einen fehlerhaften Algorithmus byteweise beschrieben werden, dann wäre dieser nach rund $\frac{100.000}{512} \cong 195$ Beschreibungen kaputt. Es macht somit Sinn den Puffer in der Größe zu wählen den die Blockgröße vorgibt.
Einzig um die Struktur der aufzuzeichnenden Daten muss sich jetzt noch gekümmert werden. Diese kann theoretisch frei gewählt werden, jedoch bietet es sich an, diese so zu wählen, dass die abzulegenden Daten in ihrem Umfang der Gleichung $2^k - 1 \forall k \in 2, 3 \ldots, 8$ genügen.
Die Daten werden nicht in ihrer binären Rohform auf die SD-Karte abgelegt, sondern vorher ins HexAscii-Format konvertiert. Dabei werden die einzelnen *Nibbles* (4 Bit) eines jeden Bytes als ein eigenes Byte gespeichert und je nach Wert um 48 (0x30) bzw. 65 (0x41) Stellen verschoben. Die binäre 3 (0x03) wird somit zur Ascii 3 (0x33) und die binäre 13 (0xD) wird zu D (0x44). In Abbildung 3.13 wurde versucht dies zu verdeutlichen. Mit der Umwandlung erklärt sich warum kein 512 Bytes großer Datensatz gewählt werden kann, denn die Konvertierung verdoppelt im Anschluss die Datenmenge. Die Reduktion der Datenmenge um 1 ist der Struktur auf der SD-Karte geschuldet. Um das Ende eines Datensatzes kenntlich zu machen, ist das letzte Byte für den Binärwert 3 (0x03) reserviert. Abbildung 3.15 zeigt die Anordnung der Daten auf der SD-Karte. Eine genauere Erklärung der Zusammenhänge folgt in den Funktionsbeschreibungen.

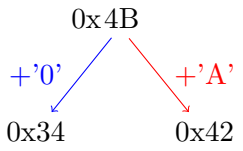

Abb. 3.13: HexAscii Konvertierung

Die explizite Struktur, also die Reihenfolge in der die einzelnen Daten abgelegt werden, wird in der Struktur DATA_LOG_STRUCT festgelegt, die wiederum in der Datei data_log.h zu finden ist. Wie bereits erwähnt, werden die Daten vor dem Speichern ins

HexAscii-Format konvertiert. Für die Konvertierung muss jedoch ebenfalls die Datenstruktur bekannt sein. Deswegen ziehen Änderungen an DATA_LOG_STRUCT Änderungen in NOFS_WRITE_DATA() nach sich. Die Funktion NOFS_WRITE_DATA() findet sich in noFS.c und bei Anpassungen muss lediglich auf den richtigen Datentyp geachtet werden.

U8 SD_InitCard(void) Bevor die SD-Karte überhaupt verwendet werden kann, muss sie erst einmal initialisiert werden. Dies wird von dieser Funktion übernommen. Dazu stößt sie die Initialisierung der darunter liegenden SPI-Schicht an und wenn diese erfolgreich initialisiert werden konnte, wird so gleich damit begonnen mit der SD-Karte genauso zu verfahren. Die Initialisierung der SD-Karte erfolgt dabei in mehreren Schritten. Da es in jedem Schritt zu einem Fehler kommen kann, bedarf es auch mehrerer möglicher Fehlerrückmeldungen.
Normalerweise wird eine SD-Karte im SD-Modus betrieben, damit sie über SPI angesprochen werden kann, muss sie vorher zurückgesetzt werden. Ist der Reset nicht erfolgreich wird mit SD_ERROR_CARD_COULD_NOT_BE_RESET abgebrochen. Im nächsten Schritt sollte die Karte in den SPI-Modus wechseln. Ist dies nicht der Fall wird dies mit der Rückgabe von SD_ERROR_CARD_DIDNT_ENTER_SPI_MODE zurückgemeldet. Wechselte die Karte jedoch in den SPI-Modus, so gilt sie als initialisiert und die Funktion liefert SD_SUCCESS zurück.

U8 SD_SetBlocklen(U16 length_in_bytes) Setzt die Blocklänge auf die übergebene Länge und liefert im Erfolgsfall SD_SUCCESS zurück. Wurde eine Blockgröße gewählt, die von der Karte nicht unterstützt wird, wird dies durch die Rückgabe von SD_ERROR_CARD_DIDNT_ACCEPT_BLOCKLENGTH kenntlich gemacht. Die gesetzte Blocklänge gilt sowohl für den Lese- als auch für den Schreibzugriff. Generell kann jedoch angenommen werden, dass SD-Karten nur mit 512 Byte große Blöcke beschrieben werden können. Gelesen können dafür nahezu beliebig große Blöcke.

U8 SD_ReadBytes(U32 address, U8 *input_buffer, U16 number_of_bytes) Mit dieser Funktion ist es möglich eine bestimmte Anzahl Bytes von der übergebenen 32-Bit Karten-Adresse an die übergebene Adresse (*input_buffer) schreiben zu lassen. Die Anzahl Bytes muss dafür vorher mittels SD_SetBlocklen(U16) eingestellt worden sein. Selbstverständlich muss sichergestellt sein, dass der Lesepuffer groß genug ist, um die gelesenen Daten aufnehmen zu können. Andernfalls ist kein zuverlässiger Betrieb zu erwarten.

U8 SD_BlockTransfer(U32 memoryaddress, U8 *dataptr) Wie erwähnt ist es zwar erlaubt nahezu beliebig große Blöcke zu lesen, das Schreiben ist jedoch auf 512 Byte große Blöcke beschränkt. Deswegen erwartet diese Funktion auch keine Angabe über die Größe, sondern nur Aussagen über Speicherbereiche. Von wo soll der Block gelesen werden (*dataptr) und an welche Adresse soll er geschrieben werden (memoryaddress). Neben einer erfolgreichen Übertragung (SD_SUCCESS) gibt es auch hier wieder mehrere Fehlerfälle. Der SD-Treiber kann z.B. noch mit einer vorher gestarteten Operation beschäftigt sein

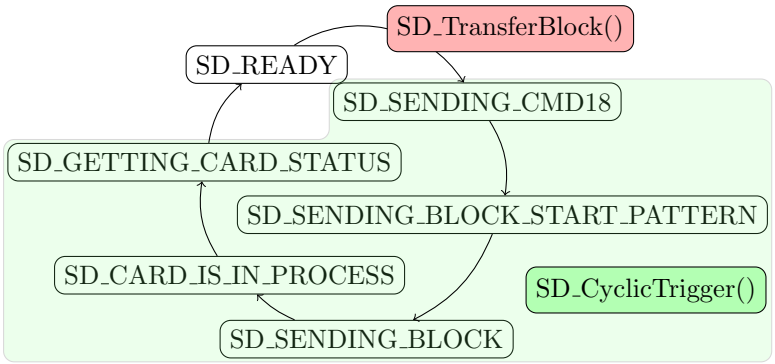

Abb. 3.14: Datentransfer zur SD-Karte

(SD_ERROR_STILL_BUSY) oder die SPI-Schnittstelle konnte nicht reserviert werden (SD_ERROR_COULDNT_ALLOCATE_SPI). Der letztmögliche erkennbare Fehler stammt von der SD-Karte selbst, wenn diese das Schreibkommando nicht verstanden hat. Zurückgegeben wird in diesem Fall SD_ERROR_CARD_DIDNT_UNDERSTAND_WRITEBLOCK_CMD.

void SD_CyclicTrigger(void) Für den automatischen Betrieb ist im Grunde genommen nur die SD_BlockTransfer() und diese Funktion von Interesse. Die anderen Funktionen werden hauptsächlich für die Fehlersuche benötigt. Während SD_BlockTransfer() den Datentransfer anstößt, wickelt SD_CyclicTrigger() ihn ab. Der Ablauf ist in der Abbildung 3.14 dargestellt. Zu Beginn ist der SD-Treiber bereit (SD_READY) und wartet darauf angestoßen zu werden. Das bewerkstelligt der Aufruf von SD_BlockTransfer(), der den Zustand der State-Machine zu SD_SENDING_CMD18 ändert. Hinter CMD18 verbirgt sich das Kommando mit dem man einen Block überträgt. Die Übertragung ist dabei in mehrere Phasen untergliedert. Zuerst wird CMD18 an die SD-Karte geschickt und auf die Antwort gewartet. Ist diese positiv wird das Startmuster übertragen und daran schließt sich die Übertragung des Datenblocks an. Im Anschluss muss gewartet werden bis die Daten auf die Karte geschrieben wurden. Ob die Schreiboperation erfolgreich war, wird im letzten Schritt ermittelt ehe die State-Machine wieder auf SD_READY gesetzt wird und eine Operation angestoßen werden. Selbstverständlich können hier und da Fehler auftreten, um die Grafik übersichtlich zu halten wurden diese jedoch nicht eingezeichnet. Für ein tieferes Verständnis des Treibers sei auf den Quellcode verwiesen.
Besonders bei den SD-Karten der Firma Transcend kam es immer wieder zu Fehlern. SD-Karten von Panasonic konnten jedoch fehlerfrei betrieben werden, weswegen die Verwendung von Panasonic Karten empfohlen wird.

Bisher wurde nur die Mikrostruktur genauer betrachtet. Was noch fehlt ist

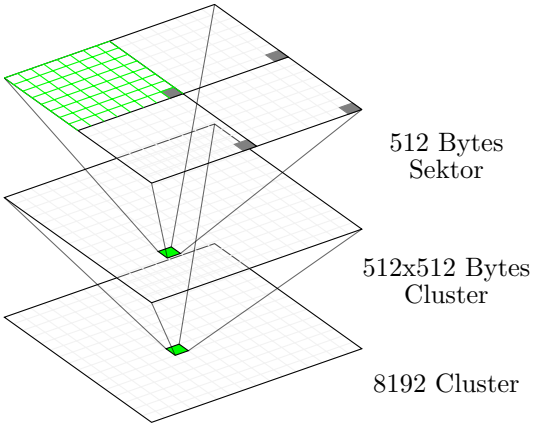

Abb. 3.15: Struktur auf der SD-Karte

die Makrostruktur. Diese wird in erster Linie vom Dateisystem bestimmt. Wie schon erwähnt wurde ein eigenes entwickelt, deren Struktur der nebenstehenden Abbildung 3.15 entnommen werden kann. Dem Bild liegt eine 2 GB große SD-Karte zugrunde, die Platz für $2 \cdot 2^{10} \cdot 2^{10} \cdot 2^{10} = 2.147.483.648$ Bytes hat. Ein Datenblock besteht wie schon erwähnt aus 512 Bytes und wird im folgenden, um Missverständnissen vorzubeugen, als Sektor bezeichnet. 512 dieser Sektoren werden wiederum zu einem *Cluster* zusammengefasst, d.h. in ein *Cluster* speichert 256 kB. Davon gibt es bei einer 2 GB Karte insgesamt 8192 Stück.
Da nun geklärt ist, wie die Daten auf der Karte abgelegt werden, soll noch kurz angerissen werden welche Funktionen dafür genutzt werden.

U8 noFS_Init(void) Während die Aufgabe der Funktion SD_Init() darin besteht die SD-Karte vorzubereiten, bestimmt diese Funktion die Position an der die Daten abgelegt werden. Dazu werden beim Booten sukzessiv die Enden aller Cluster auf 0x03 geprüft. So lange 0x03 gefunden wird, wird zum Ende des nächsten Clusters gesprungen, denn mit 0x03 wird das Ende eines Datensatzes gekennzeichnet. Findet sich kein 0x03 werden die einzelnen Sektoren des Clusters genauer untersucht. Dabei wird wieder das Ende jeden Sektors auf 0x03 geprüft und wenn das Muster nicht gefunden werden kann, ist der Anfang für die neuen Daten gefunden. Sollte die SD-Karte komplett voll sein (NOFS_ERROR_CARD_IS_FULL) bzw. die Initialisierung aus einem anderen Grund nicht erfolgreich verlaufen (NOFS_ERROR_CARDINITIALIZATION _FAILED,NOFS_ERROR_BLOCKLEN_NOT_ACCEPTED), leuchtet nach aktuelle Konfiguration die rote LED dauerhaft. Der Erfolgsfall wird auch hier durch NOFS_ SUCCESS gekennzeichnet.

U8 noFS_DataTransfer(void) Zu Beginn wurde schon auf die HexAscii-Konvertierung hingewiesen. Das Dateisystem bietet aber noch ein weiteres Feature. Normalerweise sind die aufgezeichneten Datensätze kleiner als 256 Bytes. D.h. es müssen mehrere Daten gesammelt werden bis der 512 Byte große Sektor gefüllt ist. Damit keine Daten verloren gehen, werden sie in einem Schattenpuffer zwischengespeichert, was von dieser Funktion angestoßen wird. Das Sichern selbst wird dabei von der nächsten Funktion gehandhabt. Konnte das Sichern angestoßen werden, liefert die Funktion NOFS_SUCCESS zurück und wenn nicht NOFS_FAILED.

void noFS_CyclicTrigger(void) Wie am *cyclic* im Namen erkennbar, handelt es sich um eine Funktion, die zyklisch aufgerufen wird. Bei jedem Aufruf wird überprüft, ob neue Daten vorhanden sind. Ist dies der Fall werden sie in den Schattenpuffer kopiert und der kopierte Datensatz wird zum Überschreiben freigegeben. Beim nächsten Aufruf werden die Daten aus dem Schattenpuffer in den Dateisystempuffer geschrieben, wobei sie dabei gleichzeitig die HexAscii-Konvertierung erfahren. Sobald der Dateisystempuffer voll ist, wird mittels SD_BlockTransfer() die Datenübertragung angestoßen.

Auf voran gegangenen Seiten wurde nun ausgiebig erörtert wie die Daten auf der SD-Karte abgelegt werden. Die restlichen Ausführungen beschäftigen sich mit dem Auslesen der Daten von der SD-Karte am Computer und deren Weiterverarbeitung.

Daten von der SD-Karte lesen

Da die Rohdaten im HexAscii-Format auf der Karte abgelegt werden, müssen sie nach dem Auslesen wieder zurückgewandelt werden. Um den Entwicklungsaufwand gering zu halten, hat man sich der Einfachheit halber des Programms dd bedient. Mit diesem Programm lassen sich vornehmlich unter Linux Datenströme kopieren und umleiten. Ein Datenstrom kann dabei nahezu alles sein. Eine Tastatureingabe, ein Pseudozufallszahlengenerator oder ein Datenspeicher. Dabei wird auf alle auf der untersten Ebene zugegriffen, was i.d.R. Administratorrechte erfordert, denn mit dem Befehl kann nicht nur ein lesender sondern auch ein schreibender Zugriff erfolgen und durch unvorsichtige Benutzung von letzterem kann das ausführende System beschädigt werden.
Um die SD-Karte zu identifizieren, können durch Übergabe des Parameters --list alle vorhanden Geräte ausgegeben werden, wie die folgende Ausgabe zeigt.

C:\WINDOWS\SYSTEM32\cmd.exe

```
C:\bin>dd.exe --list
rawwrite dd for windows version 0.6beta3.
Written by John Newbigin <jn@it.swin.edu.au>
This program is covered by terms of the GPL Version 2.

Win32 Available Volume Information
\\.\Volume{f7476ab8-ebd6-11de-a15a-806d6172696f}\
  link to \\?\Device\HarddiskVolume2
  fixed media
  Mounted on \\.\c:

\\.\Volume{f0af876a-ee35-11de-a168-00235a5096cb}\
  link to \\?\Device\CdRom0
  CD-ROM
  Mounted on \\.\d:

\\.\Volume{57041000-3b0a-11df-a1cc-00235a5096cb}\
  link to \\?\Device\Harddisk1\DP<1>0-0+6
  removeable media
  Mounted on \\.\e:

NT Block Device Objects
```

```
\\?\Device\CdRom0
  size is 2147483647 bytes
\\?\Device\Harddisk0\Partition0
  link to \\?\Device\Harddisk0\DR0
  Fixed hard disk media. Block size = 512
  size is 160041885696 bytes
\\?\Device\Harddisk0\Partition1
  link to \\?\Device\HarddiskVolume1
  Fixed hard disk media. Block size = 512
  size is 6448587264 bytes
\\?\Device\Harddisk0\Partition2
  link to \\?\Device\HarddiskVolume2
\\?\Device\Harddisk0\Partition3
  link to \\?\Device\HarddiskVolume3
  Fixed hard disk media. Block size = 512
  size is 43166237184 bytes
\\?\Device\Harddisk0\Partition4
  link to \\?\Device\HarddiskVolume4
  Fixed hard disk media. Block size = 512
  size is 1891782144 bytes
\\?\Device\Harddisk1\Partition0
  link to \\?\Device\Harddisk1\DR5
  Removeable media other than floppy. Block size = 512
  size is 4127195136 bytes

Virtual input devices
 /dev/zero   <null data>
 /dev/random <pseudo-random data>
 -           <standard input>

Virtual output devices
 -           <standard output>
 /dev/null   <discard the data>
```

Wie der Ausgabe entnommen werden kann, kann auf eine Festplatte (*Harddisk*), ein CD-Rom Laufwerk und auf ein entfernbares Medium (*removeable media*) zugegriffen werden. Interessant ist hierbei nur das entfernbare Medium, denn dabei handelt es sich um die im Kartenleser befindliche SD-Karte. Die Daten werden nun mit folgendem Befehl von der Karte gelesen und in eine Datei namens diskimage.img gespeichert.

C:\WINDOWS\SYSTEM32\cmd.exe

```
C:\bin>dd.exe if=\\?\Device\Harddisk1\Partition0 of=diskimage.img
```

Dabei steht if für *input file*, entspricht also der Quelle und of steht für *output file* und ist somit die Zieldatei. Neben diesen beiden Parametern gibt es noch weitere, die durch Verwendung des Parameters --help erklärt werden.

C:\WINDOWS\SYSTEM32\cmd.exe

```
C:\bin>dd.exe --help
rawwrite dd for windows version 0.6beta3.
```

```
Written by John Newbigin <jn@it.swin.edu.au>
This program is covered by terms of the GPL Version 2.

dd [bs=SIZE] [count=BLOCKS] [if=FILE] [of=FILE] [seek=BLOCKS]
   [skip=BLOCKS] [--size] [--list] [--progress]

SIZE and BLOCKS may have one of the following suffix:
 k = 1024
 M = 1048576
 G = 1073741824
default block size <bs> is 512 bytes
skip specifies the starting offset of the input file <if>
seek specifies the starting offset of the output file <of>
```

SD-Karte formatieren

Das Formatieren der Karte gestaltet sich ähnlich dem Auslesen. Während beim Auslesen von der Karte gelesen und auf die Festplatte geschrieben wurde, schreibt man beim Formatieren auf die Karte. Theoretisch könnte man eine beliebige Datei als Quelle nutzen, jedoch muss sicher gestellt werden, dass alle 0x03 Markierungen überschrieben werden, da der Sektor sonst nicht neu beschrieben wird. Deswegen bedient man sich anstelle einer zufälligen Datei dem Null-Gerät (*zero device*). Diese Quelle liefert nur Nullen, d.h. mit dem Ende des folgenden Befehls befinden sich auf der SD-Karte nur noch Nullen.

C:\WINDOWS\SYSTEM32\cmd.exe

```
C:\bin>dd.exe if=/dev/zero of=\\?\Device\Harddisk1\Partition0
```

Die Karte ist damit komplett formatiert und kann erneut beschrieben werden.
Um das Lesen und Schreiben von bzw. auf die Karte zu beschleunigen, kann man die weiteren Parameter nutzen. Möchte man z.B. nur eine bestimmte Anzahl an Datensätzen auslesen, so fügt man dem Auslesebefehl noch den Parameter ibs=512 und count=anzahl hinzu. Der erste Parameter legt die gelesene Blockgröße fest und der zweite legt fest wie viele dieser Blöcke ausgelesen werden sollen. Beim Schreiben verändert sich der Parameter ibs=512 zu obs=512.

Daten auswerten

Mit der bereits genannten Befehlszeile

C:\WINDOWS\SYSTEM32\cmd.exe

```
C:\bin>dd.exe if=\\?\Device\Harddisk1\Partition0 of=diskimage.img
```

lassen sich zwar die Daten von der SD-Karte in eine Datei schreiben, jedoch liegen diese dann immer noch im HexAscii-Format vor. Sie müssen daher noch umgewandelt werden,

um weiterverarbeitet werden zu können. Für diese Aufgabe wurde das Kommandozeilen Programm eXtractData erstellt. Es erwartet zwei Parameter. Zum Einen muss die mittels dd erstellte Imagedatei angegeben werden und zum Anderen wird mit dem Parameter –pattern= die Struktur übergeben. Anhand eines Beispiels soll die Verwendung verdeutlicht werden.

C:\WINDOWS\SYSTEM32\cmd.exe

```
C:\bin>eXtractData --pattern=YMDhmsi16x16i32i messung1.img
```

Die Imagedatei heißt im gegebenen Beispiel messung1.img und das angegebene Muster –pattern=YMDhmsi16x16i32i wird folgendermaßen interpretiert. Das erste Byte entspricht dem Jahr (Y), das nächste dem Monat (M), gefolgt vom Tag (D) an dem der Datensatz aufgezeichnet wurde sowie die Uhrzeit, die in den anschließenden Bytes hms gespeichert ist. Das Besondere an den genannten Manipulatoren ist, dass sie nur einmal pro Datensatz vorkommen dürfen und dass sie im konvertierten Ergebnis immer am Zeilenanfang stehen. Das nächste Byte wird ignoriert (i) und dann folgen sechzehn 16 Bit Werte, die wiederum von einer 32 Bit Zahl durch ein zu ignorierendes Byte getrennt werden. Das Muster endet mit einem i und für den Fall, dass der Datensatz länger ist als das übergebene Muster, wird der letzte Manipulator bis zum Datensatzende angewendet. Im vorliegenden Beispiel werden folglich alle Daten nach der 32 Bit Zahl ignoriert. Mittels des Multiplikators x kann - wie im Beispiel bereits angewandt - der Schreibaufwand deutlich verringert werden. Die konvertierten Daten werden in eine *comma separated values* Datei geschrieben, die von jedem geläufigen Tabellenkalkulationsprogramm geöffnet werden kann.

Der nächste Abschnitt widmet sich der wesentlich einfacheren seriellen Schnittstelle, aber auch diese hat ihre Tücken.

3.8 Serielle Schnittstelle

Die serielle Schnittstelle wird im englischen Sprachgebrauch auch *Universal Synchronous and Asynchronous serial Receiver and Transmitter (USART)* genannt. Da sich der Treiber auf den asynchronen Modus beschränkt, wurde der Treiber mit UART abgekürzt. Sein Können besteht im Empfangen und Senden von Zeichen mit der im Modul-Header (uart.h) eingestellten Baudrate. Die möglichen Baudraten müssen dabei der Gleichung (3.4) gehorchen, wobei die UBRRn-Register (UBRRnL und UBRRnH, *USARTn Baud Rate Register*) alle Werte zwischen 0 und 4095 annehmen können.

$$UBRRn = \frac{f_{CLKio}}{16 \cdot Baud} - 1 \tag{3.4}$$

Wie in Abschnitt 2.2.4 bereits erwähnt, müssen Sender und Empfänger die gleiche Baudrate aufweisen. Bereits kleinste Abweichungen können einen geregelten Datenaustausch unterbinden. Eine Übersicht und einen weiteren Zusammenhang liefert die Tabelle 3.7. Ihr ist zu entnehmen, dass die erlaubte Abweichung mit zunehmender Bitzahl

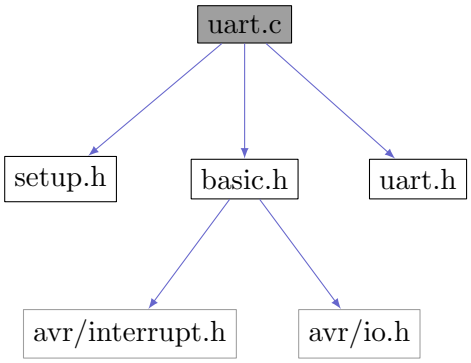

Abb. 3.16: Modulbeziehungen UART

sinkt. Verwunderlich ist das nicht, denn mit steigender Bitzahl läuft die Synchronität durch die Abweichung immer weiter auseinander.

D # (Data + Parity Bit)	$R_{slow}(\%)$	$R_{fast}(\%)$	Max Total Error (%)	Recommended Max Receiver Error (%)
5	93.20	106.67	+6.67/-6.8	± 3.0
6	94.12	105.79	+5.79/-5.88	± 2.5
7	94.81	105.11	+5.11/-5.19	± 2.0
8	95.36	104.58	+4.58/-4.54	± 2.0
9	95.81	104.14	+4.14/-4.19	± 1.5
10	96.17	103.78	+3.78/-3.83	± 1.5

Tab. 3.7: Empfohlene maximale Abweichung der Empfängerbaudrate [2, S. 193]

Die Baudrate wird über das DEFINE BAUD_RATE in der Datei uart.h eingestellt. In dieser wird auch noch die Puffergröße BUFFER_SIZE gesetzt, doch zu den Puffern wird später noch mehr gesagt.

U8 UART_Init(U8 databits, U8 paritybit, U8 stopbit) Neben dem Konfigurieren der Baudrate wird beim Initialisieren das *Frame*-Format eingestellt. Dabei sind der Phantasie fast keine Grenzen gesetzt. Es kann zwischen 5 und 9 Datenbits, einem (UART_ONE_STOPBIT) oder zwei Stopbits (UART_TWO_STOPBITS) und keinem (UART_NO_PARITY) oder einen gerade (UART_EVEN_PARITY) bzw. ungeraden (UART_ODD_PARITY) Paritätsbit gewählt werden. Sollten unzulässige Optionen gewählten worden sein, wird dies durch Rückgabe von UART_FAILED kenntlich gemacht. Nachdem das *Frame*-Format gesetzt wurde, werden die Interrupts eingeschaltet und die State-Machine auf UART_READY gesetzt.

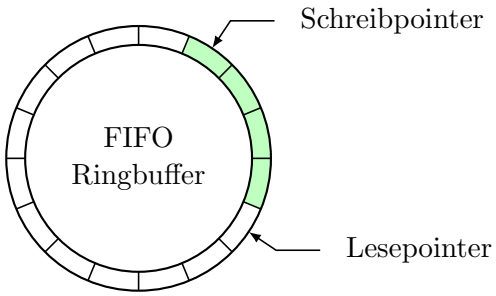

Abb. 3.17: Ringpufferprinzip

Ringpuffer

Ehe auf die einzelnen Sende- und Empfangsfunktionen eingegangen wird, soll erst noch der Ringpuffer anhand der Abbildung 3.17 erklärt werden. Wie der Abbildung zu entnehmen ist, gibt es zwei Adresszeiger. Einen Schreib- und einen Lesezeiger. Zu Beginn zeigen beide Zeiger auf die gleiche Adresse. Soll der Puffer nun mit Daten befüllt werden, so werden diese Byte für Byte an die Position geschrieben, auf die der Schreibzeiger deutet, wobei dieser nach jedem Byte erhöht wird. Dies ist so lange der Fall wie neue Daten anstehen bzw. bis das Ende des Puffers erreicht ist. In diesem Fall wird der Schreibzeiger wieder auf den Anfang des Puffers gesetzt und der Puffer wird wieder von vorne befüllt. Die Aufgabe des Lesezeigers ist es in der Zwischenzeit dem Schreibzeiger möglichst schnell zu folgen und die abgelegten Daten weiterzuverarbeiten. Andernfalls gehen Daten verloren, da es dem Schreibzeiger aus Gründen der Datenkonsistenz nicht gestattet werden kann den Lesezeiger zu überholen.
Im UART-Treiber ist sowohl für das Senden als auch für das Empfangen ein Ringpuffer implementiert.

void UART_WriteTXbuf(U8 *source) Eine Zeichenkette, d.h. mehrere Zeichen, können mit dieser Funktion verschickt werden. Dabei wird die an der übergeben Adresse befindliche Zeichenkette, in den Sendepuffer (TX) kopiert. Anschließend wird das Versenden der Zeichenkette angestoßen, indem das Interrupt-Flag gesetzt wird. Die daraufhin aufgerufene Interruptsroutine überträgt dann Zeichen für Zeichen bis der Schreibzeiger den Lesezeiger wieder eingeholt hat. Zwischen den einzelnen Zeichen wird die Routine verlassen, da sie nachdem ein Zeichen versendet wurde sowieso wieder aufgerufen wird und der Mikrocontroller sich so in der Zwischenzeit anderen Aufgaben widmen kann.

void UART_WriteSingleChar(U8 source) Soll nur ein einzelnes Zeichen versendet werden, bedient man sich dieser Funktion und übergibt ihr das Zeichen selbst. Auch hier passiert das eigentliche Versenden innerhalb der Interruptroutine.

U8 UART_ReadRXbuf(U8 *target, U8 max_chars) Die Funktion für den Empfang sieht auf den ersten Blick etwas schwieriger aus, der Funktionsaufruf erklärt sich aber nahezu von selbst. Sobald ein Zeichen empfangen wurde, wird dieses in den Ringpuffer geschrieben und der Schreibzeiger wird erhöht, damit er auf das nächste leere Feld zeigt. Der Lesezeiger verharrt hingegen beim ersten empfangenen Zeichen. Beim Aufruf dieser Funktion übergibt man ihr zwei Parameter. Der erste bestimmt wohin die im Puffer befindlichen Zeichen gespeichert werden sollen und der zweite legt fest wie viele, schließlich kann der Pufferinhalt größer sein als der vorgesehene Speicher und der daraus resultierende Überlauf kann zu einem Systemabsturz führen. Um festzustellen ob alle empfangenen Zeichen abgeholt werden konnten, liefert die Funktion die Anzahl der kopierten Zeichen zurück. Wenn weniger als MAX_CHARS Zeichen kopiert wurden, sind alle empfangenen Zeichen kopiert worden. Wurden jedoch genau MAX_CHARS Zeichen kopiert, gibt nur ein erneuter Aufruf der Funktion Aufschluss darüber ob alle Zeichen kopiert wurden.

Zu guter Letzt wird sich dem CAN-Treiber gewidmet.

3.9 Controller Area Network

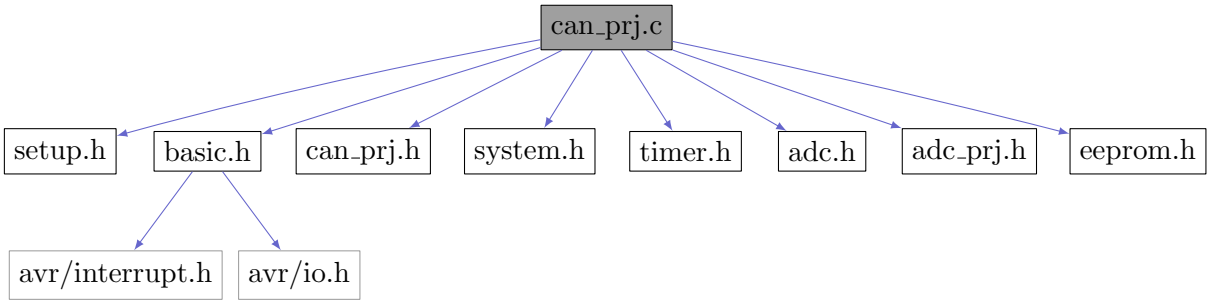

Abb. 3.18: Modulbeziehungen CAN

Das SD-Karten-Modul wurde zwar zu recht als das komplexeste bezeichnet, weil es am meisten Entwicklungsaufwand bedurfte, aber ein von Grund auf entwickelter CAN-Treiber wäre diesem in nichts nachgestanden. Um das Projekt deshalb zeitnah abschließen zu können wurde auf den Code einer Projektarbeit [3] zurückgegriffen. Herr Müsfik Akdere hat den Treiber in der Zwischenzeit nach eigener Aussage zu 50% überarbeitet und um einige Fähigkeiten erweitert. Exemplarisch soll angeführt werden, dass der Treiber in seiner Grundform in einer Endlosschleife hängen bleibt, wenn die CAN-Verbindung abreißt, z.B. durch Abziehen des Kabels, weil der auf die Empfangsbestätigung der Nachricht gewartet wird.
In Abschnitt 2.2.3 sind die möglichen Nachrichtenformate bereits angeschnitten worden.

Für die gebräuchlichsten Anwendungsfälle genügt es jedoch Nachrichten mit Daten- und Remote-Frame zu implementieren. Damit ist der Hauptzweck das Verschicken und Empfangen von Nachrichten bereits erfüllt.

100 kBit/s
125 kBit/s
200 kBit/s
250 kBit/s
500 kBit/s
1 MBit/s

Tab. 3.8: Übertragungsraten

Abbildung 3.18 zeigt neben den Modulabhängigkeiten wieder die strikte Trennung zwischen dem Treiber und der Anwendung. Der Treiber besteht aus den Dateien can.c und can.h und die Anwendung ist in den Dateien can_prj.c und can_prj.h zu finden. Beim *Controller Area Network*-Bus handelt es sich um ein asynchrones, serielles Bussystem, das mit verschiedenen Symbolraten (=Baudraten) betrieben werden kann. Wie der Abbildung 2.8 im Grundlagenabschnitt entnommen werden kann, gibt es keine separate Taktleitung. Die Information über den Takt muss somit, wie auch schon bei der seriellen Schnittstelle, in den Daten selbst enthalten sein. Deswegen ist es unabdingbar, dass alle Busteilnehmern die gleiche Baudrate gewählt werden. Zur Wahl stehen die in Tabelle 3.8 aufgelisteten Übertragungsgeschwindigkeiten.

Im Eingang dieses Kapitels wurde zwar erklärt nach welchen Regeln die Funktionen und Variabeln benannt wurden, da der CAN-Treiber jedoch nicht komplett selbst geschrieben wurde, brechen die Funktionsnamen mit der Nomenklatur. Im Folgenden werden erst einige Funktionen des Treibers beschrieben, ehe im Anschluss deren Einbettung in die Applikation aufgezeigt wird.

U8 CAN_init(U16 baud, U8 intmode) Wie jedes andere Modul bisher, muss auch das CAN-Modul initialisiert werden. Neben der Baudrate (100, 125, 200, 250, 500 oder 1000 kB/s) muss noch gesetzt werden wann und ob Interrupts ausgelöst werden sollen. Dabei kann zwischen fünf Möglichkeiten gewählt werden. NONE deaktiviert Interrupts, TX löst einen aus, wenn die Daten verschickt wurden, RX ist das Gegenstück zu TX und erzeugt einen Interrupt, wenn eine Nachricht empfangen wurde. Für den Fall dass sowohl beim Senden als auch beim Empfangen von Nachrichten ein Interrupt getriggert werden soll, nutzt man TXRX und wenn nur ein Interrupt kommen soll wenn beim Empfangen ein Fehler aufgetreten ist, fällt die Wahl auf RXERR.
Die Funktion selbst initialisiert alle MObs (*Message Object*), setzt die Baudrate und das Interruptverhalten und liefert letztlich immer den Wert 1 zurück, weil es keinerlei Fehlerbehandlung gibt. Die Funktion hätte somit auch als rückgabelos, also als Typ VOID, definiert werden können.

U8 CAN_enableMOB(U8 object, U8 mode, CAN_message msg) Bei der Verwendung des ursprünglichen Treibers ist vorgesehen nach der Initialisierung die einzelnen MObs einzuschalten. Dafür würde man sich dieser Funktion bedienen. In der Applikationsschicht wurde jedoch eine ähnliche Funktion implementiert, die die selbe Aufgabe hat, aber einfacher zu bedienen ist. Deswegen wird die Erklärung dieser Funktion ausgespart.

U8 CAN_disableMOB(U8 object) Zum Gegenstück von CAN_ENABLEMOB() gibt es keine Alternative. Das ist auch nicht nötig, denn die kurze und nahezu selbsterklärende Funktion kann kaum mehr vereinfacht werden. Sie erwartet nur einen Parameter und zwar die Nummer des MObs das ausgeschaltet werden soll. Auch diese Funktion hätte vom Typ VOID sein dürfen, denn auf die standardmäßige Rückgabe von 1 kann verzichtet werden.

U8 CAN_sendData(U8 mob, U8 *data) Mit dieser Funktion werden nun endlich Daten verschickt. Dazu muss nur das gewünschte MOb benannt und die Speicheradresse der Daten übergeben werden. Hier macht der gewählte Funktionstyp endlich Sinn, denn vor dem Versenden wird geprüft, ob das übergebene MOb überhaupt gültig ist. Ist dies nicht der Fall, wird die Funktion verlassen und eine 0 zurückgegeben. Ist das übergebene MOb gültig, wird es selektiert, die Nutzdaten werden in die zuständigen Register kopiert und das Senden wird angestoßen. Direkt danach wird die Funktion verlassen und eine 1 wird zurückgeliefert. Das Senden der Nachricht selbst wird vom integrierten CAN-Controller übernommen.

U8 CAN_sendRemote(U8 mob) Soll anstelle von Daten ein Remote-Frame gesendet werden, damit der Empfänger mit einer Nachricht antwortet, so bedient man sich dieser Funktion. Der Aufruf gestaltet sich wieder einfach denn er beschränkt sich auf die Übergabe des MObs. Dieses wird nach den gleichen Kriterien wie in CAN_SENDDATA() geprüft und führt bei Feststellung einer Verletzung dieser ebenfalls zur Rückgabe einer 0. Ansonsten wird ein Remote-Frame verschickt und die Funktion mit einer 1 verlassen.

Wie bereits angekündigt werden jetzt noch die Funktionen der Applikationsschicht erklärt. Den Anfang macht auch hier die Initialisierung.

void CAN_Initprj(void) Der erste Schritt ist auch hier das Setzen der Baudrate, dazu wird die Funktion CAN_INIT() aufgerufen. Dem schließt sich das Einrichten der MObs an. Die dafür notwendige Konfiguration findet sich in can_prj.c und Listing 3.10 zeigt den wesentlichen Ausschnitt. Dabei ist zu beachten, dass die ID Maske (idm) aus Gründen der Darstellbarkeit von 32 Bit auf 16 Bit verkürzt wurde.

```
STRUCT_CAN_MOB can_mob[]= {
// mob_nr, txrx_mob,     id,        idm, length, data[]
  { RTC_MOB,  RXMOB, {RTC_ID,  0xFFFE, 0, {0,..,0}}},
  { TEMP_MOB, TXMOB, {TEMP_ID, 0xFFFF, 0, {0,..,0}}},
  { HUM_MOB,  TXMOB, {HUM_ID,  0xFFFF, 0, {0,..,0}}},
...
}
```

Listing 3.10: Message Object Konfiguration

Die Struktur STRUCT_CAN_MOB setzt sich wiederum aus zwei weiteren Strukturen zusammen. Zum Einen aus der Struktur STRUCT_CAN_MOB die in Listing 3.11

gezeigt wird und zum Anderen aus der Struktur CAN_MESSAGE die Listing 3.12 zeigt.

```
typedef struct{
  U8          mob_nr;
  U8        txrx_mob;
  CAN_message msg;
} STRUCT_CAN_MOB;
```

Listing 3.11: Message Object Struktur

```
typedef struct {
  U32       id;  //Identifier (29 Bit)
  U32      idm;  //ID-Maske
  U8  length;  //Nachrichtenlaenge
  U8 data[8];  //Daten
} CAN_message;
```

Listing 3.12: Struktur einer CAN Nachricht

Für die Konfiguration eines MObs genügt es die entsprechende Zeile in Listing 3.10 anzupassen bzw. eine neue hinzuzufügen, wobei die maximale Anzahl von 15 MObs nicht überschritten werden darf. Eingestellt werden müssen nahezu alle Felder. Dazu gehören die MOb-Nummer, ob es sich um einen empfangenden (RXMOB) oder sendenden (TXMOB) MOb handelt und ganz wichtig die einmalige ID. Wenn das MOb nur empfängt muss der Empfangsbereich durch Setzen der Maske eingestellt werden. Sendet das MOb hingegen ist die Angabe irrelevant. Die letzten beiden Felder betreffen die eigentliche Nachricht genauer gesagt deren Format. Das Feld LENGTH definiert dabei die Länge der Nachricht (1..8 Byte) und im Datenfeld DATA stehen die eigentlichen Daten. Hier können schon voreingestellte Daten hinterlegt werden, es ist jedoch immer möglich diese zu verändern. Nur die Struktur sollte sich nicht mehr ändern, schließlich erwartet der Empfänger ein einmal definiertes Format. Zur Länge sei noch gesagt, dass zwar der Anschein erweckt wird, es würden verschiedene Nachrichtenlängen unterstützt, die Nachrichten werden jedoch generell auf eine Länge von 8 Byte festgelegt. Eine Erweiterung um diese Fähigkeiten in zukünftigen Treiberversionen ist jedoch anzunehmen.

void CAN_SendDummy(void) Bevor das Kapitel abgeschlossen wird, soll noch kurz gezeigt werden wie einfach eine neue Versandfunktion aufgebaut werden kann. Ein Blick in Listing 3.13 genügt dazu bereits.

```
void CAN_SendDummy(void) {
  can_mob[CAN_DUMMY].msg.data[0] = 42;
  can_mob[CAN_DUMMY].msg.data[1] = 43;
  ..
  CAN_sendData(CAN_DUMMY, can_mob[CAN_DUMMY].msg.data);
}
```

Listing 3.13: Beispiel für eine Versandfunktion

Im Grunde genommen bedarf es nur zweier Schritte. Im ersten Schritt wird das MOb mit den zu versendenden Werten gefüllt. Hier werden z.B. die Zahlen 42 und 43 in den Datenbereich des MObs gespeichert. Daran schließt sich der zweite und letzte Schritt an, nämlich das Anstoßen des Versands.

Nun endet die Besprechung der einzelnen Software-Module und ihrer Funktion und Strukturen. Im nächsten Kapitel wird anhand zweier Anwendungsmöglichkeiten deren Einsatz gezeigt.

KAPITEL 4

Anwendungsmöglichkeiten

Die in den beiden folgenden Abschnitten besprochenen Anwendung gehören zwar nicht zum Aufgabenprofil dieser Arbeit, aber sie geben einen ersten Eindruck wozu dieses Board genutzt werden kann und zeigen leider auch schon erste Grenzen auf.

4.1 Temperaturmessung

Für eine Temperaturmessung bedient man sich im einfachsten Fall des ohmsches Gesetzes $U = R(T) \cdot I$ und der Temperaturabhängigkeit des Widerstandes, siehe Gleichung (4.2).

$$R(T) = R_{25} \cdot (1 + \alpha \cdot (T_A - 25°C) + \beta \cdot (T_A - 25°C)^2) \tag{4.1}$$

$$R(\Delta T_A) = R_{25} \cdot (1 + \alpha \cdot \Delta T_A + \beta \cdot (\Delta T_A)^2) \tag{4.2}$$

Wie unschwer zu erkennen ist, wird der Zusammenhang zwischen Temperatur und Widerstandswert durch eine Näherung, genauer gesagt um eine Taylor-Reihenentwicklung, beschrieben. Wie bei Näherungen üblich liefert auch diese nur in der Nähe des Bezugspunktes gültige Werte. Als Bezugspunkt wurde hier der Widerstandswert des Bauteils bei 25 °C gewählt. Die Parameter α und β sind im Datenblatt [6] mit $7,88 \cdot 10^{-3} K^{-1}$ und $1,937 \cdot 10^{-5} K^{-2}$ angegeben und hinter ΔT_A verbirgt sich die Differenz zwischen der aktuellen Umgebungstemperatur (der Index A steht für *ambient* zu deutsch Umgebung) und der Bezugstemperatur (hier 25 °).

In Datenblättern wird i.d.R. der einheitenlose Temperaturfaktor k_T angegeben. Dabei handelt es lediglich um eine Normierung der Gleichung (4.2) auf den Bezugswiderstand R_{25}, wie in Gleichung (4.3) zu sehen ist.

$$k_T = \frac{R(T_A)}{R_{25}} = (1 + \alpha \cdot \Delta T_A + \beta \cdot \Delta T_A^2) = f(T_A) \tag{4.3}$$

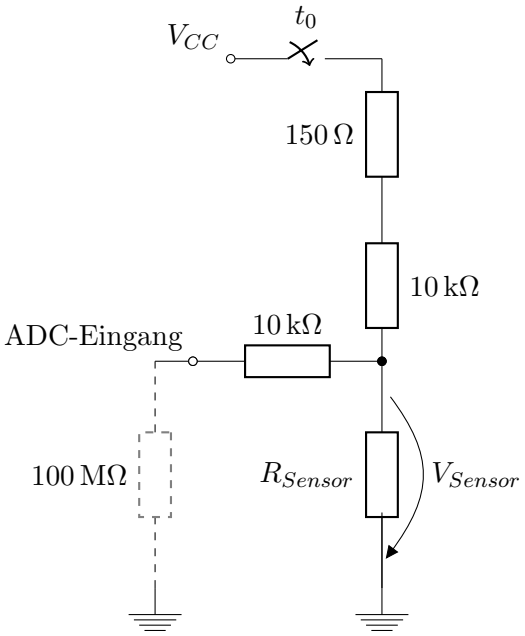

Abb. 4.2: Temperatursensorbeschaltung

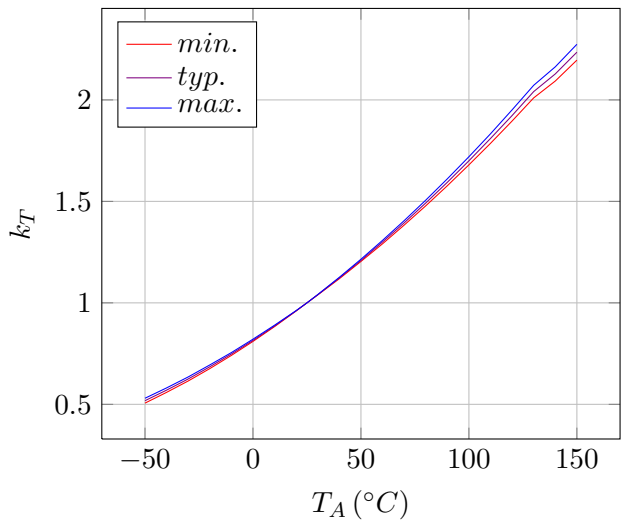

Abb. 4.1: Kennlinie: Temperatursensor [6]

Wie der Kennlinie in Abbildung 4.1 entnommen werden kann, ist der Temperaturfaktor kleiner Eins bei Temperaturen kleiner 25 ° C und größer Eins für Temperaturen größer 25 ° C. Der Faktor k_T ist somit ein Maß für die Abweichung der Temperatur von der Bezugstemperatur.
Um für einen größeren Bereich genaue Werte zu erhalten bietet es sich an eine Wertetabelle anstelle einer linearisierten Beziehung zu verwenden. Die Genauigkeit der Tabelle wird dabei vom ADC festgelegt, schließlich macht es keinen Sinn genauere Werte in der Tabelle zu hinterlegen als der ADC liefern kann.

Der Aufbau der Temperaturmessung kann Abbildung 4.2 entnommen werden. Dort ist zu sehen, dass der Temperatursensor (Widerstand) mit einem gewöhnlichen Wider-

stand in Reihe geschalten von einem digitalen Ausgang mit Spannung versorgt wird. Der Sensor reagiert nun empfindlicher bei Veränderungen der Temperatur und somit ergibt sich ein sich mit der Temperatur ändernder Spannungsteiler. Mittels des ADCs wird nun die Spannung am Sensor abgegriffen und der Widerstandswert berechnet. Als letzter Schritt wird die zugehörige Temperatur in der Wertetabelle nachgeschlagen. Um eine Verfälschung der Messung durch zusätzliche Erwärmung der Messwiderstände zu verhindern, wird die Spannungsversorgung für die Widerstandsreihe nur pulsförmig, d.h. für die Dauer der ADC-Wandlung, zugeschalten.
Für die Berechnung der Sensorspannung muss zusätzlich, wie eingezeichnet, der 150 ΩWiderstand berücksichtigt werden, der sich am digitalen Ausgang befindet. Die zur Sensorspannung gehörige Berechnungsvorschrift ist in Gleichung (4.4) zu entnehmen.

$$V_{Sensor} = \frac{R_{Sensor}}{R_{Sensor} + 10\ k\Omega + 150\ \Omega} \cdot V_{cc} \tag{4.4}$$

Etwas komplexer wirkt im ersten Augenblick das nächste Messverfahren.

4.2 Feuchtigkeitsmessung

Während bei der Temperaturmessung eine Wertetabelle bzw. eine linearisierte Kennlinie zugrunde lag, nutzt man bei der Feuchtigkeitsmessung die kapazitive Eigenschaft des Feuchtigkeitssensors in Kombination mit einem Widerstand. Die beiden Komponenten ergeben zusammen ein RC-Glied, das ein wohlbekanntes Verhalten hat.

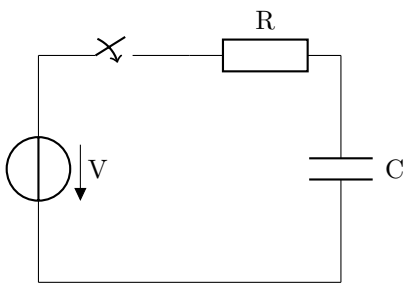

Abb. 4.3: RC-Glied

Schließt man, wie Abbildung 4.3 zeigt, einen leeren Kondensator mit einem in Serie geschalteten Widerstand an eine Spannungsquelle an, so lädt sich dieser auf. Je kleiner der Widerstand dabei ist, desto schneller. Die Bauteilgrößen Widerstand und Kapazität bestimmen dabei die Anfangssteigung der in Abbildung 4.4 dargestellten Ladekurve, da sie in die Zeitkonstante $\tau = R \cdot C$ einfließen. Wie in der Abbildung gekennzeichnet ist, sinkt die Anfangssteigung mit größer werdendem τ. Die Kurve selbst wird mit Gleichung (4.5) beschrieben.

$$U(t) = U_{max} \cdot (1 - e^{-\frac{t}{\tau}}) \tag{4.5}$$

Die Feuchtigkeit kann nun durch die Feuchtigkeitsabhängigkeit der Sensorkapazität bestimmt werden. Mit der sich ändernden Kapazität ändert sich die Zeitkonstante und dadurch die Anfangssteigung der Ladekurve. Misst man nun immer nach der gleichen

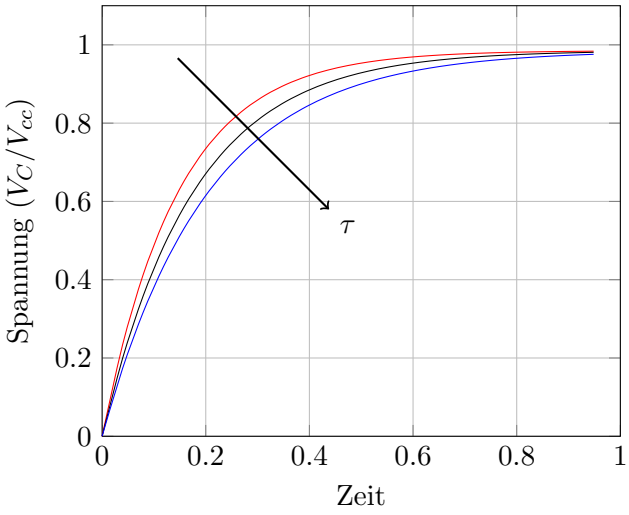

Abb. 4.4: Ladekurve RC-Glied

Zeit die Spannung über dem Kondensator, dann ist dieser in Abhängigkeit der Feuchtigkeit mal mehr mal weniger aufgeladen. Da die Kurve zu Beginn als linear angesehen werden kann, kann durch Anlegen eines einfachen Steigungsdreiecks die Zeitkonstante bestimmt werden, denn diese entspricht der Steigung des Dreiecks. Die Zeitkonstante ist aber auch das Produkt von Widerstand und Kapazität. Da der Widerstand einen festen Wert (680 kΩ) besitzt kann somit die Kapazität berechnet werden. Diese ist wiederum eine nahezu lineare Funktion der Feuchtigkeit, was dem Graph in Abbildung 4.5 entnommen werden kann. Alternativ bietet sich auch hier wieder eine Wertetabelle an.

Die mathematischen Beziehungen sollen im Folgenden aufgezeigt werden.

$$m = \frac{\partial}{\partial t}(V_{cc} \cdot (1 - e^{-\frac{t}{\tau}}) \tag{4.6}$$

$$= V_{cc} \cdot (0 - (-\frac{1}{\tau}) \cdot e^{-\frac{t}{\tau}}) \tag{4.7}$$

$$= V_{cc} \cdot \frac{1}{\tau} \cdot \underbrace{e^{-\frac{t}{\tau}}}_{\approx 0} \tag{4.8}$$

$$= \frac{V_{cc}}{\tau} \tag{4.9}$$

Zuerst wird die zeitliche Ableitung von Gleichung (4.5) gebildet, um die Kurvensteigung in jedem Punkt des Graphens zu erhalten. Es ist jedoch nur die Steigung am Anfang der Kurve von Interesse, deswegen kann die gewonnene Gleichung vereinfacht werden,

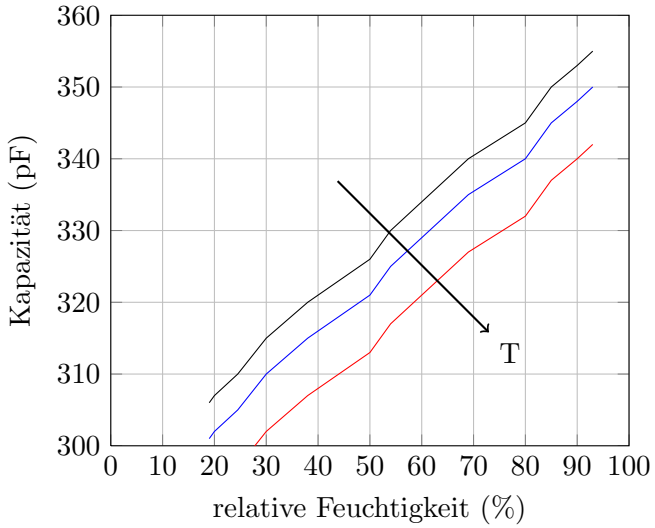

Abb. 4.5: Feuchtigkeitscharakteristik [5]

denn in diesem Bereich ist der Einfluss des Faktors $e^{-\frac{t}{\tau}}$ nicht von Bedeutung. Für die Kurvensteigung erhält man somit die Gleichung (4.9).
Die Steigung eines Steigungsdreiecks kann Gleichung (4.12) entnommen werden.

$$m = \frac{U_1 - U_0}{t_1 - t_0} \tag{4.10}$$

$$= \frac{U_1 - 0}{t_1 - 0} \tag{4.11}$$

$$= \frac{U_1}{t_1} \tag{4.12}$$

Setzt man nun beide Gleichung gleich und ersetzt τ durch das Produkt von R und C, so erhält man letztlich eine Aussage (4.15) über die Kapazität.

$$\frac{V_{cc}}{\tau} = \frac{V_1}{t_1} \tag{4.13}$$

$$\tau = \frac{V_{cc}}{V_1} \cdot t_1 = R \cdot C \tag{4.14}$$

$$C = \frac{V_{cc}}{V_1} \cdot \frac{t_1}{R} \tag{4.15}$$

Mit dieser kann nun in einem letzten Schritt die Luftfeuchtigkeit ermittelt werden. Dazu ist es wiederum nötig die Temperatur zu wissen, denn wie Abbildung 4.5 auch entnommen werden kann, sinkt die Kennlinie mit steigender Temperatur.

In der Praxis hat die Messmethode soweit zwar funktioniert, jedoch bereitet die niedrige Ladespannung Probleme. Denn die damit einhergehende geringe Variation der Geradensteigung und die geringe Auflösung des ADCs unterbinden eine feine und genaue Bestimmung der Zeitkonstante. Schnelle Feuchtigkeitsänderungen können somit nicht erfasst werden, aber bei Versuchen drängte sich auch der Verdacht auf, dass der Sensor selbst dafür nicht gemacht ist. Um nun einen stabilen Wert zu erhalten, bedient man sich einer Mittelung (*Infinite Impuls Response* (IIR)). Dabei wird der neue Messwert zum gewichteten alten addiert und die entstandene Summe wird durch die Anzahl der insgesamt gewerteten Messungen dividiert. In mathematischer Form ist die Berechnungsvorschrift noch einmal in Gleichung (4.16) zu sehen.

$$V = \frac{V_{neu} + m \cdot V_{alt}}{n} \tag{4.16}$$

Anstelle eine unendliche Anzahl an Messungen für die Auswertung zu nutzen, gibt es auch die Möglichkeit eine endliche Anzahl an Messwerten mit einzubeziehen (*Finite Impuls Response* (FIR)). Die dazugehörige Rechenvorschrift gleicht Gleichung (4.16) mit dem Unterschied, dass nach einer bestimmten Anzahl Messungen das bisherige Ergebnis verworfen wird.

Der Messvorgang selbst ist in der nebenstehenden Abbildung 4.6 veranschaulicht. Zu Beginn ist der Kondensator entladen und es liegt keine Spannung an. Wird nun eine Spannung angelegt, dargestellt durch die rote Sprungfunktion $\delta(t)$, so beginnt die Kondensatorspannung (violett) zu steigen. Nach einer gewissen Zeit wird die über dem Kondensator anliegende Spannung vom *Sample&Hold*-Glied des ADCs abgegriffen und gewandelt. Sobald das *Sample&Hold*-Glied die Spannung festhält, kann die äußere Ladespannung wieder entfernt werden. Um den Graphenverlauf übersichtlich zu halten, wurde darauf jedoch verzichtet. Es spielt für die Anwendung auch keine Rolle solange nur die Kondensatorspannung bis zum nächsten Einschalten wieder auf 0 abgefallen ist. Die Entladung kann dabei länger dauern als das Laden, da andere Zeitkonstanten vorliegen können.

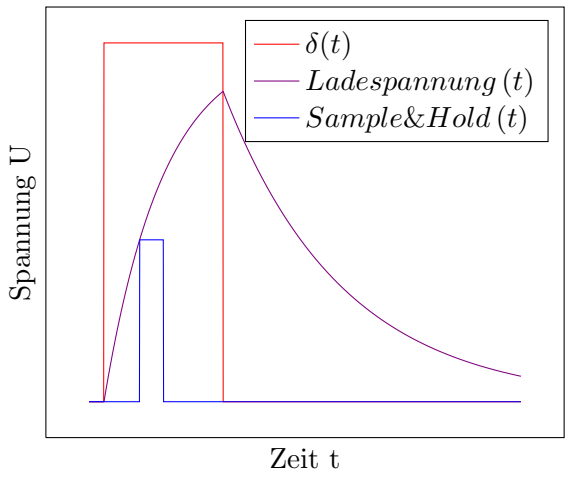

Abb. 4.6: Spannungsverlauf

Die Änderung der Zeitkonstanten kann am einfachsten mit Hilfe von Abbildung 4.7 nachvollzogen werden. Während des Aufladens wird τ durch den 680 kΩ Widerstand

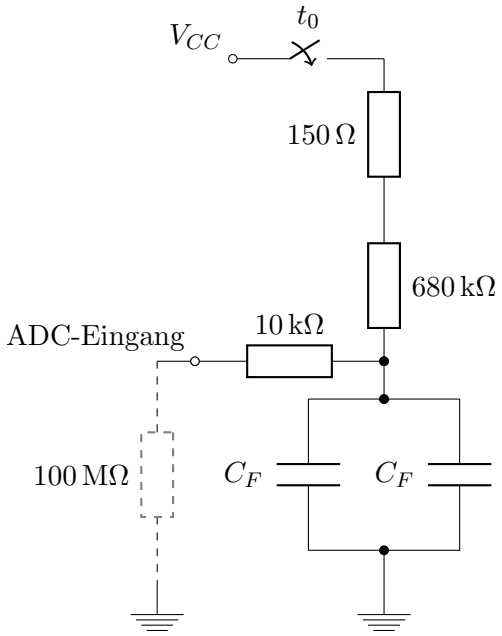

Abb. 4.7: Mikrocontroller mit Feuchtigkeitssensorbeschaltung

und der Sensorkapazität gebildet. Beim Entladen bieten sich jetzt aber zwei Wege an. Der erste führt wieder über den 680 kΩWiderstand zurück in den Mikrocontroller und der zweit führt über den 10 kΩWiderstand in den Mikrocontroller. Die Eingänge des Mikrocontrollers sind jetzt hochohmig, im Bereich von 100 MΩ, und damit ergibt sich eine wesentlich größe Zeitkonstante für das Entladen.
Der Abbildung 4.7 kann ebenfalls entnommen werden, dass zwei Sensoren parallel geschaltet wurden. Dadurch erhöht sich die Gesamtkapazität und als Folge flacht die Ladekurve ab bzw. Variationen in der Kapazität kommen deutlicher zum Vorschein. Im Versuch führte auch dies nicht zu einer genaueren Messung, so hat man sich Gedanken über Änderungen gemacht, die die Messung verbessern dürften. Die Ergebnisse und weitere Verbesserungsvorschläge finden sich im nächsten und letzten Kapitel.

KAPITEL 5

Ausblick

„Es ist noch kein Meister vom Himmel gefallen“ und so war es auch hier. Neben die schlechte Auswertung der Feuchtigkeit gesellten sich noch weitere Fehler, die in diesem letzten Kapitel kurz erfasst werden sollen, um in einem möglichen Redesign berücksichtigt zu werden.
Ein durchgehend gemachter Fehler ist, dass ungenutzte Pins **nicht** mit Masse verbunden wurden, um sie auf ein definiertes Potential zu führen. Dadurch kam es zu Problemen was bei der Inbetriebnahme des DACs festgestellt werden musste. So variierte die Spannung, die an Pin 16 (LDAC) des DACs gemessen wurde, stark. Der DAC ist nun aber intern so beschalten, dass eine steigende Flanke an diesem Pin den Inhalt der temporären Register in die Ausgangsregister laden lässt. Damit ändert sich die Ausgangsspannung und wie sich leicht vorstellen lässt, kann dies zu einem sehr undefinierten Verhalten führen. Nachdem der Pin auf Masse gezogen wurde, war der Fehler behoben und es stellte sich ein stabiler Betrieb ein.
Zu diesem unverknüpften Pin gesellen sich noch zwei falsch verbundene. So wurde im Boardlayout fälschlicherweise der Pin 13 (A0) fest mit der Versorgungspannung verbunden und der Pin 12 (IOV_{DD}), der eigentlich mit V_{cc} verbunden hätte werden sollen, mit dem Mikrocontroller. Mittels einer Lötbrücke ist dieser Fehler behoben, jedoch muss beachtet werden, dass der Mikrocontrollerpin PG3 durch diesen Fehler fest auf V_{cc} liegt und deshalb nicht als Ausgang betrieben werden darf. Andernfalls nimmt der Controller Schaden.
Kein dramatischer Fehler, aber ein Fehler der einem mehr Lötgeschick abfordert als nötig, war die Verwendung von falschen Lötpads beim DC/DC Eingangstransformator und der schlecht gewählte Leitungsverlauf vom Mikrocontroller zum RTC. Diese verlaufen unterhalb des Mikrocontroller-Quarzes, was ein Übersprechen der hochfrequenten Systemtaktleitung auf die niederfrequente RTC-Leitung begünstigt. Interferenzen konnten aber nicht beobachtet werden, was der vergleichsweise niedrigen Taktfrequenz ge-

schuldet sein dürfte. Aber generell sollten Leitungen nicht unterhalb von hochfrequenten Bauteile verlaufen.
Zu letzt noch ein Wort zu der schlechten Feuchtigkeitsauswertung. Betrachtet man nochmals Gleichung (4.15), so sieht man, dass in diese die Versorgungsspannung einfließt. Diese ist mit 5 V relativ niedrig und eine größere würde die mögliche Genauigkeit erhöhen, da die Steigungsunterschiede in der Ladekurve durch die Streckung deutlicher zum Vorschein kommen. Dabei muss dann jedoch beachtet werden, dass der Messvorgang abgeschlossen ist, bevor der maximale Bezugswert AREF von der Ladespannung überschritten wurde.

Die bisherigen Fehler sind allesamt behebbar. Bei der Entwicklung kam jedoch ein Fehler zum Vorschein, der nicht beeinflusst werden kann. Die einzelnen Pins des Mikrocontrollers können einmal als digitale Ausgänge genutzt werden, d.h. sie können auf *high* oder *low* gesetzt werden, und andermal warten die meisten mit einem speziellen Verhalten auf. Dazu zählen z.B. die ADC-, die SPI- oder die CAN-Pins. Im Allgemeinen geht man davon aus, dass die Pins ihre Spezialfunktion unabhängig von der Vorgeschichte bereitstellen. Bei den TWI-Pins ist dies jedoch nicht der Fall. Definiert man vor dem Aktivieren der TWI-Funktionen die verwendeten Pins als Ausgänge, so ist es dem Kommunikationspartner nicht möglich den Empfang des versendeten Bytes zu bestätigen, da es die Datenleitung nicht auf *low* ziehen kann.
Man kann sich leicht vorstellen wie schwierig es ist, die Ursache für dieses Verhalten ausfindig zumachen, vor allem wenn doch vorher alles funktionierte. Nichtsdestotrotz konnte diese und viele andere Hürden genommen werden und ein einsatzbereites und -taugliches Werkzeug herausgearbeitet werden, das aufgrund seiner Vielseitigkeit sicherlich in allerlei Einsatzgebieten genutzt wird.

Literaturverzeichnis

[1] ISO 11898-1:2003. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=33422, 2003.

[2] Atmel Coporation. *AT90CAN32/64/128 Manual*, 2008. [Online; accessed 08-September-2011].

[3] Jan Christ and Marco Glietsch. Realisierung einer CAN-Kommunikation mit dem AT90CAN128. http://www.mikrocontroller.net/topic/98697, 2008. [Online; accessed 10-September-2011].

[4] Hans-Jürgen Gevatter and Ulrich Grünhaupt. *Handbuch der Mess- und Automatisierungstechnik im Automobil*. Springer, 2nd edition, 2006.

[5] Hygrosense Instruments GmbH. Kapazitiver Feuchtesensor.

[6] Infineon Technologies. Silicon Temperature Sensors.

[7] Holger Klabunde. AVR Board - Full FAT. http://www.holger-klabunde.de/avr/avrboard.htm#FullFAT, 2008. [Online; accessed 08-November-2011].

[8] Reinhard Lerch. *Elektrische Messtechnik*. Springer, 4th edition, 2007.

[9] Maxim. DS1307, 64 x 8, Serial, I2C Real-Time Clock. http://datasheets.maxim-ic.com/en/ds/DS1307.pdf, 2008. [Online; accessed 04-October-2011].

[10] Ulrich Radig. AVR - mmc/sd. http://www.ulrichradig.de/home/index.php/avr/mmc-sd, 2008. [Online; accessed 08-September-2011].

[11] Roland Riegel. *AVR - SD-Reader*, 2008. [Online; accessed 08-September-2011].

[12] Thorsten Spätling. *Messadapter manual*.

[13] telos EDV Systementwicklung GmbH. I^2C Bus. http://www.i2c-bus.org/de/twi-bus/, 2008. [Online; accessed 16-September-2011].

Index