

Ansgar Meroth
Petre Sora

Sensornetzwerke in Theorie und Praxis

Embedded Systems-Projekte
erfolgreich realisieren



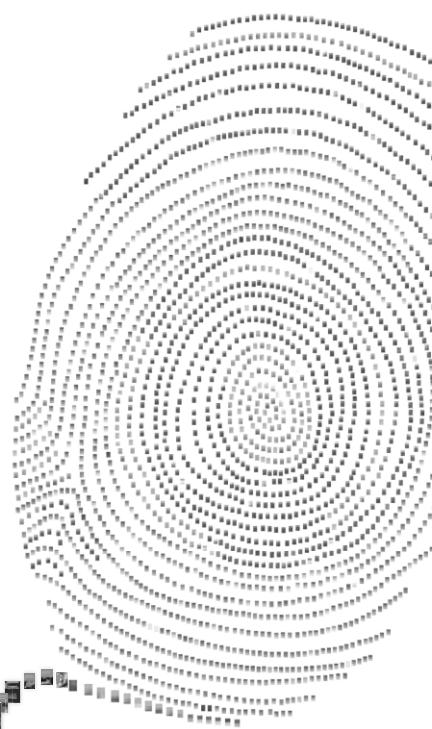
Springer Vieweg

Sensornetzwerke in Theorie und Praxis

Lizenz zum Wissen.

Sichern Sie sich umfassendes Technikwissen mit Sofortzugriff auf tausende Fachbücher und Fachzeitschriften aus den Bereichen:
Automobiltechnik, Maschinenbau, Energie + Umwelt, E-Technik,
Informatik + IT und Bauwesen.

Exklusiv für Leser von Springer-Fachbüchern: Testen Sie Springer für Professionals 30 Tage unverbindlich. Nutzen Sie dazu im Bestellverlauf Ihren persönlichen Aktionscode **C0005406** auf www.springerprofessional.de/buchaktion/



Jetzt
30 Tage
testen!

Springer für Professionals.
Digitale Fachbibliothek. Themen-Scout. Knowledge-Manager.

- ⌚ Zugriff auf tausende von Fachbüchern und Fachzeitschriften
- ⌚ Selektion, Komprimierung und Verknüpfung relevanter Themen durch Fachredaktionen
- ⌚ Tools zur persönlichen Wissensorganisation und Vernetzung

www.entschieden-intelligenter.de

Springer für Professionals

 Springer

Ansgar Meroth · Petre Sora

Sensornetzwerke in Theorie und Praxis

Embedded Systems-Projekte erfolgreich
realisieren



Springer Vieweg

Ansgar Meroth
Hochschule Heilbronn
Heilbronn, Deutschland

Petre Sora
Hochschule Heilbronn
Heilbronn, Deutschland

ISBN 978-3-658-18385-1
<https://doi.org/10.1007/978-3-658-18386-8>

ISBN 978-3-658-18386-8 (eBook)

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

Springer Vieweg

© Springer Fachmedien Wiesbaden GmbH, ein Teil von Springer Nature 2018

Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Jede Verwertung, die nicht ausdrücklich vom Urheberrechtsgesetz zugelassen ist, bedarf der vorherigen Zustimmung des Verlags. Das gilt insbesondere für Vervielfältigungen, Bearbeitungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürfen. Der Verlag, die Autoren und die Herausgeber gehen davon aus, dass die Angaben und Informationen in diesem Werk zum Zeitpunkt der Veröffentlichung vollständig und korrekt sind. Weder der Verlag noch die Autoren oder die Herausgeber übernehmen, ausdrücklich oder implizit, Gewähr für den Inhalt des Werkes, etwaige Fehler oder Äußerungen. Der Verlag bleibt im Hinblick auf geografische Zuordnungen und Gebietsbezeichnungen in veröffentlichten Karten und Institutionsadressen neutral.

Gedruckt auf säurefreiem und chlorfrei gebleichtem Papier

Springer Vieweg ist ein Imprint der eingetragenen Gesellschaft Springer Fachmedien Wiesbaden GmbH und ist ein Teil von Springer Nature.

Die Anschrift der Gesellschaft ist: Abraham-Lincoln-Str. 46, 65189 Wiesbaden, Germany

Wir widmen dieses Buch unseren Familien.

Vorwort

Das „Internet of things“ ist in aller Munde. Eine wichtige Voraussetzung für die digitale Revolution im Alltag ist die Verfügbarkeit kleinstter Mikrocontroller, die mit Sensoren und Aktoren verbunden und miteinander vernetzt sind. Die Rechenleistung spielt hierbei eine geringere Rolle als der Stromverbrauch und der Bauraum. Mit der Verfügbarkeit miniaturisierter, hoch präziser und preiswerter Sensoren wächst die Zahl der Anwendungen schnell und eröffnet auch kleineren Unternehmern, Studierenden und ambitionierten Bastlern bisher ungeahnte Möglichkeiten. Gleichzeitig wird das Verständnis für die Funktionsweise von Sensoren und ihre Ansteuerung immer schwieriger, zumal viele Sensoren von den Herstellern für die Integration in Netzwerken ausgelegt sind. Die hohe Dynamik auf dem Komponentenmarkt und die für Anfänger zum Verständnis oftmals unzureichende Dokumentation erfordern einen großen Einarbeitungsaufwand. Dabei werden in vielen Fällen nur die Basisfunktionen der Bausteine benötigt, daher enthält dieses Buch dafür sehr konkrete Anleitungen und Erklärungen.

Eine gut durchdachte, modulare Ansteuerungsarchitektur ermöglicht einen hardwareunabhängigen und leicht auf die konkrete Umgebung adaptierbaren Aufbau der Ansteuerungshard- und -software. So sind die meisten Beispiele so formuliert, dass sie sich auch für andere Plattformen eignen und nur die direkte Hardwareanbindung ausgetauscht werden muss.

Das Buch enthält jedoch nicht nur konkrete Programmieranleitungen für die eingesetzten Bausteine sondern auch viele Tipps und Tricks zum effizienten Einsatz der Programmiersprache C in Mikrocontrollern. Es soll dabei zum eigenen Weiterdenken und Weiterentwickeln anregen.

Die Autoren freuen sich daher über Feedback, Ideen und Anregungen für künftige Auflagen!

Heilbronn, im April 2018

Petre Sora
Ansgar Meroth

Inhaltsverzeichnis

1	Einleitung	1
1.1	Was finden Sie in diesem Buch?	1
1.2	Für wen ist dieses Buch geschrieben?	2
1.3	Welche Kenntnisse setzt das Buch voraus?	3
1.4	Warum beschreibt das Buch keinen Arduino?	3
1.5	Zusatzmaterialien	3
1.6	Disclaimer/Haftungsausschluss	3
1.7	Dank	4
	Literatur	4
2	Einführung in die Programmiersprache C	5
2.1	Die Sprache C: Hintergrund und Aufbau	5
2.2	Bezeichner, Schlüsselworte und Symbole in C	8
2.2.1	Bezeichner	8
2.2.2	Schlüsselworte	8
2.2.3	Symbole	9
2.2.4	Anweisungen	9
2.3	Kommentare	9
2.3.1	Einzeliger Kommentar	10
2.3.2	Mehrzeiliger Kommentar	10
2.3.3	Doxygen Kommentare	10
2.4	Typen, Variablen und Konstanten	11
2.4.1	Fundamentale Datentypen	11
2.4.2	Deklaration von Variablen	14
2.4.3	Konstanten	15
2.5	Operatoren	16
2.5.1	Arithmetische Operatoren	17
2.5.2	Logische Operatoren	18
2.5.3	Bitoperatoren	18
2.5.4	Operatoren für Speicherzugriffe	18
2.5.5	Weitere Operatoren	19

2.5.6 Assoziativitt und Prioritt von Operatoren	19
2.5.7 Typumwandlung	20
2.6 Kontrollstrukturen	21
2.6.1 Verzweigung (Auswahl)	21
2.6.2 Fallstricke	23
2.6.3 Mehrfachverzweigung	24
2.7 Schleifen	25
2.7.1 Kopfgesteuerte Schleifen	25
2.7.2 Fugesteuerte Schleifen	26
2.7.3 Zahlschleifen	27
2.7.4 Sprnge	28
2.8 Funktionen	28
2.8.1 int main()	28
2.8.2 Definition und Deklaration	29
2.8.3 Sichtbarkeit und Lebensdauer von Variablen in Funktionen	31
2.8.4 Header	32
2.8.5 Die Schlsselworte extern, volatile und static	33
2.9 Komplexe Datentypen	35
2.9.1 Arrays, Felder und Strings	35
2.9.2 Struktur	38
2.9.3 Unions	40
2.9.4 Aufzhlungstypen	42
2.9.5 Pointer	43
2.10 Aufbau eines embedded C-Programms	46
2.11 bersetzen und Binden	48
Literatur	49
3 Programmierung von AVR Mikrocontrollern	51
3.1 Architektur der AVR Familie	51
3.2 Gehuse und Anschlussbelegungen	54
3.3 Versorgung, Takt und Reset-Logik	55
3.3.1 Versorgung	55
3.3.2 Takt	55
3.3.3 Reset-Logik	57
3.3.4 Speicher	57
3.4 Umgang mit Registern	59
3.5 Digitaler Input/Output	62
3.5.1 Grundstzlicher Aufbau	62
3.5.2 Programmierung	64
3.6 Interrupts	68
3.6.1 Einstieg in den Umgang mit Interrupts	68
3.6.2 Interrupt-Programmierung am Beispiel des Pin Change Interrupt .	69

3.6.3	Die externen Interrupts INTx	72
3.7	Timer	73
3.7.1	Timer Grundlagen	73
3.7.2	Programmierung der Timer/Counter	75
3.8	Analoge Schnittstelle	88
3.8.1	Analogmultiplexer	88
3.8.2	Analogkomparator	89
3.8.3	AD-Wandler (ADC)	90
3.8.4	Beispiel: Thermometer	96
3.9	Power Management	98
3.10	Internes EEPROM	100
3.10.1	Deklaration einer Variablen im EEPROM	101
3.10.2	Lesen aus dem EEPROM	101
3.10.3	Schreiben ins EEPROM	101
3.11	Dynamische Speichernutzung	102
3.12	Verlagerung von Daten in den Programmspeicher	104
Literatur	105
4	Softwarearchitektur, Konzepte und Codierung	107
4.1	Überblick	107
4.1.1	Motivation	107
4.1.2	Sichten	108
4.2	Hardware-Abstraktion	109
4.3	Modularisierung und Zugriff auf Module	110
4.4	Zeitsteuerung	112
4.5	Wichtige Speicherkonzepte	115
4.5.1	Warteschlangen und Ringpuffer (FIFO)	115
4.5.2	Warteschlange mit dynamischen Datenstrukturen	118
4.5.3	Mehrere Warteschlangen in einem Programm	121
4.6	Zustandsautomaten (Statemachine)	122
4.6.1	Allgemeine Betrachtung	122
4.6.2	Beschreibung von Zustandsautomaten	124
4.6.3	Umsetzung von Zustandsautomaten auf Mikrocontrollern	127
Literatur	129
5	Kommunikationsschnittstellen	131
5.1	Das ISO/OSI Schichtenmodell	131
5.1.1	Schicht 1: Physikalische Bitübertragung	134
5.1.2	Schicht 2: Sicherungsschicht	145
5.2	Serielle Schnittstellen	152
5.2.1	Universal Asynchronous Receiver/Transmitter (UART)	152
5.2.2	Serial Peripheral Interface (SPI)	160

5.2.3 I ² C	173
5.2.4 Anbindung der seriellen Schnittstelle an USB	185
5.3 Bussysteme für embedded Schaltungen	186
5.3.1 CAN	186
5.3.2 MODBUS	196
5.4 Funkschnittstellen	201
5.4.1 Multiplexverfahren	201
5.4.2 Sensorknoten	202
5.4.3 Funkprotokolle im 2,4 GHz ISM-Band	204
5.4.4 Funkverbindung zwischen zwei Mikrocontrollern	211
Literatur	215
6 Sensortechnik	219
6.1 Systemtechnische Überlegungen	219
6.1.1 Abtastung	220
6.1.2 Quantisierung	223
6.1.3 Digitale Filterung	225
6.1.4 I/O-Steuerlogik	230
6.1.5 Abstraktion der I/O Pins	231
6.1.6 Ganzzahlarithmetik	232
6.2 Gyroskop	236
6.2.1 Beschaltung des L3GD20	237
6.2.2 Kommunikationsschnittstellen	238
6.2.3 Arbeitsmodi	240
6.3 MPL3115 digitaler Luftdrucksensor	250
6.3.1 Aufbau des MPL3115	251
6.3.2 Serielle Kommunikation	258
6.3.3 Power-Modi	260
6.3.4 Mess- und Lesemodi	260
6.3.5 Initialisierung des MPL3115-Sensors	261
6.4 Luftfeuchtigkeit SI7021	261
6.4.1 Aufbau des SI7021	262
6.4.2 Serielle Kommunikation	263
6.4.3 Berechnung der Temperatur und der relativen Luftfeuchtigkeit	267
6.4.4 Testbarkeit	270
6.5 HMC5883 Magnetfeldsensor	271
6.5.1 Aufbau des HMC5883	272
6.5.2 HMC5883 Messwerte lesen	276
6.5.3 Kalibrierung des Sensors	278
6.5.4 HMC5883 als elektronischer Kompass	280
6.5.5 Winkelberechnung mit dem CORDIC Algorithmus	281
6.6 Beschleunigungssensoren	283

6.6.1	Beschleunigungssensor ADXL312	283
6.6.2	MMA6525	297
6.7	Näherungssensoren	305
6.7.1	Ultraschall-Näherungssensoren	305
6.7.2	SI114x – optischer Näherungssensor	312
6.8	Strommessung mit dem LMP92064	325
6.8.1	LMP92064 Aufbau	325
6.8.2	Serielle Kommunikation	327
6.8.3	Messen mit dem LMP92064	327
6.9	Temperaturmessung mit dem TMP75	331
6.9.1	Sensorkonfigurierung	332
6.9.2	Serielle Schnittstelle	333
6.9.3	Temperaturmessung	335
6.9.4	Thermostatfunktion	337
Literatur	340
7	Vernetzbare integrierte Schaltkreise	343
7.1	PCF8574 Port Expander	343
7.1.1	Endstufe eines I/O Pins	344
7.1.2	Ausgang-Port-Modus	345
7.1.3	Eingang-Port-Modus	347
7.1.4	Interrupt-Modus	348
7.1.5	PCA9534	348
7.2	Festwertspeicher	349
7.2.1	Parallele Festwertspeicher	349
7.2.2	Serielle Festwertspeicher	350
7.3	Digital-Analog-Wandler	384
7.3.1	MCP48XX SPI-angesteuerte Digital-Analog-Wandler	384
7.3.2	PCF8591 I ² C-angesteuerter D/A- und A/D-Wandler	393
7.4	MCP41X1 digitale Regelwiderstände	402
7.4.1	Power-On-/Brown-Out-Reset Schaltung	403
7.4.2	Elektrischer Widerstand	404
7.4.3	Potentiometer-Register	404
7.4.4	Ansteuerfunktionen des Bausteins MCP4151	407
7.4.5	SPI-Kommunikation	408
7.4.6	Softwarebeispiel	410
7.5	MAX31629 Real Time Clock (RTC)	411
7.5.1	Zeitmessung	412
7.5.2	Alarmzeit	414
7.5.3	Temperaturmessung	415
7.5.4	Thermostat mit Alarmfunktion	416
7.5.5	I ² C Kommunikation	418

7.6	SI4840 Radio-IC	420
7.6.1	Bausteinbeschreibung	421
7.6.2	Auswahl des Frequenzbandes und Frequenzabstimmung	422
7.6.3	Initialisieren des Bausteins	423
7.6.4	Kommunikation mit dem Baustein	424
7.6.5	Sendersuche mit dem SI4840	430
Literatur		431
8	Anzeigen	433
8.1	Einführung	433
8.1.1	Displaylayout	433
8.1.2	Emissive und nicht emissive Anzeigen	434
8.1.3	Bildaufbau	438
8.1.4	Display Ansteuerung	439
8.2	Punktmatrix LCD-Display mit paralleler Ansteuerung	439
8.2.1	Struktur eines Displays mit einem KS0070B Controller	439
8.2.2	Befehlssatz	441
8.2.3	Bit Kommunikation	443
8.2.4	Generierung eines neuen Zeichens	445
8.2.5	Ausführen der Display Befehle ohne blockierendes Warten	446
8.3	Serielle Ansteuerung eines parallelen LCD-Displays	448
8.3.1	Display Ansteuerung über I ² C	449
8.3.2	Software Beispiel: Übertragung eines Datenbytes	450
8.4	DOGS102-6 Graphisches Display mit serieller Ansteuerung	453
8.4.1	Struktur des graphischen Displays DOGS 102-6	453
8.4.2	SPI-Kommunikation	455
8.4.3	Befehlssatz	457
8.4.4	Generierung eines Zeichens	461
Literatur		463
9	Beispielprojekt: Datenlogger	465
9.1	Aufbau der Modellrakete	465
9.2	Beschreibung des Projekts	466
9.3	Beschreibung der Software	467
9.3.1	Der Zustandsautomat	467
9.3.2	Loggen auf dem Flash	469
9.3.3	Daten auslesen	470
9.4	Auswertung	471
Literatur		472
Sachverzeichnis		473



Einleitung

1

Zusammenfassung

Was ist in diesem Buch zu finden? Ist es das Richtige für Ihr Problem? Lesen Sie dazu Kap. 1!

1.1 Was finden Sie in diesem Buch?

Das vorliegende Werk setzt an der Stelle an, an der Bücher über die Funktionsweise von Sensoren in der Regel aufhören. Neben einem Grundlagenteil, in dem die generelle Programmierung in C kompakt dargestellt wird, steht eine konkrete Anleitung über die Ansteuerung von beispielhaft ausgewählten Sensoren. Diese orientiert sich an einer Standard-Architektur und wird am Beispiel der Ansteuerung mit einem AVR-Mikrocontroller, wie er in vielen eingebetteten Systemen verwendet wird¹, bis auf Quellcodeebene beschrieben. Die Auswahl der Sensoren wurde so getroffen, dass sie möglichst repräsentativ für eine breite Anzahl von auf dem Markt befindlichen Komponenten stehen. Somit eignen sich die Beispiele nicht nur zur Nachprogrammierung sondern auch zum Schritt-für-Schritt Aufbau des Verständnisses und zur Übertragung auf eigene Anwendungen mit eigenen Bauteilen. Wegen der verwendeten Hardwareabstraktion lassen sich die allermeisten Beispiele in diesem Buch ohne großen Aufwand auf andere Prozessorfamilien übertragen. Das Buch gliedert sich mit diesen Überlegungen in folgende Teile:

- Eine Kurzeinführung in die Sprache C, die eher als Referenz zum schnellen Nachschlagen geschrieben wurde. Menschen, die noch nie in C programmiert haben, sollten die zahlreichen im Internet verfügbaren Tutorials und Anfängerkurse nutzen und zunächst einmal C üben und verstehen.

¹ So auch in der Arduino® Familie.

- Eine Kurzeinführung in den Mikroprozessor. Die Beispiele in diesem Buch sind ausnahmslos auf Prozessoren der ATmega8x Serie getestet und in der Regel auch auf den anderen 8-Bit AVR Prozessoren lauffähig, insbesondere setzen die Autoren den AT90CAN128 im Alltag ein. Atmel, das inzwischen an Microchip verkauft wurde, hat ein hervorragendes, instruktives Handbuch herausgegeben, außerdem beschreiben zahlreiche Autoren diese Familie, speziell auf der Seite www.mikrocontroller.net finden sich ausführliche Anleitungen und Antworten, mit denen die Leser das hier geschriebene vertiefen können. ATMEL wurde während der Endredaktion dieses Buches an Microchip Technology Inc. verkauft. Alle Produkte liefen zur Drucklegung noch weiter in Produktion und sind bei www.microchip.com zu finden.
- Eine Beschreibung wichtiger Softwaremechanismen, die in einem embedded Programm – unabhängig vom verwendeten Prozessor – eingesetzt werden können. Diese sind hardwareunabhängig beschrieben.
- Ein Kapitel über Bussysteme und Netzwerke. Dieses beschreibt sowohl die Kommunikationsschnittstellen UART, SPI und I²C (TWI) des Prozessors als auch, abstrakter, die Kommunikation über ausgewählte Bussysteme (CAN, Modbus) und Funkschnittstellen. Hiermit lässt sich eine Vielzahl von Sensornetzwerken aufbauen.

Der Kern des Buches besteht aus drei Kapiteln, die ausgewählte Sensoren, Anzeigen und weitere vernetzbare Bausteine beschreiben, beispielsweise Flash-Datenspeicher, Portexpander, AD-Wandler, Regelwiderstände und Radiocontroller. Die Beispiele werden mit den beschriebenen Bustreibern hardwareabstrakt eingeführt und lassen sich daher für andere Prozessorfamilien portieren. Zu den allermeisten im Alltag nutzbaren Sensortypen finden die Leserinnen und Leser hier praktische Beispiele, die die oft komplexen Bausteine auf wenige typische Applikationen reduzieren.

Am Ende schließt sich ein Kapitel an, in dem ein typisches Beispielprojekt und dessen Realisierung im Ganzen gezeigt werden.

1.2 Für wen ist dieses Buch geschrieben?

Das Buch richtet sich dementsprechend an Studierende in höheren Semestern, die beispielsweise im Rahmen ihrer Projekt- und Forschungsarbeiten konkrete Lösungen für vernetzte Sensorschaltungen suchen, sowie an Elektronikentwickler mit Messaufgaben. Die Schaltungsbeispiele und Programmierhinweise eignen sich aber auch für ambitionierte Amateure mit entsprechenden Grundfertigkeiten.

1.3 Welche Kenntnisse setzt das Buch voraus?

Obwohl das Buch in kurzen Zusammenfassungen einen Überblick über die Programmiersprache C und die physikalischen Messprinzipien der verwendeten Sensoren enthält, wendet es sich vorwiegend an Leser, die bereits Programmierkenntnisse in C besitzen und die Grundlagen der Messtechnik beherrschen. Zur Einarbeitung empfehlen sich zum Beispiel die Bücher [1] und [2]. Am Ende eines jeden Kapitels wird zudem umfangreich auf Literatur zum Weiterstudium verwiesen.

1.4 Warum beschreibt das Buch keinen Arduino?

Arduino ist eine feine und praktische Sache, um schnell zur Programmierung kleiner eingebetteter Systeme zu kommen. Die in diesem Buch beschriebenen Beispiele sind jedoch teilweise sehr komplex und werden von der Arduino-Bibliothek nicht unterstützt. In industriell eingesetzten Schaltungen wird man den Speicherplatz, den der Arduino-Bootloader benötigt, und die seriellen Schnittstellen eher einsparen. Selbstverständlich wird man die Beispiele auch mit Arduino betreiben können und sollte in diesem Fall die Funktionen zum Betrieb der seriellen Schnittstellen durch die Arduino-Funktionen ersetzen. Die Arduino Bibliothek bietet dazu eine vollständige Hardwareabstraktion der verwendeten Schnittstellen UART, SPI, I²C und der Portpins an [3].

1.5 Zusatzmaterialien

Die Autoren arbeiten ständig an den im Buch beschriebenen Programmen weiter. Die Quellcodes sind deshalb einem Überarbeitungsprozess unterworfen. Auf der Webseite <http://www.springer.com/de/> werden ausgewählte Quellen und Schaltpläne zum Download angeboten.

1.6 Disclaimer/Haftungsausschluss

Alle Schaltungs- und Codebeispiele in diesem Buch sind nach bestem Wissen und Gewissen erstellt. Dennoch wurden sie, der Lesbarkeit und Nachvollziehbarkeit der Beispiele geschuldet, lediglich prototypenhaft aufgebaut. Für sicheren und zuverlässigen Code eignen sie sich in keiner Weise, da die Autoren grundsätzlich auf das Auffangen von Fehlern oder auf Plausibilitätsprüfungen verzichtet haben um den Blick nicht von der eigentlichen Funktion abzulenken. Daher dürfen die hier gezeigten Schaltungen und Codebeispiele

nicht in militärisch oder kommerziell verwendeten Produkten eingesetzt werden. Der Betrieb der im Buch beschriebenen Schaltungen und Software kann potentiell gefährlich sein. Die Autoren und der Verlag lehnen jede Haftung ab, sollten Schaltungs- oder Codebeispiele im Betrieb nicht funktionieren oder Schaden an Sachen oder Personen anrichten.

1.7 Dank

Dieses Buch hat eine längere Vorgeschichte und wir danken den Kollegen in unserer Arbeitsgruppe und in der Fakultät für Mechanik und Elektronik, die uns mit Ideen, Anregungen oder Tipps bis zu diesem Buch begleitet haben. Insbesondere danken wir Petre Sora jr. und Nico Sußmann, die das Buch mit großer Sorgfalt Probe gelesen und wertvolle Hinweise gegeben haben. Unsere Ehefrauen sind beide selbst Ingenieurinnen und mit der Materie bestens vertraut. Wir danken ihnen für langjährige Begleitung, Unterstützung und Verständnis.

Literatur

1. Hering, E., Schönfelder, G., et al.: Sensoren in Wissenschaft und Technik, 1. Aufl. Vieweg & Teubner, Wiesbaden (2012)
2. Küveler, G., Schwoch, D.: Informatik für Ingenieure und Naturwissenschaftler 1 – Grundlagen Programmierung mit C/C++ – Großes C/C++ Praktikum, 5. Aufl. Vieweg, Wiesbaden (2006)
3. Arduino: Sprachreferenz (2017). <https://www.arduino.cc/en/Reference/HomePage>, Zugriffen: 2. März 2017



Einführung in die Programmiersprache C

2

Zusammenfassung

In diesem Kapitel wird ein kurzer Überblick über die Programmiersprache C gegeben. Das Kapitel ist bewusst stichwortartig gehalten und ersetzt für Programmieranfänger nicht das Studium eines C-Kurses. Vielmehr dient als Nachschlagehilfe. Zunächst werden Schlüsselworte und Symbole in C mit dem jeweiligen Verweis auf die entsprechenden Erklärungen aufgelistet, anschließend werden die Grundkonzepte: Variablen und Konstanten, Schleifen, Verzweigungen und Funktionen, komplexe Datentypen und Pointer erklärt und am Ende der Grundaufbau eines C-Programms beschrieben. Diese Einführung verzichtet bewusst auf eine vollständige Darstellung der Sprache, beispielsweise finden Sie hier keine Syntaxdiagramme. Dafür sei auf die einschlägige Literatur verwiesen. Stattdessen finden sich hier Tipps für den richtigen Umgang mit der Sprache C im Zielsystem AVR-Mikrocontroller.

2.1 Die Sprache C: Hintergrund und Aufbau

Die Programmiersprache C wurde 1971 in den Bell Laboratories der Firma AT&T von Dennis M. Ritchie als Überarbeitung der Sprache B mit dem Ziel entwickelt, das Betriebssystem UNIX maschinenunabhängig neu zu schreiben. Ken Thompson hatte dieses 1968 in Anfängen in Assemblersprache entwickelt. 1973 entstand dann von Thompson und Ritchie UNIX in C neu. AT&T stellte die Quellcodes öffentlich zur Verfügung und erreichte damit eine rasante Verbreitung von UNIX und C. C, das zunächst nur aus wenigen Seiten spezifiziert war, ist heute ein internationaler Standard (ANSI-C), der in mehreren Schritten (ISO/IEC 9899) bis derzeit 2011 (C11) weiterentwickelt wurde. Bjarne Stroustrup entwickelte C zu einer objektorientierten Programmiersprache C++ weiter, die seit 1998 in einer ISO Norm festgeschrieben wurde (ISO/IEC 14882:2014) [1]. UNIX wurde über die Jahre immer stärker kommerzialisiert, durch das quelloffene UNIX-Derivat LINUX

des damaligen Studenten Linus Thorvalds hat es seit 1991 insbesondere auf x86 Plattformen neue Verbreitung gefunden. C ist derzeit die weltweit am weitesten verbreitete Programmiersprache für eingebettete Systeme, nicht zuletzt deshalb, weil sie sehr hardwarenah gestaltet ist und daher zu effizientem Maschinencode führt. Da an vielen Stellen an der Überarbeitung von LINUX für funktional sichere Echtzeitsysteme gearbeitet wird, darf erwartet werden, dass sich auch das Betriebssystem noch weiter verbreiten wird.

Die Entwicklung von Software für eingebettete Systeme indes wird heutzutage in modellbasierter Form vorangetrieben. Modellierungssprachen wie Matlab/Simulink von Mathworks generieren C-Code, der so effizient ist, dass er direkt in eingebetteten Umgebungen lauffähig ist [2].

C ist eine imperative Programmiersprache. Programme werden von einem Startpunkt aus Befehl für Befehl abgearbeitet, zur Steuerung des Ablaufs werden so genannte Kontrollstrukturen (Verzweigungen, Schleifen) verwendet, absolute Sprunganweisungen (`goto` Abschn. 2.7.4) sind zwar in C grundsätzlich erlaubt, sollten aber nach Möglichkeit vermieden werden. Das prozedurale Paradigma kommt ohne diese Sprünge aus.

Um C-Programme übersichtlicher zu gestalten, strukturiert man sie in Funktionen und Modulen. Dazu später mehr.

Eine der Stärken von C besteht in seinem direkten Durchgriff auf die Hardware. So besitzen die gängigen Prozessoren einen Befehlsvorrat, der eine effiziente Übersetzung von C-Code in Maschinencode erlaubt. Diese Arbeit wird durch den Compiler erbracht, der in der Regel aus einer integrierten Entwicklungsumgebung aufgerufen wird, abgekürzt IDE (Integrated Development Environment). Folgendes Beispiel soll das verdeutlichen:

Beispiel

Die Operation `a = c+7;` weist der Variablen `a` den Wert der Summe aus dem Wert der Variablen `c` und der Konstanten 7 zu. Jede der verwendeten Variablen sei vom Typ Integer (16 Bit = 2 Byte). Nach der Übersetzung lautet der Maschinencode:

```
ldd r24, Y+1 ; Lade aus dem Speicher das niedrige Byte von c  
ldd r25, Y+2 ; Lade aus dem Speicher das höhere Byte von c  
adiw r24, 0x07 ; Addiere zum niedrigeren Byte von c die Zahl 7  
std Y+6, r25 ; Speichere Ergebnis (hohes Byte) an die Speicherstelle  
               von a  
std Y+5, r24 ; Speichere Ergebnis (niedriges Byte) an die Speicher-  
               stelle von a
```

Viele elementare Operationen in C lassen sich so direkt in Maschinencode übersetzen. Bei komplexeren Datentypen und Operationen, die nicht direkt vom Prozessor unterstützt werden, wird es komplizierter. Beispiele dazu gibt es später.

Der gesamte Ablauf der Übersetzung von C-Quellen findet sich in Abb. 2.1.

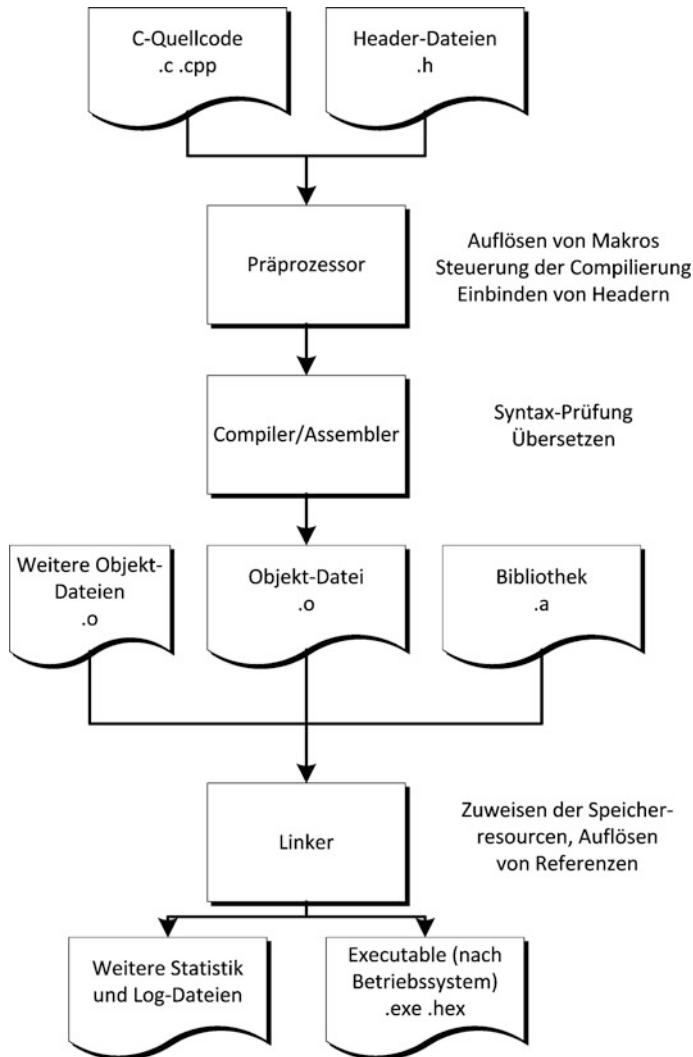


Abb. 2.1 Ablauf der Übersetzung eines C-Quellcodes bis zum lauffähigen Code

Header- und C-Dateien (Module) bilden dabei die Quellen (.c .cpp .h). Bereits übersetzte Module können nachträglich als Bibliotheken (.a) oder Binärdateien (.o) eingebunden werden. Auf einzelne Schritte des Übersetzungsvorgangs wird später noch eingegangen.

Die folgende Beschreibung der Sprache C ist eine sehr verkürzte Zusammenfassung aller in C verwendbaren Möglichkeiten. Für ein intensives Studium der Sprache sollte eines der zahlreich vorhanden Online-Tutorials (beispielsweise [3]) oder Bücher [4–9] verwendet werden.

2.2 Bezeichner, Schlüsselworte und Symbole in C

In C gibt es nur wenige Begriffe (*Schlüsselworte*), die der Sprache fest zugeordnet sind. Daneben existiert eine ganze Reihe von *Symbolen*, die als Operatoren verwendet werden. Funktionen, Variablen, Konstanten werden vom Programmierer selbst benannt, ihre Namen heißen *Bezeichner*.

2.2.1 Bezeichner

Bezeichner sind benutzerdefinierte Worte, die aus Klein-, Großbuchstaben, Ziffern und dem Sonderzeichen „_“ (Underscore) bestehen können, wobei das erste Zeichen ein Buchstabe oder ein Underscore sein muss. Sonderzeichen wie Umlaute und Satzzeichen sind dabei nicht erlaubt.

Wichtig in diesem Zusammenhang ist, dass C-Compiler zwischen Groß- und Kleinschreibung unterscheiden, also „case-sensitive“ sind. Der Bezeichner „Wert“ ist also von „WERT“ und „wert“ verschieden und bezeichnet eine andere Variable oder Funktion.

gültige Bezeichner: umu_foo Nummer_5 n_nummer

ungültiger Bezeichner: 1001_Nacht B_oder_C?! 1.Zahl

Mit einem Bezeichner werden Elementen wie z.B. Variablen und Funktionen eindeutige Namen zugeordnet.

2.2.2 Schlüsselworte

Die von C fest vorgegebenen Schlüsselworte teilen sich in fünf Gruppen auf:

1. Schlüsselworte für die Speichernutzung: `automatic`, `volatile`, `extern`, `static`, `register` (Abschn. [2.8.5](#))
2. Elementare Datentypen: `char`, `short`, `int`, `long`, `float`, `double`, `unsigned`, `(const)` (Abschn. [2.3](#))
3. Kontrollstrukturen `if`, `else`, `switch`, `case`, `break`, `continue`, `default`, `for`, `do`, `while`, `return`, `goto` (Abschn. [2.6](#))
4. Komplexe Datentypen: `enum`, `struct`, `union` (Abschn. [2.9.2](#) und folgende)
5. Weitere Schlüsselworte: `void` (Abschn. [2.8](#)), `typedef` (Abschn. [2.9.2](#)), `sizeof` (Abschn. [2.9.2](#))

2.2.3 Symbole

C nutzt eine ganze Reihe fest vorgegebener Symbole. Diese sind beispielsweise:

1. Arithmetisch/logische Operatoren: + - * / % < > ! ~ ^ & | = () und Kombinationen aus diesen
2. Symbole zur Strukturierung und Steuerung des Programms: { } ; // /* */ : ?
3. Symbole für den Zugriff auf komplexe Datenstrukturen und Pointer: & * [] . ->

Offensichtlich kommen hier Symbole doppelt vor, das heißt ihre Bedeutung ist abhängig vom sprachlichen Kontext, in dem sie benutzt werden.

2.2.4 Anweisungen

Die Sprache C besteht aus einer Folge von *Anweisungen*, die wir als auszuführende Aktionen interpretieren können. Anweisungen können Deklarationen, Ausdrucksanweisungen, Sprunganweisungen, Verzweigungen, Schleifen usw. sein. Auch Folgen von Anweisungen sind in C Anweisungen (Blockanweisung). Diese werden durch geschweifte Klammern zusammengefasst.

Beispiel

Die Anweisung

```
c = a * (b + c / 7);
```

weist der Variablen c den Wert der Berechnung zu. Die Zuweisung bildet einen *Ausdruck*, die rechte Seite, die Berechnung bildet einen Ausdruck, der je nach dem Wert der Variablen a, b und c einen Wert annimmt und seinerseits aus Ausdrücken besteht. Abstrakt kann man also sagen, Ausdrücke bestehen aus Ausdrücken und/oder Variablen, die mit Hilfe von Operatoren zusammengesetzt sind.

Ausdrucksanweisungen enden immer mit einem Semikolon (;).

2.3 Kommentare

Um das Verständnis eines Programmlistings zu verbessern, empfiehlt es sich, Kommentare direkt in den Quellcode einzufügen. Diese Kommentare sollten die Funktionsweise näher beschreiben und so die Fehlersuche oder Modifikation erleichtern (Inline-Dokumentation).

2.3.1 Einzeiliger Kommentar

```
// Dies ist ein Kommentar
```

Der einzeilige Kommentar beginnt mit dem doppelten Schrägstrich „//“ und endet automatisch am Ende der Zeile.

2.3.2 Mehrzeiliger Kommentar

```
/* Dieser Kommentar kann
   so lang sein wie er will */
```

Beim mehrzeiligen Kommentar wird der Anfang durch „/*“ und das Ende durch „*/“ gekennzeichnet.

2.3.3 Doxygen Kommentare

Spezielle Schlüsselworte in Kommentaren können von dem freien Softwarewerkzeug *Doxygen* genutzt werden, um eine automatische Dokumentation des Quellcodes zu erstellen.

Ein typischer Doxygen Filekommentar (am Beginn eines Modulfiles) könnte so aussehen:

```
/*!!
 \file      ad_basic.c
 \brief    Funktionen zum Auslesen der ADC-Wandlers.\n
 \see      Defines für die Hardwareabstraktion in hardware.h
 \author   Ansgar Meroth
 \version  V1.0 - 1.10.2015 Ansgar Meroth - HHN\n
 */
```

Jede Funktion kann dann noch beispielsweise wie folgt beschrieben werden:

```
/*!!
 \brief
 Gibt die Temperatur des ausgewählten Sensors zurück

 \return
 8 Bit Wert der Temperatur (unsigned char) zwischen -40 und 87,5°C.
 Die Temperatur berechnet sich aus (wert - 40)/2
 \par Eingangsparameter: unsigned char Nummer des Temperatursensors
 \par Beispiel:
```

```
(zur Demonstration der Parameter/Returnwerte)
\code
#define TEMPSENS0 0
#define TEMPSENS1 1
unsigned char result;
result = ucTemperaturGet(TEMPSENS0);
\endcode
*****
```

Doxxygen erzeugt aus diesen Kommentaren und aus der Analyse des Codes sehr gut lesbare Dateien in rich text, HTML und anderen Formaten und erleichtert so die Dokumentation des gesamten Codes. Dabei wird eine disziplinierte Kommentierung erzwungen, was auch im Code die Lesbarkeit steigert. Doxygen kann jedoch bereits aus unkommentiertem Code eine Dokumentation erzeugen. Die Abb. 2.2 zeigt, wie Doxygen aus den oben gezeigten Kommentarbeispielen eine rich text (MS Word) Datei generiert.

2.4 Typen, Variablen und Konstanten

2.4.1 Fundamentale Datentypen

Jede Variable hat einen *Datentyp*. Dieser legt den Speicherumfang fest und bestimmt, auf welche Weise Operatoren angewendet werden müssen.

Die Stärke von C besteht darin, dass neue Datentypen auf verschiedene Weise generiert werden können. Einige wenige Typen sind bereits vordefiniert, sie werden *fundamental* genannt. Wir unterscheiden bei den fundamentalen Datentypen die *Ganzzahltypen* und die *Gleitkommotypen*. Ein logischer Datentyp existiert seit dem ANSI Standard C99 als `_Bool`.

Bei den Ganzzahltypen kann der Vorsatz `unsigned` (ohne Vorzeichen) verwendet werden. Dadurch verschiebt sich der Wertebereich in den Zahlenbereich ≥ 0 (siehe Tab. 2.1).

Die Wortlänge von `int` ist nicht festgelegt. Kommt es auf definierte Wortlängen an und soll der Code portierbar sein, sollte man `int` als Datentyp vermeiden.

Tab. 2.1 Ganzzahlige Datentypen

Name	Größe	Wertebereich (signed)	Wertebereich (unsigned)
<code>char</code>	1 Byte (8 Bit)	$-128 \dots +127$	$0 \dots 255$
<code>short</code>	2 Byte (16 Bit)	$-32.768 \dots +32.767$	$0 \dots 65.535$
<code>int</code>	Wortlänge des Systems beim ATMega: 2 Byte, 16 Bit	$-32.768 \dots +32.767$	$0 \dots 65.535$
<code>long</code>	4 Byte (32 Bit)	$-2.147.483.648 \dots +2.147.483.647$	$0 \dots 4.294.967.295$

Datei-Dokumentation

ad_basic.c-Dateireferenz

Funktionen zum Auslesen der ADC-Wandlers.

```
#include <avr/io.h>
```

Funktionen

- unsigned char **temp** (unsigned char sens_no)
Gibt die Temperatur des ausgewählten Sensors zurück.
 - int **main** (void)
-

Ausführliche Beschreibung

Funktionen zum Auslesen der ADC-Wandlers.

Siehe auch:

Defines für die Hardwareabstraktion in hardware.h

Autor:

Ansgar Meroth

Version:

V1.0 - 1.10.2015 Ansgar Meroth - HHN

Dokumentation der Funktionen

unsigned char temp (unsigned char sens_no)

Gibt die Temperatur des ausgewählten Sensors zurück.

Rückgabe:

8 Bit Wert der Temperatur (unsigned char) zwischen -40 und 87,5°C. Die Temperatur berechnet sich aus (wert - 40)/2

Eingangsparameter: unsigned char Nummer des Temperatursensors

Beispiel:

(zur Demonstration der Parameter/Returnwerte)

```
1 #define TEMPSENS0 0
2 #define TEMPSENS1 1
3 unsigned char result;
4 result = ucTemperaturGet(TEMPSENS0);
```

Abb. 2.2 Beispiel eines Doxygen-Outputs als Word (rtf) Datei

Tab. 2.2 Gleitkommatypen

Name	Größe	Wertebereiche
float	4 Byte (32 Bit)	$-3,402823466 \cdot 10^{38} \dots -1,175494351 \cdot 10^{-38}$ 0 $+1,175494351 \cdot 10^{-38} \dots +3,402823466 \cdot 10^{38}$
double	8 Byte (64 Bit)	$-1,7976931348623157 \cdot 10^{308} \dots -2,2250738585072014 \cdot 10^{-308}$ 0 $+2,2250738585072014 \cdot 10^{-308} \dots +1,7976931348623157 \cdot 10^{308}$

Gleitkommazahlen bestehen aus einer Mantisse m und einem Exponenten e (siehe Tab. 2.2). Bei der Zahl $1.345 \cdot 10^3$ ist 1.345 die Mantisse und 3 der Exponent. 10 ist die Radix r, so dass $r = m \cdot b^e$ eine Gleitkommazahl darstellt. Man beachte die internationale Schreibweise mit dem Punkt anstelle des deutschen Kommas als Dezimalzeichen! Im C-Standard ist die interne Verwendung von Gleitkommazahlen weitgehend freigehalten.

Die in diesem Buch vorgestellte Software kommt ohne Gleitkommazahlen aus. Diese sind generell auf im Umfeld kleiner Systeme nicht zu empfehlen, da viele eingebettete Prozessoren, so auch die 8-Bit AVR Familie, keine Gleitkommaeinheit besitzen und arithmetische Ausdrücke zu erheblichem Codeaufwand führen. Beispielsweise führt die folgende Operation zu höchst unterschiedlichem Assembler-Code mit a, b, c vom Typ unsigned char

```
c=a*b;
17c: 90 91 1a 01    lds   r25, 0x011A
180: 80 91 0a 01    lds   r24, 0x010A
184: 98 9f          mul   r25, r24
186: 80 2d          mov   r24, r0
```

Wohingegen mit x, y, z vom Typ float...

```
z=x*y;
136: 80 91 0b 01    lds   r24, 0x010B
13a: 90 91 0c 01    lds   r25, 0x010C
13e: a0 91 0d 01    lds   r26, 0x010D
142: b0 91 0e 01    lds   r27, 0x010E
146: 20 91 16 01    lds   r18, 0x0116
14a: 30 91 17 01    lds   r19, 0x0117
14e: 40 91 18 01    lds   r20, 0x0118
152: 50 91 19 01    lds   r21, 0x0119
156: bc 01          movw  r22, r24
158: cd 01          movw  r24, r26
15a: 57 d3          rcall .+1710           ; 0x80a <__mulsf3>
15c: dc 01          movw  r26, r24
15e: cb 01          movw  r24, r22
```

```

160: 80 93 0f 01    sts  0x010F, r24
164: 90 93 10 01    sts  0x0110, r25
168: a0 93 11 01    sts  0x0111, r26
16c: b0 93 12 01    sts  0x0112, r27

```

... wesentlich mehr Code produziert wird und in Adresse 0x15a eine Funktion angesprungen wird, die ebenfalls in den Programmspeicher kopiert wird und dort weitere 488 Bytes schluckt.

- ▶ Im Umfeld von embedded-C auf kleinen Mikrocontrollern sollte auf die Verwendung von Gleitkommatypen verzichtet werden. In Abschn. 6.1.6 sind dazu weitere Betrachtungen angestellt.

In C gibt es keine eigenen Datentypen für druckbare Zeichen oder Zeichenketten. Ob ein Zeichen druckbar ist oder nicht hängt mit der Interpretation der druckenden Funktion oder der Ausgabeeinheit zusammen. Werden Zeichen in einem 8-Bit Code dargestellt (zum Beispiel einem ASCII Code¹), werden Zeichen-Variablen als `char` deklariert. Zeichenketten sind Arrays von `char` (Abschn. 2.9.1).

An dieser Stelle sei angemerkt, dass viele C-Compiler Synonyme für die fundamentalen Datentypen verwenden, diese sind eingängiger zu merken und haben sich weitgehend durchgesetzt. Der Gnu C-Compiler nutzt unter anderen folgenden Namenskonventionen:

```

typedef unsigned char uint8_t;
typedef unsigned short uint16_t;
typedef unsigned int uint16_t;

```

2.4.2 Deklaration von Variablen

In C gilt das Prinzip des „Declare before Use“. Alle Elemente müssen deklariert, also eingeführt werden bevor sie benutzt werden können. Von einer Variablen müssen dabei der Datentyp und der Name deklariert werden indem der Datentyp und anschließend gleich der Variablenname eingegeben wird. Damit sind Wertbereiche und die Speichernutzung festgelegt.

Beispiele

```
int iZahl;
```

Deklariert eine Variable vom Datentyp `int` und dem Namen `iZahl`. Der Variablen `iZahl` kann also ab dieser Deklaration ein Ganzzahlenwert zugewiesen werden. Das Präfix `i` kann verwendet werden, damit man sich den Wertebereich merken kann.

¹ Der ASCII Code ist ein 7-Bit Code, aber verschiedene Konventionen erweitern diesen auf einen 8-Bit Code.

```
double rZahl;
```

Deklariert eine Variable vom Datentyp `double` und dem Namen `rZahl`.

Jede Deklarationszeile wird mit einem Semikolon abgeschlossen. Haben mehrere Variable denselben Datentyp, können sie einfach als Liste mit einem trennenden Komma geschrieben werden.

Die folgende Zeile deklariert zwei Variable vom Datentyp `int` und den Namen `zahl1` und `zahl2`.

```
int zahl1, zahl2;
```

Bei der Deklarierung selbst werden noch keine Werte zugewiesen, es ist also nicht klar, welchen Wert die Variable zu diesem Zeitpunkt enthält.

Wird der Variablen bereits bei der Deklarierung ein Wert zugewiesen, so spricht man von Initialisierung.

Beispiele

```
unsigned int number = 47;
```

deklariert eine Variable vom Datentyp `unsigned int` und dem Namen `number`. Der Variablen wird sofort der Wert 47 zugewiesen.

2.4.3 Konstanten

Bei Konstanten unterscheidet man zwischen *literalen Konstanten* (Literale) und *symbolischen Konstanten*. Literale sind die Darstellung von Basistypen, beispielsweise:

- Integerzahlen 123 oder -14
- Hexadezimale Ganzzahlkonstanten: Diese werden mit dem Präfix `0x` ausgestattet. Die Zahl `0x10` entspricht also der dezimalen 16. Die Zahl `0xF` der Dezimalzahl 15.
- Gleitkommazahlen 1.234 oder `14.3e10` oder `-12e-3` (mit dem `e` wird der Exponent zur jeweiligen Radix angedeutet)
- Zeichenkonstanten 'a' (in einfachen Hochkommas, werden als `char` interpretiert)
- Zeichenkettenkonstanten: „Dies ist ein Text“ (in doppelten Hochkommas, werden vom Compiler automatisch mit einer binären 0 abgeschlossen)

Weitere Literale sind der Literatur zu entnehmen.

Symbolische Konstanten werden durch Bezeichner benannt. Zur Compilierzeit oder zur Laufzeit, je nach Sprache und Verwendung, werden diese Bezeichner durch ihren Wert ersetzt. In früheren C-Versionen gab es kein eigenes Konstrukt zur Definition symbolischer Konstanten. Stattdessen verwendete man die so genannte *Präcompileranweisung*

`#define`. Vor der Ausführung des Compilers lädt und evaluiert der *Präcompiler* die mit einem Gatter `#` gekennzeichneten Anweisungen. Durch

```
#define NAME Ausdruck
```

wird `NAME` einem Ausdruck zugewiesen und jedes Vorkommen von `NAME` durch den Ausdruck ersetzt. Man nennt `NAME` auch ein *Makro* und man kann damit mächtige Konstrukte aufbauen. Beispiele:

```
#define PI 3.1415926535
#define Laenge 32
#define ENDLESS while(1)
```

Der Ersetzungstext kann sich über mehrere Zeilen erstrecken, der Zeilensprung muss dann mit einem `\` angezeigt werden um den Inhalt der Folgezeile mit in die Anweisung einzuschließen, da Präcompileranweisungen kein Abschlusszeichen außer dem Zeilenwechsel kennen. An Makros kann man sogar Parameter übergeben, dies führt jedoch über die kurze Einführung hinaus und es sei auf ein Lehrbuch in C (s. Literaturverzeichnis) bzw. auf eine Kurzbeschreibung in Abschn. [2.8.2](#) verwiesen.

Neuere Compiler nutzen das Schlüsselwort `const` aus C++, möchte man portierbaren Code erzeugen sollte jedoch darauf verzichtet werden:

```
const int Laenge = 32;
const double PI = 3.1415926535;
```

Bei der Deklaration einer Konstanten mit `const` muss unmittelbar die Initialisierung erfolgen.

- ▶ Generell sollte im Quellcode auf die Verwendung von numerischen Konstanten verzichtet und dafür lieber symbolische Konstanten verwendet werden, dies gilt insbesondere für Feldlängen, physikalische Konstanten und andere konstante Ausdrücke.

2.5 Operatoren

Durch *Operatoren* werden 1 oder 2 *Operanden* miteinander verknüpft. Operatoren mit einem Operanden nennt man *unäre Operatoren*, solche mit zwei Operanden heißen *binäre Operatoren* (hat nichts mit dem Binärsystem zu tun!). C benutzt die Infix-Schreibweise, d.h. binäre Operatoren stehen zwischen den beiden Operanden (z.B. `a + b`). Der linke Operand wird als *lvalue* bezeichnet, der rechte Operand als *rvalue*. Operanden können dabei durchaus kompliziert werden, d.h. ihrerseits zusammengesetzt sein. Generell spricht

Tab. 2.3 Zuweisungsoperator

Operator	Beschreibung	Anmerkung
a = b	Der Zuweisungsoperator weist dem lvalue den Wert des rvalue zu, letzterer kann aus einem komplexen Ausdruck bestehen, ersterer muss eine Variable sein	

man von einem *Ausdruck*, als einem sprachlichen Konstrukt, das nach seiner Auswertung (engl. Evaluation) einen Wert als Ergebnis liefert.

Ein wichtiger Operator ist der *Zuweisungsoperator* (Tab. 2.3).

Im Folgenden sind wichtige Operatoren für verschiedene Aufgaben beschrieben:

2.5.1 Arithmetische Operatoren

Arithmetische Operatoren werden auf Zahlen angewendet. Tab. 2.4 gibt einen Überblick.

Tab. 2.4 Arithmetische Operatoren

Operator	Beschreibung	Anmerkung
i++	Der Post-Inkrementoperator (++) erhöht eine Variable um den Wert 1, nachdem die gesamte Anweisung abgearbeitet wurde	Ähnlich wie <code>i=i+1</code>
i--	Der Post-Dekrementoperator (--) erniedrigt eine Variable um den Wert 1, nachdem die gesamte Anweisung abgearbeitet wurde	Ähnlich wie <code>i=i-1</code>
++i	Der Pre-Inkrementoperator (++) erhöht eine Variable um den Wert 1, bevor die gesamte Anweisung abgearbeitet wird	
--i	Der Pre-Dekrementoperator (--) erniedrigt eine Variable um den Wert 1, bevor die gesamte Anweisung abgearbeitet wird	
a-b	Subtraktion (-)	
a*b	Multiplikation (*)	
a/b	Division (/)	
a%b	Modulo-Division (%)	Rest aus einer Ganzzahl-Division

Tab. 2.5 Logische Operatoren

Operator	Beschreibung	Anmerkung
<code>!a</code>	Logisches Nicht	Liefert 1 bei $a = 0$ oder 0 bei $a \neq 0^a$
<code>a&&b</code>	UND-Verknüpfung logisch	Liefert 1 wenn a und b von 0 verschieden, ansonsten 0
<code>a b</code>	ODER-Verknüpfung logisch	Liefert 1, wenn a oder b von 0 verschieden, ansonsten 0
<code>a < b</code>	Kleiner	Liefert 1 wenn a kleiner b sonst 0
<code>a > b</code>	Größer	Liefert 1 wenn a größer b, sonst 0
<code>a <= b</code>	Kleiner oder gleich	Liefert 1 wenn a kleiner oder gleich b sonst 0
<code>a >= b</code>	Größer oder gleich	Liefert 1 wenn a größer oder gleich b sonst 0
<code>a == b</code>	Gleich (Vergleichsoperator, Achtung! Zwei Gleichheitszeichen)	Liefert 1 wenn a gleich b, sonst 0
<code>a != b</code>	Ungleich	Liefert 1 wenn a ungleich b, sonst 0

^a Die Eins ist im Folgenden im Sinne von „ungleich Null“ zu verstehen

2.5.2 Logische Operatoren

Logische Operatoren werden auf boolesche Ausdrücke angewendet. Da in C ursprünglich kein logischer Datentyp existiert, der die Werte `true` oder `false` annehmen könnte, gilt: `false` entspricht dem Wert 0 (Null), `true` entspricht irgendeinem anderen Wert ungleich 0. Seit dem ANSI-Standard C99 existiert ein Datentyp `_Bool`. In der Standardbibliothek `stdbool.h` wird jedoch ein Makro (Abschn. 2.10) `bool` definiert, sowie die Makros `true` und `false`. Die logischen Operatoren sind in Tab. 2.5 aufgelistet.

2.5.3 Bitoperatoren

Bitoperatoren (Tab. 2.6) ermöglichen in C die Manipulation einzelner Datenbits innerhalb eines Ausdrucks. Zum Verständnis dieser Operatoren wird Kenntnis der booleschen Algebra vorausgesetzt, ein Blick in entsprechende Suchmaschinen kann denen Abhilfe verschaffen, die sich damit noch nicht beschäftigt haben. Im Kapitel über die AVR Programmierung sind einige Beispiele zu finden.

2.5.4 Operatoren für Speicherzugriffe

In Tab. 2.7 sind die Operatoren für Speicherzugriffe genannt. Diese werden für Zeiger und Zeigerarithmetik verwendet. Abschn. 2.9.5 gibt dazu nähere Erläuterungen.

Tab. 2.6 Bitoperatoren

Operator	Beschreibung	Anmerkung
a>>n	Rechts schieben	a wird um n Bits nach rechts geschoben (entspricht einer Division durch 2^n)
a<<n	Links schieben	a wird um n Bits nach links geschoben (entspricht einer Multiplikation mit 2^n)
~a	Bitweise Negation	Bitweise Negation von a
a&b	Bitweise UND-Verknüpfung	Bitweises UND von a und b
a b	Bitweise ODER-Verknüpfung	Bitweises ODER von a und b
a ^ b	Bitweises XOR (Exklusiv-ODER)	Bitweises XOR von a und b

Tab. 2.7 Operatoren für Speicherzugriffe

Operator	Beschreibung	Anmerkung
&a	Adressoperator: liefert die Speicheradresse von a	Abschn. 2.9.5
*a	Derefenzierungsoperator: Ist a die Adresse einer Variablen, liefert *a den Wert	Abschn. 2.9.5
sizeof (a)	Liefert die Größe in Byte, die a im Speicher belegt	Keine Funktion! Eine Ausnahme in C
.	Zugriffsoperator. In Strukturen und Unionen wird damit auf einzelne Elemente zugegriffen	Abschn. 2.9.5
->	Zugriffsoperator, wenn über einen Pointer zugegriffen werden soll	Abschn. 2.9.5
[]	Indizierungsoperator. Bei einem gegebenen Array a liefert a [i] das Objekt an der Position i	Abschn. 2.9.1

2.5.5 Weitere Operatoren

Die in Tab. 2.8 genannten Operatoren werden in den in der Tabelle genannten Abschnitten weiter beschrieben.

2.5.6 Assoziativität und Priorität von Operatoren

Analog zur bekannten „Punkt-vor-Strich-Regel“ aus der Mathematik, gibt es in C für Operatoren auch Prioritäten. Je weiter oben ein Operator in Tab. 2.9 steht, umso höher ist seine Priorität. Es wird empfohlen, Ausdrücke möglichst zu klammern.

Tab. 2.8 Weitere Operatoren

Operator	Beschreibung	Anmerkung
()	Typumwandlungsoperator: (typ) a liefert den Wert von a in dem von typ genannten Datentyp	Abschn. 2.3
()	Funktionsaufruf: fname () ruft die Funktion mit Namen fname auf	Abschn. 2.8
()	Klammerung: Ausdrücke in Klammern werden ausgewertet bevor der restliche Ausdruck ausgewertet wird	
,	Kommaoperator: Verknüpft zwei unabhängige Ausdrücke in einer Anweisung	
? :	Bedingungsoperator: Bedingung ? Ausdruck1 : Ausdruck2 Ist die Bedingung erfüllt wird Ausdruck 1 ausgewertet ansonsten Ausdruck 2 – einziger ternärer Operator in C	Abschn. 2.6.1

Tab. 2.9 Assoziativität von Operatoren

Art	Assoziativität	Operatoren
	Links nach rechts	() [] -> .
Unär	Rechts nach links	~ ! ++ -- sizeof (type)
Unär		- * (Referenz) &(Adresse)
Binär	Links nach rechts	* (mult)/(div) % (modulo)
Binär	Links nach rechts	+ (plus) – (minus)
Binär	Links nach rechts	<< >> (Schieben)
Binär	Links nach rechts	<<= >>= (logisch)
Binär	Links nach rechts	== != (logisch)
Binär	Links nach rechts	& (bitweise)
Binär	Links nach rechts	^ (bitweise)
Binär	Links nach rechts	(bitweise)
Binär	Links nach rechts	&& (logisch)
Binär	Links nach rechts	(logisch)
Binär	Rechts nach links	?:
Binär	Rechts nach links	= += -= *= /= %= etc.
Binär	Links nach rechts	, (Komma-Operator)

2.5.7 Typumwandlung

C benötigt das Wissen um einen Datentypen um eine Operation sinnvoll durchführen zu können. Daher werden Variablen immer deklariert. Mitunter ist es jedoch notwendig, zwischen verschiedenen Datentypen zu mischen, indem beispielsweise eine ganze Zahl (char, integer, short, long) mit einer Festkommazahl (double, float) multipliziert wird. Folgendes Codebeispiel zeigt das:

```
int a = 5;
double x = 1.5;
double result;
result = a * x ;
```

Um diese Anweisung ausführen zu können, muss der Compiler zunächst alle an der Operation beteiligten Variablen auf einen gemeinsamen Datentyp bringen. Dies geschieht durch die so genannte *implizite Typumwandlung* (engl. *implicit cast*). Die Regel lautet, dass der jeweils „schwächere“ Typ in den stärkeren umgewandelt wird. In diesem Fall wird also `a` zunächst als `double` interpretiert und dann die Operation `*` ausgeführt. Das Ergebnis ist dann vom Typ `double`.

Die implizite Typumwandlung kann auch Fallstricke beinhalten. Folgendes Beispiel macht es deutlich:

```
int a = 1;
int b = 3;
double result;
result = a / b ;
```

Das Ergebnis dieser Operation, das in `result` abgespeichert ist, lautet `0.0`! Der Grund dafür ist einfach. Die beiden ganzzahligen Operanden werden ganzzahlig geteilt, Ergebnis ist `0`, da Nachkommastellen bei Ganzahloperationen abgeschnitten werden. Erst bei der Zuweisung wird dann das Ergebnis implizit in `double` umgewandelt. Das ist eigentlich keine Überraschung.

Um sich vor der impliziten Typumwandlung zu schützen, gibt es in C die *explizite Typumwandlung*. Hierbei wird in runden Klammern der Zieltyp angegeben, zum Beispiel:

```
int a = 2;
int b = 3;
double result;
result = (double)a / (double)b;
```

`a` und `b` werden hier explizit in `double` umgewandelt und danach korrekt geteilt. Im Prinzip hätte es sogar gereicht, nur `a` oder `b` umzuwandeln, der andere Operand wäre dann implizit umgewandelt worden. Explizite Typumwandlungen kann man erweiternd durchführen (z. B. `int → double`) oder einschränkend (z. B. `double → int`), wobei dann Information verloren geht.

2.6 Kontrollstrukturen

Kontrollstrukturen dienen der Steuerung des Programmablaufs jenseits der einfachen Sequenz. C kennt als Kontrollstrukturen die Verzweigungen und die Schleifen:

2.6.1 Verzweigung (Auswahl)

Die Verzweigung wird auch bedingte Anweisung, if-Anweisung oder Auswahl genannt. **Bedingung** steht für einen Ausdruck vom Datentypen Integer. Der Ausdruck wird be-

rechnet und als Wahrheitsausdruck interpretiert, d. h. liefert er einen von 0 verschiedenen Wert, wird die Anweisung ausgeführt, andernfalls übersprungen und gegebenenfalls der Alternativzweig durchlaufen.

Eine Verzweigung sieht beispielsweise so aus:

```
if (Bedingung)
{
    Anweisung_1;
    ...
    Anweisung_n;
}
else
{
    Anweisung_1;
    ...
    Anweisung_n;
}
```

Der erste Anweisungsblock wird nur durchlaufen, wenn die Bedingung erfüllt ist. Der untere Block wird nur durchlaufen, wenn die Bedingung NICHT erfüllt ist. Dieser zweite Block, der mit `else` eröffnet wird, kann auch weggelassen werden. Als Bedingungen können beliebige Ausdrücke verwendet werden, die 0 oder nicht 0 liefern, beispielsweise

```
(a > 0)
(wert == 128)
(4 * a != b)
```

Falls nur Einzelanweisungen in den Alternativen vorgesehen sind, können die Klammern weggelassen werden:

```
if (Bedingung) Anweisung_1;
else Anweisung_2;
```

Abb. 2.3 zeigt die Verzweigung als Programmablaufplan (Flussdiagramm) nach DIN 66001 einmal mit und einmal ohne Alternative.

Eine elegante Alternative zur einfachen Verzweigung ist der *Bedingungsoperator*. Er besteht aus nur einer Zeile (`Bedingung ? Ausdruck1 : Ausdruck2`), wie im folgenden Beispiel gezeigt:

```
return (a<b)? a : b;
```

Die Bedeutung des Operators ist die folgende: Ist die Bedingung erfüllt, wird `Ausdruck1` evaluiert, ansonsten `Ausdruck2`. Dies kann natürlich innerhalb eines anderen Ausdrucks geschehen:

```
result = 4*((number1>number2)?number1:number2)+9;
```

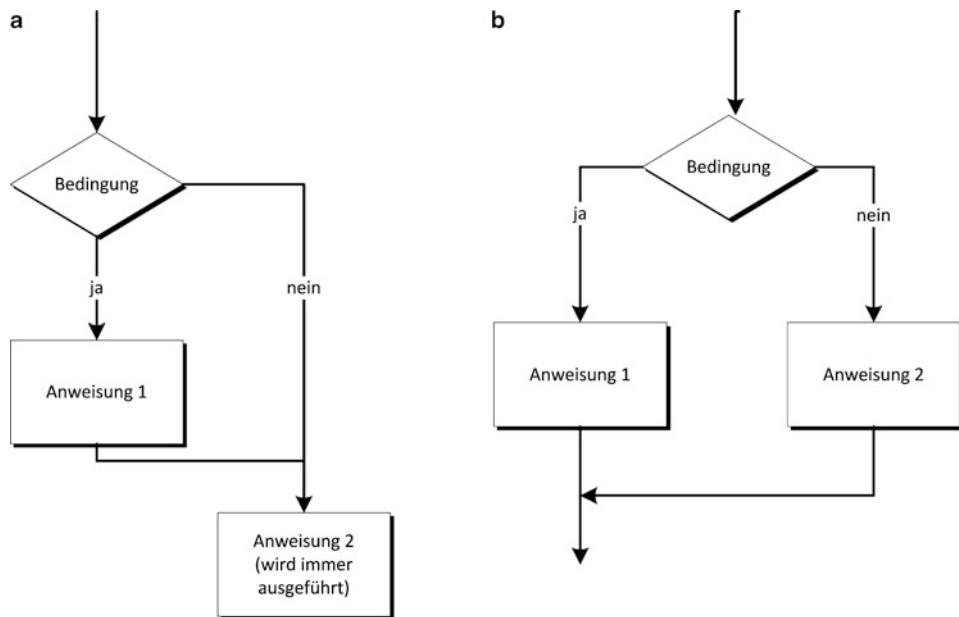


Abb. 2.3 Programmablaufplan für eine Verzweigung ohne (a) und mit else (b)

2.6.2 Fallstricke

Es lohnt sich, auch um den Preis eines längeren Quellcodes, immer mit geschweiften Klammern zu arbeiten, insbesondere für Anfänger! Im folgenden Beispiel wird Anweisung 2 beispielsweise IMMER ausgeführt, unabhängig von a.

```
if (a < 0) Anweisung_1; Anweisung_2;
```

Ein weiterer beliebter Fehler ist es, einen Vergleich mit einem einfach = statt mit dem doppelten == zu schreiben. Hier findet nämlich dann eine Zuweisung statt, ist der zugewiesene Wert größer 0 ist damit die Bedingung immer erfüllt:

```
if (a = 10)
{
    Anweisung_1;
}
```

führt dazu, dass Anweisung 1 immer ausgeführt wird!

Besser (und in sicherem Code verpflichtend gefordert) schreibt man bei Vergleichen die Konstante links, damit würde der Compiler in jedem Fall eine irrtümliche Schreibweise erkennen und abbrechen:

```
if (10 == a) Anweisung_1;
```

2.6.3 Mehrfachverzweigung

Verzweigungen können auch so miteinander kombiniert werden, dass sich faktisch eine Mehrfachauswahl ergibt. Beispiel:

```
if (a < 0)
{
    Anweisungsblock_1;
}
else if (a < 5)
{
    Anweisungsblock_2;
}
else if (a < 10)
{
    Anweisungsblock_3;
}
else
{
    Anweisungsblock_4;
}
```

Anweisungsblock 1 wird nur ausgeführt, wenn $a < 0$ ist. Falls $a \geq 0$ ist (und nur dann!) wird die nächste Bedingung $a < 5$ abgefragt. Nur wenn keine Bedingung erfüllt ist, in diesem Fall also $a \geq 10$ ist, wird Anweisungsblock 4 abgefragt. Bei solchen Mehrfachverzweigungen sollte man immer ein unkonditioniertes `else` einsetzen, um keine nicht abgedeckten Fälle zu erzeugen.

Alternativ zum den oben gezeigten Beispielen gibt es die `switch`-Anweisung. Der in den Klammern angegebene Ausdruck, der zur Auswahl des Anweisungsblocks in der `switch`-Anweisung dient, muss vom Datentyp Integer sein. Stimmt der Ausdruck mit dem Wert 1 überein, so wird der Anweisungsblock 1 ausgeführt. Nimmt der Ausdruck den Wert 2 an, dann wird Anweisungsblock 2 bearbeitet usw. Stimmt der Ausdruck mit keinem Wert überein, dann wird der `default`-Anweisungsblock ausgeführt. Fehlt der `default`-Anweisungsblock, dann wird bei keiner Übereinstimmung der gesamte Block übersprungen.

```
switch (Ausdruck)
{
    case Wert_1: Anweisung_1; break;
    case Wert_2: Anweisung_2; break;
    ...
    case Wert_n: Anweisung_n; break;
    default: Anweisung_n+1;
}
```

Das Schlüsselwort `case` dient hier als Sprungmarke, mit anderen Worten: Nach Auswertung des Ausdrucks wird die Abarbeitung an der Stelle weitergeführt, an der ein entsprechender Wert hinter dem `case` steht. Dabei gibt es aber Fallstricke zu beachten. Folgendes Beispiel verdeutlicht das:

```
switch(a)
{
    case 1: Anweisung_1; break
    case 5: Anweisung_2;
    case 10: Anweisung_3;
    default: Anweisung_4;
}
```

Ist vor der switch-Anweisung `a=1`, dann wird `Anweisung_1` ausgeführt, danach wird der Anweisungsblock verlassen. Ist hingegen `a=5`, werden `Anweisung_2`, `Anweisung_3` und `Anweisung_4` hintereinander ausgeführt, bei `a=10` werden `Anweisung_3` und `Anweisung_4` ausgeführt. Wäre `a` gleich einer anderen Zahl würde nur `Anweisung_4` ausgeführt (`default`). Grund für das Verhalten ist das fehlende `break` nach den Sprungmarken bei 5 und 10. Grundsätzlich ist die switch-Anweisung also mit Sorgfalt zu verwenden!

2.7 Schleifen

Schleifen werden in der Programmierung eingesetzt, um einen Programmabschnitt mehrmals auszuführen solange eine Bedingung erfüllt ist. Man unterscheidet dabei Schleifen, die mindestens einmal durchlaufen werden und somit die Bedingung erst am Ende des Schleifenkörpers prüfen, und solche, die gleich zu Beginn des Schleifenkörpers entscheiden, ob die Schleife überhaupt durchlaufen wird.

2.7.1 Kopfgesteuerte Schleifen

Der Anweisungsblock (Schleifenkörper) der kopfgesteuerten Schleife oder while-Schleife wird ausgeführt, solange die Bedingung im Schleifenkopf erfüllt ist. Allerdings wird diese Schleife überhaupt nicht ausgeführt, sollte die Schleifenkopf-Bedingung zu Beginn nicht erfüllt sein.

```
while(Bedingung)
{
    Anweisung_1;
    ...
    Anweisung_n;
}
```

Der Bedingungsausdruck kann dabei jeder Ausdruck sein, der den Wert 0 (nicht erfüllt) oder nicht 0 (erfüllt) annehmen kann, beispielsweise ein Vergleich, oder ein Ausdruck, der aus booleschen Verknüpfungen dieser Ausdrücke zusammengesetzt sind. Kopfgesteuerte Schleifen eignen sich immer dann, wenn die Nichterfüllung der Bedingung ein Fehlverhalten im Schleifenkörper erzeugen würde. Eine typische Anwendung ist, zu verhindern, dass ein Speicherobjekt, auf das im Schleifenkörper zugegriffen wird, nicht existiert und dies in der Bedingung abgefragt wird. Dazu später mehr. Wichtig ist zu beachten, dass sich die Bedingung im Laufe der Schleife ändern kann, ansonsten handelt es sich um eine Endlosschleife:

```
while(1)
{
    Anweisung;
}
```

In diesem Fall wird die Schleife nie abgebrochen. Der Schleifenkörper enthält in einer nicht endlosen Schleife also immer eine Iterationsanweisung oder die Ermittlung eines Wertes, der im Bedingungsausdruck verwendet wird.

Grundsätzlich kann jede Schleife mit der Anweisung `break` auch aus dem Schleifenkörper heraus abgebrochen werden, dies ist aber im Sinne der Übersichtlichkeit des Programms nur mit äußerster Vorsicht zu gebrauchen.

2.7.2 Fußgesteuerte Schleifen

Fußgesteuerte Schleifen oder *do-while*-Schleifen werden immer dann eingesetzt, wenn der Anweisungsblock (Schleifenkörper) mindestens einmal durchlaufen werden muss, beispielsweise um eine Bedingung abzufragen. Die Anweisungen nach einmaliger Ausführung solange wiederholt, wie die Bedingung, die im Fuß der Schleife abgefragt wird, erfüllt ist.

```
do
{
    Anweisung_1;
    ...
    Anweisung_n;
} while(Bedingung);
```

Die Flussdiagramme für kopf- und fußgesteuerte Schleifen sind in Abb. 2.4 dargestellt.

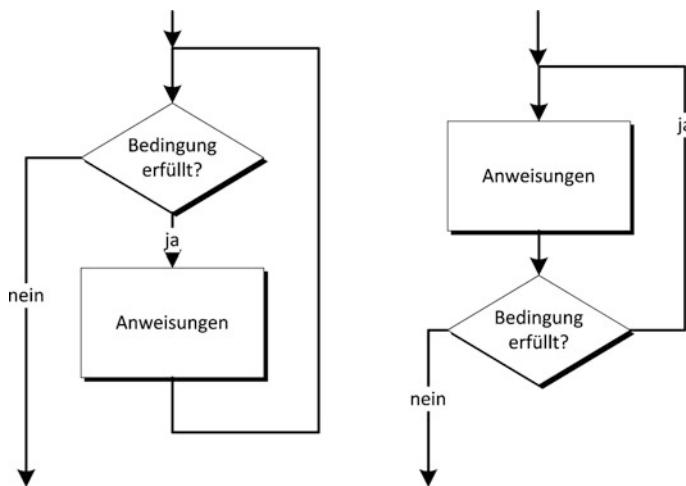


Abb. 2.4 Kopf- (a) und fußgesteuerte (b) Schleifen im Flussdiagramm

2.7.3 Zählschleifen

Eine Zählschleife bzw. *for*-Schleife verwendet man, wenn zu Beginn des Programmablaufs die Anzahl der Schleifendurchläufe bekannt ist. Im Prinzip handelt es sich um eine kopfgesteuerte Schleife, deren Iterationsausdruck mit im Schleifenkopf steht.

```
for(Startbedingung; Laufbedingung; Iterationsanweisung)
{
    Anweisung_1;
    ...
    Anweisung_n;
}
```

Auch die Zählschleife kann endlos ausgeführt werden, wie in folgendem Codebeispiel gezeigt:

```
for( ; ; )
{
    Anweisung_1;
    ...
    Anweisung_n;
}
```

2.7.4 Sprünge

In C sind auch absolute Sprünge möglich, die mit einem `goto` ausgeführt werden. Hierzu muss eine Sprungmarke definiert werden. Solche Sprünge sind jedoch grundsätzlich zu vermeiden und werden auch nicht benötigt.

2.8 Funktionen

Funktionen sind Programmteile, die eine bestimmte Aufgabe erledigen sollen. Jede Funktion steht grundsätzlich allen anderen Funktionen eines Programms zur Verfügung und kann in einem Ausdruck ausgewertet werden.

Aus der Mathematik kennt man die Schreibweise $y = f(x)$. Sie gehen von einem Wert x aus, dieser wird gemäß der Funktion verändert z. B. (x^2) . Das Ergebnis der Berechnung wird dann y zugewiesen.

Nichts anderes machen Funktionen in C. Eine oder mehrere Variablen werden an die Funktion übergeben (Übergabeparameter), dort gemäß der Anweisung verändert und dann der Ergebniswert (Rückgabeparameter) einer anderen Variablen zugewiesen. Im Unterschied zur Mathematik können in C nicht nur Zahlen verarbeitet werden, sondern auch andere Objekte. Deshalb ist es wichtig, den Datentyp der Übergabe- und Rückgabeparameter genau zu definieren. Selbstverständlich kann man auch Funktionen in Ausdrücken ohne Zuweisung verwenden, zum Beispiel sei die Funktion `sqrt(x)` vom Typ `double` und berechne die Quadratwurzel einer Zahl x , dann kann die Funktion in einem Ausdruck

```
y = a * sqrt(x) + 3;
```

verwendet werden oder sogar direkt als Übergabeparameter für eine andere Funktion:

```
printf(sqrt(x * x + y * y));
```

Grundsätzlich arbeitet C nach dem Prinzip des **call by value**, das heißt, dass nicht die Variable selbst übergeben wird, sondern nur ihr Wert. Eine aufgerufene Funktion kann also in der Regel nicht die ursprüngliche Variable verändern.

2.8.1 int main()

`main()` ist die wichtigste Funktion eines C-Programmes. Sie ist Anfangs- und Endpunkt der Ausführung. Sie muss auch nicht deklariert werden, ihr Name und ihre Funktion sind von Anfang an festgelegt. `main()` ist vom Typ `int`, gibt also einen Wert zurück.

2.8.2 Definition und Deklaration

Bei der Entwicklung eigener Funktionen ist folgendes Schema zu beachten:

Mit dem Funktionsprototyp wird jede Funktion *deklariert*. Dabei wird die Funktion formal vorgestellt. Auch für Funktionen gilt das Prinzip des **Declare before Use**. Deshalb sollte der Funktionsprototyp auch am Anfang des Quellcodes stehen.

Die Form eines Funktionsprototyps lässt sich an folgendem Beispiel ablesen:

```
double length(double x, double y);  
void wait (int t);
```

Die Funktion `length()` hat zwei Eingabeparameter `x` und `y`. Ihr Typ ist `double`, mit anderen Worten sie nimmt einen Fließkommawert an, nachdem sie ausgeführt wurde. `wait()` hingegen ist vom Typ `void`, und liefert keinerlei Wert zurück, kann also auch nicht in einem Ausdruck verwendet werden, sondern dient dazu, immer wieder benötigte Programmteile zu bündeln.

Erst nach der Deklaration kann die Funktion verwendet werden. Dazu muss sie noch nicht definiert sein. Erst beim Binden (link) wird die eigentliche Funktion dem Aufruf zugeordnet, sie kann dann aus einer anderen Quelle kommen, in der Regel einen Bibliothek. Die Definition ist die eigentliche Programmierung der Funktion als Folge von Anweisungen und Kontrollstrukturen. Diese werden durch geschweifte Klammern `{ }` (Blockklammern) zusammengefasst. Am Ende bestimmt die Anweisung `return`, welcher Wert dann wieder zurückgegeben wird. Die Definition der Funktion kann an beliebiger Stelle im Quelltext stehen.

Mit der oben bereits beschriebenen Funktion `sqrt()` könnte also `length()` wie folgt ausgeführt werden:

```
/******************************************/  
/* Liefert die Länge eines Vektors zurück */  
/******************************************/  
double length(double x, double y)  
{  
    double result;  
    result = sqrt(x * x + y * y);  
    return result ;  
}
```

Das Schlüsselwort `return` bricht die Funktion ab und liefert den Ausdruck zurück, der hinter `return` steht. Dieser kann durchaus komplexer Natur sein. Funktionen mit dem Typ `void` (leer) benötigen kein `return`.

Generell gilt also:

```
Speicherklasse Datentyp Funktionsname (Parameterliste)
{
    Deklarationen der lokalen Variablen
    Anweisungen
}
```

Die Speicherklasse wird in Abschn. [2.8.5](#) erklärt.

Bei sehr kurzen Funktionen kann man seit C99 dem Compiler mit dem Schlüsselwort `inline` empfehlen, den Funktionsaufruf durch den Funktionsrumpf zu ersetzen. Dadurch spart man sich, wenn kurze Funktionen sehr oft aufgerufen werden sollen, das Sichern des Aufrufkontextes, also der Register und des Programmzählers, was sonst bei Sprüngen in Funktionen durch den Prozessor durchgeführt werden muss, um nach dem Abarbeiten der Funktion wieder an die Aufrufstelle zurückkehren zu können. Ein Beispiel soll das verdeutlichen:

```
static inline int max(int a, int b)
{
    return (a<b) ?b:a;
}
```

Der Aufruf erfolgt wie bei einer normalen Funktion.

Statt des Schlüsselwortes `static` kann auch die Funktion als `extern` in einer Headerdatei (siehe Abschn. [2.8.4](#) und [2.8.5](#)) deklariert werden, im Header sieht das dann so aus:

```
extern inline int max(int a, int b);
```

Und im Quellcode:

```
inline int max(int a, int b)
{
    return (a<b) ?b:a;
}
```

Eine weitere Alternative zur Inline-Funktion ist die Definition von Makros mit Parametern. Makros wurden in Abschn. [2.4.3](#) bereits eingeführt. Makros mit Parametern werden vom Präcompiler vor dem Übersetzungs vorgang aufgelöst, beispielsweise:

```
#define MAX(a,b) (a<b) ?b:a
```

Die Verwendung solcher Konstrukte unterliegt jedoch großer Vorsicht und sollte von Anfängern vermieden werden. Weiterführende Erklärungen liefern die im Anhang genannten Quellen zur Sprache C.

2.8.3 Sichtbarkeit und Lebensdauer von Variablen in Funktionen

Generell gilt in C, dass eine Variable nur dort gültig ist, wo sie deklariert wurde. Eine Variable, die innerhalb einer Funktion oder eines anderen Anweisungsblocks deklariert wird, wird auch als *lokale* Variable bezeichnet, d. h. sie ist nur innerhalb der Funktion sichtbar und gültig. Dabei spielt es keinerlei Rolle, ob der Name der Variablen schon jemals anderswo deklariert war. Sobald der Anweisungsblock beendet ist, wird die Variable vernichtet und ihr Inhalt gelöscht. Das folgende Beispiel verdeutlicht dies ohne Anspruch auf Vollständigkeit:

```
/* Modul Beispiel.c */
int i,j,k; //i,j und k sind global, d. h. überall sichtbar

void function_1(int j, int m)
    //j und m sind lokale Parameter, das globale j wird überdeckt
{
    int k, l; //k und l sind lokal, das globale k wird überdeckt
    i = 5;    //i ist die globale Variable
    j = 9;
}
void function_2 (int m) //m ist hier lokaler Parameter
{
    int l; //l ist hier ein anderes l als in function_1
    int n; //n ist hier nur lokal gültig
    m = 18;
}
int main()
{
    int m = 1; //dieses m ist innerhalb main() gültig, nicht aber
               //in den Funktionen
    int o = 8;
    function_2(o); //Inhalt von o wird in function_2 kopiert
                   //(call by value)
}
```

Variablen, die außerhalb jedes Anweisungsblocks deklariert werden, sind global gültig, solange sie nicht überdeckt werden. Außerhalb von Modulen kann man globale Variablen deklarieren (in Headerfiles), die dann über mehrere Module gültig sind. Siehe dazu Abschn. 2.8.5

In früheren C-Versionen mussten Variablen immer am Anfang eines Anweisungsblocks deklariert werden, seit dem C99 Standard ist dies nicht mehr nötig. Dennoch empfiehlt es sich, aus Kompatibilitäts- und Lesbarkeitsgründen, möglichst auf das „wilde“ Deklarieren von Variablen zu verzichten.

2.8.4 Header

Bibliotheksfunktionen, die bei Bedarf aus Bibliotheken geladen werden, erweitern den Funktionsumfang. Diese werden über Bibliotheksfiles im Bindeprozess geladen (siehe später) während die Funktionsdeklaration über Header-Dateien (Datei-Endung .h) eingebunden werden.

Im Allgemeinen sind dies Grundfunktionen, die immer wieder gebraucht werden. Ein Beispiel: Das Makro `sei()`, das die Behandlung von Interrupts ermöglicht, wird durch das Einbinden der Header-Datei `interrupt.h` zur Verfügung gestellt. Aber auch mathematische Funktionen wie Sinus und Cosinus werden durch Bibliotheksfunktionen verfügbar. Weiterhin werden Header verwendet, um die Schnittstellen von Modulen (d. h. deren Funktionsaufrufe und Makros) anderen Modulen zur Verfügung zu stellen. Für jedes Modul (.c Datei) wird also eine gleich benannte .h Datei mit den zu veröffentlichten Deklarationen erstellt.

Headerdateien werden wie folgt eingebunden:

```
#include <avr/io.h>
#include "Modul1.h"
```

Der Name der Datei, die einbezogen werden soll, steht in den spitzen Klammern, wenn sie in bestimmten, dem Präprozessor bekannten Verzeichnissen (Include-Pfad) abgelegt ist. Steht der Name in „“, so wird der Pfadname des Projekts benutzt. Mit der Compileranweisung `-I` kann ein eigener Suchpfad eingegeben werden, dies geschieht anwenderfreundlich im Eigenschaften-Dialog des Projekts (Include-Pfade). Die `#include`-Anweisung ist eine „Präprozessor-Anweisung“, d. h. sie wird vor dem eigentlichen Übersetzungsvorgang abgearbeitet.

Im Allgemeinen funktioniert die `#include`-Anweisung wie ein Copy/Paste: der Text aus der Header-Datei wird an dieser Stelle in den Quelltext eingefügt und wird damit integraler Bestandteil des Programmes. Damit ist natürlich möglich, dass in einer Headerdatei weitere `#include`-Anweisungen stehen, gefährlich wird es dann, wenn über Umwege dann eine Datei rekursiv immer wieder von sich selbst eingebunden wird. Dies kann man ganz einfach durch folgende Konstruktion verhindern:

```
/** Treiber.h ***/
#ifndef TREIBER_H
#define TREIBER_H
// hier stehen die Inhalte der Header-Datei drin
#endif //TREIBER_H
```

Die Schlüsselworte `#ifndef` (falls nicht definiert) und dem abschließenden `#endif` bilden in einer Headerdatei eine sinnvolle Klammer um zu verhindern, dass bei einem versehentlichen doppelten Einbinden Zirkelreferenzen entstehen, weil alles, was in der

Klammerung steht, nur dann eingelesen wird, wenn das Makro (hier `TREIBER_H`) noch nicht definiert ist. Bei der erstmaligen Einbindung wird `TREIBER_H` definiert.

- Man beachte, dass im Header selbst Funktionen nicht definiert sondern nur deklariert werden dürfen!

2.8.5 Die Schlüsselworte `extern`, `volatile` und `static`

Mit dem Schlüsselwort `extern` kann man in einer Headerdatei eine Variable deklarieren, ohne dass dafür Speicherplatz reserviert wird. Folgendes Beispiel soll dies verdeutlichen:

In einer Treiberdatei soll ein Status als globale Variable übergeben werden. Dies könnte in der Datei Treiber.c folgendermaßen realisiert sein:

```
/*** Treiber.c ***/
#include Treiber.h
unsigned char Status;

void setStatus(unsigned char s)
{
    Status = s;
}
```

In der Datei Treiber.h steht nun eine extern-Deklaration von Status

```
/*** Treiber.h ***/
#ifndef TREIBER_H
#define TREIBER_H
extern unsigned char Status;
#endif //TREIBER_H
```

Sobald eine andere Funktion, die diesen Treiber nutzt, die Datei Treiber.h einbindet, ist durch das Schlüsselwort `extern` sichergestellt, dass keine neue Variable `Status` angelegt werden soll. Erst beim Binden mit der Bibliothek oder dem Object-File, das Treiber.c enthält, weist der Linker der nun offenen Referenz von `Status` den entsprechenden Speicherplatz zu.

Die Verwendung von externen globalen Variablen sollte nur mit äußerster Vorsicht geschehen. Im genannten Beispiel könnte eine so genannte „getter“-Funktion einen besseren Dienst leisten:

```
/*** Treiber.c ***/
#include Treiber.h
unsigned char Status;
```

```
void setStatus(unsigned char s)
{
    Status = s;
}
unsigned char getStatus(void)
{
    return Status;
}
```

Die Headerdatei dazu sieht dann wie folgt aus:

```
/** Treiber.h ***/
#ifndef TREIBER_H
#define TREIBER_H
unsigned char getStatus();
#endif //TREIBER_H
```

Diese Vorgehensweise ist in Abschn. 4.3 näher erläutert.

Mit `volatile` kann man einen optimierenden Compiler daran hindern, eine Variable, die durch die Hardware oder einen anderen Prozess direkt manipuliert werden kann, weg zu optimieren. Beispielsweise würde die Variable `testvar`

```
int testvar;

void funktion(void) {
    testvar = 0;

    while (testvar != 255)
        /* Schleifenkörper ohne Manipulation von testvar : Endlosschleife*/;
}
```

von einem optimierenden Compiler zu `true (1)` gesetzt, da sie sich in der Schleife nicht ändert. Falls diese Variable durch die Hardware geändert würde oder durch einen Prozess, der von außen Zugriff auf den Speicherbereich hat, wäre diese Änderung unwirksam.

Mit dem `volatile` kann man dies unterbinden:

```
volatile int testvar;

void funktion(void) {
    testvar = 0;

    while (testvar != 255)
        /* Schleifenkörper ohne Manipulation von testvar bis testvar = 255*/;
```

Ein praktisches Schlüsselwort ist auch `static`. Es legt fest, dass eine *lokale* Variable (siehe Abschn. 2.8.3) nach Verlassen der Funktion „überlebt“, d. h. sie ist zwar außerhalb

ihres lokalen Kontexts nicht sichtbar, aber sie bleibt im Speicher und behält auch außerhalb ihren Wert. Ein Beispiel:

```
#include <stdio.h>

void funktion() {
    static int x = 0;
    /* x wird nur beim ersten Aufruf von funktion initialisiert
       und wird bei den nächsten Aufrufen nur hochgezählt. Am Ende
       hat es den Wert 5. */
    x++;
    printf("%d\n", x); //Drucke den Wert von x
}

int main()
{
    funktion(); //druckt die Zahl 1
    funktion(); //druckt die Zahl 2
    funktion(); //druckt die Zahl 3
    funktion(); //druckt die Zahl 4
    funktion(); //druckt die Zahl 5
    return 0;
}
```

Man kann mit dieser Technik beispielsweise Funktionsaufrufe zählen oder Funktionen splitten, ohne dass der Kontext verloren geht.

2.9 Komplexe Datentypen

2.9.1 Arrays, Felder und Strings

Regelmäßig besteht in der Software die Notwendigkeit, mehrere gleichartig strukturierte Daten zu verarbeiten. So bestehen beispielsweise Zeichenketten aus der Aneinanderreihung von einzelnen Zeichen, Vektoren aus mehreren Gleitkommazahlen oder Zeitreihen aus der Abfolge von Messwerten desselben Datentyps. Man nennt diese Abfolgen *Feld* oder *Array*. In C werden Arrays definiert, indem in einer Variablen die Anfangsadresse (Pointer) des Feldes gespeichert und über einen *Index* auf das einzelne Element zugegriffen wird.

```
unsigned char ucMeinFeld[10];
```

definiert ein Array mit dem Namen `ucMeinFeld` und der Länge 10, das bedeutet, hier sind 10 Variablen vom Typ `unsigned char` gespeichert. Da die Feldgröße unveränderlich ist, heißt das Feld auch *statisch*. Zu dynamischen Feldern und Datenstrukturen soll

in dieser knappen Zusammenfassung nichts gesagt werden, der Leser sei auf die entsprechende C-Literatur verwiesen. Die Variable `ucMeinFeld` selbst enthält dabei keine Daten sondern die Adresse des Feldes. Um auf eine Variable innerhalb des Feldes zugreifen zu können, wird diese mit der eckigen Klammer durch einen Index selektiert. Die erste Variable hat dabei den Index 0, die letzte (n-te) den Index n-1.

```
ucMeinFeld[0] = 15;
ucMeinFeld[9] = 99;
```

Ein Zugriff auf einen Index außerhalb des allokierten (angelegten) Speichers wird in der Regel vom Compiler nicht erkannt, in einer Laufzeitumgebung (Windows) wird dann bei der Ausführung der Prozess abgebrochen. In einem embedded System führt ein solcher Zugriff zu einem nicht vorhersehbaren Verhalten, dessen Ursache auch schwer zu finden ist.

Oftmals besteht die Notwendigkeit, ein Feld einmal komplett zu durchlaufen, beispielsweise um es zu initialisieren oder zu kopieren. Werden Felder als Vektoren im mathematischen Sinne verwendet, so werden vektorielle Operationen (beispielsweise das Skalarprodukt) ebenfalls durch das serielle Durchlaufen umgesetzt. Hierzu eignen sich Zählschleifen:

```
#define UC_MEIN_FELD_LEN 10
unsigned char ucMeinFeld[UC_MEIN_FELD_LEN];
int i;
...
for (i = 0; i < UC_MEIN_FELD_LEN; i++)
{
    ucMeinFeld[i] ....
}
```

Auch mehrdimensionale Felder sind möglich:

```
#define MAT_COL 10
#define MAT_ROW 8
unsigned char ucMat [MAT_ROW] [MAT_COL];
int i,j;
...
for (i = 0; i < MAT_ROW; i++) //durchläuft zunächst Spalten
                                //und dann Zeilen
{
    for (j = 0; i < MAT_COL; j++)
    {
        ucMat [i] [j] ....
    }
}
```

Ein weiteres spezielles Array ist die *Zeichenkette* (String). Hierfür gibt es in C keinen eigenen Datentyp, sondern es wird aus einem Array von char gebildet:

```
char name[20];
char ort[] = "Berlin";
```

In einer embedded-Umgebung werden Strings gerne benutzt, wenn über eine serielle Schnittstelle Daten ausgegeben werden sollen oder wenn die Zeile eines Displays beschrieben werden soll. Bereits 1963 standardisierte die American Standards Association den American Standard Code for Information Interchange (ASCII, alternativ US-ASCII) als 7-Bit Code. Er codiert die im Amerikanischen vorkommenden Buchstaben, Ziffern, das Leerzeichen (Space), eine Reihe von Sonderzeichen und diverse Steuerzeichen (in Tab. 2.10 kursiv gedruckt), die nicht dargestellt werden, sondern als Protokollzeichen beispielsweise für die Fernsteuerung von Fernschreibern verwendet werden. Tab. 2.10 zeigt den ASCII Code in hexadezimaler Form, zunächst wird die entsprechende Zeile gesucht und dann die Spalte. Das Zeichen „F“ ist damit hexadezimal 0x46, das Zeichen „Z“ ist 0x5A.

Im Array werden die Zeichen durch die jeweilige Ausgabefunktion als ASCII-Zeichen interpretiert. Interessant ist, dass viele Strings, speziell die durch Konstanten mit Doppelhochkomma gebildeten, *nullterminiert* sind. Das heißt, neben den sichtbaren Zeichen steht ein weiteres, die 0x00, da diese im ASCII-Code nicht druckbar ist – eine ASCII-Null, also die druckbare Ziffer 0, ist im ASCII Code der Zahl 0x30 (dezimal 48) zugewiesen.

Von den nichtdruckbaren Steuerzeichen sind insbesondere zu erwähnen:

- DLE: Das übliche Escape Zeichen für Bytestuffing in der Sicherungsschicht
- CR: Das Zeichen für Wagenrücklauf (\r in C)
- LF: Das Zeichen für eine neue Zeile (\n in C)
- HT: Der horizontale Tabulator (\t in C)

Tab. 2.10 ASCII Tabelle – die kursiven Zeichen sind nicht druckbar

0x...	...0	...1	...2	...3	...4	...5	...6	...7	...8	...9	...A	...B	...C	...D	...E	...F
0...	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1...	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2...	Space	!	"	#	\$	%	&	,	()	*	+	,	-	.	/
3...	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4...	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5...	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6...	*	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7...	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

Im obigen Programmbeispiel würde also der String „Berlin“ aus folgenden Zeichen bestehen

```
0x42 0x65 0x72 0x6c 0x69 0xe 0x00
B   e   r   l   i   n   NUL
```

Viele Funktionen nutzen die Nullterminierung um das Ende des Strings zu erkennen, darunter auch die bekannten Ausgabefunktion `printf()` und ihre Derivate, auf die hier im embedded Umfeld nicht eingegangen wird, der Leser möge sich das nötige Wissen durch eine einfache Internetrecherche aneignen.

Oftmals ist es notwendig, eine Zahl in einen String zu verwandeln um diesen dann über eine serielle Schnittstelle (beispielsweise an ein Terminal) auszugeben. Alternativ kann die Zahl natürlich binär übertragen werden, jedoch macht dies bei Displays und Terminals keinen Sinn. Hierzu existieren verschiedene Varianten der Funktionen `itoa()` bzw. `ftoa()`, die in `stdlib.h` zu finden sind. Die Verwendung ist im folgenden Beispiel zu sehen:

```
char text[4];
itoa(523, text, 10);
```

Die Zahl 523 wird als ASCII in einem nullterminierten String ausgegeben also `0x35 0x32 0x33 0x00`, daher ist das Array `text` auch länger als die drei Stellen der Zahl. Allgemein gilt

```
char* itoa (int Zahl, char* reservierter_String, int Zahlenbasis)
```

`itoa()` lässt sich durch eine einfache Schleife auch händisch realisieren:

```
do
{
    text[1-i] = value % 10 + 48;
    value = value / 10;
}while(value > 0);
```

Hier werden die Stellen durch eine Modulo-10 Operation isoliert und danach durch eine ganzzahlige Division nach rechts verschoben.

Hinweis: Arrays können nicht nur aus Elementartypen bestehen sondern auch aus Strukturtypen, dazu mehr in Abschn. [2.9.2](#).

2.9.2 Struktur

Im Gegensatz zum Array kann man in einer Struktur Variable verschiedenen Typs zusammenfassen und auf sie durch so genannte Selektoren zugreifen. Die Variablen werden

dabei Komponenten (englisch: members) genannt. Folgendes Beispiel soll dies verdeutlichen (man beachte das Semikolon am Ende der Definition):

```
struct datum
{
    unsigned char tag;
    char monat[10];
    unsigned char jahr;
};
```

Mit dieser Strukturdefinition kann man nun Strukturvariablen deklarieren, zum Beispiel

```
struct datum geburtstag, einschulung;
```

Auf die Komponenten kann man beim `geburtstag` dann wie folgt zugreifen:

```
geburtstag.tag = 15;
geburtstag.jahr = 1980;
```

Eine Zuweisung zwischen zwei Strukturvariablen führt dazu, dass die Inhalte kopiert werden

```
einschulung = geburtstag;
```

Initialisieren kann man eine Struktur mit einer geschweiften Klammer:

```
struct datum geburtstag = {15, "August", 1980};
```

Strukturvariablen kann man auch direkt mit der Strukturdefinition deklarieren, was aber nicht anzuraten ist:

```
struct test
{
    int a;
    char b;
} variable_a, variable_b;
```

Sinnvoller ist es, mit der Strukturdefinition gleich einen neuen Datentypen zu erzeugen und dies im .h-File abzulegen

```
typedef struct sMessung {
    unsigned int uiTemp;
    unsigned int uiBrightness;
} tMessung;
```

Ab diesem Moment können Variablen mit dem Typ tMessung deklariert werden:

```
tMessung Messreihe[10];
```

definiert ein Array `Messreihe`, in dem zehn Strukturvariablen vom Typ `tMessung` abgelegt sind. `typedef` ist auch für andere Dinge gültig. `typedef unsigned int u16;` bedeutet beispielsweise, dass ein neuer Datentyp `u16` eingeführt wird, der einem `unsigned int` entspricht.

Allgemein gilt

```
typedef struct
{
    // Struktur-Komponenten
} Strukturtyp;
```

Um die im Speicherplatz benötigte Größe einer Struktur zu ermitteln, benötigt man das C-Schlüsselwort `sizeof`.

```
sizeof(struct datum)
sizeof(tMessung)
```

liefern die Größen der Strukturen, die nicht zwingend mit der Summe der Einzelkomponenten identisch sein müssen, da die Compiler die Möglichkeit haben, den Zugriff zu optimieren.

2.9.3 Unions

Unions sind komplexe Datentypen, bei denen mehrere Strukturen oder Datentypen im Speicher übereinandergelegt werden können. Je nach Verwendung kann dann auf verschiedene Weise darauf zugegriffen werden. Im folgenden Beispiel

```
union vector3d {
    struct { float x, y, z; } vkart;
    struct { float r, phi, theta; } vpolar;
    struct { float r, z, phi; } vzyl;
    float vec3[3];
};
```

kann auf einen Vektor mit den Selektoren `x`, `y` und `z` als kartesische Koordinaten oder als Polarkoordinaten mit den Selektoren `r`, `phi` und `theta` oder als Zylinderkoordinaten `r,z,phi` oder als Array zugegriffen werden:

```
#include <math.h>
vector3d pa, pb;
pa.vpolar.r = 1.0;
pa.vpolar.phi = M_PI;
pa.vpolar.theta = M_PI/2;

pb.vkart.x = pa.vpolar.r * sin(pa.vpolar.theta) * cos(pa.vpolar.phi);
pb.vkart.y = pa.vpolar.r * sin(pa.vpolar.theta) * sin(pa.vpolar.phi);
pb.vkart.z = pa.vpolar.r * cos(pa.vpolar.theta);
```

Unions sind aber insbesondere dann hilfreich, wenn es darum geht, unterschiedliche Interpretationen der gespeicherten Inhalte zu ermöglichen, wie das folgende Beispiel zeigt.

```
union ui32
{
    struct {char i,j,k,l ;} bytes;
    long lZahl;
    float fZahl;
    char data[4];
} z;
```

kann `z` also als `float`, `long` oder in vier Einzelbytes betrachtet werden. Eine häufige Anwendung besteht darin, dass Daten von einer seriellen Schnittstelle eingelesen werden und dann als Bytarray vorliegen. Je nachdem, was im ersten Byte steht, wird der Rest der Botschaft unterschiedlich interpretiert. Es sei beispielsweise eine Botschaft aus 8 Bytes gegeben. Das erste Byte wird als Service ID (SID) interpretiert, das den Rest der Botschaft wie folgt festlegt (Tab. 2.11).

Tab. 2.11 Beispiel für die Interpretation einer seriellen Botschaft

SID	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0
0x01	char temp	short pressure		char ax	char ay	short velocity	
0x02	char[7] text						
0xFF	char errCode	char status	long timestamp				char flag

Als union würde das so umgesetzt:

```
union umsg {
    char raw_msg[8];
    struct {
        unsigned char sid;
        char temp;
        short pres;
        char ax;
```

```

        char ay;
        short v;} proc_msg;
    struct { unsigned char sid; char text[7];} text_msg;
    struct {
        unsigned char sid;
        char errCode;
        char status;
        long timestamp;
        char flag;} err_msg;
};


```

Mit einer Funktion `ReadDataFromSerial(char* data)`² könnte ein entsprechender Interpreter so aussehen:

```

umsg msg;
ReadDataFromSerial (msg.raw_msg);
switch (msg.proc_msg.sid)
{
    case 0x01: ProcessProcMsg (msg); break;
    case 0x02: ProcessTextMsg (msg); break;
    case 0xFF: ProcessErrMsg (msg); break;
}


```

2.9.4 Aufzählungstypen

Aufzählungstypen erleichtern den Zugriff auf feste Werte über symbolische Konstanten, ersetzen also aufwändige `#defines`.

Beispielsweise ersetzt

```

enum Farbe {
    Blau, Gelb, Rot, Gruen, Schwarz };


```

den folgenden Code:

```

#define Blau      0
#define Gelb     1
#define Rot      2
#define Gruen    3
#define Schwarz  4


```

² Zur Verwendung des `char*` Zeigers siehe Abschn. 2.9.5.

Man kann das dann wie folgt benutzen:

```
typedef struct {
    double x, y;
} Punkt;
struct {
    Farbe color;
    Punkt x,y;
} linie;
linie.color = Gruen;
```

Zugriff auf die Linienfarbe würde dann das Ergebnis 3 liefern. Aufzählungen werden immer aufsteigend inkrementell nummeriert, man kann dies aber auch durchbrechen, indem man die Nummer direkt explizit definiert:

```
enum Primzahl {
    Zwei = 2, Drei, Fuenf = 5, Sieben = 7
};
```

Dies sollte man jedoch nur sparsam verwenden!

2.9.5 Pointer

Im Abschn. 2.9.1 wurde bereits erwähnt, dass die Feld-Variable die Adresse des Feldes enthält, man spricht hier von einem *Zeiger* oder *Pointer* (Referenz) auf das Feld. Mit Hilfe von Pointern kann man darüber hinaus einige sehr elegante Dinge erledigen. Die gesamte Mächtigkeit von Pointern hier darzustellen würde den Rahmen dieses Buches sprengen. Stattdessen sei auf die C-Literatur verwiesen. Dennoch sollen einige Beispiele die Möglichkeiten von Pointern aufzeigen. Zur grundsätzlichen Verwendung: Eine Deklaration mit dem * liefert zunächst eine Pointervariable ähnlich wie in Abschn. 2.9.1 die Deklaration mit den eckigen Klammern []:

```
char *text;
int *array;
double *vector;
```

Allerdings ohne dazu Speicherplatz zu reservieren. Ein Feldzugriff mit Index würde zu einer Speicherschutzverletzung führen, in einem kleinen embedded System ohne Betriebssystem allerdings zu einem unkontrollierten Überschreiben von Werten aus anderen Variablen.

Hier hilft die Bibliotheksfunktion `malloc(size_t size)`, die in vielen Standardbibliotheken angeboten wird. Sie reserviert `size` Speicher und gibt einen Zeiger auf diesen

Speicher zurück, den man in einer Zeigervariablen ablegt. Nutzt man die Variable im Lauf des Programms anderweitig, so muss der Speicher mit `free()` wieder freigegeben werden. Im Rahmen des Buches soll hierauf zunächst nicht eingegangen werden.

Um mit einer Zeigervariablen zu arbeiten gibt es folgende Möglichkeiten:

```
int a; //normale Integervariable
int *b; //Pointervariable (Achtung! Ohne Speicher)
&a //liefert die Adresse (Pointer) der Variablen a
*b //liefert den Inhalt der Speicherzelle, auf die b zeigt
b=&a //setzt die Pointervariable b auf die Adresse der Variablen a
a=*b //kopiert den Inhalt der Speicherzelle, auf die b zeigt, in a
```

In der zweitletzten Zeile `b=&a` des obigen Beispiels wird das Verhalten deutlich. Die Variable `b` verweist auf den Speicherplatz von `a`, d. h. bei einer Änderung von `*b`:

```
a = 3;
b = &a;
*b = 9;
```

würde sich auch der Inhalt der Variablen `a` ändern! Dies kann man zu einigen interessanten Anwendungen nutzen, wie im Folgenden gezeigt wird:

2.9.5.1 Verwendung beim Funktionsaufruf

Die Verwendung von Zeigern lässt sich anschaulich erklären, wenn man Felder unbekannter Größe in Funktionen manipulieren möchte. Als Beispiel sei die folgende Funktion `ToCapital()` genannt, die die Buchstaben eines Textes in Großbuchstaben umwandelt. Selbstverständlich könnte man bei einem nullterminierten String Abschn. 2.9.1 auch die Abschluss-Null als Textendezeichen detektieren, dieselbe Technik lässt sich jedoch in vielen weiteren Beispielen anwenden, die auch im Verlauf des Buches beschrieben werden.

Die Funktion sieht wie folgt aus:

```
int ToCapital(char *text, int len)
{
    int i, j = 0;
    for (i = 0; i < len; i++) //über den gesamten String
    {
        //WENN Buchstabe zwischen a und z
        if (text[i] >= 0x61 && text[i] <= 0x7A)
        {
            text[i] -= 0x20; //verringere Wert auf A..Z
            j++;
        }
    }
    return j; //Anzahl der Treffer
}
```

Ein Aufruf

```
char text[] = "Hallo Erde!";
int y;
y = ToCapital(text, 11);
```

liefert in `text` die Zeichenkette „HALLO ERDE“ und setzt `y` auf 7, da insgesamt sieben Zeichen umgesetzt wurden.

Eine weitere Anwendung von Pointern in Funktionsaufrufen ist durch Funktionen gegeben, die mehrere Rückgabewerte haben sollen. Der „Klassiker“ ist das Vertauschen zweier Variablen. Die Funktion `swap(double *a, double *b)` erledigt dies wie folgt:

```
void swap(double *x, double *y)
{
    double tmp;
    double = *x;
    *x = *y;
    *y = tmp;
}
```

Ohne Zeiger würde das call-by-value Prinzip diese Funktion sinnlos machen.

2.9.5.2 Zeigerarithmetik

Grundsätzlich kann mit Zeigern rechnen. Ein Beispiel:

```
int x[10];
int *y;
y = &x[5];
```

Am Ende dieser Zeilen zeigt `y` auf das sechste Element im Feld `x`. Mit

```
y++;
```

zeigt `y` nun auf das siebte Element des Feldes `x`. Der Compiler erkennt am Typ (Zeiger auf integer) automatisch, wie groß die Elemente sind und berechnet die nächste Position. Dies unterscheidet die Zeigerarithmetik grundsätzlich vom Arbeiten mit Adressen in Variablen. Will man nach dem Inkrementieren des Speichers gleich den Wert des nächsten Elements ausgeben, muss man die Bindungspriorität des Dereferenzierungsoperators (Abschn. 2.5.6) beachten:

```
*y++; //liefert den Wert des nächsten Elements
(*y)++; //erhöht den Wert des aktuellen Elements um 1
```

2.9.5.3 Zeiger auf Funktionen

Auch auf Funktionen kann man Zeiger setzen, da sie ebenfalls Objekte im Speicher darstellen. Die Deklaration:

```
int (*fun) (int);
```

liefert in der Variablen `fun` die Referenz auf eine Funktion, die vom Typ `int` ist und einen Eingabeparameter vom Typ `int` hat. Hätten wir nun zwei Funktionen zur Auswahl:

```
int Inkrement(int a);
int Dekrement(int a);
```

wovon die eine Funktion die Zahl `a` inkrementieren, die andere die Zahl `a` dekrementieren soll, könnten wir nun schreiben:

```
typedef enum {gross, klein} sizes;
int KleinOderGross(int zahl, sizes richtung)
{
    int (*fun) (int);
    if (richtung == klein)
    {
        fun = &Dekrement;
    }
    else fun = &Inkrement;
    return fun(zahl); //führt die Funktion aus, auf die der
                      //Zeiger zeigt
}
```

Je, nachdem ob der Übergabeparameter `richtung` nun „klein“ oder „gross“ ist, liefert die Funktion das Inkrement oder Dekrement der übergebenen Zahl.

```
y=KleinOderGross(5,klein);
```

setzt `y` also auf 4;

Wir werden in den nächsten Kapiteln noch Beispiele mit mehr Sinn sehen.

2.10 Aufbau eines embedded C-Programms

Für die folgenden Betrachtungen ist es sinnvoll, eine grundsätzliche Programmstruktur festzulegen. Diese wird in den folgenden Kapiteln erweitert werden.

Wir gehen von den folgenden Grundsätzen aus:

- Die main-Funktion ist die Einstiegsfunktion, die den generellen Ablauf steuert.
- Die main-Funktion ist so kurz wie möglich zu halten
- Jedes Programm besteht aus einem Initialisierungsteil und einem Arbeitsteil oder Prozesssteil (Hauptschleife), der ständig ausgeführt wird und die eigentliche Arbeit verrichtet.
- Alle Funktionen, die auf Hardware (Register) zugreifen, werden zusätzlich in eigenen Modulen beschrieben, um das „Kern“-Programm hardwareunabhängig zu machen.
- Zu jedem Source-File existiert ein Headerfile, in dem die (öffentlichen) Funktionen deklariert werden! Öffentlich heißt, dass auf diese auch aus anderen Modulen zugegriffen werden soll. Wir gehen zunächst einmal davon aus, dass alle Funktionen öffentlich sind.

Wir benötigen also zwei Files `Programmname.c` und `Programmname.h`. Im Interesse der Lesbarkeit wurde hier auf aufwendige Kommentierung verzichtet. Die ausführlich kommentierten Quellen sind im den begleitenden Quellen im Downloadbereich des Buches zu finden.

Eine völlig leere Programmstruktur sieht dann so aus:

```
#include "Programmname.h"      // Headerfile des Hauptmoduls
/*****************************************/
/* Deklaration der modulglobalen Variablen */
/*****************************************/
// noch keine vorhanden
/*****************************************/
/*          Hauptprogramm           */
/*****************************************/
int main()
{
    Init(); //Initialisierungsteil
    while(1) //Endlosschleife
    {
        //Hier werden die Prozess-Funktionen aufgerufen
    }
    return 0;
}
/*****************************************/
/*          Initialisierung         */
/*****************************************/
void Init(void)
{
    //Hier werden die Initialisierungen aufgerufen
}
```

Die Deklaration der Init()-Funktion erfolgt im Header-File Programmname.h

```
#ifndef PROGRAMMNAME_H
#define PROGRAMMNAME_H
/*********************************************************************
/*                                Deklaration der Funktionen      */
/*********************************************************************
void Init(void);
#endif
```

Durch das Konstrukt `#ifndef PROGRAMMNAME_H` wird, wie bereits erwähnt, versehentliches doppeltes Einbinden verhindert, indem nach dem Makro `PROGRAMMNAME_H` gesucht wird. Ist es nicht vorhanden, war das Headerfile noch nicht eingebunden, das Makro wird dann definiert und die Headerinformationen können gelesen werden. War das Makro jedoch definiert, wird der Rest übersprungen (Abschn. 2.8.4). Um eine eindeutige Bezeichnung der Makros zu erreichen, wird der Name des Headerfiles mit dem Suffix `_H` verwendet.

2.11 Übersetzen und Binden

Üblicherweise verwendet man für die Erstellung eines Programms eine integrierte Entwicklungsumgebung (engl. IDE für Integrated Development Environment). Diese steuert den gesamten Entwicklungsablauf von der Erstellung der Quellcodes bis zum Upload auf das Zielsystem. Bekannte IDEs sind Eclipse, Visual Studio oder die Umgebungen von Greenhill und Keil. Wir haben bei der Erstellung dieses Buches das AtmelStudio benutzt, das Microchip unter <http://www.microchip.com/avr-support/atmel-studio-7> gegen Registrierung kostenlos verteilt. Grundsätzlich kann man aber auch kommandozeilenorientierte Tools verwenden, beispielsweise die bekannte GNU-Toolkette, die auch den C-Compiler für AtmelStudio bereitstellt.

Der Prozess ist jeweils derselbe und findet sich in Abb. 2.1: .c und .h Dateien bilden den Quellcode, der in einem geeigneten Editor erstellt wird. Übliche Editoren bieten Syntax Highlighting (farbliche Kennzeichnung von Sprachelementen) und Auto-Vervollständigen bekannter Funktions- und Variablennamen an. Der Präkomplier steuert das Einbinden der .h Dateien und die Auflösung von Macros. Daraus entsteht letztlich der finale Quellcode, der dann vom Compiler zunächst auf seine Syntax hin überprüft wird. Anschließend wird ein so genannter Object-Code erzeugt. Dieser ist nicht lauffähig, da die Referenzen auf Funktionen und externe Variablen noch nicht aufgelöst sind. Das kann erst geschehen, wenn alle Quelldateien einzeln übersetzt sind. Der Linker (Binder) bindet die Objektdateien zusammen und löst darauf die externen Referenzen auf, das heißt, er weist jeder Funktion und jeder Variable einen Speicherplatz zu. Hier können auch bereits vorcompilierte Objektcodesammlungen (Bibliotheken, in der Regel mit .a benannt) eingebunden werden, diese haben der Vorteil, dass der Nutzer den Quellcode nicht sieht, sondern über die .h Datei nur die Schnittstelle, also die deklarierten Funktionen, Macros und externen Variablen. Auf diese Weise ist eine kommerzielle Nutzung von Bibliotheken möglich. Der

lauffähige Code wird schließlich in geeigneter Form mit einem Tool zum Flashen auf das Zielsystem übertragen, das in der Regel ebenfalls in die Entwicklungsumgebung integriert ist. Hierzu ist ein Programmiergerät notwendig, das die Prozessorhersteller meist kostengünstig anbieten. Für AVR®-Prozessoren mit Programmierschnittstelle nach dem JTAG Standard (IEEE-Standard 1149.1) kann beispielsweise das JTAG-ICE für Upload und On-Chip Debugging verwendet werden. Das Programmiergerät STK500 ist für Prozessoren geeignet, die für so genannte In System Programmierung (ISP) oder andere Programmiermodi der AVR-Familie über serielle Schnittstellen verwendet werden. Für dieses System existieren sehr preiswerte Nachbauten oder Bauanleitungen im Netz.

Weiterhin enthalten viele IDEs einen Debugger. Dieser verknüpft die Zeilen im Quellcode mit dem lauffähigen Zielcode und überwacht den Zustand der Variablen und Register. Das Programm kann schrittweise oder durchlaufend ausgeführt werden, gezielt können einzelne Funktionen aufgerufen werden und der Ablauf kann jederzeit an einem Haltepunkt (Breakpoint) oder aufgrund einer Bedingung angehalten werden. Im AtmelStudio ist ein Simulator integriert, mit dem man den Code ohne Zielsystem debuggen kann. Bei Verwendung eines Programmiergerätes mit JTAG kann der Prozess direkt auf dem Prozessor stattfinden. Üblich sind auch so genannte Emulatoren, die das Zielsystem hardwaremäßig nachbilden und über erweiterte Diagnosemöglichkeiten verfügen.

Literatur

1. ISO/IEC 9899:2011: *Information technology—Programming languages—C*. www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=57853, Zugegriffen: 25. April 2016
2. Mathworks: *Embedded Coder*. <http://de.mathworks.com/products/embedded-coder/features.html>, Zugegriffen: 25. April 2016
3. Wikibooks: *C-Programmierung*. <https://de.wikibooks.org/wiki/C-Programmierung>, Zugegriffen: 25. April 2016
4. Erlenkötter, H.: C: Programmieren von Anfang an, 23. Aufl. Rowohlt, Reinbeck (1999)
5. Kernighan, B.W., Ritchie, D.M.: Programmieren in C: Mit dem C-Reference Manual in deutscher Sprache. Hanser, München (1990)
6. Kernighan, B.W., Ritchie, D.M.: The C programming language, 2. Aufl. Prentice Hall, London (2010)
7. Gookin, D.: C für Dummies, 2. Aufl. Wiley-VCH, Weinheim (2010)
8. Goll, J.: C als erste Programmiersprache – Mit den Konzepten von C11, 8. Aufl. Springer Vieweg, Wiesbaden (2014)
9. Wiegemann, J.: Softwareentwicklung in C für Mikroprozessoren und Mikrocontroller, 7. Aufl. VDE Verlag, Berlin (2017)

Weiterführende Literatur

10. ISO/IEC 14882:2014: *Standard for Programming Language C++*. <https://isocpp.org/std/the-standard>, Zugegriffen: 25. April 2016
11. Wikibooks, Die freie Bibliothek: C-Programmierung: Komplexe Datentypen (2015). https://de.wikibooks.org/w/index.php?title=C-Programmierung:_Komplexe_Datentypen&oldid=764916, Zugegriffen: 21. März 2016

Programmierung von AVR Mikrocontrollern

3

Zusammenfassung

In diesem Kapitel wird der Aufbau der wichtigsten Peripherieelemente der AVR-Familie besprochen, deren Mechanismen natürlich auch auf andere Prozessoren übertragbar sind. Dabei lernen Sie die grundsätzlichen Funktionen kennen, mit denen ein Mikrocontroller der AVR-Familie mit der Außenwelt kommuniziert und wie sich die Abläufe mit Timern steuern lassen. Außerdem wird auf die Funktion und Verwendung von Interrupts eingegangen. Am Ende des Kapitels sind Sie in der Lage, einfache, digitale I/O Anschlüsse einzulesen und zu schalten, analoge Sensoren auszulesen, das interne EEPROM zu beschreiben und einen Motor mit einer Vollbrücke anzusteuern. Auch das Powermanagement wird kurz besprochen. Alle Beispiele in diesem und den folgenden Kapiteln sind getestet und mit dem AtmelStudio (Version 6 und höher) kompiliert worden.

Kap. 3 erhebt keinen Anspruch auf Vollständigkeit, sondern erläutert die Zusammenhänge insoweit, als sie für das weitere Verständnis der Beispiele im Rest des Buches notwendig sind. Im Literaturanhang sind einige Bücher ausgewiesen, in denen ausführlich auf einzelne Funktionen eingegangen wird. Fortgeschrittenen Leserinnen und Lesern wird im Anschluss die Lektüre des sehr umfangreichen und gut geschriebenen Datenblattes [1] empfohlen. Daneben empfehlen die Autoren, die hervorragende Webseite mikrocontroller.net zu konsultieren [2].

3.1 Architektur der AVR Familie

Jeder Mikrorechner, so auch die Prozessoren der AVR Familie (Abb. 3.1), besteht zunächst aus einer CPU (Central Processing Unit). Diese enthält mindestens zwei Komponenten: ein Steuerwerk, auch Leitwerk genannt, das für die Abarbeitung und Interpretation von

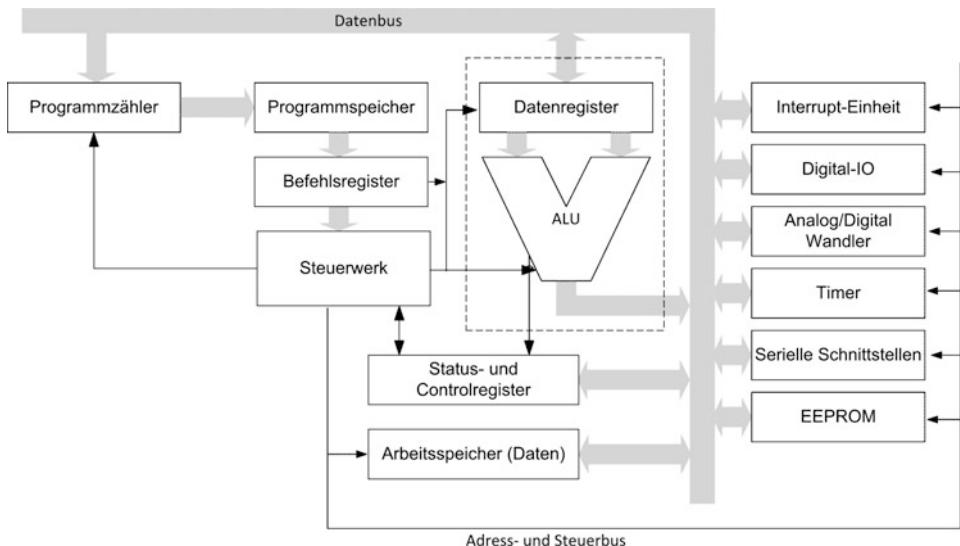


Abb. 3.1 Grundlegende Architektur von AVR Mikrocontrollern

Befehlen zuständig ist und die anderen Komponenten steuert, und ein Operationswerk, das aus den Arbeits- oder Datenregistern und dem Rechenwerk, der ALU (Arithmetic Logic Unit), besteht. Es ist für arithmetische und logische Rechenoperationen zuständig. Das Leitwerk und das Rechenwerk können auf zwei Arten ausgeführt sein:

- Bei sehr einfachen CPUs bestehen sie aus einer festverdrahteten Logik
- Bei komplexeren CPUs sind sie selbst mikroprogrammiert, d. h. ein Befehl startet ein eigenes, im Controller hinterlegtes Programm.

Mit dem Prozessortakt, der durch einen internen Oszillatior oder einen externen Quarzoszillatior generiert wird, wird nun eine Zustandsfolge nach (grob vereinfacht) dem folgenden Schema abgearbeitet:

- Ein Programmzähler (PC program counter) wird um eins erhöht (inkrementiert) und adressiert in einem Programmspeicher einen Befehl. Dieser wird in der Regel in einem Befehlsregister zwischengespeichert (Instruction Fetch Cycle).
- Der Befehl wird dekodiert und ggf. werden die zum Befehl gehörenden Operanden in das Arbeitsregister geladen. Beispielsweise benötigt eine Addition zwei Operanden, nämlich die beiden Zahlen, die addiert werden müssen. Diesen Zustand nennt man Instruction Decode/Register Fetch Cycle. In einem mit festverdrahteter Logik ausgeführten Leitwerk fallen die beiden Zyklen zusammen. In einem mikroprogrammierten Leitwerk wird hier über mehrere Taktzyklen die Befehlausführung vorbereitet.
- Im dritten Schritt berechnet die ALU das Ergebnis der Operation (Execute Cycle), gegebenenfalls werden hier auch noch Speicherzugriffe ausgeführt. Bei einer fest-

verdrahteten ALU fällt die Ausführung mit dem letzten Zyklus zusammen, bei einer mikroprogrammierten ALU wird auch hier ein Programm über mehrere Taktzyklen gestartet. Im Statusregister werden bestimmte Werte gesetzt, beispielsweise ob eine Operation mit dem Ergebnis 0 endet oder einen Überlauf (Carry) erzeugt, den man für folgende Operationen nutzen kann.

- Im vierten Schritt werden die Rechenergebnisse in die Arbeitsregister zurückgeschrieben (Write Back Cycle)

Komplexere Mikrorechner durchlaufen mehr als diese vier Zustände, einfachste Architekturen begnügen sich mit zweien (Fetch/Decode/Execute und Write Back). Zur Beschleunigung können so genannte Pipelines eingesetzt werden, die die Zyklen auf aufeinanderfolgende Befehle parallel ausführen können. Dies sprengt jedoch den Rahmen dieser Einführung. Mikroprogrammierte Leit- und Operationswerke ermöglichen komplexe Befehle, deren Abarbeitung allerdings viel Zeit benötigt (CISC: Complex Instruction Set Computer). Die Intel- und AMD-Prozessoren für PCs sind solche CISC-Prozessoren. Dagegen basiert das Konzept der RISC (Reduced Instruction Set Computer) auf sehr simplen Befehlen, die möglichst in einem Takt abgearbeitet werden können. Der AVR-Controller, der in diesem Buch beschrieben ist, ist ein solcher RISC-Rechner. Er ist sehr schnell beim Bearbeiten einfacher Befehle und benötigt für komplexere Aufgaben dann einfach mehr Befehle.

Grundsätzlich unterscheidet man grob die folgenden Befehlsarten, die eine CPU ausführen kann:

- Arithmetisch/Logische Befehle, z. B. Addition, Subtraktion, Multiplikation, boolesche Funktionen, Negierung usw.
- Datentransferbefehle, d. h. Kopieren von Daten aus Registern in den Arbeits- oder Programmspeicher oder umgekehrt aus dem Speicher in Register, zwischen Registern oder zwischen Speicherzellen.
- Sprungbefehle: Der Programmzählerstand wird nicht einfach inkrementiert, sondern mit einem Wert geladen, der z. B. vom vorhergehenden Berechnungsergebnis abhängig sein kann. Dadurch kann der Programmablauf abhängig von der Berechnung verändert werden (konditionale Sprünge).

Viele Befehle (Operationen) benötigen einen oder mehrere Operanden. Diese können fest oder variabel sein. Variable Operanden und die Rechenergebnisse werden in den Arbeitsregistern oder im Arbeitsspeicher abgelegt. Steuerbefehle, Adressen und Daten werden über zentrale Leitungsstränge (Busse) im System verteilt. Grundsätzlich unterscheidet man zwei Architekturtypen, die durch die Busanbindung von Arbeits- und Programmspeicher gekennzeichnet sind.

Bei der „von Neumann“-Architektur sind Programmspeicher und Arbeitsspeicher an einen Bus angeschlossen, Zugriffe auf Programme und Daten erfolgen deshalb sequentiell („von Neumann-Flaschenhals“), dafür ist der Zugriff flexibler. In einem PC sind

Programm- und Arbeitsspeicher sogar identisch, die Programme werden bei Bedarf von der Festplatte in den Arbeitsspeicher kopiert. In einer Speicherzelle kann nicht mehr zwischen Daten und Anweisung unterschieden werden.

Embedded Controller hingegen sind oft in Harvard-Architektur gehalten, so auch die Prozessoren der AVR Familie. Diese an der Harvard Universität in Boston entwickelte Architektur fordert strikte Trennung zwischen Programm, Daten, Rechenwerk und Peripherie. Der Vorteil dieses Konzeptes besteht darin, dass die Software kompakt in einem (normalerweise nichtflüchtigen) Speicher abgelegt ist, die Daten liegen im Arbeitsspeicher, bzw. in weiteren nichtflüchtigen Speichern (EEPROM). Da während der Laufzeit das Programm ohnehin nicht geändert wird, ist die Flexibilität der von Neumann Architektur nicht notwendig. Der Zugriff auf Programm und Daten ist wesentlich schneller und effizienter als bei der von Neumann Architektur. Weiterhin können Programm- und Datenspeicher hinsichtlich der Daten- und Befehlswortbreite getrennt optimiert werden, was zu besserer Speicherbelegung führen kann. Außerdem wird bei Softwarefehlern kein Programmcode überschrieben, was bei der von Neumann Architektur möglich ist. Bei der AVR-Familie ist die Trennung nicht ganz so strikt. Es gibt einen direkten Durchgriff aus dem Programmspeicher zum Rechenregister, damit können konstante Operationen, die Teil des Programms sind, ohne Umweg über den Arbeitsspeicher in die Berechnung einbezogen werden.

3.2 Gehäuse und Anschlussbelegungen

Die AVR Familie enthält hunderte von verschiedenen Prozessoren. In diesem Buch beziehen wir uns in den Beispielen jeweils auf den ATmega48/88/168, insbesondere, weil diese in praktischen 28-poligen PDIP-Gehäusen geliefert werden, die auch für den schnellen



(PCINT14/RESET) PC6	1		28	PC5 (ADC5/SCL/PCINT13)
(PCINT16/RXD) PD0	2		27	PC4 (ADC4/SDA/PCINT12)
(PCINT17/TXD) PD1	3		26	PC3 (ADC3/PCINT11)
(PCINT18/INT0) PD2	4		25	PC2 (ADC2/PCINT10)
(PCINT19/OC2B/INT1) PD3	5		24	PC1 (ADC1/PCINT9)
(PCINT20/XCK/T0) PD4	6		23	PC0 (ADC0/PCINT8)
VCC	7		22	GND
GND	8		21	AREF
(PCINT6/XTAL1/TOSC1) PB6	9		20	AVCC
(PCINT7/XTAL2/TOSC2) PB7	10		19	PB5 (SCK/PCINT5)
(PCINT21/OC0B/T1) PD5	11		18	PB4 (MISO/PCINT4)
(PCINT22/OC0A/AIN0) PD6	12		17	PB3 (MOSI/OC2A/PCINT3)
(PCINT23/AIN1) PD7	13		16	PB2 (SS/OC1B/PCINT2)
(PCINT0/CLKO/ICP1) PB0	14		15	PB1 (OC1A/PCINT1)

Abb. 3.2 Pinbelegung des ATMega8 im PDIP Gehäuse. (Nach [1])

Brettaufbau geeignet sind. Für den Einsatz im Produkt eignen sich die 28- und 32-poligen MLF oder das TQFP Gehäuse in SMD Löttechnik. Für sehr kleine Anwendungen, beispielsweise in Dimmern, Brandmeldern, Modellbautechnik usw. eignet sich der ATtiny in verschiedenen Varianten, diese besitzen weniger Speicherplatz und weniger Peripherie als die ATmega, dafür sind sie bereits im 8-poligen PDIP oder im 20-poligen QFN/MLF Gehäuse zu haben und damit sehr kompakt aufgebaut. Für Anwendungen mit CAN Vernetzung eignet sich in der ATmega-Familie der AT90CANxx mit deutlich mehr Features im 64-poligen TQFP Gehäuse.

Abb. 3.2 zeigt die Pinbelegung des ATmega48/88/168. Die einzelnen Pins werden später ausführlich beschrieben.

3.3 Versorgung, Takt und Reset-Logik

3.3.1 Versorgung

Die Prozessoren der AVR Familie werden in der Regel mit 1,8 V oder 3,3 V bis 5 V versorgt. Bei höherer Quarzfrequenz wird in der Regel auch eine höhere Spannung benötigt. Sinkt die Spannung ab, z. B. infolge einer leeren Batterie, kann es zu Instabilitäten kommen. Daher besitzen viele Mikrocontroller eine so genannte Brown-Out-Detection, die unterhalb eines einstellbaren Versorgungsspannungspegels den Prozessor zurücksetzt.

Durch eine Messung der Versorgungsspannung mit einer internen, versorgungsunabhängigen Referenz können z. B. Schreibvorgänge im EEPROM abgeschlossen werden bevor ein Brown-Out Reset einsetzt. In realen Schaltungen sollte man beispielsweise mit Akkus oder Supercaps dafür sorgen, dass im Fall eines Einbruchs der Versorgungsspannung für die Abarbeitung dieser Notmaßnahmen noch ausreichend Energie zur Verfügung steht (meist einige 10 ms).

Nach einem Brown-Out-Reset ist im zentralen MCU Status Register (MCUSR) der AVR Familie das Brown-out reset flag gesetzt (BORF). Dieses kann nach einem Reset für weitere Maßnahmen herangezogen werden. Die Versorgung wird im späteren Abschn. 3.8.1 noch einmal näher beleuchtet.

3.3.2 Takt

Die Prozessoren der AVR-Familie besitzen eine Reihe von Taktsignalen, die von einem zentralen Takt abgeleitet werden können. Die wichtigsten Quellen für den Takt sind:

- Ein interner Oszillator, der ohne externe Beschaltung mit 8 MHz getaktet ist und ein interner Oszillator mit 128 kHz, der den Prozessor in einem extrem stromsparenden Modus betreiben kann. Die Oszillatoreingänge können dann bei den meisten Prozessoren als zusätzliche IO-Pins verwendet werden

- Ein Oszillator, der mit einem externen Resonator (Quarz oder Keramik) an den Pins XTAL1 und XTAL2 nach Abb. 3.3 beschaltet wird und bei der ATmega-Familie bis zu 20 MHz getaktet werden kann.
- Ein externer Oszillator, der ein Rechtecksignal auf den Oszillator XTAL1 des Prozessors legt, XTAL2 kann dann als IO-Pin verwendet werden.

Typische Nutzungsszenarien sind:

- Nutzung nur des internen Oszillators
- Nutzung eines externen Quarzes (Abb. 3.3) mit der vollen Taktfrequenz (z. B. 18,432 MHz)
- Nutzung des internen Oszillators mit voller Taktfrequenz und Anschluss eines Quarzes mit 32.768 Hz um in einem stromsparenden Modus lediglich eine Echtzeituhr zu betreiben.

Die Wahl des Oszillators erfolgt über die Programmiersoftware über die Registerkarte „Fuses“ über die Einstellung der Fuse: SUT_CKSEL. Fuses heißen so, weil man früher die Hardware durch Durchbrennen einer Leitung (Sicherung) konfigurieren konnte. Dieses Prinzip wird in so genannten PROMS (Programmable Read Only Memory) verwendet, in denen jede Speicherzelle einmalig und irreversibel durch einen Stromimpuls beschrieben werden konnte. Die Fuses im Mikrocontroller sind heute nicht mehr irreversibel. Dennoch sollte man sie mit Vorsicht bedienen, insbesondere, wenn man mit einem SPI Programmiergerät arbeitet und der Prozessor eingelötet ist. Eine falsche Einstellung kann das System vollständig lahmlegen (beispielsweise durch Deaktivierung der Programmierschnittstelle oder falsche Wahl des Oszillators).

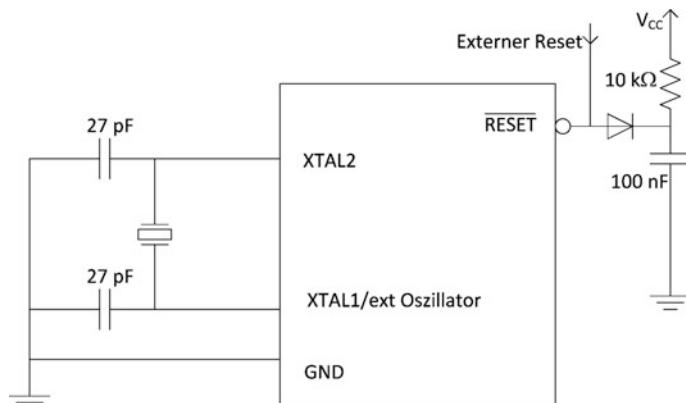


Abb. 3.3 Beschaltung des Oszillators und des RESET Eingangs

3.3.3 Reset-Logik

Das Zurücksetzen des Prozessors (RESET) bewirkt, dass der Programmzähler auf die Adresse 0 im Programmspeicher zeigt und u. a. die Ausgänge hochohmig geschaltet werden. An der Adresse 0 steht ein Sprungbefehl zum Speicherort des Hauptprogramms main(), das heißt, das Programm startet dort.

Der Prozessor kann über verschiedene Mechanismen extern zurückgesetzt werden:

- Beim Einschalten (Power-On-Reset) bildet der ungeladene Kondensator am RESET-Pin einen Kurzschluss gegen Masse. Dadurch wird der Reset-Pin 1 kurz auf 0 gelegt bis sich C1 über R1 aufgeladen hat.
- Eine RESET-Taste am RESET-Pin bewirkt einen manuellen Reset.
- Am RESET-Pin ist ein so genannter Watchdog angeschlossen. Dieser versucht regelmäßig, nach einer bestimmten Zeit von einigen ms bis Sekunden, den Prozessor zurückzusetzen. Um ihn daran zu hindern, muss der Watchdog vor Ablauf seiner „Time-out“ Zeit zurückgesetzt werden. Dies geschieht über eine digitale IO-Leitung, die aus dem Programm angesteuert wird. „Hängt“ das Programm aus irgendwelchen Gründen, läuft die Watchdog-Zeit ab und der Prozessor wird zurückgesetzt.
- Die Programmierung des ATmega88 kann nur erfolgen, wenn der Prozessor sich im RESET Modus befindet, daher wird die RESET-Leitung vom Programmiergerät auf Masse gezogen

Hinweis für Fortgeschrittene: Intern wird der Prozessor durch Brown-Out (Unterspannung) oder durch den intern ebenfalls vorhandenen Watchdog zurückgesetzt, falls dieser aktiv ist. Dies kann eine üble Fehlerquelle sein, deshalb empfiehlt der Hersteller, das Watchdog System Reset Flag (WSRF) im MCU Control Register (MCUCR) und das Watchdog System Reset Enable (WDE) Flag im Watchdog Timer Control Register stets zu löschen, auch wenn der Watchdog nicht benutzt wird.

3.3.4 Speicher

Die Abb. 3.4 zeigt die drei Speicherarten der AVR Familie.

In Abb. 3.4a ist der Flash-Speicher, der beim ATmega88 8 kByte groß ist. Im Sinne der Harvard-Architektur stellt er den Programmspeicher (Firmware) dar. In Abb. 3.4c ist der EEPROM Speicher zu sehen, dieser hält persistente (also über das Ausschalten hinaus gültige) Daten, z. B. Fehlerspeichereinträge oder Parametrierungen, die man unabhängig vom Quellcode einstellen will (für Kundenzwecke).

Flash und EEPROM werden auf zwei Arten programmiert:

- Solange der RESET-Eingang auf 0 (low) gehalten wird, sind Flash und EEPROM über die SPI-Schnittstelle programmierbar. Dies wird durch viele Programmiergeräte (z. B. ISP Programmer) so durchgeführt.

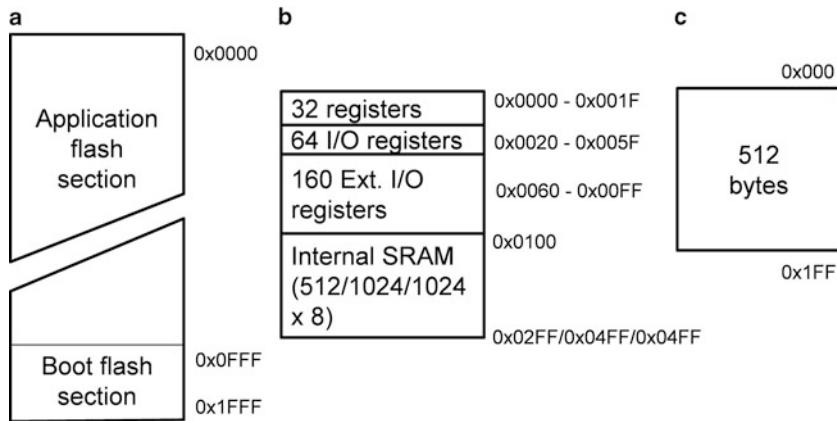


Abb. 3.4 Speicher der AVR-Familie am Beispiel ATmega88. **a** Programmspeicher, **b** Datenspeicher, **c** EEPROM. (Nach [1])

- Beide Speicher sind auch aus dem Programm heraus programmierbar, so kann ein Programm „sich selbst“ umprogrammieren, üblicherweise wird dies aus einer Bootloaderroutine oder aus einer Routine geschehen, die von der seriellen Schnittstelle einen Programmcode liest.
- Manche Vertreter der AVR-Familie besitzen eine eigene Boot-Flash-Section. Diese hält die Bootloaderoutine und kann in ihrer Größe variiert und optional verriegelt werden, so dass sie nur noch überschrieben werden kann, wenn sie zuvor entriegelt wurde. Einstellbar kann diese Sektion nach einem RESET ausgeführt oder als Unterprogramm angesprungen werden, beispielsweise durch eine entsprechende Botschaft über eine serielle Schnittstelle. Damit kann man über einen Diagnosezugriff beispielsweise über eine serielle Schnittstelle die Firmware des Systems umprogrammieren.

In Abb. 3.4b sind die Register und der Arbeitsspeicher angedeutet. Im Fall des ATmega88 besitzt dieser 1 kByte freien Speicherplatz¹ und dient dazu, Variablen und Felder zwischen zu speichern. Da der Compiler bevorzugt lokale Variablen in Arbeitsregistern statt im Arbeitsspeicher hält, wird dieser in der Regel nur für globale Variablen benötigt. In der Abbildung sind die 32 Arbeitsregister skizziert, die direkt von der Recheneinheit adressiert werden können und deshalb entsprechend schnelle Befehle ermöglichen. Daten im Arbeitsspeicher müssen vor der Verarbeitung erst in ein Arbeitsregister kopiert werden, was entsprechend Zeit benötigt.

Weiterhin sieht man in der Abbildung die so genannten IO-Register. Diese steuern die Funktionen des Mikrocontrollers. In den folgenden Abschnitten werden die wichtigsten davon beschrieben. Eine ausführliche Referenz findet sich in [1].

¹ In Abb. 3.4b sind auch die Speicherausdehnungen des ATmega48 und des ATmega168 mit 512 Byte bzw. 1 KByte dargestellt.

3.4 Umgang mit Registern

Um die Bedienung aller I/O Ports und Module zu verstehen, muss man zunächst etwas über die Speicherorganisation des AVR wissen. Der Zugriff auf die Peripherie erfolgt über *Register*, die im Adressbereich des Arbeitsspeichers am Datenbus liegen (Abb. 3.4).

Am Beispiel des ATmega88 sei dies näher erläutert: Die ersten 32 Byte (0x0000 bis 0x001F) sind für die Arbeitsregister vorgesehen, die eine schnelle Zugriffsmöglichkeit bieten und die Operanden bei Operationen aufnehmen. Die nächsten 64 Byte (0x0020 bis 0x005F) sind für die I/O Register, weitere 160 Byte für die erweiterten I/O Register und anschließend (ab Adresse 0x0100) folgen beim ATmega88 noch 1024 Byte Arbeitsspeicher, wo Variablen abgelegt werden.

Der Zugriff auf die Peripherie erfolgt also ebenso wie auf eine Variable. Die Register sind ein Byte groß entsprechend dem Datentyp `unsigned char`.

Der direkte Zugriff auf eine Adresse ist fehlerträchtig und auch nicht sehr intuitiv, deshalb haben die Register Namen, die üblicherweise im include-File `<avr/io.h>` definiert sind².

Für alle, die es genau wissen wollen: Die Registermakros stellen eine Referenz auf die Portadresse dar, z. B.:

```
#define PORTB    _SFR_IO8 (0x05)
```

Das Makro `_SFR_IO8(io_addr)` ist in `sfr_defs.h` definiert, das ebenfalls von `io.h` eingebunden wird:

```
#define _SFR_IO8(io_addr) ((io_addr) + __SFR_OFFSET)
```

mit `__SFR_OFFSET = 0x020`.

Mit dieser Definition ist es möglich, statt mit der Adresse (hier 0x25) mit dem Namen `PORTB` auf das Register zuzugreifen, und zwar über den Zuweisungsoperator `=`. Beispiel:

```
PORTB = 0x17;
unsigned char value = PORTB;
```

Oft will man jedoch lediglich ein einziges Bit eines Registers verändern oder abfragen. Da C keine Bitmanipulationsbefehle kennt, hilft man sich mit den Bitoperatoren:

- & bitweises UND
- | bitweises ODER
- ~ bitweises Negieren, Einerkomplement

² Um genau zu sein, bindet `io.h` das dem Prozessortyp entsprechende File ein, z. B. `iomx8.h` für die ATmegax8 Serie.

<< bitweises Linksschieben
 >> bitweises Rechtsschieben

Das Setzen eines einzelnen Bits erfolgt also durch das „Verodern“ des Registers mit einer einzelnen 1, d.h.

0000 0001 entspricht der hexadezimalen 0x01 (0. Bit)
 0000 0010 entspricht der hexadezimalen 0x02 (1. Bit)
 0000 0100 entspricht der hexadezimalen 0x04 (2. Bit)
 0000 1000 entspricht der hexadezimalen 0x08 (3. Bit)
 0001 0000 entspricht der hexadezimalen 0x10 (4. Bit)
 0010 0000 entspricht der hexadezimalen 0x20 (5. Bit)
 0100 0000 entspricht der hexadezimalen 0x40 (6. Bit)
 1000 0000 entspricht der hexadezimalen 0x80 (7. Bit)

Will man also im Port B das vierte Bit setzen (ACHTUNG: Zählung beginnt von 0!), ver-ODERt man Port B mit einer 0x10. Alle anderen Bit bleiben gemäß der Definition des ODER Operators unverändert.

```
PORTE = PORTE | 0x10;
```

oder in der Kurzschreibweise:

```
PORTE |= 0x10 ;
```

Eine weitere Variante besteht darin, eine 1 durch Linksschieben an die richtige Stelle zu setzen, dafür sind in den Include-Files bereits die notwendigen Definitionen verfügbar, im Fall des Port B gibt es PB0 (=0), PB1 (=1), PB2 (=2), PB3 (=3), PB4 (=4) usw. bis PB7 (=7).

Der Zugriff heißt also

```
PORTE |= (1 << PB4);
```

Mit anderen Worten, die 1 wird um vier Stellen nach links verschoben (Resultat binär: 0001 0000) und dann mit dem Inhalt des Registers PORTE ver-ODERt. Da die Null das neutrale Element der ODER-Operation darstellt, wird nur das vierte Bit von PORTE auf 1 gesetzt, die anderen bleiben unverändert.

Will man im umgekehrten Fall ein Bit auf 0 setzen, muss man das Register mit dem Einer-Komplement ver-UND-en, im Fall des vierten Bits also mit 1110 1111. Gemäß der Definition des UND-Operators bleiben damit alle anderen Bits unbeeinflusst. Um keine

komplizierten Umrechnungen machen zu müssen, wird das Komplement mit dem Negationsoperator ausgerechnet:

```
PORTRB &= ~0x10 ; //oder in Schiebeschreibweise
PORTRB &= ~(1 << PB4) ;
```

setzt also das vierte Bit auf 0, alle anderen bleiben so wie sie vor der Operation waren.

Ein weiterer Fall, in dem diese Technik sinnvoll angewendet wird, ist die Abfrage eines einzelnen Bits, beispielsweise um den Zustand eines Ports zu prüfen. In diesem Fall kann man aus dem Register alle außer dem interessierenden Bit ausmaskieren. Der Ausdruck

```
(PINB & (1 << PB5))
```

ist 0, wenn das fünfte Bit im Register `PINB` 0 ist, sonst ungleich 0. Hier hilft uns die Definition von logischen Ausdrücken in C, die besagt, dass ein Ausdruck „wahr“ ist, wenn er ungleich 0 ist. Wenn er gleich 0 ist, ist er „falsch“. Somit führt die Abfrage

```
if(PINB & (1 << PB7))
{
}
```

dann in die Klammer hinein, wenn in `PINB` das siebte Bit auf 1 gesetzt ist.

Selbstverständlich können alle Bitmasken kaskadiert werden:

```
DDRC &= ~(1 << DDC0) & ~(1 << DDC1) & ~(1 << DDC2) & ~(1 << DDC3) ;
```

alternativ:

```
DDRC &= ~((1 << DDC0) | (1 << DDC1) | (1 << DDC2) | (1 << DDC3)) ;
```

maskiert das Register `DDRC` mit 11110000, setzt also die vier niedrigsten Bits des Registers auf 0.

```
PORTRB |= (1 << PB3) | (1 << PB2) | (1 << PB0) ;
```

setzt die Bits 0, 2 und 3 des Registers `PORTRB` auf 1.

Interessant wird diese „Schiebeschreibweise“ dann, wenn die Zählung der Bits nicht mehr wirklich logisch ist. Hierzu werden in den folgenden Abschnitten noch reichliche Beispiele geboten. An dieser Stelle sei darauf hingewiesen, dass eine umfangreiche Funktionsbibliothek AVR Libc existiert, die beispielsweise mit dem Atmel Studio oder dem WinAVR Compiler mitkommt. In dieser Bibliothek findet sich eine reiche Anzahl von Funktionen, die einfacher zu bedienen sind als die Registerprogrammierung dies zulassen würde. Wir verwenden, wo möglich und sinnvoll, diese Funktionen in den weiteren Teilen des Buches. Eine ausführliche Beschreibung ist in [3] zu finden.

3.5 Digitaler Input/Output

3.5.1 Grundsätzlicher Aufbau

Nahezu jeder Pin eines AVR Prozessor, mit Ausnahme der Stromversorgungen für Prozessor und A/D-Wandler und der Referenzspannungseingänge, ist als digitaler Ein- oder Ausgang schaltbar. Abb. 3.5 zeigt grob den Aufbau.

Abb. 3.6 zeigt mögliche Eingangsbeschaltungen. Statt eines Tasters können natürlich auch die Endstufe einer digitalen Schaltung oder der Fototransistor eines Optokopplers angenommen werden.

Das Datenrichtungs-Register (DDRxⁿ³) bestimmt nun, ob der Pin einen Ein- oder einen Ausgang darstellt. Wird es auf 0 gesetzt (bei Reset automatisch), ist der Ausgangsschalter auf „tristate“ also hochohmig abgeschaltet und die Betriebsart ist „Eingang“. In diesem Fall lässt sich über eine 1 im Port-Register der Pullup-Widerstand zuschalten. Alle Pull-up-Widerstände lassen sich unabhängig davon durch ein gesetztes Bit 4 (PUD=Pull-up Disable) im MCU-Control Register (MCUCR) ausschalten:

```
MCUCR |= (1 << PUD);
```

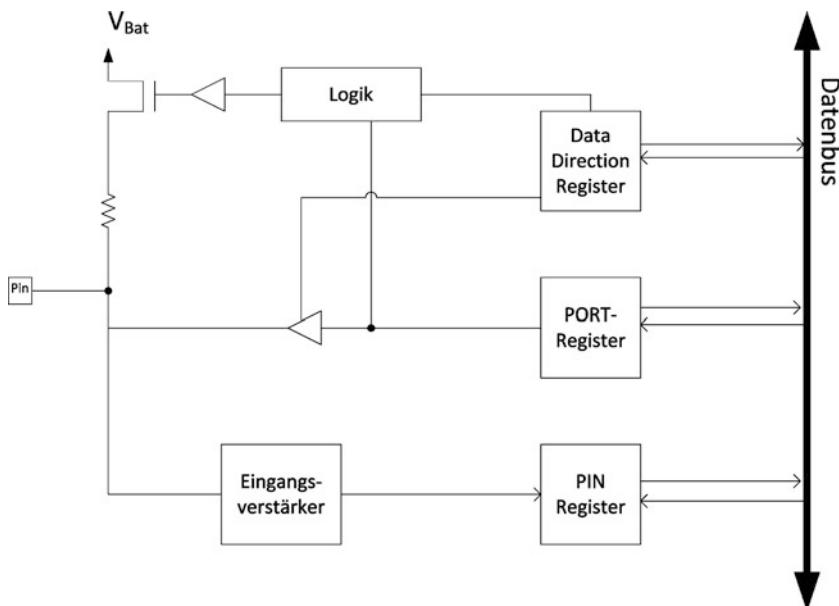
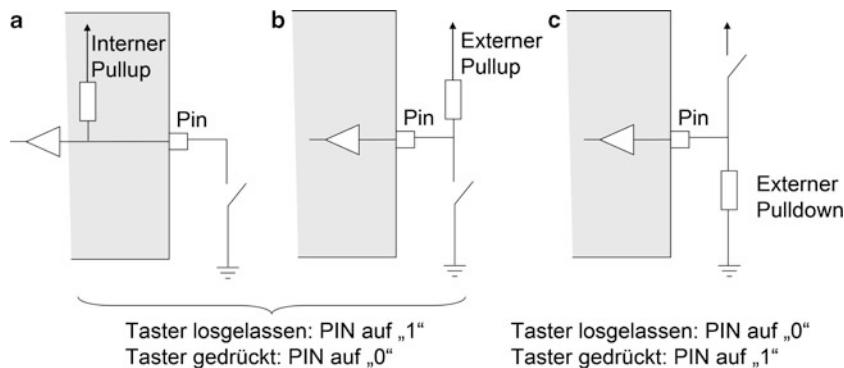


Abb. 3.5 Vereinfachtes Prinzipbild der digitalen Ein- und Ausgänge. (Nach [1])

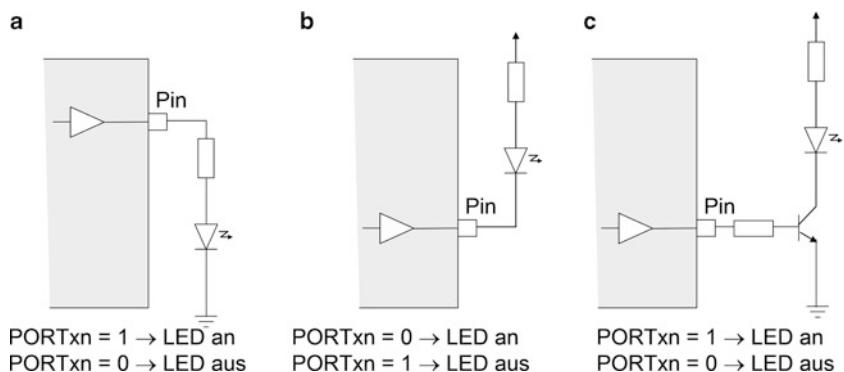
³ n steht für das entsprechende Bit, x steht für „B“, „C“ oder „D“ im Fall des ATmega88, also für eines der drei Port-Register.

**Abb. 3.6** Beschaltungsmöglichkeiten für digitale Eingänge

Hinweis: Bei allen Eingängen sollte auf einen definierten Pegel geachtet werden, indem entweder der interne Pullup-Widerstand aktiviert wird oder durch eine externe Beschaltung ein Pegel eingestellt wird.

Der Pegel am Eingang kann über das Register PINxn gelesen werden. Dieses nimmt durch die Synchronisierschaltung den Eingangsspegl nach der nächsten Taktflanke an.

Abb. 3.7 zeigt mögliche Beschaltungen für digitale Ausgänge. Wird das DDRxn auf 1 gesetzt, ist der Ausgangstreiber aktiv, der Port ist als Ausgangsport im so genannten „Push-Pull-Betrieb“ geschaltet. Abhängig von PORTxn ist der Pin entweder High, also niederohmig an der Versorgungsspannung (PORTxn = 1) oder Low, niederohmig an Masse (PORTxn = 0). Dabei kann der Baustein maximal 40 mA treiben. Abhängig von der äußeren Beschaltung kann aber die Spannung auch bei geschaltetem High-Pegel geringer ausfallen. Mit Hilfe des PINxn-Registers kann der reale Pegel am Ausgang mit einer Taktzeit Verzögerung ausgelesen werden.

**Abb. 3.7** Beschaltungsmöglichkeiten für digitale Ausgänge

Tab. 3.1 Betriebsarten der digitalen Ein/Ausgänge

Betriebsart	DDRx	PORTx	PINx	Elektrischer Anschluss
Eingang (tristate)	0	0	Offen	Tristate
Eingang (high)	0	1	1 (0 wenn an Masse)	High über ca. 30 kΩ
Ausgang	1	0	0	Low
Ausgang	1	1	1	High

Hinweis: Unabhängig von DDRxn kann bei einigen Prozessoren durch Schreiben einer 1 in das PINxn Register der entsprechende Port getoggelt, also umgeschaltet werden. Damit ist es möglich, den Pegel eines Anschlusses unabhängig von seinem aktuellen Zustand zu invertieren. Da diese Funktion nicht von allen Prozessoren unterstützt wird, sollte man sie nicht nutzen und stattdessen eine eigene Toggle-Funktion schreiben, falls dies nötig ist. Mehr dazu im Abschn. 3.5.2.

Mit einem externen Pullup-Widerstand kann der Pegel am Pin auch umgeschaltet werden, indem PORTxn zu 0 gesetzt und mit DDRxn umgeschaltet wird (DDRxn=1 → Ausgang Low, DDRxn=0 → Ausgang über externen Pullup auf High). Dies entspricht einem Open-Drain (ähnlich Open-Collector in TTL-Logik) Betrieb.

In Wirklichkeit ist die Beschaltung nach Abb. 3.5 etwas komplizierter, denn bei allen Pins sind auch noch alternative Funktionen darstellbar. Im Handbuch des Prozessors ist detailliert aufgelistet, unter welchen Bedingungen die digitalen I/O Funktionen und wann die alternativen Funktionen aktiv sind.

Tab. 3.1 fasst die wichtigsten Betriebsarten nochmals zusammen.

3.5.2 Programmierung

Die Programmierung der digitalen Ein/Ausgänge erfolgt, wie bereits oben besprochen, durch Registerzugriff. Im folgenden Beispiel sei an PORTB am Ausgang PB2 eine LED nach dem Muster von Abb. 3.7a angeschlossen.

Um für die Anwendungsprogrammierung eine Hardware-Abstraktion zu erhalten, schreibt man in einem Modul LED.c (Entsprechend natürlich mit einem Headerfile LED.h) die Funktionen `LED_Init()`, `LED_On()` und `LED_Off()`. Alternativ kann man sich natürlich auch die Funktion `LED(char state)` vorstellen, über den Eingangsparameter `state`, der die Werte ON und OFF annehmen kann, wird gesteuert, ob die LED an oder aus sein soll.

Das File LED.h sieht wie folgt aus:

```
#ifndef LED_H_
#define LED_H_

#define OFF 0
#define ON 1
```

```
/* Deklaration der Funktionen*/
void LED_Init(void);
void LED_On();
void LED_Off();
void LED(unsigned char state);

#endif /* LED_H_ */
```

Im File LED.c sind die eigentlichen Routinen ausgeführt:

```
#include <avr/io.h>
#include "LED.h"

void LED_Init()
{
    DDRB |= (1 << DDB2); //Datenrichtungsregister auf 1 setzen: Ausgang
    PORTB &= ~(1 << PB2); //LED zunächst aus
}

void LED_On()
{
    PORTB |= (1 << PB2);
}

void LED_Off()
{
    PORTB &= ~(1 << PB2);
}

void LED(unsigned char state)
{
    if (ON==state) { LED_On(); }
    else { LED_Off(); }
}
```

Manche Leser werden sich im Vergleich `if (ON==state)` wundern, warum die Konstante `ON` links steht und nicht die Variable `state`. Die Erklärung findet sich in Abschn. 2.6.2: Beim Programmieren verwechseln selbst erfahrene Programmierer den Zuweisungsoperator (`=`) mit dem Vergleichsoperator (`==`). Dem Compiler fällt dies nicht auf, da es sich in beiden Fällen um eine syntaktisch korrekte Anweisung handelt. Ge-wöhnt man sich daran, dass die zu vergleichende Konstante links steht (*lvalue*), würde der Compiler bei einer versehentlichen Zuweisung mit einer Fehlermeldung reagieren, da als *lvalue* bei einer Zuweisung immer eine Variable stehen muss.

Eine weitere Funktion zum Toggeln der LED, also zum Umschalten bei jeder Ansteuerung, kann wie folgt aussehen:

```
void LED_toggle()
{
    if (PORTB & (1 << PB2)) PORTB &= ~(1 << PB2);
    else PORTB |= (1 << PB2);
}
```

Der Bedingungsausdruck `PORTB & (1<<PB2)` nimmt für jedes Bit den Wert 0 an, außer dem Bit 2. Ist dieses in `PORTB` gesetzt, ist der ganze Ausdruck ungleich 0 und damit logisch „wahr“. Ist es nicht gesetzt, ist der Ausdruck 0 und damit „falsch“. Damit wird der `else`-Zweig ausgeführt.

Alternativ kann die Funktion auch so geschrieben werden, dass das Port-Register bitweise XOR 1 gesetzt wird:

```
void LED_toggle()
{
    PORTB ^= (1 << PB2);
}
```

Die beiden Files können nun in die Grundstruktur aus Abschn. 2.10 eingebunden werden. Selbstverständlich muss `LED.h` durch die entsprechende `#include`-Anweisung im Hauptprogramm eingebunden sein. `LED.c` kann auch Teil einer Bibliothek sein. Die Initialisierungsfunktionen (auch die folgenden) werden in der `Init()`-Funktion aus Abschn. 2.10 nacheinander aufgerufen. Manchmal muss eine Initialisierung auch im laufenden Programm wiederholt werden.

Im zweiten Anwendungsbeispiel seien vier Taster oder andere schaltende Bauteile nach Abb. 3.6a oder b in den Ports D2, D3, D4 und D5 eingebaut. Die Initialisierungsfunktion in einem File `key.c` sieht mit externem Pullup wie folgt aus:

```
void key_Init(void)
{
    DDRD &= (~(1 << DDD2) &~(1 << DDD3) &~(1 << DDD4) &~(1 << DDD5));
}
```

Möchte man stattdessen den internen Pullup nutzen, kann man die Funktion so schreiben:

```
void key_Init(void)
{
    DDRD &= (~(1 << DDD2) & ~(1 << DDD3) & ~(1 << DDD4) & ~(1 << DDD5));
#ifndef INTERNAL_PULLUP
```

```
MCUCR &= ~(1 << PUD); //Pullup Disable auf 0  
PORTD |= (1 << PD2) | (1 << PD3) | (1 << PD4) | (1 << PD5);  
//Pullup einschalten  
#endif  
}
```

In diesem Fall kann durch ein `#define INTERNAL_PULLUP` in einem generell einzubindenden Konfigurationsfile der interne Pullup zugeschaltet werden. Diese Technik der Compilersteuerung ist sehr beliebt, zumal die Defines auch aus der IDE beziehungsweise aus der Compileraufrufzeile mit dem Parameter `-D` ausgeführt werden können. So kann man den Code effektiv und schlank an die Hardware anpassen.

Die Abfrage der Tasten erfolgt durch Ausmaskieren:

```
char key_get(char key)  
{  
    return !(PIND & (1 << key));  
}
```

Die Funktion `key_get()` liefert einen von 0 verschiedenen Wert wenn die Taste gedrückt ist und 0, wenn die Taste losgelassen wurde. Im `key.h` File können griffige Namen für die Tasten definiert werden, beispielsweise:

```
#define keyS1 PIND2  
#define keyS2 PIND3  
#define keyS3 PIND4  
#define keyS4 PIND5
```

Durch die Abfrage

```
if (key_get(keyS1)) LED_On();  
else LED_Off();
```

wird also die LED aus dem obigen Beispiel eingeschaltet, wenn der Schalter S1 gedrückt wurde. Hinweis: Hier wird ausgenutzt, dass in C ein von 0 verschiedener Wert in einem Bedingungsausdruck als „wahr“ interpretiert wird.

Im Kap. 4 werden wir im Rahmen des Softwareframeworks auf die Abfrage und insbesondere auch das Entprellen von Tastern eingehen.

3.6 Interrupts

3.6.1 Einstieg in den Umgang mit Interrupts

Ein Interrupt ist eine Unterbrechung des normalen Programmablaufs⁴. Sobald ein Interrupt angefordert wird (das auslösende Ereignis wird Unterbrechungsanforderung oder Interrupt Request, IRQ genannt), pausiert das normale Programm und eine Interruptfunktion (Interrupt Service Routine ISR) wird ausgeführt. Ist diese abgearbeitet, wird das normale Programm fortgesetzt.

Der ATmega88 kennt 26 verschiedene Interrupts, die durch die Peripherie ausgelöst werden können und in Tab. 3.2 aufgelistet sind. In der Folge werden diese näher erläutert.

Damit hat man bei der Abfrage von Peripherieschnittstellen grundsätzlich zwei Betriebsmöglichkeiten: Den sequenziellen Abfragebetrieb (Polling) oder den Interruptbetrieb. Beim Polling werden in einer Endlosschleife alle Peripheriebausteine hintereinander abgefragt. Im Interruptbetrieb muss man zunächst die Interruptquelle aktivieren und natürlich eine ISR bereitstellen. Der Speicher enthält eine Sprungtabelle mit den Speicheradressen aus Tab. 3.2, dort steht die Einsprungadresse der jeweiligen ISR. Diese wird über ein Makro vom Compiler festgelegt. Man nennt diese Einsprungtabelle.

Sobald das Interruptereignis auftritt, speichert der Prozessor in einem Stack sowohl den Wert des Programmzählers als auch die verwendeten Register und arbeitet die ISR ab. Anschließend werden die Register und der Programmzähler aus dem Stack zurückgeladen und das Programm damit an der ursprünglichen Stelle fortgesetzt. Damit ein Interrupt ausgelöst werden kann, müssen also drei Bedingungen erfüllt sein:

1. Freigabe der Interrupts im Statusregister durch `I = 1`
2. Freigabe des betreffenden Interrupts in einem Maskenregister
3. Auftreten des Interrupt-Ereignisses

Die Freigabe aller Interrupts erfolgt in vielen Prozessoren, so auch generell in der AVR-Familie durch das Makro `sei()`, das im Fall der AVR-Familie im File `avr/interrupt.h` definiert ist. Dieses .h-File muss dort eingebunden werden, wo entweder ISR definiert werden oder die Makros `sei()` und `cli()` verwendet werden. `cli()` dient zum grundsätzlichen Sperren von Interrupts. Da die AVR-Familie keine Interrupt-Priorität kennt, kann mit diesem Makro verhindert werden, dass während der Abarbeitung einer ISR ein weiterer Interrupt eintrifft.

Jedem der oben aufgeführten Interrupts ist eine Interrupt Service Routine (ISR) zugeordnet. In dieser ISR werden die Anweisungen hinterlegt, die im Falle des Interrupts ausgeführt werden sollen. Bei der Definition der ISR hilft das Makro `ISR(..._vect)`, das ebenfalls in `avr/interrupt.h` definiert ist. Exemplarisch sind in Tab. 3.3 die Namen der Makros aufgeführt.

Für das RESET ist keine ISR hinterlegt, denn diese ist bekanntlich die `main()` Funktion.

⁴ Aus dem Englischen: to interrupt für unterbrechen, Lateinisch: interrumpere.

Tab. 3.2 Interrupts des ATmega88 [1]

Vektor Nummer	Programm Adresse	Ausgelöst durch	Interrupt Definition
1	0x000	RESET	External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	0x001	INT0	External Interrupt Request 0
3	0x002	INT1	External Interrupt Request 1
4	0x003	PCINT0	Pin Change Interrupt Request 0
5	0x004	PCINT1	Pin Change Interrupt Request 1
6	0x005	PCINT2	Pin Change Interrupt Request 2
7	0x006	WDT	Watchdog Time-out Interrupt
8	0x007	TIMER2 COMPA	Timer/Counter2 Compare Match A
9	0x008	TIMER2 COMPB	Timer/Counter2 Compare Match B
10	0x009	TIMER2 OVF	Timer/Counter2 Overflow
11	0x00A	TIMER1 CAPT	Timer/Counter1 Capture Event
12	0x00B	TIMER1 COMPA	Timer/Counter1 Compare Match A
13	0x00C	TIMER1 COMPB	Timer/Counter1 Compare Match B
14	0x00D	TIMER1 OVF	Timer/Counter1 Overflow
15	0x00E	TIMER0 COMPA	Timer/Counter0 Compare Match A
16	0x00F	TIMER0 COMPB	Timer/Counter0 Compare Match B
17	0x010	TIMER0 OVF	Timer/Counter0 Overflow
18	0x011	SPI, STC	SPI Serial Transfer Complete
19	0x012	USART, RX	USART Rx Complete
20	0x013	USART, UDRE	USART, Data Register Empty
21	0x014	USART, TX	USART, Tx Complete
22	0x015	ADC	ADC Conversion Complete
23	0x016	EE	EEPROM Ready
24	0x017	ANALOG COMP	Analog Comparator
25	0x018	TWI	2-wire Serial Interface
26	0x019	SPM READY	Store Program Memory Ready

3.6.2 Interrupt-Programmierung am Beispiel des Pin Change Interrupt

Ein Pin Change Interrupt (PCI) wird ausgelöst, wenn sich ein Bit eines Ports verändert, also z. B. eine Taste gedrückt wird. Die AVR Familie kennt die einfachen PCI, die über jeden Port ausgelöst werden können und die so genannten External Interrupts, die beim betrachteten ATmega88 nur von den Pins INT0 und INT1 ausgelöst werden können. Um einen Interrupt auslösen zu können, müssen verschiedene Register gesetzt werden. In Fall des einfachen PCI ist es das PCICR (Pin Change Interrupt Control Register) und eines der PCMSKn (Pin Change Mask Register). Dabei kann jeder einzelne Pin eines der 3 Ports einen Interrupt auslösen, sobald sich sein Wert ändert (positive und negative Flanke). Jeder Port bildet eine eigene Gruppe von Interrupts (PORTB: PCINT 0..7, PORTC:

Tab. 3.3 Namen der ISR für den ATmega88

Vektor Nummer	Ausgelöst durch	Interrupt Service Routine ISR (..._vect) {...}
1	RESET	
2	INT0	INT0_vect
3	INT1	INT1_vect
4	PCINT0	PCINT0_vect
5	PCINT1	PCINT1_vect
6	PCINT2	PCINT2_vect
7	WDT	WDT_vect
8	TIMER2 COMPA	TIMER2_COMPA_vect
9	TIMER2 COMPB	TIMER2_COMPB_vect
10	TIMER2 OVF	TIMER2_OVF_vect
11	TIMER1 CAPT	TIMER1_CAPT_vect
12	TIMER1 COMPA	TIMER1_COMPA_vect
13	TIMER1 COMPB	TIMER1_COMPB_vect
14	TIMER1 OVF	TIMER1_OVF_vect
15	TIMER0 COMPA	TIMER0_COMPA_vect
16	TIMER0 COMPB	TIMER0_COMPB_vect
17	TIMER0 OVF	TIMER0_OVF_vect
18	SPI, STC	SPI_STC_vect
19	USART, RX	USART_RX_vect
20	USART, UDRE	USART_UDRE_vect
21	USART, TX	USART_TX_vect
22	ADC	ADC_vect
23	EE	EE_READY_vect
24	ANALOG COMP	ANALOG_COMP_vect
25	TWI	TWI_vect
26	SPM READY	SPM_READY_vect

PCINT 8..14⁵, PORTD: PCINT 15..23). Es gilt die in Tab. 3.4, 3.5 und 3.6 genannte Zuordnung:

Soll der Taster am PIND4 aus Abschn. 3.5.2 einen Interrupt auslösen, muss das entsprechende Bit PCINT20 im Register PCMSK2 gesetzt werden.

Tab. 3.4 Port B – PCMSK0 (Pin Change Mask Register 0)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PCINT7	PCINT6	PCINT5	PCINT4	PCINT3	PCINT2	PCINT1	PCINT0

⁵ Port C hat nur sieben Eingänge.

Tab. 3.5 Port C – PCMSK1 (Pin Change Mask Register 1)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
–	PCINT14	PCINT13	PCINT12	PCINT11	PCINT10	PCINT9	PCINT8

Tab. 3.6 Port D – PCMSK2 (Pin Change Mask Register 2)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PCINT23	PCINT22	PCINT21	PCINT20	PCINT19	PCINT18	PCINT17	PCINT16

Tab. 3.7 Pin Change Interrupt Control Register – PCICR

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
–	–	–	–	–	PCIE2	PCIE1	PCIE0

Anschließend muss noch die gesamte PCI-Gruppe freigeschaltet werden, dies geschieht im Register PCICR (Tab. 3.7).

Der Code für die Initialisierungsroutine sieht dementsprechend so aus:

```
void PCI_Init(void)
{
    PCICR |= (1 << PCIE2); //PCI Gruppe 2
    PCMSK2 |= (1 << PCINT20); //PCI an PORTD PD4 frei
    sei();
}
```

Die Interrupt Service Routine für den PCI kann dann so aussehen:

```
ISR (PCINT1_vect)
{
    cli(); //andere Interrupts ausschalten, falls gewünscht
    uiPCIcnt++;
    sei(); //andere Interrupts wieder einschalten
}
```

Die globale Variable `uiPCIcnt` zählt in diesem Fall nur die Flanken. Sie wird im Bereich der modulglobalen Variablen aus Abschn. 2.10 definiert.

```
unsigned int uiPCIcnt = 0;
```

Generell sollten Interrupt Service Routinen möglichst kurz gehalten werden, insbesondere, wenn während der Abarbeitung der ISR weitere Interrupts zugelassen sein sollen (man spricht von „nested interrupts“). Weitere Beispiele für Interrupt Service Routinen folgen.

3.6.3 Die externen Interrupts INTx

Die Prozessoren der AVR Familie besitzen zusätzlich zu den Pin Change Interrupts deutlich mächtigere externe Interrupts, der ATmega88 zum Beispiel die Interrupts INT0 und INT1, die im Anschlussplan an den Pins D2 (Pin 4 im DIP Gehäuse) und D3 (Pin 5) liegen.

Über die Register EICRA – External interrupt control register A und EIMSK – External interrupt mask register werden sie gesteuert (Tab. 3.8).

Dabei haben die Bits die folgende Bedeutung (Tab. 3.9).

Bevor eine Interruptanforderung generiert werden kann, muss sie entsprechend freigeschaltet werden. Dies geschieht im Register EIMSK, wie in Tab. 3.10 beschrieben.

Wobei INT0 den externen Interrupt 0 und INT1 den externen Interrupt 1 einschaltet.

Im folgenden Codebeispiel wird der externe Interrupt 0 auf eine steigende Flanke getriggert.

```
void INT0_Init()
{
    EICRA |= (1 << ISC00) | (1 << ISC01);
    EIMSK |= (1 << INT0);
}
```

Im zweiten Beispiel wird er auf einen 0-Pegel getriggert:

```
void INT0_Init()
{
    EICRA &= ~(1 << ISC00) & ~(1 << ISC01);
    EIMSK |= (1 << INT0);
}
```

Tab. 3.8 EICRA – External interrupt control register A

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
–	–	–	–	ISC11	ISC10	ISC01	ISC00

Tab. 3.9 Steuerung des Interruptverhaltens beim externen Interrupt

ISCx1	ISCx0	Beschreibung
0	0	„Low“-Pegel (0) an INTx (0 oder 1) generiert eine Interruptanforderung
0	1	Irgendeine Pegeländerung an INTx generiert eine Interruptanforderung
1	0	Eine fallende Flanke an INTx generiert eine Interruptanforderung
1	1	Eine steigende Flanke an INTx generiert eine Interruptanforderung

Tab. 3.10 EIMSK – External interrupt mask register

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
–	–	–	–	–	–	INT1	INT0

Die passende ISR toggelt die LED wie oben beschrieben:

```
ISR (INT0_vect)
{
    LED_toggle();
}
```

3.7 Timer

Ein Herzstück der Peripherie jedes Mikrocontrollers und gleichzeitig die vielseitigsten Bausteine bilden die Timer. Sie erzeugen definierte Zeitintervalle, zählen Ereignisse und können für die PWM-Ansteuerung von Aktoren oder für die Zeitmessung verwendet werden.

3.7.1 Timer Grundlagen

Der ATMega88 enthält 3 Timer/Counter TCNT. TCNT0 und TCNT2 sind 8 Bit, TCNT1 16 Bit. Sie werden durch Register initialisiert und laufen programmunabhängig. Beim Erreichen eines bestimmten Zustandes erzeugen sie einen Interrupt oder setzen einen Ausgangsport.

Herzstück des Timer/Counters ist der Zähler (Counter). Er zählt mit 8 oder 16 Bit die Ereignisse, die entweder von einem speziell ausgewiesenen Pin kommen oder vom Systemtakt. Weil dieser in der Regel viel zu schnell ist um eine vernünftige Zeitmessung zu realisieren, ist in jeden Timerbaustein ein sogenannter Prescaler (Vorteiler) integriert. Dieser Prescaler teilt dann den Systemtakt um einen einstellbaren Faktor. Ein typischer Systemtakt ist 18,432 MHz. Bei einem Faktor von 1024 erhält man damit genau 18 kHz als Timertakt.

In Abb. 3.8a sieht man den Eingang vom „Prescaler“ (Vorteiler). Dieser lässt sich mit den Bits CS10, CS11 und CS12 in den Timer/Counter Control Registern TCCR0B, TCCR1B und TCCR2B (je nach Timernummer) wie in Tab. 3.11 gezeigt auf f_{CLK_IO} , $f_{CLK_IO} / 8$, $f_{CLK_IO} / 64$, $f_{CLK_IO} / 256$ oder $f_{CLK_IO} / 1024$ setzen. Das Ganze geschieht über einen Multiplexer.

Hinweis: Die unterstrichene Zahl in der Tabelle ist die Timer-Nummer, kann also 0, 1 oder 2 sein.

Die Timer 0 und 1 können statt über den Prescaler auch über einen Pin angesteuert werden, namentlich die Pins T0 und T1, die auf Port PD4 und PD5 belegt sind. Somit ist der Baustein in der Lage Ereignisse zu zählen.

Um nun aus dem vorgeteilten Eingangstakt ein definiertes Zeitintervall zu machen, vergleicht man den Wert des Zählers mit einem im Output Compare Register (OCR 0/1/2) voreingestellten Wert. Wird dieser Wert erreicht, wird ein Interrupt ausgelöst (Compare Mode) und/oder ein Ausgangsbit gesetzt (damit kann direkt ein Hardware-Takt abge-

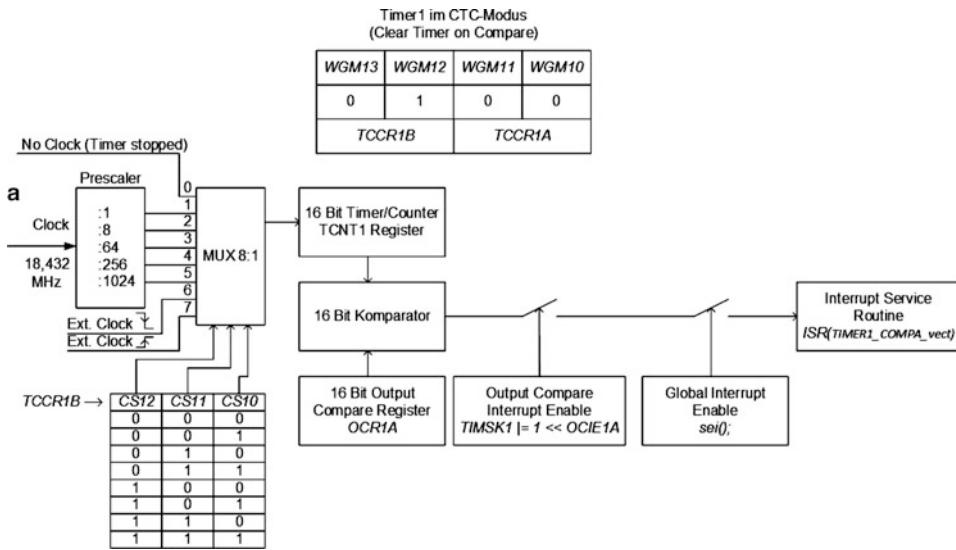


Abb. 3.8 Prinzipieller Aufbau des 16-Bit Timers 1 (analog zu den Timern 0 und 2)

leitet werden, ohne dass der µC belastet wird). Dabei können zwei verschiedene Werte zum Vergleich herangezogen werden (OCR A und B). Es ist zu beachten, dass dabei die Datenbreite des Timers begrenzt ist (das heißt, es können nicht beliebig große Zahlen verglichen werden). Nach dem Erreichen dieses Wertes kann der Wert des Zählers auf 0 gesetzt werden (Clear Timer on Compare CTC). So ist gewährleistet, dass der nächste Interrupt ebenfalls nach derselben Zeitspanne erfolgt.

Alternativ kann man den Interrupt auslösen, wenn der Zähler überläuft (d. h. von 0xFF auf 0x00 bzw. 0xFFFF auf 0x0000). Auch dieser Überlauf wird durch einen Interrupt angezeigt. Man könnte aus der ISR heraus den Timer auf einen bestimmten Wert initialisieren und damit denselben Effekt erzielen wie mit dem Output Compare Register. In der Regel wird aber für Zwecke der Zeitsteuerung der CTC Modus gewählt.

Tab. 3.11 Auswahl der Taktquelle im Register TCCR1B

CS12	CS11	CS10	Beschreibung
0	0	0	Keine Taktquelle (Timer anhalten)
0	0	1	f _{CLK_I/O} (vom Prescaler)
0	1	0	f _{CLK_I/O} / 8 (vom Prescaler)
0	1	1	f _{CLK_I/O} / 64 (vom Prescaler)
1	0	0	f _{CLK_I/O} / 256 (vom Prescaler)
1	0	1	f _{CLK_I/O} / 1024 (vom Prescaler)
1	1	0	Externe Taktquelle (Pin T0 bzw. T1) Taktung bei fallender Flanke
1	1	1	Externe Taktquelle (Pin T0 bzw. T1) Taktung bei steigender Flanke

3.7.2 Programmierung der Timer/Counter

Im Folgenden werden drei Anwendungsfälle der Timer näher betrachtet:

Fall 1: Erzeugen eines Sekundentaktes

Fall 2: Erzeugen einer festen Frequenz an einem Output-Pin

Fall 3: Erzeugen eines PWM Signals an einem Output-Pin

Zunächst werden dafür die Register für Timer 0 betrachtet. Die beiden anderen Timer des ATmega88 funktionieren entsprechend, die Zähl- und Vergleichsregister beim 16-Bit Timer 1 sind allerdings 2 Byte lang, außerdem beherrscht er doppelt so viele Modi und besitzt deshalb mehr Mode-Bits. Bei Timer 2 gibt es außerdem keine externen Taktquellen. In diesem Buch werden bewusst nur einige wenige Modi besprochen, es wird auf das sehr umfangreiche und gut beschriebene Manual [1] verwiesen. In den Tabellen Tab. 3.12 und 3.13 sind die Register TCCR0A und TCCR0B beschrieben. Diese dienen der Steuerung der Modi, des Verteilers und des Verhaltens der Timer an den Ausgängen.

Die Bedeutung der einzelnen Bits wird in den jeweiligen Beispielen näher erläutert. Die weiteren notwendigen Register sind:

- Das eigentliche Timer/Counter Register TCNT0
- Die beiden Output Compare Register OCR0A und OCR0B

und weiterhin das Timer Interrupt Mask Register (Tab. 3.14)

In diesem Register wird ausgewählt, ob ein „Vergleichsereignis“ des Output Compare Registers A oder B oder ein Overflow des Timers einen Interrupt auslösen sollen:

OCIEnB: Output Compare Interrupt Enable Output Compare Match B

OCIEnA: Output Compare Interrupt Enable Output Compare Match A

TOIEn: Timer Counter Overflow Interrupt Enable

Tab. 3.12 Timer/Counter Control Register TCCR0A

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
COM0A1	COM0A0	COM0B1	COM0B2	–	–	WGM01	WGM00

Tab. 3.13 Timer/Counter Control Register TCCR0B

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
FOC0A	FOC0B	–	–	WGM02	CS02	CS01	CS00

Tab. 3.14 Timer Interrupt Mask Register TIMSKn

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
–	–	–	–	–	OCIEnB	OCIEnA	TOIEn

3.7.2.1 Erzeugung eines Sekundentaktes

Ein empfohlener Schwingquarz für den ATmega88 erzeugt eine Frequenz von 18,432 MHz als Systemtakt. Wird diese Frequenz durch den Prescaler durch 1024 geteilt, dann ergibt sich eine Frequenz f von 18 kHz. Durch die Formel

$$T = \frac{1}{f} = \frac{1}{18 \text{ kHz}} = 55,5 \mu\text{s} \quad (3.1)$$

ergibt sich damit eine Periodendauer T von 55,5 μs . Dies bedeutet, dass der Interrupt alle 55,5 μs ausgelöst wird. Dies ist für unsere Anwendung natürlich zu kurz, deshalb muss dieser Systemtakt weiter geteilt werden. Das ist Aufgabe des Output Compare Registers. Im obigen Beispiel bewirkt der OCR0A-Wert von 179, dass nur bei jedem 180. Mal die Interrupt-Service-Routine aufgerufen wird (Bitte beachten: 0–179 sind 180 Durchläufe). Dadurch wird die ISR alle $180 \times 55,5 \mu\text{s} = 10 \text{ ms}$ aufgerufen. Damit haben wir aber immer noch keinen Sekundentakt. Deshalb wird in der ISR der Takt per Software nochmals um den Faktor 100 geteilt. Daraus ergibt sich der Takt von genau 1 s. Mit den oben beschriebenen Einstellungen wählen wir die Initialisierungsroutine wie folgt

```
void Timer0_Init()
{
    //Initialisierung Timer 0
    //Clear Timer on Compare (CTC)
    TCCR0A |= (1 << WGM01);
    //Prescaler auf Teilung durch 1024 setzen
    TCCR0B |= ((1 << CS02) | (1 << CS00));
    //Output Compare Register A belegen
    OCR0A = 179;
    //Interrupt Timer/C0 Output Compare Match A
    TIMSK0 |= (1 << OCIE0A);
    sei();
}
```

Durch Setzen von WGM01 wird der CTC (Clear Timer on Compare Match) Modus eingestellt (Tab. 3.15), folglich zählt der Timer bis 179 und fängt dann wieder mit 0 an. Im Folgenden sei zunächst dieser Modus näher betrachtet, später (Abschn. 3.7.2.5) auch einige der PWM Modi.

Tab. 3.15 Einige Waveform Generation Modes für Timer 0

WGM02	WGM01	WGM00	Beschreibung
...	
0	1	0	Timer nach Vergleich löschen (CTC)
0	1	1	Fast PWM
...	

Durch Setzen von OCIE0A im Register TIMSK0 wird der Interrupt ausgelöst und wir benötigen eine entsprechende ISR:

```
ISR (TIMER0_COMPA_vect)
{
    ucFlag10ms = 1;
    ucCount10ms++;
    if (ucCount10ms == 100)
    {
        ucCount10ms = 0;
        ucFlag1s = 1;
    }
}
```

Man beachte die drei globalen Variablen, die hier eingesetzt sind und die natürlich außerhalb einer Funktion deklariert werden müssen:

```
unsigned char ucFlag10ms = 0;
unsigned char ucFlag1s = 0;
unsigned char ucCount10ms = 0;
```

Nach jeweils 10 ms ist die Variable ucFlag10ms gesetzt, nach einer Sekunde auch die ucFlag1s, dies kann in einer Endlosschleife abgefragt werden, wo die Variablen dann auch sofort wieder zu 0 gesetzt werden müssen, damit sie nach den entsprechenden Zeiten wieder auf 1 stehen.

```
while(1)
{
    if (ucFlag1s)
    {
        ucFlag1s = 0;
        //Hier stehen die Dinge, die jede Sekunde ausgeführt werden sollen
    }
}
```

Werden die Timer in einem externes Modul Timer.c programmiert, dann müssen die beiden globalen Flags natürlich im File Timer.h als extern deklariert werden.

```
extern unsigned char ucFlag10ms;
extern unsigned char Flag1s;
```

Alternativ und meistens der bessere Weg ist es, statt der modulübergreifenden Flags im File Timer.c Abfrageroutinen zu definieren, die die Flags abfragen und auch wieder löschen. Dies wird im Kap. 4 näher erläutert.

3.7.2.2 Erzeugen einer festen Frequenz an einem Ausgang

In diesem Beispiel soll ein Signal einer bestimmten Frequenz auf den Ausgang PORTB1 gelegt werden, beispielsweise um einen Ton über einen Lautsprecher auszugeben. Die Frequenz soll einstellbar sein. Die Wahl des Ausgangs ist der Tatsache geschuldet, dass sechs Pins mit den Namen OC_{xy} (Abb. 3.2) direkt mit den Timern verbunden sind. Die Nummern *x* stehen für die Timernummer, die Buchstaben *y* für die Output Compare Register (A und B), also beispielsweise OC1A, das mit PORTB1 identisch ist.

Für dieses Beispiel benutzen wir den 16-Bit Timer 1. Er wird genau wie zuvor initialisiert, diesmal aber ohne Verteiler, d.h. mit dem Quarztakt f_{osc} am Eingang des Zählerregisters.

```
void Generator_Init(void)
{
    DDRB |= 1 << DDB1; //Pin 1 von Port B wird auf Ausgang gesetzt
    TCCR1A = (1 << COM1A0);
    //Timer 1 schaltet PB1 um wenn der Zähler TCNT1 = OCR1A
    TCCR1B = (1 << CS10); //Systemtakt ohne Vorverteiler (Prescaler)
    TCCR1B |= 1 << WGM12; //CTC-Modus gewählt
}
```

Der CTC-Modus sorgt dafür, dass der Timer nach jedem Compare Match auf 0 zurückgesetzt wird. Setzen des Bits COM1A0 bewirkt, dass bei jedem Compare Match das Pin 1 umgeschaltet wird. Damit ist die Frequenz am Pin 1

$$f = \frac{f_{clk_IO}}{2 \cdot n \cdot (OCR1A + 1)} \quad (3.2)$$

Die Zahl *n* steht hier für das Vorteilverhältnis, das im obigen Beispiel 1 beträgt. Die Frequenz f_{clk_IO} entspricht ohne weitere Einstellungen der Quarzfrequenz. Das OCR1A Register wird nun in einer weiteren Funktion gesetzt, die Variable uiFreq ist dabei natürlich nicht die Frequenz, diese leitet sich aus Gl. 3.2 ab.

```
void Generator_ChangeFreq(unsigned int uiFreq)
{
    OCR1A = uiFreq;
}
```

Man beachte, dass die 16-Bit Register von Timer 1 den Typ unsigned int haben. Um das Signal ein- und auszuschalten setzen oder löschen wir das Bit COM1A0, das den Ausgangspin mit dem Timer verbindet.

Tab. 3.16 Register zur Steuerung des Output-Verhaltens an den Pins OC1A und OC1B

COM1A1/ COM1B1	COM1A0/ COM1B0	Verhalten
0	0	Pins nicht an den Timer angeschlossen (normale Portausgänge)
0	1	Pins werden beim Output Compare Match umgeschaltet (toggle)
1	0	Pins werden beim Output Compare Match auf 0 gesetzt
1	1	Pins werden beim Output Compare Match auf 1 gesetzt

```

void Generator_Set_On(void)
{
    TCCR1A |= (1 << COM1A0);
}

void Generator_Set_Off(void)
{
    TCCR1A &= ~(1 << COM1A0);
}

```

Alternativ könnte man die Taktquelle abschalten, indem man im Register TCCR1B das Bit CS10 löscht beziehungsweise anschalten, indem man es setzt.

Generell ist die Bedeutung der Bits COM1A0, COM1A1, COM1B0 und COM1B1 im Register TCCR1A der Tab. 3.16 zu entnehmen. Dies gilt allerdings nur dann, wenn kein PWM-Modus gewählt ist. Die Bits COM1Ax steuern den Ausgang OC1A, die Bits COM1Bx den Ausgang OC1B.

3.7.2.3 Ausgabe eines PWM-Signals

In diesem Beispiel soll aus Port B1 ein PWM-Signal (Pulsweitenmodulation, englisch pulse width modulation) ausgegeben werden um einen Motor anzusteuern. Die Idee der Pulsweitenmodulation besteht darin, einen Rechteckimpuls mit verändertem Tastverhältnis⁶ D auszugeben. Der Mittelwert der Signalamplitude eines solchen Signals ist:

$$\bar{y} = \frac{1}{T} \int_0^T y(t) dt = \frac{1}{T} \left(\int_0^{DT} y_{\max} dt + \int_{DT}^T y_{\min} dt \right) \quad (3.3)$$

Und damit

$$\bar{y} = \frac{1}{T} (D \cdot T \cdot y_{\max} + T (1 - D) y_{\min}) = D \cdot y_{\max} + (1 - D) y_{\min} \quad (3.4)$$

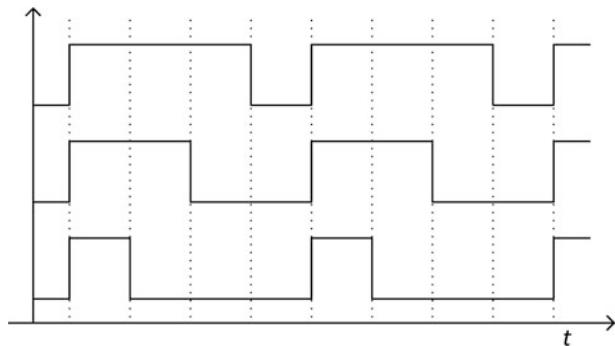
Wenn $y_{\min} = 0$ ist, wie in Abb. 3.9, dann ist

$$\bar{y} = D \cdot y_{\max} \quad (3.5)$$

und damit proportional zum Tastverhältnis.

⁶ Verhältnis zwischen Einschaltzeit und Gesamtzeit eines Rechteckssignals.

Abb. 3.9 Signal mit Tastverhältnissen (von oben) 75 %, 50 %, 25 %



Bei der Ausgabe eines Signals an einem Pin wie in Abschn. 3.7.2.2 kann man das Tastverhältnis einstellen, indem man den Timer auf den Maximalwert laufen lässt und beim Erreichen einer Schwelle den Pin umschaltet. Für den Timer 1 existieren zwölf verschiedene PWM-Arten. In der Folge werden zwei grundsätzliche Arten beschrieben.

In Abb. 3.10a sieht man die so genannte schnelle PWM (fast PWM). Hier läuft der Timer von 0 bis zum Maximalwert. Dieser kann bei Timer 1 entweder 0x00FF (8 Bit), 0x01FF (9 Bit), 0x03FF (10 Bit) oder 0xFFFF (16 Bit) betragen oder durch den Wert im Register OCR1A oder auf weitere Arten festgelegt werden⁷. Anschließend wird der Timer zurückgesetzt und beginnt bei 0 (BOTTOM) zu zählen. Erreicht der Timer den im OCRnX Register eingestellten Wert (X steht für A oder B), wird am Ausgang das entsprechende Bit gesetzt oder gelöscht. Durch `COM1A1 = 1` wird beispielsweise der Eingang OC1A gelöscht (=0), wenn der Vergleichswert im Register OCR1A erreicht ist. Beim Timerübergang und Rücksetzen auf 0 wird der Ausgang wieder gesetzt (Nichtinvertierender Modus).

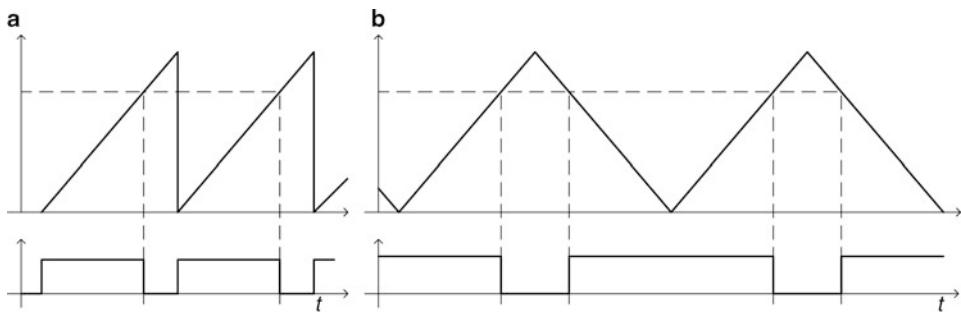


Abb. 3.10 Pulsweitenmodulation. **a** schnell, **b** phasen- und frequenzrichtig

⁷ Im Sinne der Lesbarkeit wird hier auf eine vollständige Darstellung verzichtet. In [1] und [4] sind ausführliche Beschreibungen.

Die Frequenz des PWM-Signals ergibt sich aus der Einstellung des Maximalwerts (auch TOP genannt) und dem Wert des Vorteilers. Im Fall der schnellen PWM ist dies – wieder mit n als Vorteilverhältnis –

$$f = \frac{f_{\text{clk_IO}}}{n \cdot (\text{TOP} + 1)} \quad (3.6)$$

In der Abbildung ist zu erkennen, dass sich in diesem Modus die ansteigende Flanke des Ausgangssignals, die ja aus dem periodischen Overflow resultiert, isochron wiederholt, während die abfallende Flanke mit dem Stand des Vergleichsregisters „wandert“. Die Phasenlage der Impulsmitte verändert sich damit. Dem gegenüber steht der phasenrichtige Modus (Abb. 3.10b). Hier zählt der Timer bis zum Maximalwert und zählt dann zu Null zurück. Das Output-Pin wird beim Hochzählen gelöscht, sobald der Vergleichswert erreicht wird und beim Herunterzählen gesetzt, sobald der Vergleichswert wieder unterschritten wird. Die Impulsmitte ist damit immer in Phase. Durch das Auf- und Abwärtszählen halbiert sich die Impulsfrequenz zu:

$$f = \frac{f_{\text{clk_IO}}}{2 \cdot n \cdot (\text{TOP})} \quad (3.7)$$

Der folgende Code zeigt die Initialisierung von Timer 1 für ein phasenrichtiges PWM-Signal an Ausgang OC1A mit 8 Bit Auflösung.

```
void Timer1_PWM_Init(void)
{
    DDRB |= (1 << DDB1); //Port B Bit 1 - PWM Ausgang
    TCCR1A = (1 << WGM10) | (1 << COM1A1);
    TCCR1B = (1 << CS10); //CS10 = 1 kein Prescaler (fOSC)
    OCR1A = 0; //Tastverhältnis auf 0 setzen
}
```

Das Tastverhältnis wird durch Setzen des Registers OCR1A eingestellt:

```
void Motor_Set_Speed(unsigned int uiSpeed)
{
    OCR1A = uiSpeed;
}
```

Der Unterschied zwischen fast PWM und phasenkorrigiertem PWM ist meist marginal, bei phasenkorrigiertem PWM sind die Impulse symmetrisch und für jedes Tastverhältnis in Phase. Dies kann bei manchen Brückenschaltungen erforderlich sein um die Kommunikationsströme zu begrenzen.

3.7.2.4 Entprellen der Tastatur mit dem Timer

Tasten können zwei Zustände annehmen „geschlossen“ und „geöffnet“. Dazwischen liegen die Ereignisse „Taste wurde gedrückt (pressed)“ und „Taste wurde losgelassen (released)“.

Diese Ereignisse lassen sich nicht einfach durch einen „pin change“ erfassen, denn Tasten „prellen“. Das heißt, nach dem mechanischen Auslösen kann es geschehen, dass das Potential am angeschlossenen Pin mehrfach um die Auslöseschwelle schwankt und damit der binäre Wert des Pins zwischen den Werten „0“ und „1“ hin- und herpendelt bis der Wert dann nach einigen Millisekunden zur Ruhe kommt (Abb. 3.11).

Um die Ereignisse „pressed“ und „released“ sicher zu detektieren, kann also kein Pin change interrupt herangezogen werden, vielmehr müssen im Abstand von ca. 5–10 ms die Tastenzustände überprüft werden und dann aufgrund der Änderung bestimmt werden, ob eine Taste aktuell gedrückt wurde oder nicht.

```
if (ucFlag10ms)
{
    ucFlag10ms = 0;
    event = keyCheckEvent();
}
```

Der passende „Eventgenerator“ sieht dann wie folgt aus (im key.h File Abschn. 3.5.2)

```
#define EVENT_void 0
#define EVENT_S1_PRESSED 1
#define EVENT_S1_RELEASED 10
```

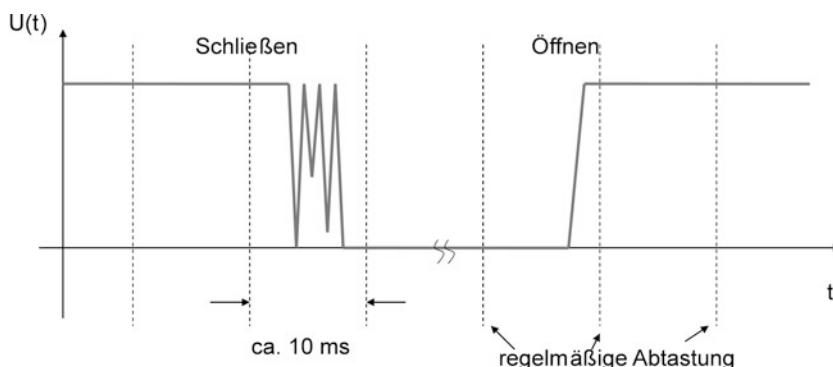


Abb. 3.11 Prellen eines Tasters

Der entsprechende Code lautet dann:

```
unsigned char S1Old;
unsigned char S1State;

char keyCheckEvent()
{
    S1Old = S1State; //Wie war die Taste vor 10 ms?
    S1State = key_get(keyS1); //Wie ist die Taste jetzt?
    if ((S1State) && (S1Old != S1State))
        return EVENT_S1_PRESSED;
    else if ((!S1State) && (S1Old != S1State))
        return EVENT_S1_RELEASED;
    else return EVENT_void;
}
```

Die Funktion liefert also zurück, ob die Taste seit dem letzten Aufruf der Funktion gedrückt (EVENT_S1_PRESSED) oder losgelassen (EVENT_S1_RELEASED) wurde. Falls nichts passiert ist, wird EVENT_void zurückgeliefert.

3.7.2.5 PWM für Fortgeschrittene

In der Tab. 3.17 sind die Timer-Modi in Abhängigkeit der Bits WGM13 und WGM12 im Register TCCR1B und WGM11 bzw. WGM10 im Register TCCR1A zusammengefasst.

Tab. 3.17 Timer-Modi des Timers 1

WGM13	WGM12	WGM11	WGM10	Modus	TOP
0	0	0	0	Normal	0xFFFF
0	0	0	1	PWM 8 Bit phasenkorrekt	0x00FF
0	0	1	0	PWM 9 Bit phasenkorrekt	0x01FF
0	0	1	1	PWM 10 Bit phasenkorrekt	0x03FF
0	1	0	0	CTC	OCR1A
0	1	0	1	Fast PWM 8 Bit	0x00FF
0	1	1	0	Fast PWM 9 Bit	0x01FF
0	1	1	1	Fast PWM 10 Bit	0x03FF
1	0	0	0	PWM phasen- und frequenzkorrekt	ICR1
1	0	0	1	PWM phasen- und frequenzkorrekt	OCR1A
1	0	1	0	PWM phasenkorrekt	ICR1
1	0	1	1	PWM phasenkorrekt	OCR1A
1	1	0	0	CTC	ICR1
1	1	0	1	Reserviert	
1	1	1	0	Fast PWM	ICR1
1	1	1	1	Fast PWM	OCR1A

Die PWM-Signale im phasenkorrekten und im phasen/frequenzkorrekten Modus unterscheiden sich nicht, solange sich der Maximalwert (Top), der die Frequenz vorgibt, und der Vergleichswert (OCRnx) (für das Tastverhältnis) nicht ändern. Die OCRnx Register sind doppelt gepuffert und werden in den zwei Modi einmal auf Top und einmal auf Bottom aktualisiert so wie in Abb. 3.12 zu sehen ist. Für eine genaue Beschreibung sei hier auf das Datenblatt verwiesen (Abschn. 16.10 in [1])

Ein typischer Anwendungsfall für eine ständige Änderung der OCRx-Register ist die Ausgabe von Audiodaten oder eines Sinussignals. Hierzu benötigt man zwei Timer. Einer ist für die Abtastung zuständig (Hier: Timer 0). Bei jedem Abtastimpuls wird ein neuer Wert am PWM-Ausgang erzeugt, indem das OCRxn Register des zweiten Timers (im folgenden Beispiel Timer 2) neu beschrieben wird. Dieser zweite Timer ist dann für die Ausgabe des PWM Signals zuständig, dessen Tastverhältnis bis zum folgenden Abtastimpuls anliegt.

Die Werte des OCRxn Registers werden in einer Tabelle gespeichert, die als Ganzahltyp den Verlauf des (integrierten) Ausgangssignals darstellt. Ein Quadrant eines Sinussignals ($0\dots 90^\circ$ bzw. $0\dots \pi/2$) mit einer Auflösung von 64 Zeitschritten und 128 Werteschritten sieht beispielsweise so aus:

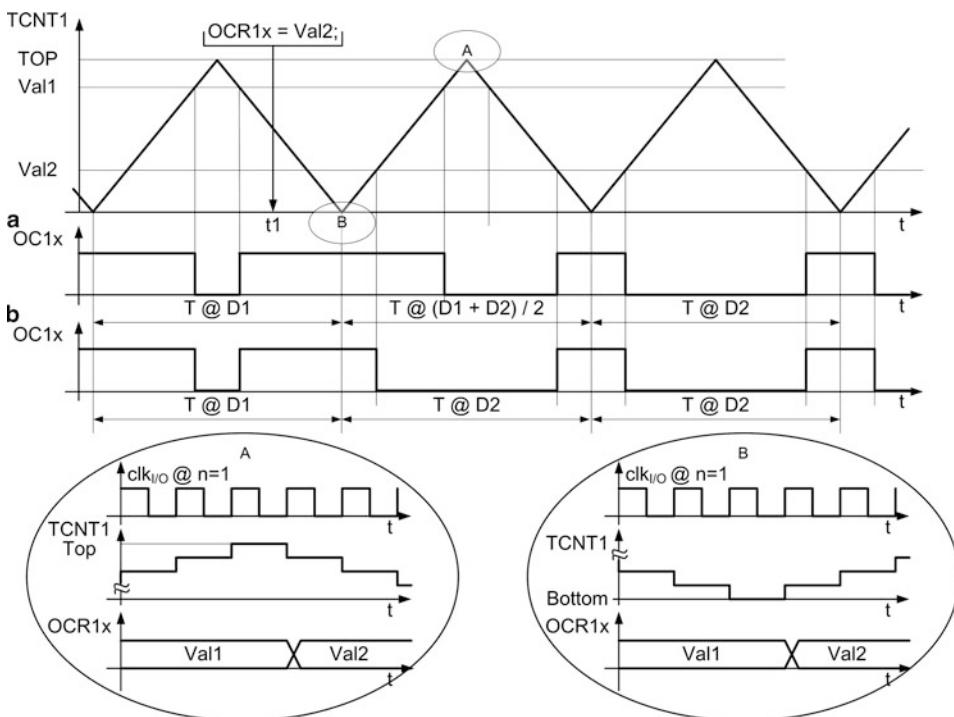


Abb. 3.12 Unterschied zwischen phasenkorrektem und phasen- und frequenzkorrektem PWM bei AVR. **a** phasenrichtiger PWM-Modus, **b** phasen- und frequenzrichtiger PWM-Modus

```
unsigned char pucSintab[] = { 0,3,6,9,12,16,19,22,25,28,31,34,37,40,43,
    46,49,51,54,57,60,63,65,68,71,73,76,78,81,83,85,
    88,90,92,94,96,98,100,102,104,106,107,109,111,
    112,113,115,116,117,118,120,121,122,123,124,
    125,125,126,126,126,127,127,127};
```

Um ein nach Integration vollständiges Sinussignal zu erhalten, addiert man für den ersten Quadranten mit jedem Ablauf von Timer 0 den Wert am dem Index der Tabelle, der dem jeweiligen Zeitpunkt entspricht, auf 128 (entspricht 50 % Tastverhältnis) auf. Im zweiten Quadranten zählt man die Tabelle rückwärts. Im dritten Quadranten zieht man die wieder vorwärts gezählten Werte von 128 ab, im vierten Quadranten durchläuft man die Liste wieder rückwärts. So setzt sich dann eine vollständige Sinuskurve zusammen. Bei einer Zeitauflösung von 64 Schritten pro Viertelwelle ergibt sich nach 256 Schritten eine volle Sinuskurve mit der Frequenz:

$$f_{\sin} = \frac{f_{\text{timer0}}}{256} \quad (3.8)$$

In der Interrupt Service Routine von Timer 0 (dem Abtasttimer) steht folgender Code⁸:

```
ucCnt1 += ucMult;
ucQuad = ucCnt1 / 64;
ucCnt = ucCnt1 % 64;
switch(ucQuad)
{
    case 0:  OCR2A = 128 + pucSintab[ucCnt++]; break;
    case 1:  OCR2A = 128 + pucSintab[63 - ucCnt++];break;
    case 2:  OCR2A = 128 - pucSintab[ucCnt++];break;
    case 3:  OCR2A = 128 - pucSintab[63 - ucCnt++];break;
}
```

ucCnt1 zählt bei jedem Timerinterrupt (dem Abtastzeitpunkt) bis 255 hoch und beginnt dann wieder bei 0, daraus wird der Tabellenindex ucCnt durch die Modulooperation jeweils von 0 bis 63 abgeleitet. Der Quadrant ucQuad entsteht durch eine ganzzahlige Division von ucCnt durch 64 im Bereich zwischen 0 und 255. Das Ergebnis ist entweder 0, 1, 2 oder 3. So wird am Ausgang OC2A ein PWM-Signal mit einem Tastverhältnis zwischen 0 (entspricht dem negativen Minimum) und 100 % (entspricht dem positiven Maximum der Sinuswelle) ausgegeben, also ein Sinus mit einem Offset von $U_{\text{Bat}}/2$. Voraussetzung ist, dass Timer 2 wie folgt konfiguriert ist:

```
void Timer2_PWM_Init(void)
{
    DDRB |= (1 << DDB3); //Port B Bit 3 - PWM Ausgang
```

⁸ Wir nehmen zunächst an, dass die Variable ucMult = 1 ist.

```

TCCR2A = (1 << WGM20) | (1 << COM2A1);
TCCR2B = (1 << CS20);
}

```

Abb. 3.13 verdeutlicht diesen Sachverhalt.

Durch die Wahl von 64 Abtastschritten (einem ganzzahligen Teiler von 256) ist es möglich, eine ganze Reihe von Sinussignalen zu erzeugen (nämlich die 1,2,3,4,5 ... fache Frequenz des oben beschriebenen Grundsignals), indem man den Zähler nicht um 1 sondern um 2,3,4,5 ... usw. erhöht. Die zeitdiskrete und wertediskrete Auflösung des Sinussignals wird dadurch allerdings schlechter. Die Frequenz des ausgegebenen Sinussignals ist dann:

$$f_{\text{sin}} = \frac{f_{\text{timer}0}}{256} \cdot \text{ucMult} \quad (3.9)$$

Damit das PWM-Signal vollständig erzeugt wird, muss Timer 2 theoretisch mindestens einen vollständigen PWM-Impuls pro Abtastschritt ausgeben. Praktisch sollten es etwa 3 bis 5 Impulse sein, damit das Signal korrekt integriert wird. Bei einer Auflösung von n Bit muss Timer 2 damit mindestens mit $3 \cdot 2^n f_{\text{timer}0}$ getaktet sein.

Natürlich kann man auch ganze Audioströme (soweit es der Speicher hergibt) damit abspielen, deren Abtastfrequenz man bei weitem nicht so fein zu machen braucht, denn es gilt das Nyquist-Shannon-Abtasttheorem, nach dem ein auf f_{max} bandbegrenztes Signal mit mindestens $2 \cdot f_{\text{max}}$ abgetastet werden muss, um es aus dem zeitdiskreten Signal vollständig zu rekonstruieren. Stellt man Timer 0 beispielsweise auf 125 µs ein, so ergibt sich eine darstellbare Bandbreite von 4 kHz, was für Sprachausgaben durchaus ausreicht. Timer 2 wäre dann bei einer völlig ausreichenden Auflösung von sieben Bit mit $128 \cdot 34 \text{ kHz} = 1,536 \text{ MHz}$ zu takten. Aus der Diskretisierung mit 7 Bit ergibt sich ein theoretischer Signal-Rauschabstand von $76 \text{ dB} = 42 \text{ dB}$ (also ein Verhältnis von Signal zu Diskretisierungsrauschen von 1:2¹⁴). Damit sind die Grenzen des Verfahrens durch den begrenzten Prozessortakt schnell erreicht, daher werden in der Praxis andere Modulations-

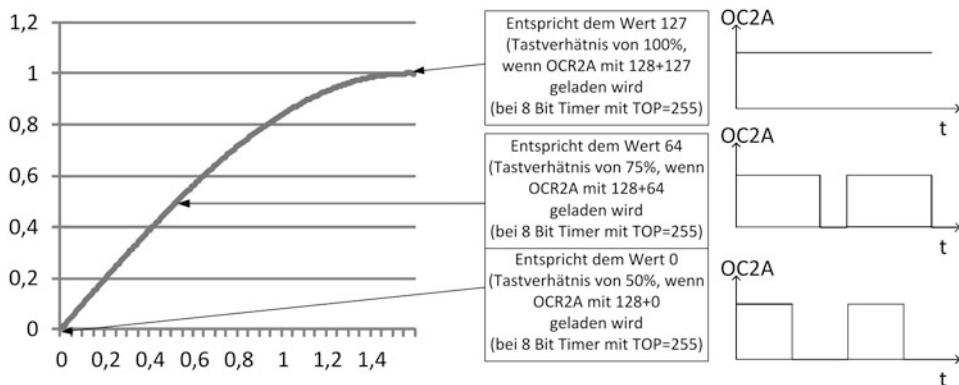


Abb. 3.13 Sinus-Viertelwelle aus einem PWM-Signal

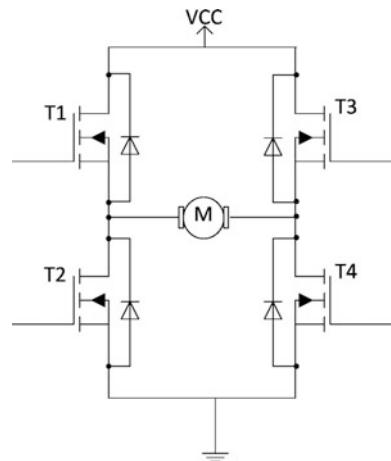
arten (Delta-Sigma-Modulation) verwendet, die pro Abtastschritt nur ein Bit Auflösung erfordern. Details über die Abtastung von Analogsignalen und weitere Literaturempfehlungen dazu können beispielsweise in [5] nachgelesen werden.

Mit einem externen Flashspeicher (Kap. 7) lässt sich so beispielsweise ein Anrufbeantworter oder eine Sprachausgabe realisieren.

3.7.2.6 Hardwareansteuerung von Motoren mit PWM-Signalen

Pulsweitenmodulation wird oft in Verbindung mit der so genannten H-Brücke, auch Vierquadrantensteller, eingesetzt. Das Prinzip ist in Abb. 3.14 dargestellt. Von den vier Transistoren sind jeweils 2 angesteuert. Schalten T1 und T4 dreht der Motor rechts herum, schalten T2 und T3 dreht er links herum. Schalten T2 und T4 wird der Motor gebremst. Die Dioden (Freilaufdioden) nehmen den Strom auf, der in der Schaltlücke durch die im Motor gespeicherte magnetische und mechanische Energie weiterfließt. Die Schaltung erfolgt so, dass T2 oder T4 (low-side-Treiber) durchgeschaltet werden, während auf den so genannten high-side-Treibern T1 und T3 die PWM-Signale angelegt werden. Aufpassen muss man natürlich, dass die Transistoren eines Strangs (T1 – T2 und T3 – T4) nicht gleichzeitig eingeschaltet werden. Dies würde zu einem Kurzschluss und zur Zerstörung der Brücke führen. Deshalb schützt in der Regel eine in Hardware aufgebaute Logik die Schaltung vor diesem Zustand und stellt dem Anwender lediglich einen PWM-Eingang und einen Richtungseingang zur Verfügung, der dann von einem zusätzlichen IO-Port des Mikrocontrollers angesteuert werden muss. Weiterhin müssen die beiden high-side Treibertransistoren mit speziellen Ladungspumpen angesteuert werden, damit die Gatespannung über der Sourcespannung liegt und damit höher als die Versorgungsspannung. In integrierten H-Brücken (z. B. ATA6823 von Microchip ohne Schalttransistoren oder L6205 von ST Microelectronics mit Schalttransistoren) sind Ladungspumpe, Logik und Leistungstreiber sowie Schutz vor Überhitzen und Überstrom in einem Gehäuse oder sogar auf einem Chip vorhanden.

Abb. 3.14 Vierquadrantensteller



Eine integrierte Brücke mit Richtungseingang hat die Besonderheit, dass man das PWM-Signal auch auf den Richtungseingang legen kann. Sobald der PWM-Eingang auf High ist, schaltet der Motor ein. Bei einem Tastverhältnis von 50 % bleibt er stehen, da er mit der PWM-Frequenz umgepolt wird, die resultierende Drehzahl ist damit 0 aber der Motor erzeugt ein Gegenmoment bei Stillstand. Tastverhältnisse kleiner 50 % führen dann zu Linkslauf, Tastverhältnisse größer 50 % zur Rechtslauf (wenn der Richtungseingang den Rechtslauf bei 1 definiert).

3.8 Analoge Schnittstelle

Der ATMega88 besitzt zwei analoge Eingangsschnittstellen. Der *Analogkomparator* vergleicht zwei analoge Eingangsspannungen miteinander. Der *Analog-Digitalwandler* (AD-Wandler oder ADC für Analog Digital Converter) misst die Spannung an einem der Analogmultiplexeingänge mit einer Auflösung von acht bis zehn Bit und gibt sie relativ zu einer Vergleichsspannung aus. Die Vergleichsspannung kann von außen anliegen, aus einer internen Referenz (Bandgap) mit 1,1 V erzeugt werden oder einfach die Batteriespannung sein.

3.8.1 Analogmultiplexer

Mehrere Ports der Prozessoren der AVR-Familie lassen sich als analoge Eingänge nutzen, jedoch immer nur einer auf ein Mal. Die Eingänge ADC0 bis ADC5 entsprechen beim ATmegax8 den Pins PC0 bis PC5, je nach Gehäuse sind ADC6 und ADC7 extra herausgeführt oder weggelassen. Der jeweilige Eingang, sofern er gleichzeitig digitaler Eingang ist, muss natürlich als solcher konfiguriert sein (Abschn. 3.5), der Pull-up-Widerstand muss ausgeschaltet sein. Über das Register ADMUX (ADC Multiplexer Select Register) kann eingestellt werden, welcher Eingang genutzt werden soll (Tab. 3.18).

MUX3, MUX2, MUX1 und MUX0 codieren binär den Eingang, wobei MUX3 beim beschriebenen ATmegax8 freigehalten ist. Einzig in der Kombination 1110 und 1111 werden auf den (invertierenden) Eingang des Messverstärkers die interne Referenzspannung von 1,1 V oder Masse gelegt. In Tab. 3.19 sind die entsprechenden Eingänge aufgelistet.

Die Bits REFS1, REFS0 und ADLAR werden in Abschn. 3.8.3 beschrieben.

Tab. 3.18 ADMUX – ADC Multiplexer Select Register

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
REFS1	REFS0	ADLAR	–	MUX3	MUX2	MUX1	MUX0

Tab. 3.19 Bedeutung der MUXn

MUX3..0	Eingang	Pin beim ATmega88 im PDIP Gehäuse	Pin beim ATmega88 im TQFP (32) Gehäuse
0000	ADC0	23	23
0001	ADC1	24	24
0010	ADC2	25	25
0011	ADC3	26	26
0100	ADC4	27	27
0101	ADC5	28	28
0110	ADC6	–	19
0111	ADC7	–	22
1xxx	(reserviert)		
1110	1.1 V (VBG)		
1111	0 V (GND)		

3.8.2 Analogkomparator

Der Analogkomparator besitzt intern einen Differenzverstärker, der die Spannungen an den Eingängen AIN0 und AIN1 bzw. ADC0...7 miteinander vergleicht. Ist die Spannung am positiven Eingang (AIN0) höher als die am negativen Eingang (AIN1 bzw. ADC0...7), ist die Ausgabe ACO (Analog Comparator Output) 1, sonst 0. Der Analogkomparator wird dabei durch das Register ACSR gesteuert (s. Tab. 3.20)

Über das Bit ACBG wird gesteuert, ob statt des Vergleichseingangs AIN0 ($ACBG=0$) die interne Referenzspannungsquelle genutzt werden soll. Das Bit ACD=1 schaltet den Analogkomparator aus, dies sollte immer gesetzt werden, wenn er nicht genutzt wird und der Prozessor Energie sparen soll. ACO liefert den Ausgangswert (Wahrheitswert $AIN0 > AIN1$) und wenn ACIE gesetzt ist (Analog Comparator Interrupt Enable) wird die Interrupt Service Routine des Analogkomparators ISR (ANALOG_COMP_vect) ausgeführt, dabei geht das Bit ACI solange auf 1 bis die Interrupt Service Routine gestartet wurde. Der Modus der Ausführung wird von ACIS1 und ACIS0 gemäß Tab. 3.21 gesteuert.

Tab. 3.20 ACSR – Analog Comparator Status Register

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ACD	ACBG	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0

Tab. 3.21 Interruptausführung des Analogkomparators

ACIS1	ACIS0	Interrupt wird ausgeführt ...
0	0	... wenn sich ACO ändert
0	1	... (reserviert)
1	0	... wenn ACO zu 0 wird (fallende Flanke)
11	1	... wenn ACO zu 1 wird (steigende Flanke)

Schließlich kann mit ACIC noch eingestellt werden, dass der Analogkomparator den Timer 1 triggert. Timer 1 kann also so konfiguriert werden, dass er losläuft oder zählt, wenn sich bei ACO etwas ändert. Mehr dazu im Datenblatt des jeweiligen Prozessors.

3.8.3 AD-Wandler (ADC)

3.8.3.1 Funktionsweise

Der AD-Wandler erlaubt eine einfache Messung von analogen Sensorwerten mit einer Auflösung von 10 Bit, wobei der Hersteller die Genauigkeit auf $+\!-\! 2 \text{ LSB}^9$ angibt, eine 8 Bit Messung schöpft damit die Genauigkeit des Analogeingangs bereits hinreichend aus.

Eine vereinfachte Übersicht über die Funktion des AD-Wandlers findet sich in Abb. 3.15.

Die Messtechnik basiert auf dem Prinzip der sukzessiven Approximation (schrittweisen Näherung), indem zunächst die halbe Referenzspannung $V_c = V_{\text{REF}} / 2$ angelegt und mit der Eingangsspannung V_{in} verglichen wird. Ist diese größer, wird die Vergleichsspannung um ein Viertel der Referenzspannung erhöht, ansonsten um ein Viertel verringert. Im

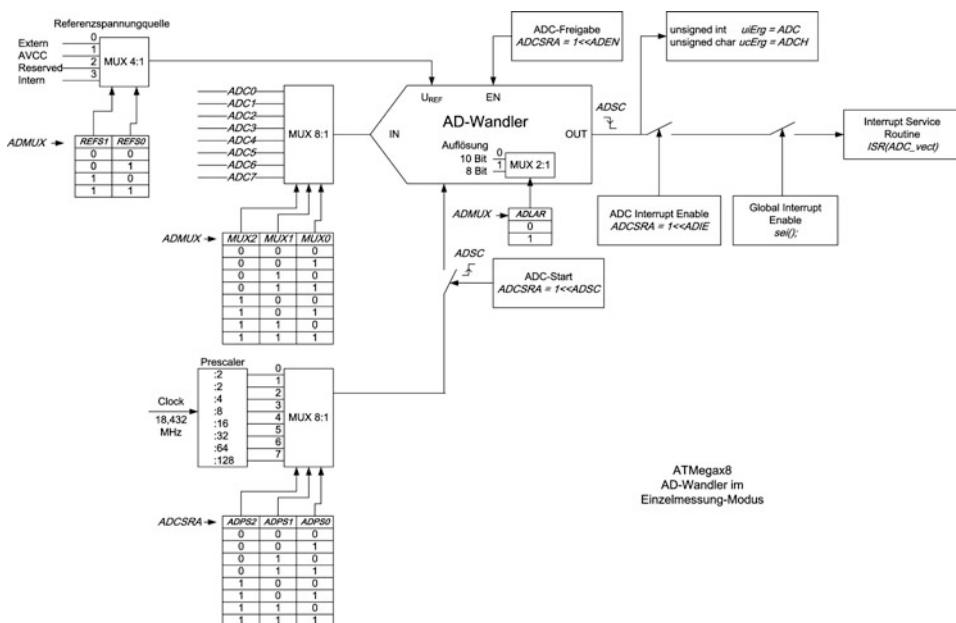


Abb. 3.15 Vereinfachte Funktionsübersicht des AD-Wandlers

⁹ LSB heißt „least significant bit“, also die niederwertigsten Bit.

nächsten Schritt beträgt die Veränderung ein Achtel der Referenzspannung bis schließlich Vergleichsspannung und Eingangsspannung übereinstimmen.¹⁰

$$\text{Wenn } V_{in} < V_c \rightarrow V_c = V_c - V_{REF} / 4$$

$$\text{sonst } \rightarrow V_c = V_c + V_{REF} / 4$$

$$\text{Wenn } V_{in} < V_c \rightarrow V_c = V_c - V_{REF} / 8$$

$$\text{sonst } \rightarrow V_c = V_c + V_{REF} / 8$$

usw.

Dies ist ein schrittweiser Prozess, der eine **Taktung** und damit auch **Zeit** benötigt. Der Takt wird aus einem eigenen Verteiler generiert (s. u.) und sollte laut Hersteller zwischen etwa 50 und 200 kHz liegen. Zur Abschätzung der Messdauer ist die Kenntnis des Timings wichtig. 10 Takte sind für das Approximationsverfahren nötig, drei weitere für die Vor- und Nachbereitung der Messung. Beispielsweise setzen wir bei 18.432 MHz den Prescaler (Verteiler) auf 128 $\rightarrow f_{ADC} = 144 \text{ kHz}$.

Wenn die Messung also 13 Takte $@f_{ADC}$ benötigt, gilt:

$$t_{\text{Sample}} = \frac{13}{144 \cdot 10^3} \text{ s} = 9,027 \cdot 10^{-5} \text{ s} \approx 90 \mu\text{s} \quad (3.10)$$

bzw.

$$f_{\text{Sample}} = \frac{144 \cdot 10^3}{13} \text{ Hz} \approx 11 \text{ kHz} \quad (3.11)$$

Der ADC hat insgesamt eine Auflösung von 10 Bit (1024), so dass der ausgelesene Wert das Verhältnis:

$$\text{ADC} = 1024 \frac{V_{in}}{V_{ref}} \quad (3.12)$$

repräsentiert. Damit Störungen während der Messung keine Rolle spielen, entkoppelt eine *Sample-and-Hold* Schaltung die zu messende Eingangsspannung vom ADC während der Messung.

Die Referenzspannung kann extern angelegt werden und darf dann maximal so hoch sein wie die Versorgungsspannung des ADC, die vom restlichen System getrennt ist. Alternativ wird die über Filter angekoppelte Versorgungsspannung des ADC als Referenzspannung verwendet. Dies muss bei der Initialisierung berücksichtigt werden. Weiterhin kann auch die interne Referenzspannungsquelle mit 1,1 V ausgewählt werden.

Die Messung kann auf verschiedene Weisen durchgeführt werden. Im „Einzelmessung“-Betrieb muss sie durch Setzen eines Bits gestartet werden, im „Freilauf“-Betrieb wird kontinuierlich gemessen. Darüber hinaus existieren weitere Betriebsarten, z. B. Triggierung über den Analogkomparator (Start der Messung bei Über-/Unterschreiten eines bestimmten Wertes), über ein externes digitales Signal oder über einen Timer (z. B. Messung jede Sekunde). Diese Betriebsarten sind im Handbuch des Bausteins näher beschrieben.

¹⁰ Siehe auch Abschn. 7.3.2.

3.8.3.2 Register für die AD-Programmierung

Die Ergebnisse der AD-Wandlung finden sich in den beiden 8 Bit Registern ADCH (High Byte) und ADCL (Low Byte). Dabei sind bei 10 Bit Auflösung nur die beiden untersten Bit in ADCH relevant (right alignment).

Netterweise erlaubt es der C-Compiler, mit der 16 Bit Variablen `ADC` direkt auf beide Register gleichzeitig zuzugreifen:

```
unsigned int uiData;
uiData = ADC;
```

Das Register ADMUX besitzt neben den oben erwähnten MUX2...0 Bits auch noch die Bits für die Wahl der Referenzspannung REFS1 und REFS0. Diese wird gemäß Tab. 3.22 eingestellt:

Eine interessante Variante bietet ADLAR (ADC Left Adjust Result) im selben Register (Tab. 3.18): Ist das Bit auf 1 gesetzt, wird das Ergebnis (10 Bit) im 16 Bit Register links ausgerichtet, d. h. das MSB¹¹ des Ergebnisses ist Bit 15 des Ergebnisregisters. Um den ADC mit nur acht Bit auszulesen, was in der Regel vollkommen ausreicht und keine wirklichen Genauigkeitseinbußen mit sich bringt, genügt es also, bei gesetztem ADLAR das höherwertige Byte des Ergebnisregisters ADCH mit 8 Bit Auflösung auszulesen. Abb. 3.16 verdeutlicht diesen Sachverhalt. Die mit D gekennzeichneten Bits sind die informationshaltigen Datenbits.

Die Register ADCSRA und ADCSRB (ADC Control and Status Register A und B) steuern das Wandlungsverfahren mit den folgenden Bits:

Tab. 3.22 Auswahl der Referenzspannungsquelle

REFS1	REFS0	Referenzspannungsquelle
0	0	AREF, Interne Referenzspannungquelle ist aus
0	1	AVCC mit externem Kondensator am AREF Pin angeschlossen
1	0	Reserviert
1	1	Interne 1,1 V Referenzspannungsquelle

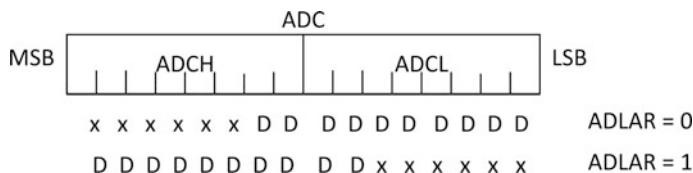


Abb. 3.16 Bedeutung des left alignment im AD Wandler

¹¹ MSB: Most significant bit, also die höherwertigsten Bits.

ADCSRA (siehe Tab. 3.23):

- ADEN (ADC enable, Bit 7): Ein- (1) oder Ausschalten (0) des ADC
- ADSC (ADC start conversion, Bit 6): Start der Messung (1) oder Ende der Messung (0). Das Bit wird im Betriebsmodus „Einzelne Messung“ von der Steuerung auf 0 gesetzt, sobald die Messung abgeschlossen ist.
- ADATE (ADC auto trigger enable select, Bit 5): Hier kann zwischen dem Modus Einzelmessung (0) in den Auto-Trigger-Modus (1) umgeschaltet werden. Im Auto-Trigger-Modus kann dann im Register ADCSRB ausgewählt werden, ob die Messung im Freilauf oder auf ein Triggersignal hin erfolgt.
- ADIF (ADC interrupt flag, Bit 4): Das Bit wird von der Steuerung auf 1 gesetzt, sobald die Messung abgeschlossen ist. Wird ein Interrupt durch den ADC ausgelöst, wird das Bit wieder auf 0 gesetzt. Ohne Interruptsteuerung kann das Bit in einer Warteschleife abgefragt werden und durch Einschreiben einer 1 (tatsächlich! Einer Eins) gelöscht werden, wenn die Messdaten ausgelesen sind.
- ADIE (ADC Interrupt Enable, Bit 3): Wenn dieses Bit gesetzt ist, löst ein Ende der Messung einen Interrupt aus. Dies ist eine empfohlene Methode. Mit einer globalen Variablen `unsigned int uiResult` sieht die Interrupt Service Routine so aus:

```
ISR (ADC_vect)
{
    uiResult = ADC;
}
```

- ADPS2, ADPS1, ADPS0 (ADC Prescaler Select Bits): Legen das Teilverhältnis des Vorteilers und damit die Messgeschwindigkeit fest: 000 ist halber Quarztakt, 010 vier-tel, 011 achtel, 100 sechzehntel, 101 zweiunddreißigstel, 110 vierundsechzigstel und 111 1/128 Quarztakt. Eine Messfrequenz von 50..200 kHz wird vom Hersteller für maximale Genauigkeit empfohlen.

Das Register ADCSRB (Tab. 3.25) besitzt die folgenden Bits:

- ACME (Analog Comparator Multiplex Enable, Bit 6): Schaltet die analogen Multiplexeingänge zwischen ADC (0) und Analog Comparator (1) um. Wir lassen es auf 0.
- ADTS2, ADTS1, ADTS0 (ADC Auto Trigger Source): Diese Bits kodieren die Quelle für die automatische Triggerung der Messung, wenn ADATE in ADCSRA auf 1 gesetzt ist (Tab. 3.24).

Tab. 3.23 ADCSRA – AD control and status register A

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0

Tab. 3.24 Triggerquellen für den Start einer Analogmessung

ADTS2	ADTS1	ADTS0	Triggerquelle
0	0	0	Freilaufbetrieb
0	0	1	Analog Comparator
0	1	0	Externer Interrupt Request 0
0	1	1	Timer/Counter0 Compare MatchA
1	0	0	Timer/Counter0 Überlauf
1	0	1	Timer/Counter1 Compare Match B
1	1	0	Timer/Counter1 Überlauf
1	1	1	Timer/Counter1 Capture Event

Tab. 3.25 ADCSRB – AD control and status register B

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
	ACME				ADTS2	ADTS1	ADTS0

Tab. 3.26 DIDR0 – Digital Input Disable Register 0

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
		ADC5D	ADC4D	ADC3D	ADC2D	ADC1D	ADC0D

Zum Stromsparen kann man während der Messung die korrespondierenden Digitaleingänge noch abschalten, indem man im Register DIDR0 (Digital Input Disable Register) das betreffende der Bits ADC0D ... ADC5D auf 1 setzt.

Eine weitere Stromsparmöglichkeit ist, den ADC vom „free running“ Mode auf Einzelmessung umzuschalten, indem man im Power Reduction Register (PRR) den Wert PRADC auf 0 setzt.

Wenn man eine Leitung ausschließlich als analoge Eingangsleitung benötigt und auf den digitalen Eingang verzichten kann, sollte der digitale Eingang zum Stromsparen abgeschaltet werden. Dies geschieht im Register DIDR0 (Digital Input Disable Register 0, siehe Tab. 3.26). Der korrespondierende Digitaleingang wird durch Setzen einer 1 abgeschaltet (disable).

3.8.3.3 ADC Initialisierung

Um den ADC des µECU zu initialisieren, nehmen wir zunächst das Szenario an, am Eingang ADC4 eine freilaufende Messung zu starten.

Zudem soll die Versorgungsspannung nach Abb. 3.17 als externe Referenzspannung dienen, somit beziehen sich die Werte des AD-Wandlers auf die Versorgungsspannung.

Wir wählen ADC4 als Multiplexer-Input und eine Messfrequenz von 144 kHz, d. h. ein Vorteilverhältnis von 1/128 ($144.000 * 128 = 18.432.000$). Damit ist die Initialisierung:

```
void ADC_Init(void)
{
    ADMUX = (1 << REFS0) | (1 << MUX2);
```

```

    //Vcc Referenz + Kanal 4 ausgewählt.
ADCSRA |= (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);
    //Prescaler :128
DIDR0 |= (1 << ADC4D); //Digitalen Eingang abschalten
ADCSRA |= (1 << ADEN); //AD-Wandler wird freigegeben
ADCSRA |= (1 << ADSC); //AD-Wandler wird zum 1. Mal gestartet
ADCSRA |= (1 << ADIE); //AD-Interrupt wird freigegeben
}

```

Hinweis: In der zweitletzten Zeile wird die erste Messung gestartet. Diese liefert aus verschiedenen Gründen einen falschen Messwert (1024). Sofort nach dem Start der Messung wird der Interrupt „scharf“ geschaltet. Das bedeutet, dass der erste Interrupt einen falschen Messwert liefert. In der Regel ist dies nicht weiter von Bedeutung, falls dennoch, sollte der Interrupt erst eingeschaltet werden, wenn die erste Messung sicher beendet ist.

Ein zweiter Hinweis bezieht sich auf mögliches Rauschen des Systems: Im „Noise Reduction Mode“ wird der Prozessor während des Messvorgangs angehalten um Störungen durch die CPU zu vermeiden. Näheres ist dem Datenblatt [1] zu entnehmen.

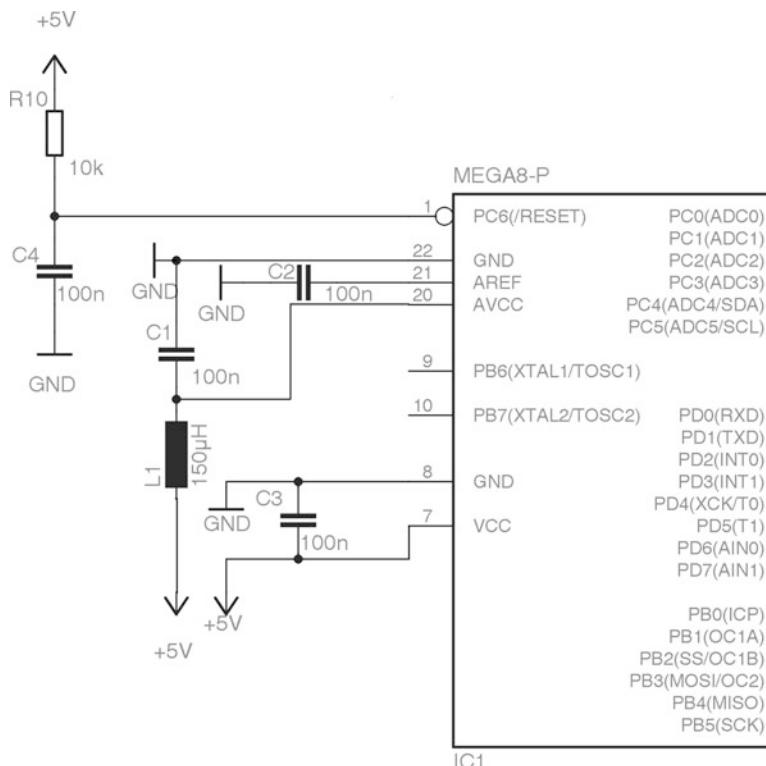


Abb. 3.17 Beispiel für die Verwendung der Versorgungsspannung als externe Referenzspannung

3.8.4 Beispiel: Thermometer

Eine einfache analoge Sensorschaltung ist ein Thermometer. Mit einem Messwiderstand mit negativem Temperaturkoeffizienten (NTC oder Heißleiter) lässt sich die in Abb. 3.18 gezeigte einfache Schaltung aufbauen. Selbst der Quarz wird eingespart, über Fuses, wird der interne Oszillator mit 8 MHz ausgewählt (Abschn. 3.3.2). In der Schaltung wird ausgespart, wie die gemessene Temperatur ausgewertet wird (beispielsweise über ein Display, eine serielle Schnittstelle oder einen Digitalausgang).

In diesem Fall wird der Prescaler auf 64 gesetzt damit die Messfrequenz zwischen 50 und 200 kHz liegt, gemäß Gln. 3.10 und 3.11 beträgt sie dann 125 kHz.

Heißleiter haben eine Kennlinie, die der folgenden Funktion genügt:

$$R_T = R_R \cdot e^{B \cdot \left(\frac{1}{T} - \frac{1}{T_R} \right)} \quad (3.13)$$

Dabei ist:

- R_T der Widerstand bei der Temperatur T
- R_R der Widerstand bei der Bezugstemperatur T_R in Kelvin, beispielsweise $R_{25} = 10 \text{ k}\Omega$ bei $T_R = 25^\circ\text{C} = 298,15 \text{ K}$
- B die Thermistorkonstante in Kelvin, die im Datenblatt angegeben oder experimentell ermittelt wird.

Messtechnisch kann man B ermitteln, indem man den Widerstandswert bei zwei Temperaturen $R_R = R(T_R)$ und $R_T = R(T)$ bestimmt:

$$B = \frac{T \cdot T_R}{T_R - T} \ln \frac{R_T}{R_R} \quad (3.14)$$

In der gezeigten Schaltung wird der NTC in einem Spannungsteiler verwendet, der bei 25°C ein Teilverhältnis von 0,5 aufweist, d. h. die Spannung beträgt 2,5 V. Das obere Diagramm zeigt den Widerstandswert des NTC 10k, das untere Diagramm zeigt die Spannung am Analogeingang jeweils in Abhängigkeit zur Temperatur, so dass der Wert von ADC gemäß Gl. 3.12 ausgegeben wird.

Um nun aus der angelegten Spannung auf die Temperatur zu schließen, kann man folgendes Verfahren anwenden. Man definiert eine Tabelle, die die Temperaturwerte in der gewünschten Genauigkeit enthält, im folgenden Beispiel von 0 bis 40°C in Schritten von 1 K

```
unsigned int uiAnaTemp[40] = {791, 781, 771, 761, 751, 740, 730, 719, 708, 697,
                            686, 674, 663, 652, 640, 628, 617, 605, 593, 582,
                            570, 558, 547, 535, 523, 512, 501, 489, 478, 467,
                            456, 445, 434, 424, 413, 403, 393, 383, 373, 363};
```

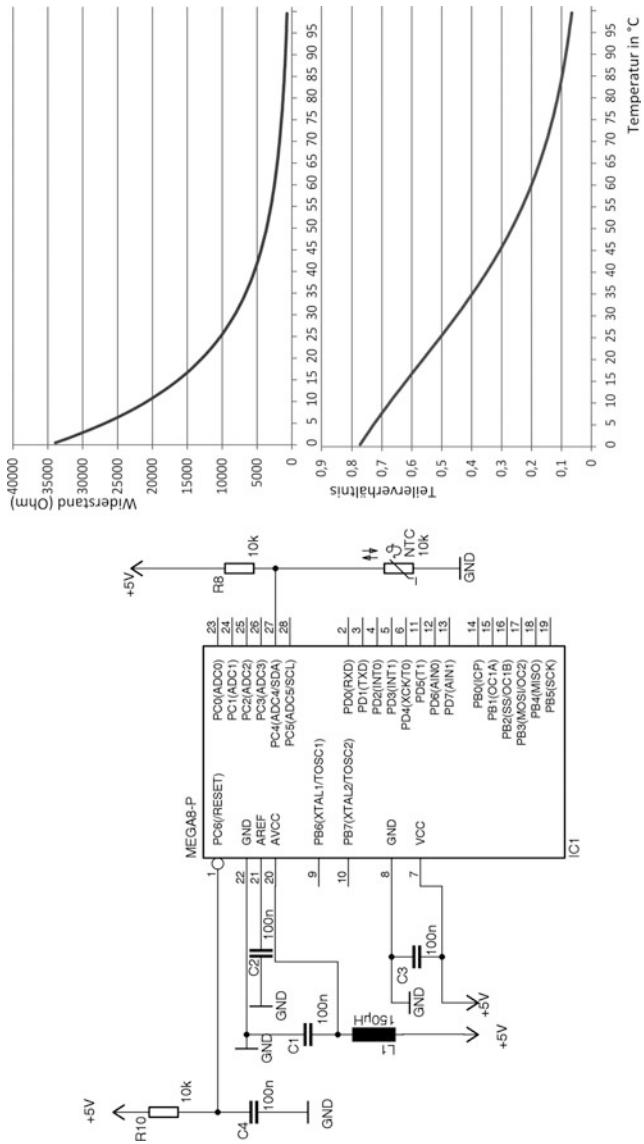


Abb. 3.18 Schaltung einer einfachen Temperaturnmessung mit NTC mit typischen Kennlinien

Der Wert des ADC liegt in der Variablen `uiData` und wurde vorher aus dem Register ADC ausgelesen. In einer Schleife wird nun der Wert sukzessive mit denen in der Tabelle verglichen. Der Tabellenindex an der Stelle, die dem Messwert entspricht beziehungsweise gerade noch kleiner als der Messwert ist, wird als Temperatur ausgegeben.

```
for (j = 1; j < 40; j++)
{
    if ((uiAnaTemp[j-1] >= uiData) && (uiAnaTemp[j] < uiData))
    {
        temp = j - 1;
        break;
    }
}
```

3.9 Power Management

Eingebettete Systeme müssen oftmals mit unzureichenden Energiereserven auskommen. Wie die meisten Mikrocontroller besitzt auch die AVR-Familie verschiedene Mechanismen um den Energieverbrauch an die Rechnerfordernisse anzupassen. Insbesondere bei batteriebetriebenen Schaltungen, die nicht explizit ausgeschaltet werden können oder sollen, macht es Sinn, den Prozessor und die Peripheriebausteine gezielt „schlafen zu legen“ und nur bei Bedarf wieder aufzuwecken. Im Kraftfahrzeug schalten viele Steuergeräte sogar die Stromversorgung selbst in einen Notbetrieb, so dass sich auch die Netzteilverluste reduzieren lassen. Der Hauptgrund für Stromverbrauch in Mikrocontrollern ist das Umladen der Gates. Abschalten oder Reduzieren von Takt ist also die gängige Stromsparmethode. Dies kann pro Komponente einzeln geschehen (s. u.) oder global, je nach Powermanagementkonzept.

Die AVR Familie hat ein mehrstufiges Powermanagementkonzept und kennt die fünf Sleep-Modi:

Idle: Hier werden der CPU- und der Flashtakt abgeschaltet, die interne Peripherie bleibt versorgt, so dass Botschaften über die USART- die SPI- oder die TWI-Schnittstelle (siehe Kap. 5), ein Pinchange (PCINT) oder externer Interrupt (INT0 und INT1) und weitere Quellen, z. B. Timer, Watchdog und weitere Resets und Interrupts die CPU wieder aufwecken können.

ADC noise reduction: Hier wird zusätzlich zum Idle-Mode noch der IO-Takt abgeschaltet. Dieser Modus dient dazu, Störeinflüsse beim Messen mit dem AD-Wandler zu reduzieren. Sobald der AD-Wandler die Messung abgeschlossen hat, weckt er die restlichen Komponenten wieder auf, ebenso wie die meisten Weckquellen des Idle-Mode.

Power down:	In diesem Modus sind alle internen Taktquellen abgeschaltet. Geweckt wird das System durch Pinchange Interrupt, Pegel-Interrupt an den Pins INT0 und INT1, einer empfangenen TWI-Botschaft oder ein Reset.
Power safe:	Dieser Modus ist nahezu identisch mit Power down, mit der Ausnahme, dass hier auch der Timer 2 einen Weckgrund erzeugt. Man kann diesen Modus beispielsweise nutzen, das System in den Ruhezustand zu versetzen und periodisch zu wecken um beispielsweise eine Uhr weiter zu betreiben. Man kann Timer 2 asynchron mit einem 32.768 kHz Quarz betreiben und den restlichen Systemtakt aus dem internen 8 MHz Oszillator generieren. Dieser Modus eignet sich für batteriebetriebene Geräte mit Uhren.
Standby:	Ebenfalls nahezu identisch mit Power down, mit der Ausnahme, dass der externe Oszillator weiterläuft. Das kostet zwar Strom, hat jedoch den Vorteil, dass die CPU nach dem Aufwachen nach nur sechs Takten wieder zur Verfügung steht. Quarzoszillatoren haben auf Grund ihrer sehr hohen Güte (sehr schmalbandiger Schwingkreis) eine lange Einschwingzeit von 10...100 ms. Der Standby-Modus ist ideal, wenn ein Quarz verwendet wird und kürzeste Reaktionszeiten nach dem Aufwecken nötig sind. Da alle Takte gestoppt sind, funktionieren nur noch die asynchronen Module.

Der Stromverbrauch im Power down Modus sinkt dramatisch. Prozessoren vom Typ ATmega88V verbrauchen im Normalbetrieb etwa 250 µA bei 1,8 V Betriebsspannung und 1 MHz Systemtakt, während sie im Power down Modus nur 0,1 µA benötigen. Bei 8 MHz und 5 V Betriebsspannung steigt der Stromverbrauch auf ca. 12 mA während er im Power down Mode immer noch im kleinen einstelligen Mikroamperebereich bleibt.

Die Sleep-Modi können durch Schreiben in das SMCR (Sleep mode control register) selektiert und dann mit dem Assemblerbefehl SLEEP gestartet werden. Es empfiehlt sich jedoch, die vom Hersteller zur Verfügung gestellten Makros aus avr/sleep.h zu verwenden. Ein Programm zum Einschlafen und Wecken durch einen Interrupt sieht wie folgt aus:

```
if (key_get(PIND4))  
{  
    set_sleep_mode(SLEEP_MODE_PWR_DOWN);  
    sleep_mode();  
}
```

Sobald PIND4 auf null geht, schaltet sich der Mikrocontroller in den Sleep Mode. Selbstverständlich muss ein externen Interrupt eingerichtet sein um ihn wieder zu wecken

Tab. 3.27 Power reduction register (PRR)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PRTWI	PRTIM2	PRTIM0	–	PRTIM1	PRSPI	PRUSART0	PRADC

(s. Abschn. 3.6.3). Wichtig ist, dass im Power Down Modus nur der Levelinterrupt zur Verfügung steht, also nur die Konfiguration:

```
EICRA &= ~(1 << ISC00) & ~(1 << SC01);
```

Weitere Energieeinsparungen können dadurch erreicht werden, dass man gezielt die Peripherieeinheiten abschaltet, die nicht benötigt werden. Dafür steht das Power reduction register zur Verfügung (Tab. 3.27).

Durch Setzen einer 1 in eines der Felder werden die Two-Wire-Interface (TWI) Schnittstelle (Bit 7), der Timer 2 (Bit 6), der Timer 0 (Bit 5), der Timer 1 (Bit 3), die SPI-Schnittstelle (Bit 2), der USART (Bit 1) und der ADC (Bit 0) ausgeschaltet. Auch hier unterstützt die Libc in avr/power.h. Beispielsweise schalten `power_adc_disable()` und `power_adc_enable()` den AD-Wandler aus und ein. Gleiches gilt für `power_twi_enable()` und `power_twi_disable()` und andere. Hier empfiehlt sich die Lektüre von [3].

3.10 Internes EEPROM

Das interne EEPROM dient dem Zweck, Daten persistent abzulegen. Diese überstehen dann ein Reset oder eine Unterbrechung der Stromversorgung. EEPROMs benötigen einige Zeit bis die Daten gespeichert sind, daher eignen sie sich nicht als Arbeitsspeicher. Typische Anwendungsfälle sind Parametrierungen und Anpassungen, die nicht im Quellcode stehen sollen, und Fehlerspeichereinträge. Da zudem die Zahl der EEPROM-Schreibzugriffe begrenzt ist, sollte von der Verwendung sparsam Gebrauch gemacht werden. Über die Programmierschnittstelle lässt sich das interne EEPROM beschreiben und auslesen, im ersten Fall lässt sich damit ein Baustein parametrieren (z. B. mit einer individuellen Seriennummer oder mit einem Wert, der eine SW-Variante festlegt). Im zweiten Fall kann das EEPROM als Fehlerspeicher genutzt werden oder um Variablenwerte über Neustarts zu „retten“. Als Datenlogger eignet sich das EEPROM nicht besonders gut, hier empfiehlt sich der Einsatz eines externen Flash-Speichers.

Um auf das interne EEPROM zugreifen zu können, empfiehlt sich die Nutzung der Makros aus dem Include-File `avr/eeprom.h`.

3.10.1 Deklaration einer Variablen im EEPROM

Mit dem Makro `EEMEM` wird der Compiler angewiesen, eine Variable ins EEPROM zu verlagern, beispielsweise

```
unsigned char ucEPByte EEMEM; //Ein Byte
unsigned int uiEPWord EEMEM; //Ein Word - also ein 16 Bit Integer
unsigned char pucEPByte[10] EEMEM; //Zeiger auf ein Feld
```

Die Adresse der Variablen wird dann für die folgenden Operationen herangezogen. Es sei erwähnt, dass diese Definition auch für `float` und für `dword` (entspricht `long`) gilt.

3.10.2 Lesen aus dem EEPROM

Das Lesen aus dem EEPROM stellt eine einfache Möglichkeit dar, externe Parametrierungen vorzunehmen, indem das EEPROM über die Programmierschnittstelle programmiert wird und im Code nach den Einstellungen entsprechende Parameter vergeben werden.

Das Lesen erfolgt durch `eeprom_read_byte()`:

```
unsigned char ucWert;
unsigned char pucFeld[10];

ucWert = eeprom_read_byte(&ucEPByte);
eeprom_read_block(pucFeld, pucEPByte, 10);
```

Entsprechendes gilt auch für `float`, `word` und `dword`. Man beachte die Reihenfolge beim Blocktransfer. Der erste Parameter ist das Ziel, also der Zeiger auf ein Feld im Arbeitsspeicher, der zweite Parameter ist die Adresse im EEPROM, die mit dem `EEMEM`-Makro generiert wurde, der dritte Parameter ist die Länge.

3.10.3 Schreiben ins EEPROM

Das Schreiben ins EEPROM erfolgt durch

```
unsigned char ucWert = 0x1A;
unsigned int uiWert = 0x2345;
unsigned char pucFeld[] = "0123456789";

cli();
eeprom_write_byte(&ucEPByte, ucWert);
eeprom_write_word(&uiEPWord, uiWert);
eeprom_write_block(pucFeld, pucEPByte, 10);
sei();
```

Entsprechendes gilt für float und dword. Beim Blocktransfer wird zunächst die Adresse des Feldes im Arbeitsspeicher übergeben, danach die Adresse im EEPROM und schließlich die Länge des Feldes.

Anstelle von `eeprom_write_xxx()` kann auch `eeprom_update_xxx()` genutzt werden. Diese Funktionen prüfen zuvor, ob sich der Wert der Variablen gegenüber dem EEPROM geändert hat und schreiben nur die Änderungen ins EEPROM. Dadurch kann man unter Umständen erheblich Schreibzyklen sparen und damit die Haltbarkeit des EEPROM erhöhen.

Interrupts dürfen während des Schreibens nicht auftreten, da das Timing unbedingt eingehalten werden muss. Die Schreibzeiten sind verhältnismäßig lang (3,4 ms pro Byte für Löschen und neu Setzen) und der Mikrocontroller blockiert während dieser Makros. Sie sind daher für schnelle persistente Speicheraufgaben nicht zu empfehlen. In Kap. 7 werden besser geeignete, vernetzbare Speicherbausteine für Logging-Aufgaben vorgestellt. Diese werden in der Regel über SPI oder TWI Schnittstelle beschrieben und puffern ihre Daten selbst, was zu erheblichen Zeitvorteilen führt.

Um sicher zu sein, dass das EEPROM lese- und schreibbereit ist, kann man das EEPROM Control Register (`EECR`) abfragen. Auch dies erledigt ein Makro aus dem `eeprom.h` File namens `eeprom_is_ready()`. Man kann sich den Abschluss eines Schreib- oder Lesevorgangs auch durch einen Interrupt signalisieren lassen, wenn größere Blocktransfers anstehen. Damit wird die Blockade des Systems zumindest abgemildert.

Im folgenden Beispiel wird die Anzahl der Resets des Mikrocontrollers abgelegt, indem man im Initialisierungssteil folgende Zeilen schreibt und das EEPROM an der entsprechenden Stelle vor dem ersten Aufruf durch das Programmiergerät löscht.

```
if (eeprom_is_ready())
    ucWert = eeprom_read_byte(&ucEPByte);
    eeprom_write_byte(&ucEPByte, ucWert + 1);
```

3.11 Dynamische Speichernutzung

Die Standardlib (<avr/stlib.h>) erlaubt auch für die AVR-Familie die dynamische Nutzung des Speichers. Obwohl man wegen der limitierten Größe des internen RAMs extrem vorsichtig damit umgehen sollte, soll die Funktionalität hier kurz beschrieben werden. Die Verhältnisse sind in Abb. 3.19 dargestellt.

Der AVR-C-Compiler kennt vier verschiedene Arbeitsspeicher [2, 3]:

- Den Bereich der initialisierten Daten, d. h. Variablen, die im Code vorbelegt wurden, mit der Bezeichnung `.data`

```
char err_str[] = "Fataler Fehler im Code aufgetreten";
```

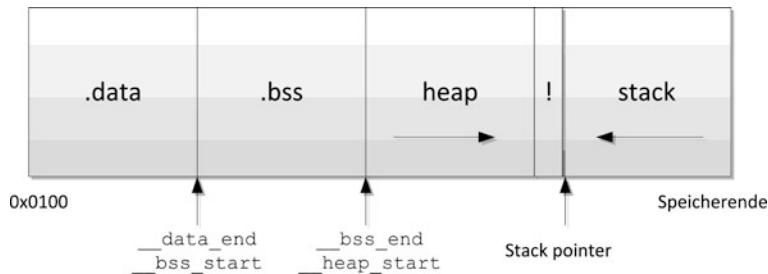


Abb. 3.19 Aufbau des Speichers [2, 3]

- Den Bereich der nicht initialisierten Daten, das sind globale oder als statisch (`static`) gekennzeichnete Variablen. Der Bereich heißt .bss
- Den Stack (oder Stapel, bzw. Kellerspeicher)
- Den Heap („Haufen“)

Der Stack beginnt am Ende des Speicherbereichs und wird für den Aufruf von Funktionen und Interrupt Service Routinen benutzt. Sobald ein Aufruf erfolgt, werden lokale Variablen, Registerinhalte und der Programmzähler auf den Stack geschoben und nach Abarbeitung wieder „gerettet“. Dies macht der Prozessor bei einem Funktionsaufruf (`rcall`) bzw. einer ISR automatisch. Mit der Aufruftiefe (also der Zahl der Funktionen, die aus Funktionen aufgerufen wurden oder mit der Zahl der verschachtelten Interrupts) wächst der Stack nach vorne hin an. Insbesondere wenn man Funktionen rekursiv aufruft oder viele lokale Variablen nutzt, kann der Stack schnell groß werden.

Der Heap beinhaltet den so genannten *dynamischen* Speicher. Er beginnt direkt nach der .bss Sektion und wächst mit dem Bedarf bis zu einer einstellbaren Grenze zwischen Stack und Heap.

Um den Heap zu nutzen benötigt man zwei Funktionen: `malloc()` und `free()`, die beide in der Standardlib verfügbar sind.

Ein Beispiel:

```
struct sListItem {char i; struct sListItem *next;};
typedef struct sListItem tListItem;
```

erzeugt einen Datentyp `tListItem`. Dieser besteht aus drei Bytes, nämlich dem character `i` und einem Zeiger auf eine Struktur `next`. Mit dem C-Schlüsselwort `sizeof` kann die Größe des Datentyps ermittelt werden (Abschn. 2.9).

Um einen dynamischen Speicher im Heap zu allozieren¹² (anzulegen), wird die Funktion `malloc(size)` verwendet.

¹² Allokieren meint wörtlich: „einen Ort zuweisen“.

```
tListItem *first;
...
first = (tListItem*) malloc (sizeof(tListItem));
```

Der Aufruf von `malloc(size)` liefert eine gültige Adresse im Heap oder einen null-Pointer, wenn kein Heap mehr zur Verfügung steht. Damit wächst der Heap wie ein Haufen langsam an, bis er mit dem Stack „kollidiert“, was im Falle eines ATmega88 recht schnell vonstattengeht. Neben dem Heap legt `malloc(size)` auch eine so genannte „freelist“ an, in der die Adresse und die Größe des allokierten Speichers abgelegt werden. Mit dem Aufruf `free(adress)` wird der Speicher wieder freigegeben, sobald er nicht mehr benötigt wird.

```
free(first);
```

Vergisst man das, kann ein Speicherüberlauf die Folge sein.

Ein Nutzungsbeispiel für die dynamische Speicherverwaltung findet sich in Abschn. 4.4.

3.12 Verlagerung von Daten in den Programmspeicher

Größere Tabellen belegen gleich sehr viel Platz im Arbeitsspeicher. Insbesondere wenn diese Tabellen konstant sind (z. B. für die Umrechnung einer Sensor Kennlinie wie in Abschn. 3.8.4), kann der Compiler durch das Attribut PROGMEM aufgefordert werden, diese Daten in den Programmspeicher zu schreiben. Im Fall des Thermometers sieht dies so aus:

```
#include <avr/pgmspace.h>
(...)
const unsigned int uiAnaTemp[40] PROGMEM = {791, 781, 771, 761, 751, 740,
                                             730, 719, 708, 697, 686, 674, 663, 652, 640, 628,
                                             617, 605, 593, 582, 570, 558, 547, 535, 523, 512,
                                             501, 489, 478, 467, 456, 445, 434, 424, 413, 403,
                                             393, 383, 373, 363};
```

Entgegen den Empfehlungen aus [3] und [2] fordert der Gnu C Compiler 4.8.1 dazu auf, das Feld als `const` zu deklarieren.

Mit der Verlagerung in den Flash lässt sich auf das Feld allerdings nicht mehr direkt zugreifen. Der Zugriff geschieht nun, ähnlich wie beim EEPROM, über ein Macro aus pgmspace.h. Der entsprechende Zugriff auf `uiAnaTemp` aus Abschn. 3.8.4 sieht dann so aus:

```
if ((pgm_read_word(&(uiAnaTemp[j-1])) >= uiData)
    && (pgm_read_word(&(uiAnaTemp[j])) < uiData))
(...)
```

An `pgm_read_word` (analog `pgm_read_byte`, `pgm_read_dword`, `pgm_read_float`) wird der Zeiger auf den zu lesenden Inhalt übergeben.

```
pgm_read_word(&(uiAnaTemp[j]));
```

Einige Fallstricke dieser Verlagerung werden in den zitierten Referenzen beschrieben. In vielen Fällen ist sie jedoch erheblich gerechtfertigt und schont den Speicherplatz im Arbeitsspeicher.

Literatur

1. Microchip Technology Inc.: Reference manual ATmega48/88/168, document 2545 (2016). <http://www.microchip.com/wwwproducts/en/atmega88pa>, Zugegriffen: 14. April 2018
2. Microchip Technology Inc.: AVR Libc (2016). https://www.microchip.com/webdoc/AVRLibcReferenceManual/overview_overview_gcc.html, Zugegriffen: 14. April 2018
3. NONGNU: AVR Libc (2016). <http://www.nongnu.org/avr-libc/user-manual/index.html>, Zugegriffen: 14. April 2018
4. Schmitt, G.: Mikrocomputertechnik mit Controllern der Atmel AVR-RISC-Familie, 4. Aufl. Oldenbourg, München (2008)
5. Meroth, A., Tölg, B.: Infotainmentsysteme im Kraftfahrzeug. Grundlagen, Komponenten, Systeme und Anwendungen. Vieweg, Wiesbaden (2008)

Weiterführende Literatur

6. Bernstein, H.: Mikrocontroller: Grundlagen der Hard- und Software der Mikrocontroller ATTiny2313, ATTiny26 und ATmega32 – 1. Springer Vieweg, Wiesbaden (2015)
7. Gaicher, H., Gaicher, P.: AVR Mikrocontroller – Programmierung in C: Eigene Projekte selbst entwickeln und verstehen Bd. 1. tredition, Hamburg (2016)
8. Salzburger, L., Meister, I.: AVR-Mikrocontroller-Kochbuch, 1. Aufl. Franzis, Haar/München (2013)
9. Spanner, G.: AVR-Mikrocontroller in C programmieren: Über 30 Selbstbauprojekte mit ATTiny13, ATmega8, ATmega32 (PC & Elektronik), 1. Aufl. Franzis, Haar/München (2010)
10. Williams, E.: Make: AVR programming: learning to write software for hardware, 1. Aufl. O'Reilly & Associates, Newton, MA (2014)
11. Schäffer, F.: AVR: Hardware und Programmierung in C. Elektor, Aachen (2014)
12. Mikrocontroller.net: (2018). <http://www.mikrocontroller.net>, Zugegriffen: 14. April 2018



Zusammenfassung

Nach dem Studium von Kap. 3 sollten Sie in der Lage sein, wichtige Funktionen des AVR Mikrocontrollers zu verstehen und zu bedienen. In Kap. 4 geht es nun darum, den Code zu organisieren und mit dem Ziel der Lesbarkeit und Wiederverwendbarkeit zu modularisieren. Außerdem werden zwei wichtige Mechanismen eingeführt, die in der Welt der „embedded Systems“ immer wieder benötigt werden, namentlich die Warteschlange (FIFO) und der Zustandsautomat. Daneben enthält das Kapitel einige Betrachtungen zur Zeitsteuerung der Software, mithin ein einfaches Multitasking-Schema.

4.1 Überblick

4.1.1 Motivation

Nachdem in den ersten Kapiteln die grundsätzliche Programmierung der wichtigsten Funktionen eines Mikrocontrollers beschrieben wurde, wird es nun Zeit, ein wenig Ordnung in die Programmstruktur zu bringen. Dazu benötigt man eine zumindest rudimentäre Softwarearchitektur. Architektur ist hier wie in der IEEE 1471 gemeint als:

Architecture is the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution [1].

Ein Architekturentwurf verfolgt stets bestimmte Ziele [2]. Im vorliegenden Fall sollen zunächst die folgenden Ziele erreicht werden:

- Darstellung in sich geschlossener Module
- Gute Wiederverwendbarkeit der Softwaremodule
- Testbarkeit der Module
- Gute Lesbarkeit und Nachvollziehbarkeit
- Kombinierbarkeit der verschiedenen Lösungen
- Hardwarekapselung beziehungsweise -abstraktion
- Unabhängigkeit von der Betriebssystemumgebung

Codeeffizienz, niedriger Energieverbrauch und optimale Ressourcennutzung kommen natürlich dazu, werden in diesem Buch aber nicht zu sehr vertieft. Microchip hat hierzu eine eigene Applikationsschrift [3] herausgegeben.

4.1.2 Sichten

Eine Softwarearchitektur nutzt gewöhnlich verschiedene Sichten auf das System. Diese dienen dazu, das Verhältnis der Komponenten untereinander und mit der Außenwelt statisch und dynamisch zu beschreiben. Einige wichtige Sichten sind:

- Die statische Komponentensicht (Abb. 4.1), beschreibt das relative Verhältnis der Funktionen/Komponenten untereinander: Wer greift auf wen zu?
- Die statische Verteilungssicht (Deployment): Wie sind die Funktionen/Komponenten auf physischen Geräten in einem verteilten System aufgeteilt?
- Die Kontextsicht: Wie sieht die Schnittstelle nach außen aus?
- Dynamische Sichten: Hier wird das dynamische Zusammenspiel von Komponenten (Sequenzdiagramm oder sequence chart) beziehungsweise die Abläufe innerhalb einer Komponente (Beispiel: Zustandsautomat) in zeitlicher und logischer Abhängigkeit beschrieben.

Einige dieser Sichten werden im Lauf des Kapitels näher vorgestellt.

In Abb. 4.1 ist eine einfache Softwarearchitektur für ein Messsystem dargestellt. Die Sensoren sind entweder an einer analogen Schnittstelle (Temperatursensoren, Helligkeits-sensoren, Näherungssensoren, Beschleunigungssensoren mit Analogausgang), an deiner digitalen Schnittstelle (Präsenz- und Annäherungssensoren, End- oder Schwellwertschalter) oder an einer der seriellen Schnittstellen angeschlossen, die noch im Kap. 5 beschrieben werden. Die im folgenden Abschnitt beschriebene Hardwareabstraktionsschicht dient dazu, die Hardware von der Applikation zu entkoppeln. Die Applikation übernimmt schließlich die Verarbeitung der Daten, wobei die Vorverarbeitung (zum Beispiel Filtrierung) in eine eigene Sensorabstraktionsschicht ausgelagert werden kann. Das Ergebnis der Verarbeitung steht dann entweder an einem Bussystem als Nachricht zur Verfügung (Kap. 5), wird auf einem Display angezeigt (Kap. 8) oder für zu einem Eingriff mit einer Aktorik (Lampen, Motoren, Ventile, thermische Elemente), wie in Kap. 3 angedeutet, letzteres in der Regel über eine PWM-Ansteuerung oder durch einen digitalen Ausgang.

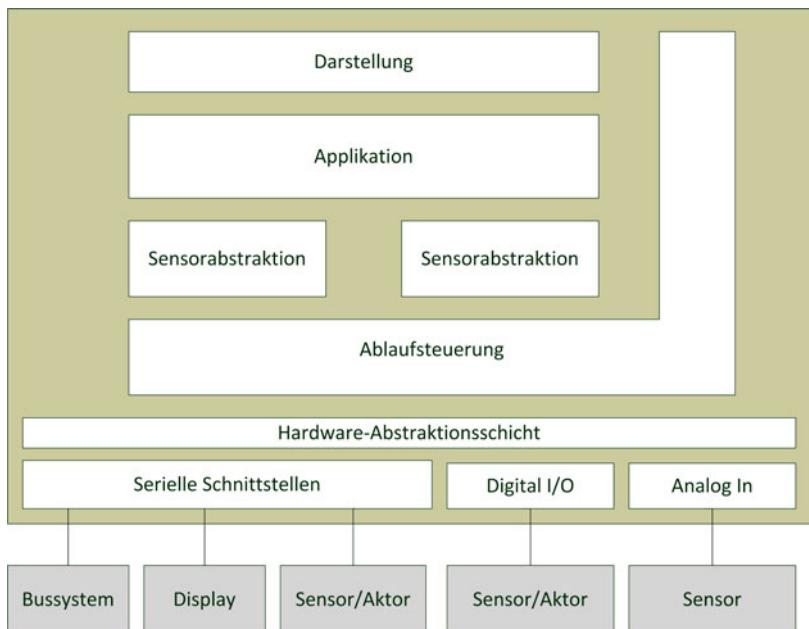


Abb. 4.1 Einfache Softwarearchitektur (statische Komponentensicht) für ein Messsystem

4.2 Hardware-Abstraktion

Hardware-Abstraktion stellt eine wichtige Maßnahme zur Verbesserung der Lesbarkeit und Portierbarkeit von Software dar und erleichtert die Wiederverwendung in unterschiedlichen Projekten. In diesem Buch bezieht sich Hardware-Abstraktion auf folgende Bereiche (Abb. 4.1):

- **Abstraktion von der Mikrocontroller-Familie:** Hier werden gängige Konzepte, beispielsweise Timer, analoge und digitale Ein- und Ausgänge, EEPROM und andere (siehe Kap. 3) durch eigene Funktionen und Module dargestellt (man spricht auch von „wrapping“, weil die Funktionalität in eigene Funktionen eingepackt wird). Der Zugriff erfolgt dann nur noch über diese Wrapper-Funktionen. Beispielsweise kann man für einen digitalen Ein-/Ausgang die folgenden Funktionen in einem eigenen Modul DIO bereithalten, das von Prozessortyp zu Prozessortyp ausgetauscht wird:
 - `DIO_SetDirection(address)`
 - `DIO_Write(address,value)`
 - `DIO_Read(address)`
- **Abstraktion vom Board:** Diese bezieht sich darauf, wie der Mikrocontroller auf der Platine beschaltet ist, hier können sich beispielsweise PORT Nummern, Analogeingänge, PWM-Ausgänge oder Schnittstellennummern ändern. Abhilfe schafft es, de-

ren Adressen in gängige und lesbare abstrakte Begriffe zu verpacken, in einem File `board_abstraction.h` könnten diese etwa so aussehen:

```
- #define LED1OUT PORTB
- #define LED1BIT (1<<PB4)
```

Damit lautet die Ansteuerung von LED 1 dann:

```
LED1OUT |= LED1BIT ;
```

und die Abschaltung von LED 1:

```
LED1OUT &= ~LED1BIT;
```

- **Abstraktion vom verwendeten Kommunikationsmechanismus:** Die unterschiedlichen Kommunikationsschnittstellen können über eine einheitliche Kommunikationsabstraktionsschicht aufgerufen werden. Nach der Initialisierung werden alle seriellen Schnittstellen einheitlich über `read()` und `write()` Funktionen angesprochen. Diese übergeben nur noch einen Zeiger auf eine Datenstruktur, in der die notwendigen Adressinformationen und Daten stehen, an einen Ringpuffer (Abschn. 4.5.1), aus dem dann im Hintergrund die Daten an die betreffende Schnittstelle übergeben werden.
- **Abstraktion von den Sensoren:** In der Sensorabstraktionsschicht stehen Datenstrukturen und Funktionen zur Verfügung, die sich wie ein generischer Sensor eines bestimmten Typs verhalten. Diese werden von der eigentlichen funktionalen Programmebene angesprochen. Letztere muss dann nicht mehr angefasst werden, wenn ein neuer Sensor eingeführt wird. Lediglich an der Abstraktionsschicht sind Anpassungen durchzuführen.

Selbstverständlich bedeutet die Einführung solcher Abstraktionsschichten immer einen Verlust an Performance und Speicherplatz. Gleichzeitig steigt möglicherweise der Energiebedarf des Prozessors an. Es müssen hier individuelle Kompromisse gefunden werden.

4.3 Modularisierung und Zugriff auf Module

Bei der Modularisierung müssen ebenfalls Kompromisse zwischen Codeeffizienz (Flash), Speichereffizienz und Lesbarkeit beziehungsweise Wiederverwendbarkeit geschlossen werden. Anfängern und Teams, die gemeinsam an einer Software arbeiten, wird empfohlen, zunächst die Lesbarkeit und Wiederverwendbarkeit im Auge zu behalten. Diese beinhalten, dass Funktionen, die zusammengehören, in ein Modul zusammengefasst werden. Das Modul besteht aus einer .c Datei mit den Quellen und einer .h Datei, in der gemeinsam genutzte symbolische Platzhalter (`#define`) und die Modulschnittstellen in Form von Zugriffsfunktionen deklariert sind. An dieser Stelle sei auf die Verwendung von über das Modul hinaus globalen Variablen (`extern`) zunächst abgeraten (siehe auch Abschn. 2.8.5). Diese stellen zwar eine effiziente Möglichkeit des Zugriffs in das Modul dar, haben aber auch den Nachteil, dass ihre Verwendung nur schwer zu kontrollieren ist. Eine einfache, im Sinne des Programmspeichers nicht zu teure Alternative sind Zugriffsfunktionen („setter“ und „getter“ Funktionen) auf die globalen Variablen im Modul.

Später, in Abschn. 4.6 werden wir einen Zustandsautomaten verwenden. Dieser nutzt eine im Modul statemachine.c deklarierte modulglobale Variable `ucState`. Von anderen Modulen sollte diese Variable abgefragt und gesetzt werden können. Dazu vereinbaren wir die folgenden setter und getter Funktionen, deren Aufgabe unschwer zu erkennen ist:

```
unsigned char ucState ;

void set_ucState(unsigned char state)
{
    ucState = state ;
}

unsigned char get_ucState(void)
{
    return ucState;
}
```

Somit muss die Variable nicht mehr übermodulglobal durch `extern` veröffentlicht werden. Zusätzlich bietet diese Vorgehensweise den Vorteil, dass beim Zugriff gleich eine Wertebereichsüberprüfung stattfinden kann oder der Zugriff durch Interrupts unterbunden werden kann, wie das folgende Beispiel zeigt, in dem eine Variable `cPercentage` geschrieben werden soll, sofern sie im Wertebereich bleibt:

```
#define ERROR_OUT_OF_BOUNDS    0
#define WRITE_SUCCESSFUL        1

char cPercentage;

char set_cPercentage(char value)
{
    char result;
    cli(); //alle Interrupts sperren
    if ((value >= 0) && (value <100))
    {
        cPercentage = value;
        result = WRITE_SUCCESSFUL;
    }
    else result = ERROR_OUT_OF_BOUNDS;
    sei(); //alle Interrupts wieder freigeben
    return result;
}
```

Weiterhin können Module auf die beschriebene Weise besser gegen versehentliches Überschreiben globaler Variablen geschützt werden, weil über das .h-File nur die Variablen bekannt werden, die auch öffentlich genutzt werden können. Ein weiteres Beispiel für den Zugriff auf modulglobale Variablen zeigt der folgende Abschn. 4.4.

4.4 Zeitsteuerung

Für einen koordinierten Ablauf der Software ist eine Zeitsteuerung meist unerlässlich. Wie in Abschn. 3.7.2 bereits beschrieben, nutzen wir einen Timer für die globale Softwareablaufsteuerung. Dieser erzeugt einen Interrupt, der in der kürzesten im System vorkommenden Zeit gefeuert wird. In der Interrupt Service Routine werden daraus dann alle wichtigen Systemzeiten abgeleitet. Mit einer geeigneten Abstraktion lässt sich daraus ein kleines Betriebssystem bauen, in dem verschiedene Aufgaben (Tasks) zeitgesteuert regelmäßig abgearbeitet werden. Im folgenden Beispiel werden im System die Zeiten 10ms, 50ms und 100ms benötigt, außerdem soll eine „Stoppuhrfunktion“ mitteilen, wie viel Zeit seit dem letzten Aufruf der Funktion in 100ms verstrichen ist. In einem Modul Timer.c werden folgende globalen Variablen deklariert:

```
unsigned char ucTimer1_Flag_10ms = 0; //wird 1, wenn 10 ms
                                         //verstrichen sind

unsigned char ucTimer1_Flag_50ms = 0; //wird 1, wenn 50 ms
                                         //verstrichen sind

unsigned char ucTimer1_Cnt_50ms = 0; //zählt die 1 bis 50 ms
                                         //verstrichen ist

unsigned char ucTimer1_Flag_100ms = 0; //wird 1, wenn 100 ms
                                         //verstrichen sind

unsigned char ucTimer1_Cnt_100ms = 0; //zählt die 1 bis 100 ms
                                         //verstrichen ist

unsigned long ulSystemClock = 0;      //zählt global die 100 ms
```

Im vorliegenden Beispiel wird Timer 1 verwendet, genauso gut kann ein anderer Timer verwendet werden.

Timer 1 wird also wie in Abschn. 3.7 beschrieben so initialisiert, dass er alle 10ms einen Interrupt auslöst, was bei einem 16-Bit Timer @8 MHz mit hoher Genauigkeit möglich ist, wenn man die Quarzfrequenz durch 64 teilt und von 0...1249 zählt:

```
void Timer1_Init(void)
{
    TCCR1B |= (1 << WGM12) ; //CTC Mode
    TCCR1B |= (1 << CS11) | (1 << CS10); //Vorteiler 64
    OCR1A = 1249; //Vergleichswert tritt alle 10 ms ein
    TIMSK1 |= (1 << OCIE1A); //Interrupt für Compare Register freigeben
    sei(); //alle Interrupts freigeben
}
```

Bei 18.432 MHz Quarzen würde man dasselbe Resultat erreichen, wenn man durch 1024 teilt und von 0..179 zählt. Dies ist dann auch mit Timer 0 oder Timer 2 möglich.

In der ISR des Timers werden nun die oben genannten Zeitvariablen hochgezählt. Sobald sie ihren Zielwert (das ist 5 bei der 50 ms Variablen, 10 bei der 100 ms Variablen und 100 bei der 1 s Variablen) erreicht haben, werden die entsprechenden Flags auf 1 gesetzt und die Zählung unmittelbar von vorne begonnen:

```
ISR(TIMER1_COMPA_vect) //wird alle 10 ms vom Timer ausgelöst
{
    ucTimer1_Flag_10ms = 1;

    ucTimer1_Cnt_1s++;
    if (ucTimer1_Cnt_1s == 100)
    {
        ucTimer1_Cnt_1s = 0;
        ucTimer1_Flag_1s = 1;
    }

    ucTimer1_Cnt_50ms++;
    if (ucTimer1_Cnt_50ms == 5)
    {
        ucTimer1_Cnt_50ms = 0;
        ucTimer1_Flag_50ms = 1;
    }

    ucTimer1_Cnt_100ms++;
    if (ucTimer1_Cnt_100ms == 10)
    {
        ucTimer1_Cnt_100ms = 0;
        ucTimer1_Flag_100ms = 1;
        ulSystemClock++;
    }
}
```

Die Auswertung dieser „Uhren“ geschieht im Modul Timer.c, hier beispielhaft für die 100 ms Uhr dargestellt:

```
char Timer1_get_100msState(void)
{
    if (ucTimer1_Flag_100ms == 1)
    {
        ucTimer1_Flag_100ms = 0;
        return TIMER_TRIGGERED;
    }
    else return TIMER_RUNNING;
}
```

Wenn das Flag gesetzt ist, wird es beim Aufruf der getter-Funktion gelöscht und der Wert `TIMER_TRIGGERED` zurückgegeben, der von 0 verschieden ist. Andernfalls wird 0

zurückgegeben. Dazu müssen natürlich in Timer.h noch folgende Definitionen vorgenommen werden:

```
#define TIMER_RUNNING    0
#define TIMER_TRIGGERED  1
```

Das Hauptprogramm wird nun wie folgt aufgebaut:

```
int main(void)
{
    Timer1_Init();
    while(1)
    {
        if (Timer1_get_50msState()) Task10ms();
        if (Timer1_get_100msState()) Task50ms();
        if (Timer1_get_100msState()) Task100ms();
        IdleTask();
    }
}
```

Wie man leicht sieht, wird die Funktion `Task10ms()` nun alle 10 ms aufgerufen, sofern die While-Schleife in signifikant kürzeren Zyklen durchlaufen wird. Wir nennen diese Funktion daher den 10 ms Task¹. Gleichermaßen gilt für die anderen zeitgesteuerten Tasks. Lediglich der `IdleTask();` wird in jedem Schleifendurchlauf aufgerufen und sollte keinen langen Code enthalten, allenfalls kann man dort einen Pin toggeln um mit dem Oszilloskop die Auslastung des Prozessors prüfen. Im finalen Code wird sie meistens weggelassen.

Mit diesem einfachen Hilfsmittel lässt sich bereits ein einfaches, echtzeitfähiges „Multitasking“-Betriebssystem aufbauen. Die while-Schleife kann man als *Scheduler* bezeichnen. Es ist allerdings äußerst wichtig, dass alle Tasks zusammen schneller ablaufen als die kürzeste Zeit im System. Im oben genannten Beispiel darf also die Prozessorzeit für alle aufgerufenen Tasks zusammen nicht länger als 10 ms betragen, da sonst die Echtzeit nicht mehr eingehalten wird. Man nennt dieses Schema daher auch *kooperatives Multitasking*, da sich alle Tasks kooperativ verhalten müssen und nicht überwacht werden. Im Gegensatz dazu werden beim *präemptiven Multitasking* die Tasks unterbrochen, wenn ein anderer Task an der Reihe ist. Dies geschieht im Timerinterrupt und garantiert eine wirklich genaue Einhaltung der Zeitvorgaben (Echtzeit). Allerdings muss hier erheblicher Aufwand getrieben werden um die Integrität gemeinsam verwendeter Variablen sicherzustellen. Daher wird in diesem Buch auf eine tiefergehende Erläuterung verzichtet.

Im Beispiel oben wurde noch eine Systemuhr eingebaut. Diese zählt als `unsigned long` Variable die 100 ms und würde daher nur alle 429.496.729,6 s wieder bei 0 beginnen (das sind immerhin über 119 Tausend Stunden). Aber schon wenn man die 10 ms oder

¹ Während in der Frühzeit der Datenverarbeitung „Task“ im Deutschen noch mit weiblicher Form („die Task“) genutzt wurde, gilt heute der männliche Genus als korrekt.

gar die Millisekunden zählt, schrumpft dieser Wert wieder unter die Schwelle, in denen realistisch zur Laufzeit ein Überlauf stattfinden kann. Um diesen Zähler als Stoppuhr zu nutzen, wird in timer.c eine Hilfsvariable `ulLastClock` global angelegt. Bei jedem Aufruf der Zählfunktion wird diese auf den aktuellen Timerwert gesetzt und die Differenz zwischen dem aktuellen und dem alten Wert ausgegeben. Erfolgte zwischenzeitliche eine (einfache) Bereichsüberschreitung, dann zählt die Differenz der beiden Zahlen, die vom Maximum des Wertebereichs abgezogen wird. Dieses kann durch das Macro `ULONG_MAX` aus dem Includefile `limits.h` ermittelt werden, das automatisch geladen wird. Auf diese Weise kann die Stoppuhr natürlich auch mit kürzeren Variablen implementiert werden.

```
unsigned long TimerMeasure(void) //liefert zurück, wieviel Zeit
// zwischen dem letzten und dem aktuellen Aufruf vergangen ist
{
    unsigned long diff;

    if (ulSystemClock > ulLastClock)
    {
        diff = ulSystemClock - ulLastClock;
    }
    else
    {
        diff = ULONG_MAX - (ulLastClock - ulSystemClock) ;
    }
    ulLastClock = ulSystemClock;
    return diff;
}
```

Es muss klar sein, dass hier keine Überrundungen vorgesehen sind. Gegebenenfalls muss noch ein „Rundenzähler“ mit eingebaut werden.

4.5 Wichtige Speicherkonzepte

4.5.1 Warteschlangen und Ringpuffer (FIFO)

Warteschlangen² werden immer dann eingesetzt, wenn zwei zeitlich voneinander unabhängige, mitunter stochastische Prozesse miteinander gekoppelt werden. Im Bereich eingebetteter Systeme der in diesem Buch beschriebenen Größe können Warteschlangen eingesetzt werden, wenn beispielsweise eine regelmäßige Messung über eine unsicher verfügbare Funkschnittstelle übertragen wird oder die Daten über ein gebündeltes Paket übertragen werden sollen. Ein weiterer Anwendungsfall liegt vor, wenn asynchrone Eingangsrößen aus einem Netzwerk in einem isochronen Prozess³ verarbeitet werden sollen

² Engl. *queue*.

³ Von griech $\iota\sigma o\varsigma$ = „gleich“: Abtastprozesse mit immer gleichen Abtastabständen.

(Puffer). Aufgrund der Zugriffsreihenfolge First in – First Out werden Warteschlangen auch FIFO genannt.

Eine Warteschlange kennt zwei Befehle: `QueuePut()` und `QueueGet()`. Damit wird ein neues Element in die Warteschlange geschoben (put) und aus ihr entnommen (get).

Eine Warteschlange kann durch ein Array oder durch eine verkettete Liste dargestellt werden. Bei einem Array mit definierter Länge markieren zwei Indizes den Zustand des FIFO: `QueueIn` und `QueueOut`. Beide stehen nach der Initialisierung auf 0, wie in Abb. 4.2a zu sehen ist. Wird nun ein Element in den FIFO geschrieben, wird der Zeiger `QueueIn` inkrementiert, bis er schließlich am letzten Element des Array angekommen ist. Wird ein Element aus dem FIFO entnommen, wird der Zeiger `QueueOut` inkrementiert und damit auch der Platz im Array freigegeben. Man muss nun sicherstellen, dass `QueueOut` nicht über `QueueIn` hinausweist, da sonst ein ungültiges, d.h. nicht aktuell eingeschriebenes Element ausgegeben würde. Das Hauptproblem ist aber, dass die Feldgrenzen des Arrays schnell erschöpft würden und ein regelmäßiges Umspeichern der Inhalte zu aufwändig wäre. FIFOs mit Array werden daher in der Regel als *Ringpuffer* ausgelegt.

Bei einem Ringpuffer werden die beiden Indizes auf jeweils wieder auf Null gesetzt, wenn sie die Arraygröße erreicht haben. Damit wird ein geschlossener Ring erreicht. Der Out-Index wandert immer hinter dem In-Index hinterher, wenn er ihn eingeholt hat, ist der FIFO leer und kein Element kann mehr entnommen werden. Umgekehrt, wenn der In-Index den Out-Index von hinten „überschritten“ hat, ist der FIFO voll und es muss entschieden werden, ob alte Elemente überschrieben werden oder die Befüllung gestoppt wird.

Wir definieren zunächst:

```
#define QUEUE_SIZE 10
#define QUEUE_FULL -1
#define QUEUE_EMPTY -2
#define QUEUE_NOERROR 0
int QueueIn, QueueOut;
```

Außerdem benötigen wir den Inhalt des FIFO, dies kann beispielsweise eine Struktur sein, oder einfach ein elementarer Datentyp.

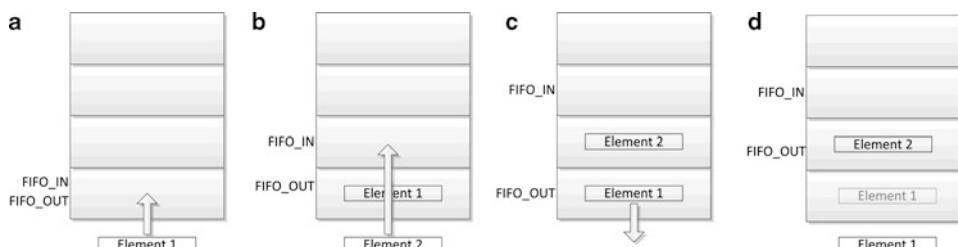


Abb. 4.2 FIFO-Prinzip

```
typedef int FIFO_t;
```

Im Fall einer Busbotschaft, könnte er auch so aussehen:

```
typedef struct msg_s {
    long msg_addr;
    unsigned char data[8];
} FIFO_t;
```

Die eigentliche Queue ist dann:

```
FIFO_t Queue[QUEUE_SIZE];
```

Die Inkrementierung des In-Index sieht im einfachsten Fall so aus:

```
QueueIn++;
if (QueueIn+1 >= QUEUE_SIZE) QueueIn= 0;
```

Damit ist ein Ringpuffer realisiert. Es fehlt noch die Abfrage, ob der Puffer voll ist. Dies ist der Fall, wenn QueueIn genau vor QueueOut steht bzw. wenn QueueIn gerade am Ende des Arrays steht und QueueOut am Anfang des Arrays:

```
int QueuePut(FIFO_t new_element)
{
    if ((QueueIn + 1 == QueueOut) ||
        (QueueOut == 0 && QueueIn == QUEUE_SIZE-1))
        return QUEUE_FULL ;
    Queue[QueueIn] = new_element;

    QueueIn++;
    if (QueueIn >= QUEUE_SIZE) QueueIn= 0;
    return QUEUE_NOERROR; //No errors
}
```

Fehlt noch das Lesen aus dem FIFO, das in derselben Weise stattfindet:

```
int QueueGet(FIFO_t *result)
{
    if(QueueIn == QueueOut)
    {
        return QUEUE_EMPTY;
    }
```

```

    *result = Queue[QueueOut];
    QueueOut++;
    if (QueueOut >= QUEUE_SIZE) QueueOut = 0;
    return QUEUE_NOERROR;
}

```

Wir übergeben einen Fehlercode als Funktionsergebnis, während der eigentliche Inhalt der Queue als Zeiger übergeben wird.

Die aufwändige Abfrage nach der Arraygrenze kann man sich sparen, wenn man den Index über eine Modulo-Operation ermittelt [4]:

```
QueueIn = (QueueIn + 1) % QUEUE_SIZE;
```

Dies hat jedoch einige Nachteile, beispielsweise ist die Modulo-Operation nur dann „billig“, wenn QUEUE_SIZE eine Zweierpotenz ist, dann kann sie nämlich durch eine Bitmaske ersetzt werden [5]:

```
#define QUEUE_SIZE 16 //muss 2^n betragen (8, 16, 32, 64 ...)
#define QUEUE_MASK (QUEUE_SIZE-1) //Klammern auf keinen Fall vergessen
```

Und die Inkrement-Operation sieht dann wie folgt aus:

```
QueueIn = ((QueueIn + 1) & QUEUE_MASK);
```

Neben diesem relativ simplen Beispiel, existieren noch beliebig viele Beispiele im Netz, wie man an [4] und [5] sehen kann.

- Wenn gewünscht ist, dass die Warteschlange aus einem Interrupt gelesen oder beschrieben werden soll, ist es wichtig, dass alle Interrupts vor dem Zugriff auf die Queue deaktiviert werden!

4.5.2 Warteschlange mit dynamischen Datenstrukturen

Alternativ kann man eine Warteschlange auch mit so genannten verketteten Listen realisieren. Der Vorteil dieser dynamischen Datenstrukturen besteht darin, dass nur so viel Speicher genutzt wird, wie tatsächlich benötigt wird, der Nachteil ist, dass man extrem vorsichtig sein muss um den Speicher nicht bis zum Anschlag zu füllen.

In einer Liste existiert jedes Element zunächst einmal für sich unabhängig im Speicher. Um in einer Liste von einem Element zum anderen zu kommen, besitzt jedes Element einen Zeiger auf seinen Nachfolger (siehe Abb. 4.3). Somit entsteht ohne weitere Verwaltung die Verkettung von Elementen, die man durchlaufen kann, indem man sich das erste Element merkt und dann von Element zu Element springt. Genau ein Element in der

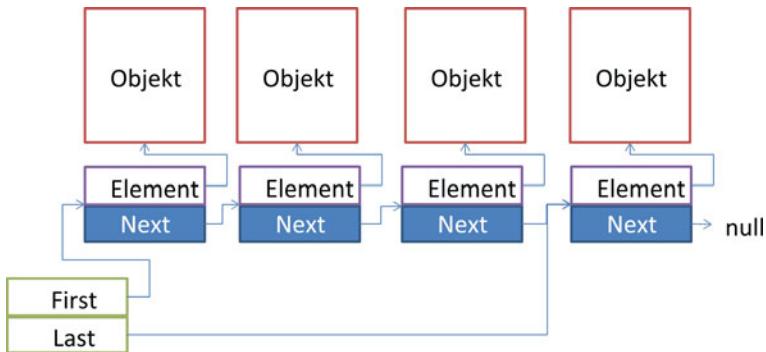


Abb. 4.3 Aufbau einer einfach verketteten Liste

Liste hat keinen Nachfolger, genauer gesagt, dort zeigt der Zeiger auf den Nachfolger auf NULL. Dieses Element nennt man das letzte Element in der Liste.

Wenn eine Liste aufgebaut wird, wird ein neues Element erzeugt und mit einer Referenz seinem Vorgänger bekannt gemacht. In einer einfach verketteten Liste kennt es den Vorgänger allerdings nicht. Die Nachfolger-Beziehung ist also eine einfache, gerichtete Beziehung. Neben den Elementen benötigen wir also mindestens eine weitere Hilfsgröße, nämlich eine Referenz auf das erste Element der Liste `first`. Praktischerweise werden wir allerdings auch eine Referenz auf das letzte Objekt `last` speichern, da sonst beim Erzeugen jedes neuen Elements zunächst die gesamte Liste bis zum letzten Element transversiert werden müsste.

Würde der Zeiger `last` auf das erste Element zeigen, hätten wir wieder einen Ringpuffer oder Zyklus. In Abb. 4.3 ist angedeutet, dass die Listenelemente wiederum auf den eigentlichen Inhalt verweisen. Dies kann, muss aber nicht der Fall sein. Im am Ende von Kap. 3 bereits verwendeten Beispiel wird ein Listenelement als Struktur wie folgt aufgebaut:

```
struct sListItem {char i; struct sListItem *next;} ;
typedef struct sListItem tListItem;
tListItem *first;
tListItem *last;
```

Die Variable `i` steht für den Inhalt, der natürlich auch komplexer sein kann.

Um aus einer verketteten Liste einen FIFO zu bauen, machen wir folgendes Gedankenexperiment: Jedes neue Element, das in den FIFO geschoben wird, wird einfach an das letzte Element angehängt. Eine Funktion zum Anhängen eines Elements an eine verkettete List muss zunächst prüfen, ob die Liste noch leer ist oder ob sie bereits Elemente enthält. Im ersten Fall wird dem Zeiger `first` die Adresse des ersten Elements zugewiesen, das damit natürlich auch zum letzten Element wird. Ein neues Element wird mit der in Abschn. 3.11 beschriebenen Funktion `malloc()` erzeugt. Schlägt die Erzeugung fehl,

weil beispielsweise kein Heap-Speicher mehr zur Verfügung steht, muss die Funktion natürlich abbrechen und einen Fehlercode ausgeben. Sind bereits Elemente vorhanden, wird das neue Element an das bisherige letzte Element angehängt und wird damit selbst zum letzten Element.

```
char FIFOput(char value) //Rückgabe von 0, wenn Funktion erfolgreich war
{
    if (first==NULL) //Liste war bisher leer
    {
        first = (tListItem*) malloc (sizeof(tListItem));
        if (first == NULL) return -1; //malloc fehlgeschlagen
        last = first;
    }
    else //Liste ist bereits vorhanden
    {
        last->next = (tListItem*) malloc (sizeof(tListItem));
        if (last->next == NULL) return -1; //malloc fehlgeschlagen
        else last = last->next; //last muss angepasst werden
    }
    last->i = value; //Zuweisung des Wertes
    last->next = NULL; //Sicherstellen, dass das letzte Element
                        //auf NULL zeigt
    return 0;
}
```

Das Gedankenexperiment geht weiter: Jedes Mal, wenn ein Element aus dem FIFO entnommen worden ist, kann es gelöscht, das heißt zum weiteren Beschreiben freigegeben werden. Dies erfolgt, indem man das erste Element löscht und den Arbeitszeiger `first` auf das bisher zweite Element zeigen lässt. Hier muss man natürlich aufpassen, dass man nicht aus Versehen einen Zeiger verliert, daher wird zunächst ein Hilfszeiger `buf` angelegt:

```
char FIFOget(char *value)
{
    tListItem *buf;
    buf = first;      //Retten des Zeigers
    if (buf == NULL) //Liste war leer
    {
        return -1;
    }
    else first = first->next; //Umhängen des Zeigers
    *value = buf->i; //Ausgeben des Wertes
    free(buf); //...und Freigeben des Heaps
    return 0;
}
```

Selbstverständlich muss überprüft werden, ob die Liste überhaupt ein Element enthalten hat. Aus diesem Grund und auch deshalb, weil Listenelemente (fast) beliebig komplex aufgebaut sein können, gibt man den Inhalt des Elements indirekt über einen Zeiger aus (Abschn. 2.9.5). Der eigentliche Rückgabewert der Funktion ist 0, wenn sie erfolgreich war und im übergebenen Bereich ein gültiger Wert steht oder –1, wenn die Liste leer war.

Man kann nun beispielsweise aus einer ISR oder einem Task den FIFO befüllen:

```
FIFOput(val);
```

und dann in einem anderen Task über eine serielle Schnittstelle ausgeben (Abschn. 5.2.1)

```
char res;  
FIFOget(&res);  
uartTransmitChar(res);
```

4.5.3 Mehrere Warteschlangen in einem Programm

Sollte es tatsächlich einmal vorkommen, dass man mehrere Warteschlangen in einem Programm nutzen möchte, dann müssen die Funktionen umgeschrieben werden. Jede Warteschlange bekommt dann einen so genannten „Handle“, also einen Handgriff. Im Fall der Ringliste ist das natürlich der Zeiger auf das betreffende Array sowie die beiden Arbeitsindizes QueueIn und QueueOut:

```
typedef struct sFIFOH {  
    int QueueIn;  
    int QueueOut;  
    FIFO_t *Fifo;} FIFOH_t;
```

Die Initialisierung sieht dann wie folgt aus:

```
FIFOH_t MeinFIFO;  
FIFO_t Daten[FIFOLENGTH];  
QueueInit(&MeinFIFO, Daten);
```

Entsprechend müssen dann in den beiden Zugriffs Routinen die beiden Arbeitsindizes aus dem Handle genutzt werden. Der notwendige Beispielcode ist dem Zusatzmaterial zum Buch zu entnehmen.

Auf dieselbe Weise wird man mit den FIFOs vom Typ verkettete Liste umgehen, diese sind entsprechend einfacher aufgebaut und der Handle enthält nur die Zeiger auf der ersten und das letzte Element:

```
typedef struct sFIFOH {
    tListItem *first;
    tListItem *last;
} FIFOH_t;
```

Beim Zugriff wird dann statt auf `first` und `last` zu verweisen auf den jeweiligen Zeiger des Handles verwiesen:

```
char FIFOput(FIFOH_t *Handle; char value)
{
    if (Handle->first == NULL)
    {
        ...
    }
```

Für jede neue Warteschlange wird ein Handle angelegt

```
FIFOH_t MeinFIFO;
MeinFIFO.first = NULL;
MeinFIFO.last = NULL;
...
FIFOput(&MeinFIFO, value);
...
```

Sollen die verschiedenen Warteschlangen unterschiedliche Datentypen halten, muss der Handle ausgebaut werden. In den Codebeispielen auf der Webseite des Buches ist dies ausgeführt.

4.6 Zustandsautomaten (Statemachine)

4.6.1 Allgemeine Betrachtung

Endliche Zustandsautomaten (engl. Finite State Machine) sind mächtige Konzepte für das Verhalten von Maschinen, indem sie die gedächtnisbehaftete Reaktion eines Systems auf innere oder äußere Ereignisse modellieren. Dazu wird zunächst eine endliche (finite) Menge von Zuständen (states) definiert, in denen – und nur in denen – sich die Maschine befinden kann. Weiterhin existiert eine Liste von Ereignissen, auf die die Maschine reagiert, indem sie von einem Zustand in einen anderen wechselt. Dieser Wechsel kann mit einer äußerlich sichtbaren Aktion, also einem Output verbunden sein, auch kann das Verharren in einem Zustand mit einem Output verbunden sein. Wichtig dabei ist das „Gedächtnis“, damit ist ein neuer Zustand nicht nur vom Input sondern auch vom bisherigen Zustand abhängig. Man kann also sagen, dass ein Zustandsautomat die Menge von Zuständen z_i^n zum Zeitpunkt n auf eine Menge von Zuständen z_i^{n+1} zum Zeitpunkt $n + 1$ abbildet

und zwar als Funktion von Ereignissen oder Eingangsgrößen e_j .

$$z_i^{n+1} = f(z_i^n, e_j) \quad (4.1)$$

und zusätzlich wird die Menge der aktuellen Zustände und Eingangsgrößen auf eine Menge von Reaktionen oder Ausgangsgrößen o_k abgebildet:

$$o_k = f(z_i^n, e_j) \quad (4.2)$$

Abb. 4.4 verdeutlicht diesen Zusammenhang.

Der Übergang zwischen zwei Zuständen heißt Transition. Äußere Transitionen sind Transitionen, bei denen ein Zustand verlassen und ein neuer Zustand eingenommen wird. Innere Transitionen sind solche, bei denen ein Ereignis eine Aktivität aber keinen Zustandswechsel hervorruft.

Eine Transition kann asynchron zustande kommen, indem ein spontanes Ereignis auch einen spontanen Wechsel auslöst. Meistens jedoch wird man Zustandsautomaten getaktet (synchron) ausführen, das Ereignis wird zwischengespeichert und im nächsten Schritt ausgewertet.

Ereignisse können sein:

- Empfang eines Zeichens oder einer Zeichenkette über eine serielle Schnittstelle
- Ein Messwert verändert sich oder überschreitet eine definierte obere oder untere Schwelle
- Der Zustand eines digitalen Eingangs verändert sich (Flanke), beispielsweise wenn eine Taste gedrückt wurde

Transitionen können durch Bedingungen (guard) eingeschränkt werden, in diesem Fall wird die Transition bei Eintreten des Ereignisses nur durchgeführt, wenn auch die Bedingung erfüllt ist.

Bedingungen können beispielsweise sein:

- Eine bestimmte Zeit ist seit dem letzten Zustandswechsel verstrichen oder allgemein:
- Eine berechnete Variable überschreitet eine definierte Schwelle

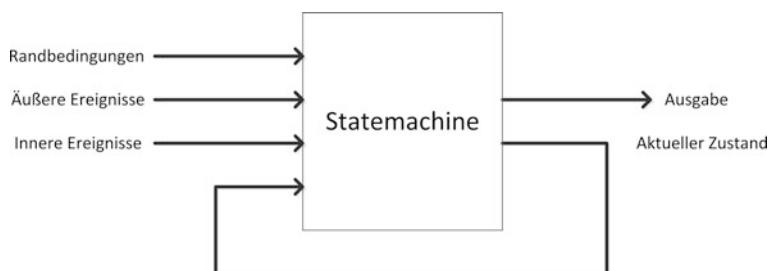


Abb. 4.4 Allgemeine Ausführung eines Zustandsautomaten

Besonders häufige Anwendungsfälle von Zustandsautomaten sind:

- Die Steuerung sequentieller Abläufe, die auf externen Ereignissen beruhen
- Mensch-Maschine-Schnittstellen: Reaktionen auf Benutzereingaben und Maschinenergebnisse
- Die Realisierung von Kommunikationsprotokollen

Das äußere Verhalten des Zustandsautomaten ist mit den Zuständen und den Zustandsübergängen verknüpft und wird durch Aktivitäten (oder Aktionen) beschrieben. Wir unterscheiden:

- Aktivitäten, die ausgeführt werden, wenn der Zustandsautomat in den Zustand eintritt (entry behaviour)
- Aktivitäten, die ausgeführt werden, wenn der Zustandsautomat den Zustand verlässt (engl. exit behaviour)
- Aktivitäten, die ausgeführt werden, während sich der Zustandsautomat im Zustand befindet (engl. doActivity)
- Aktivitäten, die ausgeführt werden, wenn ein bestimmtes Event eintritt (eng. event behaviour)

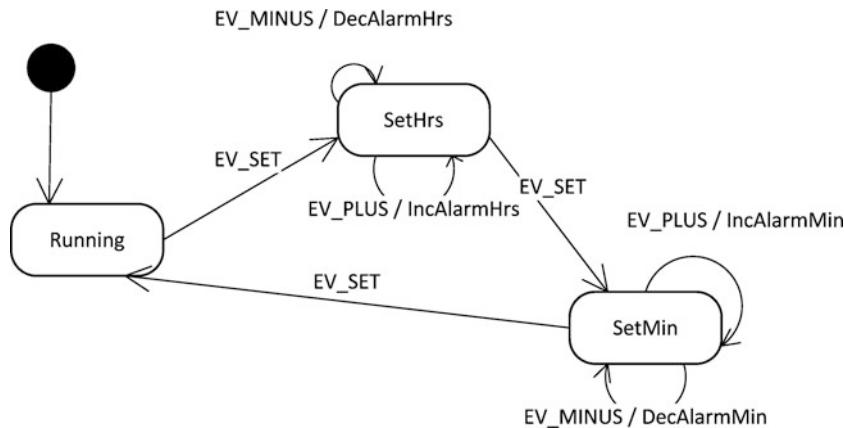
4.6.2 Beschreibung von Zustandsautomaten

Für die Beschreibung von Zustandsautomaten wird entweder das in der UML (unified modelling language) definierte Zustandsdiagramm (engl. state chart) genutzt [6, 7] oder die Zustandsübergangstabelle. Die beiden Techniken werden hier anhand des folgenden Beispiels vorgestellt.

Beispiel

HMI-Anwendung: Oftmals müssen über wenige Eingabetasten die Parameter einer Maschine eingestellt werden. Ein einfaches Beispiel ist das Setzen der Zeit für einen Wecker, der die Tasten SET und + bzw. – kennt. Durch Drücken von SET erwartet der Wecker die Eingabe (Inkrement/Dekrement) zunächst von Stunden, dann von Minuten.

Das Zustandsdiagramm sieht dabei so aus, wie in Abb. 4.5 gezeigt, der Lesbarkeit halber wurden hier die Aktionen weggelassen. Man erkennt in diesem Diagramm bereits die wichtigsten Sprachelemente des UML-Zustandsdiagramms. Der schwarze gefüllte Kreis ist ein so genannter Pseudozustand, hier der Startzustand. Nach dem Reset befindet sich der Automat im Startzustand und wechselt dann ohne weiteres Ereignis sofort in den Zustand „running“. Der Startzustand besitzt demnach nur eine einzige Transition, die angibt, in welchem Zustand die Maschine startet. Die Transitionen sind mit den Ereignissen belegt, die die Transition auslösen. Im obigen Beispiel ist EV_SET ein Ereignis, das eine

**Abb. 4.5** Zustandsautomat des Weckers für die Zeiteinstellung

äußere Transition auslöst, während EV_PLUS und EV_MINUS innere Transitionen auslösen.

In einer Zustandsfolgetabelle sieht dieselbe Beschreibung wie in Tab. 4.1 aus.

Aus Übersichtsgründen wurden hier die Aktionen nicht in do/entry/exit und Ereignisaktionen unterschieden. Eine typische do-Aktion wäre hier, die Ziffer für die Stunden blinken zu lassen, während sich der Automat im Zustand SetHrs befindet und entsprechend im Zustand SetMin die Ziffer für die Minuten blinken zu lassen (Abb. 4.5).

In Abb. 4.6 sind wichtige Sprachelemente für das Zustandsdiagramm zusammengefasst.

Insbesondere sind hier die folgenden Pseudozustände zu erkennen:

- Der Startzustand besitzt eine Transition, die auf den Zustand hinweist, der beim Start der Statemachine eingenommen wird
- Der Endzustand bezeichnet die Terminierung der Statemachine. Er besitzt keine ausgehenden Transitionen

Tab. 4.1 Zustandsfolgetabelle für den Wecker

Zustand	Ereignis	Folgezustand	Aktion
Running	Set	SetHrs	
SetHrs	Plus	SetHrs	IncAlarmHrs
SetHrs	Minus	SetHrs	DecAlarmHrs
SetHrs	Set	SetMin	
SetMin	Plus	SetMin	IncAlarmMin
SetMin	Minus	SetMin	DecAlarmMin
SetMin	Set	Running	

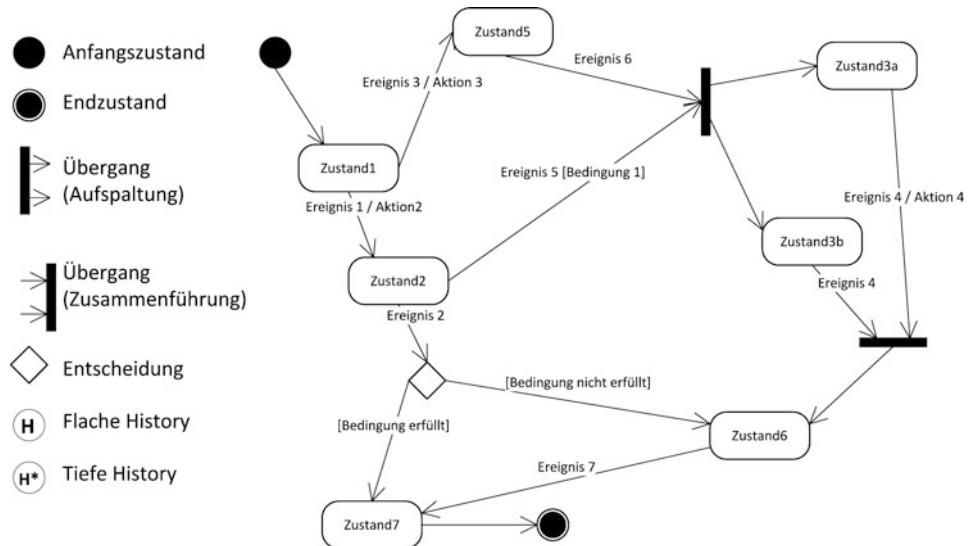


Abb. 4.6 UML-Sprachelemente des Zustandsdiagramms

- Der Übergang ist eine Aufspaltung in zwei parallele Zustände oder eine Zusammenführung zweier paralleler Zustände zu einem.
- Bei der Entscheidung können Bedingungen die Auswahl alternativer Transitionen steuern

Zustandsautomaten kann man auch hierarchisch gliedern. Im folgenden Beispiel Abb. 4.7 ist ein System im Überzustand „running“ und kann dort die Zustände 1 bis 3 annehmen. Sobald ein Fehler auftaucht, wechselt das System in den Überzustand „error“ und zwar aus jedem der Zustände 1 bis 3. Dort wird der Fehler angezeigt und kann quittiert werden. Nach der Quittierung wartet das System auf die Behebung des Fehlers. Sobald er

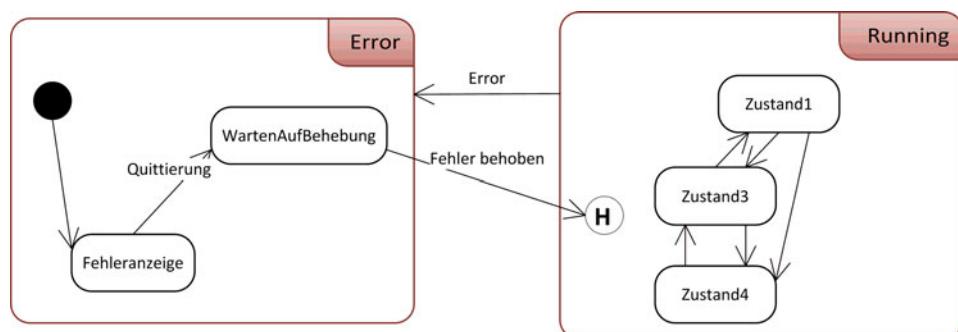


Abb. 4.7 Hierarchische Zustandsautomaten

beoben ist, springt das System in den Überzustand „running“ zurück und zwar dort, wo es zuletzt gestanden hatte. Dies wird durch die so genannten „History“-Pseudozustände markiert. Dabei bezeichnet eine „tiefe“ History einen Merker, der über mehrere Ebenen von hierarchischen Zustandsautomaten geht, während die „flache“ History immer nur die nächste Ebene berücksichtigt.

4.6.3 Umsetzung von Zustandsautomaten auf Mikrocontrollern

Im oben genannten Beispiel des Weckers sind drei Zustände und drei Ereignisse definiert, die in C so abgebildet werden können, wobei ein „Nicht-Ereignis“ `EV_NOEVENT` anzeigt, dass gerade kein Ereignis vorliegt:

```
#define STATE_RUNNING 0
#define STATE_SETHRS 1
#define STATE_SETMIN 2
#define EV_SET 1
#define EV_PLUS 2
#define EV_MINUS 3
#define EV_NOEVENT 0
unsigned char state = STATE_RUNNING; //Startzustand
unsigned char event = EV_NOEVENT;
```

In der Hauptschleife des Programms wird zunächst das Ereignis ausgewertet, dann die Transition durchgeführt beziehungsweise die Transitionsaktivität ausgeführt, anschließend wird die do-Aktivität ausgeführt:

```
while(1)
{
    event = ReadEvent();
    state = statemachine(event);
    PerformDoAction(state);
}
```

Der eigentliche Zustandsautomat kann auf verschiedene Weise aufgebaut sein. Im folgenden Code beruht er auf geschachtelte `switch`-Anweisungen, die natürlich auch durch Mehrfach-Verzweigungen mit `if` ersetzt werden können. Der Vorteil von `switch` ist die Tatsache, dass der Compiler diese Anweisungen als Tabellen übersetzt und insofern deutlich effizienter arbeiten lässt.

```
unsigned char statemachine(unsigned char event)
{
    switch (state)
```

```

{
    case STATE_RUNNING: switch (event)
    {
        case EV_SET: state = STATE_SETHRS; break;
        default: break;
    } break;
    case STATE_SETHRS: switch (event)
    {
        case EV_SET: state = STATE_SETMIN; break;
        case EV_PLUS: IncAlarmHrs(); break;
        case EV_MINUS: DecAlarmHrs(); break;
        default: break;
    } break ;
    case STATE_SETMIN: switch (event)
    {
        case EV_SET: state = STATE_RUNNING; break;
        case EV_PLUS: IncAlarmMin(); break;
        case EV_MINUS: DecAlarmMin(); break;
        default: break;
    } break;
    default: break;
}
event = EV_NOEVENT;
return state;
}

```

Die Transitionsaktionen, hier die Inkrementierungen und Dekrementierungen von Stunden und Minuten sehen beispielsweise so aus:

```

void IncAlarmHrs (){
    alarmHrs++;
    if (alarmHrs == 24) alarmHrs = 0;
}

```

Anstelle der bisweilen unübersichtlichen Verzweigungen kann auch direkt eine Zustandstabelle programmiert werden, in diesem Fall ist das eine mit Zeigern auf die Transitionsaktionen. Ein Eintrag sieht wie folgt aus:

```

struct sStateTableEntry{
    unsigned char state; //aktueller Zustand
    unsigned char event; //Ereignis
    unsigned char newstate; //Neuer Zustand
    void (*transition)(); //Transitionsaktion
};

```

Für den Wecker lautet die beispielhafte Implementierung der Tabelle:

```
#define STLEN 7
struct sStateTableEntry stm[] = {{STATE_RUNNING,EV_SET,STATE_SETHRS,NULL},
{STATE_SETHRS,EV_SET,STATE_SETMIN,NULL},
{STATE_SETHRS,EV_PLUS,STATE_SETHRS,IncAlarmHrs},
{STATE_SETHRS,EV_MINUS,STATE_SETHRS,DecAlarmHrs},
{STATE_SETMIN,EV_PLUS,STATE_SETMIN,IncAlarmMin},
{STATE_SETMIN,EV_MINUS,STATE_SETMIN,DecAlarmMin},
{STATE_SETMIN,EV_SET,STATE_RUNNING,NULL}};
```

Um nun den Automaten zu betreiben, wird die Funktion `statemachine()` von oben durch `statemachine2()` ersetzt:

```
unsigned char statemachine2(unsigned char ev)
{
    int i;
    if (ev == EV_NOEVENT) return state;
    for (i = 0; i < STLEN; i++)
    {
        if (state == stm[i].state && ev == stm[i].event)
        {
            state = stm[i].newstate;
            if (stm[i].transition != NULL) stm[i].transition();
            event = EV_NOEVENT;
            return state;
        }
    }
}
```

Zunächst wird abgeprüft, ob ein gültiges Ereignis eingetroffen ist, ansonsten wird der Zustand beibehalten. Dann sucht der Algorithmus in der For-Schleife, ob ein Zustand und ein Ereignis der aktuellen Kombination entsprechen und wechselt in den passenden Folgezustand. Falls eine Transitionsaktion definiert war, wird diese noch ausgeführt.

Literatur

1. IEEE: IEEE 1471 – recommended practice for architectural description of software intensive systems (2007). <http://www.iso-architecture.org/ieee-1471>, Zugegriffen: 3. Januar 2015
2. Starke, G., Hruschka, P.: Software-Architektur kompakt, 2. Aufl. Spektrum akademischer Verlag, Heidelberg (2011)
3. Microchip Technology Inc.: AVR035: efficient C coding for AVR (2004). www.microchip.com, Zugegriffen: 16. April 2018
4. <http://stackoverflow.com/questions/215557/how-do-i-implement-a-circular-list-ring-buffer-in-c>, Zugegriffen: 14. April 2018

5. <http://www.mikrocontroller.net/articles/FIFO>, Zugegriffen: 18. April 2018
6. Booch, G., Rumbaugh, J., Jacobson, I.: Das UML-Benutzerhandbuch. Aktuell zur Version 2.0. Addison-Wesley, Bonn (2006). ISBN 978-3827325709
7. Balzert, H.: UML 2 in 5 Tagen. W3L, Dortmund (2005). ISBN 978-3937137612

Kommunikationsschnittstellen

5

Zusammenfassung

Dieses Kapitel öffnet das Fenster des Mikrocontrollers zur Außenwelt. Bewusst wurde in den vorhergehenden Kapiteln die Kommunikation ausgespart. Nun ist es jedoch an der Zeit, die seriellen Schnittstellen der AVR-Familie vorzustellen und anschließend auf wichtige Netzwerktypen einzugehen, die über diese Schnittstellen angesprochen werden können. Mithin verwenden alle später vorgestellten Peripheriebausteine eine dieser Schnittstellen. Hierzu sind jedoch einige Vorbetrachtungen notwendig (Abschn. 5.1) bevor die „Klassiker“ UART, SPI und I²C beschrieben werden (Abschn. 5.2). Anschließend sind zwei in der Automatisierungstechnik wichtige Busysteme an der Reihe: Das aus dem Automobilbau stammende CAN-Netzwerk und der im Fabrikumfeld genutzte MODBUS. Der letzte Abschnitt widmet sich dann in aller Kürze einigen wichtigen Funkstandards, für deren Implementierung sei auf die Literatur verwiesen.

5.1 Das ISO/OSI Schichtenmodell¹

Kommunikation findet statt, wenn Daten² von mindestens einer Quelle (Sender) an mindestens eine Senke (Empfänger) über einen Nachrichtenkanal übertragen werden und diese Daten für Sender und Empfänger dieselbe Bedeutung (Semantik) besitzen. Dazu müssen vorab Informationen über die Art der Kommunikation, speziell

¹ Dieser Abschnitt wurde dem Buch Meroth, Tolg: „Bussysteme im Kfz“ [1] entnommen und stark überarbeitet.

² Erst durch Interpretation durch den Empfänger werden daraus Informationen.

- die Codierung der Daten durch Signale,
- die physikalischen Parameter der Signale und des Nachrichtenkanals und
- das verwendete Protokoll

bekannt sein. Diese Vereinbarungen über den Ablauf der Kommunikation und die dabei ausgetauschten Daten nennt man Protokolle. Sie können sich auf physikalische Abläufe (z. B. Herstellen einer physikalischen Verbindung), Abläufe zur Kommunikationssteuerung (z. B. Verbindungsaufbau, Routing von Datenpaketen, Diagnose) und auf die Übertragung der Nutzdaten selbst beziehen. Damit Kommunikationsteilnehmer unterschiedlicher Hersteller dieselben Protokolle nutzen können, sind diese in der Regel standardisiert. Den meisten Kommunikationsstandards liegt das so genannte OSI³ Schichtenmodell zugrunde. Es untergliedert die Verfahren zur Datenübertragung in sieben Schichten mit jeweils speziellen Aufgaben. Diesem Ansatz liegt die Idee zugrunde, dass Software-Anwendungen nicht direkt mit ihrer Gegenanwendung kommunizieren – wie auch? –, sondern über einen Dienstzugangspunkt mit einem Dienst, der die Nachricht an untere Instanzen des Betriebssystems und dann über die Hardware, gegebenenfalls über Vermittlungsstel-

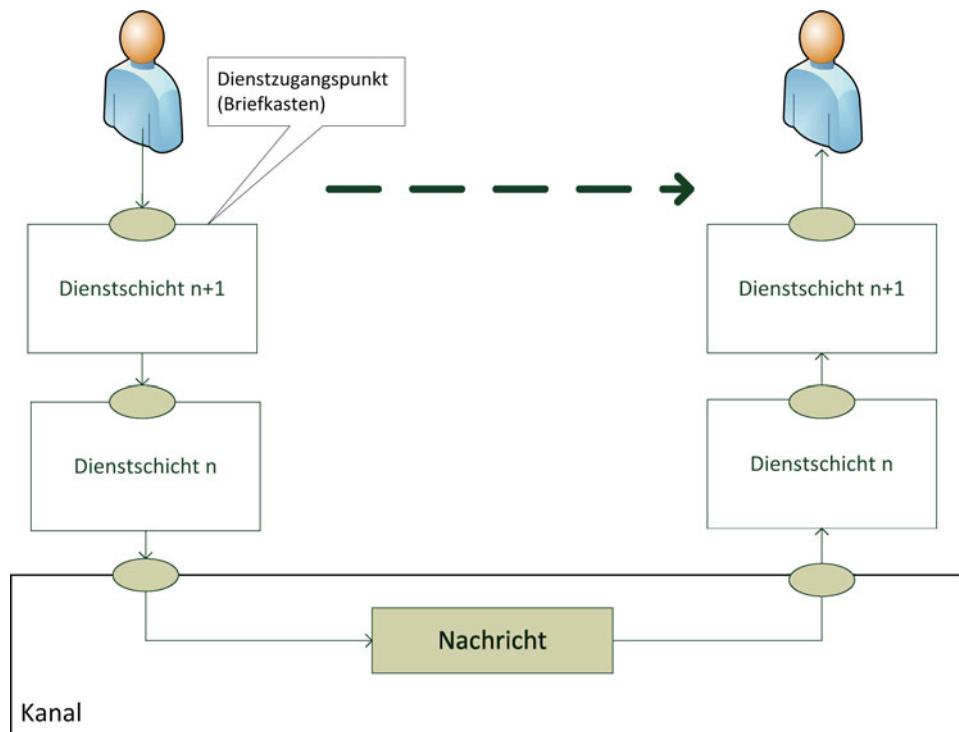


Abb. 5.1 Vertikale und horizontale Kommunikation im Schichtenmodell

³ Open Systems Interconnection Reference Model.

len an den Empfängerrechner weiter gibt, der seinerseits die Nachricht der betroffenen Anwendung zur Verfügung stellt (vertikale Kommunikation vs. horizontale Kommunikation), wie die Abb. 5.1 zeigt.

Für die Kommunikationssteuerung hat dies beträchtliche Konsequenzen. Die Schichten, die als eigenständige Instanzen, beispielsweise als Teil eines Betriebssystems oder einer Laufzeitumgebung existieren können, benötigen für die (virtuelle) horizontale Kommunikation untereinander Protokolle aber auch die Nutzung der nächsttiefenliegenden Schicht für die vertikale Kommunikation erfolgt nach einem Protokoll. Die dafür benötigten Protokolldaten (für die horizontale) bzw. Interfacedaten (für die vertikale Kommunikation) werden vor (Header) oder hinter (Trailer) die eigentlichen Nutzdaten (engl. Payload) angehängt. Damit erhöht sich die Größe des Datenpakets von Schicht zu Schicht, wobei Protokolldaten und Nutzdaten der nächsthöheren Schicht als Nutzdaten der nächsttiefen Schicht interpretiert werden (s. Abb. 5.2). Man spricht hier von einem Protokollstapel (engl. protocol stack). Bei der Betrachtung der Datenrate auf einem Bussystem ist sorgfältig zwischen der Brutto-Datenrate und der Netto-Datenrate zu unterscheiden, die sich je nach Protokolleffizienz deutlich davon unterscheiden kann.

Das OSI-Modell fungiert als Ordnungsrahmen, der beim Entwurf neuer Protokolle eine generelle Vorgehensweise vorgibt⁴.

In der Tab. 5.1 sind die Schichten des OSI-Modells im Überblick dargestellt, in den folgenden Abschnitten werden dann Aspekte aus den Schichten 1–4 vorgestellt.

Primär unterscheiden sich die in diesem Kapitel beschriebenen drahtgebundenen Netzwerke zunächst in den beiden untersten Schichten. Darüber liegen unterschiedliche Protokolle, die allerdings nicht in diesem Buch beschrieben werden. Es existiert aber ein starker Trend zur Vereinheitlichung der oberen Schichten in Richtung TCP/IP.

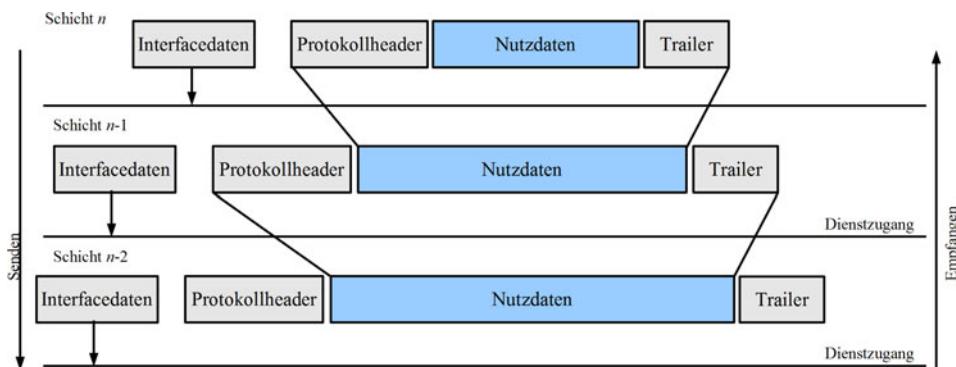


Abb. 5.2 Header und Trailer als Bestandteil des Protokollstapels

⁴ Spezifiziert im Standard ISO 7498.

Tab. 5.1 Schichten des ISO/OSI Modells

Schicht	Titel	Funktionen	Standard (ISO-OSI)	Beispiel Protokoll
1	Physical (Bitübertragung)	Elektrische Konnektivität, z. B. Kabel, Lichtleiter oder Antenne	ISO 10022	RS232, 10Base-T, WLAN
2	Data Link (Sicherung)	Sicherung der Übertragung (Fehlerbehandlung, Netzwerkzugriff)	ISO 8886	Ethernet, LLC (IEEE 802.2)
3	Network (Vermittlung)	Schalten von Verbindungen, Weiterleiten von Datenpaketen; Aufbau und Aktualisierung von Routing-Informationen; Flusskontrolle	ISO 8348	IP
4	Transport (Transport)	Übertragung der Nutzdaten; Segmentation und Multiplexen von Datenpaketen; Kontrolle der Integrität der Daten mit Prüfsummen; Fehlerbehebungsmechanismen; Abstraktion der unteren Schichten gegenüber der Anwendung	ISO 8073	TCP, UDP
5	Session (Sitzung)	Stellt die Unversehrtheit der Verbindung zwischen zwei Netzteilnehmern sicher; Steuerung logischer Verbindungen; Synchronisation des Datenaustausches	ISO 8326	ISO 8327 ISO 9548
6	Presentation (Darstellung)	Systemunabhängige Bereitstellung/Darstellung der Daten ermöglicht den Austausch von Daten zwischen unterschiedlichen Systemen; Datenkompression und Verschlüsselung	ISO 8823	ASN.1 (ISO 8824)
7	Application Layer (Anwendung)	Austausch der Nutzdaten spezieller Anwendungen (z. B. E-Mail-Client und -Server)	ISO 8649	HTTP, SMTP, FTP, CANopen

Für die in diesem Buch beschriebenen Netzwerke und Softwarelösungen genügt das Verständnis der Layer 1 und 2 des ISO/OSI-Schichtenmodells. Für TCP/IP Kommunikation sei daher auf die Literatur verwiesen.

5.1.1 Schicht 1: Physikalische Bitübertragung

Im folgenden Abschnitt werden einige grundlegende Überlegungen zur physikalischen Datenübertragung auf Leitungen angestellt, soweit dies zum Verständnis der beschriebe-

nen Schnittstellen und Sensoren notwendig ist. Auf eine umfassende Beschreibung der Bussystemtechnik wurde verzichtet, hier ist es gegebenenfalls notwendig, die am Ende des Kapitels beschriebene weitere Literatur zu studieren.

5.1.1.1 Leitungen

Elektrische Leitungen übertragen Energie über die sie umgebenden elektrischen und magnetischen Felder. Die Übertragung basiert auf dem Prinzip, dass ein sich änderndes elektrisches Feld ein Magnetfeld in seiner Umgebung erzeugt, das wiederum ein elektrisches Feld erzeugt usw. Folge ist, dass sich eine elektromagnetische Wellenfront ausbreitet, analog zu einer akustischen Wellenfront in einem Medium.

Da die Ausbreitungsgeschwindigkeit c endlich ist, benötigt das Signal eine gewisse Zeit, bis es vom Sender zum Empfänger gelangt. c hängt dabei vom umgebenden Medium ab, im Vakuum ist die Lichtgeschwindigkeit $c = c_0 \approx 3 \cdot 10^8$ m/s, bei realen Leitungen wird sie hauptsächlich vom verwendeten Isolierstoff bestimmt und liegt etwa bei $c = 0,6 c_0$. Damit stellt eine Leitung ein Speicherglied für Energie und für Information dar. Bei einer 5000 km langen Leitung mit $c = 2 \cdot 10^8$ m/s beträgt die Laufzeit beispielsweise 25 ms, was bedeutet, dass bei einer Datenübertragungsrate von einem Mbit/s bereits 25.000 Bit in der Leitung gespeichert sind. Bei einer vergleichbaren 10 m langen Leitung beträgt die Laufzeit 50 ns. Die absolute Länge einer Leitung spielt weniger eine Rolle als das Verhältnis zwischen Länge und Datenrate, mit anderen Worten, auch eine kurze Leitung ist für die Übertragung lang, wenn nur die Datenrate hoch genug ist.

Durch Leitungsverluste und Polarisationseffekte im Isolierstoff wird jedoch auch Energie dissipiert, so dass eine reale Leitung in erster Näherung ein Tiefpassfilter aus einem Serienwiderstand und einer Parallelkapazität darstellt⁵. Dadurch wird das Signal verschliffen, so dass bei großen Leitungslängen eine Signalregenerierung notwendig ist. Dieser ist bei den hier beschriebenen Anwendungen in der Regel noch nicht relevant, muss bei größeren Entfernungen zu den Sensoren aber berücksichtigt werden.

Neben der Speicherung und Dämpfung hat eine Leitung jedoch auch die Eigenschaft, Energie abzustrahlen und Energie aus der Umgebung der Leitung zu absorbieren. Dies führt zum Nebensprechen, engl. *crosstalk*, wenn mehrere Leitungen parallel nebeneinander geführt werden, oder zur Einstreuung von Störimpulsen, wenn Leitungen direkt neben Verbrauchern geführt werden, die von hohen dynamischen Strömen durchflossen werden (elektrische Antriebe, Schaltaktoren). Andererseits können Leitungen mit hohen Datenraten wie Antennen wirken, die andere Verbraucher direkt beeinflussen. Dies zu verhindern ist Aufgabe der elektromagnetischen Verträglichkeit, EMV. Aus Platzgründen sei hier auf die Literatur verwiesen [2]. Neben dem Verdrillen und Schirmen von Leitungen und der Schirmung von Gehäusen gehen EMV-Anforderungen direkt in die Spezifikation der Bus-systeme ein. Auf der Platine ist unter anderem auf ausreichende Masseflächen zu achten, sowie darauf, Signal- und (geschaltete) Stromleitungen nicht parallel zu führen, bzw. über

⁵ Dies gilt für Leitungen, bei denen die Laufzeit kurz gegenüber der Anstiegszeit der Signale ist, andernfalls bildet die Leitung eine grenzwertig infinitesimale Kette von Tiefpässen.

Masseflächen gegeneinander abzuschirmen. Dies gilt auch für das Nebeneinander von hoch getakteten Signalleitungen.

Auf einen speziellen Punkt sei hier jedoch eingegangen: Das Zusammenspiel zwischen elektrischen und magnetischen Feldern um die Leitung legt ein konstantes, materialabhängiges Verhältnis zwischen Spannung und Strom an jedem Punkt der Leitung fest. Dieses Verhältnis heißt Wellenwiderstand Z_L . Er ist unabhängig von jeder Beschaltung an den Leitungsenden, da eine auf der Leitung entlangwandernde Wellenfront diese Enden nicht „sieht“.

Es ist nun interessant zu überlegen, wie sich eine hinlaufende Welle der Amplitude U_{2h} bzw. I_{2h} verhält, die das Leitungsende erreicht und dort auf einen Verbraucherwiderstand Z_V trifft. Ist die Leitung am Ende mit der Impedanz $Z_V = Z_L$ belastet (abgeschlossen), dann entspricht das Spannungs-/Stromverhältnis am Ende genau dem auf der Leitung. Die Folge: Die gesamte Energie kann absorbiert werden. Man spricht dann von Anpassung. Ist die Leitung dagegen kurzgeschlossen, wird eine Spannung 0 erzwungen. Dies führt zu einem Ausgleichsvorgang zwischen der von Null verschiedenen Spannung auf der Leitung und der Spannung 0 am Leitungsende, der sich als rücklaufende Welle der Amplitude U_{2r} bzw. I_{2r} zum Leitungsanfang hin fortsetzt. Bei am Ende offenen Leitungen wird dagegen ein Strom 0 erzwungen und auch dies führt zu einer rücklaufenden Welle. Zwischen diesen Extremen wird es zu einer Teilabsorption der Welle durch den Verbraucher und einer Teilreflexion in der Leitung kommen.

Das Verhältnis von hinlaufender zu rücklaufender Welle wird über den Reflexionsfaktor r mit

$$U_{2r} = r \cdot U_{2h} \quad (5.1)$$

$$I_{2r} = -r \cdot I_{2h} \quad (5.2)$$

angegeben, der sich wie folgt berechnet:

$$r = \frac{Z_V - Z_L}{Z_V + Z_L} \quad (5.3)$$

Abb. 5.3 zeigt die Messung eines Impulses von 320 ns Dauer bei abgeschlossenem Leitungsanfang auf einem 100 m langen Koaxialkabel mit einem Wellenwiderstand von 50 Ω . Die obere Kurve in den Abbildungen zeigt den Impuls am Leitungsende, der offensichtlich 500 ns verzögert eintrifft. Gut zu sehen ist die Abflachung des Impulses durch die Tiefpasseigenschaft der Leitung. Nach weiteren 500 ns erreicht der reflektierte Impuls wieder den Leitungsanfang (untere Kurve). Im Fall der offenen Leitung (Abb. 5.3a) wird er offensichtlich positiv reflektiert, im Fall der kurzgeschlossenen Leitung negativ. Die weitere Verflachung zeigt den erneuten Durchlauf durch das Tiefpasssystem. Ist auch der Leitungsanfang nicht richtig abgeschlossen, wandert der Impuls zwischen den Leitungsenden hin und her, wobei er sich immer mehr verschleift und schließlich verschwindet.

Es ist jedoch leicht einzusehen, dass reflektierte Bits auf einer Leitung zumindest in einer frühen Phase wie „echte“ Bits aussehen und deshalb Übertragungsfehler auftreten können. Daher ist ein korrekter Leitungsabschluss auch bei kurzen Leitungen extrem

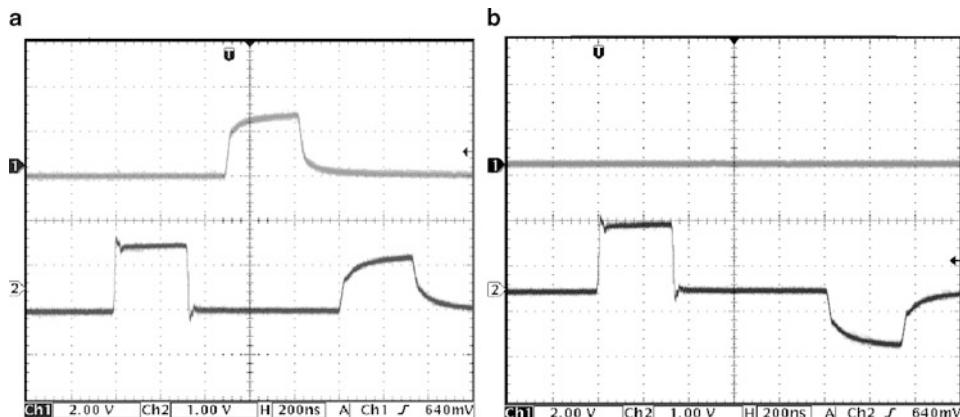


Abb. 5.3 Impuls von 320 ns Dauer an einer 100 m langen Koaxialleitung (50Ω); **a** offenes Ende (jeweils *oberes Signal*), **b** Kurzschluss am Ende. Kanal 1 (*oben*) ist jeweils am Leitungsende gemessen, Kanal 2 (*unten*) am Leitungsanfang

wichtig. In der Regel sind die Abschlusswiderstände rein ohmsch, bestehen bisweilen aber auch aus einem R-C-Glied.

5.1.1.2 Transceiver

Der Begriff Transceiver ist ein Kunstwort, zusammengesetzt aus dem Englischen Transmitter (Sender) und Receiver (Empfänger). Seine Aufgabe ist auf der Senderseite die Signalformung bzw. Signalverstärkung und auf der Empfängerseite die Messung und damit Erkennung des Signals. Dabei spielen folgende Überlegungen eine Rolle:

- Übertragungsart: Die Übertragungsart entscheidet, wie zwei Busteilnehmer kommunizieren können. Ist einer der Busteilnehmer nur Sender, der andere nur Empfänger, spricht man vom Simplex- oder Richtungs-Betrieb. Kann die Kommunikation in bei-

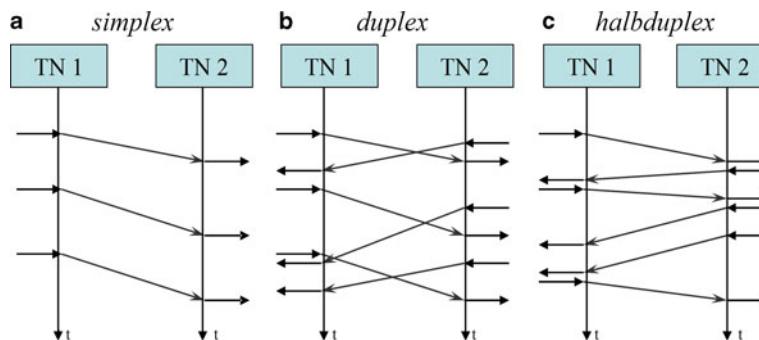


Abb. 5.4 Übertragungsarten Simplex, Vollduplex und Halbduplex

de Richtungen verlaufen, spricht man von Duplex-Betrieb, wobei zwischen Voll duplex und Halbduplex unterschieden wird. Abb. 5.4 verdeutlicht den Zusammenhang anhand eines vereinfachten Sequenzdiagramms (siehe auch Abb. 7.10). Beim Voll duplexbetrieb ist gleichzeitiges Senden und Empfangen möglich, in der Regel durch zwei Signalleitungen (Tx für Senden, Rx für Empfangen) oder durch zwei andere Übertragungskanäle. Beim Halbduplexbetrieb kann nur entweder gesendet oder empfangen werden.

- Verbindungsart Point-to-Point oder Multipoint: Viele Bussysteme (Ethernet, CAN, LIN, I²C) lassen einen gleichzeitigen Zugriff auf das Übertragungsmedium zu, das heißt, die Kommunikation findet nicht nur zwischen einem Sender und einem Empfänger (Point-to-Point), sondern zwischen mehreren Sendern und mehreren Empfängern (Knoten) statt. Elektrisch bedeutet dies, dass die Transceiver sicherstellen müssen, dass gleichzeitige Sendeversuche zweier Busteilnehmer keine elektrische Beschädigung (Kurzschluss) zur Folge haben. Hierbei hat sich eine Schaltung durchgesetzt, die je nach Betrachtungsweise wired-AND oder wired-OR genannt wird: Über einen Pullup-Widerstand wird der Bus kontinuierlich auf dem High-Pegel gehalten. Jeder Teilnehmer ist mit dem offenen Kollektor seines Ausgangstransistors an den Bus geschlossen (Open-Collector). Sendet irgendein Teilnehmer ein Low-Signal (0 V), so liegt der Buspegel insgesamt auf Low. Deshalb wird der Low-Pegel als *dominant* bezeichnet, der High-Pegel als *rezessiv*. Daher die Bezeichnung wired-AND (Abb. 5.5), da eine 1 nur anliegen kann, wenn alle Teilnehmer eine 1 senden. Da die logische 0 am Kollektor anliegt, wenn an der Basis des Transistors eine 1 anliegt, wird die Schaltung manchmal auch wired-OR genannt (negative Logik).
- Jeder Knoten kann bei dieser Schaltung über seine Empfängerseite den Buspegel mitlesen und bei einer Abweichung zwischen einem eigenen gesendeten High-Pegel und einem gemessenen Low-Pegel feststellen, dass ein anderer Busteilnehmer ebenfalls

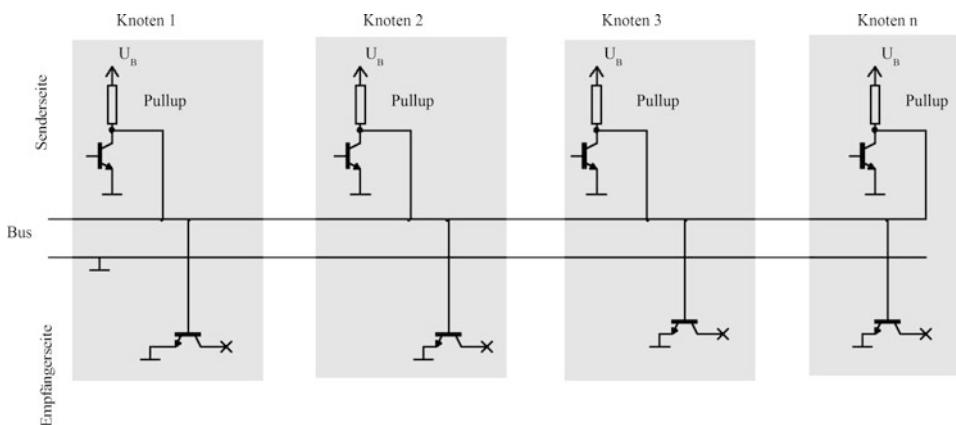


Abb. 5.5 Wired-AND-Schaltung (Open Collector)

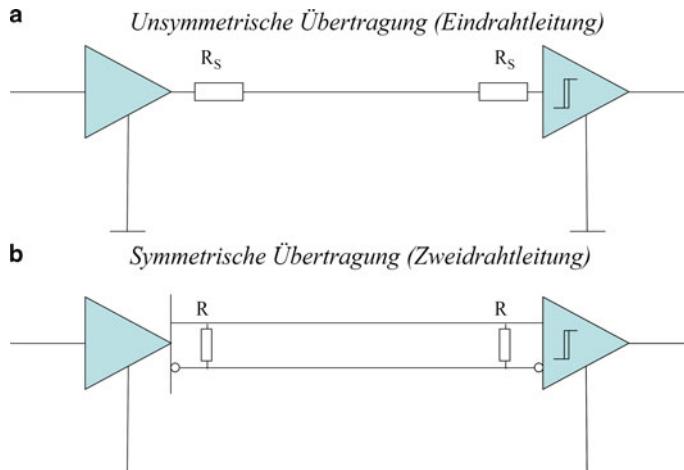


Abb. 5.6 Unsymmetrische und symmetrische Übertragung

einen Zugriffsversuch unternimmt. Eine Konsequenz dieser Kollisionsmöglichkeit ist die Forderung nach einer Zugriffssteuerung, die später beschrieben wird.

- Leitungsart Eindrahtleitung oder Zweidrahtleitung: Je nach Zuverlässigkeitssanforderung werden die Signalleitungen als Ein- oder Zweidrahtleitungen ausgeführt. Die Eindrahtleitung (unsymmetrische Schnittstelle) (Abb. 5.6a) ist die kostengünstigere Variante. Der Signalpegel wird gegen die allgemeine Masse angelegt. Dies macht die Eindrahtleitung anfällig gegenüber Störungen durch Einstreuung⁶ oder durch Masseanhebung. Sie wird daher nur in Bussystemen mit niedrigen Datenraten (z. B. LIN) eingesetzt. Bei der Zweidrahtleitung wird auf zwei Adern ein jeweils invertiertes Signal übertragen. Ein Differenzverstärker ermittelt aus der Differenz zwischen den Spannungspeichern den Signalpegel unabhängig vom Potenzial gegenüber der Fahrzeugmasse, weswegen man von einer differentiellen Übertragung spricht (Beispiel: CAN, RS485). Durch Verdrillen der Leitungen wird diese an sich bereits robuste Verkabelung noch besser gegen Einstreuungen geschützt, weil sich diese in den kleinen Schleifen, die die Verdrillungen bilden, gegenseitig aufheben.

5.1.1.3 Übertragungssicherheit

Der Transceiver ist letztlich für die sichere und fehlerarme Übertragung des Signals verantwortlich. Auf der Senderseite geschieht dies durch eine übertragungsmediengerechte Pulsformung. Nicht immer ist ein steilflankiger Rechteckimpuls eine vorteilhafte Lösung, denn durch die Tiefpasseigenschaft der Leitung wird er verschmiert und läuft nach. Wird eine Impulsfolge übertragen, überlagern sich die Nachläufer der Rechteckimpulse und können zu *Intersymbolinterferenz* führen [3], das heißt zur gegenseitigen Beeinflussung

⁶ Hin- und Rückleiter bilden eine Schleife mit sehr großer Schleifenfläche.

bis zur Unkenntlichmachung der Impulse. Eine wirksame Möglichkeit dies zu verhindern, ist, die Impulse so zu formen, dass ihr Spektrum keine signifikanten Frequenzanteile jenseits der durch die Bandbreite des Kanals gegebenen Grenzfrequenzen (Nyquist-Kriterium) enthält. Hierfür werden in der Regel so genannte Kosinus-Roll-Off-Impulse verwendet, die von einem gleichnamigen Filter aus den ursprünglichen Rechteckimpulsen erzeugt werden.

Auf der Empfängerseite besteht die Schwierigkeit darin, zu einem gegebenen Messzeitpunkt zu entscheiden, ob ein High-Pegel oder ein Low-Pegel anliegt. Ein von Rauschen überlagertes oder mit Jitter (Abschn. 5.1.2) behaftetes Digitalsignal erschwert zum einen die Wahl des Messzeitpunktes aber auch die Entscheidung selbst. Die Wahrscheinlichkeit, dass eine „1“ erkannt wurde, wenn eine „0“ gesendet wurde, und umgekehrt, lässt sich aus der Normalverteilung berechnen und heißt Restfehlerwahrscheinlichkeit. Hierzu sei auf die Literatur verwiesen.

5.1.1.4 Netzwerktopologien

Ein Netzwerk mit mehreren Knoten kann nach dem Aufbauprinzip klassifiziert werden. Man spricht von Topologien. In lokalen Netzwerken (LAN = Local Area Network) haben sich die folgenden Topologien (Abb. 5.7) durchgesetzt.

- *Bus*: Der Bus, (auch Linie oder Line-Bus) ist die häufigste Topologie, vertreten z. B. durch CAN und I²C, die auf dem Point-to-Multipoint-Zugriff beruht. Alle Knoten senden und empfangen über dasselbe Medium, so dass eine Zugriffssteuerung notwendig wird. Bei Wegfall eines Knotens steht der Bus in der Regel den anderen Knoten weiterhin zur Verfügung, bei einem Leitungsbruch teilt sich der Bus in zwei Teilbusse auf.
- *Ring*: Der Ring besteht aus einer geschlossenen Folge von Point-to-Point-Verbindungen. In aller Regel reichen die Knoten ihre Daten synchron im Ring herum, eine Nachricht an den Vorgänger eines Knotens muss fast den gesamten Ring passieren, bevor sie an den gewünschten Empfänger durchgestellt wird. Da das Signal in jedem Knoten regeneriert wird, sind Ringtopologien robust, allerdings führt ein Leitungsbruch zum Totalausfall des Systems. Die Ring-Topologie wird im MOST-Netzwerk verwendet, das in [1] beschrieben ist.
- *Stern*: Bei der Stern topologie, wie sie in modernen Büro-LANs verwendet wird, kommunizieren alle Teilnehmer über eine Point-to-Point-Verbindung mit einem Sternpunkt. Ein passiver Stern verteilt dabei nur das gegebenenfalls verstärkte Signal (Repeater), somit ist diese Topologie logisch mit dem Bus vergleichbar. Beim aktiven Stern übernimmt der Sternpunkt eine Masterrolle oder selektiert zumindest die Datenströme nach Adressaten.
- *Mesh*: Bei der Mesh- oder Netzt topologie sind die einzelnen Knoten mit mehreren, im Extremfall mit allen Knoten im Netz direkt verbunden. Ein Beispiel findet sich in Abschn. 5.4 bei ZigBee
- *Hierarchische Topologien*: Hierarchische Topologien sind solche, in denen einzelne Netzketten den Ausgangspunkt für Subnetze darstellen (z. B. als Gateway). Man kann

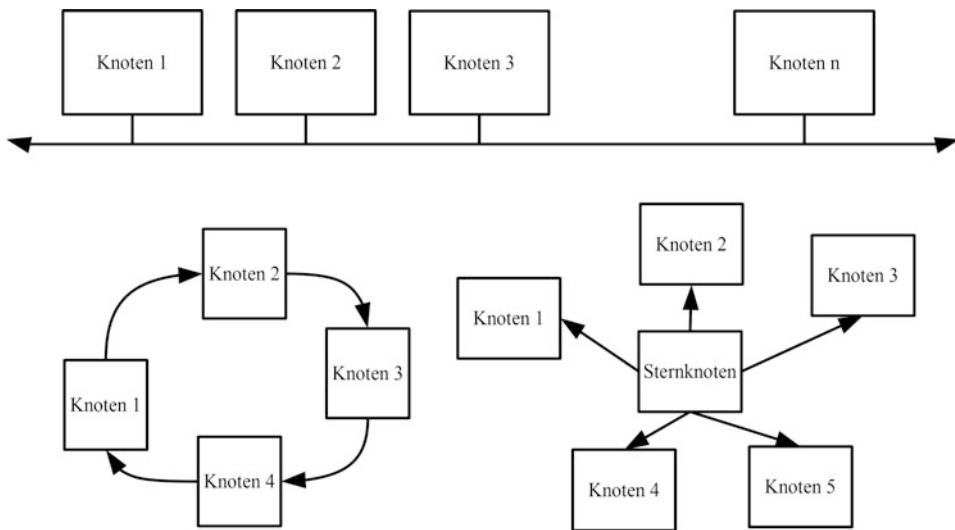


Abb. 5.7 Bus, Ring und Stern-Topologie

diese Strukturen als Baum-Topologie darstellen. Ihr Vorteil liegt darin, dass untergeordnete Netzwerke lokal autonom arbeiten können und im übergeordneten Netzwerk nur die Systemsteuerung erfolgt. Dieses Prinzip aus der Automatisierungstechnik findet im Fahrzeug im Bereich lokaler Subnetze (LIN) und in den Diagnoseschnittstellen Anwendung.

5.1.1.5 Synchronisation und Leitungscodierung

Die Leitungscodierung hat die Aufgabe, die physikalische Signaldarstellung auf dem Übertragungsmedium festzulegen (siehe Abb. 5.8). Den binären Bitsequenzen werden Pegelfolgen, z. B. Spannungen, Ströme oder Lichtintensitäten, zugeordnet. Dafür sind verschiedene Anforderungen relevant:

- **Synchronisation und Taktrückgewinnung:** Der Empfang der Signale ist ein periodischer Messvorgang, der zu einem festgelegten Zeitpunkt startet und mit einer festgelegten Periodendauer (Schrittweite) abläuft. Sender und Empfänger müssen dabei synchronisiert werden, d. h. den Wechsel zum jeweils nächsten übertragenen Zeichen nahezu gleichzeitig durchführen. Dies gelingt, indem entweder über ein zusätzliches Medium (zusätzliche Leitung) ein Taktsignal übertragen wird (wie beim I²S- oder I²C-Bus) oder indem die Leitungscodierung so geschickt gewählt wird, dass der Takt im Signal vorhanden ist und aus diesem zurückgewonnen werden kann. Lediglich bei asynchronen Übertragungsverfahren, wie sie in UART basierten Netzen, beispielsweise der RS232 oder RS485 Schnittstelle oder im LIN-Bus verwendet werden, arbeiten Sender und Empfänger mit freischwingenden Oszillatoren. Hier müssen zusätzliche Maßnahmen

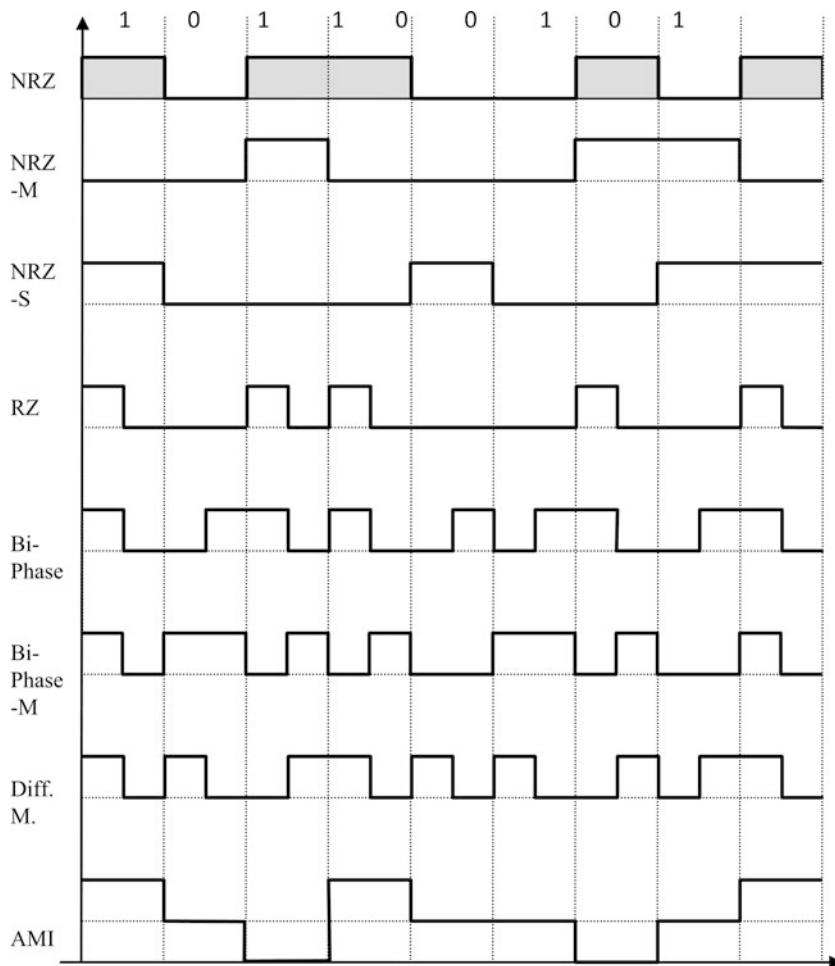


Abb. 5.8 Leitungscodierung

getroffen werden, damit die Taktabweichungen nicht über Bitgrenzen hinauslaufen und damit eine Fehlmessung eintritt.

- **Gleichspannungsfreiheit:** Wenn das Datenübertragungssystem integrierendes Verhalten (Tiefpass) aufweist, muss der Gleichspannungsanteil des Signals im Mittel 0 sein, da sich sonst das System „auflädt“.
- Unterscheidung zwischen „Signal 0“ und „kein Signal“: Die Frage ist zu klären, wie festgestellt werden kann, ob gerade eine Folge von Nullen übertragen wird oder einfach gar nichts.

Im einfachsten Fall wird eine logische 1 durch einen Spannungspiegel V_1 (High) dargestellt, eine logische 0 durch einen Spannungspiegel V_2 , der im Spezialfall Low oder

0 V (unipolares Signal) oder $-V_1$ (bipolares Signal) ist. Dieses Format wird NRZ (Non Return to Zero) genannt, weil die Signalamplitude im Pegel des letzten Bits verharrt. Längere Eins- oder Nullfolgen werden damit als Gleichspannung dargestellt. Weitere Formate umgehen das Problem teilweise:

- NRZ-M und NRZ-S: Hierbei wird nicht der Signalpegel, sondern die Pegeländerung zur Darstellung der Zeichen genutzt. Beim NRZ-M(ark) bedeutet eine Pegeländerung eine logische 1, keine Pegeländerung eine logische 0. Beim NRZ-S(pace) ist dies genau umgekehrt. Mark steht für die 1, Space für die 0. In Unterscheidung dazu wird das NRZ-Format auch als NRZ-L(evel) bezeichnet.
- RZ: Beim Return to Zero Format kehrt der Signalpegel in der Mitte der Taktperiode zu Null zurück (bei der logischen 1). Zur Darstellung des Zeichens wird also ein Rechteckimpuls benutzt, dessen Breite gleich der halben Taktperiodendauer gewählt wird.
- Bi-Phase oder Manchester: Bezeichnungen für dasselbe Format. Nullen und Einsen werden durch zwei verschiedene Impulse dargestellt. Eine 1 wird durch einen 1-0-Sprung in der Mitte des Taktintervalls dargestellt, eine 0 durch einen 0-1-Sprung. Eine andere Interpretation ist, dass die 1 durch einen Rechteckimpuls in der ersten Hälfte des Taktintervalls dargestellt wird, während die 0 durch einen Rechteckimpuls in der zweiten Hälfte des Taktintervalls dargestellt wird. Damit ist sichergestellt, dass mindestens einmal pro Takt ein Pegelwechsel auftritt. Bei der Variante Bi-Phase-M(ark)-Format ist sichergestellt, dass zu jedem Taktbeginn ein Pegelwechsel auftritt. Eine 1 wird durch einen weiteren Pegelwechsel in der Mitte des Taktintervalls dargestellt, eine 0 durch Ausbleiben des Pegelwechsels. Dieses Format wird z. B. beim Ethernet oder beim MOST-Bus eingesetzt und erlaubt eine einfache Taktrückgewinnung auch bei langen 0- oder 1-Folgen. Invertierung der Ursprungsfolge führt zum Bi-Phase-Space-Format. Eine Alternative ist das Differential Manchester Format: Hier findet der Pegelwechsel immer in der Mitte des Signalintervalls statt, zusätzlich stellt ein Wechsel zu Beginn des Signalintervalls eine 0 dar. Kein Wechsel zu Beginn des Signalintervalls zeigt die 1.

Da mit jedem Übertragungsschritt genau ein Bit (also zwei Zustände) übertragen werden kann, heißen die oben genannten Formate auch binäre Formate. Unter vielen weiteren Formaten sei noch das AMI-Format (Alternate Mark Inversion) erwähnt, ein Pseudoternärformat. Ternär heißt, dass das Signal drei Pegel annehmen kann. Da aber die Pegel +1 und -1 jeweils für eine 1 stehen, ist das Format eben doch nicht wirklich ternär. Jede 1 wird durch eine invertierte Polarität des 1-Pegels dargestellt, die 0 durch einen 0-Pegel. Dadurch ist für einen Pegelwechsel auch bei langen Einsfolgen gesorgt. Dies hat auch zur Folge, dass das Signal gleichspannungsfrei ist (Anwendung: ISDN Basisanschluss).

Auch längere Nullfolgen werden in Varianten des AMI-Formats berücksichtigt. Dazu zählen u. a. HDB-n Codes und der BnZS-Code. Das Grundprinzip besteht darin, dass längere Nullfolgen durch eine ternäre Zeichenfolge ersetzt werden, die das Prinzip des AMI-Formats gezielt verletzt.

Selbstverständlich kann man auch noch mehr Signalpegel zur Übertragung verwenden, z. B. vier Pegel (Beispiel: -2 V , -1 V , 1 V , 2 V), also ein quaternäres Format. Da damit $\log_2 4 = 2$ Bit (also vier Zustände) übertragen werden können, verdoppelt sich die „gefühlte“ Datenrate und wir unterscheiden nunmehr zwischen der Schrittgeschwindigkeit v_s (Einheit: Baud, d. h. Signalwechsel bzw. Messschritte pro Sekunde) und der Datenrate in Bit/s. In realen Verfahren werden meistens vier, acht oder sechzehn Amplitudenstufen gewählt, entsprechend zwei, drei oder vier Bit, die gleichzeitig übertragen werden.

Nyquist hatte herausgefunden, dass die maximale Kanalkapazität des Übertragungsmediums c_c in Bit/s

$$c_c = 2 \cdot B \quad (5.4)$$

sein kann, wobei B die Bandbreite des Kanals in Hz ist. Dies kann man sich damit veranschaulichen, dass pro Halbwelle einer Sinusschwingung ein Bit übertragen werden kann. Kommen jetzt noch N unterschiedliche Amplitudenstufen hinzu, so wächst die Kanalkapazität nach den oben angestellten Überlegungen auf

$$c_c = 2 \cdot B \log_2 N \quad (5.5)$$

Leider werden mit einer wachsenden Zahl von Pegelstufen auch die Einflüsse eines immer gleichbleibenden Rauschpegels höher, so dass nach Claude Shannon die maximale Kanalkapazität des Übertragungsmediums c_c auf

$$c_c = 2 \cdot B \log_2 (1 + \text{SNR}) \quad (5.6)$$

begrenzt bleibt, worin SNR der Signal-Rauschabstand, also das Amplitudenverhältnis zwischen mittlerer Signalleistung und mittlerer Rauschleistung, ist.

An dieser Stelle sei angemerkt, dass die Leitungscodierung nur einen Teil der Signaldarstellung auf dem Medium ausmacht. Genauso interessant ist die Frage, wie der Spannungspegel auf dem Medium selbst dargestellt wird (Modulation). Die direkte Abbildung des Signalpegels auf den Spannungspegel auf der Leitung wird als Basisbandmodulation bezeichnet. Diese findet ihre Grenze spätestens dann, wenn keine Leitung mehr zur Verfügung steht, sondern über eine Luftschnittstelle übertragen werden muss. Hier wird in der Regel ein hochfrequentes Trägersignal mit dem Nutzsignal moduliert. Im einfachsten Fall schwankt die Amplitude des Trägersignals mit dem Signalpegel des Nutzsignals (Amplitudenmodulation oder Amplitude shift keying ASK), indem das Trägersignal mit dem Nutzsignal multipliziert wird. Alternativ können die unterschiedlichen Signalpegel als unterschiedliche Frequenzen dargestellt werden (Frequenzmodulation oder Frequency shift keying, FSK) oder aber die Phasenlage oder mehrere dieser Kenngrößen des Trägersignals werden in Abhängigkeit vom Nutzsignal beeinflusst.

5.1.2 Schicht 2: Sicherungsschicht

Bitübertragungsschicht und Sicherungsschicht bilden zusammen die Grundlage für die Daten-Übertragung selbst bei der einfachsten Point-to-Point Kommunikation. Wenn man über Netzwerktechnologien redet, werden diese beiden Schichten meist in einem Atemzug genannt. Das bekannteste Beispiel ist die IEEE 802 Familie, die u. a. für übliche WLANs und Ethernet die beiden unteren Schichten spezifiziert. Im Automobilbereich ist der Basis-Standard von CAN (ISO 11898 bzw. Bosch Spezifikation) ebenfalls auf diese beiden Schichten beschränkt. Auf diesen Protokollen sitzen dann Vermittlungs- und/oder Transportprotokolle auf (z. B. TCP/IP oder CAN-TP). Da die drei Hauptaufgaben der Sicherungsschicht: Rahmenbildung, Fehlerbehandlung und Medienzugriff weitgehend autonom gehandhabt werden können, unterteilt man die Sicherungsschicht gerne in die drei Subschichten:

- Framing Sublayer für die Rahmenbildung
- Media Access Control (MAC) Sublayer für die Zugriffssteuerung auf den Physical Layer
- Logical Link Control (LLC) für die Kommunikationssteuerung und gegebenenfalls die Fehlerbehandlung

5.1.2.1 Rahmenbildung

Aufgabe der Rahmenbildung ist es, zum einen dem Empfänger mitzuteilen, wo eine Nachrichteneinheit beginnt und wo sie endet, zum anderen, um Protokollinformation an ein Nutzdatenpaket zu hängen. Rahmen werden in der Regel an den Anfang eines Datenpaketes gehängt (Header), in diesem Fall ist die Paketlänge fix oder im Header angegeben. In einigen Protokollen werden auch am Ende angehängte Rahmenbestandteile (Trailer) verwendet, um z. B. Prüfsummen zu transportieren, die während der Datenübertragung gebildet wurden. Grundsätzlich unterscheidet man

- Zeichenbasierte Übertragung (asynchrone Übertragung, Abschn. 5.2.1): Hier wird vom Sender jeweils nur ein Byte übertragen, der Beginn der Übertragung wird in NRZ-Codierung durch ein Startbit eingeleitet, das den Buspegel vom Ruhezustand (High) in den aktiven Zustand (Low) holt und damit den Beginn der Übertragung kennzeichnet. Damit der Bus nach Ende der Übertragung wieder im High-Zustand ist, werden ein oder zwei Stopppbits (High) angehängt. Da der Bittaktoszillator des Empfängers (siehe oben) nach jedem Byte neu synchronisiert werden kann (Zeichensynchronisation), muss kein Takt übertragen werden und die Oszillatoren können mit relativer grober Toleranz von ca. 5 % aufgebaut werden, z. B. als RC-Oszillator.
- Bitstrombasierte Übertragung: Bei diesem Übertragungsverfahren wird ohne weitere Zeichensynchronisation ein Datenpaket bestehend aus Rahmen und Nutzdaten übertragen. Die Bitsynchronisation erfolgt in der Regel durch ein geeignetes Formatierungsverfahren und eine Taktrückgewinnung. Wird dennoch eine NRZ-Codierung verwendet

(z. B. bei CAN), muss sichergestellt werden, dass regelmäßige Pegelwechsel vorliegen, auch wenn lange 0- oder 1-Folgen gesendet werden.

Bei CAN geschieht dies mit Hilfe des Bit-Stuffing-Verfahrens (Abb. 5.9). Sobald der Sender mehr als fünf aufeinander folgende Bit gleichen Wertes senden soll, schiebt er ein Bit des komplementären Wertes in den Bitstrom ein [4]. Der Empfänger kann ebenso, wenn er fünf aufeinanderfolgende Bit desselben Wertes empfangen hat, das Folgebit verwerfen, wenn es den komplementären Wert hat, oder einen Empfangsfehler melden, wenn es denselben Wert hat. Bei der bitstrombasierten Übertragung werden Datenpakete von 8 (CAN) bis 1500 Byte (Ethernet) ohne weiteren Overhead außer dem Protokoll-overhead übertragen. Interessant für dieses Übertragungsverfahren ist die Frage, wie der Anfang eines neuen Rahmens erkannt wird. In vielen Protokollen ist dies so gelöst, dass als erstes Zeichen ein Sonderzeichen gesendet wird, das sonst nirgendwo in der Botschaft vorkommen darf.

Dies läuft der Forderung zuwider, den gesamten darstellbaren Coderaum für Nutzdaten zur Verfügung zu haben (Codetransparenz). Gelöst wird das Dilemma beispielsweise durch *Bytestuffing* oder *Zeichenstopfen* (Abb. 5.10). Kommt das Rahmenanfangszeichen im Code vor, muss es durch ein Escape-Zeichen eingeleitet werden, das zu diesem Zweck in die Nutzdaten eingebracht wird. Das Escape-Zeichen selbst muss natürlich ebenfalls mit einem Escape-Zeichen eingeleitet werden, wenn es als gültiges Nutzdatenzeichen kommt. Alternativ können als Rahmenbegrenzer in einem bitstrombasierten System auch Zeichen gesendet werden, die z. B. die Bit-Stuffing-Regel verletzen und deshalb in den Nutzdaten nicht vorkommen können.

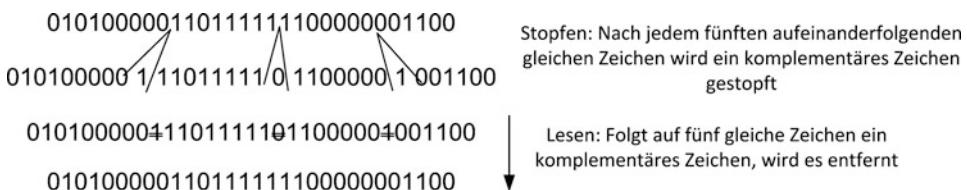


Abb. 5.9 Bitstuffing

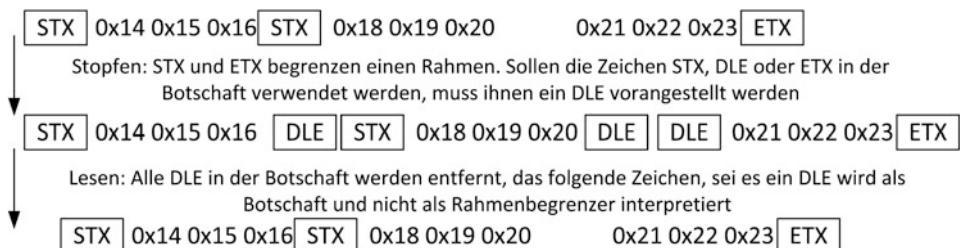


Abb. 5.10 Bytestuffing

5.1.2.2 Medienzugriff – MAC

Die Kommunikation zwischen mehreren Teilnehmern über ein Medium führt zwangsläufig zu Kollisionen, wenn die Teilnehmer untereinander keine Information über den Sendewunsch eines anderen Teilnehmers haben als die, die ihnen auf dem Bus zur Verfügung steht. Eine Kollision wird erkannt, wenn ein Busteilnehmer ein anderes Signal auf dem Bus misst, als er soeben gesendet hatte, sofern das Bit rezessiv war und ein anderer Teilnehmer gleichzeitig ein dominantes Bit sendet. In diesem Fall müssen alle Teilnehmer ihren Rahmen verwerfen und neu beginnen. Netzwerkprotokolle können Kollisionen per se vermeiden oder aber eine geeignete Strategie für die Reaktion auf Kollisionen mitbringen.

Bei kollisionsvermeidenden Protokollen kann es aufgrund der Sendesteuerung keine Kollisionen geben:

- Master-Slave-Protokolle: Beispielsweise beim LIN-Bus existiert nur ein Busteilnehmer (Master), der die Datenübertragung beginnen kann. Er fragt alle anderen Busteilnehmer (Slaves) regelmäßig nach einem vorgegebenen Plan auf Sendewünsche hin ab. Die Slaves senden nur nach Freigabe durch den Master, auch wenn sie Botschaften an andere Slaves zu verschicken haben.
- Token-Passing-Prinzip: Dieses eher selten genutzte Verfahren basiert auf einer speziellen Botschaft (Token), die demjenigen Busteilnehmer eine Sendeberechtigung einräumt, der im Besitz des Tokens ist. Hat er seine Sendung beendet, gibt er das Token (beispielsweise durch Hochzählen) an einen anderen Busteilnehmer weiter, so dass es reihum weitergereicht wird.
- TDMA, Time Division Multiple Access: Das TDMA-Verfahren basiert auf einer exakten Zeitsteuerung. In einem periodischen Zeitfenster, das durch eine Synchronisationsbotschaft gestartet wird, werden Zeitschlüsse exklusiv für einen Busteilnehmer (statisch) reserviert. Hat er keinen Sendewunsch, so bleibt der Zeitschlitz für diesen Kommunikationszyklus leer. Dieses Verfahren wird im Mobilfunkstandard GSM, im MOST-Protokoll und bei FlexRay eingesetzt und eignet sich immer dann, wenn isochrone Messwerte⁷ mit geringem Jitter übertragen werden müssen.

Andere Protokolle akzeptieren die Tatsache, dass Kollisionen prinzipiell möglich sind, und reagieren mit entsprechenden Strategien, um nach einer erkannten Kollision eine neue, geordnete Sendereihenfolge aufzubauen:

- ALOHA-Verfahren: Bei diesem auf Hawaii entwickelten Verfahren, sendet jeder Knoten, sobald ein Sendewunsch vorliegt. Eine Kollision führt zu einem gestörten Rahmen, der vom Empfänger detektiert und entsprechend quittiert wird. Wurde eine Kollision erkannt, unternehmen alle Busteilnehmer einen erneuten Sendeversuch nach einer be-

⁷ Isochron heißt, dass die Signale periodisch mit exakt gleichbleibender Periodendauer übertragen werden.

stimmten Zeit, die durch einen Zufallsgenerator ermittelt wird, um zu verhindern, dass der erneute Sendeversuch wiederum zu Kollisionen führt.

- CSMA/CD-Verfahren: Das ALOHA Verfahren ist nicht sehr effizient, daher wird es beispielsweise im Ethernet-LAN durch eine Busüberwachung ergänzt (CSMA steht für Carrier Sense Multiple Access), die verhindert, dass ein Sendeversuch unternommen wird, wenn bereits ein anderer Teilnehmer sendet. Dennoch kann es, bedingt durch Signallaufzeiten auf der Leitung, zu Kollisionen kommen, die der Sender selbst detektiert (CD steht für Collision Detection) und dann ebenfalls nach einer Zufallszeit mit einem erneuten Sendeversuch reagiert. Steigt die Zahl der Botschaften auf dem Medium, so verlängern die Teilnehmer den Rahmen für die Zufallszeit exponentiell (Exponential Backoff), so dass theoretisch nach einer beliebig langen Zeit jede Botschaft übertragen wird.
- Binärer Countdown oder CSMA/CA-Verfahren: Zuverlässigkeitssanforderungen beispielsweise im Kfz erlauben nicht, beliebig lange auf bestimmte Botschaften warten zu müssen. Daher wird in einem erweiterten Verfahren, das z. B. beim CAN-Bus zur Anwendung kommt, dafür gesorgt, dass in den Botschaftenheadern eine Priorität codiert ist. Im Fall einer Kollision hat der Teilnehmer mit der höheren Priorität das Recht, weiter zu senden, der Teilnehmer mit der niedrigeren Priorität unternimmt einen neuen Sendeversuch. Das Verfahren vermeidet also Kollisionen und heißt daher CSMA/CA für Collision avoidance (Abschn. 5.3.1).

5.1.2.3 Kommunikationssteuerung

Eine weitere Aufgabe der Sicherungsschicht ist die Kommunikationssteuerung. Prinzipiell unterscheidet man zwischen

- verbindungsorientierten Protokollen, also Protokollen, in denen ein Medium exklusiv für die Dauer der Kommunikation für die Partner reserviert ist⁸ (geschaltete Verbindung, engl. circuit switching), das „alte“ Einwahlverfahren beim Telefon arbeitet nach dem Prinzip und
- verbindungslosen Protokollen oder datagrammorientierten Protokollen, engl. packet switching, das heißt Protokollen, in denen jeder Kommunikationsvorgang kontextfrei und isoliert vonstattengeht und somit bei jedem Kommunikationsvorgang durch das Protokoll mitgeteilt wird, zwischen welchen Partnern kommuniziert werden soll.

In jedem Fall erfordert die Kommunikation zwischen potenziell mehreren unterschiedlichen Partnern die Information, an wen eine Botschaft gerichtet ist (also eine Adresse) und gegebenenfalls eine Information über den Absender, zumindest wenn eine Antwort erwartet wird. In Netzwerken unterscheidet man:

⁸ Damit ist die Verbindung kontextbehaftet und das Protokoll muss für die Bereitstellung des Kontextes sorgen, d. h. für den Verbindungsau- und -abbau.

- Instanzen- oder geräteorientierte Adressierung: Hier wird der Empfänger selbst adressiert, der ein Steuergerät oder eine Applikation in der Software eines Steuergeräts sein kann. Der Empfänger wertet exklusiv die Botschaft aus, während die anderen Busteilnehmer den Rest des Datenpakets verwerfen (Beispiel: Ethernet).
- Botschaftenorientierte Adressierung: Hier wird nicht ein Empfänger, sondern die Botschaft selbst gekennzeichnet. Alle Empfänger, die sich für die Botschaft interessieren, können sie auswerten. Dieses Verfahren wird insbesondere in Protokollen angewendet, in denen Informationen an mehrere Teilnehmer versendet werden müssen, z. B. im CAN (Multicasting). Die Adressen und die Priorität der Botschaften werden während der Systementwicklung festgelegt und in einer Tabelle festgehalten, in der zudem alle möglichen Empfänger vermerkt sind. Ändert man die Adresse oder das Format einer Botschaft, so müssen die Entwickler aller betroffenen Steuergeräte benachrichtigt werden. Der Integrationsaufwand für botschaftenorientierte Netzwerke ist dementsprechend hoch. Manche Systeme (z. B. LIN) haben konfigurierbare Schnittstellen, mit denen man die Adressen der Botschaften nachträglich ändern kann.

5.1.2.4 Zeitverhalten

Ein wichtiges Kriterium bei der Auswahl eines Bussystems ist das Zeitverhalten. Es wird maßgeblich von der Frage bestimmt, ob eine Nachricht mit einem determinierten Zeitverhalten gesendet wird, das heißt ob zu jedem Zeitpunkt klar ist, wann die Nachricht gesendet wurde und wie aktuell sie zu diesem Zeitpunkt war. Deterministische Netzwerke legen dies fest, bei nichtdeterministischen Netzwerken kann beispielsweise die Übermittlung wegen eines Prioritätskonflikts verzögert worden sein. Weiterhin sind zwei Größen für das Zeitverhalten entscheidend:

- Die *Latenzzeit*: Diese beschreibt die Zeit zwischen einem Ereignis (also dem Sendevorgang) und dem Eintreten der Reaktion (dem Empfangsvorgang). Sie setzt sich zusammen aus der Zeit für die Sendevorbereitung, der Wartezeit bis zur Freigabe des Mediums, der eigentlichen Übertragungszeit, die sich aus der Paketgröße mal der Übertragungsgeschwindigkeit (in Bit/s) ergibt, der Ausbreitungsgeschwindigkeit, die nur für sehr lange Leitungen oder Satellitenstrecken relevant ist, sowie der Zeit, die der Empfänger benötigt, die Botschaft zu dekodieren. Wird eine Antwort erwartet, so ist in der Regel die Roundtrip-Zeit interessanter, die etwa doppelt so lang wie die Latenzzeit ist, plus der Zeit, die der Empfänger benötigt um auf die Botschaft zu reagieren.
- Der *Jitter* ist ein Maß für die Abweichung der tatsächlichen Periodendauer von der vorgeschriebenen Periodendauer bei isochronen Botschaften. Er spielt eine Rolle, wenn Messwerte mit sehr zuverlässiger Wiederholrate übertragen werden müssen. Für Multimediadaten, die ebenfalls einen niedrigen Jitter benötigen, behilft man sich auf der Empfängerseite mit einem Puffer, der die empfangenen Datenpakete zwischenspeichert und mit geringem Jitter an die verarbeitende Instanz weitergibt.

5.1.2.5 Fehlerbehandlung

Als letzte Aufgabe der Sicherungsschicht sei hier die Fehlerbehandlung beschrieben. Werden Daten über einen Nachrichtenkanal versendet oder auch nur von einem Speichermedium gelesen⁹, besteht das Risiko von Übertragungsfehlern. Je nach Störung wird zwischen Einzelbitfehlern, Bündelfehlern (ganze Worte werden falsch übertragen) oder Synchronisationsfehlern (die gesamte Botschaft wird falsch gelesen) unterschieden. Je nachdem, wo der Fehler auftritt (Lokalisation) unterscheidet man *Nutzdatenfehler*, die die eigentliche Botschaft betreffen, und *Protokollfehler*, die die Protokollinformation betreffen und damit zu einer Fehlinterpretation der Botschaft führen können. Die Sicherungsschicht ist die erste Schicht, die mit Übertragungsfehlern konfrontiert wird. Ihre Aufgabe teilt sich in zwei Teile:

- Fehlererkennung (error detection) und
- Fehlerbehandlung (error correction)

Die Fehlererkennung beruht auf dem Prinzip der Redundanz. Die grundsätzliche Idee ist es, den Informationsgehalt eines einzelnen Zeichens oder einer Zeichengruppe innerhalb einer Botschaft gezielt zu verringern, indem der Botschaft zusätzliche Zeichen hinzugefügt werden, die keine neue Information enthalten. Man spricht dann von *Redundanz*.

Dahinter steht die Kommunikationstheorie¹⁰ bzw. die Codierungstheorie. Ein einfaches und bekanntes Beispiel für eine Redundanzerhöhung ist die Einführung eines Paritätsbits: Alle Einsen in einem Codewort werden gezählt. Wenn die Anzahl ungerade ist, wird eine Eins an das Codewort angehängt, andernfalls eine Null (oder umgekehrt). Damit ist sichergestellt, dass jedes Codewort eine gerade (im umgekehrten Fall: ungerade) Zahl von Einsen enthält. Abweichungen von dieser Regel werden als Fehler erkannt. Damit erhöht sich die Zahl der Stellen um ein Bit, was einer Verdoppelung der möglichen Codewörter gleichkommt. Da aber die Information dieselbe geblieben ist, ist die Information pro Bit gesunken und die Hälfte aller möglichen Codewörter ungültig. Dieses Verfahren wird bei einfachen Kommunikationssystemen angewandt, ist aber nicht sehr effizient. Ein zweiter Fehler im Codewort hebt die Fehlererkennung vollständig wieder auf.

Andere Verfahren bilden mehrstellige Prüfsummen aus größeren Nachrichtenblöcken oder gar der gesamten Botschaft mit dem Ziel, auch einen zweiten oder mehr Fehler erkennen zu können. Die Prüfsumme muss im Empfänger nachgebildet und mit der übermittelten Summe verglichen werden. Die am häufigsten eingesetzte, weil einfach durch Hardware abbildbare und sehr effiziente Form der Prüfsummenbildung ist der CRC-Code¹¹ (Polynomial Code, zyklischer Code).

⁹ Auch ein Speichermedium ist im weitesten Sinne ein Nachrichtenkanal mit hoher Latenzzeit.

¹⁰ Dieses mathematische Großbauwerk geht auf die Arbeit von Claude Shannon zurück, die dieser 1948 veröffentlicht hat.

¹¹ Cyclic redundancy check.

Für Übertragungsnetze, die nicht mit festen Codewortlängen arbeiten, existieren Faltungscodierer, die aus einem theoretisch unendlich langen Bitstrom einen neuen, längeren Bitstrom mit demselben Informationsgehalt aber einem geringeren mittleren Informationsgehalt (pro Bit) erzeugen.

Da die notwendige Höhe der Redundanz bei gegebener Zielfehlerrate von der Fehlerrate des Nachrichtenkanals abhängt, lässt sich kein Verfahren für alle Nachrichtenkanäle generalisieren. Man spricht daher bei den Maßnahmen zur Übertragungsfehlererkennung und -vermeidung von *Kanalcodierung*.

Zwei grundsätzlich mögliche Maßnahmen zur *Fehlerkorrektur* sind zu unterscheiden:

- (Vorwärts-)Fehlerkorrektur, auch FEC (Forward Error Correction): Der Empfänger erkennt aufgrund der Redundanz Fehler im Code und kann die Originalnachricht rekonstruieren, weil die fehlerhafte Nachricht (z. B. bei genau einem Fehler) eindeutig auf die Originalnachricht zurückzuführen ist. Zu diesem Zweck ist eine höhere Redundanz auch bei guten Übertragungsbedingungen notwendig, allerdings benötigt man keinen Rückkanal und die Korrektur ist schneller, weil sie im Empfänger stattfindet. Allerdings besteht die Gefahr, dass beim Auftreten von mehr als der zugelassenen Zahl von Fehlern die Korrektur auf eine falsche aber gültige Botschaft führt. FEC wird z. B. bei CDs eingesetzt um Kratzer zu kompensieren.
- Fehlerkontrolle durch fehlererkennende Codes, auch ARQ (Automatic Repeat Request): Der Empfänger erkennt Fehler und meldet diese an den Sender zurück, damit die Sendung wiederholt werden kann. Dies ist allerdings nur bei einem vorhandenen Rückkanal möglich, bei einer CD oder einem Satellitenempfänger wäre dies sinnlos.

Beim ARQ lassen sich verschiedene Strategien unterscheiden. Ein einfaches Stop-and-Wait-Protokoll basiert darauf, dass der Sender auf jede Nachricht hin eine Quittierung des korrekten Empfangs oder gegebenenfalls eine Fehlermitteilung durch den Empfänger erwartet (Handshake). Kommt nach einer gewissen Zeit keine Quittungsbotschaft (Acknowledgement) oder trifft eine Fehlerbotschaft ein, wird die Sendung der Nachricht wiederholt (Timeout). Dies führt zu einer sehr ineffizienten Übertragung, da in der Zeit, in der der Empfänger die Nachricht verarbeitet, gleich mehrere neue Nachrichten gesendet werden könnten. Außerdem können auch die Quittungs- bzw. Fehlerbotschaften verloren gehen, so dass der Sender nicht weiß, welche Nachricht er bei Bedarf zu wiederholen hat. Zwei Mechanismen helfen hier insbesondere weiter [5]:

1. Die Nachrichten werden mit einer laufenden Sequenznummer ausgestattet. Um das notwendige Datenfeld zu begrenzen, toleriert man, dass sich die Nummer gelegentlich, z. B. nach 256 Nachrichten, wiederholt. Quittungs- oder Fehlerbotschaften beziehen sich immer auf diese Nummer.
2. Eine festgelegte Anzahl von Nachrichten wird im Sender und im Empfänger zwischengespeichert und zunächst en bloc versendet. Der Empfänger fordert nun gezielt fehlerhafte Nachrichten nach (Selective Repeat) oder er meldet die erste fehlerhafte

Nachricht zurück und erwartet den neuerlichen Empfang aller seit dieser fehlerhaften Nachricht versendeten Daten (Go-back-N). Auch hier wird mit Timeout gearbeitet. Der Empfänger setzt nun aufgrund der Sequenznummer alle Nachrichten reihenfolgerichtig zusammen. Sobald sich Sender und Empfänger darüber einig sind, dass eine Nachricht wohlbehalten empfangen wurde, wird sie aus dem Zwischenspeicher beider Kommunikationsteilnehmer gelöscht. Dieses Verfahren ist unter dem Namen Sliding-Window bekannt.

Je nach Anforderungen an die Übertragungsgeschwindigkeit und die Restfehlerwahrscheinlichkeit (zwei Forderungen, die sich widersprechen) unterscheiden sich die Strategien zur Fehlerkontrolle bei den einzelnen Netzwerken.

Die Beschreibung des ISO/OSI Stacks endet an dieser Stelle, da für ein weiteres Verständnis der gezeigten seriellen Schnittstellen die Kenntnis der beiden ersten Schichten genügt. Ein weitergehender Überblick über die Rechnerkommunikation ist unter anderem bei [5] oder [6] zu erhalten.

5.2 Serielle Schnittstellen

5.2.1 Universal Asynchronous Receiver/Transmitter (UART)

Die UART Schnittstelle (Universal Asynchronous Receiver/Transmitter) gehört zur regelmäßigen Ausstattung von Mikrocontrollern und sehr viele Protokolle basieren auf dieser Schnittstelle, zum Beispiel das etwas außer Mode gekommene RS232, aber auch die in der Industrie nach wie vor verbreiteten Protokolle EIA-422, EIA-485 und zum Teil auch MODBUS und Profibus. Auch der im Automobil zunehmend an Bedeutung gewinnende LIN-Bus basiert in Grundzügen auf der UART-Schnittstelle und kann, wie später gezeigt wird, einfach mit einem UART aufgebaut werden. Ein UART-Treiber dient der Hardwareanbindung von verschiedenen höheren Protokollsichten und ist für Sensornetzwerke insofern interessant, als dass sich einfache Verbindungen zu PCs mit einer „virtuellen COM-Schnittstelle“ aufbauen lassen. Außerdem erlauben verschiedene Funkstandards die transparente Weiterleitung von seriellen Protokollen über eine UART Verbindung, beispielsweise das später noch näher zu erläuternde Bluetooth® Protokoll mit dem serial profile oder verschiedene andere Funkstandards.

Die Grundidee des UARTs besteht darin, eine Partner-zu-Partner (engl. *Peer-to-peer*) Verbindung aufzubauen, indem Datenpakete von 5 bis 9 Bit Größe im Non-return-to-Zero (NRZ) Verfahren aus einem Schieberegister getaktet werden, d. h. eine logische 1 wird durch einen Spannungspegel dargestellt, eine logische 0 durch einen zweiten, meistens 0 V. Die Bezeichnung *asynchronous* röhrt daher, dass keine Taktinformation mitgesendet wird und der Empfänger mit seinem eigenen Taktgenerator das Signal abtasten muss. Die Schrittweite (Baudrate) und die Länge des Datenpakets müssen vor der Übertragung zwischen Empfänger und Sender vereinbart sein. Manche UARTs besitzen jedoch darü-

ber hinaus noch eine optionale Leitung zur Taktübertragung, zum Beispiel in der gesamten AVR Familie; sie werden dann USART genannt (universal synchronous/asynchronous Receiver/Transmitter).

Ein klarer Nachteil der UART-Schnittstelle ist die Tatsache, dass die Taktgeneratoren, zum Beispiel aufgrund von Temperaturdrift oder Bauteiletoleranzen, bei längeren Sequenzen nicht mehr synchron laufen, daher ist die Schnittstelle nur für verhältnismäßig langsame Datenraten ausgelegt. Wie in Abb. 5.11 zu sehen, beginnt die Übertragung mit einem zum Ruhepegel des Busses komplementären so genannten Startbit. An diesem erkennt die Gegenstelle, dass ein Datenwort folgt und startet die Abtastung. Nach der Übertragung des Datenworts, das zwischen fünf und neun Bit lang sein kann – in der Regel werden acht Bit festgelegt – werden zunächst ein optionales Paritätsbit und dann ein oder zwei Bit im Ruhepegel (Stoppbits) übertragen, dies ist für den Empfänger ein Zeichen, dass die Übertragung beendet ist bevor ein erneutes Startbit gesendet wird. In der Abbildung sind die optionalen Bit mit einer eckigen Klammer gekennzeichnet.

Das Paritätsbit dient der Feststellung einer korrekten Übertragung. Durch seine Einführung wird der Coderaum der zu übertragenen Daten verdoppelt¹², so dass nur noch jeder zweite theoretisch mögliche Datenrahmen eine gültige Botschaft trägt. Damit kann pro Datenrahmen genau ein fehlerhaftes Bit erkannt werden. Die Ermittlungsvorschrift¹³ für die Parität ist

$$P_{\text{even}} = d_{n-1} \oplus \dots \oplus d_3 \oplus d_2 \oplus d_1 \oplus d_0 \oplus 0$$

$$P_{\text{odd}} = d_{n-1} \oplus \dots \oplus d_3 \oplus d_2 \oplus d_1 \oplus d_0 \oplus 1$$

Mit anderen Worten: Bei gerader (even) Parität wird mit einer „1“ die Zahl der Einsen im Nutzbyte auf eine gerade Zahl aufgefüllt, bei ungerader (odd) Parität auf eine ungerade Zahl. Bei der Datenübertragung müssen also folgende Parameter vorab geklärt werden: Datenrate, Zahl der übertragenen Bit, Parität (odd, even, no) und Zahl der Stoppbits (1 oder 2).

Abb. 5.11 zeigt zusätzlich die minimale Beschaltung zur Kommunikation mit einem UART. Die meisten UART Schnittstellen sind *vollduplexfähig*, d. h. sie besitzen einen Ausgang TxD zum Senden und einen Eingang RxD zum gleichzeitigen Empfangen eines Datenwertes. Schaltet man zwei Partner zusammen, müssen die beiden Leitungen selbstverständlich gekreuzt werden, d. h. RxD muss an TxD angeschlossen werden und umgekehrt.

5.2.1.1 Hardwareanbindung in der AVR-Familie

In der AVR-Familie ist mindestens ein USART auf jedem Prozessor vorhanden. An dieser Stelle soll nur der asynchrone Modus beschrieben werden. Die grundsätzliche interne Schaltung zeigt Abb. 5.12. Die USARTs sind bei AVR durchnummert, wir benutzen hier der Einfachheit halber immer den USART 0.

¹² Man spricht von einer „Hamming-Distanz“ von 2.

¹³ Das Zeichen \oplus steht für eine Exklusiv-ODER-Funktion.

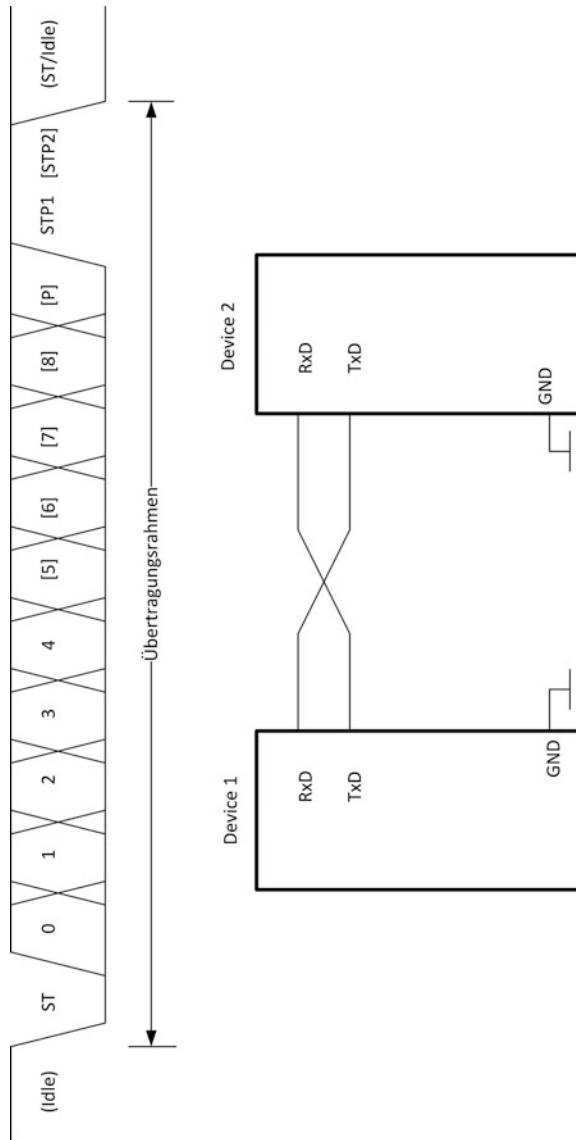


Abb. 5.11 Prinzipieller Aufbau eines UART Datenrahmens und der Beschaltung

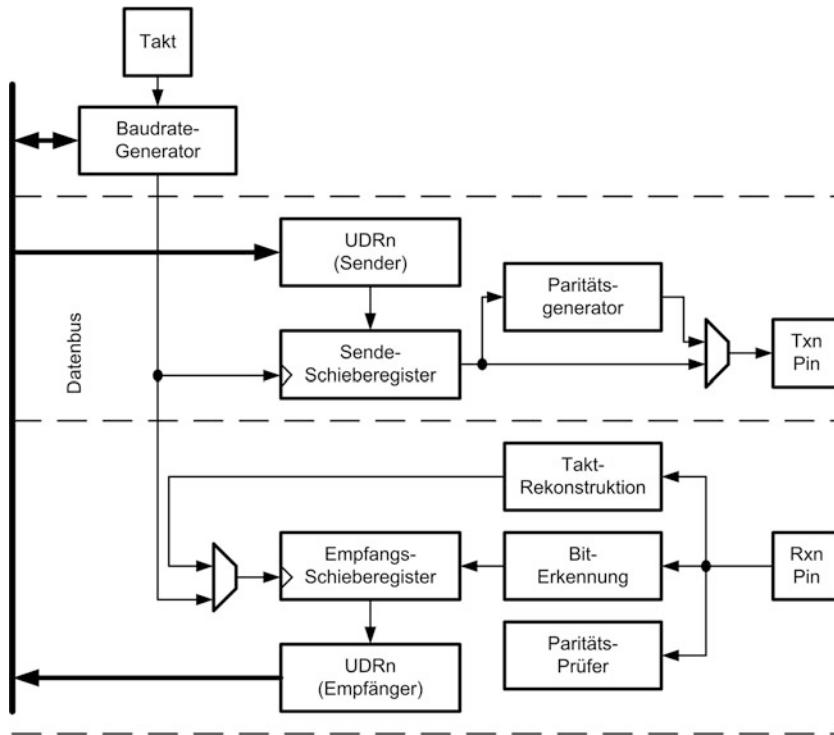


Abb. 5.12 Prinzip des UART im asynchronen Modus bei der AVR Familie

Die Datenrate wird beim ATMega88 im Baudrate-Generator durch

$$\text{BAUD} = \frac{\text{fosc}}{16 \cdot (\text{UBBR0} + 1)}$$

gebildet, wobei UBBR0 der Wert des gleichnamigen Baudratenregisters darstellt. Mit einer vorgegebenen Baudrate muss dieses also auf

$$\text{UBBR0} = \frac{\text{fosc}}{16 \cdot \text{BAUD}} - 1$$

gesetzt werden. Bei 9600 Baud @ 18.432 MHz ist der Wert von UBBR0 = 119.

5.2.1.2 UART-Register beim ATmega88

Neben dem Baudratenregister sind noch weitere Register für die Konfiguration und den Betrieb der USART Schnittstelle notwendig. Das Sende- und Empfangsregister der USART Schnittstelle sind auf derselben Adresse ansprechbar. Senden erfolgt durch Schreiben in das Register UDR0, nach einem erfolgreichen Empfangsvorgang sind die empfangenen

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
UCSR0A	RXC0	TXCO	UDRE0	FE0	DOR0	UPE0	U2X0	MPCM0
Richtung	R	R/W	R	R	R	R	R/W	R/W
Anfangswert	0	0	1	0	0	0	0	0

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
UCSR0B	RXCIE0	TXCIE0	UDRIE0	RXENO	TXENO	UCSZ02	RXB80	TXB80
Richtung	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W
Anfangswert	0	0	0	0	0	0	0	0

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
UCSR0C	UMSEL01	UMSEL00	UPM01	UPM00	USBS0	UCSZ01	UCSZ00	UCPOLO
Richtung	R/W							
Anfangswert	0	0	0	0	0	1	1	0

Abb. 5.13 Die Register zur Konfiguration des UART im ATmega88. (Nach Microchip)

Bit in diesem Register zu lesen. Die Schnittstelle wird in den USART Control and Status Registern konfiguriert (UCSR0A, UCSR0B, UCSR0C) (siehe Abb. 5.13).

In Register **UCSR0A** wird RXC0 vom Prozessor gesetzt, wenn ein neues Byte empfangen wurde. TXC0 wird gesetzt, wenn ein Sendevorgang abgeschlossen ist. UDRE0 ist gesetzt, wenn UDR0 leer ist. FE0 ist gesetzt, wenn ein Fehler beim Empfang detektiert wurde. DOR0 wird gesetzt, wenn UDR0 nicht gelesen wurde und ein neues Datenwort am Eingang ansteht. UPE0 zeigt einen Paritätsfehler an. U2X0 verdoppelt die Übertragungsgeschwindigkeit (Beschreibung dieses Modus s. ATMega88 Handbuch). MPCM0 schaltet den Multiprocessor-Modus ein. In diesem Modus werden mehrere Empfänger parallel geschaltet. Zunächst wird eine Adresse übertragen und nur derjenige Prozessor reagiert, der dieser Adresse zugeordnet ist. Weitere Beschreibung dieses speziellen Modus ist dem Handbuch zu entnehmen.

Das Register **UCSR0B** steuert das Interruptverhalten der USART-Schnittstelle und schaltet das Senden bzw. das Empfangen ein. RXCIE0 schaltet den Receive Complete Interrupt ein. TXCIE0 den Transmit Complete Interrupt. UDRIE0 den USART data register empty Interrupt. RXEN0 und TXEN0 schalten das Empfangen und Senden ein (Receive enable bzw. Transmit enable), UCSZ02 zusammen mit UCSZ01 und UCSZ00 im Register UCSR0C gibt die Zahl der Datenbits an (s. u.). RXB80 und TXB80 enthalten das 9. Bit im 9-Bit Modus.

UMSEL01 und UMSEL00 im Register **UCSR0C** geben den Schnittstellen-Modus an. Mit jeweils 0 ist der asynchrone Modus eingeschaltet, den wir in diesem Buch ausschließlich nutzen. UMP01 und UPM00 selektieren die Parität gemäß folgendem Schema (Tab. 5.2):

Tab. 5.2 Einstellung der Parität des UART

UPMn1	UPMn0	Parity mode
0	0	Disabled
0	1	Reserved
1	0	Enabled, even parity
1	1	Enabled, odd parity

USBS0 gesetzt bedeutet „zwei Stoppbits“, nicht gesetzt „ein Stoppbit“. Schließlich geben die Bits UCSZ02...UCSZ00 (USCR0B) die Zahl der Datenbits an. 0 1 1 bedeutet „8 Bit“, das übliche Maß, s. Codebeispiel. Schließlich gibt UCPOLO die Polarität des Taktes im synchronen Modus an, was wir hier nicht nutzen.

5.2.1.3 Initialisieren der UART Schnittstelle beim ATmega88

Mit einem üblichen Wert von 8 Datenbits, keiner Paritätsprüfung und 2 Stoppbits und bei 9600 Baud (9600,8,N,2), wird in einem File uart.c folgende Konfiguration stehen (Quelle: ATMega88 Handbuch):

```
void UART_Init(unsigned int baud)
{
    /* Baudrate setzen */
    UBRR0H = (unsigned char)(baud>>8);
    UBRR0L = (unsigned char)baud;
    /* Empfänger (receiver) und Sender (transmitter) sowie
       Received Complete Interrupt einschalten*/
    UCSR0B = (1<<RXEN0) | (1<<TXEN0) | (1<<RXCIE0);
    /* Rahmenformat: 8data, 2stop bit, no parity */
    UCSR0C = (1<<USBS0) | (3<<UCSZ00);
}
```

In diesem Fall wird baud=119 (im Fall 18.432 MHz Quarzfrequenz und einer ange strebten Baudrate von 9600 Baud) beim Aufruf übergeben. Diese kann man z. B. mit

```
#define F_OSC 18432000uL
#define BAUDRATE 9600uL
#define BAUDPARAM (F_OSC/(BAUDRATE*16))-1
```

so einstellen, dass der Wert im Precompiler ausgerechnet und als Konstante übertragen wird. Wenn sich die Baudrate zur Laufzeit nicht ändern muss, spart diese Maßnahme sehr viel Prozessorkapazität. Wichtig ist das Suffix uL hinter den Konstanten, damit der Precompiler den richtigen Wertebereich kennt. Ein Blick auf den übersetzten Code (Assembler) zeigt die Wirksamkeit dieser Maßnahme, hier wird nur noch eine Konstante in die Register r24 und r25 eingetragen, die zusammen den Übergabeparameter vom Typ unsigned short ergeben:

```

        uartInit (BAUDPARAM) ;
70:   87 e7      ldi     r24, 0x77    ; 119
72:   90 e0      ldi     r25, 0x00    ; 0
74:   f0 df      rcall   .-32      ; 0x56 <uartInit>
76:   ff cf      rjmp   .-2       ; 0x76 <main+0x6>

```

5.2.1.4 Empfangen von Daten

Grundsätzlich wird im Register **UCSR0A** das Bit RXC0 gesetzt, wenn ein neues Datenwort empfangen wurde. Da das Empfangen von Daten aber asynchron erfolgt, müsste eine Empfangsfunktion blockierend auf ein neues Byte warten, was nicht sinnvoll ist. Alternativ kann man das Empfangsbit in einem Task beispielsweise alle 10ms abfragen oder den entsprechenden Interrupt nutzen. Weiterhin soll der Treiber in einem eigenen Modul laufen. Generell ist es in C zwar möglich, globale Variable über die Modulgrenzen hinweg zu definieren, allerdings kann dies auch zu Fehlern durch unbeabsichtigte Verwendung führen. Wenn der Gesamtumfang des Programms es zulässt, sollte auf die Kapselung zurückgegriffen werden. Wenn nicht, kann natürlich von jeder Stelle des Programms direkt auf das UDR Datenregister des UART zugegriffen werden. Die Treiberdatei `uart.c` enthält für die Kapselung den folgenden Code:

```

#define DATA_RECEIVED 1
#define DATA_WAITING 0

unsigned char ucRecBuf;
unsigned char ucRecFlag=0;

ISR (USART_RX_vect)
{
    ucRecBuf = UDR0;
    ucRecFlag = DATA_RECEIVED;
}

/* liefert DATA_RECEIVED, wenn neue Daten anliegen und setzt ucRecFlag
zurück. Dient der Kapselung der Flags im Modul */
unsigned char UART_CheckReceived()
{
    if (ucRecFlag)
    {
        ucRecFlag=DATA_WAITING; //Reset Flag
        return DATA_RECEIVED;
    }
    else return DATA_WAITING
}

/* Dient der Kapselung der Daten im Modul */
unsigned char UART_GetReceived()
{
    return ucRecBuf;
}

```

Im Interrupt wird sichergestellt, dass ein empfangenes Byte stets abgeholt und in einen Puffer geschrieben wird. Der Lesepuffer ist damit wieder empfangsbereit und kann nicht versehentlich von einem weiteren empfangenen Datenwort überschrieben werden. Allerdings entbindet auch diese Lösung nicht vom Leeren des eigenen Datenpuffers, in einem Task muss regelmäßig `UART_CheckReceived()` aufgerufen werden, etwa mit

```
if (UART_CheckReceived()) {data=UART_GetReceived();}
```

Ist dies nicht sicher möglich, können die Daten auch in einen FIFO (Kap. 6) geschrieben werden, die Interrupt Service Routine ändert sich dann entsprechend:

```
ISR (USART_RX_vect)
{
    FIFOput (FIFOHandle, UDR0);
    ucRecFlag = DATA_RECEIVED;
}
```

Selbstverständlich muss ein entsprechender FIFO zunächst eingerichtet werden. Die Anwendungssoftware kann dann regelmäßig den FIFO auslesen Kap. 6.

Eine andere wichtige Bedeutung der Funktion `UART_CheckReceived()` liegt in der Kapselung der globalen Variablen. Die gesamte Funktionalität der USART Schnittstelle kann damit in ein einzelnes, vorcompiliertes Modul ausgelagert werden, ohne dass die Anwendungssoftware, die diese nutzt, auf globale Variablen dieses Moduls zugreifen muss.

5.2.1.5 Senden von Daten

Das Senden mit dem USART erfolgt durch einfaches Einschreiben in das USART Data Register UDR0. Der Sendevorgang dauert gewisse Zeit, 9600 Baud zum Beispiel entsprechen $1,041 \cdot 10^{-4}$ s pro Bit und bei einem 8,N,2-Rahmen (1 Startbit, 2 Stopppbit, 8 Datenbit = 11 Bit) beträgt die Übertragungszeit 1,146 ms pro Sendevorgang. Es empfiehlt sich also, das Senden in einen Task zu verlagern, der seltener als die Übertragungszeit aufgerufen wird, oder durch das Bit UDRE0 im Register UCSR0A, das das Ende des Übertragungsvorgangs anzeigen kann. Alternativ kann man die zu sendenden Bytes in einen Schreibpuffer füllen und diesen in Abhängigkeit vom Transmit Complete Interrupt leeren. Dazu später. Der einfachste Ein-Byte-Sendevorgang mit blockierendem Warten wäre zunächst:

```
void UART_TransmitChar(char data)
{
    /* Warten bis der Sendpuffer leer ist */
    while ( !( UCSR0A & (1<<UDRE0) ) )
    ;
    /* und abschicken */
    UDR0 = data;
}
```

Dies wird auch im Handbuch des ATMega88 vorgeschlagen. Allerdings bleibt im Fall der oben genannten Konfiguration der Prozessor für 1,146 ms blockiert.

5.2.1.6 Implementierung von `UART_WriteBuffer()`

Eine praktische Möglichkeit, einen ganzen Datenpuffer zu versenden, bietet die Nutzung der ISR (`USART_TXC_vect`). Liegen die Daten in einem Bytarray vor, so lassen sie sich leicht aus der ISR versenden, indem man den ersten Sendevorgang anstößt und in der ISR einen Positionsindex im Sendepuffer so lange erhöht, bis der Inhalt des Puffers am aktuellen Index NULL ist (Nullterminierter String) oder eine global übergebene Pufferlänge erreicht wurde. Fehlt dieses Abbruchkriterium, sendet der Controller mit der hier vorgeschlagenen Lösung ununterbrochen!

```
unsigned char ucSendIndex;
unsigned char ucBufLength;
unsigned char ucUART_Tx_Complete;
#define TXCOMPLETE 1
#define TXNOTCOMPLETE 0

void uartWriteBuffer() (char* data, unsigned char length)
{
    ucSendIndex=0;
    UDR0 = data[ucSendIndex++];
    ucBufLength = length;
    ucUART_Tx_Complete= TXNOTCOMPLETE;
}

ISR (USART_TXC_vect)
{
    if (data[ucSendIndex] && ucSendIndex<ucBufLength)
        UDR0 = data[ucSendIndex++];
    else ucUART_Tx_Complete= TXCOMPLETE;
}
```

Eine einfache Kontrolle ob der Datenpuffer vollständig übertragen wurde, besteht im regelmäßigen Test von `ucUART_Tx_Complete` auf `TXCOMPLETE`.

5.2.2 Serial Peripheral Interface (SPI)

5.2.2.1 Aufbau und Funktionsweise

Die getaktete serielle Schnittstelle SPI (Serial Peripheral Interface), bisweilen auch Clocked Serial Interface (CSI) genannt, die in vielen Prozessortypen zur Verfügung steht, bietet einen der populärsten Wege, auf Sensoren auf der Leiterplatte oder außerhalb des Gehäuses zuzugreifen. Dementsprechend kommen viele Sensoren mit einer SPI

Anbindung auf den Markt. Durch die Übertragung des Taktes auf einer getrennten Taktleitung (CLK, zuweilen auch SCK genannt) erlaubt sie die Synchronisation zwischen Sender und Empfänger bei wesentlich höheren Datenraten als bei asynchronen Schnittstellen. Allerdings sei darauf hingewiesen, dass mit zunehmender Länge der Zuleitung ein erhöhtes EMV-Risiko auftritt. Auf der Leiterplatte beziehungsweise in der Leitung müssen deshalb unter Umständen EMV-Maßnahmen ergriffen werden, beispielsweise durch Masseflächen oder durch entsprechende Schirmmaßnahmen. Die SPI Schnittstelle ist immer dann eine schnelle und sehr bequeme Kommunikationsschnittstelle, wenn es um den Betrieb einer Master CPU mit mehreren untergeordneten Knoten (Slaves) geht. Der Takt geht dabei immer vom Master aus, insofern muss er beim Slave nicht konfiguriert werden. Der jeweilige Slave wird über die Slave Select (SS) Verbindung angesteuert. Ist diese low, ist der Slave aktiv, andernfalls nicht. Beim Takten schiebt der Master mit der entweder aufsteigenden oder abfallenden Flanke (wählbar über Register) die Daten aus dem Master-Schieberegister in das Slave-Schieberegister und mit demselben Takt, im Ring herum, die Daten vom Slave in den Master. Es findet damit also ein Datenaustausch zwischen Master und Slave statt (Abb. 5.14). Die Anbindung geschieht über die Leitungen MISO (Master In Slave Out) als Eingang des Masters und MOSI (Master Out Slave In) als Ausgang des Masters. Aufgrund der Funktionalität der Leitungen ist also MISO beim Master immer als Eingang geschaltet und bei allen Slaves auf Ausgang. Die Slave Select Leitungen verhindern, dass mehrere Ausgänge parallel auf den Bus schalten.

Prozessoren der AVR Familie besitzen einen Slave Select Eingang, so dass sie auch als Slave betrieben werden können. Werden sie als Master betrieben, müssen je nach Anzahl der Slaves die entsprechenden Portpins als Ausgang zur Verfügung gestellt werden. Takt und Datenleitungen werden parallelgeschaltet.

In Abb. 5.15 ist skizziert, wie mehrere Sensoren über eine SPI Schnittstelle an einem Bus angeschlossen werden können. Wird ein µC der AVR Familie als Slave betrieben, weil er beispielsweise mit einem analogen Sensor einen digitalen Busteilnehmer bildet, kann sein Slave Select Pin auf Eingang betrieben werden.

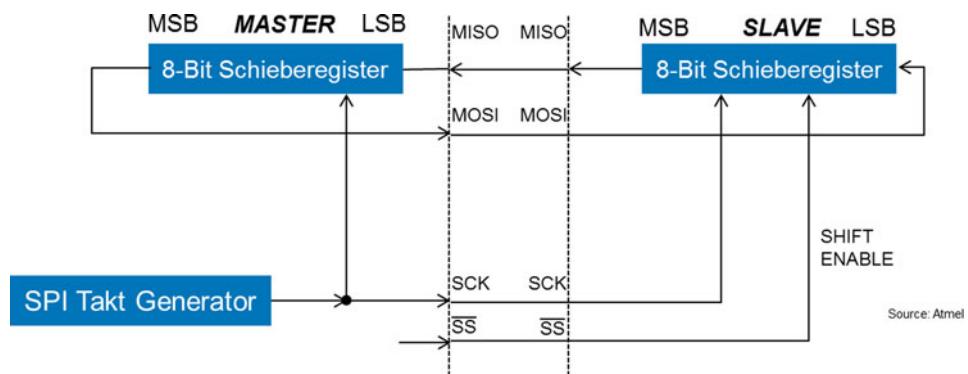
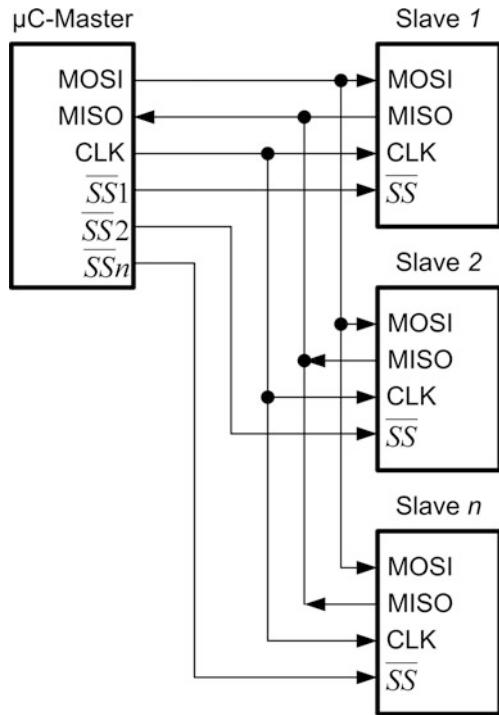


Abb. 5.14 Schemazeichnung der SPI Schnittstelle. (Nach Microchip)

Abb. 5.15 Beschaltung mehrerer Sensoren über die SPI Schnittstelle



Es ist zu beachten, dass es keine eigene Empfangsroutine für die SPI Schnittstelle gibt! Jeder Sendevorgang des Masters lädt gleichzeitig die Pufferdaten des Slaves in den Puffer des Masters um. Aus Sicht des Slaves müssen also die Daten zur Verfügung gestellt werden, bevor der Master eine neue Anfrage startet. Aus diesem Grund bestehen Sensorprotokolle zwischen dem Mikrocontroller und dem Sensor oft aus mehreren Datenworten, häufig wird zunächst ein Befehl an den Slave getaktet, anschließend wird eine 0x00 oder 0xFF nachgesendet, die dann dafür sorgt, dass die inzwischen bereitgestellten Daten des Sensors an den Mikrocontroller übertragen werden. In den folgenden Abschnitten finden sich einige Beispiele zu diesem Vorgehen.

5.2.2.2 Konfiguration der SPI-Schnittstelle

Um die Konfiguration der SPI-Schnittstelle zu verstehen, muss man sich zunächst über die Darstellung eines Bytes und die Art der Abtastung klarwerden. Bei Sensoren ist diese in der Regel festgelegt, manchmal lässt sie sich auch konfigurieren, in jedem Fall müssen Master und Slave sich auf gleiches Verhalten einigen. Zunächst legt man fest, in welcher Reihenfolge die einzelnen Bit eines Bytes übertragen werden. LSB first bedeutet, dass das Bit 0 (least significant bit) zuerst übertragen wird, MSB first steht dafür, dass das höchste Bit 7 (most significant bit) zuerst übertragen wird. Weiterhin gibt es vier verschiedene Modi (engl. Modes), die sich in

- Der Polarität des Taktes (CPOL)
- Der Takтphase (CPHA)

unterscheiden (vergl. Abb. 5.16, 5.17, 5.18 und 5.19).

CPOL=0 bedeutet, dass der Takt den Ruhepegel 0 (Low) hat, bei CPOL=1 ist der Ruhepegel 1 (High). Die Takтphase gibt an, ob die Übernahme des Signals bei der ersten oder der zweiten Flanke erfolgt, nachdem \overline{SS} auf Low geht. In den folgenden Abbildungen sind die vier möglichen Modi, die sich daraus ergeben, zu sehen.

Bei CHPA=0 stellen Master und Slave ihre Daten mit der fallenden \overline{SS} Flanke an, danach wird je nach CPOL auf die steigende oder fallende Flanke des Takтsignals CLK das anliegende Signal abgetastet (Master tastet MISO ab, Slave tastet MOSI ab). Die darauf folgende umgekehrte Flanke veranlasst Master und Slave jeweils dazu, das nächste Bit auf die Datenleitungen zu geben, bis das gesamte Datenwort übertragen ist.

Bei CHPA=1 wird erst auf die zweite Takтflanke abgetastet und die erste (im Fall CPOL=0 steigende, im Fall CPOL=1 fallende) Flanke, nachdem \overline{SS} auf Low ist, veranlasst den Slave, sein erstes Bit auf dem Bus zur Verfügung zu stellen, die komplementäre Flanke triggert wiederum die Abtastung.

Da die Taktung der SPI-Schnittstelle ausschließlich vom Master abhängt, ist es nicht notwendig, genaue Taktraten einzustellen. AVR Mikrocontroller stellen einen Verteiler zur Verfügung, der eine Taktung von $f_{osc}/2$ bis $f_{osc}/128$ ermöglicht. Bei 10 Mhz Prozessortakt also zwischen 78 kBit/s und 5 Mbit/s. Bei der Auswahl der Taktfrequenz ist lediglich darauf zu achten, dass sie innerhalb des spezifizierten Bereichs aller angeschlos-

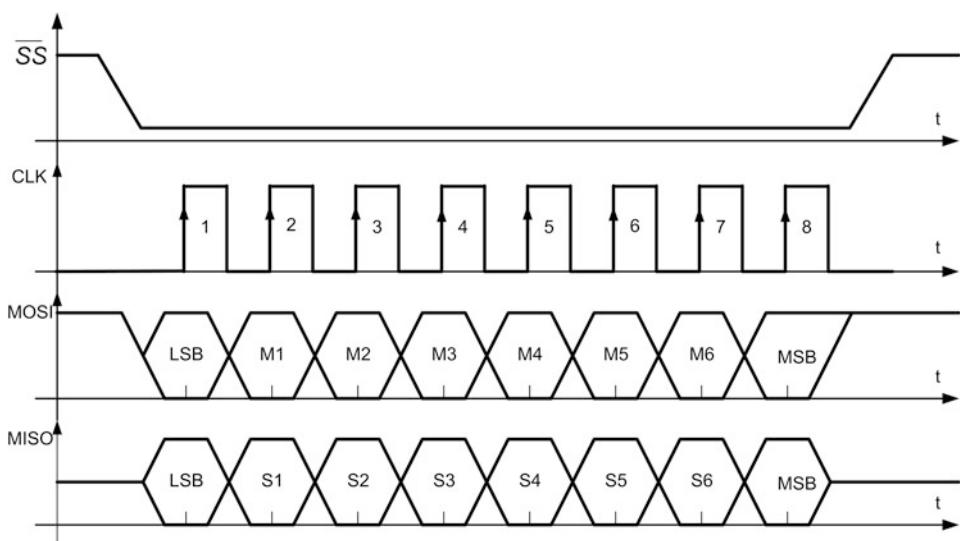


Abb. 5.16 SPI Modus 0: CPOL = 0, CPHA = 0 LSB first

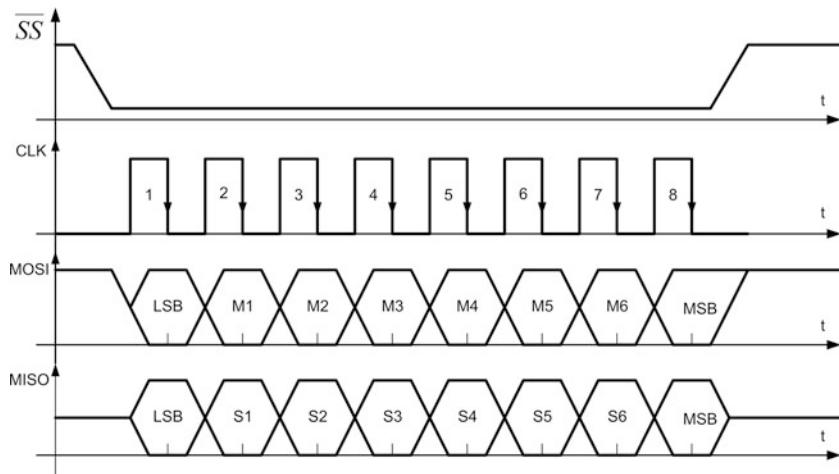


Abb. 5.17 SPI Modus 1: CPOL = 0, CPHA = 1 LSB first

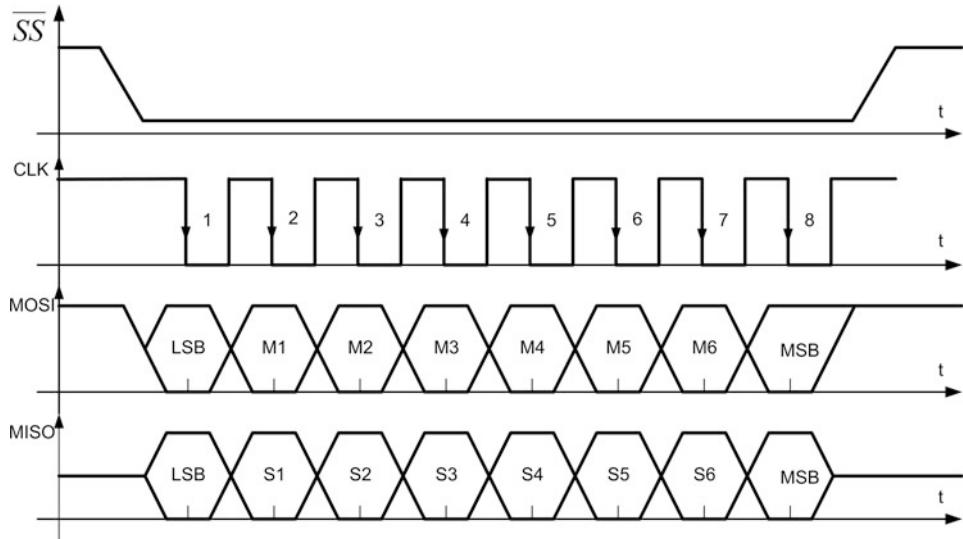


Abb. 5.18 SPI Modus 2: CPOL = 1, CPHA = 0 LSB first

senen Slaves bleibt. In der AVR Familie werden die Taktfrequenzen über die Bits SPR0 und SPR1 in SPCR-Register gesteuert wie in Tab. 5.3 dargestellt.

Über das Bit SPI2X (double SPI speed bit) im SPSR (SPI status register) können die Taktraten auf $f_{\text{osc}}/2$, $f_{\text{osc}}/8$, $f_{\text{osc}}/32$ und weiterhin auf $f_{\text{osc}}/64$ verdoppelt werden.

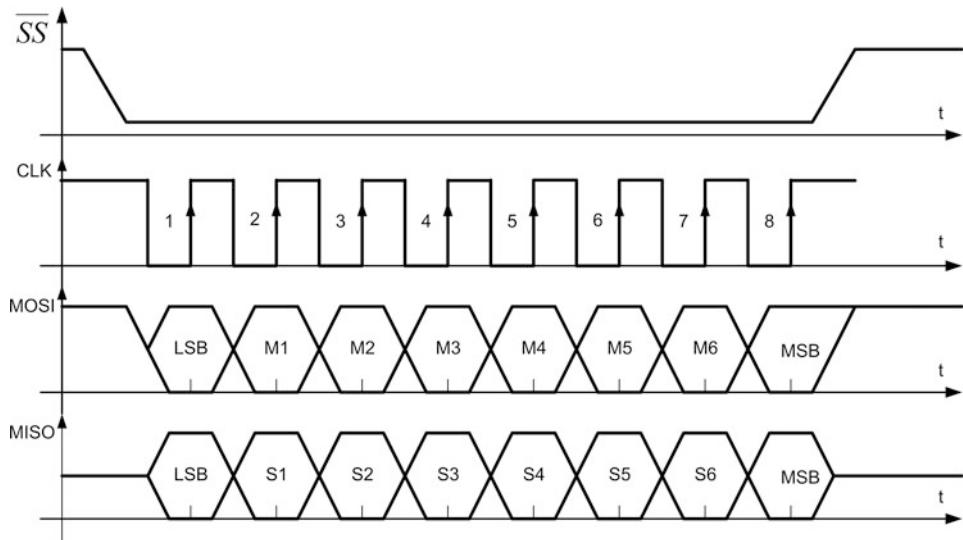


Abb. 5.19 SPI Modus 3: CPOL = 1, CPHA = 1 LSB first

Tab. 5.3 Taktraten im SPCR Register. (Nach Microchip)

SPR1	SPR0	Takt
0	0	fosc/4
0	1	fosc/16
1	0	fosc/64
1	1	fosc/128

Dementsprechend ist die Konfiguration relativ einfach zu bewerkstelligen:

```
void SPI_MasterInit()
{
    /* MOSI und SCK (CLK) als Output konfigurieren */
    DDR_SPI = (1<<DD_MOSI) | (1<<DD_SCK);
    /* SPI im Mastermode einschalten, Taktrate fOSC/16, Modus 0 */
    SPCR = (1<<SPE) | (1<<MSTR) | (1<<SPR0);
    /* Modus 1: SPCR |= (1 << CPHA), Modus 2: SPCR |= (1<<CPOL) */
    /* Modus 3: SPCR |= (1<<CPH) | (1<<CPOL) */
}
```

Über Senden und Empfangen lässt sich ähnliches sagen, wie bei der UART Schnittstelle. Das Senden wird im Master veranlasst, indem ein Byte in das SPDR Datenregister geschrieben wird.

- Der Slave ist dann empfangsbereit, wenn sein Chip Select (CS) Eingang auf Low liegt. Daher muss in jedem Fall eine Leitung von einem Port-Pin des Masters auf die CS-Leitung verdrahtet und konfiguriert werden.

Sobald die Übertragung beendet ist, wird das SPIF Flag im Statusregister SPSR gesetzt und auf Wunsch (SPIE=1 im SPCR Register) wird ein Interrupt ausgelöst. Demzufolge sieht die Senderoutine im Master ähnlich aus wie bei der UART Schnittstelle

```
void SPI_MasterTransmitChar(char cCharToSend)
{
    /* Starte Übertragung*/
    SPDR = cCharToSend;
    /* Achtung! Übertragung läuft weiter auch wenn Funktion beendet wurde*/
}
```

Auch hier haben wir das Problem, dass beim aufeinanderfolgenden Senden gewartet werden muss, bis der Sendepuffer leer ist. Analog zur UART Schnittstelle kann man einfach blockierend warten (Achtung! Solche Schleifen werden oft vom Optimierer des Compilers entfernt), wie hier:

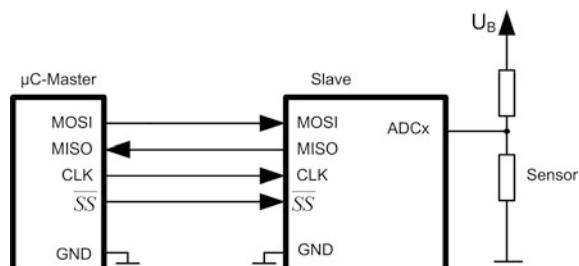
```
/* Starte Übertragung */
SPDR = cCharToSend;
/* Warte bis Übertragung beendet ist */
while(!(SPSR & (1<<SPIF)));
```

Oder man sendet aus einem zeitgesteuerten Task heraus, bzw. interruptgesteuert, bis der Sendepuffer leer ist. Nach dem Sendevorgang befindet sich gemäß Abb. 5.14 der Inhalt des Slavesenderegisters im Register SPDR. Insofern gibt es keine eigene Empfangsfunktion für die SPI Schnittstelle.

5.2.2.3 SPI Schnittstelle im Slave-Betrieb

Im Slavebetrieb empfiehlt sich dringend die Nutzung der Interruptserviceroutine der SPI Schnittstelle. An folgendem Szenario sei dies verdeutlicht (Abb. 5.20)

Abb. 5.20 Nutzung eines Mikrocontrollers mit analogem Sensor im Slavebetrieb



Der rechte Mikrocontroller ist im Slavebetrieb konfiguriert:

```
void SPI_SlaveInit(void)
{
    /* MISO als Ausgang setzen, die anderen (CLK,/SS) als Eingang */
    DDR_SPI = (1<<DD_MISO);
    /* SPI Einschalten und Interrupt freigeben*/
    SPCR = (1<<SPE) | (1<<SPIE);
    sei();
}
```

Im Schaltungsbeispiel in Abb. 5.20 soll nur ein Byte des AD-Wandlers aktiv genutzt werden. Der Ablauf gestaltet sich wie folgt: Der Master sendet ein Befehlsbyte, beispielsweise um den aktuellen Sensorwert auszulesen. In der Praxis wird ein ganzer Befehlssatz verwendet, in dem unter anderem der Sensor kalibriert werden kann, ein Fehlerregister ausgelesen und gelöscht, die Versions- und Seriennummer des Sensors ausgelesen oder Messmodi umgeschaltet werden können. Da die SPI-Kommunikation immer einen Austausch beinhaltet, kann der Sensor während dieser Übertragung ein Statusbyte zurückgeben, das beispielsweise beinhaltet, ob gültige Daten vorliegen oder ob ein Fehler speichereintrag existiert. Nach dem Befehlswort sendet der Master acht Nullen (0x00), in dieser Zeit liegt im SPDR Register des Sensors der abzufragende Wert bereit.

Der prinzipielle Ablauf ist in Abb. 5.21 zu sehen. Viele SPI Sensoren sind jedoch so komplex, dass 16 Bit Sequenzen übertragen werden, dies gestaltet sich dann analog.

Im Master müssen also nacheinander zwei Byte gesendet werden, mit einer angemessenen Pause dazwischen, damit der Slave seine Daten bereitstellen kann. Dies kann in einem zeitgesteuerten Task (siehe Kap. 6) erfolgen oder durch blockierendes Warten, was allerdings in der Regel im Echtzeitbetrieb nicht erlaubt und auch nicht sinnvoll ist. Der Einfachheit halber soll der Master im folgenden Codebeispiel zumindest blockierend warten, bis die Schnittstelle das Datenbyte versendet hat. Außerdem wird dieses Beispiel angenommen, dass die CS-Leitung des Sensors auf Port B2 verdrahtet ist. Wir erweitern die Senderoutine im Master folgendermaßen:

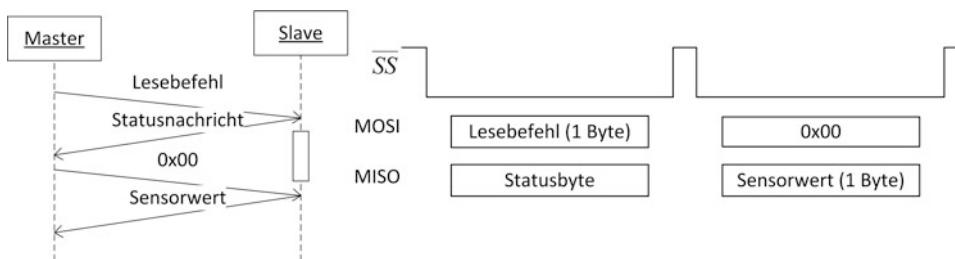


Abb. 5.21 Ablauf einer SPI Sequenz mit Befehlsbyte

```
#define SPI_SLAVESELECT_TEMPERATURE (1<<PB2)
unsigned char SPI_send(unsigned char cData)
{
    PORTB &= ~SPI_SLAVESELECT_TEMPERATURE; //Slave select auf low
    SPDR = cData;
    while(!(SPSR & (1<<SPIF))) //Blockierendes Warten
    {
    }
    PORTB |= SPI_SLAVESELECT_TEMPERATURE; //Slave select auf high
    return SPDR ; //Slave Antwort zurückschicken
}
```

In einem 1 ms Task sieht der Mastercode dann so aus:

```
//Zustände des SPI Masters
#define SPI_SENDCOMMAND 0x10
#define SPI_WAITANSWER 0x20

#define SPI_READSENSOR 0x15
#define SPI_VOID 0x00

(...)

if (SPI_SENDCOMMAND == masterstate)
{
    slavestate = SPI_send(SPI_READSENSOR);
    //Hier kann man den Status des Slaves auswerten
    masterstate = SPI_WAITANSWER ; //Warte nun auf Antwort
}
else if (SPI_WAITANSWER == masterstate)
{
    sensorvalue = SPI_send(SPI_VOID);
    masterstate = SPI_SENDCOMMAND; //Bereit für das nächste Senden
}
(...)
```

Wenn dieser Code jede Millisekunde aufgerufen wird, sendet der Master im Wechsel das Sendekommando und die 0x00, die den Slave zum Aussenden des Messwertes veranlasst. Dies entspricht einem Zustandsautomaten mit den Zuständen `SPI_SENDCOMMAND` und `SPI_WAITANSWER`. Durch den Aufruf jede Millisekunde kann man sich das blockierende Warten sparen, da die Übertragung mit Sicherheit beendet ist.

Im Slave (also dem zweiten Mikrocontroller in Abb. 5.20) wird der Betrieb durch den SPI Interrupt gesteuert. Auch der Slave kennt zunächst zwei Zustände: Warten auf einen Befehl und Versenden des Sensorwertes. Für den folgenden Code wird angenommen, dass ein Statusbyte `unsigned char slavestate` existiert, in dem immer der aktuelle Status eingeschrieben wird.

```

ISR(SPI_STC_vect)
{
    SPI_recByte = SPDR ;
    if (SPI_READSENSOR == SPI_recByte)
    {
        /* Beim nächsten Takten wird der Sensorwert übergeben */
        SPDR = sensorvalue;
    }
    else if (SPI_VOID == SPI_recByte)
    {
        /* Sensorwert wurde übergeben und damit wird in die
           Ausgangslage zurückgesprungen */
        SPDR = slavestate;
    }
    else
    {
        /* Hier kann man eine Fehlerbehandlung triggern */
    }
}

```

Die weitere Konfiguration der SPI Schnittstelle kann dem Datenblatt des jeweiligen Prozessors entnommen werden.

5.2.2.4 SPI Schnittstelle in einem Sensornetzwerk

Für die effektive Programmierung eines Mikrocontrollers, der als Master ein SPI-Sensornetzwerk ansteuert, kann ein SPI-Softwaremodul erstellt werden. Der Zugriff auf die Funktionen dieses Moduls kann sowohl aus der main-Funktion, als auch aus den entsprechenden Softwaremodulen der Sensoren stattfinden. Das Modul stellt Funktionen für die Initialisierung der SPI-Schnittstelle, für die Ansteuerung der Slave Select Leitung eines Sensors und für den Byteaustausch zwischen Master und Slave zur Verfügung. Das Modul kann auch für andere Mikrocontroller der Familie ATmega benutzt werden, indem man die Definitionen in der Headerdatei SPI.h ändert, die die Schnittstelle charakterisieren. Folgendes Beispiel stellt die Definitionen eines ATmegaX8 dar:

```

//MOSI-Signal
#define SPI_MOSI_DDR_REG          DDRB
#define SPI_MOSI_PORT_REG         PORTB
#define SPI_MOSI_BIT              PB3
//MISO-Signal
#define SPI_MISO_DDR_REG          DDRB
#define SPI_MISO_PORT_REG         PORTB
#define SPI_MISO_BIT              PB4
//Clock-Signal
#define SPI_CLK_DDR_REG           DDRB

```

```
#define SPI_CLK_PORT_REG PORTB
#define SPI_CLK_BIT PB5
```

Die Abb. 5.22 stellt das Flussdiagramm eines Programms für einen Mikrocontroller dar, der als Master über SPI zwei Slaves (Sensoren) mit unterschiedlichen Modi ansteuert.

Nach der allgemeinen Initialisierung, bevor der Mikrocontroller als SPI-Master konfiguriert wird, muss der dedizierte Slave Select Anschluss als Ausgang konfiguriert werden. Damit das SPI-Modul von der jeweiligen Board-Konfiguration unabhängig bleibt, werden die Mikrocontrolleranschlüsse für die Ansteuerung der Slave Select Eingänge für jeden Sensor in der Hauptdatei definiert. Dazu gibt es für jeden über die SPI-Schnittstelle anzu-steuernden Baustein eine Datenstruktur, in der die Pins aufgeführt sind, die den Baustein ansteuern:

```
typedef struct{

    volatile uint8_t* CS_DDR;
    volatile uint8_t* CS_PORT;
    uint8_t CS_pin;
    uint8_t CS_state;
} tspiHandle;
```

Diese Struktur speichert die Adressen des DDR- und PORT-Registers und den An-schluss der Slave Select Leitung eines Sensors. Die Mikrocontrolleranschlüsse für die Ansteuerung der Slave Select Leitung der Sensoren werden als Ausgang konfiguriert und mit dem Aufruf der Funktion `SPI_MasterInit_CS` auf High gesetzt. Als Aufrufparameter dieser Funktion wird das entsprechende Definitionsarray des Sensors benutzt. Der Code dieser Funktion ist hier aufgelistet:

```
void SPI_MasterInit_CS(tspiHandle tspi_pins)
{
    // entsprechendes Bit im Data-Direction-Register wird auf High
    // gesetzt (Output)
    *sdevice_pins.CS_DDR |= 1 << sdevice_pins.CS_pin;
    // entsprechendes Bit im PORT-Register wird auf High gesetzt
    *sdevice_pins.CS_PORT |= 1 << sdevice_pins.CS_pin;
}
```

Beispiele für konkrete Strukturen sind in Kap. 6 und 7 beschrieben.

Unterschiedliche SPI-Konfigurationen der Sensoren aus einem Netzwerk erzwingen eine neue Initialisierung der SPI-Schnittstelle des Masters vor der Ansteuerung eines Slaves. Mit der Funktion `SPI_MASTER_Init()` wird die Schnittstelle des Mikrocontrollers entsprechend konfiguriert.

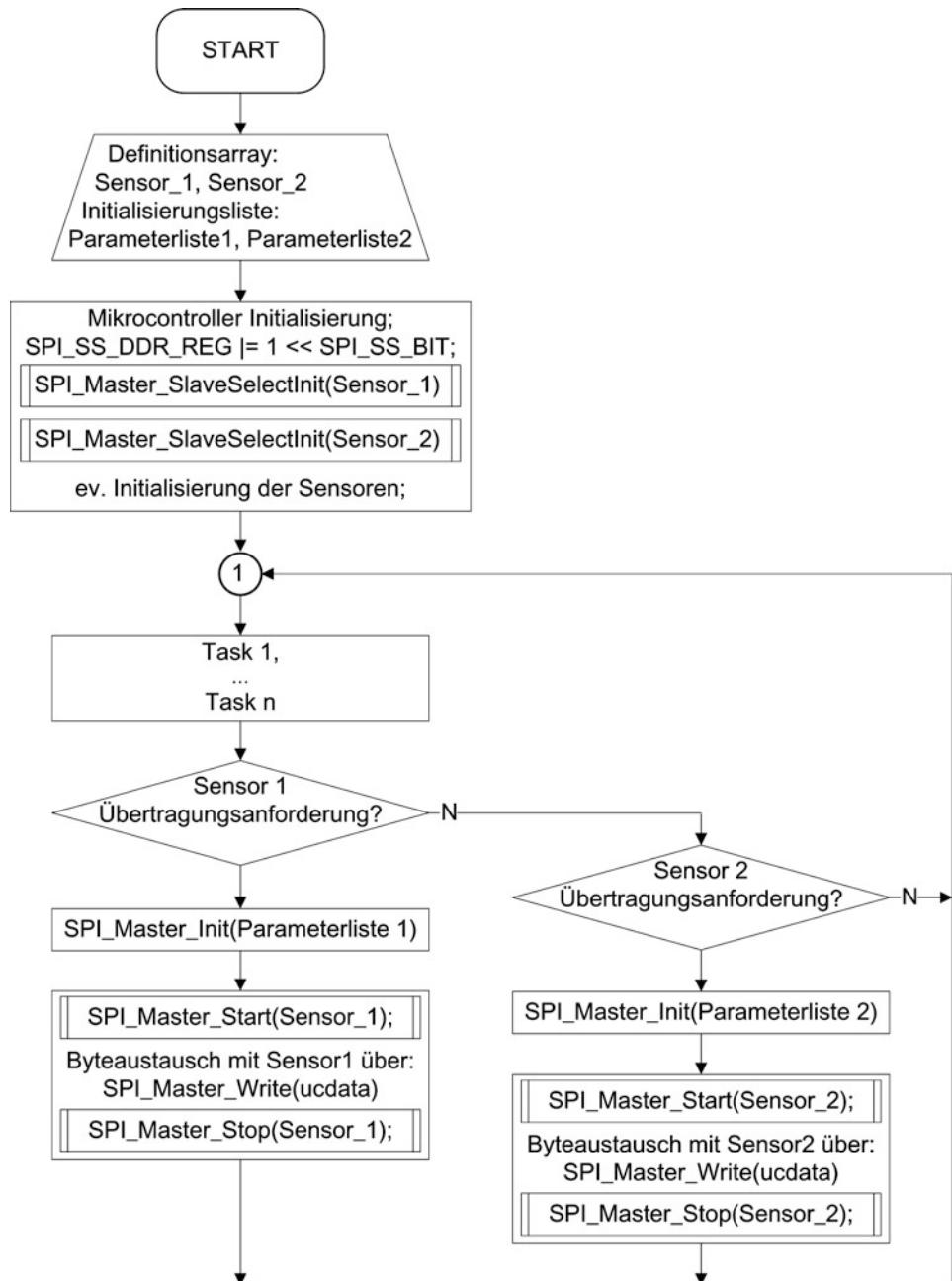


Abb. 5.22 Ansteuerung zweier Sensoren mit unterschiedlichen SPI-Modi

```

void SPI_Master_Init(uint8_t ucspi_interrupt, uint8_t ucspi_data_order,
                     uint8_t ucspi_mode, uint8_t ucspi_sck_freq)
{
    SPI_MOSI_DDR_REG |= 1 << SPI_MOSI_BIT;
    //MOSI-Pin wird auf Ausgang deklariert
    SPI_MISO_DDR_REG &= ~(1 << SPI_MISO_BIT);
    //MISO-Pin wird auf Eingang deklariert
    SPI_CLK_DDR_REG |= 1 << SPI_CLK_BIT;
    //CLK-Pin wird auf Ausgang deklariert
    SPI_CONTROL_REGISTER = SPI_MASTER | SPI_ENABLE | ucspi_data_order |
                           ucspi_interrupt | (ucspi_mode << 2) | (ucspi_sck_freq % 4);
    //uC wird als Master deklariert, die SPI-Schnittstelle wird freigegeben
    //falls gewünscht, wird der SPI-Interrupt freigegeben und die
    //Bitreihenfolge wird bestimmt
    //der Übertragungsmodus wird bestimmt
    SPI_STATUS_REGISTER = ucspi_sck_freq / 4;
    //die gewünschte SPI-Taktfrequenz wird gewählt
}

```

Für eine bessere Lesbarkeit der Software können als Aufrufparameter der Initialisierungsfunktion folgende Definitionen benutzt werden. Die Aufrufparameter sind im Flussdiagramm als Parameterliste zusammengefasst.

```

//SPI Register
#define SPI_CONTROL_REGISTER           SPCR
#define SPI_DATA_REGISTER             SPDR
#define SPI_STATUS_REGISTER          SPSR
// 
#define SPI_ENABLE                    0x40
#define SPI_MASTER                   0x10
//SPIE-Bit in das SPCR-Register (SPI-Interrupt Enable Bit)
#define SPI_INTERRUPT_ENABLE         0x80
#define SPI_INTERRUPT_DISABLE        0x00
//DORD-Bit in das SPCR-Register (data order: LSB or MSB first)
#define SPI_LSB_FIRST                0x20
#define SPI_MSB_FIRST                0x00
//SPI-Modus wird über 2 Bits bestimmt CPOL und CPHA im SPCR-Register
#define SPI_MODE_0                   0x00
#define SPI_MODE_1                   0x01
#define SPI_MODE_2                   0x02
#define SPI_MODE_3                   0x03
//Taktfrequenz der SPI-Schnittstelle
//(FOSC = die Taktfrequenz des Mikrocontrollers)
#define SPI_FOSC_DIV_2               0x04
#define SPI_FOSC_DIV_4               0x00

```

```
#define SPI_FOSC_DIV_8          0x05
#define SPI_FOSC_DIV_16         0x01
#define SPI_FOSC_DIV_32         0x06
#define SPI_FOSC_DIV_64         0x02
#define SPI_FOSC_DIV_128        0x03
#define SPI_RUNNING             (! (SPSR & (1 << SPIF)))
```

Vor jeder Übertragungsanforderung muss der Master neu konfiguriert werden. Er startet die Kommunikation mit dem Slave, indem er den Chip Select Eingang mit dem Aufruf der Funktion `SPI_Master_Start` auf Low setzt und beendet sie mit dem Aufruf der Funktion `SPI_Master_Stop`.

```
void SPI_Master_Start(tspiHandle tspi_pins)
{
    *tspi_pins.CS_PORT &= ~(1 << tspi_pins.CS_pin);
}

void SPI_Master_Stop(tspiHandle tspi_pins)
{
    *tspi_pins.CS_PORT |= (1 << tspi_pins.CS_pin);
}
```

Mit `SPI_Master_Write` sendet der Master ein Byte an den Slave. Dieses Byte wird als Parameter beim Aufruf der Funktion übergeben. Gleichzeitig empfängt der Master ein Byte vom Slave, das von der Funktion zurückgegeben wird. Somit kann die gleiche Funktion für die bidirektionale Kommunikation verwendet werden.

```
uint8_t SPI_Master_Write(uint8_t ucdata)
{
    #define SPI_DATA_REGISTER      SPDR //Definition in der SPI.h Datei
    SPI_DATA_REGISTER = ucdata;
    while(SPI_RUNNING);
    return SPI_DATA_REGISTER;
}
```

5.2.3 I²C

Der I²C-Bus¹⁴ (oder I2C) wurde von der Firma Philips (heute NXP) [7–9] entwickelt und ermöglicht die Vernetzung von Mikrocontrollern, Sensoren und anderen Bausteinen wie: RAM- und EEPROM-Speicher, A/D-, D/A-Wandler, Port Erweiterungen, LCD- und LED-Display Ansteuerungen, Echtzeituhren, Radio- und TV-Empfänger, I²C-Bus Erweiterungen und vielen anderen. Basierend auf dieser Schnittstelle wurden weitere Protokolle

¹⁴ I²C – Inter-IC.

entwickelt wie z. B.: CBUS, System Management Bus (SMBus), Power Management Bus (PMBus), Intelligent Platform Management Interface (IPMI), Advanced Telecom Computing Architecture (ATCA) und Display Data Channel (DDC). Diese Protokolle erweitern die ursprüngliche Schnittstelle, sie verbessern sie oder passen sie auf spezielle Anwendungsbereiche an. Sie ermöglichen zum Beispiel die dynamische Vergabe der Slaveadressen, das Erzwingen der I²C-Kommunikation durch die Slaves über eine zusätzliche Interrupt-Leitung, oder die Erkennung und das Lösen eines den Bus blockierenden „Aufhängens“ eines Slaves. Bedingt ist es sogar möglich, dass ein Mikrocontroller als Master innerhalb eines einzigen Busses Slaves mit unterschiedlichen Protokollen ansteuert. Je nach Übertragungsrate und -richtung werden folgende Modi definiert (Tab. 5.4).

Der I²C-Bus funktioniert nach dem Master-Slave Prinzip. Master ist derjenige Busknoten, der die I²C-Kommunikation initiiert und beendet, den Takt für die Synchronisation der Datenübertragung bereitstellt und entsprechend der einprogrammierten Software Kommunikationsprobleme erkennt und löst. Jeder Slave besitzt eine Adresse, die ihn in einem Bus eindeutig identifiziert. Die Adresse kann 7 oder 10 Bit groß sein und ermöglicht die Adressierung von 128 bzw. 1024 unterschiedlichen Slaves im selben Bus. Devices mit unterschiedlichen Adresslängen können im selben Bus koexistieren. Bevor ein Master Daten überträgt oder empfängt, muss er einen Slave mit einer vorher vereinbarten Adresse ansprechen (adressieren). Die Adressgruppen 0000xxx und 1111xxx sind allerdings reserviert und sollen den Slaves nicht vergeben werden. Das Protokoll sieht vor, dass ein Master alle Slaves in einem Bus über die reservierten Adresse 0000000 gleichzeitig adressieren kann, um ihnen gleichzeitig eine Botschaft zu übermitteln. Man nennt diese eine Broadcast-Botschaft. Achtung, nicht alle I²C-Bauteile haben diese Funktion implementiert. I²C-Bauteile wie Speicher oder manche Sensoren bieten neben der fest programmierten Device Type Adresse eine über externe Anschlüsse einstellbare Device Chip Adresse an. Wenn n die Zahl der Adressanschlüsse ist und m die Zahl der möglichen logischen Zustände, dann können bis zu n^m gleiche Bauteile mit unterschiedlichen Adressen in einem Bus identifiziert werden. Im Allgemeinen ist m gleich 2, ein Anschluss kann an der Masse oder an der Versorgungsspannung angeschlossen sein. Wenn man den Speicher M24C64 als Beispiel nimmt, der drei Adresseingänge besitzt und dessen Device Typ Adresse 1010xxx lautet, dann gibt es 8 mögliche Adresskombinationen die in der Tab. 5.5 aufgelistet sind. Ein mit V_{cc} verbundener Adresseingang setzt das entsprechende Adressbit auf „1“. Selten, wie im Fall des Temperatursensors TMP175, kann einem

Tab. 5.4 I²C Modi

Modus	Übertragungsrate	Übertragungsrichtung
Standard	< 100 kBit/s	Bidirektional
Fast-mode	< 400 kBit/s	Bidirektional
Fast-mode Plus	< 1 Mbit/s	Bidirektional
High-speed mode	< 3,4 Mbit/s	Bidirektional
Ultra Fast-mode	< 5 Mbit/s	Unidirektional

Tab. 5.5 Adressierung eines M24C64-Bausteins

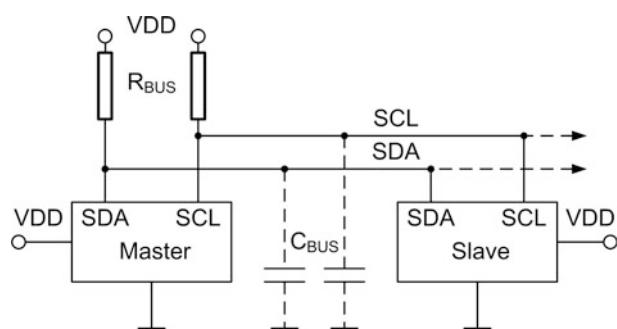
Adresseingänge			Chip Adresse								
E2	E1	E0	A6	A5	A4	A3	A2	A1	A0	R/W	
GND	GND	GND	1	0	1	0	0	0	0	1/0	
GND	GND	V _{CC}	1	0	1	0	0	0	1	1/0	
GND	V _{CC}	GND	1	0	1	0	0	1	0	1/0	
GND	V _{CC}	V _{CC}	1	0	1	0	0	1	1	1/0	
V _{CC}	GND	GND	1	0	1	0	1	0	0	1/0	
V _{CC}	GND	V _{CC}	1	0	1	0	1	0	1	1/0	
V _{CC}	V _{CC}	GND	1	0	1	0	1	1	0	1/0	
V _{CC}	V _{CC}	V _{CC}	1	0	1	0	1	1	1	1/0	

nicht angeschlossenen Pin ein dritter Zustand zugewiesen werden. Aus dem Datenblatt des Bausteins ist die Adresse entsprechend jeder Eingangskombination zu entnehmen.

5.2.3.1 I²C Bus Konfiguration

In der minimalen Konfiguration aus der Abb. 5.23 wird der Master mit dem Slave über die bidirektionalen Busleitungen: SDA (Datenleitung) und SCL (Taktleitung) verbunden. Diese Busleitungen werden über pull-up Widerstände (oder Bus-Widerstände) an der Versorgungsspannung angeschlossen, was zu einer UND-Verdrahtung der Devices führt. Weitere Devices (Master oder Slaves) können durch das Verbinden der dedizierten Anschlüsse SDA und SCL mit den entsprechenden Busleitungen an den Bus angeschlossen werden. In diesem Beispiel wird sowohl der Master als auch der Slave mit der gleichen Spannung versorgt, was nicht immer der Fall ist.

I²C wurde als true Multi-Master Bus konzipiert in dem jeder angeschlossene Mikrocontroller zeitweise als Master oder Slave funktionieren kann. In einem solchen Bus kann jedes Device, das als Master konfiguriert ist, die Kontrolle der Kommunikation übernehmen, wenn der Bus frei ist. Ein Bus gilt als frei, wenn beide Busleitungen eine spezifizierte Zeit auf High stehen. Eventuelle Kollisionen, die entstehen können, werden durch Arbitrierung erkannt und ohne Unterbrechung der Kommunikation oder Datenverlust gelöst.

Abb. 5.23 I²C – minimale Konfiguration

Die Arbitrierung, die hardwaremäßig in einem Master implementiert ist, greift ein, wenn zwei Master quasi gleichzeitig die Kontrolle des Busses übernehmen wollen. Ein Master vergleicht jedes von ihm gesendete Bit mit dem logischen Zustand der SDA-Leitung. Bei Unstimmigkeit (eine „1“ wurde gesendet aber die SDA-Leitung steht auf Low) verliert er die Arbitrierung und gibt durch die Software die Kontrolle des Busses ab. Wenn die Arbitrierung in der Adressierungsphase verloren geht, muss der Master sofort in den Slave-Modus umschalten, um reagieren zu können, falls er selbst adressiert werden soll.

Über I²C können Devices mit unterschiedlichen Technologien vernetzt werden (bipolar, NMOS, CMOS), die mit unterschiedlichen Spannungen versorgt werden oder mit verschiedenen maximalen Taktfrequenzen arbeiten. Um das zu ermöglichen, werden die minimale Ausgangsspannung und die maximale Eingangsspannung genormt:

$$\begin{aligned} V_{IL} &= 0,3 \times V_{DD} \\ V_{OH} &= 0,7 \times V_{DD}, \end{aligned} \quad (5.7)$$

wobei V_{DD} die Versorgungsspannung des Devices ist. Die maximale Anzahl der Busteilnehmer und die Länge der Busleitungen sind nicht normiert, jedoch durch die festgelegte maximale Bus-Kapazität von 400 pF begrenzt. Die Bus-Kapazität ist die gesamte elektrische Kapazität zwischen einer Busleitung und Masse. Sie setzt sich zusammen aus der Parallelschaltung der Eingangskapazitäten aller an der Leitung angeschlossenen Devices, der Leitungskapazität gegen Masse und die Kapazität der Anschlüsse. Da alle diese Kapazitäten parallelgeschaltet sind, addieren sie sich einfach. Die Erhöhung der Anzahl der Busteilnehmer oder der Leitungslänge kann zur Überschreitung der empfohlenen maximalen Bus-Kapazität führen. In diesem Fall kann die maximale Taktfrequenz nicht mehr gewährleistet werden. Die Taktfrequenz des Busses richtet sich in der Regel nach dem langsamsten Device. Ein Nachteil des I²C Protokolls besteht darin, dass ein Slave keine Möglichkeit hat, dem Master die Verfügbarkeit neuer Daten zu melden. Der Master muss die Daten stets im Polling-Betrieb abfragen.

Um ein I²C Netzwerk zu gestalten und die Software eines Masters zu erstellen, müssen die Slaveadressen und die Taktfrequenz des Busses bekannt sein. Ein Master muss mit den Slaves keine Vereinbarungen treffen, was das Erweitern eines Busses vereinfacht.

Hardwaremäßig muss nur der Buswiderstand bestimmt werden. Die Ausgänge der Busteilnehmer können zusammengeschaltet werden, weil sie einen Open-Collector- oder Open-Drain-Ausgang haben. Während der Kommunikation entlädt sich die Buskapazität schnell über einen leitenden Transistor, lädt sich aber nur langsam über den Buswiderstand auf, wenn alle Ausgangstransistoren sperren. Beim Aufladen verläuft die Spannung an der Bus-Kapazität nach der Gleichung:

$$V_{Cbus} = V_{DD} \left(1 - e^{-\frac{t}{\tau_{Bus}}} \right) \quad (5.8)$$

mit der Zeitkonstante:

$$\tau_{Bus} = R_{Bus} \cdot C_{Bus} \quad (5.9)$$

Die Anstiegszeit der Spannung $V_{C_{bus}}$ zwischen den, in der Gl. 5.7, angegebenen Grenzen muss kleiner sein als die spezifizierte Anstiegszeit t_r für jeden I²C Modus. Mit diesen Überlegungen ergibt sich für den Buswiderstand der maximale Wert [1]:

$$R_{Bus(max)} = \frac{t_r}{0,8473 \cdot C_{Bus}} \quad (5.10)$$

Um den maximalen Strom durch die Ausgangsstufen auf den spezifizierten Wert zu begrenzen, wird der minimale Buswiderstand entsprechend [1] berechnet:

$$R_{Bus(min)} = \frac{V_{DD} - V_{OL(max)}}{I_{OL}} \quad (5.11)$$

In den Datenblättern der Bauteile sind für den leitenden Zustand des Ausgangstransistors die Ausgangsspannung V_{OL} und der Ausgangstrom I_{OL} spezifiziert. Ein kleiner Buswiderstand ermöglicht die Kommunikation über längeren Leitungen, verursacht aber höhere Energieverluste. Bei der Wahl des Buswiderstandes muss die Versorgungsspannung, die Taktfrequenz, die Buskapazität und der Energieverbrauch berücksichtigt werden. Man kann sich bei der Wahl auch an den Empfehlungen der Bauteilhersteller orientieren.

5.2.3.2 Bus-Erweiterung

Um komplexe I²C-Busse dynamisch zu gestalten, längere Busleitungen zu ermöglichen, die maximale Buskapazität zu überschreiten, oder um Devices anzuschließen, die mit verschiedenen Spannungen versorgt werden, die gleiche Adresse haben, oder mit unterschiedlichen Taktfrequenzen arbeiten, benötigt man Bus-Erweiterungen. Eine Bus-Erweiterung ist ein Baustein, der auch über I²C angesteuert wird. Die genauen Bedingungen, unter denen man diese Bus-Erweiterungen einsetzen kann, sind in [8] und [9] beschrieben. Alle Buserweiterungen erzeugen eine Laufzeitverzögerung der Signale, die man berücksichtigen muss.

I²C-Repeater

Ein I²C-Repeater besitzt bidirektionale Treiber für die Busleitungen und spaltet den Bus in zwei Busabschnitte wie in Abb. 5.24. Über den EN-Eingang kann der rechte Busabschnitt vom linken isoliert werden. Kapazitiv werden die zwei Abschnitte durch den Repeater getrennt, so dass die Kapazität jedes Abschnittes 400 pF erreichen kann. Die Abschnitte können mit verschiedenen Spannungen versorgt werden. Der Repeater wird mit der niedrigen Spannung versorgt (meist 3,3 V), seine Eingänge sind aber 5 V tolerant. Wenn der Master aus dem Beispiel mit bis zu 400 kBit/s arbeiten kann, aber einige Devices nur im Standard Modus, dann kann man die Devices nach Bustakt trennen: im linken Busabschnitt der Master mit den schnellen Devices, im rechten die Langsamten. Bevor der Master die Kommunikation mit 400kBit/s initiiert, muss er über den Repeater den rechten Busabschnitt isolieren. Der EN-Eingang des Multiplexers darf während einer Kommunikation nicht umgeschaltet werden.

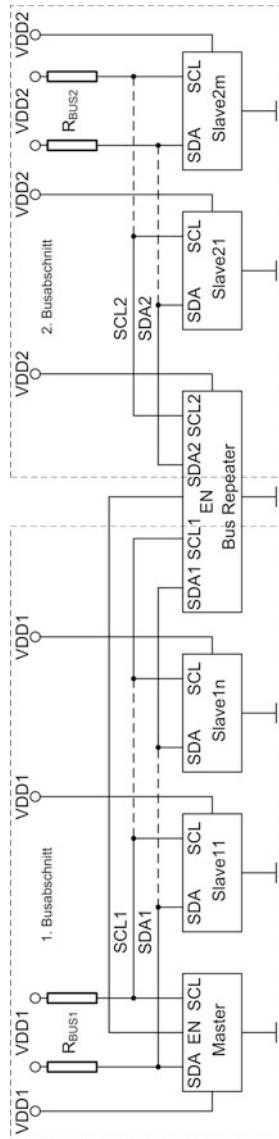


Abb. 5.24 I²C-Bus Erweiterung mit einem Repeater

I²C-Hub

Ein I²C-Hub ist eine Erweiterung eines I²C-Repeaters. Damit können mehrere Busabschnitte vernetzt werden. Diese können mit unterschiedlichen Spannungen versorgt werden, über verschiedenen Taktfrequenzen kommunizieren und jeder Busabschnitt kann 400 pF erreichen. Über mehrere EN-Eingänge kann der Master Busabschnitte isolieren.

I²C-Multiplexer

Um Devices mit der gleichen Adresse anzusteuern ohne dass Konflikte auftreten, verwendet man einen I²C-Multiplexer. Über den Multiplexer wählt der Master ein Device oder einen Busabschnitt an und bildet so eine neue Konfiguration des Busses. Die Bus-Kapazität Grenze errechnet sich in diesem Fall für den, über den angewählten Pfad, entstandenen Bus. Ein I²C-Multiplexer ermöglicht die Versorgung jedes Busabschnittes mit einer anderen Spannung ohne zusätzliche Pegelwandler.

I²C-Switch

Ein I²C-Switch ermöglicht komplexere Konfigurationen eines Busses als ein Multiplexer weil der Master gleichzeitig mit mehreren Busabschnitten vernetzt werden kann. Jeder Busabschnitt kann mit einer anderen Spannung versorgt werden, die Bus-Kapazitätsgrenze gilt für den aktiven Bus.

5.2.3.3 TWI

TWI¹⁵ ist eine Schnittstelle, die standardmäßig in den Mikrocontroller der ATmega Familie implementiert und mit I²C kompatibel ist [10]. Ein Mikrocontroller dieser Familie kann als Master oder als Slave im Fast Modus (bis 400 kHz) mit einer 7 Bit Adressierung arbeiten und ist Multi-Master fähig. An dieser Stelle wird nur der TWI Master Modus beschrieben. Die Kommunikation ist Byte-orientiert und die Schnittstelle interruptbasiert. Es werden immer 8 Bit (Bytes) übertragen, mit dem höherwertigen Bit (MSB) zuerst. Das Ende eines TWI-Vorgangs kann vom Mikrocontroller entweder in einer Funktion blockierend festgestellt werden, oder mit dem TWI-Interrupt. Die Übertragung eines Bytes (8 Datenbits + Bestätigungsbit) im Fast Modus (400 kHz) dauert ca. 22,5 µs. Die Clock-Leitung wird vom Master auf Low gehalten, bis der Interrupt bedient wurde, um die Busübernahme durch einen anderen Master zu verhindern. Der Master erzeugt acht Bittakte für die Byte-Synchronisation und einen neunten Takt für die Empfangsbestätigung. Wenn die SDA Leitung vom Empfänger der Daten während des neunten Taktes auf Low gezogen wird, spricht man von Acknowledge (ACK), ansonsten von Not Acknowledge (NACK). Eine vom Master gesendete Slaveadresse, die im Bus nicht vorhanden ist, wird mit NACK quittiert. Ein Slave sendet auch ein NACK, wenn er aufgrund interner Probleme keine weiteren Daten mehr annehmen kann. Mit einem NACK muss schließlich die Master Software auf das letzte von einem Slave gesendete Byte antworten.

¹⁵ TWI – two wire interface.

TWI-Register beim ATmega88

Die TWI-Schnittstelle besitzt einen Registersatz, der für die Konfiguration und den Betrieb der Schnittstelle notwendig ist. Die dazu gehörenden Register sind: Datenregister, Baudratenregister, Statusregister, Kontrollregister und für den Slave Betrieb ein Adressregister **TWAR** (TWI Slave address register) und ein Register für die Maskierung dieser Adresse **TWAMR** (TWI Slave address mask register).

Sende- und Empfangsregister **TWDR** (TWI data register) der TWI-Schnittstelle sind unter dem gleichen Namen ansprechbar. Im Sendemodus wird in das Register das zu übertragende Byte geschrieben, im Empfangsmodus enthält das Register das empfangene Byte.

Die Taktfrequenz f_{SCL} der Kommunikation wird durch

$$f_{SCL} = \frac{f_{OSC}}{16 + 2 \cdot TWBR \cdot TWI_{Prescaler}} \quad (5.12)$$

bestimmt, wobei f_{OSC} die Taktfrequenz des Mikrocontrollers ist. Der Wert von **TWBR** (TWI bit rate register) wird gemäß Gl. 5.13 eingestellt und der Wert des $TWI_{Prescaler}$ wird entsprechend Tab. 5.6 selektiert. TWPS1 und TWPS0 gehören zum Statusregister.

Wenn man die Gl. 5.12 nach TWBR auflöst, erhält man:

$$TWBR = \left(\frac{f_{OSC}}{f_{SCL}} - 16 \right) / (2 \cdot TWI_{Prescaler}) \quad (5.13)$$

Mit der Einstellung TWI Prescaler gleich 1 lassen sich Werte des TWBR Registers für Taktfrequenzen f_{SCL} ab ca. 40.000 Hz @ $f_{OSC} = 20$ MHz näherungsweise mit folgendem Makro ermitteln:

```
#define F_CPU 20000000UL      //Clock des Board-µC in Hz
#define TWI_SCL_FREQ 400000UL   /*gewünschte TWI Taktfrequenz in Hz;
                                wird entsprechend der konkreten Anwendung geändert*/
#define TWI_MASTER_CLOCK (((F_CPU / TWI_SCL_FREQ) - 16) / 2 +1)
//Wert des TWBR Registers
```

Über das **TWCR** (TWI control register) Register wird die TWI Kommunikation gesteuert:

Bit 7 – TWINT – wird von der Hardware bei Vorliegen der folgenden Bedingungen gesetzt: nach der Ausführung vom Master einer START- oder RESTART-Sequenz,

Tab. 5.6 Einstellung des TWI Prescalers

	TWPS1	TWPS0	TWI Prescaler
0	0	1	
0	1		4
1	0		16
1	1		64

nach dem Senden oder dem Empfang eines Bytes, nach der Erkennung der eigenen Adresse als Slave, nach dem Verlieren der Arbitration, nach einem unerlaubten START oder STOP Vorgang. Das Bit muss von der Software vor jedem TWI Vorgang durch das Einschreiben einer „1“ gelöscht werden. Wenn das Bit TWIE und das I-Bit im Statusregister SREG des Mikrocontrollers gesetzt sind, erzeugt das Setzen von TWINT einen Interrupt. Ansonsten kann der Zustand dieses Bits durch Polling ausgewertet werden.

Bit 6 – TWEA – das Setzen dieses Bits durch die Software generiert ein ACK nach jedem empfangenen Byte und zusätzlich im Slave Betrieb nach der Erkennung der eigenen Adresse.

Bit 5 – TWSTA – durch das Setzen dieses Bits von der Software im Master Modus wird eine START-Sequenz generiert und die Kommunikation wird initiiert.

Bit 4 – TWSTO – wenn der Master dieses Bit setzt, wird eine STOP-Sequenz generiert und die Kommunikation wird beendet.

Bit 3 – TWWC – ist ein Kollisionsflag, das von der Hardware beim Versuch gesetzt wird, in das Datenregister zu schreiben während das Bit TWINT auf „0“ steht. Wenn dieses Bit „0“ ist, kann das bedeuten, dass eine Byteübertragung noch nicht vollständig ist.

Bit 2 – TWEN – mit dem Setzen dieses Bits wird die TWI Schnittstelle aktiviert und übernimmt die Kontrolle über die dedizierten Anschlüsse SDA und SCL ohne zusätzliche Einstellungen in den I/O-Registern.

Bit 1 – ist reserviert.

Bit 0 – TWIE – mit dem Setzen dieses Bits wird der TWI-Interrupt freigegeben. In der entsprechenden Interrupt Service Routine muss das Bit TWINT zurückgesetzt werden.

Das Statusregister **TWSR** enthält nach jedem TWI Vorgang einen Code in den Bits 7...3, der zeigt, ob die Operation erfolgreich war oder nicht. Dieser Code muss von der Software ausgewertet und für den nächsten Schritt berücksichtigt werden. Die Bits 1..0, TWPS 1..0 dienen zur Prescalerwahl (siehe Tab. 5.6). Um den Statuscode auszuwerten, muss das Statusregister maskiert werden, so dass die oben erwähnten Prescalerbits ausgeblendet sind:

```
#define TWI_STATUS_REGISTER (TWSR &0xF8)
```

Initialisieren der TWI Schnittstelle

Für die Initialisierung der TWI Schnittstelle für den Master Modus muss lediglich die Schnittstelle aktiviert und die Übertragungsrate festgelegt werden:

```
void TWI_Master_Init(unsigned char uctwi_clock)
{
    TWCR = 1 << TWEN;
```

```

TWBR = uctwi_clock;      //Codierung der Taktfrequenz
}
//Die Funktion wird folgendermaßen aufgerufen:
TWI_Master_Init(TWI_MASTER_CLOCK);

```

TWI-Kommunikation

Im Ruhezustand des Busses sind die Leitungen SDA und SCL auf High. Der Master initiiert die Kommunikation mit einer START-Sequenz indem er die SDA-Leitung vor der SCL-Leitung auf Low legt. Mit dem Aufruf der Funktion TWI_Master_Start()

```

void TWI_Master_Start(void)
{
    TWCR = (1 << TWINT) | (1 << TWSTA) | (1 << TWEN);
    //Start: SDA-Leitung vor der SCL-Leitung auf Low setzen
    while(!(TWCR & (1 << TWINT))); //warten bis SDA gesetzt wird
}

```

wird der Mikrocontroller eine START-Sequenz generieren, sobald der Bus frei ist. Nach dem START gilt der Bus als besetzt. Nach einem erfolgreichen START enthält das Statusregister den Code 0x08 und die Software leitet die Adressierung eines Slaves ein. Der Master überträgt ein Byte, dessen Bit 7...1 die Adresse des gewünschten Slaves bilden und über das Bit0 (Read/Write) wird die Übertragungsrichtung bestimmt. Auf dem neunten Takt bestätigt der adressierte Slave mit ACK die Erkennung der eigenen Adresse. Während der gesamten Übertragung darf sich ein Bit auf der SDA-Leitung nur ändern solange die SCL-Leitung auf Low ist und muss während des gesamten SCL-Pulses einen stabilen Zustand einnehmen, so wie in Abb. 5.25 skizziert. Diese Anforderung ist in der hardwareseitigen Sicherungsschicht des Protokolls implementiert und muss durch die Anwendersoftware nicht mehr implementiert werden.

Master als Sender

Der Master ist im Sendermodus, wenn er in der Adressierungsphase das Read/Write Bit auf „0“ setzt. Der zeitliche Verlauf einer beispielhaften Datenübertragung von Master zu Slave ist in Abb. 5.25a dargestellt. Nach erfolgreicher Übermittlung der Slaveadresse enthält das Statusregister den Code 0x18, und mit der Funktion TWI_Master_Transmit kann jeweils ein Byte gesendet werden. Um Kollisionen zu vermeiden (dadurch wäre das Bit **TWWC** gesetzt), wird zuerst das Byte **ucdata** im Datenregister geschrieben und danach das Bit **TWINT** gelöscht.

```

void TWI_Master_Transmit(unsigned char uedata)
{
    TWDR = uedata;
    TWCR = (1 << TWINT) | (1 << TWEN);
    while(!(TWCR & (1 << TWINT)));
}

```

```

/*die Hardware setzt das Bit TWINT wenn das Byte uedata vollständig
übertragen wurde*/
}

```

Der adressierte Slave bestätigt jedes empfangene Byte mit ACK, was zum Code 0x28 im Statusregister führt. Nach dem letzten Byte beendet der Master die Kommunikation mit dem Slave durch eine STOP-Sequenz. Hardwaremäßig bedeutet diese Sequenz das Setzen auf High der SCL- vor der SDA-Leitung, was softwaremäßig mit dem Aufruf der Funktion `TWI_Master_Stop()` zu realisieren ist:

```

void TWI_Master_Stop(void)
{
    TWCR = (1 << TWINT) | (1 << TWEN) | (1 << TWSTO);
}

```

Nach dem STOP gilt der Bus nach einer spezifizierten Verzögerung als frei und kann von einem anderen Master in Anspruch genommen werden.

Master als Empfänger

Weil die meisten I²C Bausteine eine komplexe Struktur aufweisen, kann ein Master auf die gewünschten Daten nur indirekt über Register oder Funktionen zugreifen. Bevor er in den Empfänger Modus schaltet, muss er als Sender wie im letzten Abschnitt beschrieben den Schlüssel zu den Daten übermitteln. Das kann für einen Speicherbaustein die Adresse des gewünschten Bytes sein. Um die Kontrolle des Busses nicht zu verlieren, generiert der Master eine so genannte RESTART-Sequenz durch einen erneuten START und besetzt weiterhin den Bus. Es folgt eine neue Adressierung des Slaves, diesmal mit dem Read/Write Bit gesetzt. Wenn der Slave den Empfang mit ACK bestätigt, dann enthält das Statusregister den Code 0x40. Ab jetzt ändert sich die Übertragungsrichtung, der Slave sendet und der Master empfängt die Daten. Der Master generiert weiterhin die Taktfrequenz und muss auf dem neunten Takt eines Bytes mit ACK (Code 0x50 im Statusregister) oder mit NACK (Code 0x58 im Statusregister) antworten. Das letzte geforderte Byte wird mit NACK quittiert, was dem Slave das Ende der Kommunikation signalisiert. Zum Schluss wird noch eine STOP-Sequenz generiert. Dieses Verfahren ist in Abb. 5.25b graphisch dargestellt. Der Master initiiert die Kommunikation, adressiert den Slave im Write-Modus und überträgt einen Befehlscode. Nach dem RESTART und erneuter Adressierung im Read-Modus schiebt der Slave ein Byte auf die SDA-Leitung heraus. Eine RESTART-Sequenz wird mit beiden Busleitungen auf High generiert, bevor der Bus als frei betrachtet werden kann.

Im Folgenden werden zwei Empfangsfunktionen vorgestellt, je nachdem ob der Master mit ACK oder NACK den Byteempfang bestätigt:

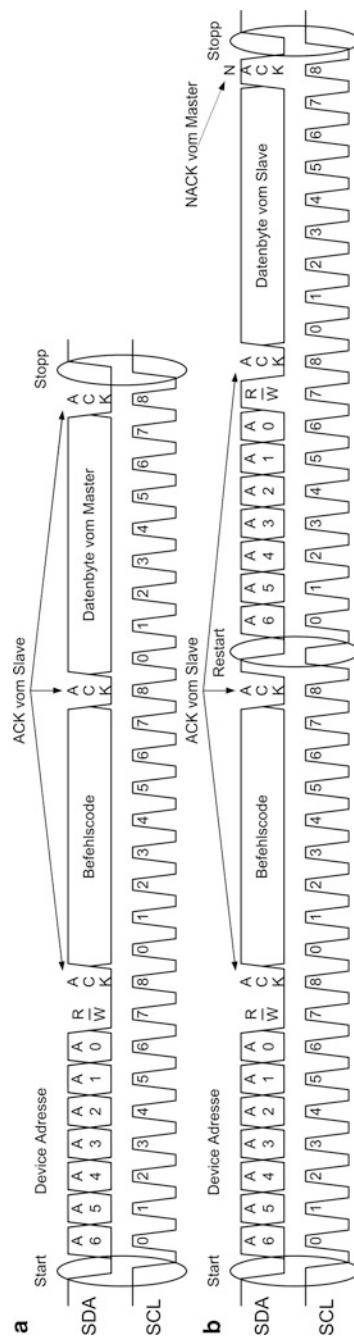


Abb. 5.25 TWI Kommunikation

```

//der Master antwortet mit ACK auf das empfangene Byte
unsigned char TWI_Master_Read_Ack(void)
{
    TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWEA) ;
    while (!(TWCR & (1 << TWINT)));
    return TWDR;
}

//der Master antwortet mit NACK auf das empfangene Byte
unsigned char TWI_Master_Read_NAck(void)
{
    TWCR = (1<<TWINT) | (1<<TWEN) ;
    while (!(TWCR & (1 << TWINT)));
    return TWDR;
}

```

Beide Funktionen geben an die Anrufstelle das empfangene Byte zurück. Zahlreiche Beispiele für Schreib-Lese-Funktionen sind in den folgenden Kapiteln über Sensorbeispiele vorgestellt.

5.2.4 Anbindung der seriellen Schnittstelle an USB

Für die Kommunikation des Prozessors mit einem PC kann die serielle Schnittstelle unter Verwendung von USB genutzt werden. Hierzu ist die Umsetzung des UART auf USB notwendig. Die gängigste Variante dafür ist die Verwendung eines Bausteins von FTDI.

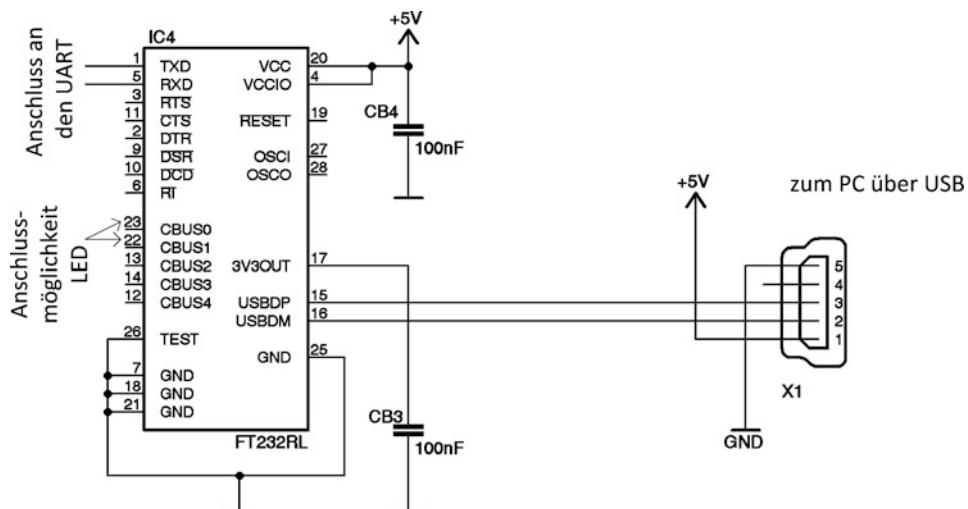


Abb. 5.26 Einfachste Beschaltung des FT232R

Aus der vielfältigen Familie von USB Controllern wurde hier der FT232R Controller gewählt, der auf dem PC durch einen virtuellen COM-Treiber angesprochen wird und bis zu 12 Mbit/s Datenrate unterstützt. Damit wird die serielle Schnittstelle direkt als solche im PC angesprochen, beispielsweise über ein Terminal-Programm oder über jede andere Software, die auf die COM Schnittstellen zugreifen kann. Die Treiber können kostenlos bei FTDI (<http://www.ftdichip.com/Products/ICs/FT232R.htm>) abgerufen werden.

Abb. 5.26 zeigt die einfachste mögliche Beschaltung des FT232R. Darüber hinaus lassen sich noch LEDs anschließen, die die Kommunikationszustände anzeigen, sowie weitere digitale Ein/Ausgänge schalten. Außerdem verfügt der Chip über die Möglichkeit, einen Hardwarehandshake durchzuführen.

5.3 Bussysteme für embedded Schaltungen

Neben der Vernetzung mit I²C und SPI setzen sich auch immer mehr Bussysteme durch, die selbst auf kleineren Mikrocontrollerschaltungen preiswert realisierbar sind. Im Folgenden werden CAN und MODBUS kurz behandelt, darüber hinaus sei auf die umfangreiche Literatur verwiesen, die in den jeweiligen Abschnitten zitiert ist.

5.3.1 CAN

5.3.1.1 CAN Grundlagen kompakt

CAN (Controller Area Network) ist in der Automobilindustrie der Klassiker unter den Sensornetzwerken und hat sich aufgrund seiner Einfachheit, seiner angemessen hohen Datenraten (bis 1 Mbit/s bei bis 40 m Länge) und seiner Robustheit nicht nur im Fahrzeug zum Standard etabliert sondern speziell auch in der Automatisierungstechnik. Ursprünglich von Bosch spezifiziert [4] ist er nun als ISO 11898 (Teile 1–6) international genormt [5]. Teil 1 gibt die ursprüngliche Spezifikation wieder, Teil 2 und Teil 3 legen den physical Layer für High-Speed bzw. Low-Speed CAN fest und Teil 4 ist die Erweiterung auf zeitgesteuerte synchrone Kommunikation (Time Triggered CAN). Die Teile 5 und 6 beziehen sich auf den low-power mode und einen selektiven Wakeup. Einen Überblick über weitere Normen im Umfeld von CAN geben [11] und [12]. CAN wird in der Regel als symmetrische Zweidrahtleitung in Wired-AND-Technik ausgelegt, ist aber auch als Eindrahtleitung spezifiziert. Die Datenübertragung ist bitstromorientiert und nutzt Bit-Stuffing nach fünf gleichen Bit (Abschn. 5.1.2.1), um lange 0- und 1-Folgen zu verhindern.

CAN nutzt die CSMA/CA Technik zur Kollisionsvermeidung. Solange ein Teilnehmer sendet, verhalten sich die anderen Teilnehmer ruhig. Zunächst sendet CAN nach einem Startbit (dominant = 0) einen 11 Bit Nachrichtenidentifier (CAN2.0 A), ist alsobotschaftenorientiert. In der CAN2.0 B Spezifikation ist der Identifier auf 29 Bit erweiterbar. Jeder Identifier ist zwingend und eindeutig an eine Botschaft eines Teilnehmers gebunden. Da alle Teilnehmer am Bus „mithören“, unternimmt keiner einen Sendeversuch, solange

ein anderer sendet. Unternehmen zwei Teilnehmer gleichzeitig einen Sendeversuch, wird derjenige Teilnehmer das Problem als erstes spätestens dann bemerken, wenn er ein rezeptives (= 1) Bit gesendet hat (logisch 1), auf dem Bus aber ein dominantes Bit anliegt (logisch 0). Er muss dann sofort die Sendung unterbrechen. Konsequenterweise setzt sich die Botschaft mit der niedrigeren Adresse ungestört durch. Da die Botschaften relativ kurz sind, ist die Chance hoch, nach kurzer Zeit Übertragungskapazität zu bekommen.

In Abb. 5.27 ist ein entsprechender Verlauf zu sehen, hier versuchen vier Stationen Botschaften mit den CAN IDs 0x64C, 0x658, 0x704, 0x6DC Botschaften zu senden. Sobald eine Station feststellt, dass das von ihr gesendete Signal auf dem Bus durch ein dominantes Bit übersteuert wurde, bricht sie die Übermittlung ab. Man nennt die Übertragungsphase der Identifier auch Konkurrenzphase, weil sich dort entscheidet, welche Botschaft sich auf dem Bus durchsetzt.

CAN nutzt einen Protokollrahmen (Abb. 5.28) mit bis zu 64 Bit Nutzdaten und einem Header von 19 Bit Länge bei der CAN2.0A-Spezifikation bzw. 37 Bit bei CAN2.0B und einen Trailer von 25 Bit Länge. Dazu kommen mindestens 3 Bit Wartezeit, nach der ein Knoten nach einer erfolgten Sendung selbst einen Sendeversuch unternehmen darf (Interframe Space, IFS), und insgesamt bis zu 15 Stuffbits. Zwischen Header und Trailer

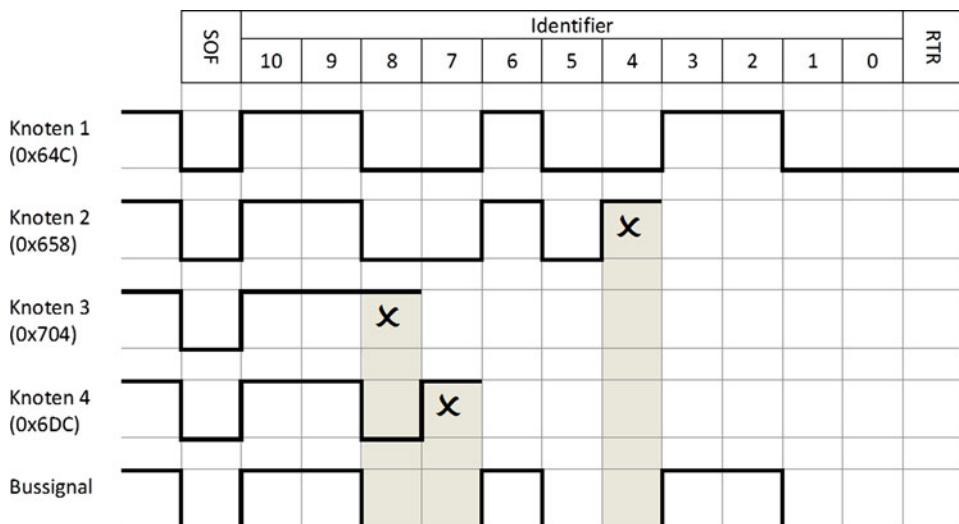


Abb. 5.27 CSMA/CA Konkurrenzperiode

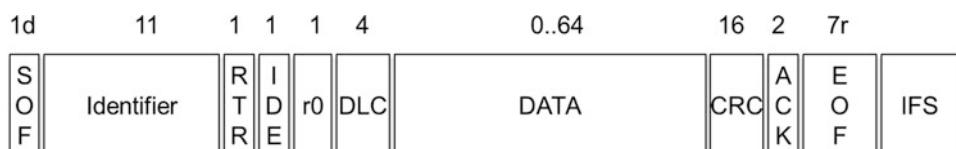


Abb. 5.28 CAN-Frame

ist Platz für maximal 8 Datenbyte. Im ungünstigsten Fall liegt die Protokolleffizienz, also das Verhältnis zwischen den Nutzdaten und der gesamten Paketlänge bei knapp 50 %. Bei 500 kbit/s Bitrate beträgt die Übertragungsdauer (Latenzzeit) 225 µs beim kurzen und 260 µs beim langen Header, was einer Nutzdatenrate von 34 kByte/s bzw. 30 kByte/s entspricht. Der Header besteht neben dem Message-Identifier noch aus:

- SOF: Start of Frame: Dominantes Bit, das den Beginn einer Nachricht signalisiert, wenn der Bus vorher mindestens 11 Bit auf rezessivem Pegel gelegen hat (siehe unten Acknowledge Delimiter, EOF und IFS)
- RTR: Remote Transmission Request: Ein CAN-Teilnehmer, der eine bestimmte Botschaft erwartet, sendet ein Frame mit der Botschaftskennung dieser Nachricht, mit rezessivem (1) RTR-Bit aber ohne Datenfeld. Der Teilnehmer, der diese Botschaft normalerweise generiert, erkennt das Remote Frame und komplettiert daraufhin die Botschaft mit den entsprechenden Daten. Auf diese Weise lässt sich ein einfaches Client-Server-Modell etablieren.
- IDE: Identifier Extension: Ist es rezessiv, zeigt es an, dass der Identifier auf 29 Bit erweitert ist.
- r0: Reserviert (wird dominant gesetzt).
- DLC: Data Length Code: zeigt die Länge des folgenden Datenfelds an.
- DATA: Nutzdaten: können 0 bis 8 Byte umfassen.
- CRC: CRC-Prüfsumme.
- ACK: Acknowledge: Besteht aus dem vom Sender rezessiv gesendeten Acknowledge Slot und einem festgelegten rezessiven Begrenzungsbetrag (Acknowledge Delimiter). Ein Busteilnehmer, der die Nachricht konsistent mit der Prüfsumme empfangen hat, setzt den Acknowledge-Slot auf dominanten Pegel.
- EOF: End of Frame: Die Übertragung wird mit 7 rezessiven Bit abgeschlossen.

Die ISO 11898 spezifiziert darüber hinaus ein Fehlerbehandlungsverfahren, welches die Möglichkeit vorsieht, dass sich ein Knoten selbst vom Bus abschaltet, wenn er die Ursache von gehäuften Fehlern ist. Hierzu sendet ein Knoten, der einen Fehler detektiert, eine Nachricht mit sechs dominanten Bit (Error Frame), die damit die Bit-Stuffing Regel verletzen und von jedem anderen Knoten erkannt wird. Diese antworten darauf ebenfalls mit einem Error Frame. Ein Knoten, der erkennt, dass während einer von ihm initiierten Übertragung ein Fehler aufgetreten ist, erhöht einen Fehlerzähler (Sendefehlerzähler). Ein Knoten, der erkennt, dass er eine fehlerhafte Botschaft empfangen hat, erhöht einen anderen Fehlerzähler (Empfangsfehlerzähler). Knoten, die als erste einen Fehler entdecken, bekommen mehr „Fehlerpunkte“, weil die Gefahr besteht, dass die Entdeckung selbst ein Fehler war. Korrekt gesendete bzw. empfangene Nachrichten erniedrigen den Zähler wieder.

Ein Knoten, dessen Fehlerzähler > 127 ist darf keine dominanten Fehlerbotschaften mehr versenden, sondern nur noch mit sechs rezessiven Bit den Fehler anzeigen, stört aber damit nicht den Netzwerkverkehr. Bei einem Zählerstand von 255 verliert er die

Berechtigung, am Verkehr teilzunehmen. So werden Knoten abgeschaltet, die zu häufig Fehler im Bus generieren oder fehlerhaft detektieren.

Ist ein Knoten überlastet, darf er in einer Sendepause (Interframe Space) ein Overload Frame verschicken, das genauso aussieht wie ein Error Frame, aber durch seine Lage im Interframe Space davon unterschieden werden kann. Dadurch werden andere Knoten am Senden gehindert.

Das CAN-Protokoll entspricht der Sicherungsschicht (Schicht 2) im OSI-Schichtenmodell. Darüber definiert die ISO 15765-2 ein Transportprotokoll auf Schicht 4 im OSI-Schichtenmodell, das beispielsweise in der Diagnose eingesetzt wird. Eine ausführliche Beschreibung findet sich beispielsweise bei [11] und [12].

5.3.1.2 CAN Timing

Ein CAN Bit besteht aus vier Segmenten und wird in TQ (Time Quantum) gemessen:

- Das Sync Segment dient dazu, allen Knoten die notwendige Zeit zu geben, den Beginn des Bits zu erkennen. Es ist ein TQ lang
- Das Propagation Segment gleicht Verzögerungen durch Laufzeiteffekte aus der Leitung aus. Es ist 1..8 TQ lang
- Die Phase Segments PS1 und PS2 bilden das eigentliche Bit ab. PS1 ist zwischen 1 und 8 TQ lang, PS2 ist mindestens 2 TQ und bis 8 TQ lang. Der Sample Point, also der Zeitpunkt, an dem der Wert des Bits gemessen wird, liegt am Übergang zwischen PS1 und PS2. Damit ist PS zwei auch für die Verarbeitung zuständig (Information Processing Time).

Abb. 5.29 zeigt den Aufbau eines Bit graphisch an. Die gesamte Bitzeit ist also

$$t_{\text{Bit}} = t_{\text{SyncSeg}} + t_{\text{PropSeg}} + t_{\text{PS1}} + t_{\text{PS2}} \quad (5.14)$$

Hieraus ergibt sich dann die nominelle Bitrate.

$$\text{NBR} = \frac{1}{t_{\text{Bit}}} \quad (5.15)$$

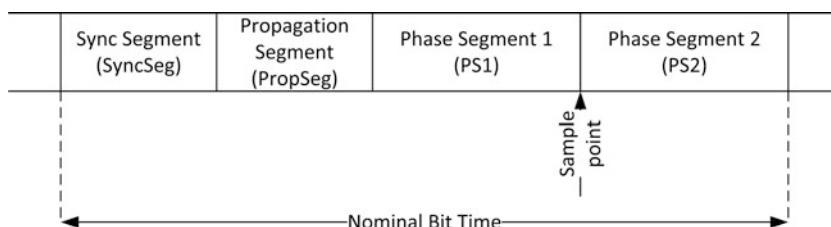


Abb. 5.29 Bit-Timing im CAN

Alle CAN-Controller in einem Netzwerk müssen über dieselbe nominelle Bitrate verfügen, damit die Bits sicher gelesen werden können, über eine PLL werden Schwankungen der jeweiligen Oszillatoren ausgeglichen.

5.3.1.3 Nutzung von CAN mit Prozessoren der AVR Familie

Die AVR Familie besitzt mit dem AT90CAN128 einen Prozessor, der direkt einen CAN Controller an Bord hat. Lediglich ein Baustein zur Anpassung der elektrischen Übertragung ist notwendig, hier bietet sich beispielsweise ein PCA82C251 oder ein TJA1050 jeweils von NXP an. Um andere Prozessoren über CAN zur Kommunikation zu bringen, wird gerne ein MCP2515 von Microchip verwendet. Dieser benötigt ebenfalls einen physikalischen Transceiver, beide Bausteine sind als Adapterplatine bereits ab ca. 2 € zu bekommen. In diesem Fall erfolgt die Kommunikation mit dem Prozessor über SPI. Im folgenden Abschnitt werden nur die groben Grundlagen der Verwendung des MCP2515 kurz beschrieben. Im Netz sind fertige CAN-Treiber verfügbar, unter anderem die freie Software des Roboterclub Aachen, die über entsprechende Präcompilerschalter unter anderem beide Varianten und weiterhin den CAN-Controller SJA1000 unterstützt. Die hier verwendeten Codebeispiele beziehen sich im gesamten Abschnitt auf diese Quelle [10].

In Abb. 5.30 ist ein CAN Netzwerk zu sehen, in dem drei Teilnehmer in unterschiedlicher Beschaltung auf den CAN Bus zugreifen. Hier sieht man die verschiedenen Beschaltungen in Abhängigkeit der verwendeten Bausteine.

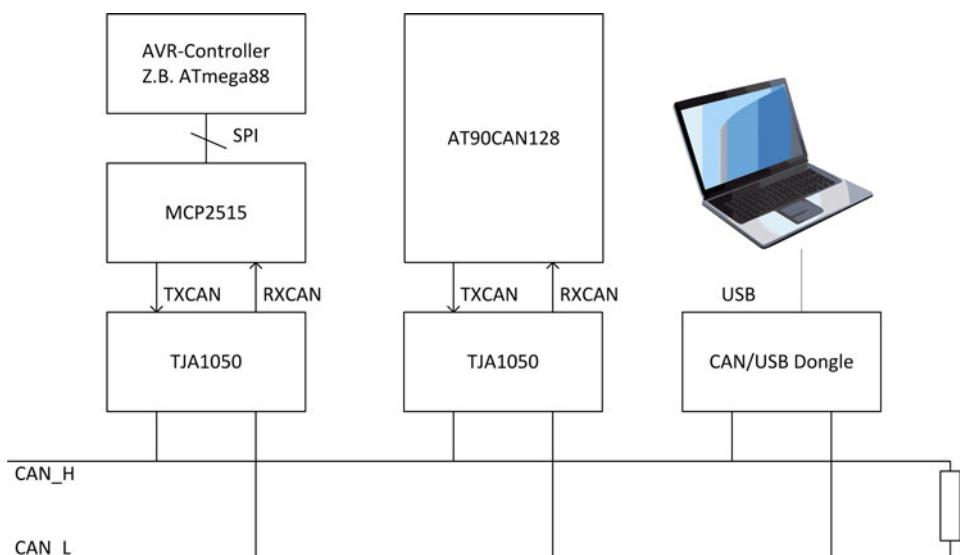


Abb. 5.30 CAN Netzwerk mit drei Teilnehmern

MCP2515

Der CAN Controller MCP 2515 von Microchip [13] ist einer der erfolgreichsten CAN Controller auf dem Markt. Er ist über SPI anschließbar und liefert am Ausgang Signale für CAN Receive (CAN Rx) und Transmit (CAN Tx), die über einen Transceiver elektrisch an das Bussystem angeschlossen werden müssen (hier: TJA1050). Eine Übersicht über die Architektur ist in Abb. 5.31 zu sehen.

Der Controller besitzt drei Transmit-Puffer, zwei Receive-Puffer, zwei Empfangsmasken und insgesamt sechs Empfangsfilter. Er führt sämtliche Fehler- und Überlastbehandlungen selbst durch. Eine vollständige Beschreibung würde den Rahmen dieses Buches sprengen, daher ist auf das sehr umfangreiche und instruktive Datenblatt [13] verwiesen. Die Transmit-Puffer enthalten die Botschaften, den CAN-Identifier und weitere Steuerdaten. Sie sind als Register über SPI ansprechbar. Der Baustein versendet mit dem CS-MA/CA Verfahren die Inhalte dieser Puffer, die untereinander priorisierbar sind. Insofern ist der eigentliche Prozessor vollständig entlastet, sobald die Botschaften im Transmit-Puffer abgeliefert sind. Die Protocol-Engine baut das CAN-Frame zusammen und erleidet das relativ komplexe Timing und die Prüfsummenbildung.

Auf der Empfangsseite nimmt ein Message Assembly Buffer (MAB) eingehende Nachrichten aus der Protocol Engine ab und verteilt jede Nachricht, die die Empfangsfilter und -masken passiert in einen der beiden Receive-Puffer RXB0 und RXB1. RXB0 sind zwei Filter und eine Maske zugeordnet, RXB1 die zweite Maske und vier Filter. Wird eine Nachricht empfangen, auf deren Identifier die Filterkriterien für beide Puffer zutreffen, wird sie nur in RXB0 gespeichert. Solange eine ungelesene Nachricht im Puffer ist, ist dieser für weitere Nachrichten gesperrt. Kommt allerdings eine Nachricht an, die für RXB0 bestimmt ist obwohl dort bereits eine ungelesene Nachricht gespeichert ist, kann der Controller die alte Nachricht in RXB1 umspeichern und die neue Nachricht empfangen (rollover-Betrieb), dies muss eigens konfiguriert werden. Sobald ein Puffer eine neue Nachricht enthält, kann optional das entsprechende Pin RXB0 oder RXB1 von High

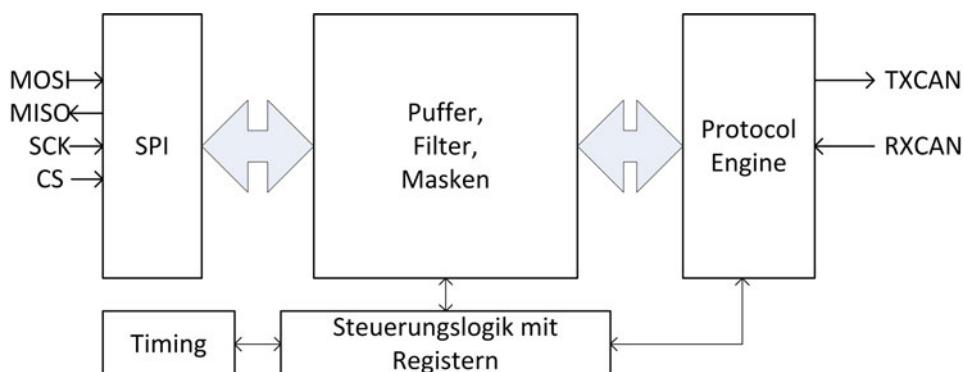


Abb. 5.31 Blockschaltbild_des_MCP_2515. (Nach [13])

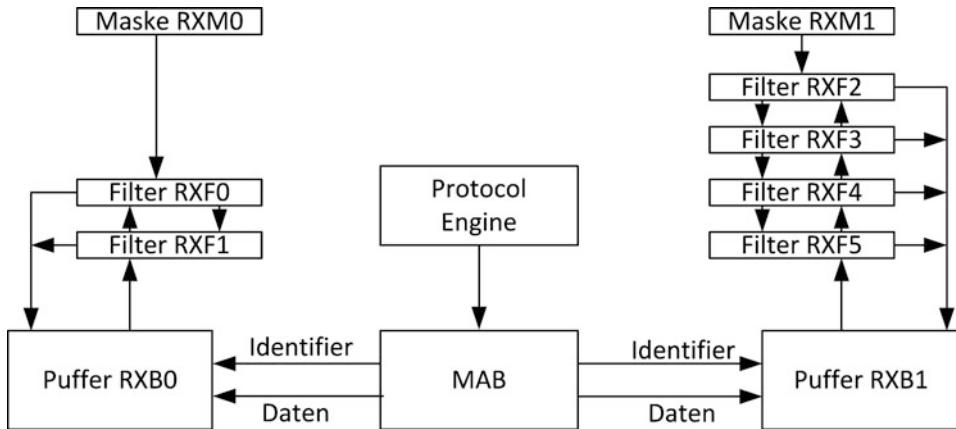


Abb. 5.32 Strukturen der Empfangspuffer im MCP2515. (Nach [13])

auf Low gezogen werden, so dass der MCP2515 beim Prozessor einen Interrupt auslösen kann.

Die Maske steuert dabei jeweils, ob das entsprechende Bit im Filter wirksam ist, das heißt, wenn ein Maskenbit auf 0 sitzt, wird das Bit des Message Identifiers auf jeden Fall akzeptiert, ansonsten wird es nur akzeptiert, wenn es mit dem zugehörigen Filterbit übereinstimmt. Eine Nachricht passiert die Filter, wenn alle von der Maske freigeschalteten Bits des Identifiers akzeptiert werden. So kann man aus einer Mischung aus Masken einzelne Nachrichten und ganze Nachrichtengruppen passieren lassen. Filter und Masken lassen sich über Register beschreiben, ebenso wie die gesamte CAN Konfiguration, die das Timing, das Fehlerverhalten und das Powermanagement steuert. Abb. 5.32 gibt einen Überblick über die Struktur des Empfangssystems.

Initialisierung

Die Initialisierung des MCP2515 erfordert eine genaue Kenntnis des CAN-Timing. Letztlich setzt man abhängig vom verwendeten Quarzoszillator einen Prescaler (BRP), der das Time Quantum bestimmt und legt danach die TQs für die einzelnen Segmente fest.

$$TQ = \frac{2 \cdot BRP}{f_{osc}} \quad (5.16)$$

In [10] erledigt dies die Bibliothek mit einem einfachen Aufruf

```

typedef enum {
    BITRATE_10_KBPS      = 0,
    BITRATE_20_KBPS      = 1,
    BITRATE_50_KBPS      = 2,
    BITRATE_100_KBPS     = 3,
}
  
```

```

BITRATE_125_KBPS      = 4,
BITRATE_250_KBPS      = 5,
BITRATE_500_KBPS      = 6,
BITRATE_1_MBPS        = 7,
} can_bitrate_t;

can_init(BITRATE_250_KBPS);

```

Botschaften senden

Die zentrale Datenstruktur für den Einsatz von CAN ist die CAN-Botschaft. Nicht nur in [10] wird sie für Standard CAN Nachrichten wie folgt definiert:

```

typedef struct
{
    unsigned int id;           //ID der Nachricht (11 Bit)
    struct {
        int rtr : 1;          //Remote-Transmit-Request-Frame?
        } flags;
    unsigned char length;      //Anzahl der Datenbytes
    unsigned char data[8];     //Daten der CAN Nachricht
} can_t;

```

Für extended Identifier Botschaften (22 Bit) ist der Identifier vom Typ `unsigned long` und ein weiteres Flag `int extended : 1` wird zur Flag-Struktur hinzugefügt. Manchmal wird auch ein Zeitstempel (16 Bit genügen meist) mitgespeichert, der allerdings von der Hardware des MCP 2515 nicht unterstützt wird, sondern dann als Botschaftsbestandteil mit versendet werden muss.

```

typedef struct
{
    unsigned long id;          //ID der Nachricht (11 Bit)
    struct {
        int rtr : 1;          //Remote-Transmit-Request-Frame
        int extended : 1;      //Die ID ist extended (29 Bit)
        } flags;
    unsigned char length;       //Anzahl der Datenbytes
    unsigned char data[8];      //Daten der CAN Nachricht
    unsigned long timestamp;    //Zeitstempel
} can_t;

```

Das Senden einer Botschaft läuft bei der Nutzung des MCP2515 wie folgt ab:

- Zunächst prüft man, ob eines der drei Senderegister frei ist, indem man über die SPI Schnittstelle den Code für READ STATUS versendet, dieser lautet 0xA0. Erneutes

Senden irgendeines Wertes (meist 0x00 oder 0xFF) schiebt den aktuellen Status ins SPI Empfangsregister.

- Der Status der drei Senderegister wird in den Bits 2, 4 und 6 (TXBnCNTRL.TXREQ) dargestellt, n bedeutet hier die Nummer des Senderegisters. Ist das jeweilige Bit = 1, so befindet sich noch eine zu sendende Botschaft in dem entsprechenden Register. Ist das Bit = 0 kann die Botschaft in dieses Register geschrieben werden.
- Das Schreiben erfolgt durch Versenden des Load TX Buffer Kommandos (0x40) zusammen mit der Adresse des TX Puffers mit anschließender Übertragung des Identifiers, des Längenbytes, eventuell zusammen mit einem RTR (siehe [13]) und den entsprechenden Daten.
- Anschließend wird mit dem Request to Send (RTS) Kommando der Sendevorgang gestartet. Dieses besteht aus einer 0x80 ver-ODER-t mit der Adresse des TX Puffers (0x01, 0x02 oder 0x04).

In [10] ist dies wie folgt realisiert:

```
can_t mymsg ;
mymsg.length=4;
mymsg.id=0x123;
mymsg.data[0]=0;
mymsg.data[1]=1;
mymsg.data[2]=2;
mymsg.data[3]=3;
can_send_message(&mymsg);
```

Hier wird also eine Nachricht mit dem Identifier 0x123 und einer Länge von vier Byte, des Inhalts 0,1,2,3 verschickt. Eine einfache Sensorschaltung, die alle 100 ms einen Sensorwert verschiickt, muss lediglich in einem 100 ms Task einen Sendebefehl mit den entsprechenden Daten absetzen.

Botschaften empfangen

Das Empfangen von Botschaften kann über mehrere Wege erfolgen:

- Sind die Interruptleitungen RX0BF und RX1BF verdrahtet, löst eine neue Botschaft in einem der beiden Empfangspuffer einen Interrupt aus, der den Inhalt der Register abholen kann
- Verzichtet man auf den Interrupt, kann man über das Read Status Kommando (0xB0) den Empfangsstatus der beiden Puffer abfragen (siehe [13]) und erhält den Empfangsstatus und die Filternummern, über die der Empfang freigegeben wurde (Polling Betrieb).

In beiden Fällen muss anschließend die Nachricht abgeholt werden, indem das Read Rx-Buffer Kommando gesendet wird und anschließend die Pufferinhalte ausgelesen wer-

den. Bei hohen Buslasten können im ersten Fall bei der verwendeten Bibliothek [10] Stacküberläufe auftreten, wenn mehr Nachrichten kommen, als Interrupts abgearbeitet werden können. Dies führt zu einem Totalabsturz des Systems, was die Autoren erfahren durften. Um dies zu vermeiden, muss man die Filterfunktion nutzen, so dass nur noch Botschaften abgeholt werden, die für den eigenen Zweck notwendig sind. Oder man sperrt die Interruptabarbeitung in der ISR, wie in Kap. 3 beschrieben und nimmt in Kauf, dass Botschaften verloren gehen.

Wenn der Prozessor sonst wenig zu tun hat, ist der Polling Betrieb die einfachste Form des Botschaftenempfangs:

```
char res;
if (can_check_message());
{
    res=can_get_message(&recmsg); //FALSE falls keine Nachricht vorliegt,
        //ansonsten der Filtercode, über den die Nachricht
        //akzeptiert wurde
    if (res)
    {
        //Hier kann man die Nachricht verarbeiten
    }
}
```

Die einfachste Art, einen Filterbetrieb zu realisieren, ist, die Filter statisch vorzubereiten. In der zitierten Bibliothek gibt es dazu die Möglichkeit, alle Filterwerte und die der beiden Masken hintereinander in ein Array zu schreiben und dieses zur Programmierung der Filter zu nutzen. Im folgenden Code werden zwei Filter genutzt, die Botschaften mit den IDs 0x123 und 0x789 durchlassen, durch Setzen der Masken auf 0x7FF sind alle genutzten Bit des Standardidentifiers abgedeckt.

```
const uint8_t can_filter[] PROGMEM =
{
// Gruppe 0
    MCP2515_FILTER(0x123),      //Filter 0
    MCP2515_FILTER(0x0),         //Filter 1

//Gruppe 1
    MCP2515_FILTER(0x789),      //Filter 2
    MCP2515_FILTER(0x0),         //Filter 3
    MCP2515_FILTER(0x0),         //Filter 4
    MCP2515_FILTER(0x0),         //Filter 5

    MCP2515_FILTER(0x7ff),       //Maske 0 (fuer Gruppe 0)
    MCP2515_FILTER(0x7ff),       //Maske 1 (fuer Gruppe 1)
};

can_static_filter(can_filter); //Setzt den Filter
```

Die Macros `MCP2515_FILTER` helfen hier nur, die Bytes der Identifier richtig zu verteilen (die drei MSB des Identifiers müssen in einem eigenen Byte stehen). Die Funktion `can_static_filter()` erfordert, dass dieses Array im Flash steht (PROGMEM, siehe Abschn. 3.12).

Etwas mehr Programmcode erfordert das Setzen einzelner Filter zur Programmlaufzeit, da hier für die Programmierung mehr Fallunterscheidungen notwendig sind. Hierzu steht die Funktion `can_set_filter()` zur Verfügung. Sie wird, wie im folgenden Beispiel gezeigt, mit der Nummer des Filters und einem Pointer auf eine Struktur `can_filter_t` aufgerufen. Statische und dynamische Filterung arbeiten auch zusammen.

```
can_filter_t cf;
(...)

cf.id=0x654;
cf.mask=0x7ff;
can_set_filter(3,&cf);
```

Die Empfangsfunktion `can_get_message()` liefert die Nummer des Filters von 1...6, während die zitierte Filterfunktion die Filter von 0...5 nummeriert. Dies muss gegebenenfalls berücksichtigt werden.

5.3.2 MODBUS

Der MODBUS Standard spezifiziert die Anwendungsschicht (die 7. Schicht des OSI-Modells) des Kommunikationsprotokolls zwischen vernetzten Geräten [14]. Die Kommunikation kann entweder über TCP/IP, oder über eine serielle, asynchrone Schnittstelle wie TIA¹⁶/EIA¹⁷-232 oder TIA/EIA-485 stattfinden [15].

Für das serielle Protokoll definiert MODBUS die Sicherungsschicht (Data Link Layer) des OSI-Modells. Es wird ein Master-Slave-Bus spezifiziert mit bis zu 247 Slaves. Nur der Master kann eine Kommunikationssitzung mit einem Slave initiieren, oder alle Slaves mit einem Rundruf über die globale Adresse 0x00 ansprechen. Der Bus kann auch als multi-Master betrieben werden unter der Bedingung, dass zu jedem Zeitpunkt nur ein Master aktiv ist. Der Datentransfer zwischen einem Master und einem nahegelegenen Slave (point-to-point Verbindung) bei niedrigen Übertragungsrate kann über TIA/EIA-232 stattfinden, ansonsten über TIA/EIA-485.

5.3.2.1 TIA/EIA-485 als Bitübertragungsschicht für MODBUS

Der industrielle Standard TIA/EIA-485, bekannt auch als RS-485, ist als Nachfolger von RS¹⁸-232 (offiziell TIA/EIA-232) entstanden, um höhere Datenraten (<50 Mbit/s) zu er-

¹⁶ TIA – Telecommunications Industry Association.

¹⁷ EIA – Electronic Industry Alliance.

¹⁸ RS – Radio Sector.

reichen, längere Distanzen ($< 1200 \text{ m}$) zu überbrücken, unempfindlicher gegenüber der Störungen zu sein und das Verbinden mehreren Geräten zu ermöglichen. RS-485 ist eine serielle Schnittstelle, die die erste Schicht des OSI-Modells implementiert. Die Bitübertragung findet im Gegenbetrieb (Vollduplex) oder Wechselbetrieb (Halbduplex) statt und ist symmetrisch und asynchron. Die symmetrische Übertragung benutzt zwei Signalleitungen die verdrillt, ev. auch geschirmt sein können. Es werden die Spannungen $U_A(t)$ und $U_B(t)$, mit $U_B(t) = -U_A(t)$ übertragen. Der Empfänger wertet die Spannung $U_{AB} = U_A - U_B$ aus und rekonstruiert sowohl den Bittakt, als auch die Bitfolge. Die Amplitude der Differenzialspannung U_{AB} muss größer 200 mV sein. Eine Störspannung $U_q(t)$ wirkt sich während der Übertragung additiv auf die zwei Signalspannungen. Die symmetrische Übertragung unterdrückt weitgehend die Gleichtaktstörungen (Abb. 5.33), die auf den Signalleitungen auftreten, und verbessert die Bitfeherrate.

RS-485 ermöglicht theoretisch die Verbindung mehrerer Geräte oder Sensoren, die kein gemeinsames Bezugspotenzial (Masse) haben. In der Praxis wird das Verbinden der Massen über eine elektrische Leitung empfohlen. Es wird ebenso empfohlen, dass alle Ge-

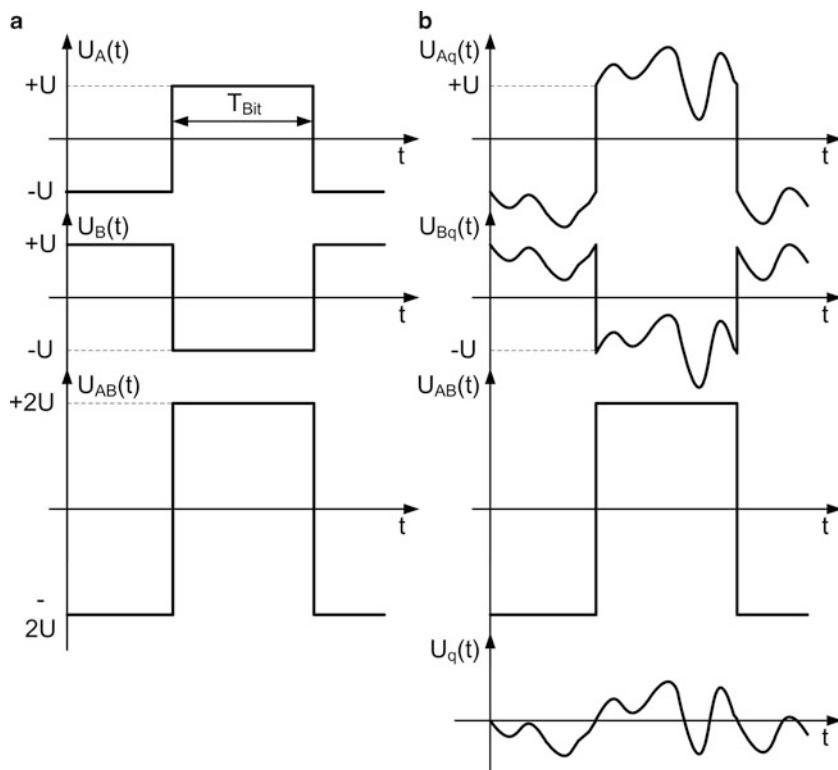
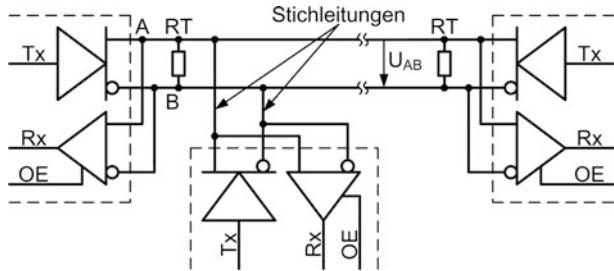


Abb. 5.33 a Störungsfreie symmetrische Übertragung; b symmetrische Übertragung mit Gleichtaktstörungen

Abb. 5.34 MODBUS Netzwerktopologie



räte direkt an den Signalbus angeschlossen werden (Daisy-Chain Topologie), oder mittels kurzen Stichleitungen wie in der Abb. 5.34. Der Datenbus soll mit Abschlusswiderständen (RT) terminiert werden. Diese Widerstände entsprechen dem Wellenwiderstand der benutzten Leitungen. Sie verhindern, dass elektrische Reflexionen am Leitungsende entstehen, die zu einer Verfälschung der Daten führen können. Weitere Schutzmaßnahmen gegen Überspannung, Leerlauf und Kurzschluss sind dem [16] zu entnehmen.

5.3.2.2 MODBUS Kommunikation

Die MODBUS-Übertragung ist Datenwort-orientiert und kann im Remote Terminal Unit- oder ASCII-Modus stattfinden.

Remote Terminal Unit Übertragung

Im RTU-Modus besteht eine MODBUS-Nachricht aus der Adresse des gezielten Slaves, einem Befehlscode, 0 bis 252 Datenbytes und einer CRC¹⁹-Prüfsumme. Jedem Slave aus dem Netzwerk wird eine 1-Byte Adresse aus dem Bereich 1...247 zugewiesen. Die Adresse 0 ist für den Rundruf reserviert. Der Master hat keine Adresse. Über einen gültigen Befehlscode aus dem Bereich 1...128 wird dem Slave die Aktion, die durchgeführt werden soll, übermittelt. Der Nachricht wird eine 16-Bit CRC-Prüfsumme angehängt die, neben der Paritätsprüfung, zur Feststellung der Datenintegrität dient.

Der Datenrahmen eines Datenwortes ist im RTU-Modus immer 11 Bit groß und ist ähnlich wie der der UART-Schnittstelle (Abschn. 5.2.1) aufgebaut. Der Pulsrahmen in Anfangseinstellung beginnt mit einem Startbit, gefolgt von 8 Datenbits, einem Paritätsbit und einem Stopppbit. Die Übertragung des Datenbytes beginnt immer mit dem niederwertigsten Bit zuerst. In der Standard-Einstellung wird mit gerader Parität gerechnet. Die MODBUS-Geräte sollen auch auf ungerade Parität eingestellt werden können. Soll das Paritätsbit wegfallen, muss der Datenrahmen mit zwei Stopppbits beendet werden. Das Paritätsbit wird auf gleicher Weise wie in Abschn. 5.2.1 berechnet. Die Baudrate und die Struktur des Datenrahmens müssen bei allen Busteilnehmern eines Netzwerks gleich eingestellt werden, damit die Kommunikation stattfinden kann. Im RTU-Modus kann der Master eine Nachricht senden, wenn vorher auf der Dauer der Übertragungszeit von mindestens 3,5 Bytes keine Busaktivitäten stattfanden. Wenn diese Wartezeit zwischen zwei

¹⁹ CRC –cyclic redundancy check (zyklische Redundanzprüfung).

Nachrichten nicht eingehalten wird, betrachten die Slaves die neue Nachricht als fehlerhafte Fortsetzung der vorigen. Bei der Übertragung einer Nachricht wird zwischen den einzelnen Bytes eine Wartezeit erlaubt, die nicht größer als die Übertragungszeit von 1,5 Bytes sein darf. Nach einem Rundruf wartet der Master keine Antwort ab, verzögert aber das Senden der nächsten Nachricht, um den Slaves die Dekodierung und Ausführung des Befehls zu ermöglichen. Ein direkt adressierter Slave antwortet dem Master mit einer ähnlich aufgebauten Nachricht. Er sendet die eigene Adresse zuerst, um sich zu identifizieren, gefolgt von dem empfangenen Befehlscode oder einem Fehlercode. Der Master prüft ob die Antwort von dem, von ihm adressierten, Slave kommt und die Integrität der Nachricht. Im Fehlerfall kann er die Nachricht noch einmal senden.

ASCII-Übertragung

Im ASCII-Modus wird die, zu übertragende, Grundnachricht wie im RTU-Modus gebildet. Anstatt der CRC-Prüfsumme wird aber eine 1-Byte große Längsparität²⁰-Prüfsumme berechnet und angehängt. Ausführliche Beispiele für die Berechnung der zwei Prüfsummen befinden sich in [15]. In diesem Modus werden jedem Byte der Grundnachricht zwei ASCII-Zeichen („0“, „1“...„9“, „A“...„F“) zugewiesen. Diese Zeichen codieren die hexadezimalen Werte (0...F) des höherwertigen und niedrigen Nibble eines Bytes. Dadurch verdoppelt sich die Anzahl der zu übertragenden Bytes und die Nettodatenrate wird kleiner als beim RTU-Modus. Der Code des höherwertigen Nibble wird zuerst übertragen. Der Datenrahmen eines Datenwortes besteht immer aus 10 Bits und beinhaltet nur sieben Datenbits. Der folgende Programmausschnitt wandelt die Grundnachricht, bestehend aus der Slaveadresse, *uiLength* Datenbytes und die LRC-Prüfsumme, gespeichert in der Variable *ucBasicFrame* in eine ASCII-codierte Nachricht um.

```

uint8_t ucTemp;
for(uint8_t ucI = 0; ucI < (uiLength + 2); ucI++)
{
    //das höherwertige Nibble des aktuellen Bytes wird codiert
    ucTemp = ucBasicFrame[ucI] & 0xF0;
    ucTemp = ucTemp >> 4;
    //wenn der Wert kleiner 10 ist, wird er mit dem ASCII-Code
    //der Ziffer codiert
    if(ucTemp < 0x0A) ucCodedFrame[2 * ucI] = ucTemp + 0x30;
    //ansonsten, mit den Buchstaben von A bis F
    else ucCodedFrame[2 * ucI] = ucTemp + 0x37;
    //das niedrige Nibble des aktuellen Bytes wird codiert
    ucTemp = ucBasicFrame[ucI] & 0x0F;
    if(ucTemp < 0x0A) ucCodedFrame[(2 * ucI) + 1] = ucTemp + 0x30;
    else ucCodedFrame[(2 * ucI) + 1] = ucTemp + 0x37;
}

```

²⁰ En. LRC – longitudinal redundancy check.

Als Startzeichen einer neuen Nachricht wird ein Doppelpunkt Zeichen „;“ gesendet. Die codierte Grundnachricht wird zeichenweise übertragen und mit der Zeichenfolge „CR²¹“ und „LF²²“ beendet. Die Wartezeit zwischen der Übertragung zweier Zeichen

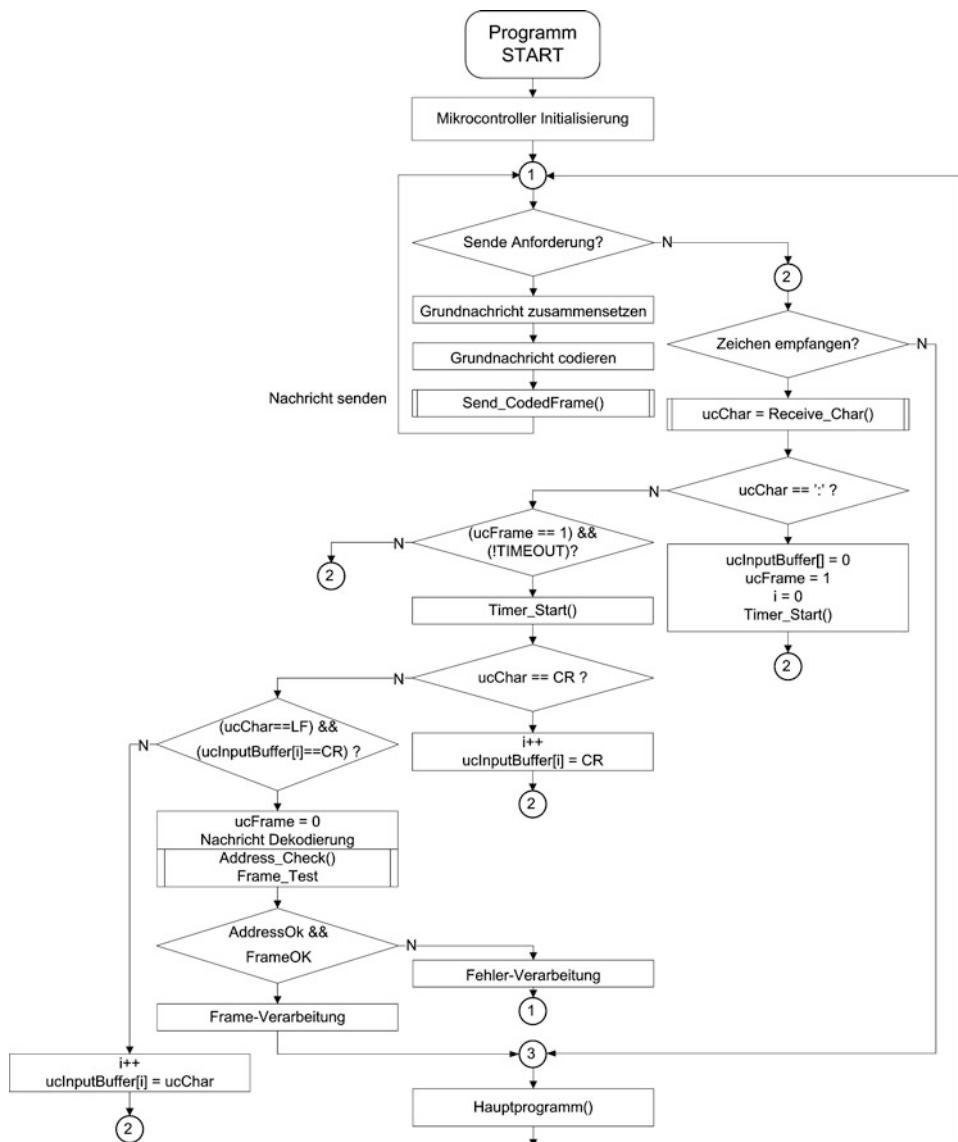


Abb. 5.35 MODBUS Zustandsdiagramm des ASCII-Modus

²¹ CR – carriage return = 0x0D (Wagenrücklauf).

²² LF – line feed = 0x0A (Zeilenumbruch).

soll im Sekundenbereich einstellbar sein. Sowohl der Master als auch der Slave implementieren im ASCII-Modus das gleiche Zustandsdiagramm (siehe Abb. 5.35). Nach dem vollständigen Empfang einer Nachricht wird die Parität der einzelnen Zeichen geprüft. Durch die Dekodierung der Nachricht wird die Grundnachricht wiederhergestellt, die Adresse und die Längsparität-Prüfsumme werden verglichen.

5.4 Funkschnittstellen

Der Einsatz von Mikrocontrollern und Sensoren in Bereichen, wo das Verlegen von Kommunikationsleitungen schwierig ist, begünstigt die Funkübertragung. Gleichermaßen gilt für Netzwerke mit einer großen Anzahl von Knoten oder mit einem sporadischen Datenverkehr, bei denen die leitungsgebundene Vernetzung unwirtschaftlich ist. Die technische Entwicklung der Funkübertragung korreliert mit der Preissenkung der benötigten Hardware. Die Erhöhung der Datenschutzmechanismen und der Übertragungssicherheit ermöglicht den Einzug der Funknetzwerke in das moderne Auto und in Anwendungen wie „Industrie 4.0“, „Smart Home“, „Smart Metering“ oder „Internet der Dinge²³“. Die leitungsgebundene Übertragung ist sicherer und zuverlässiger als die Funkübertragung, die Funknetzwerke können aber leichter erweitert werden.

Der Funkverkehr in Deutschland unterliegt den Bestimmungen der Bundesnetzagentur für Elektrizität, Gas, Telekommunikation, Post und Eisenbahnen. Die Bundesnetzagentur bestimmt für die Funkgeräte mit niedriger Reichweite (SRD²⁴) die maximale Sendeleistung für jeden Frequenzbereich [17]. Diese Begrenzung trägt zur Störsicherheit der Funkkommunikation bei. Für industrielle, wirtschaftliche, medizinische, häusliche oder ähnliche Anwendungen (ISM²⁵) werden unlizenzierte Frequenzbereiche zugewiesen, die mit wenigen Einschränkungen frei benutzt werden können [18]. Das 2,4 GHz ISM-Band, dem in Deutschland der Frequenzbereich 2,4 bis 2,5 GHz zugewiesen ist, ist weltweit definiert und bildet den Auswahlraum für den Übertragungskanal zahlreicher Funkprotokolle.

5.4.1 Multiplexverfahren

Zwei Funksender, die sich in Reichweite befinden, können sich gegenseitig stören. Um das zu vermeiden, muss dafür gesorgt werden, dass die Sender unterschiedliche Trägerfrequenzen benutzen oder nicht gleichzeitig senden.

²³ Englisch IoT = Internet of Things.

²⁴ SRD – Short Range Device.

²⁵ ISM – Industrial, Scientific and Medical.

Frequenzmultiplex (FDMA²⁶)

Beim Frequenzmultiplex-Verfahren wird jedem Sender ein anderer Funkkanal zugewiesen. Die Frequenzbereiche der verwendeten Funkkanäle sollen sich nicht überlappen.

Zeitmultiplex (TDMA²⁷)

Das Zeitmultiplex-Verfahren ermöglicht die mehrfache Nutzung des gleichen Funkkanals. Die Sendezeit wird in Zeitrahmen fester Dauer und der Zeitrahmen in Zeitschlüsse unterteilt. Beim synchronen Verfahren ist die Anzahl der Zeitschlüsse gleich der Senderzahl und jedes Gerät darf nur in dem ihm zugeordneten Zeitschlitz senden. Das asynchrone Verfahren sieht eine dynamische Vergabe der Zeitschlüsse vor, abhängig vom Sendebedarf der Geräte [19].

Codemultiplex (CDMA²⁸)

Das Codemultiplex-Verfahren gehört zu der Spreizbandtechnik. Jedes Bit einer Nachricht wird vor der Modulation mit einem speziellen, binären Spreizcode multipliziert, dessen Einheiten Chips genannt werden. Die Nachricht kann nach der Demodulation nur von dem Gerät dekodiert werden, das den Spreizcode kennt. Dieses Verfahren führt zu einem breitbandigen Spektrum und zu einer erhöhten Informationsredundanz. Diese Redundanz wird genutzt um die Sendeleistung zu senken.

5.4.2 Sensorknoten

Ein einfacher Funkknoten, der, wie in Abb. 5.36 dargestellt, Sensoren und Aktoren beinhaltet, wird als Sensorknoten bezeichnet [20]. Der Mikrocontroller bildet den digitalen Kern des Knotens ab und erfüllt folgende Aufgaben:

- Er initialisiert die Sensoren statisch (einmalig beim Programmstart) oder dynamisch, entsprechend den von außen empfangenen Anforderungen;
- Er liest die Messwerte ein, setzt sie zu einem Datenpaket zusammen und führt eine Kanalcodierung aus;
- Er stellt den Funkkanal, die Sendeleistung und die Empfindlichkeit des Empfängers ein;
- Er steuert die Aktoren;
- Er tauscht digitale Daten mit dem Transceiver aus;
- Er implementiert entsprechend dem gewählten Protokoll die oberen Schichten des ISO OSI-Schichtenmodells.

²⁶ FDMA – Frequency Division Multiplex Access.

²⁷ TDMA – Time Division Multiplex Access.

²⁸ CDMA – Code Division Multiplex Access.

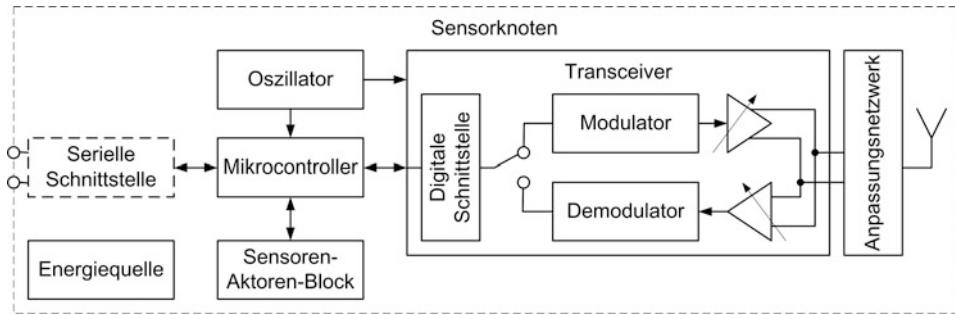


Abb. 5.36 Funkknoten Blockschaltbild

Der Transceiver ist zuständig für die Funkübertragung und implementiert die Schicht der physikalischen Bitübertragung. Er deckt einen definierten Frequenzbereich ab und generiert eine einstellbare Trägerfrequenz. Der Transceiver besteht aus einem Modulator mit einstellbarer Ausgangsleistung, einem Demodulator mit einstellbarer Eingangsempfindlichkeit und einem RF-Schalter. Zusätzlich können einige Transceiver Aufgaben der Sicherungsschicht übernehmen (z. B. [21–23]) wie zum Beispiel:

- Berechnung einer CRC-Prüfsumme für jedes Datenpaket;
- Erkennung und eventuell Korrektur von Übertragungsfehlern;
- automatisches Senden einer Empfangsbestätigung;
- Nachsenden von Datenpaketen die als fehlerhaft gemeldet wurden;
- Ver- und Entschlüsselung der Datenpakete.

Die Transceiver können durch Messungen am Funkkanal folgende Parameter liefern:

- das RSSI²⁹ als analoges oder digitales Signal ist proportional mit der Leistung des empfangenen Signals;
- das LQI³⁰ bildet die Empfangsqualität ab;
- die Energie des Funkkanals; dadurch kann die Sendeleistung auf das Minimum angepasst werden, das die fehlerfreie Übertragung gerade noch sichert.

Das Anpassungsnetzwerk passt den Innenwiderstand des Transceivers auf den Wellenwiderstand der Antenne an. Dadurch kann für jede Anwendung die passende Antenne gewählt werden und der Energieverbrauch wird optimiert.

Die elektrische Energie für die Versorgung der gesamten Elektronik wird meist von einer Batterie geliefert oder wird lokal durch Umwandlung der mechanischen, kinetischen, Solar- oder thermischen Energie erzeugt (energy harvesting).

²⁹ RSSI – Received Signal Strength Indication.

³⁰ LQI – Link Quality Indication.

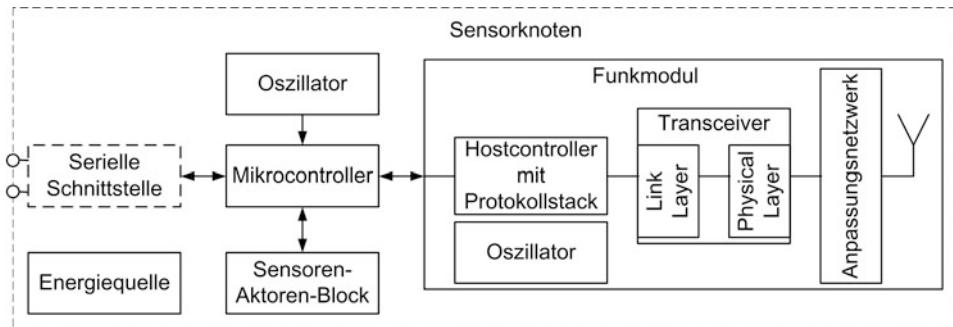


Abb. 5.37 Funkknoten mit Funkmodul

Der in Abb. 5.36 vorgestellte Sensorknoten kann mit einer einfach aufgebauten Software, mit der Anwendungsschicht als Schwerpunkt, für eine point-to-point Kommunikation verwendet werden. Die Entwicklung einer proprietären Software für die Vernetzung mehreren Sensorknoten, bei der die Anwendungsschicht in den Hintergrund rückt, ist aufwendig. Eine flexible und preiswerte Lösung wird in Abb. 5.37 dargestellt. Auf dem Mikrocontroller sind weiterhin die oberen Schichten des ISO/OSI-Modells implementiert. Der Transceiver, als Teil des Funkmoduls, integriert meist die Bitübertragung- und die Sicherung-Schicht, während auf dem Hostcontroller ein Protokoll-Stack mit den weiteren Schichten gespeichert ist. Manche Funkmodule bieten zusätzlich digitale I/O's eventuell auch analoge Eingänge um Sensoren direkt einzulesen, bzw. Aktoren direkt anzusteuern.

5.4.3 Funkprotokolle im 2,4 GHz ISM-Band

5.4.3.1 Bluetooth

Bluetooth ist ein Funkprotokoll basierend auf dem IEEE 802.15.1 Standard [24], der die Bitübertragungsschicht und die Medienzugriffssteuerung des Protokolls spezifiziert. Ziel der Entwicklung war das Ersetzen der kabelgebundenen Übertragung zwischen nah beieinander liegenden Geräten mit einer Funkübertragung. Die Anzahl der Netzwerknoten soll niedrig sein, die Kommunikation soll aber die Qualität der kabelgebundenen Übertragung erreichen, was die Bitrate und die Sicherheit betrifft. Mit den bis jetzt spezifizierten Versionen konnten folgende Bruttoübertragungsraten [25] erreicht werden:

Bluetooth Version	Übertragungsrate
1.1	1 Mbps
2.0	3 Mbps
3.0	54 Mbps
4.0	0,3 Gbps

Bluetooth hat sich als Standard etabliert, der den Weg in Massenprodukte wie Computer aller Art und Handys gefunden hat. Mit der Version 4.0, die als „Low Energy“ (BLE) bezeichnet wurde, wurden die Übertragungsrate und gleichzeitig der Energieverbrauch drastisch reduziert um das Protokoll auch für batteriebetriebene Geräte attraktiv zu machen. Der Aufbau einer Kommunikation bei den älteren Versionen findet in Sekunden statt, was bei langen Kommunikationen in Kauf genommen wird. Eine gesamte Kommunikationssitzung (Aufbau, kurze Übertragung und Abbau) unter der Version 4 soll innerhalb von 3 ms möglich sein [25]. Zur Drucklegung des Buchs ist Bluetooth 5 mit höherer Reichweite und doppelter Datenrate veröffentlicht.

Bitübertragungsschicht

Die Übertragung findet im 2,4 GHz ISM-Band statt, in dem 79 Funkkanäle zwischen 2402 und 2480 MHz mit einer Bandbreite und einem Abstand von 1 MHz definiert sind. Die Daten werden grundsätzlich mit einer gaußschen Frequenzumtastung³¹ moduliert. Um höhere Bitraten (EDR = Enhanced Data Rate) bei gleichem Frequenzkanalabstand zu erreichen, wird die Phasenumtastung³² benutzt. Um Interferenzen mit anderen Protokollen aus dem 2,4 GHz ISM-Band zu vermeiden, benutzt Bluetooth ein adaptives „frequency hopping“ Verfahren. Dieses Frequenzsprungverfahren sieht einen kontinuierlichen Wechsel der Frequenzkanäle vor. Die Frequenzsprünge werden nach einem 625 µs Takt synchronisiert. Dieser Takt wird vom Master, also dem Gerät, das die Verbindung initiiert hat, dem Netz zur Verfügung gestellt. Die pseudozufällige Frequenzsprungfolge wird aus der Adresse des Masters abgeleitet, die den Slaves bei der Herstellung der Verbindung vermittelt wird. Das Verfahren heißt adaptiv, weil der Master die Möglichkeit hat, aus der Frequenzsprungtabelle jene Funkkanäle auszuschließen, auf denen er hohe Funkaktivitäten misst, die zur Störung der Kommunikation im eigenen Netz führen könnten. Der Datenverkehr zwischen Master und Slaves wird mit Hilfe des Zeitmultiplex-Verfahrens geregelt. In der ersten Hälfte einer Taktperiode kann nur der Master beginnen eine Nachricht zu senden, in der zweiten Hälfte nur der angesprochene Slave. Bei längeren Übertragungen wird spätestens nach 5 Taktperioden der Funkkanal gewechselt. Es können bis zu 1600 Frequenzsprünge in der Sekunde stattfinden.

Die Geräte werden in drei Leistungsklassen hergestellt:

Klasse 1 +20 dBm (100 mW), Reichweite < 100 m;

Klasse 2 +4 dBm (2,5 mW), Reichweite < 50 m;

Klasse 3 0 dBm (1 mW), Reichweite < 10 m.

Durch die Messung der Signalqualität kann die Ausgangsleistung der Umgebung angepasst werden.

³¹ Engl. Frequency Shift Keying (FSK).

³² Engl. Phase Shift Keying (PSK).

Netzwerk Topologien

Bluetoothnetze besitzen eine Sterntopologie. Sie werden auch Piconet genannt und bestehen aus einem Master, bis zu sieben aktiven und 255 inaktiven (oder geparkten) Slaves. Die Anzahl der Slaves ist auf sieben begrenzt, weil sie nach der Verbindungsherstellung über drei Bit adressiert werden und die Adresse „000“ für den Rundruf reserviert ist. Der Master kann den Zustand (aktiv oder inaktiv) eines Slaves ändern. Die Verbindung zwischen Geräten in Reichweite kann ad-hoc stattfinden, also ohne Verabredung. Folgende Regeln gelten beim Aufbau der Netze:

- in jedem Piconetz darf nur ein Master sein;
- ein Gerät darf Teil zweier Piconetze sein;
- ein Gerät darf nur in einem Piconetz Master sein.

Durch die Teilnahme eines Gerätes an zwei Piconetzen entsteht ein Scatternet wie in Abb. 5.38 aber kein Meshnetz weil die Spezifikation des Protokolls das Weiterleiten des Datenflusses von einem Piconetz zum anderen nicht erlaubt. Falls gewünscht, kann der Datenaustausch zwischen Piconetzen auf einer anderen Ebene geregelt werden.

Aufbau eines Piconetzes

Ein Gerät, das eingestellt wurde um ein Piconetz aufzubauen, sucht nach anderen Geräten in Reichweite. Die Suche kann nach beliebigen oder vorab durch ihre Adresse definierten Geräten stattfinden. Der Aufbau eines Piconetzes benötigt mehrere Phasen:

- in der Inquiry-Phase sendet der Master Anforderungen um entdeckbare Geräte zu finden;
- in der Paging-Phase prüft der Master ob das Gerät, das seine Anforderungen beantwortet hat, eine Verbindung akzeptiert;

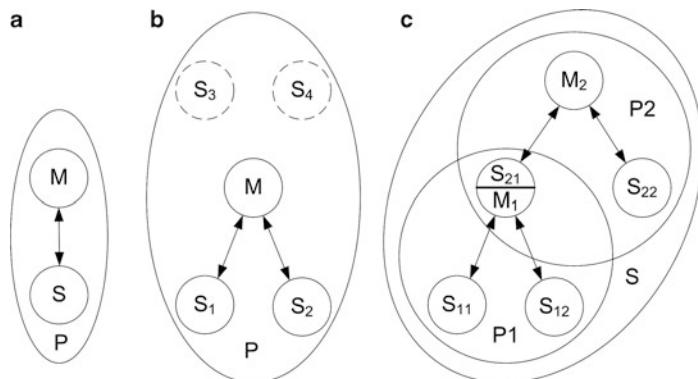


Abb. 5.38 Bluetooth Netztopologien. **a** Piconetz mit einem Slave, **b** Piconetz mit 2 aktiven (S_1 und S_2) und 2 inaktiven (S_3 und S_4) Slaves, **c** Scatternet (S) bestehend aus den Piconetzen P_1 und P_2

- in der Connecting-Phase werden Informationen zwischen Master und Slave ausgetauscht um die Verbindung unter der gewünschten Sicherheitsstufe herzustellen.

Als Pairing wird der Austausch eines Verbindungsschlüssels zwischen Geräten, die sich vorher gegenseitig authentifiziert haben, bezeichnet.

Ein Bluetooth Gerät kann vom Nutzer so eingestellt werden, dass es entweder immer, oder nur unter definierten Bedingungen für andere Geräte sichtbar ist. Und selbst wenn es sichtbar ist, kann das Gerät so eingestellt werden, dass es nur unter bestimmten Bedingungen einen Verbindungsauflaufbau akzeptiert.

Bluetooth definiert *Profile*, das ist ein Katalog von Regeln und Protokollen, damit die Geräte von verschiedenen Herstellern miteinander kommunizieren können. Die Profile sind standardisiert und müssen nicht alle auf jedem Gerät implementiert sein.

Sicherheit der Kommunikation

Bluetooth stellt für die Sicherung der Kommunikation mehrere Mechanismen zur Verfügung wie: Authentifizierung, Autorisierung und Verschlüsselung. Das Protokoll definiert folgende Sicherheitsmodi:

- Modus 1 (non secure mode). Das Gerät setzt in diesem Modus keine Sicherheitsmechanismen ein.
- Modus 2 (service Level enforced security). Die ausgewählten Sicherheitsmechanismen werden eingesetzt nachdem die Verbindungsauflaufforderung akzeptiert wurde.
- Modus 3 (link level enforced security). In diesem Modus wird die Authentifizierung während des Verbindungsauflaufbaus gefordert, eine Verschlüsselung der Kommunikation bleibt aber optional.

Die Sicherheitsmechanismen basieren auf:

- Einmaligkeit der Bluetooth Adresse des Gerätes;
- einer einstellbaren 4- bis 8-stellige PIN-Nummer;
- Zufallszahlen die zwischen Master und Slave vereinbart werden um die Kommunikation zu sichern.

Die Version 4 des Protokolls [26] erweitert den PIN auf 16 alphanumerischen Zeichen und definiert einen vierten Sicherheitsmodus, in dem die Geräte einen authentifizierten Verbindungsschlüssel verwenden um die Verbindung sicherer zu gestalten.

5.4.3.2 ZigBee

ZigBee wurde von der ZigBee Aliance als Funkprotokoll für die Vernetzung von Sensorknoten entwickelt. Das Protokoll erlaubt komplexe Netzwerkstrukturen mit bis zu 2^{16} Knoten. Niedrige Funkleistung zusammen mit kurzen Nachrichten ermöglichen die Versorgung eines Knotens mit einer Batterie über Jahre, einfache Knoten können sogar über Energy Harvesting versorgt werden.

ZigBee Geräte

Abhängig von der Komplexität der Geräte unterscheidet ZigBee zwischen FFD³³ - und RFD³⁴ -Geräten (siehe Abb. 5.39). Ein FFD-Gerät implementiert die Protokollfunktionen vollständig, kann ein Netzwerk aufbauen und verwalten und kann mit allen Arten von Geräten kommunizieren. Ein RFD-Gerät, das mit weniger Speicher ausgestattet ist, implementiert nur einen Teil der Protokollfunktionen und kann nur mit anderen RFD-Geräten kommunizieren.

Ein ZigBee-Gerät kann in einem Netzwerk folgende Rolle haben:

- Ein **Coordinator** ist ein FFD-Gerät, das ein gesamtes Netzwerk aufbaut und verwaltet.
- Ein **End-Device** ist ein RFD- oder FFD-Gerät, das nur teilweise die Anforderungen der MAC-Schicht implementiert um den Energieverbrauch zu verringern.
- Ein **Router** ist ein FFD-Gerät, das die Reichweite eines Netzwerks erhöht.
- Ein **Gateway** ist ein FFD-Gerät, das ein ZigBee- mit einem anderen Netzwerk (LAN, WLAN, usw.) verbindet.
- **Trust Center** ist ein ZigBee Coordinator, der zentral die Netzwerksicherheit organisiert.

Die Komplexität der implementierten Funktionen sowie die Rolle in einem Netzwerk bestimmen den Energieverbrauch eines Gerätes. Man unterscheidet zwischen:

- Green Power Devices: Das sind Geräte mit einem extrem niedrigen Energieverbrauch, der über Energy Harvesting oder lebenslang von einer Batterie abgedeckt werden kann.
- End-Devices befinden sich überwiegend im Sleep-Modus und werden über eine austauschbare Batterie versorgt.
- Router/Coordinator müssen die ganze Zeit aktiv sein, um die Nachrichten weiterzuleiten und werden über das Netz versorgt.

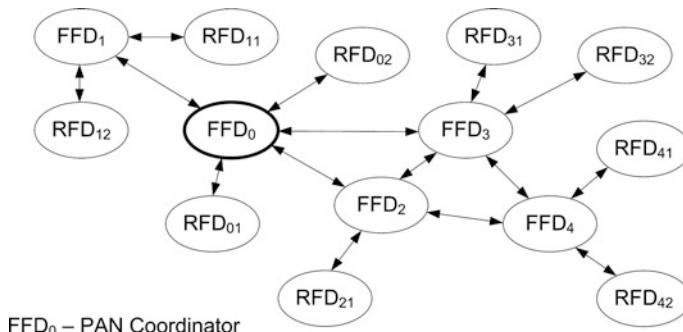
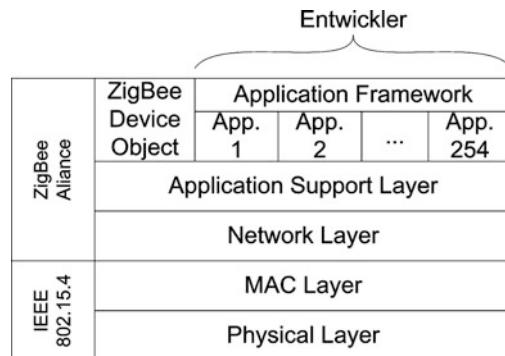


Abb. 5.39 ZigBee Netzwerk

³³ FFD – Full Function Device.

³⁴ RFD – Reduced Function Device.

Abb. 5.40 ZigBee 3 Schichten Modell



Protokoll Aufbau

Abb. 5.40 stellt das Schichten Modell der ZigBee Version 3 dar [27]. Die Bitübertragungsschicht und die Medienzugriffssteuerung des Protokolls werden vom IEEE 802.15.4 Standard spezifiziert, während die oberen Schichten von der ZigBee Aliance spezifiziert sind. Im Vordergrund stehen die Standardisierung des Protokolls auf allen Ebenen, sowie der Energieverbrauch.

Bitübertragungsschicht

Das ZigBee Protokoll definiert 27 Kommunikationskanäle (siehe Tab. 5.7). Der erste Kanal ist nur in Europa, weitere 10 sind nur in USA und die Kanäle 11 bis 26 im 2,4 GHz ISM-Band definiert. Um die Koexistenz mit anderen Funkstandards, die den gleichen Funkraum benutzen, zu gewährleisten, verwendet ZigBee das Codemultiplex-Verfahren. Dieses Verfahren erlaubt die Kommunikation innerhalb des Netzwerks bei einer Sendeleistung von 1 mW (0 dBm). Trotz der niedrigen Sendeleistung kann eine große Reichweite erzielt werden, weil das Protokoll das Weiterleiten von Botschaften vorsieht. Die Geräte besitzen eine weltweit einmalige Adresse, mit der sie sich beim Netzwerkbeitritt identifizieren müssen. Anschließend wird ihnen eine 16-Bit netzwerkinterne Adresse zugewiesen um den Energieverbrauch zu reduzieren.

Network Layer

Die Netzwerkschicht des Protokolls beinhaltet die nötigen Funktionen für den Aufbau, Verwaltung, Pflege und Sicherheit eines Netzwerks. Jedes ZigBee Netzwerk wird von einem einzigen Coordinator aufgebaut und verwaltet. Der Coordinator initialisiert das

Tab. 5.7 ZigBee – Frequenzbänder

Frequenzband	0	1	2	...	9	10	11	12	...	25	26
Frequenz [MHz]	868	906	908	...	922	924	2405	2410	...	2475	2480
Trägerabstand [MHz]	-	2					5				
Bitrate [kbit/s]	20	40					250				

Netzwerk, bestimmt den Kommunikationskanal und den Namen des Netzwerks. Über den Netzwerknamen identifizieren sich die ZigBee Netzwerke, die den gleichen Funkraum benutzen. Weil das ZigBee Protokoll das „Frequency hopping“ Verfahren nicht unterstützt, muss der Coordinator in der Initialisierungsphase des Netzwerks einen Funkkanal suchen, dessen niedrige Funkaktivität eine ungestörte Kommunikation gewährleistet. Dafür misst der Coordinator die Funkenergie eines Kanals und sucht nach eventuellen ZigBee Netzwerken, die auf diesem Kanal schon aktiv sind. Das Protokoll sieht vor, dass die Netzwerke sich selbst organisieren. Neue Geräte können sich einem Netzwerk anschließen oder es verlassen. Jeder neue Knoten bekommt eine interne, einmalige 16-Bit Netzwerkadresse zugeteilt. Der Coordinator weist sich selber die Netzwerkadresse zu (z. B. 0x0000). Eine logische 16-Bit Adresse erlaubt eine große Anzahl von Funkknoten in einem Netzwerk und führt verglichen mit der physikalischen 64-Bit Identifikationsnummer, zur Verkürzung der Botschaften und dadurch zur Minderung des Energieverbrauchs. Die Router erstellen und aktualisieren kontinuierlich eine Liste mit den Adressen der Geräte, die sich in Reichweite befinden. Die Netzwerke können sich in einer Stern-, Baum- oder Mesh-Struktur organisieren.

ZigBee Device Objekt

Das ZigBee Device Objekt ist ein wichtiger Bestandteil des Protokolls, der Verwaltungsfunktionen bezüglich des Aufbaus des Netzwerkes, der Aktualisierung der Tabelle der Knoten in Reichweite, der Suche der passenden Endgeräte (Application Endpoints) und der Verwaltung der logischen Verbindungen (bindings) beinhaltet.

Application Framework

In der Anwendungsschicht (application framework) eines ZigBee Gerätes können bis zu 254 Anwendungsobjekte (application endpoints) definiert werden. Jedes Anwendungsobjekt ist einzeln adressierbar und beschreibt den Zugang zu den einzelnen Objekten: Sensoren und/oder Aktoren. Die logische Verbindung zwischen den Anwendungsobjekten unterschiedlicher Knoten aus demselben Netzwerk wird in einer Tabelle (bindings table) festgelegt. Der Coordinator, bzw. die Router, als Geräte, die dauernd versorgt werden, speichern die Tabelle mit den Adressen aller Knoten und ihren Anwendungsobjekten, sowie die Bindungstabelle.

Um die Kommunikation zwischen Geräten von verschiedenen Herstellern zu gewährleisten, hat die ZigBee Alliance Anwendungsprofile definiert. Folgende Profile sind bereits normiert:

- Building Automation (Überwachung und Steuerung von Gewerbegebäuden);
- Home Automation (Überwachung und Steuerung im privaten Bereich);
- Health Care (Übertragung und Überwachung klinischer Parameter im medizinischen und Fitness-Bereich);
- Input Device (kabellose Anbindung von Eingabegeräten wie Maus, Tastatur, usw. an Computer);

- Light Link (Ansteuerung von komplexen Beleuchtungssystemen);
- Remote Control (Fernsteuerung);
- Retail Service (z. B. bargeldlose Bezahlung);
- Smart Energy (Übertragung von Energiemesswerten und Steuerung von energierelevanten Anlagen);
- Telecom Services (Informationsübertragung).

In der Anwendungsschicht eines ZigBee Knotens können Anwendungsobjekte aus verschiedenen Anwendungsprofilen definiert werden.

5.4.4 Funkverbindung zwischen zwei Mikrocontrollern

Der typische Anwendungsfall, der in diesem Beispiel gewählt wurde, ist die Verbindung zweier Mikrocontroller, die kabellos erfolgen soll. Bei der Wahl des Protokolls für die Funkverbindung zwischen zwei Mikrocontrollern spielen Faktoren wie Kosten, Entfernung, Datenrate, Hardware- und Softwarekomplexität, Stör- und Datensicherheit eine wichtige Rolle. Eine gute Lösung für Entfernungen unterhalb von 100 m bietet Bluetooth, das für solche Anwendungen spezifiziert wurde. Unterschiedliche Hersteller bieten preiswerte Funkmodule an, welche neben einem 2,4 GHz Transceiver, einen Hostcontroller der eine Bluetooth-Version implementiert, eine Taktquelle und eventuell eine angepasste Antenne beinhalten. Die Module stellen verschiedene Bluetooth-Profile und für die Kommunikation mit dem ansteuernden Mikrocontroller (siehe Abb. 5.41) mehrere serielle Schnittstellen wie USB, UART, SPI, PCM zur Verfügung.

Für die bilaterale Kommunikation bietet sich das SPP³⁵-Profil an, das eine serielle RS-232 Schnittstelle emuliert. Es soll dafür gesorgt werden, dass die Bluetooth-Versionen der zwei Module BTM1 und BTM2 kompatibel sind. Für dieses Beispiel wurden Module der Firma Rayson aus der Reihe BTM (z. B. BTM230) und wegen der Einfachheit die UART-Schnittstelle für die Kommunikation mit dem Mikrocontroller gewählt. Die Kommunikati-

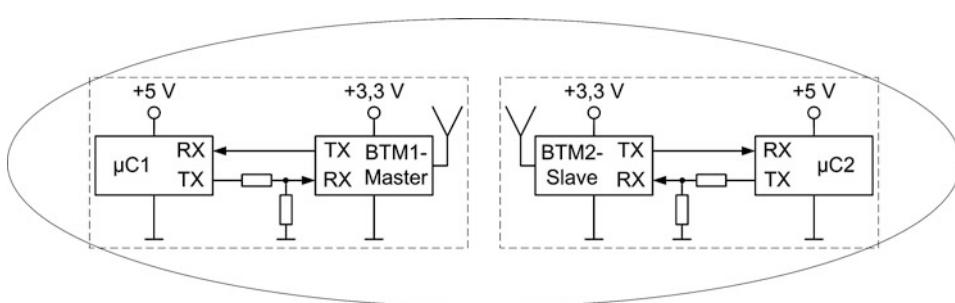


Abb. 5.41 Piconet mit zwei Mikrocontrollern

³⁵ SPP – Serial Port Profile.

onsparameter sind auf 8 Datenbit, keine Parität, 1 Stoppbit und 19.200 Bit/s voreingestellt, können aber geändert werden. Die Ansteuerung der Funkmodule und die gesamte Kommunikation zwischen den Mikrocontrollern laufen ausschließlich über UART und dafür können, die im Abschn. 5.2.1 vorgestellten Funktionen benutzt werden.

5.4.4.1 Betriebsmodi

Die BTM-Module können im Daten- oder Befehlsmodus arbeiten. Nach dem erfolgreichen Pairing-Verfahren befinden sich die Module im Datenmodus und alle über UART empfangenen Bytes werden per Funk gesendet.

Im Befehlsmodus können die Einstellungen des Moduls geändert und in einem nicht flüchtigen Speicher abgelegt werden. Wenn sich das Modul im Datenmodus befindet, kann der Befehlsmodus aktiviert werden, indem der Mikrocontroller eine Sequenz bestehend aus drei „+“-Zeichen gefolgt von „CR“ über UART sendet. Das Funkmodul sendet bei erfolgreicher Umstellung nach circa einer Sekunde die Meldung „OK“ zurück.

5.4.4.2 Befehlssatz

Über den implementierten Befehlssatz werden die Rolle des Moduls als Master oder Slave, die Einstellungen der UART-Schnittstelle, die Verbindungsart (automatisch oder manuell), die Bluetooth-Adresse des Kommunikationspartners, usw. eingestellt. Der gesamte Befehlssatz ist dem Datenblatt des benutzten Moduls zu entnehmen [28]. Es werden AT-Befehle verwendet, die zuerst für die Ansteuerung von Modems benutzt wurden. Sie bestehen aus einer Sequenz von ASCII-Zeichen, orientieren sich an die Empfehlungen der International Telecommunication Union (ITU) [29] und haben folgende Struktur:

[Präfix][Befehlskürzel]<Parameter>[Abschlusszeichen]

Die BTM-Module verwenden als Präfix die Zeichenfolge „AT“ und die Befehle werden mit einem einzigen Buchstaben codiert. Die Parameter sind befehlsspezifisch, in wenigen Ausnahmen können sie auch fehlen. Als Abschlusszeichen wird nur das Steuerzeichen „CR³⁶“ akzeptiert.

Beispiel

„AT1\r“ – ändert die UART-Übertragungsrate auf 9600 Bit/s
 „ATO\r“ – schaltet das Modul vom Befehlsmodus auf den Datenmodus um

„\r“ steht in der Programmiersprache C für „CR“ und \n für „LF“.

³⁶ Im ASCII Code mit 0x0D codiert.

Die syntaktisch korrekten Befehle werden meist in ca. 100 ms ausgeführt und mit „OK“ quittiert, ansonsten empfängt der Mikrocontroller die Meldung „ERROR“. Die zwei Meldungen werden gefolgt von „CR“ und „LF“. Die Herstellung einer Verbindung wird mit „CONNECT“, ihre Unterbrechung mit „DISCONNECT“ zurückgemeldet. Im Anschluss dieser Meldungen steht die Adresse des Kommunikationspartners.

5.4.4.3 Initialisierung des Funkmoduls

Mit der Funktion BTM_UART_Init wird die UART-Schnittstelle des Mikrocontrollers für die Kommunikation mit dem Funkmodul initialisiert, bzw. werden die Kommunikationsparameter (Baudrate, Parität und Anzahl der Stoppbits) entsprechend der neuen Einstellungen geändert.

```
void BTM_UART_Init(uint16_t uibaudrate, uint8_t ucparity,
                    uint8_t ucstopbit)
{
    /* Set baud rate */
    UBRR0 = uibaudrate;
    /* Enable receiver, transmitter and receive complete interrupt*/
    UCSR0B = (1<<RXEN0) | (1<<TXEN0) | (1<<RXCIE0);
    /* Set frame format*/
    UCSR0C = (1<<UCSZ00) | (1 << UCSZ01) | ucparity | ucstopbit;
}
```

Für die Kommunikation mit einem Modul mit Werkseinstellungen wird die Funktion folgendermaßen aufgerufen:

```
BTM_UART_Init(BAUD_19200, PAR_NO, STOP_BIT_1);
mit:
#define BAUD_19200      59 //F_CPU = 18,432 MHz
#define PAR_ODD         0x30
#define PAR_EVEN        0x20
#define PAR_NO          0x00
#define STOP_BIT_1       0x00
#define STOP_BIT_2       0x08
```

Im Folgenden wird die Initialisierung zweier BTM-Module beispielhaft erläutert, die zusammen ein Piconet bilden sollen. Die Adresse des Masters lautet „00126F250A6A“ und des Slaves „00126F250A6E“. Das Speichern der Adresse des Kommunikationspartners verhindert das Einbetten des Moduls in einem Scatternet. Die Funkverbindung soll manuell hergestellt werden. Für die Übertragung der AT-Befehle wird die in Abschn. 5.2.1.6 vorgestellte Funktion UART_WriteBuffer verwendet.

Initialisierung des Bluetooth-Masters

Der einmalige Aufruf der Funktion BTM_Master_Init initialisiert ein BTM-Modul als Master und speichert persistent die Einstellungen, vorausgesetzt das Modul befindet sich im Befehlsmodus. Die Zeichenfolge „+++“ aktiviert den Befehlsmodus, wenn dieser bereits aktiv ist, wird sie mit der Meldung „ERROR“ quittiert und die nachfolgenden Befehle werden ausgeführt.

```
void BTM_Master_Init(void)
{
    delay(2000); //2 s Warten vor dem ersten Befehl
    //das Funkmodul wird in den Befehlsmodus versetzt
    UART_WriteBuffer((uint8_t*) "+++\r\0",5);
    delay(1500); //1,5 s Warten um den vorigen Befehl auszuführen
    //das Echo seitens des Funkmoduls wird ausgeschaltet
    UART_WriteBuffer((uint8_t*) "ATE0\r\0",6);
    delay(100); //100 ms Warten um den vorigen Befehl auszuführen
    //die automatische Herstellung der Funkverbindung wird deaktiviert
    UART_WriteBuffer((uint8_t*) "ATO1\r\0",6);
    delay(100); //100 ms Warten um den vorigen Befehl auszuführen
    //das Modul wird als Master eingestellt
    UART_WriteBuffer((uint8_t*) "ATR0\r\0",6);
    delay(3000); //3 s Warten um den vorigen Befehl auszuführen
    /*die Adresse des Slaves mit dem die Verbindung hergestellt
     *werden soll, wird gespeichert*/
    UART_WriteBuffer((uint8_t*) "ATD=00126F250A6E\r\0",18);
    delay(100); //100 ms Warten um den vorigen Befehl auszuführen
}
```

Das Pairing-Verfahren kann jetzt mit dem Aufruf der Funktion BTM_Master_Connect initiiert werden. Der Master versucht, innerhalb von 60 s die Verbindung mit dem Slave, dessen Adresse er gespeichert hat, herzustellen. Wenn der Slave ausgeschaltet oder nicht in Reichweite ist, wird dem Mikrocontroller „Time out, Fail to connect“ gemeldet und das Verfahren muss neu gestartet werden. Die vorgeschlagenen Wartezeiten sind Erfahrungswerte und können bei anderen Modulen abweichen.

```
void BTM_Master_Connect(void)
{
    delay(100);
    //das Pairing-Verfahren wird initiiert
    UART_WriteBuffer((uint8_t*) "ATA\r\0",5);
}
```

Wenn der Befehl „ATO1“ in der Initialisierungsroutine mit dem „ATO0“ ersetzt wird, so wird die Funkverbindung automatisch aufgebaut, sobald der Master und der Slave

eingeschaltet und in Reichweite sind. Die Funktion `BTM_Master_Connect()` muss nicht mehr aufgerufen werden.

Initialisierung des Bluetooth-Slaves

Im vorgestellten Piconet kann der Slave wie folgt initialisiert werden. Mit diesen Einstellungen wird das lokale Echo abgeschaltet und der Slave erlaubt eine Verbindung nur mit dem Master, dessen Adresse er speichert.

```
void BTM_Slave_Init(void)
{
    delay(2000); //2 s Warten vor dem ersten Befehl
    //das Funkmodul wird in den Befehlsmodus versetzt
    UART_WriteBuffer((uint8_t*) "+++\\r\\0",5);
    delay(1500); //1,5 s Warten um den vorigen Befehl auszuführen
    //das Echo seitens des Funkmoduls wird ausgeschaltet
    UART_WriteBuffer((uint8_t*) "ATE0\\r\\0",6);
    delay(100); //100 ms Warten um den vorigen Befehl auszuführen
    //das Modul wird als Slave eingestellt
    UART_WriteBuffer((uint8_t*) "ATR1\\r\\0",6);
    delay(3000); //3 s Warten um den vorigen Befehl auszuführen
    //die Adresse des Masters der die Verbindung hergestellt hat,
    //wird gespeichert
    UART_WriteBuffer((uint8_t*) "ATD=00126F250A6A\\r\\0",18);
    delay(100); //100 ms Warten um den vorigen Befehl auszuführen
}
```

Literatur

1. Meroth, A., Tolg, B.: Infotainmentsysteme im Kraftfahrzeug. Grundlagen, Komponenten, Systeme und Anwendungen. Vieweg, Wiesbaden (2008)
2. Schwab, A.J., Kürner, W.: Elektromagnetische Verträglichkeit. Springer, Berlin, Heidelberg (2007)
3. Roppel, C.: Grundlagen der digitalen Kommunikationstechnik. Hanser, München (2006)
4. NXP: CAN Bosch Controller Area Network (CAN) Version 2.0 PROTOCOL STANDARD (2016). <http://www.nxp.com/assets/documents/data/en/reference-manuals/BCANPSV2.pdf>, Zugriffen: 1. Dezember 2016
5. ISO 11898: Road vehicles – Controller area network (CAN) (6 Teile)
6. Riggert, W., Märtin, C., Lutz, M.: Rechnernetze – Grundlagen, Ethernet, Internet, 5. Aufl. Hanser, München (2015)
7. Semiconductors, N.X.P.: UM10204 – I2C-bus specification and user manual – Rev.6.4 (2014). www.nxp.com, Zugriffen: 28. Dezember 2014
8. NXP Semiconductors: Application Note AN255-02 – I2C/SMBus Repeaters, Hubs and Expanders (2015). www.nxp.com, Zugriffen: 14. April 2018
9. NXP Semiconductors: Application Note AN262_2 – PCA954x Family of I²C/SMBus Multiple-
xers and Switches (2015). www.nxp.com, Zugriffen: 14. April 2018

10. Greif, F.: Roboterclub Aachen: Universelle CAN Bibliothek (2008). <http://www.kreatives-chaos.com/artikel/universelle-can-bibliothek>, Zugegriffen: 14. April 2018
11. Etschberger, K. (Hrsg.): CAN Controller Area Network – Grundlagen, Protokolle, Bausteine, Anwendungen. Hanser, München (1994)
12. Zimmermann, W., Schmidgall, R.: Bussysteme in der Fahrzeugtechnik – Protokolle, Standards und Softwarearchitektur, 5. Aufl. Springer Vieweg, Wiesbaden (2014)
13. Microchip Technology Inc.: MCP2515 Datasheet. <http://ww1.microchip.com/downloads/en/DeviceDoc/21801d.pdf>, Zugegriffen: 12. Januar 2017
14. Modbus.org: MODBUS application protocol specification V1.1b3 (2016). www.modbus.org
15. Modbus.org: MODBUS over serial line. Specification and implementation guide V1.02 (2016). www.modbus.org, Zugegriffen: 26. April 2016
16. Corrigan, S.: Interface Circuits for TIA/EIA-485 (RS-485). Application Report SLLA036D Revised August 2008 (2008). www.ti.com, Zugegriffen: 1. Mai 2016
17. Bundesnetzagentur: VFG 30/2014, geändert mit Vfg 36/2014, geändert mit Vfg 69/2014 (2016). http://www.bundesnetzagentur.de/SharedDocs/Downloads/DE/Sachgebiete/Telekommunikation/Unternehmen_Institutionen/Frequenzen/Allgemeinzuteilungen/2014_69_SRD_pdf.pdf?__blob=publicationFile&v=1, Zugegriffen: 26. August 2016
18. Bundesnetzagentur: VFG 76 / 2003. Allgemeinzuteilung von Frequenzen in den Frequenzteilbereichen gemäß Frequenzbereichszuweisungsplanverordnung (FreqBZPV), Teil B: Nutzungsbestimmungen (NB) D138 und D150 für die Nutzung durch die Allgemeinheit für ISM-Anwendungen (2003). www.bundesnetzagentur.de, Zugegriffen: 1. September 2016
19. Beuth, Breide, Lüders, Kurz, Hanebuth: Nachrichtentechnik. Vogel Industrie Medien, Würzburg (2009)
20. Shuang-Hua Yang: Wireless sensor networks. Principles, design and applications. Springer, London (2014)
21. Microchip: AT86RF231 Low Power 2.4 GHz Transceiver for ZigBee, IEEE 802.15.4, 6LoWPAN, RF4CE, SP100, Wireless HART and ISM Applications (2016). www.microchip.com, Zugegriffen: 14. April 2018
22. Nordic Semiconductor: nRF24L01+ single chip 2.4 GHz transceiver. Product specification v.1.0 (2016). www.nordicsemi.com, Zugegriffen: 5. Juli 2016
23. Microchip Technology: MRF24J40 Data Sheet. IEEE 802.15.4TM 2.4 GHz RF Transceiver (2016). www.microchip.com, Zugegriffen: 5. Juli 2016
24. IEEE Std 802.15.1TM-2002. Wireless Medium Access Control (MAC) and Physical Layer (PHY) specifications for Wireless personal Area Networks (WPANs). abgerufen am 30. August 2016 von <http://ieeexplore.ieee.org>
25. Heydon, R.: Bluetooth low energy – the developer's handbook. Prentice Hall, Upper Saddle River, NJ (2013)
26. Bluetooth: Specification of the Bluetooth system core 4.2 (2014). www.bluetooth.com, Zugegriffen: 9. September 2016
27. ZigBee 3: Präsentation (2014). www.zigbee.org/zigbee-for-developers/zigbee3-0/, Zugegriffen: 8. November 2016
28. Rayson Technology: BTM-230 Data sheet. BC04-EXT Class1 Module BTM-230 (2016). <http://www.tme.eu/de/Document/dd4f4d23feb6055a8b3eec77ed90af65/BTM-230.pdf>, Zugegriffen: 22. August 2016
29. International Telecommunication Union: ITU-T V.250. Serial asynchronous automatic dialing and control (2016). <http://www.itu.int/rec/T-REC-V.250-200307-I/en>, Zugegriffen: 23. August 2016

Weiterführende Literatur

30. Tanenbaum, A.S., Wetherall, D.J.: Computernetzwerke, 5. Aufl. Pearson, München (2012)
31. Microchip Technology Inc.: 8-bit microcontroller with 32K/64K/128K bytes of ISP flash and CAN controller (2015). www.microchip.com, Zugegriffen: 14. April 2018
32. Microchip Technology Inc.: 8-bit atmel microcontroller with 4/8/16k bytes in-system programmable flash (2015). www.microchip.com, Zugegriffen: 14. April 2018
33. Modbus-IDA.org: MODBUS Messaging on TCP/IP Implementation Guide V1.0b (2016). www.modbus-ida.org, Zugegriffen: 26. April 2016
34. Soltoro, M. et al.: RS-422 and RS-485 standards overview and system configurations application report. SLLA070D revised May 2010 (2010). www.ti.com, Zugegriffen: 5. Mai 2016
35. Bundesnetzagentur: (2003). http://www.bundesnetzagentur.de/SharedDocs/Downloads/DE/Sachgebiete/Telekommunikation/Unternehmen_Institutionen/Frequenzen/Allgemeinzuteilungen/2003_76_ISM_pdf.pdf?__blob=publicationFile&v=5, Zugegriffen: 26. Juni 2016
36. Gessler, R., Krause, T.: Wireless-Netzwerke für den Nahbereich. Vieweg + Teubner | GWV Fachverlage, Wiesbaden (2009)
37. Labiod, H., Afifi, H., De Santis, C.: WiFi™ Bluetooth™ ZigBee™ and WiMax™. Springer, Berlin, Heidelberg (2007)
38. Lüders, C.: Lokale Funknetze. Wireless LANs (IEEE 802.11), Bluetooth, DECT. Vogel, Würzburg (2007)



Zusammenfassung

In den vorangegangenen Kapiteln konnten Sie sich mit der Verwendung der AVR-Controller vertraut machen. Das vorliegende Kapitel ist nun das „Herzstück“ des Buches: Hier werden verschiedene Sensoren und ihre Verwendung vorgestellt, darunter ein Gyroskop, ein digitaler Luftdruck/Höhensensor, ein Luftfeuchtesensor, der auch die Temperatur misst, ein Magnetfeldsensor, Beschleunigungssensoren, Näherungssensoren, Stromsensoren und ein Thermometer. Programmbeispiele und Beschaltungshinweise sollen Ihnen helfen, Ihre eigenen Projekte zu realisieren, aber mehr noch, auch andere Sensoren, die nicht in diesem Buch beschrieben sind, mit den vorgestellten Tipps und Tricks auf ähnliche Weise einzubinden.

Ein analoger Sensor besteht in der Regel aus einem oder mehreren Messfühlern, einem rauscharmen Verstärker und eventuell einem Spannungsregler. Ein Messfühler wandelt eine physikalische Größe in eine elektrische um. Dieses elektrische, analoge Signal wird verstärkt und für Messung, weitere Verarbeitung oder Übertragung zur Verfügung gestellt. Zusätzliche Verarbeitung wie Linearisierung der Kennlinie, Kompensierung des DC-Offsets oder des Einflusses anderen Größen wie Temperatur, Luftdruck, oder Luftfeuchtigkeit sind einfacher zu realisieren, wenn das analoge Signal zuvor in ein digitales umgewandelt wird. Die Digitalisierung soll so nah wie möglich am Sensor stattfinden um die Störeinflüsse zu minimieren.

6.1 Systemtechnische Überlegungen

Moderne Technologien, insbesondere die Mikromechanik (MEMS) und die Hochintegration unterschiedlicher Halbleiterstrukturen, ermöglichen, eine immer größere Anzahl von Sensoren in immer mehr Geräten einzubauen beziehungsweise in ein einziges Gehäuse

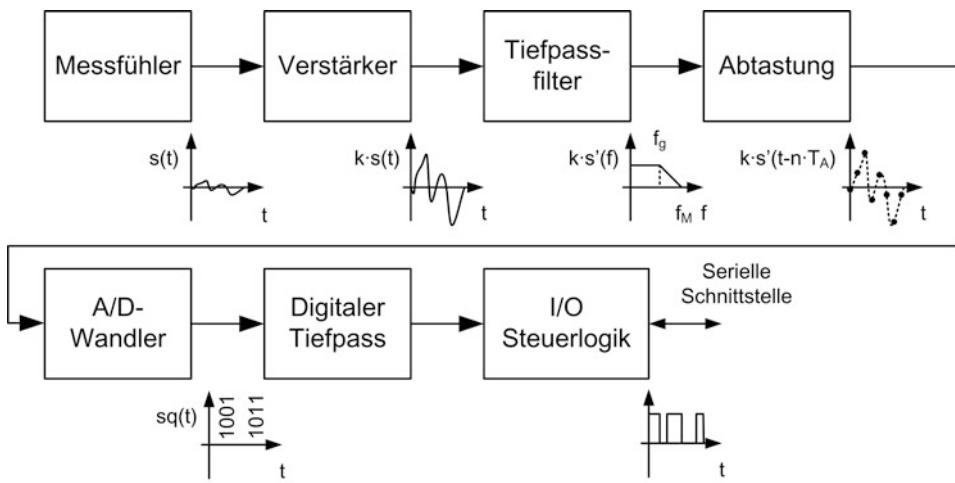


Abb. 6.1 Digitaler Sensor – Blockschaltbild

zu integrieren. Um die Vorteile der digitalen Verarbeitung und Übertragung zu nutzen, um den Energieverbrauch und die Schaltungskomplexität zu reduzieren und um den steuernden Mikrocontroller zu entlasten sind digitale Sensoren entstanden. Ein beispielhaftes Blockschaltbild eines solchen Sensors ist in Abb. 6.1 zu sehen. Eventuelle Störungen auf der analogen Seite des Sensors werden wegen der extrem kurzen Leitungen auf ein Minimum reduziert. Die Messwerte werden in digitaler Form mit auswählbarer Auflösung auf einem Datenbus, der der Vernetzung mehrerer Schaltungskomponenten dient, zur Verfügung gestellt.

6.1.1 Abtastung

Ein analoges Signal, dessen Übertragung störanfällig und dessen Verarbeitung schwierig ist, enthält auch irrelevante und redundante Anteile. Ein solches zeit- und wertkontinuierliches Signal $s(t)$ wird durch eine zeitdiskrete und wertkontinuierliche Folge $s(t - n \cdot T_A)$ vollständig beschrieben, wenn das Spektrum des Signals $s(f)$ auf eine Frequenz f_M begrenzt ist, so dass:

$$2 \cdot f_M \leq f_a = \frac{1}{T_A}, \quad (6.1)$$

wobei n eine natürliche Zahl und f_a die Abtastfrequenz ist. Die mit der Gl. 6.1 beschriebene Forderung ist als Abtasttheorem oder Nyquist-Kriterium bekannt und die Frequenz $f_a / 2$ wird in der Literatur als Nyquist-Frequenz bezeichnet.

Die Begrenzung des Spektrums wird mit einem Tiefpassfilter realisiert wodurch auch die irrelevanten Anteile des analogen Signals entfernt werden. Dieser Vorgang ist irreversibel, die entfernten Anteile können später nicht mehr zurückgewonnen werden. Die

Bandbegrenzung wird meist mit aktiven Filtern realisiert. Mit steigender Komplexität (Ordnung) des Filters steigt die Steilheit, mit der der Amplitudengang des Filters ab der Grenzfrequenz f_g abfällt. Ein Filter höherer Ordnung ermöglicht die Wahl der Grenzfrequenz nah an der Nyquist-Frequenz wie es in Abb. 6.2 verdeutlicht ist. Das Bild stellt den Betrag der Übertragungsfunktion $|H(j\omega)|$ / dB für Butterworth-Filter 1., 2. und 3. Ordnung mit unitärem Verstärkungsfaktor in Abhängigkeit der normierten Kreisfrequenz $\Omega = \omega / \omega_0$ mit $\omega = 2\pi f$ dar. Diese Filter weisen bei der Grenzfrequenz einen Amplitudenabfall um $1/\sqrt{2}$ und ab der Grenzfrequenz eine Dämpfung von $n \cdot 20$ dB/Dekade auf, wobei n die Ordnung des Filters ist. Durch das Zurückrechnen auf lineare Werte erhält man einen Dämpfungsfaktor von 10^n /Dekade.

Durch die Abtastung eines analogen Signals entsteht ein PAM¹-Signal, aus dem das ursprüngliche Basisbandsignal fehlerfrei rekonstruiert werden kann, wenn die Forderung des Abtasttheorems eingehalten wird. Die Rekonstruktion wird mit einem Tiefpassfilter realisiert. Abb. 6.3a zeigt die Abtastung eines harmonischen Signals mit unterschiedlichen Verfahren. Während in der Theorie die Abtastwerte durch die Multiplikation des ursprünglichen Signals mit einer Diracstoßreihe berechnet werden, wird in der Praxis die Abtastung mit Hilfe von FET²-Transistoren als elektronische Schalter realisiert wie in Abb. 6.3 II prinzipiell dargestellt ist. Die gleitende und die Momentanwertabtastung Abb. 6.3 II, III werden benutzt um die Abtastwerte mehrerer Kanäle auf eine Datenleitung im Zeitmultiplexverfahren zu übertragen. Das Sample and Hold³ Verfahren Abb. 6.3 IV wird in allen Analog/Digital-Wandlern dazu benutzt, eine konstante Spannung während der Umwandlung zu gewährleisten. Auch bei der Wahl der Umwandlungsrate des Analog/Digital-Wandlers eines Mikrocontrollers muss das Abtasttheorem berücksichtigt werden.

Das Spektrum $s(f)$ eines analogen bandbegrenzten Signals $s(t)$ ist in Abb. 6.4 dargestellt. Der mathematische Ausdruck des Fourier-Spektrums $s_a(f)$ des abgetasteten Signals

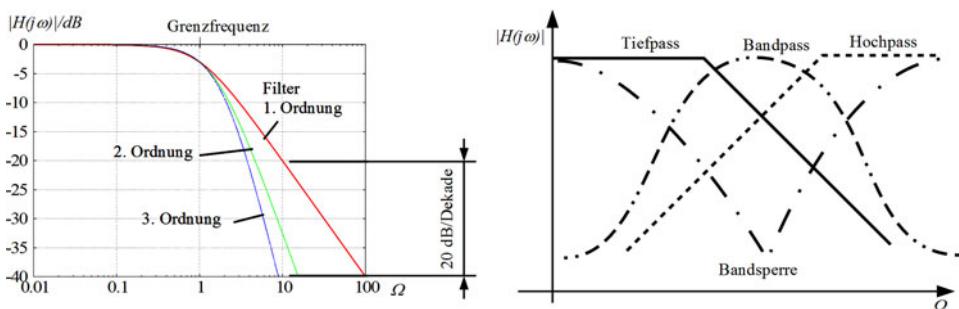


Abb. 6.2 Butterworth Filter – Bodediagramme. (Aus [1])

¹ PAM -Pulsamplitudenmodulation.

² FET – Field Effect Transistor (Feldeffekttransistor).

³ Sample and Hold – Abtasthalteglied.

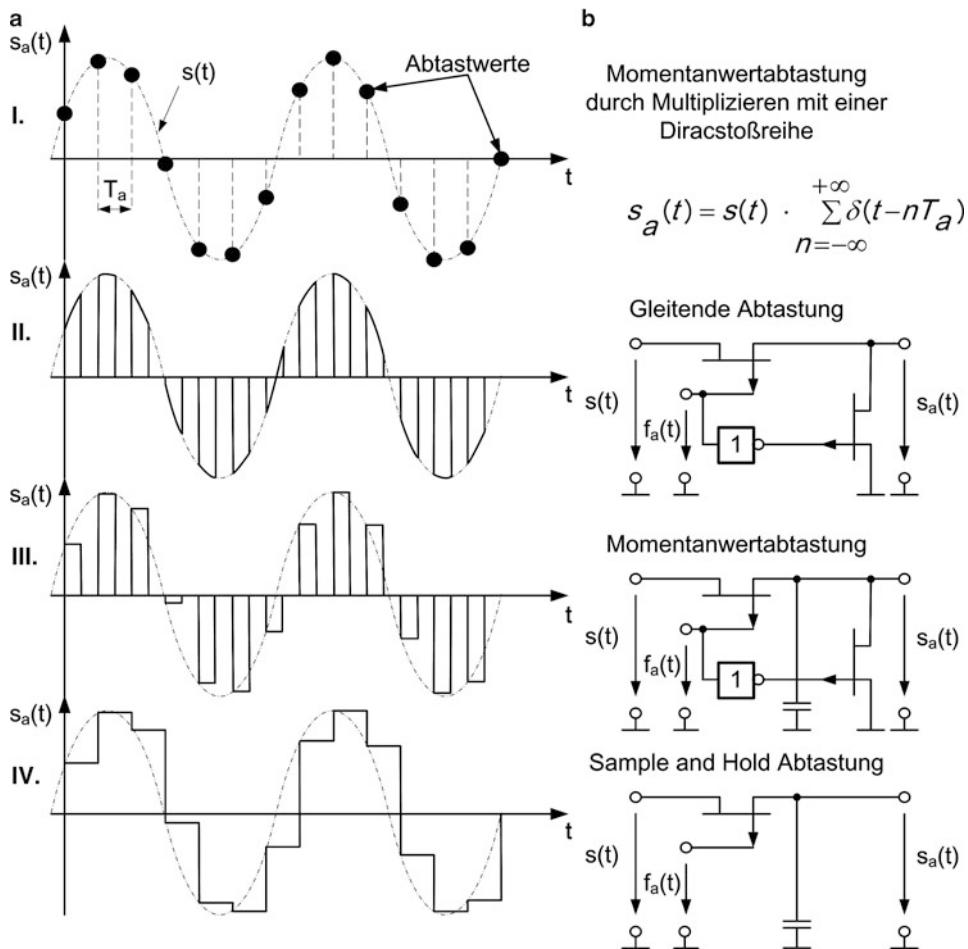


Abb. 6.3 Abtastverfahren

lautet [2]:

$$s_a(f) = \frac{1}{T_a} \cdot \sum_{n=-\infty}^{+\infty} s(f - nf_a) \quad (6.2)$$

Gl. 6.2 zeigt, dass:

- das Spektrum des Basisbandsignals sich um die Abtastfrequenz und ihre Vielfachen wiederholt;
- die Abtastfrequenz und ihre Vielfachen im Spektrum des abgetasteten Signals nicht vorhanden sind.

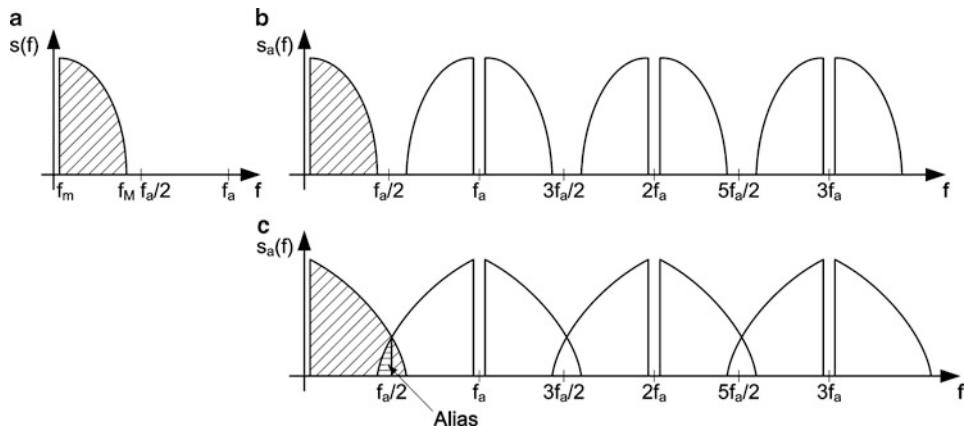


Abb. 6.4 Spektrum des bandbegrenzten Signals $s(f)$ und des abgetasteten Signals $s_a(f)$ für $2f_M < f_a$ und $2f_M > f_a$

Abb. 6.4b zeigt eine Abbildung des realen Spektrums wenn die Forderung des Abtasttheorems erfüllt ist. Es ist leicht zu erkennen, dass mit einem steilen Tiefpassfilter (Rekonstruktionstiefpassfilter) das ursprüngliche Spektrum rekonstruierbar ist.

Wenn das Eingangsfilter fehlt oder eine zu hohe Grenzfrequenz hat oder nicht steil genug ist, entsteht eine Überlappung der Teilspektren wie in Abb. 6.4c, was als Aliasing bezeichnet wird. Die Spektralkomponenten mit der Frequenz f_x mit $f_M > f_x > f_a/2$ die, die Bedingung $|n \cdot f_a - f_x| < f_a/2$ erfüllen, werden in das rekonstruierte Signal hineingespiegelt und können nicht mehr entfernt werden. Diese gespiegelten Frequenzen heißen Alias-Komponenten. Wenn beispielsweise das Eingangsfilter eines Übertragungssystems, das mit einer Abtastfrequenz von 8 kHz arbeitet, das Signalspektrum auf 5 kHz begrenzt, dann wird eine 4,5 kHz Spektralkomponente mit einer Frequenz von 3,5 kHz in das rekonstruierte Signal auftauchen. Das Eingangsfilter wird als Antialiasing-Filter bezeichnet.

6.1.2 Quantisierung

Durch die Quantisierung als Folgeschritt der Abtastung wird das zeitdiskrete und wertkontinuierliche PAM-Signal in ein zeit- und wertdiskretes Signal umgewandelt. Die quantisierten Werte, die dadurch entstehen, sind über eine Anzahl von n Bits codiert. Man unterscheidet zwischen linearen und nichtlinearen Quantisierung. Bei der linearen Quantisierung wird der Aussteuerbereich in 2^n gleich große Intervalle unterteilt, wie in Abb. 6.5 beispielhaft für $n=4$ dargestellt ist. Jedem Quantisierungsintervall wird ein binärer Code als Ganzzahl aus dem Bereich $[0; 2^n - 1]$ zugewiesen. Bei Sensoren, die eine Wechselspannung umwandeln müssen, wie unter anderem Beschleunigungs-, Winkelgeschwindigkeits- oder Magnetfeld-Sensoren, werden die Quantisierungsintervall-

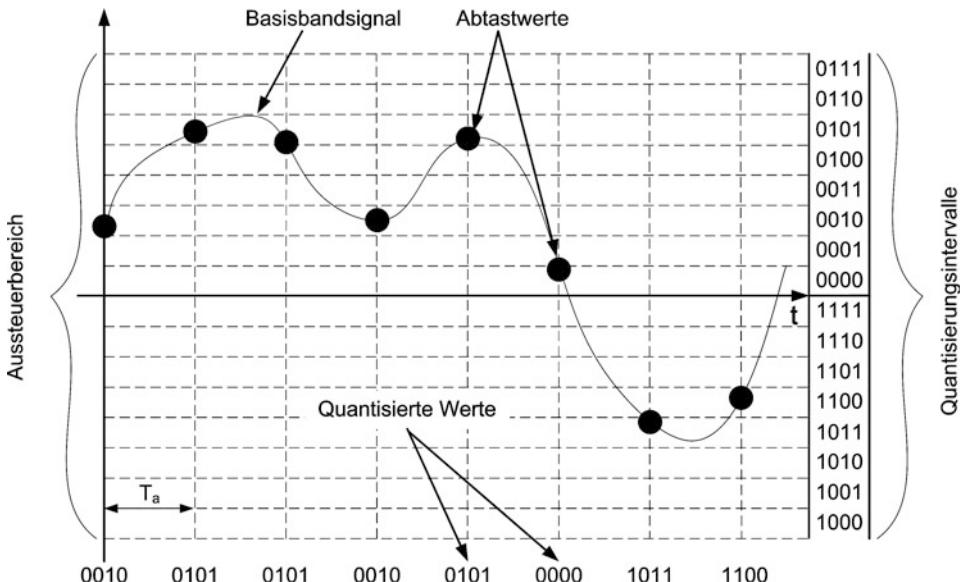


Abb. 6.5 Lineare Quantisierung

le der A/D-Wandler im Zweierkomplement codiert um eine spätere Datenverarbeitung zu erleichtern. Bei Sensoren, die den Luftdruck, die Temperatur oder die Luftfeuchtigkeit messen, kann die Codierung vorzeichenlos sein. Die Quantisierung ist ein Vergleichsverfahren bei dem jedem analogen, abgetasteten Wert der entsprechende Code des Intervalls zugewiesen wird, in das der abgetastete Wert fällt. Durch diese Zuordnung werden auch die redundanten Anteile aus dem Signal eliminiert. Der Vorgang ist reversibel und das abgetastete Signal kann bis auf einen Fehler zurückgewonnen werden. Das Fehlersignal wird als Quantisierungsrauschen bezeichnet und entsteht durch die Rundung, die bei der Quantisierung gemacht wird. Wenn u_q die Höhe eines Quantisierungsintervalls ist und man sich auf die Mitte des Intervalls bezieht [3], dann nimmt das Quantisierungsrauschen Werte im Bereich $[-u_q/2; +u_q/2]$ an. Ein Maß für dieses Rauschen bildet der Störabstand ab [4]:

$$\left| \frac{S}{N_q} \right|_{\text{dB}} = 10 \log \frac{S}{N_q} \approx n \cdot 6 \text{ [dB]} \quad (6.3)$$

wobei S die Signalleistung, N_q die Leistung des Quantisierungsrauschens und n die Zahl der Bits ist. Beim Überschreiten des Aussteuerbereiches redet man von Übersteuerung, ein Effekt, der den Störabstand stark beeinträchtigt. Eine Untersteuerung von 50 % vermindert den Signal-Rausch-Abstand um ca. 6 dB, was aus dieser Sicht einer Quantisierung mit $(n - 1)$ Bit entspricht.

Die nichtlineare Quantisierung wird in der Audiotechnik eingesetzt, um einen konstanten Störabstand über große Aussteuerbereiche zu sichern.

6.1.3 Digitale Filterung

Ein digitales Filter wird so eingesetzt, wie das Blockschaltbild eines digitalen Sensors Abb. 6.1 zeigt. Vor der Abtastung wird das Spektrum des zu messenden Signals mit einem Antialiasing-Filter niedriger Ordnung auf eine Frequenz f_M begrenzt, die viel größer als die Messfrequenz ist. Um das Nyquist-Kriterium unter diesen Bedingungen zu erfüllen, muss das Signal überabgetastet werden. Nach der Quantisierung wird das Signal digital gefiltert. Das digitale Filter ersetzt ein aufwendiges, analoges Filter, das schwer integrierbar, parametrierbar und schlecht reproduzierbar ist. Die neue Grenzfrequenz des digitalen Filters richtet sich nach der gewünschten Messrate. Die erreichte Flankensteilheit des Filters ist sehr hoch. Die Filtereigenschaften können im laufenden Betrieb geändert werden und werden von Temperatur- und Betriebsspannungs-Schwankungen nicht beeinflusst. Das Verhalten eines digitalen Filters kann schon in der Entwurfsphase genau simuliert werden.

6.1.3.1 Finite Impulse Response Filter (FIR)

FIR-Filter sind Filter mit endlicher Impulsantwort mit der Übertragungsfunktion [2]:

$$H(z) = \sum_{k=0}^N h[k] \cdot z^{-k} \quad (6.4)$$

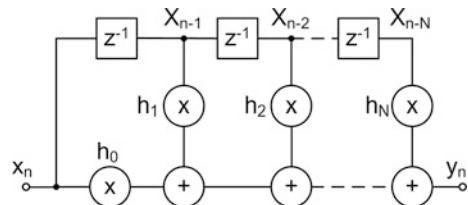
N ist die Filterordnung, $h[k]$ sind die Filterkoeffizienten, die durch die Abtastung der Impulsantwort des Filters berechnet werden können [2] und z^{-1} bedeutet eine Zeitverzögerung.

$$h[k] = \frac{1}{2} \cdot \text{si}\left(\frac{1}{2} \cdot \left(k - \frac{N}{2}\right)\right) \quad \text{mit } i = 0 \dots N. \quad (6.5)$$

Die si-Funktion, oder Spaltfunktion wird definiert als $\text{si}(x) = \sin(x) / x$. Die Struktur eines FIR-Filters ist in Abb. 6.6 dargestellt. Für die Berechnung des gefilterten Wertes y_n , werden außer dem aktuellen Abtastwert nur noch die N vorangegangenen Werte betrachtet:

$$y_n = \sum_{k=0}^N x[n-k] \cdot h[k] \quad (6.6)$$

Abb. 6.6 FIR Filter



Mit diesen Überlegungen können digitale Tiefpassfilter entworfen werden, die eine steigende Steilheit der Filterflanke mit der Erhöhung der Filterordnung aufweisen. Die minimale Dämpfung dieser Filter im Sperrbereich erreicht 20 dB und ist von der Filterordnung unabhängig. Die Welligkeit des Filters wird reduziert und die Dämpfung im Sperrbereich wird durch die Gewichtung der Filterkoeffizienten mit einer Fensterfunktion $h'[k] = h[k] \cdot w[k]$ erhöht. Folgende Gleichung:

$$w[k] = a - b \cdot \cos\left(\frac{2\pi \cdot k}{N}\right) \quad (6.7)$$

beschreibt die Funktionen *Hanning* (mit $a=b=0,5$) und *Hamming* (mit $a=0,54$ und $b=0,46$), zwei der bekanntesten Fensterfunktionen. Mit der Fenstermethode können die Koeffizienten eines FIR-Filters mit vorgegebener Grenzfrequenz, Steilheit und Dämpfung im Sperrbereich berechnet werden. In der Literatur [2, 5] wird als weitere Entwurfsmethode die Optimalmethode beschrieben, die zu einer niedrigeren Filterordnung führt.

Ein digitales Filter, das in Echtzeit arbeitet, kann hardwaremäßig – wie in Abb. 6.7 dargestellt – realisiert werden. Die hohe Taktfrequenz und die parallelen Abläufe erlauben eine komplexere Filterstruktur. Eine anwendungsspezifische integrierte Schaltung (ASIC⁴) kann ein digitales Filter zusammen mit dem Antialiasing-Tiefpassfilter und dem A/D-Wandler integrieren. Ein FPGA⁵ braucht zusätzlich ein externes Filter und die weniger komplexen CPLD⁶'s können die digitalen Werte von einem externen A/D-Wandler verarbeiten. Die digitale Steuerung kann die Auflösung des A/D-Wandlers, bzw. die Grenzfrequenz des Filters ändern.

Softwaremäßig kann die Gleichung Gl. 6.6 auch mit einem Digital Signal Processor (DSP) oder einem Mikrocontroller gelöst werden. Dies ist in Abb. 6.8 prinzipiell darge-

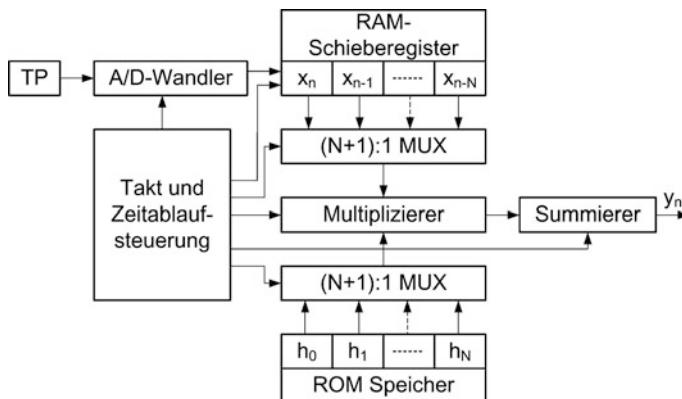


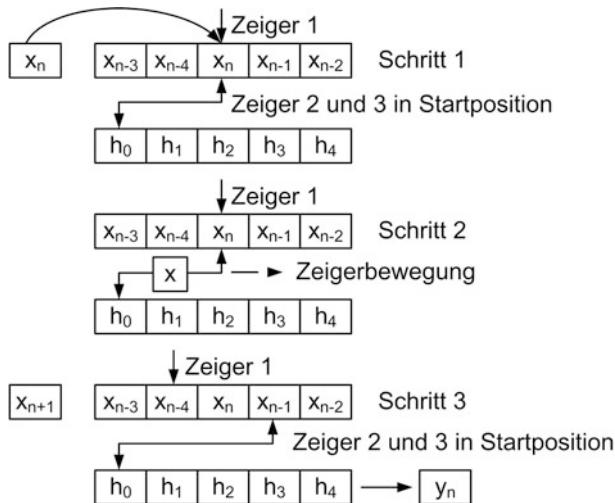
Abb. 6.7 Digitales Filter – prinzipieller Aufbau

⁴ ASIC – application specific integrated circuit.

⁵ FPGA – field programmable gate array.

⁶ CPLD – complex programmable logic device.

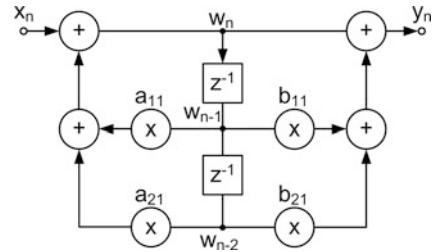
Abb. 6.8 FIR Filter Software-lösung



stellt. Der Zeiger 1 zeigt auf die Vektorstelle, an der der nächste digitale Wert gespeichert wird. Nach dem Speichern finden im zweiten Schritt die Multiplikationen gemäß Gl. 6.6 zwischen den Filterkoeffizienten und den digitalen Werten statt. Die Zeiger 2 und 3 werden synchron inkrementiert und zeigen auf die Werte, die multipliziert werden sollen. Gleichzeitig werden die Ergebnisse der Multiplikationen aufsummiert. Im letzten Schritt wird der gefilterte Wert y_n ausgegeben und die Zeiger neu gesetzt. Die Dauer des vorgestellten Ablaufs muss kleiner als die Abtastperiode sein.

Anders als die DSPs, die mit Fließkommazahlen rechnen können, rechnen die meisten 8 Bit Mikrocontroller bevorzugt mit Festkomma- oder mit Ganzzahlen. Aus diesem Grund werden die Filterkoeffizienten quantisiert, was zu einer Abweichung von der idealen Übertragungsfunktion führt. Unpassend ausgewählte Variablentypen für die Berechnung des digitalen Filters können auch zu einem nichtidealen Effekt führen. Das maximale Zwischenergebnis das bei den Multiplikationen entsteht, kann gut geschätzt werden, weil sowohl die Bitauflösung des A/D-Wandlers als auch die Größe der Koeffizienten bekannt sind. Bei FIR-Filtern kann ein Überlauf meist bei der Aufsummierung der Zwischenergebnisse stattfinden. Variablentypen, die viel mehr Bits umfassen als die Busbreite des Mikrocontrollers, senken das Überlaufrisiko, verlangsamen aber die Rechengeschwindigkeit. Um die Rechengeschwindigkeit bei gleichzeitig niedrigem Überlaufrisiko zu erhöhen, kann die Skalierungs-Methode angewendet werden. Die Filterkoeffizienten werden alle durch den gleichen Faktor geteilt. Wenn sich ein zufälliger Überlauf trotzdem nicht vermeiden lässt, können die Ergebnisse begrenzt werden (Sättigungs-Methode). FIR-Filter sind leicht zu implementieren, sind tolerant gegenüber der Quantisierung der Filterkoeffizienten und besitzen nur Nullstellen, deshalb sind sie stets stabil. Wegen der hohen Ordnung dieser Filter, werden viele mathematische Operationen benötigt.

Abb. 6.9 IIR Filter 2. Ordnung



6.1.3.2 Infinite Impulse Response Filter (IIR)

Die Übertragungsfunktion eines IIR-Filters bildet die eines analogen Filters nach:

$$H(z) = \frac{1 + b_1 \cdot z^{-1} + \dots + b_N \cdot z^{-N}}{1 - a_1 \cdot z^{-1} - \dots - a_M \cdot z^{-M}} \quad (6.8)$$

Diese Filter besitzen sowohl Pol- als auch Nullstellen und sind rekursiv, was zu einer unendlich langen Impulsantwort führt. Durch unpassende Quantisierung der Filterkoeffizienten und durch Überlauf können sie leicht instabil werden. Der Überlauf muss durch die Herunterskalierung der Koeffizienten und die Begrenzung (Sättigung) der Zwischenergebnisse verhindert werden. Diese Filter bieten aber eine hohe Filtersteilheit bei einer relativ niedrigen Ordnung des Filters, was zu einer Reduktion der mathematischen Operationen pro gefilterten Wert führt. Die Übertragungsfunktion eines Filters $(2 \cdot n + 1)$ -ten Ordnung kann folgendermaßen zerlegt werden:

$$H(z) = H_1 + \prod_{k=1}^n H_{2k}(z) \quad (6.9)$$

mit $H_1(z)$ die Übertragungsfunktion eines Filters 1. Ordnung und $H_{2k}(z)$ die Übertragungsfunktion eines Filters 2. Ordnung. Durch diese Zerlegung wird die Abweichung von der idealen Übertragungsfunktion wegen der Rundungen verringert. Die Struktur einer digitalen IIR-Filterstufe 2. Ordnung ist in der Abb. 6.9 dargestellt.

6.1.3.3 Filterung am Beispiel eines FIR-Filters

Das folgende Beispiel zeigt die Implementierung eines FIR-Filters mit Hilfe eines 8-Bit Mikrocontrollers der Familie ATmega in Ganzzahlarithmetik. Der Mikrocontroller soll Wechselspannungen mit einem Spitzenwert von bis zu 0,5 V filtern. Das analoge Signal wird mit einem externen Tiefpassfilter bandbegrenzt und mit einem DC-Offset von +0,5 V versehen. Der Mikrocontroller tastet mit 5 kHz das entstandene Signal ab und quantisiert es. Das digitale Filter soll eine Grenzfrequenz von 340 Hz, eine Oberwelligkeit von 1 dB im Durchlassbereich und eine Dämpfung von 40 dB bei 816 Hz aufweisen. Die gestellten Bedingungen können mit einem Equiripple-Filter 16. Ordnung erfüllt werden. Wegen der Koeffizienten-Symmetrie sind in der Tab. 6.1 nur die ersten 9 aufgelistet. Die mit Matlab berechneten Koeffizienten werden mit 128 multipliziert und anschließend gerundet.

Tab. 6.1 Filterkoeffizienten

Berechnete Koeffizienten	Ganzzahl Koeffizienten (char cH[k])
-0,011408912567595797	-1
-0,017514503197375211	-2
-0,017666528020283574	-2
-0,0013393806576736487	0
0,036486634909693227	5
0,092260634158284921	12
0,15297263055073201	20
0,20026952064209985	26
0,21814559786716198	28

Die gefilterten Werte müssen nach der Berechnung durch 128 geteilt werden, was eine Verschiebung nach rechts um sieben Stellen bedeutet, damit die Amplitude des gefilterten Signals nicht verfälscht wird. Die Einschränkungen wegen der Ganzzahlarithmetik müssen in einer Simulationsphase untersucht werden, damit die erwünschten Bedingungen vom digitalen Filter erfüllt werden.

Die Abtastperiode wird mit einem Timerinterrupt festgelegt, dessen Serviceroutine (ISR) im Folgenden aufgelistet ist:

```
ISR (TIMER1_COMPA_vect)
{
    uiADCWert = ADC; //der umgewandelte Wert wird zwischengespeichert
    ADCSRA |= (1 << ADSC); //eine neue AD-Wandlung wird gestartet
    ucADCFlag = 1;
}
```

In der Endlosschleife der main-Funktion findet die Filterung statt:

```
#define n 17 //Anzahl der Filterkoeffizienten
//global deklarierte Variablen
volatile uint16_t uiADCWert; //Zwischenspeicher für den
                             //quantisierten Wert
uint16_t uiFIFOin[n]; //Eingangspuffer für die n abgetasteten Werte
int iAktPosFIFOin = n-1; //Zeiger auf die aktuelle Position im
                         //Eingangspuffer (Zeiger 1)
uint8_t ucLvFIFOin; //Laufvariable für den Eingangspuffer (Zeiger 2)
long lYout = 0; //Gefilterter Wert

if(ucADCFlag)
{
    ucADCFlag = 0;
    lYout = 0; //der gefilterte Wert wird initialisiert
```

```

uiFIFOin[iAktPosFIFOin] = uiADCWert; //der abgetastete Wert
                                //wird gespeichert
ucLvFIFOin = iAktPosFIFOin; //die Laufvariable wird neu berechnet
for(uint8_t k = 0; k < n; k++) //k = Zeiger 3
{ //in der Schleife wird der neu gefilterte Wert berechnet
    lYout = lYout + uiFIFOin[ucLvFIFOin] * cH[k];
    ucLvFIFOin++;
    if(ucLvFIFOin == n) ucLvFIFOin = 0;
}
iAktPosFIFOin--; //die aktuelle Position des Zeigers wird
                  //dekrementiert
if(iAktPosFIFOin < 0) iAktPosFIFOin = n - 1;
if(lYout < 0) lYout = 0; //Wertbegrenzung um den Überlauf
                         //zu verhindern
else lYout = lYout / 128; //weil die Koeffizienten um 128
                          //multipliziert wurden
}

```

Nach dem Filtern können die gefilterten Werte lYout weiter verarbeitet werden.

6.1.4 I/O-Steuerlogik

Die von dem digitalen Sensor gemessenen Werte werden in digitaler Form einem Mikrocontroller über eine serielle Schnittstelle zur Verfügung gestellt. Die Sensoren werden als Slaves vorkonfiguriert, was bedeutet, dass sie die Kommunikation nicht selbst initiieren können. Manche digitalen Sensoren besitzen externe Ausgänge, die zum Auslösen eines Interrupts am Mikrocontroller benutzt werden können. Damit wird der Master benachrichtigt, dass ein neuer Messwert aufgenommen wurde oder dass der Messwert eine gestellte Grenze überschritten hat. Die Sensoren besitzen meist eine serielle Schnittstelle wie I²C oder SPI. Der Mikrocontroller als Master initiiert die Kommunikation, liest die Messwerte und überträgt die Einstellungen. Gemäß diesen Einstellwerten steuert die Steuerlogik die Baugruppen des Sensors und parametert ihn. Die I/O-Steuerlogik kann folgende Aktionen durchführen:

- Steuerung der seriellen Schnittstelle;
- Start einer neuen Messung entsprechend dem Messmodus: Einzelmessmodus oder automatischer Messbetrieb;
- Steuerung der Messung, wenn mehrere Messfühler vorhanden sind;
- Speicherung der Messwerte in Registern oder in einem FIFO-Puffer;
- Speicherung der empfangenen Einstellungen in die entsprechenden Register;
- Parametrierung des Sensors entsprechend der gespeicherten Einstellungen;

- Änderung der Messrate mit entsprechender Anpassung der Abtastfrequenz und des digitalen Tiefpasses;
- Bei Bedarf Ausführen einer Temperaturkompensation oder einer DC-Offset-Korrektur der gemessenen Werte;
- Vergleich der gemessenen Werte mit den eingestellten Grenzwerten;
- Steuerung der externen Interrupt-Ausgänge.

6.1.5 Abstraktion der I/O Pins

Im Allgemeinen besitzen Sensoren außer den seriellen Datenleitungen weitere Steuereingänge. Um den Code möglichst leicht portierbar zu halten, empfiehlt es sich, die Verdrahtung dieser Leitungen am Mikrocontroller an einem Punkt zu beschreiben (Board Abstraction Layer in der Software Architektur). Dafür gibt es in dem Software-Modul des Sensors eine Datenstruktur, in der diese Steueranschlüsse aufgeführt sind:

```
typedef struct
{
    tspiHandle SENSORspi;

    volatile uint8_t* Pin1_DDR;
    volatile uint8_t* Pin1_PORT;
    uint8_t Pin1_pin;
    uint8_t Pin1_state;

    volatile uint8_t* Pin2_DDR;
    volatile uint8_t* Pin2_PORT;
    uint8_t Pin2_pin;
    uint8_t Pin2_state;
} SENSOR_pins;
```

Diese Datenstruktur speichert im Falle einer SPI Anbindung die Struktur `tspiHandle` (siehe Kap. 5) für die Ansteuerung des Chip Select Eingangs und die Adressen der DDR- und PORT-Register für die Ansteuerung der Pin1 und Pin2 des Sensors. Somit bleibt die Ansteuerung der Sensoren unabhängig von der Board-Konfiguration. Mit demselben SPI-Modul können damit unterschiedliche Sensoren, die am gleichen Bus angeschlossen sind, angesteuert werden und mit einem Sensor-Softwaremodul mehrere Sensoren vom gleichen Typ. Wenn der Sensor keine ansteuerbaren Pins besitzt, speichert seine SPI-Datenstruktur nur die Datenstruktur für die Ansteuerung des Chip Select Pins.

```
typedef struct
{
    tspiHandle SENSORspi;
} SENSOR_pins;
```

Der Platzhalter SENSOR in den oben genannten Beispielen muss für jeden Sensor anders gewählt werden, es empfiehlt sich, den Namen des Sensors zu verwenden. Für alle Sensoren, die am gleichen Mikrocontroller angeschlossen sind, wird in der Hauptdatei je eine Datenstruktur deklariert und initialisiert.

6.1.6 Ganzzahlarithmetik

Ein 8-Bit Mikrocontroller der Familie ATmega implementiert arithmetische Operationen (Addition, Subtraktion und Multiplikation) mit Ganzzahl- (Integer) und Festkommatypen. Festkomma-Datentypen ermöglichen die Abbildung der rationalen Zahlen als Vielfachen von 2^{-7} oder 2^{-15} . Mit dieser Abbildung werden die reellen Zahlen angenähert. Division ist hardwaremäßig nicht vorgesehen und wird als Bibliotheksfunktion durch Schieben und Anwendung von Schieben, Multiplizieren und Subtrahieren durchgeführt.

In der Programmiersprache C dagegen werden reellen Zahlen mit den Gleitkommatypen *float* und *double* angenähert. Aus den (in Kap. 3) erwähnten Gründen sollten ganzzahlige Datentypen in der Programmierung unbedingt bevorzugt werden. Diese bilden ein begrenztes, geschlossenes Intervall der ganzen Zahlen ab. Dieses Intervall wird als Wertebereich bezeichnet.

6.1.6.1 Mikrocontrollerinterne Zahlenformate

Die Arbeitsregister eines 8-Bit Mikrocontroller der Familie ATmega und die Speicherzellen, die einzeln adressierbar sind, sind alle 8 Bit groß. Um einen 64 kByte großen Speicherbereich adressieren zu können, werden Registerpaare verwendet. Man unterscheidet zwischen vorzeichenlosen (*unsigned*) und vorzeichenbehafteten (*signed*) Ganzzahltypen. Ein vorzeichenbehafteter Ganzzahltyp speichert Ganzzahlen, ein vorzeichenloser Typ nur natürliche Zahlen und die „0“. Tab. 6.2 stellt die interne Zahlendarstellung eines hypothetischen 3-Bit Mikrocontrollers im binären Format dar. Wenn das höchstwertige Bit „1“ ist, wird der gleiche Zahleninhalt unterschiedlich interpretiert. Die Interpretation hängt vom gewählten Ganzzahltyp ab. Das Inkrementieren des höchsten Wertes in der internen Darstellung (alle Bits „1“) verursacht einen Sprung auf den niedrigsten Wert (alle Bits „0“). Beim Dekrementieren des niedrigsten Wertes findet der Sprung auf dem höchsten Wert statt. Dieser Überlauf ist auch bei dem *unsigned*-Datentyp zu beobachten. Beim *signed*-Datentyp findet einen Sprung bei der Änderung des höchstwertigen Bits statt, was als Zweierkomplement-Überlauf bezeichnet wird.

Ein 8-Bit Ganzzahltyp nimmt in der *unsigned*-Darstellung Werte im Intervall [0; 255] an und in der *signed*-Darstellung Werte im Intervall [-128; +127]. Der Mikrocontroller unterstützt den Umgang mit Ganzzahltypen, die breiter als 8-Bit sind, durch das Statusregister SREG:

Tab. 6.2 Mikrocontrollerinterne Zahlendarstellung

Interne binäre Darstellung	111	000	001	010	011	100	101	110	111	000
Unsigned	7	0	1	2	3	4	5	6	7	0
Signed (Zweierkomplement-Darstellung)	-1	0	1	2	3	-4	-3	-2	-1	0
Signed (Einerkomplement-Darstellung)	0	0	1	2	3	-3	-2	-1	0	0

Bit5 – Half carry flag (H) – wird gesetzt, wenn nach einer arithmetischen Operation eine Bytehälfte des Ergebnisses den Wert 9 überschreitet. Dieses Bit wird für die Darstellung der binär codierten Dezimalzahlen verwendet.

Bit4 – Sign bit (S) – wird folgendermaßen berechnet: $S = N \oplus V$, und wird beim Testen von *signed*-Datentypen verwendet.

Bit3 – Two's complement overflow flag (V) – wird gesetzt wenn sich das höherwertige Bit des Ergebnisses einer arithmetischen oder logischen Operation geändert hat. Dieses Bit zeigt ein möglicher Zweierkomplement-Überlauf an.

Bit2 – Negative flag (N) – bildet das höchstwertige Bit des Ergebnisses einer arithmetischen oder logischen Operation ab.

Bit1 – Zero flag (Z) – wird gesetzt, wenn das Ergebnis einer arithmetischen oder logischen Operation Null ist.

Bit0 – Carry flag (C) – wird gesetzt um einen Übertrag an der höherwertigen Stelle zu signalisieren.

Eine ausführliche Beschreibung des Statusregisters ist im Datenblatt [6] zu finden. Die komplette Liste mit den Operationen, die die einzelnen Bits beeinflussen, sind [7] zu entnehmen.

6.1.6.2 Vorzeichenlose Ganzzahltypen

Der Wertebereich eines Ganzzahltyps ohne Vorzeichen, das n -Bit groß ist, ist $[0; +2^n - 1]$. n kann 8, 16 oder 32 sein. Bei diesen Datentypen muss beachtet werden, dass:

$$\begin{aligned} (2^n - 1) + 1 &= 0 \quad \text{bzw. f\"ur } m < (2^n - 1) & (2^n - 1) + m &= m - 1 \\ 0 - 1 &= 2^n - 1 & 0 - m &= 2^n - m \end{aligned} \tag{6.10}$$

Operationen (siehe Gl. 6.10), die als Ergebnis eine Zahl außerhalb des Wertebereiches liefern, f\"uhren zu einem Überlauf-Fehler. Eine sorgf\"altige Analyse des zu implementierenden Algorithmus und die Wahl der passenden Datentypen f\"uhren zu einem kompakten Programm, das schnell und fehlerfrei ausgef\"uhrt wird. F\"ur die folgenden \"Uberlegungen werden zwei Operanden *ucZahl1* und *ucZahl2* und das Ergebnis *ucErgebnis* vom Typ *unsigned char* gew\"ahlt. \"Ahnliche \"Uberlegungen gelten auch f\"ur andere vorzeichenlosen Ganzzahltypen. Folgende F\"alle sollen bei der Benutzung der Ganzzahltypen ohne Vorzeichen ber\"ucksichtigt werden:

Zuweisung	Wenn einer Variablen vom Typ <code>unsigned char</code> ein negativer Wert zugewiesen wird, speichert die Variable nach dem Ausführen der Anweisung (z. B. <code>ucZahl = -1;</code>) das Zweierkomplement des Zahlbetrags (in diesem Beispiel 255).
Addition und Multiplikation	Grundsätzlich kann die Addition und die Multiplikation zweier Variablen <code>ucZahl1</code> und <code>ucZahl2</code> zu einer Bereichsüberschreitung führen. Wird das nicht berücksichtigt, wird das Ergebnis folgendermaßen entstehen: <code>ucErgebnis = (ucZahl1 + ucZahl2) % 256</code> bzw. <code>ucErgebnis = (ucZahl1 * ucZahl2) % 256.</code>
Division	Bei der Ganzzahl Division kann ein Rundungsfehler auftreten.
Vergleich	Die <i>Anweisung1</i> aus dem folgenden Beispiel wird nie ausgeführt, weil die Bedingung als falsch ausgewertet wird für alle Werte von <code>ucZahl1</code> und die <i>Anweisung2</i> wird immer ausgeführt, weil für alle Werte der Variable <code>ucZahl2</code> , die Bedingung wahr ist:

```
if(ucZahl1 < 0) Anweisung1;
if(ucZahl2 > -7) Anweisung2;
```

Schieben Solange der Wertebereich nicht überschritten wird, entspricht das Schieben nach links um n Bits einer Multiplikation mit 2^n . Ansonsten entsteht das Ergebnis wie folgt:

```
ucErgebnis = (ucZahl1 << n) % 256;
```

Das Schieben nach rechts um n Bits entspricht einer Ganzzahldivision mit 2^n .

6.1.6.3 Vorzeichenbehaftete Ganzzahltypen

Die vorzeichenbehafteten Ganzzahldatentypen können 8, 16 oder 32 Bit groß sein, was zu 1, 2 oder 4 Bytes entspricht. Um negative Zahlen darstellen zu können, wird das höchstwertige Bit des Datentyps als Vorzeichen benutzt. Damit wird der Wertebereich des Betrags halbiert, da ein Bit für die Betragsdarstellung wegfällt (Der Wertebereich bei 8 Bit beträgt dann 0 ... 127 und -0 ... -127). Ein Nachteil dieser Darstellung ist, dass die Null doppelt vorhanden ist und eine geschlossene Arithmetik nicht mehr möglich ist.

In der alternativen Einerkomplement-Darstellung entstehen die negativen Zahlen durch bitweise Invertierung einer positiven Zahl. In dieser Darstellung ist der Wert „0“ allerdings ebenfalls doppelt vorhanden, wenn alle Bits „0“ oder alle „1“ sind.

Mikrocontroller verwenden stattdessen das Zweierkomplement für die Darstellung von negativen Zahlen. In dieser Darstellung ist die „0“ eindeutig definiert, wenn alle Bits einer Zahl „0“ sind. Weil die Anzahl der 2^n Kombinationen einer n-Bit Zahl gerade ist, gibt

Tab. 6.3 Zahlenbeispiel für den Zweierkomplement-Überlauf

Operation	Wert vor der Operation	Wert nach der Operation
scZahl++;	+127	-128
scZahl--;	-128	+127

es keine Symmetrie der positiven und negativen Zahlen gegenüber der Null. Der Wertebereich eines n-Bit vorzeichenbehafteten Ganzzahltyps ist $[-2^{n-1}; +2^{n-1} - 1]$. Wenn in dieser Darstellung $-x$ eine negative Zahl und $+x$ ihr Betrag ist, dann gilt die bekannte Gleichung:

$$x + (-x) = 0 \quad (6.11)$$

Um die Darstellung einer negativen Zahl im Zweierkomplement zu berechnen wird die positive Zahl bitweise invertiert und zum Ergebnis eine „1“ addiert. Ähnlich geht man vor, wenn man den Betrag einer negativen Zahl, wie im folgenden Beispielcode berechnet:

```
unsigned char uczahl;
signed char scZahl = -5; // -510 = 1111 10112
ucZahl = ~scZahl; // Ergebnis: ucZahl = 0000 01002 = 410
ucZahl = ucZahl + 1; // Ergebnis: ucZahl = 0000 01012 = +510
```

Die Gl. 6.12 stellt den Zweierkomplement-Überlauf in mathematischer Form dar. Ein Zahlenbeispiel für solch einen Zweierkomplement-Überlauf ist in der Tab. 6.3 veranschaulicht. Die Variable scZahl ist vom Typ signed char.

$$\begin{aligned} (2^{n-1} - 1) + 1 &= -2^{n-1} \\ (-2^{n-1}) - 1 &= (2^{n-1} - 1) \end{aligned} \quad (6.12)$$

In Tab. 6.4 werden Operationen mit vorzeichenbehafteten, 8-Bit Variablen, die zu einer Fehlinterpretation oder zu einem unerwarteten Ergebnis führen, erläutert. Die Variable scZahl ist vom Typ signed char und n ist eine natürliche Zahl.

Ein Rundungsfehler findet auch bei der Division der vorzeichenbehafteten Datentypen statt. Der Tab. 6.4 ist der Unterschied zwischen der Ganzahldivision und der Rechtsschiebung der negativen Werte zu entnehmen.

6.1.6.4 Erkennung und Verhinderung eines Überlaufs

Die vom Überlauf verursachten Fehler haben besonders in rekursiven Algorithmen gravierende Konsequenzen als Rundungsfehler. Ein Überlauf, der beim Rechnen eines FIR-Filters auftritt, wird im gefilterten Signal eine Einzelstörung in Form eines Sprungs mit der Dauer einer Abtastperiode hervorrufen. Ein Überlauf beim Rechnen eines IIR-Filters wird das Ausgangssignal zum Schwingen bringen. Das Abklingen des Schwingens ist stark von der Filterordnung abhängig. Je höher die Filterordnung, desto länger dauert das Schwingen.

Tab. 6.4 Operationen mit vorzeichenbehafteten Ganzzahlen

Operation	Operanden	Anweisung	Ergebnis
Zuweisung	scZahl +127 < n < 256	scZahl = n;	scZahl = n - 256 Die interne Darstellung von scZahl und n sind gleich
Division	scZahl = -(2n + 1)	scZahl = scZahl / 2;	scZahl = -n
	scZahl = -(2n)		scZahl = -n
	scZahl = -1		scZahl = 0
Verschiebung nach rechts	scZahl = -(2n + 1)	scZahl = scZahl >> 2;	scZahl = -(n + 1)
	scZahl = -(2n)		scZahl = -n
	scZahl = -1		scZahl = -1
Multiplikation	scZahl = -128	scZahl = scZahl * (-1);	scZahl = -128
Vergleich	scZahl +127 < n < +255	if (scZahl > n) Anweisung;	Die Anweisung wird nie ausgeführt weil die Bedingung immer falsch ist

Mit Hilfe der vom Statusregister gelieferten Informationen kann ein Überlauf erkannt werden. Um einen Überlauf erkennen zu können, muss jedes Ergebnis getestet werden. Dies schlägt sich negativ auf die Programmkomplexität und auf die Ausführungszeit nieder. Um bei der Erkennung eines Überlaufs dessen Auswirkung zu minimieren, wird dem Ergebnis abhängig von der Überlaufrichtung eine der Grenzen des Wertebereiches zugewiesen. Diese Methode wird als Sättigung bezeichnet.

Um einen Überlauf grundsätzlich zu verhindern werden entweder Variablen mit einem größeren Wertebereich gewählt oder alle Eingangswerte werden durch die gleiche Konstante dividiert. Diese zweite Option ist als Skalierung bekannt.

In sicherheitsrelevanten Anwendungen ist unbedingt nach wichtigen Operationen eine Plausibilitäts- und Wertebereichsprüfung durchzuführen. In der Tat haben Wertebereichsüberschreitungen bereits zu dramatischen Unfällen geführt, zum Beispiel dem Absturz der Ariane Rakete beim Jungfernflug 501 im Jahr 1996⁷.

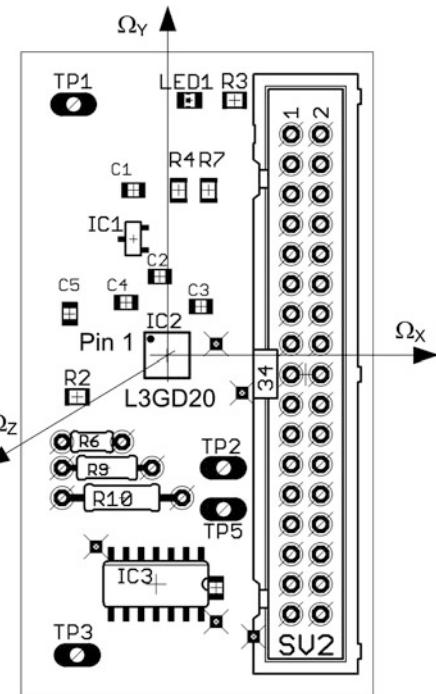
6.2 Gyroskop

Ein digitales Gyroskop ist ein integrierter Schaltkreis, der ein digitales Signal proportional zur Winkelgeschwindigkeit um eine oder mehreren Achsen liefert. Der Sensor L3GD20 [8, 9] misst die Winkelgeschwindigkeit um drei Achsen mit der positiven Messrichtung gegen den Uhrzeigersinn, wie in Abb. 6.10 dargestellt. Der Sensor bietet drei Messbereiche an: ± 250 , ± 500 und ± 2000 grad/s. Ein integrierter Temperatursensor misst

⁷ Vgl. <http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf> bzw. <http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html>.

Abb. 6.10 L3GD20 Achsen

Anordnung



jede Sekunde die Temperatur mit einer Auflösung von $1\text{ }^{\circ}\text{C}$ und kann benutzt werden, um die Temperaturentwicklung zwischen zwei Messzeiten zu bestimmen. Der gesamte Drehwinkel kann als Integral der Winkelgeschwindigkeit berechnet werden. Über einen konfigurierbaren Hochpassfilter kann der DC-Anteil des Signals entfernt und somit die Winkelbeschleunigung ermittelt werden. Über diesen Hochpassfilter kann auch der Einfluss der hochfrequenten Vibrationen reduziert werden. Die Konfiguration des Schaltkreises und das Auslesen der gemessenen Werte können über SPI oder über I²C erfolgen. Ein aufwendiges Interrupt Konzept mit zwei einstellbaren Ausgängen kann zur Entlastung des steuernden Mikrocontrollers führen.

6.2.1 Beschaltung des L3GD20

In Abb. 6.11 wird beispielhaft die Beschaltung eines Sensors L3GD20 mit einem Mikrocontroller der ATmegax8 Familie über SPI im 4-wire Modus dargestellt. Die zwei Baugruppen werden mit unterschiedlichen Spannungen versorgt und die Pegelanpassung wird mit Schmitt-Triggern realisiert. Über die pull-up-Widerstände soll in der Initialisierungsphase verhindert werden, dass eine I²C Kommunikation aus Versehen initiiert wird.

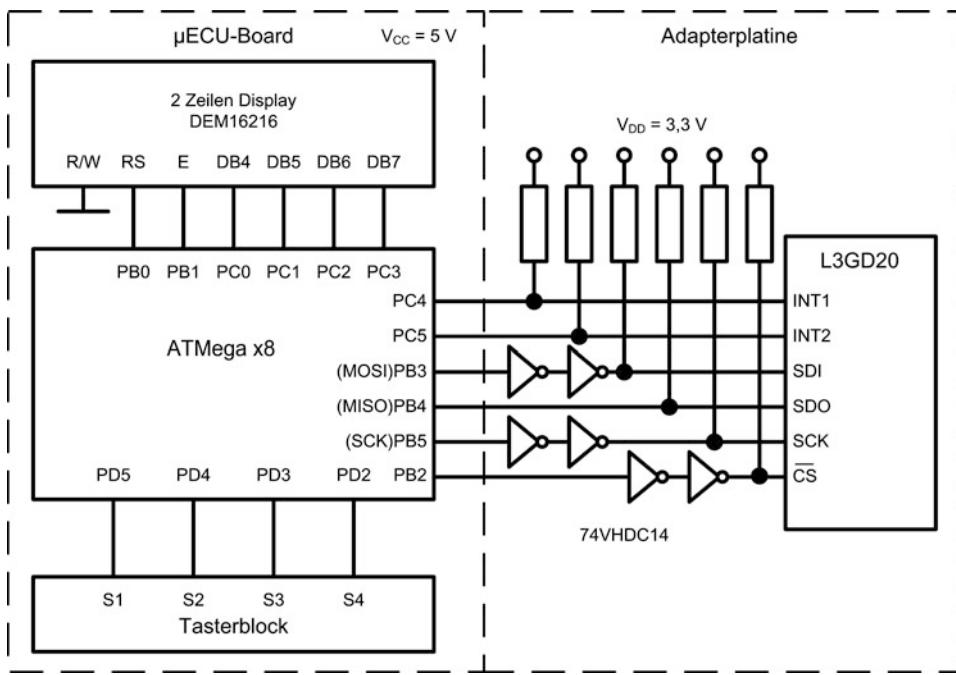


Abb. 6.11 L3GD20 Beschaltungsbeispiel

6.2.2 Kommunikationsschnittstellen

Für die Kommunikation mit der Außenwelt stellt der Sensor zwei Interrupt Ausgänge und vier Schnittstellenanschlüsse zur Verfügung, deren Bedeutung in Tab. 6.5 zu finden ist.

Der Baustein besitzt zwei serielle Schnittstellen: I²C und SPI die, die gleichen Anschlüsse benutzen. Eine I²C Kommunikation mit bis zu 400 kBit/s kann über die Pins SDA und SCL stattfinden, wenn der Pin CS auf High geschaltet ist. Der SDO Anschluss agiert bei dieser Übertragung als Adress-Eingang und ermöglicht den Anschluss zweier Sensoren am gleichen I²C-Bus. Diese Schnittstelle wird während einer SPI-Übertragung deaktiviert.

Tab. 6.5 L3GD20 Kommunikationsanschlüsse

Anschluss		I ² C	SPI 4-wire	SPI 3-wire
CS	0	Inaktiv	Aktiv	
	1	Aktiv	Inaktiv	
SCL/SPC		Clock Eingang		
SDA/SDI/SDO		Datenanschluss	Dateneingang	Daten Ein/Ausgang
SDO		Device Chip Adresse	Datenausgang	Nicht benutzt

Im SPI-Modus findet die Kommunikation mit bis zu 10 Mbit/s statt, während der CS-Eingang auf Low geschaltet ist. Im SPI 3-wire-Modus wird ein einziger Anschluss für die bidirektionale Datenübertragung benutzt wie in Abschn. 7.4 beschrieben. Dieser Modus wird durch Setzen des Bit 0 `SIM` im Register `CTRL_REG4` aktiviert.

Im Folgenden behandeln wir die Ansteuerung des Sensors nur im klassischen SPI 4-wire-Modus mit getrennten Anschlüssen für den Dateneingang und -ausgang. Die Kommunikation mit dem Sensor findet im SPI-Modus 3 statt und das höchstwertige Bit wird zuerst übertragen, siehe Abb. 6.13.

Für die Beispielschaltung aus Abb. 6.11 lautet die Datenstruktur zur Ansteuerung der Pins, inklusive des Chip-Select Eingangs (Abschn. 6.1.5):

```
typedef struct
{
    tspiHandle L3GD20spi;
} L3GD20_pins;
```

Diese Struktur muss im Main-Modul befüllt werden, damit sie von überall benutzt werden kann.

```
L3GD20_pins L3GD20_1 = {{/*CS_DDR*/      &DDRB,
                           /*CS_PORT*/      &PORTB,
                           /*CS_pin*/       PB2,
                           /*CS_state*/     ON}}; //ON = 1
```

Die hier belegten vier Werte entsprechen genau der Datenstruktur `tspiHandle`, die in der Struktur `L3GD20_pins` an erster Stelle steht. Alle Register des Bausteins sind 8 Bit groß und einzeln adressierbar. Um den Inhalt eines Registers zu lesen oder zu schreiben, sendet der Master, nachdem er die Slave Select Leitung auf Low geschaltet hat, ein Startbyte mit der Konfiguration aus Tab. 6.6.

Das höchstwertige Bit des Startbytes bestimmt die Richtung des Datentransfers. Ist dieses Bit „0“, so wird ein Schreibvorgang gestartet, ansonsten ein Lesevorgang. Der Mikrocontroller sendet nach dem Startbyte ein Datenbyte beim Schreiben oder ein beliebiges (Dummy) Byte beim Lesen. In einer SPI-Sequenz können dem Startbyte mehrere Datenbytes oder Dummys folgen. Wenn das Bit 6 (*MS*) „0“ ist, überschreibt ein neues Datenbyte das alte, mit jedem Dummy-Byte wird der Inhalt des adressierten Registers neu gelesen. Wenn das Bit *MS* gesetzt ist, wird der interne Adresszähler nach jeder Übertragung eines Bytes inkrementiert und das Lesen oder Schreiben eines gesamten Bereiches von zusammenhängenden Registern in einem einzigen Vorgang ermöglicht. Die Bit 5:0 aus dem

Tab. 6.6 L3GD20 Konfiguration des Startbytes bei der SPI-Übertragung

Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
R/W	MS	Register-Adresse					

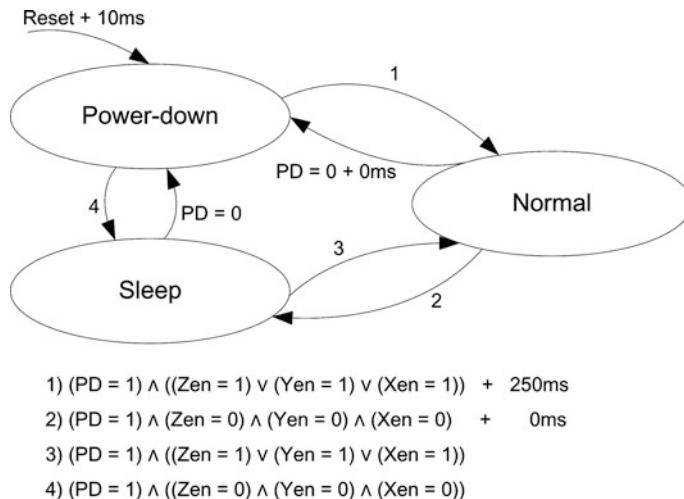
Startbyte speichern die Adresse des ersten Registers aus dem Bereich. Ein Beispiel einer solchen Lese-Funktion ist hier aufgelistet:

```
void L3GD20_Read_RegBlock(L3GD20_pins sdevice_pins,
                           volatile uint8_t* ucarray2read,
                           uint8_t ucfirst_address, uint8_t ucbyte_number)
{
    uint8_t ucAddress = 0, ucDummy = 0x80, ucI;
    /*READ = 0x80 bedeutet ein Lesebefehl
    INKR_ON = 0x40 führt zum Inkrementieren des internen Adresszählers
    nach jedem ausgelesenen Register
    ucfirst_address die erste Adresse des auszulesenden Registerbereiches*/
    ucAddress = ucfirst_address | READ | INKR_ON;
    SPI_Master_Start(sdevice_pins.L3GD20spi); //die Slave Select-Leitung
                                                //wird auf Low gesetzt
    SPI_Master_Write(ucAddress); //die Adresse des Registers im
                                 //Read-Modus wird übertragen
    for(ucI = 0; ucI < ucbyte_number; ucI++)
    {
        //das ausgelesene Byte wird in das Array ucarray2read gespeichert
        ucarray2read[ucI] = SPI_Master_Write(ucDummy);
    }
    //die Slave Select-Leitung wird auf High gesetzt und somit die SPI-
    //Übertragung beendet
    SPI_Master_Stop(sdevice_pins.L3GD20spi);
}
```

Beim Aufrufen der Funktion müssen die Datenstruktur (`sdevice_pins`), die Adresse des Arrays, in dem die Daten gespeichert werden sollen (`ucarray2read`), die Adresse des ersten Registers aus dem auszulesenden Bereich (`ucfirst_address`) und die Zahl der auszulesenden Register (`ucbyte_number`) als Parameter übergeben werden.

6.2.3 Arbeitsmodi

Etwa 10 ms nach dem Einschalten befindet sich der Sensor im Power-down-Modus (Abb. 6.12), in dem der Stromverbrauch auf ca. 5 µA reduziert wird. Lediglich die Kommunikationsschnittstellen werden in diesem Modus versorgt. Im Sleep-Modus verbraucht der Sensor ca. 2 mA, wesentlich mehr als im Power-down-Modus, die Umschaltung zum Normal-Modus findet aber viel schneller statt. Mit dem Setzen des Bits *Power_down* aus dem Register *CTRL_Reg1* und der Aktivierung wenigstens einer Messachse wird der Sensor in den Normal-Modus umgeschaltet. In diesem Modus werden die Winkelgeschwindigkeiten kontinuierlich mit einer zuvor eingestellten Messfrequenz gemessen

**Abb. 6.12** L3GD20 Arbeitsmodi Zustandsübergangsdiagramm**Tab. 6.7** Grenzfrequenzen des Hochpassfilters (HPF) und der Tiefpassfilter (LPF1 und LPF2)

Grenzfrequenz LPF2 [Hz]		DR1:0			
		00	01	10	11
BW1:0	00	12,5	12,5	20	30
	01	25	25	25	35
	10	25	50	50	50
	11	25	70	100	100
Grenzfrequenz LPF1 [Hz]		32	54	78	93
Abtastfrequenz [Hz]		95	190	380	760
Grenzfrequenz HPF [Hz]					
HPCF3:0	0000	7,2	13,5	27	51,4
	0001	3,5	7,2	13,5	27
	0010	1,8	3,5	7,2	13,5
	0011	0,9	1,8	3,5	7,2
	0100	0,45	0,9	1,8	3,5
	0101	0,18	0,45	0,9	1,8
	0110	0,09	0,18	0,45	0,9
	0111	0,045	0,09	0,18	0,45
	1000	0,018	0,045	0,09	0,18
	1001	0,009	0,018	0,045	0,09

(siehe Tab. 6.7). Die Umschaltzeiten die in der Abb. 6.12 nicht eingegeben sind, betragen sechs Abtastperioden oder eine, abhängig davon ob ein interner Tiefpassfilter LPF2 aktiviert ist oder nicht.

Tab. 6.8 Hochpass Modi

HPM1	HPM0	Hochpass Modus
x	0	Normal Modus; der DC-Anteil der Messwerte wird beim Auslesen des <i>REFERENCE</i> Registers (Adresse 0x25) auf „0“ gesetzt
0	1	Referenz Modus; aus dem Messwert wird der Inhalt des <i>REFERENCE</i> Registers subtrahiert, bevor er im Ausgangsregister gespeichert wird
1	1	Autoreset-Modus; der DC-Anteil der Messwerte wird auf „0“ gesetzt beim Auslösen eines Interrupts

6.2.3.1 Winkelgeschwindigkeitsmessung

Die MEMS-Messfühler des Sensors liefern analoge Spannungen, die verstärkt werden und deren Frequenzspektren mit einem Tiefpassfilter LPF1 begrenzt werden, damit kein Alias Effekt auftritt. Mit der Wahl der Messfrequenz stellt sich automatisch die Grenzfrequenz des Filters LPF1 ein. Man kann die gemessenen Werte zusätzlich mit einem zweiten Tiefpassfilter LPF2 und/oder einem Hochpassfilter HPF filtern, bevor sie intern gespeichert werden. Dadurch können hochfrequente und/oder niederfrequente Anteile der Signale herausgefiltert werden.

Über das Steuerregister *CTRL_REG1* mit der Adresse 0x20 werden der Arbeitsmodus des Sensors, die Abtastrate und die Grenzfrequenzen der Tiefpassfilter LPF1 und LPF2 gewählt:

Bit 7:0 – DR 1:0 – Auswahl der Abtastfrequenz gemäß Tab. 6.7

Bit 5:4 – BW 1:0 – Auswahl der Grenzfrequenz des Tiefpassfilters LPF2 gemäß Tab. 6.7

Bit 3 – PD = „0“ – der Sensor befindet sich im Power-down-Modus

Bit 2 – Zen = „1“ – Aktivieren der Z-Achse

Bit 1 – Yen = „1“ – Aktivieren der Y-Achse

Bit 0 – Xen = „1“ – Aktivieren der X-Achse

Wenn der Sensor im Normal-Modus arbeitet, wird nach jeder Abtastperiode für jede freigeschaltete Achse jeweils ein Wert aufgenommen, der mit einem 16-Bit A/D-Wandler quantisiert wird und die Messwerte werden im Zweierkomplement gespeichert. Das Steuerregister *CTRL_REG2* (0x21) bestimmt den Arbeitsmodus des Hochpasses und dessen Grenzfrequenz (siehe Tab. 6.7) in Abhängigkeit von der ausgewählten Abtastfrequenz:

Bit 7:6 – müssen beide auf „0“ gesetzt werden

Bit 5:4 – HPM 1:0 – bestimmen den Arbeitsmodus des Hochpasses gemäß Tab. 6.8

Bit 3:0 – HPCF 3:0 – Grenzfrequenz-Konfiguration des Hochpasses gemäß Tab. 6.7

6.2.3.2 Zwischenspeichern der Messwerte

Nach der Aufnahme können die Messwerte für jede Achse vor dem Speichern zusätzlich gefiltert werden. Mit dem Bit 4 (*HPEN*) und den Bits 1:0, (*Out_Sel 1:0*) aus dem

Tab. 6.9 L3GD20 Ausgangsregister mit deren Adressen

Messachse	Höherwertiges Byte	Niederwertiges Byte
X	OUT_X_H 0x29	OUT_X_L 0x28
Y	OUT_Y_H 0x2B	OUT_Y_L 0x2A
Z	OUT_Z_H 0x2D	OUT_Z_L 0x2C

Steuerregister *CTRL_REG5* (0x24) kann die Art der Filterung vor dem Speichern gewählt werden:

- OUT1_SEL1:0 = „00“ – keine Filterung
- OUT1_SEL1:0 = „01“ – Filterung mit dem Hochpass HPF
- HPEN = „0“ OUT1_SEL1 = „1“ – Filterung mit dem Tiefpass LPF2
- HPEN = „1“ OUT1_SEL1 = „1“ – Filterung mit einem Bandpass bestehend aus dem Tiefpass LPF2 und dem Hochpass HPF; die Grenzfrequenz des Tiefpasses muss höher sein als die des Hochpasses.

Der gemessene Wert für eine Achse kann entweder in ein 16 Bit großes Ausgangsregister oder in einen der 32x16-Bit großen FIFO⁸-Datenpuffer gespeichert werden.

Direktes Speichern

In diesem Modus werden die drei FIFO-Puffer deaktiviert. Die Messwerte werden für die Dauer einer Abtastperiode in die Ausgangsregister gespeichert, danach werden sie überschrieben. Die Konfiguration dieser Register im Little-Endian-Format⁹ ist in Tab. 6.9 aufgelistet. In diesem Speicherformat wird das niederwertige Byte des Messwertes an der Anfangsadresse gespeichert und dann das höchstwertige Byte. Das Speicherformat der Register kann auf Big-Endian umgestellt werden, indem das Bit 6, *BLE* des Steuerregisters *CTRL_REG4* (0x23) gesetzt wird. Diese Ausgangsregister sind Teil der FIFO-Puffer und speichern den ältesten ungelesenen Messwert.

Gepuffertes Speichern

Das Setzen des Bit 6, *FIFO_EN* im Steuerregister *CTRL_REG5* (0x24) aktiviert das Speichern der Messwerte in den Puffern. Um das Auslesen der Puffer zu optimieren und flexibel zu gestalten, können über das Register *FIFO_CTRL_REG* (0x2E) mehrere Speichermodi konfiguriert werden:

Bit 7:5 – FM 2:0 – bestimmen den FIFO-Modus

Bit 4:0 – WTM 4:0 – diese Bits speichern die Füllgrenze der Datenpuffer [0, 30]; beim Überschreiten dieser Füllgrenze kann ein Interrupt ausgelöst werden.

⁸ FIFO – first in first out.

⁹ Little-Endian-Format – in einem zwei-Byte Wort wird das niedlerwertige Byte an der ersten Adresse gespeichert, im Big-Endian-Format wird zunächst das höherwertige Byte gespeichert.

Die Informationen über den Füllstand der Datenpuffer werden von dem Sensor im Nur-Lese-Register *FIFO_SRC_REG* (0x2F) gespeichert:

Bit 7 – WTM – dieses Bit zeigt an, wenn es gesetzt ist, dass die Anzahl der ungelesenen Messwerte aus dem Puffer die eingestellte Grenze (WTM 4:0) überschritten hat

Bit 6 – OVR – wird gesetzt, wenn der Datenpuffer voll ist

Bit 5 – EMPTY – wird gesetzt, wenn in den Puffern keine ungelesenen Messwerte sind

Bit 4:0 – FSS 4:0 – dieser Zähler zählt die ungelesenen Messwerte aus dem Datenpuffer

Bypass-Modus (*FIFO_EN* = „1“ und *FM2:0* = „000“)

Wird das Bit *FIFO_EN* gesetzt und die Bits *FM 2:0* zurückgesetzt, wird der Bypass-Modus aktiviert und der gesamte Inhalt der drei Datenpuffer gelöscht. Die Messwerte werden wie beim direkten Speichern über die Dauer einer Abtastperiode in den Ausgangsregistern gespeichert. Dieser Modus wird verwendet um zwischen den anderen FIFO-Modi umzuschalten.

FIFO-Modus (*FIFO_EN* = „1“ und *FM2:0* = „001“)

In diesem Modus füllen die Messwerte den Datenpuffer voll. Mit der Überschreitung der eingestellten Füllgrenze oder spätestens beim vollfüllen der Puffer kann ein Interrupt ausgelöst werden um dem Master den Beginn des Auslesens zu signalisieren. Die ältesten gespeicherten Messwerte stehen in den Ausgangsregistern zum Auslesen bereit. Um weitere Messwerte in diesem Modus speichern zu können, muss zuerst der Bypass- und gleich danach der FIFO-Modus aktiviert werden.

Stream-Modus (*FIFO_EN* = „1“ und *FM2:0* = „010“)

In diesem Modus wird mit jedem neuen Messwert, der im Puffer gespeichert wird, der Zähler *FSS 4:0* inkrementiert. Beim Auslesen des Ausgangsregisters wird der älteste ungelesene Messwert übertragen und der Zähler *FSS 4:0* dekrementiert. Wenn die Kapazität des Puffers erreicht ist und kein Speicherplatz mehr frei ist, wird der älteste Messwert überschrieben.

In den Modi Stream-to-FIFO (*FIFO_EN* = „1“ und *FM 2:0* = „011“) und Bypass-to-Stream (*FIFO_EN* = „1“ und *FM 2:0* = „100“) kann der Sensor zwischen Stream- und FIFO-Modus, bzw. zwischen Bypass- und Stream-Modus umschalten, um nähere Informationen über den Verlauf eines Interrupts zu gewinnen.

6.2.3.3 Auslesen der Messwerte

Die gespeicherten Messwerte werden immer aus den Ausgangsregistern (Tab. 6.9) ausgelesen. Das Auslesen kann seriell entweder über SPI oder I²C stattfinden. Bei der Wahl der Schnittstelle und des Bustaktes muss die gewählte Messfrequenz berücksichtigt werden.

Auslesen der direkt gespeicherten Messwerte

Wenn alle drei Messachsen frei geschaltet sind, müssen innerhalb einer Abtastperiode (1/ODR) drei Messwerte a 2 Bytes, also insgesamt 48 Datenbits übertragen werden, da-

zu kommen noch die Steuerbits. Um festzustellen ob ein neuer Datensatz vorhanden ist, kann das Bit 3 (ZYXDA) vom Register *STATUS_REG* (Adresse 0x27) im *polling*, also regelmäßig in kurzen Zeitabständen, abgefragt werden. Der steuernde Mikrocontroller wird entlastet, wenn der Sensor so konfiguriert wird, dass ein vollständig abgeschlossener Messvorgang am INT2-Ausgang signalisiert wird. Wenn das Bit *I2_DRDY* im Register *CTRL_REG3* gesetzt ist, so wird der Ausgang INT2 auf High gesetzt sobald ein neuer Datensatz vorhanden ist, und zurückgesetzt nachdem ein Messwert vollständig ausgelesen wurde. Die neuen Messwerte können für jede Achse einzeln gelesen werden wie in Abb. 6.13a oder für alle drei Achsen in einem Lesevorgang wie in Abb. 6.13b. In der Abb. 6.13b sind die zeitlichen Abläufe von Lesevorgängen mehrerer Registern über SPI dargestellt. Das gesetzte Bit 7 des Steuerbytes codiert den Lesevorgang, das gesetzte Bit 6 führt zum Inkrementieren der Adresse nach jedem gelesenen Byte und die Bits 5:0 bilden die Anfangsadresse des Registerblocks.

In der Abb. 6.13a wird beispielhaft die Übertragung des Messwertes der Z-Achse dargestellt. Wenn der Inhalt des Registers *OUT_Z_L* (mit der niederwertigen Adresse) in die Variable *ucZValue1* und der Inhalt des Registers *OUT_Z_H* in die Variable *ucZValue2* gespeichert wird, dann wird abhängig von dem Speicherformat der gemessene 16 Bit Wert *iZValue* folgendermaßen berechnet:

```
//unsigned char ucZValue1, ucZValue2;
//int iZValue;
//Bit6, BLE aus dem Register CTRL_REG4 gleich "0", Little-Endian Format
iZValue = ucZValue1 + (ucZValue2 << 8);
//Bit6, BLE aus dem Register CTRL_REG4 gleich "1", Big-Endian Format
iZValue = ucZValue2 + (ucZValue1 << 8);
```

Auslesen der gepufferten Messwerte

Das gepufferte Speichern der Messwerte bietet den Vorteil, dass das Auslesen der neuen Messwerte nicht unmittelbar nach deren Speichern erfolgen muss. Die gewählte Strategie muss aber einen Überlauf der Datenpuffer verhindern, wenn alle Messwerte gelesen werden sollen. Von Vorteil ist die Auswahl eines Modus, bei dem der Lesebeginn über einen externen Interrupt signalisiert wird und die Anzahl der auszulesenden Datensätze bekannt ist. So wird die Abfrage nach neuen Datensätzen überflüssig. Nach der Übertragung eines Datensatzes wird der nächste Wert aus dem Puffer in das Ausgangsregister verschoben. Im FIFO-Modus (siehe Abschn. 6.2.3.2, „FIFO-Modus (*FIFO_EN* = ,1‘ und *FM2:0* = ,001‘)“) wird ein Interrupt ausgelöst sobald der Datenpuffer voll ist. In diesem Modus muss innerhalb der nächsten Abtastperiode, die dem Interrupt folgt, der gesamte Datenpuffer ausgelesen und der Sensor für eine neue Aufnahme vorbereitet werden. Im Folgenden wird das Auslesen der Messwerte im FIFO-Modus mit einem Mikrocontroller der ATmega-Familie über SPI näher betrachtet:

Mikrocontroller Taktfrequenz: $f_q = 16 \text{ MHz}$
 maximale SPI-Bitrate: $f_{\text{Bit}} = f_q / 4 = 4 \text{ MHz}$

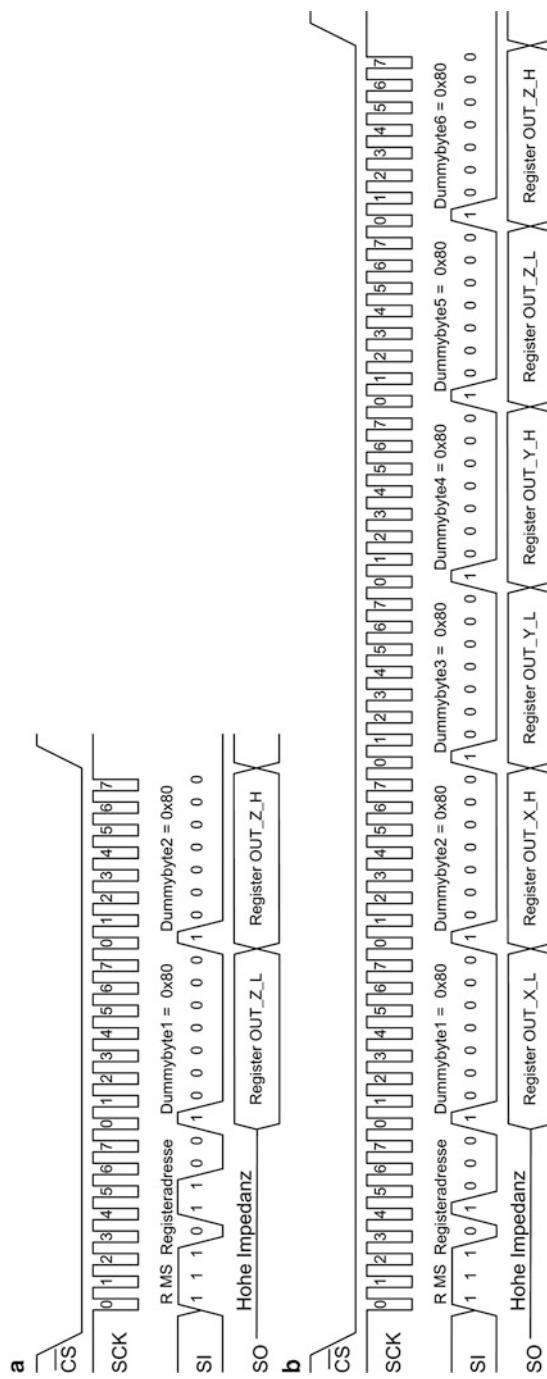


Abb. 6.13 L3GD20 Auslesen der Ausgangsregister

minimale Bitdauer:	$T_{\text{Bit}} = 1 / f_{\text{Bit}} = 250 \text{ ns}$
höchste Datenrate des Sensors:	$\text{ODR}_{\text{max}} = 760 \text{ Hz}$
kürzeste Abtastperiode:	$T_A = 1 / \text{ODR}_{\text{max}} \approx 1,315 \text{ ms}$
FIFO Speicherzeit:	$T_{\text{FIFO}} = 32 \times T_A = 42,1 \text{ ms}$
gesamte Bitzahl:	$N_{\text{Bit}} = (1 \text{ Befehl-Byte} + 3 \text{ Achsen} \times 2 \times 32 \text{ Byte}) \times 8$ Bit = 1544 Bit
gesamte Übertragungsdauer:	$T_D = T_{\text{BIT}} \times N_{\text{Bit}} = 386 \mu\text{s}$
Restzeit:	$\Delta t = T_A - T_D = 929 \mu\text{s}$

In der Restzeit von ca. 929 µs muss der Sensor vom FIFO-Modus auf den Bypass-Modus und wieder auf den FIFO-Modus umgeschaltet werden, um weitere Messwerte aufnehmen zu können. In der Zeit bis T_{FIFO} erreicht wird, kann der Mikrocontroller die Daten verarbeiten und weitere Aufgaben erledigen.

Im Stream-Modus kann ein Interrupt ausgelöst werden, wenn die eingestellte Füllgrenze erreicht ist. Die Vorteile dieses Modus bestehen darin, dass für die Übertragung der ungelesenen Datensätze mehr Zeit zur Verfügung steht und die Aufnahme der Daten und das Speichern in den Datenpuffern kontinuierlich läuft.

Die gepufferten Messwerte können nach dem Beispiel aus Abb. 6.13 gelesen werden. Als Anfangsadresse wird die erste Adresse aus dem Ausgangsregisterbereich (0x28) gewählt, für jeden gelesenen Messwert muss der Master zwei beliebige Dummy-Bytes übertragen. Nach jedem gelesenen Byte wird der interne Adresszähler des Sensors inkrementiert; nach der letzten Adresse aus diesem Bereich (0x2D) springt der Adresszähler automatisch auf die erste.

6.2.3.4 Interruptsteuerung

Dank einer komplexen Interrupt-Struktur des Sensors kann ein Mikrocontroller als Master große Datenmengen ohne blockierendes Warten auslesen und zeitnah die Winkelgeschwindigkeit oder Winkelbeschleunigung überwachen um Regelungsaufgaben zu lösen. Der Baustein besitzt zwei Anschlüsse, die als Interrupt-Ausgänge konfiguriert werden können. Interrupt 1 entsteht als Folge der Überwachung der Messwerte, während Interrupt 2 abhängig vom Füllstand des FIFO-Puffers ausgelöst wird, oder wenn ein neuer Messwert vorhanden ist. Die Interrupt-Modi werden vom Steuerregister *CTRL_REG3* (0x22) verwaltet:

Bit 7 – I1_Int1 – mit dem Setzen dieses Bits wird der Interrupt 1 freigegeben;

Bit 6 – I1_Boot – wenn dieses Bit gesetzt ist, zeigt Pin INT1 das Booten des Bausteins an;

Bit 5 – H_L_active – konfiguriert den Pegel des INT1 Ausgangs beim Auslösen eines Interrupts;

Bit 4 – PP_OD = „0“ / „1“ – der Ausgang INT1 wird als push-pull oder open-drain konfiguriert;

- Bit 3 – I2_DRDY = „1“** – ein Interrupt 2 wird ausgelöst, wenn ein neuer Messwert vorhanden ist;
- Bit 2 – I2_WTM = „1“** – ein Interrupt 2 wird ausgelöst, wenn die Füllgrenze des FIFO überschritten wird;
- Bit 1 – Ovrn = „1“** – ein Interrupt 2 wird ausgelöst, wenn die Kapazität des FIFO erreicht ist;
- Bit 0 – I2_Empty = „1“** – ein Interrupt 2 wird ausgelöst, wenn im FIFO keine ungelesenen Messwerte sind.

Wenn von Bit 2:0 mehrere Bits gleichzeitig gesetzt sind, so wird die Ver-ODER-ung der dadurch ausgewählten FIFO-Ereignisse einen Interrupt auslösen. Über das Register *INT1_CFG* (0x30) (siehe Tab. 6.10) werden die Interrupt-Ereignisse und deren logischen Verknüpfungen, die den Interrupt 1 auslösen, ausgewählt:

- Bit 7** – AND/OR = „1“ – die ausgewählten Ereignisse werden Ver-UND-et, ansonsten Ver-ODER-t;
- Bit 6** – LIR = „0“ – der Interrupt 1 Zustand wird automatisch gelöscht, wenn das Ereignis, das dazu geführt hat, verschwindet. Bei LIR = „1“ bleibt dieser Zustand bis zum nächsten Zugriff auf das Register *INT1_SRC* erhalten, das die Interrupt 1 auslösenden Ereignisse speichert.
- Bit 5:0** – wenn ein Bit gesetzt ist, kann ein Interrupt 1 ausgelöst werden, wenn die Messwerte einer Achse (X, Y oder Z) die eingestellte Wertgrenze überschritten („H“) oder unterschritten („L“) haben.

Das Ereignis das ein Interrupt 1 auslöst (Tab. 6.11), wird in das Register *INT1_SRC* (0x31) gespeichert. Dieses Register kann nur gelesen werden und hat folgende Konfiguration:

- Bit 7** – ist immer „0“;
- Bit 6** – IA – wird gesetzt, wenn ein Interrupt 1 ausgelöst wurde;
- Bit 5:0** – werden intern gesetzt, wenn die Messwerte einer Achse (X, Y oder Z) die gesetzten Grenzwerte über- oder unterschreiten.

Tab. 6.10 Interrupt1 Konfigurationsregister

AND/OR	LIR	ZHIE	ZLIE	YHIE	YLIE	XHIE	XLIE
--------	-----	------	------	------	------	------	------

Tab. 6.11 INT1_SRC Register

0	IA	ZH	ZL	YH	YL	XH	XL
---	----	----	----	----	----	----	----

Tab. 6.12 L3G20 Grenzwertregister

Achse	Höherwertiges Byte	Niederwertiges Byte
X	INT1_THS_XH 0x32	INT1_THS_XL 0x33
Y	INT1_THS_YH 0x34	INT1_THS_YL 0x35
Z	INT1_THS_ZH 0x36	INT1_THS_ZL 0x37

Grenzwerteinstellung für den Interrupt 1

Um eine Überwachung der Messwerte zu gewährleisten gibt es für jede Achse jeweils ein 16-Bit-Register, in dem der Betrag des Grenzwertes gespeichert wird. Der interne A/D-Wandler hat eine 16-Bit-Auflösung, gemessen wird aber in beiden Drehrichtungen, deshalb kann der Betrag höchstens 15 Bit groß sein. In der Tab. 6.12 sind die Grenzwertregister-Paare und Ihre Adressen für alle Achsen aufgelistet.

Einstellung der Pulsdauer am Pin INT1

Die gemessenen Werte werden kontinuierlich mit den Grenzwerten verglichen. Der Vergleich kann mit ungefilterten oder gefilterten Messwerten stattfinden und kann zum Auslösen eines Interrupt 1 sofort oder erst nach einer einstellbaren Verzögerung führen. Mit Bit 4, HPEN und Bit 3:2, INT1_SEL1:0 aus dem Steuerregister CTRL_REG5 (0x24) wird die Art der Filterung vor dem Vergleich wie folgt bestimmt:

- INT1_SEL1:0 = „00“ – keine Filterung;
- INT1_SEL1:0 = „01“ – Filterung mit dem Hochpass HPF;
- HPEN = „0“ INT1_SEL1 = „1“ – Filterung mit dem Tiefpass LPF2;
- HPEN = „1“ INT1_SEL1 = „1“ – Filterung mit einem Bandpass bestehend aus dem Tiefpass LPF2 und dem Hochpass HPF.

Über das Register INT1_DURATION (0x38) kann die Verzögerung mit den Bit 6:0 in Vielfachen der Abtastperiode eingestellt werden und es kann eingestellt werden, ob die Verzögerung nur beim Eintreten, oder auch beim Verschwinden des den Interrupt auslösenden Ereignisses stattfindet. Wenn das Bit 7 WAIT des Registers gesetzt ist, wird das Ein- und Ausschalten des Ausgangspulses verzögert. Ist dieses Bit zurückgesetzt, findet die Verzögerung nur beim Einschalten statt, wie in Abb. 6.14 dargestellt.

6.2.3.5 Temperaturmessung

Der Temperatursensor des Gyroskops misst die Umgebungstemperatur und speichert sie jede Sekunde mit einer Auflösung von 1 °C in das Register OUT_TEMP (0x26) im Zweierkomplementformat. Der gespeicherte Wert ist für absolute Temperaturmessungen eigentlich weder genau noch genug aufgelöst, da der Sensor für relative Messungen kalibriert ist und zur Temperaturkompensation dient. Der Temperaturunterschied zwischen zwei Messungen, die in den vorzeichenbehafteten Variablen cTemp1 und cTemp2

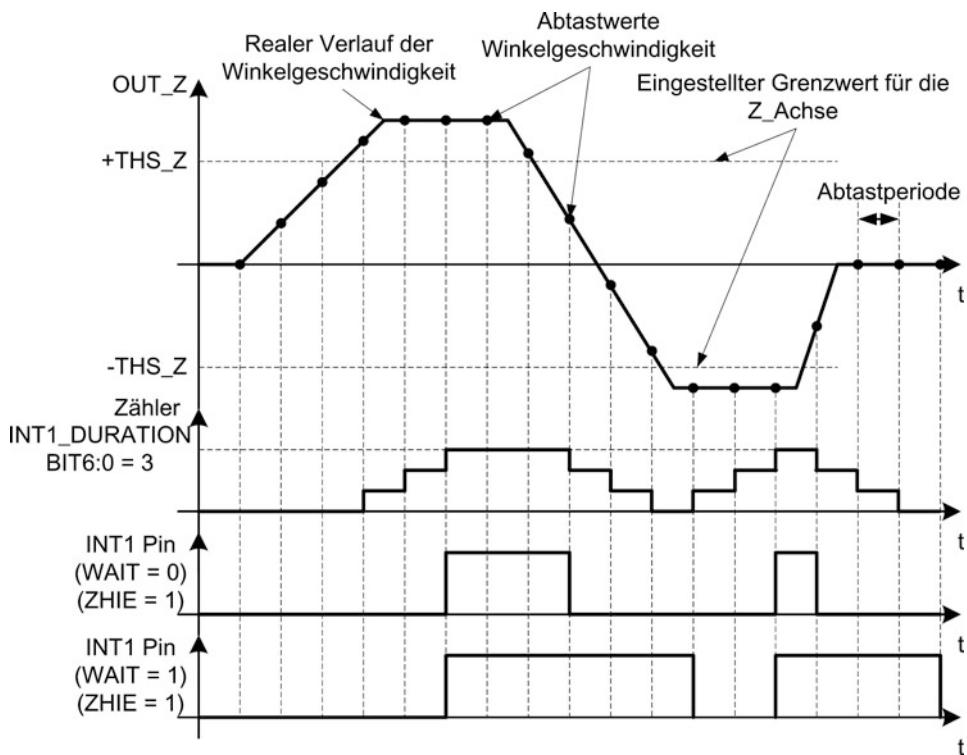


Abb. 6.14 L3GD20 Interrupt1 Beispiel

gespeichert sind, wird folgendermaßen berechnet:

$$c\Delta\text{Temp} = (-1) \cdot (c\text{Temp}_2 - c\text{Temp}_1)$$

Ein positives Ergebnis bedeutet Temperaturerhöhung.

6.3 MPL3115 digitaler Luftdrucksensor

MPL3115 [10, 11] und [12] ist ein digitaler Luftdrucksensor, der für die Messung des absoluten Luftdrucks im Bereich 200 hPa...1100 hPa eingesetzt werden kann. Der Sensor ist intern für Messungen oberhalb 500 hPa (mbar) kalibriert. Die 20-Bit A/D-Wandlung ermöglicht eine Auflösung von 0,01 hPa (1 Pa). Der gemessene Luftdruck kann intern in Höhe umgerechnet werden, was den Sensor in ein Altimeter umwandelt. Neben dem Luftdruck kann er auch die Temperatur im Bereich $-40^{\circ}\text{C}...+85^{\circ}\text{C}$ messen. Der Sensor ist als I²C Slave vorkonfiguriert und kann mit einem Master im Fast-Modus kommunizieren. Über I²C überträgt der Master die gewünschten Einstellungen und liest die gemessenen

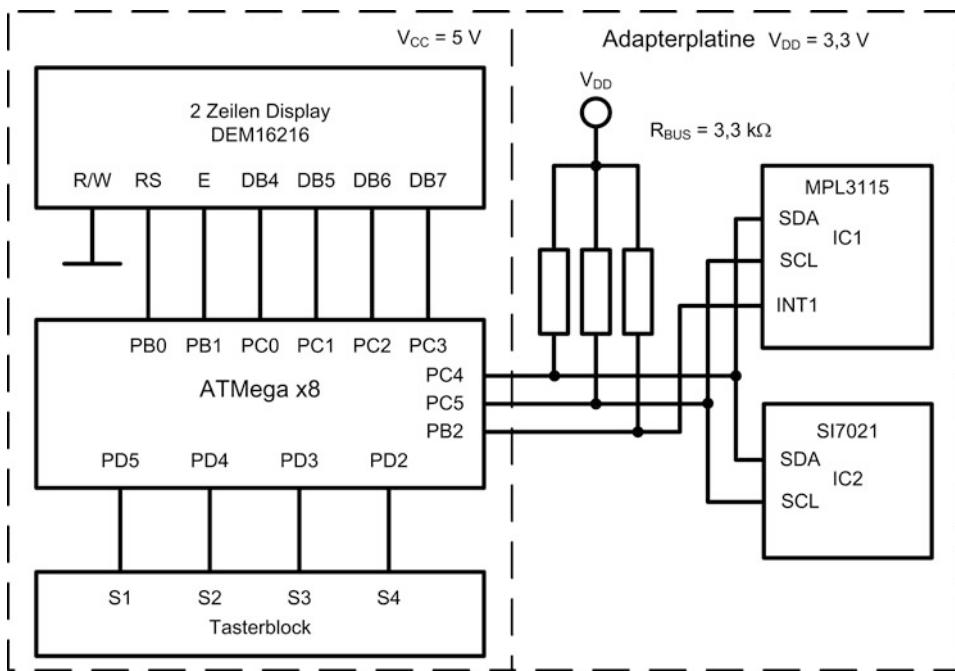


Abb. 6.15 Elektronische Wetterstation mit MPL3115 und SI7021

Werte aus. Über einstellbare Interrupts und ansteuerbare Ausgänge können dem Master eingetretene Ereignisse signalisiert werden.

In der Abb. 6.15 wird die Schaltung einer elektronischen Wetterstation dargestellt, die aus einem Luftdrucksensor MPL3115 und einem Luftfeuchtigkeitssensor SI7021 besteht. Der Feuchtigkeitssensor wird im Abschn. 6.4 beschrieben.

6.3.1 Aufbau des MPL3115

Ein schematisches Blockschaltbild des Bausteins zeigt Abb. 6.16.

6.3.1.1 Messfühler

Der Luftdruck wirkt auf den Luftdruckmessfühler durch ein kleines Loch im Gehäuse des Sensors. Dieser Messfühler ist in einer Brückenschaltung eingebaut, deren analoge Ausgangsspannung proportional zum absoluten Luftdruck ist. Der absolute Luftdruck p gemessen auf der Höhe H oberhalb des Meeresspiegels kann mit der barometrischen Höhenformel [13] berechnet werden:

$$p = p_0 \cdot e^{-\frac{H}{8000}}, \quad (6.13)$$

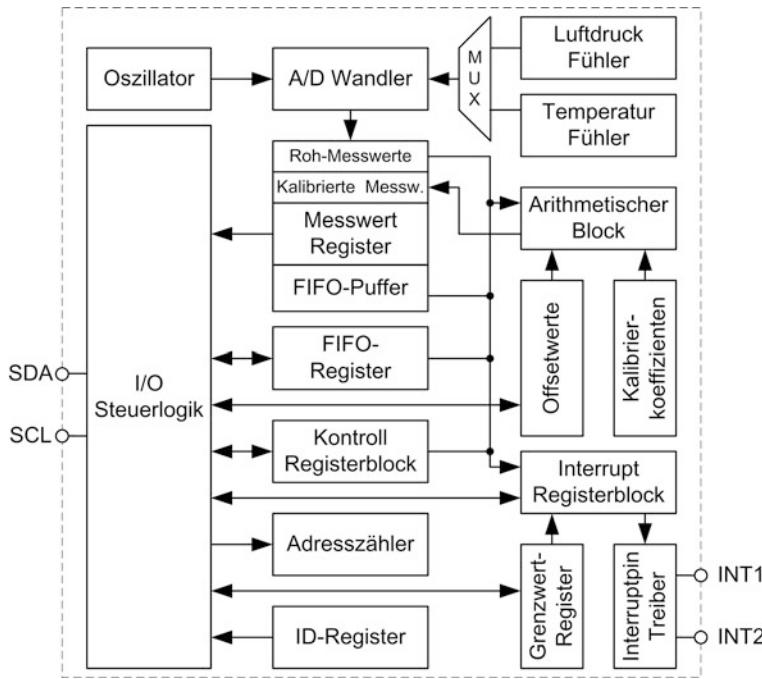


Abb. 6.16 MPL3115 Blockschaltbild

wobei p_0 der normale Luftdruck auf Meereshöhe ist. Mit dem gemessenen Wert des absoluten Luftdrucks p , wenn die Höhe H bekannt ist, kann der Luftdruck auf Meeresebene p_0 mit der Gl. 6.14 aus [10] berechnet werden:

$$p_0 = \frac{p}{\left(1 - \frac{\text{Höhe}}{44.330,77}\right)^{0,1902632}} \quad (6.14)$$

Der Koeffizient 44.330,77 berücksichtigt die Abhängigkeit des Luftdrucks von der globalen Mitteltemperatur ($T_m = 15^\circ\text{C} = 288,15\text{K}$) und dem vertikalen Temperaturgradient ($\gamma = -0,65\text{K}/100\text{m}$) (siehe [13]):

$$44.330 \approx \left| \frac{T_m}{\gamma} \right| \quad (6.15)$$

Der Sensor berechnet mit dem normalen Luftdruck auf Meereshöhe $p_0 = 1013,25\text{ hPa}$ die Höhe mit der Gl. 6.16 [10]. Um eine genauere Höhe berechnen zu können muss in die flüchtigen Register *BAR_IN_MSB_REG* (Adresse 0x14) und *BAR_IN_LSB_REG* (0x15) der aktuelle, relative Luftdruck mit einer Auflösung von 2 Pa/LSB gespeichert werden. Eine eventuelle Höhenkorrektur wird in das Register *OFF_H_REG* (0x2D) in Meter im

Zweierkomplement gespeichert. Diese Korrektur wird bei der Berechnung der Höhe immer berücksichtigt.

$$H = 44.330,77 \cdot \left(1 - \left(\frac{p}{p_0} \right)^{0,1902632} \right) \quad (6.16)$$

Der absolute Luftdruck ist temperaturabhängig. Um den Temperatureinfluss kompensieren zu können besitzt der Baustein auch einen Temperaturfühler. Die von den Messfühlern erzeugten analogen Spannungen werden verstärkt und mit einem Tiefpass gefiltert. Nach der Begrenzung der Frequenzspektren werden die Messsignale quantisiert und weiterhin digital verarbeitet.

6.3.1.2 Register

Die Einstellungen und die Messergebnisse des Bausteins MPL3115 werden in flüchtigen, 8-Bit großen Registern gespeichert. Diese Register belegen den Speicherbereich 0x00...0x2D und werden über einen Adresszähler einzeln adressiert. Der Adresszähler wird mit der gewünschten Adresse über I²C geladen. Mit dem Lesen bzw. Schreiben eines Registers wird der Adresszähler inkrementiert und somit ein weiteres Register adressiert. Ausnahmsweise wird der Adresszähler, wenn der FIFO-Modus deaktiviert ist, nach dem Zugriff auf das Register mit der Adresse 0x05 bzw. 0x0B mit dem Wert 0x00 bzw. 0x06 geladen. Diese Ausnahmen ermöglichen das kontinuierliche Lesen der Messwertregister und des Statusregisters. Nach dem Hochfahren werden die Inhalte der meisten Register auf „0“ gesetzt. Die Arbeitsmodi, die die Änderung der Registerinhalte erlauben, sind dem Datenblatt [10] zu entnehmen. Die Werte werden in die Register im Big-Endian-Format gespeichert.

Kontroll-Registerblock

Der Kontroll-Registerblock besteht aus fünf Registern, die den Adressbereich 0x26...0x2A belegen. Diese Register steuern den gesamten Messverlauf des Sensors. Das erste Register *CTRL_REG1* (0x26) speichert die Messeinstellungen und hat folgende Konfiguration:

- Bit 7** – ALT – mit dem Setzen dieses Bits wird der gemessene Luftdruck in Höhe umgerechnet und anstatt des Luftdruckwerts in das Messregister gespeichert;
- Bit 6** – RAW – wenn dieses Bit gesetzt ist, werden die unkompenzierten Messwerte vom A/D-Wandler direkt in die Messwertregister gespeichert; bei dieser Wahl müssen der FIFO-Speichermodus und die Interrupts deaktiviert werden;
- Bit 5:3** – OS[2:0] – Eine bekannte Methode zur Reduzierung des Messrauschpegels ist die Mittelwertbildung von mehreren Messergebnissen (Oversampling = Überabtastung). Über diese drei Bits wird die Überabtastrate (2^{OS}) und die minimale Messdauer t_{OS} bestimmt:

$$t_{OS} = 2^{OS} \cdot 4 + 2 \text{ [ms]} \quad (6.17)$$

- Bit 2** – RST – mit dem Setzen dieses Bits wird der gesamte Baustein zurückgesetzt und die Register werden mit den voreingestellten Werten geladen. Anschließend wird das Bit zurückgesetzt.
- Bit 1** – OST – Wenn dieses Bit im Standby-Modus gesetzt wird, wird eine Einzelmesung, welche die Bits ALT, RAW und OST berücksichtigt, gestartet. Die gemessenen Werte werden gespeichert und die interne Steuerung setzt anschließend das Bit zurück.
- Bit 0** – SBYB – Der Sensor befindet sich im Standbymodus, wenn dieses Bit „0“ ist bzw. im Aktivmodus, wenn das Bit „1“ ist.

Das Register *CTRL_REG2* dient der Alarmeinstellung und Bestimmung der Abtastperiode im FIFO-Modus. Die Abtastperiode wird mit den Bits 3:0 des Registers *CTRL_REG2* im Bereich [1... 2^{15}] s eingestellt. Das Register *CTRL_REG3* (0x28) konfiguriert die zwei Interrupt-Ausgänge als push-pull oder open-drain und legt die Polarität der Ausgänge fest. Über das Register *CTRL_REG4* (0x29) werden die Interrupts freigegeben und mit dem Register *CTRL_REG5* werden die freigegebenen Interrupts auf den INT1 oder INT2 Ausgang geschaltet. Wenn mehrere Interrupts auf den gleichen Ausgang geschaltet sind, werden sie ver-ODER-t.

Statusregister

Wenn der FIFO-Modus deaktiviert ist, melden die Bits 7, 6 und 5 des Statusregisters (0x00), dass ungelesene Messwerte (Temperatur- und/oder Luftdruckwerte) überschrieben wurden. Die Bits 2, 1 und 0 signalisieren das Speichern neuer Messwerte. Die Bits 0 und 5 betreffen die Temperaturwerte, die Bits 1 und 6 die Luftdruckwerte und die Bits 2 und 7 Beides. Das Register mit der Adresse 0x06 bildet den Inhalt des Statusregisters ab.

Messwertregister

Absolute Messwerte

Der gemessene Luftdruck bzw. die berechnete Höhe wird in die Register *OUT_P_MSB_REG* (0x01), *OUT_P_CSB_REG* (0x02) und *OUT_P_LSB_REG* (0x03) linksbündig gespeichert. Der korrigierte Luftdruckwert wird in Pascal (1 Pa = 0,01 mbar) vorzeichenlos auf 20 Bits gespeichert, 2 davon für die Nachkommastellen. Der gespeicherte Wert berücksichtigt einen eventuellen Offsetwert der im Zweierkomplement im Register *OFF_P_REG* (0x2B) mit einer Auflösung von 4 Pa/LSB gespeichert ist. Die Funktion *MPL3115_Read_DataReg* speichert das Statusregister und die Messwertregister in den Vektor *ucDataByte*. *ucDataByte[0]* speichert den Inhalt des Statusregisters, *ucDataByte[1]* den Inhalt des Registers *OUT_P_MSB_REG* usw. Der folgende Code berechnet den absoluten Luftdruck in Pa und speichert ihn in die Variable *lPressureData*.

```
lPressureData = ucDataByte[1];
lPressureData = (lPressureData << 8) | ucDataByte[2];
```

```
lPressureData = (lPressureData << 8) | ucDataByte[3];
lPressureData = lPressureData >> 6;
```

Die korrigierte Höhe wird in Meter im Zweierkomplement auf 20 Bits gespeichert, 4 davon für die Nachkommastellen. Durch das Speichern des aktuellen relativen Luftdrucks in die Register *BAR_IN_MSB_REG* (0x14) und *BAR_IN_LSB_REG* (0x15) mit einer Auflösung von 2 Pa/LSB wird die lokale Höhe genauer berechnet. Eine eventuelle Korrektur kann in das Register *OFF_H_REG* (0x2D) in Meter, im Zweierkomplement gespeichert werden.

Die gemessene Temperatur wird in die Register *OUT_T_MSB_REG* (0x04) und *OUT_T_LSB_REG* (0x05) in °C linksbündig gespeichert. Der Temperaturwert belegt 12 Bits, 4 davon für die Nachkommastellen. Die Temperaturkorrektur kann in das Register *OFF_T_REG* (0x2C) im Zweierkomplement mit einer Auflösung von 0,0625 °C/LSB gespeichert werden. Folgender Programmcode berechnet aus den mit der Funktion *MPL3115_Read_DataReg* gelesenen Werten die positive Temperatur mit einer Auflösung von 0,1 °C und speichert sie in die Variable *uiTemperatureData*.

```
uiTemperatureData = ucDataByte[4];
uiTemperatureData = ((uiTemperatureData << 8) | ucDataByte[5]) >> 4;
uiTemperatureData = (uiTemperatureData * 10) >> 8;
```

Relative Messwerte

Nach jeder Messung rechnet die arithmetische Einheit des Sensors die Differenzen zwischen den aktuellen und vorigen Messwerten und speichert sie, ähnlich wie die absoluten Messwerte, im Zweierkomplement. Die Luftdruckdifferenz/Höhdifferenz wird in die Register *OUT_P_DELTA_MSB* (0x07), *OUT_P_DELTA_CS* (0x08) und *OUT_P_DELTA LSB* (0x09), während die Temperaturdifferenz wird in die Register *OUT_T_DELTA_MSB* (0x0A) und *OUT_T_DELTA_LSB* (0x0B) gespeichert.

Extremwerte

Der Sensor speichert die Minima und die Maxima der gemessenen Luftdruck- und Temperaturwerte im gleichen Format wie die absoluten Messwerte. Die Speicherregister sind in der Tab. 6.13 dargestellt. Sie können jederzeit auf „0“ gesetzt werden um nach einem neuen Minimum bzw. Maximum zu suchen.

FIFO-Register

Der Sensor stellt für das Zwischenspeichern der Messergebnisse einen FIFO-Puffer für 32 Messdatensätze mit je 5 Bytes zur Verfügung, was das asynchrone Lesen der Messwerte erleichtert. Die Byte-Konfiguration eines Messdatensatzes ist gleich mit der der absoluten Messwerte. Über die Bits 7:6 des Registers *F_SETUP_REG* (0x0D) wird bei „00“ der FIFO-Modus deaktiviert, bei „01“ bzw. „10“ wird der FIFO-Modus aktiviert. Wenn die zwei Bits die Kombination „01“ aufweisen, wird der älteste Messdatensatz überschrieben und bei „10“ wird das Speichern der Messwerte gestoppt sobald der Puffer voll ist.

Tab. 6.13 Speicherregister für die Minima und Maxima der gemessenen Messwerte

	Minima		Maxima	
	Register	Adresse	Register	Adresse
Luftdruck	P_MIN_MSB_REG	0x1C	P_MAX_MSB_REG	0x21
	P_MIN_CSB_REG	0x1D	P_MAX_CSB_REG	0x22
	P_MIN_LSB_REG	0x1E	P_MAX_LSB_REG	0x23
Temperatur	T_MIN_MSB_REG	0x1F	T_MAX_MSB_REG	0x24
	T_MIN_LSB_REG	0x20	T_MAX_LSB_REG	0x25

Die Kombination „11“ ist nicht implementiert. Wenn die Anzahl der ungelesenen Datensätze die Füllgrenze erreicht, die mit den Bits 5:0 des Registers *F_SETUP_REG* festgelegt wurde, wird das Bit 6 im Register *F_STATUS_REG* (0x0D) gesetzt und kann zum Auslösen eines Interrupts führen. Ein weiterer Interrupt kann im FIFO-Modus dann ausgelöst werden, wenn der Puffer voll ist und dadurch das Bit 7 des Registers *F_STATUS_REG* gesetzt wurde. Die Bits 5:0 des letzten Registers speichern die Anzahl der ungelesenen Messsätze aus dem Puffer. Die gespeicherten Datensätze werden im FIFO-Modus aus dem Register *F_DATA_REG* (0x0E) gelesen. Alternativ kann für das Lesen das Register *OUT_P_MSB_REG* verwendet werden. Die weiteren Register, die die absoluten Messwerte speichern, liefern beim Lesen im FIFO-Modus den Wert 0x00. Für das Lesen der Messergebnisse kann die Funktion `MPL3115_Read_DataReg` verwendet werden, die im Folgenden aufgelistet ist. Beim Aufrufen der Funktion werden als Parameter die Adresse des ersten Registers aus dem Registerbereich (`ucfirst_reg_address`), die Adresse der Variable, die die Messwerte speichern soll (`uctarget_address`) und die Anzahl der zu lesenden Register (`ucnumber_of_reg`) übergeben.

```
uint8_t MPL3115_Read_DataReg(uint8_t ucfirst_reg_address,
                               uint8_t* uctarget_address, uint8_t ucnumber_of_reg)
{
    uint8_t ucDeviceAddress, ucI;
    //Adresse des Luftdrucksensors bilden
    ucDeviceAddress = MPL3115_DEVICE_TYP_ADDRESS << 1;
    ucDeviceAddress |= TWI_WRITE; //Write-Modus
    TWI_Master_Start(); //Start
    if((TWI_STATUS_REGISTER) != TWI_START) return TWI_ERROR;
    TWI_Master_Transmit(ucDeviceAddress); //Device Adresse senden
    if((TWI_STATUS_REGISTER) != TWI_MT_SLA_ACK) return TWI_ERROR;
    //die Anfangsadresse des Registerbereiches wird gesendet
    TWI_Master_Transmit(ucfirst_reg_address);
    if((TWI_STATUS_REGISTER) != TWI_MT_DATA_ACK) return TWI_ERROR;
    TWI_Master_Start(); //Restart
    if((TWI_STATUS_REGISTER) != TWI_RESTART) return TWI_ERROR;
    ucDeviceAddress = (MPL3115_DEVICE_TYP_ADDRESS << 1) | TWI_READ;
    //Read-Modus
```

```

TWI_Master_Transmit(ucDeviceAddress); //Device Adresse im
                                     //Read-Modus senden
if((TWI_STATUS_REGISTER) != TWI_MR_SLA_ACK) return TWI_ERROR;
//Inhalt der adressierten Register wird eingelesen
for(ucI = 0; ucI < (ucnumber_of_reg - 1); ucI++)
{
    uctarget_address[ucI] = TWI_Master_Read_Ack();
    if((TWI_STATUS_REGISTER) != TWI_MR_DATA_ACK) return TWI_ERROR;
}
uctarget_address[ucnumber_of_reg - 1] = TWI_Master_Read_NAck();
if((TWI_STATUS_REGISTER) != TWI_MR_DATA_NACK) return TWI_ERROR;
TWI_Master_Stop(); //Stop
return TWI_OK;
}

```

Interrupt-Register

Die Ereignisse, die einen Interrupt auslösen können, werden durch Setzen des entsprechenden Bits ins Register *INT_SOURCE_REG* (0x12) definiert:

Bit 7 – signalisiert zusammen mit den Einstellungen aus dem Register *PT_DATA_CFG_REG* einen neuen Messwert;

Bit 6 – der FIFO-Puffer löst beim Überlauf einen Interrupt aus oder wenn die Anzahl der ungelesenen Datensätze die eingestellte Füllgrenze erreicht hat;

Bit 5 – der gemessene Luftdruck/die berechnete Höhe befindet sich im Bereich $p_g \pm \Delta p$; p_g wird in die Register *P_TGT_MSB_REG* (0x16) und *P_TGT_LSB_REG* (0x17) und Δp ($\neq 0$) in die Register *P_WIND_MSB_REG* (0x19) und *P_WIND_LSB_REG* (0x1A) gespeichert;

Bit 4 – die gemessene Temperatur befindet sich im Bereich $t_g \pm \Delta t$; t_g wird in die Register *T_TGT_REG* (0x18) und Δt ($\neq 0$) in die Register *T_WIND_REG* (0x1B) gespeichert;

Bit 3 – der gemessene Luftdruck/die berechnete Höhe überquert einen der Grenzwerte: Δp , $p_g - \Delta p$ oder $p_g + \Delta p$;

Bit 2 – die gemessene Temperatur überschreitet einen der Grenzwerte: Δt , $t_g - \Delta t$ oder $t_g + \Delta t$;

Bit 1 – die Luftdruckänderung löst einen Interrupt aus;

Bit 0 – die Temperaturänderung löst einen Interrupt aus.

Die gewünschten Interrupts müssen mit der Einstellung des Registers *CTRL_REG4* freigegeben und mit *CTRL_REG5* auf einen der Interrupt-Pins geschaltet werden.

Mit der folgenden Funktion *MPL3115_Write_DataReg* können die Inhalte der Register, die unmittelbar aufeinanderfolgen, wie die Interrupt-Register oder die Register, die die Grenzwerte speichern, geändert werden. Die neuen Inhalte müssen in ein Array gespeichert werden, dessen Adresse zusammen mit der Adresse des ersten Registers aus

dem Bereich und die Anzahl der Einstellwerte als Parameter beim Aufruf der Funktion übergeben werden. Es können bis zu zehn Einstellwerte übertragen werden.

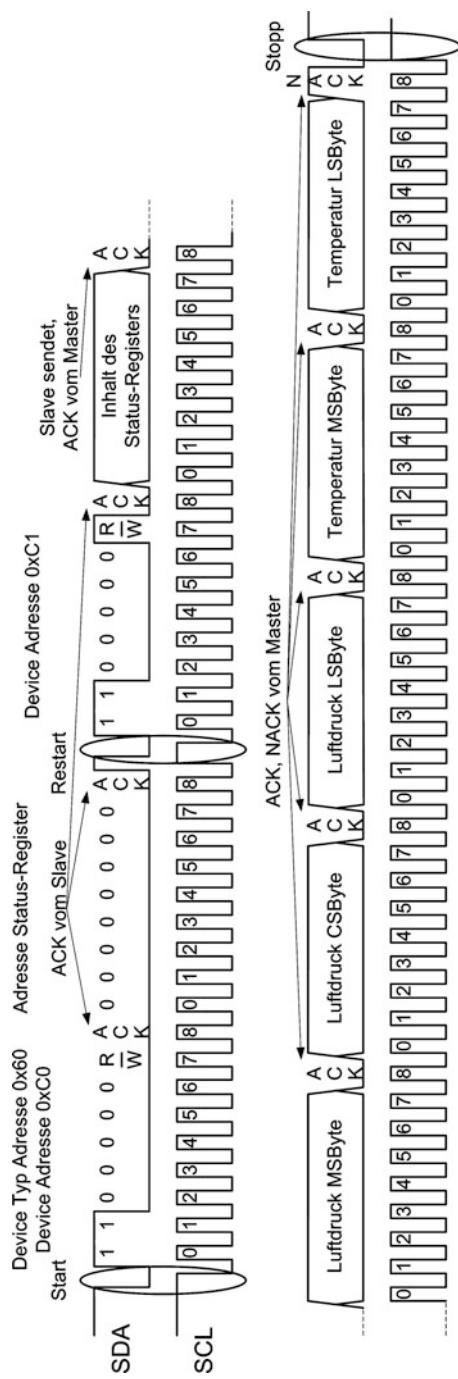
```
uint8_t MPL3115_Write_DataReg(uint8_t ucfirst_reg_address,
                               uint8_t* ucsource_address, uint8_t ucnumber_of_reg)
{
    uint8_t ucDeviceAddress, ucI, ucArray[12];
    //Adresse des Luftdrucksensors bilden
    ucDeviceAddress = MPL3115_DEVICE_TYP_ADDRESS << 1;
    ucDeviceAddress |= TWI_WRITE;      //Write-Modus
    ucArray[0] = ucDeviceAddress;
    ucArray[1] = ucfirst_reg_address;
    for(ucI = 0; ucI < ucnumber_of_reg; ucI++) ucArray[ucI + 2] =
    ucsource_address[ucI];
    TWI_Master_Start(); //Start
    if((TWI_STATUS_REGISTER) != TWI_START) return TWI_ERROR;
    for(ucI = 0; ucI < (ucnumber_of_reg + 2); ucI++)
    {
        TWI_Master_Transmit(ucArray[ucI]);
        if((TWI_STATUS_REGISTER) != TWI_MT_DATA_ACK) return TWI_ERROR;
    }
    TWI_Master_Stop(); //Stop
    return TWI_OK;
}
```

Device-Identifikation-Register

Das Device-Identifikation-Register *WHO_I_AM_REG* (0x0C) speichert den Wert 0x0D und kann in einem Bussystem zur Identifikation des Bausteins und zum Testen des Busses dienen.

6.3.2 Serielle Kommunikation

Der Sensor MPL3115 implementiert das I²C-Protokoll im Fast-Modus und ermöglicht die Kommunikation mit dem Master bei einer maximalen Taktfrequenz von 400 kHz. Er antwortet auf die 7-Bit Device-Typ-Adresse 0x60, nicht auch auf die 0x00. Die Abb. 6.17 stellt den zeitlichen Verlauf, der beim Auslesen des Statusregisters und der Messwert-Register mit der Funktion *MPL3115_Read_DataReg* stattfindet, dar. Bei einer Taktfrequenz von 370 kHz dauert der gesamte Verlauf 258 µs.

**Abb. 6.17** MPL3115 serielle Kommunikation

6.3.3 Power-Modi

Der Baustein befindet sich nach dem Einschalten, mit den meisten Registern auf 0x00 gesetzt, im Standby-Modus. In diesem Arbeitsmodus können Einstellungen geändert werden, der analoge Teil des Sensors ist abgeschaltet um Energie zu sparen und der volle Zugriff auf die Register ist gewährleistet. Mit dem Setzen des Bits 0 des Kontroll-Registers 1 schaltet der Sensor in den Aktiv-Modus um. In diesem Modus werden die Messungen kontinuierlich durchgeführt, die Inhalte der Register können aber nur zum Teil geändert werden.

6.3.4 Mess- und Lesemodi

Die Komplexität des Bausteins ermöglicht unterschiedliche Mess- und Lesestrategien. Der Sensor kann den Luftdruck und die Temperatur im Einzelmess-, im Freilauf- oder im FIFO-Modus messen. Im Standby-Modus können Messungen im Einzelmessmodus durchgeführt werden. Durch das Setzen des Bits 1 des Kontroll-Registers 1 wird eine Messung gestartet. Für die Dauer der Messung befindet sich der Sensor im Aktiv-Modus. Mit dem Speichern der Messergebnisse in die Messregister schaltet der Sensor in den Standby-Modus um und das Bit 1 des Kontroll-Registers wird zurückgesetzt. Im Aktiv-Modus werden die Messungen kontinuierlich im Freilauf- oder FIFO-Modus durchgeführt.

Im Einzelmessmodus bestimmt der Master den Zeitpunkt der Abtastung. Das Lesen der Messwerte findet synchron zur Messung statt wenn die Messdauer (siehe Gl. 6.17) berücksichtigt wird. In diesem Modus können bis zu 100 Messungen/s realisiert werden.

Ist der FIFO-Puffer deaktiviert (im Register *F_SETUP_REG* sind die Bits 7:6 = „00“), finden die Messungen im Aktiv-Modus kontinuierlich statt. Das Auslesen der Messwerte kann zeitgesteuert, synchron oder messwertabhängig stattfinden. Beim zeitgesteuerten Auslesen legt der Master den Zeitpunkt des Lesens fest. Dadurch kann allerdings nicht gewährleistet werden, dass alle Messwerte genau einmal gelesen werden. Mit dem Konfigurieren des Registers *PT_DATA_CFG_REG* werden neue Messwerte durch passende gesetzte Bits (für Temperatur und/oder Luftdruck) im Statusregister signalisiert. Somit wird die Möglichkeit geschaffen, alle Messwerte synchron zu lesen. Der Zustand dieser Bits kann im polling-Betrieb getestet werden was aber zum blockierenden Warten des Mikrocontrollers führt. Wenn zusätzlich die Interrupt-Register konfiguriert werden, wird die Messung eines neuen Wertes dem Master über einen externen Interrupt signalisiert. Dafür muss der konfigurierte Interrupt-Ausgang des Sensors mit einem Interrupt-fähigen Eingang des Masters verbunden werden. Bei Bedarf können die Messwerte auch nur dann gelesen werden, wenn sie einen Grenzwert erreicht oder überschritten haben.

Wenn der FIFO-Modus aktiviert ist (die Bits 7:6 im Register *F_SETUP_REG* sind „01“ oder „10“) und der Baustein sich im Aktiv-Modus befindet, werden die Messungen intern zeitgesteuert gestartet und die Messwerte in den FIFO-Puffer gespeichert. Die Zeitperiode zwischen zwei Messungen wird mit den Bits 3:0 des Kontroll-Registers 2 festgelegt. Mit

der passenden Einstellung der Interrupt-Register wird ein Interrupt ausgelöst, wenn der Puffer voll ist oder die eingestellte Anzahl der Messdatensätze erreicht hat. Durch den Interrupt wird der Zeitpunkt des Lesens festgelegt und trotz des asynchronen Lesens gehen in diesem Fall keine Messwerte verloren. Der Kommunikationsumfang wird dadurch stark reduziert, was zur Entlastung des Masters und zum Energiesparen führt.

6.3.5 Initialisierung des MPL3115-Sensors

Die einstellbaren Register des MPL3115 sind flüchtig und müssen deshalb nach jedem Einschalten, jeder Initialisierung oder Reset neu eingestellt werden. Der Master muss die Kontroll- und Interrupt-Register konfigurieren, um den Mess- und Interrupt-Modus einzustellen und die Offsetwerte, die Grenzwerte und den aktuellen relativen Luftdruck als Referenz für die Messung der Höhe laden.

6.4 Luftfeuchtigkeit SI7021

Die Luftfeuchtigkeit oder Luftfeuchte wird definiert als der Wasserdampfgehalt der Luft. Man unterscheidet zwischen der absoluten und relativen Luftfeuchtigkeit. Die absolute Luftfeuchte gibt die Wasserdampfmasse in einem Luftvolumen an, wird in g/m³ oder kg/m³ gemessen und ist von der Temperatur unabhängig. Als Sättigungszustand wird der Zustand bezeichnet, in dem bei einer bestimmten Lufttemperatur und einem bestimmten Luftdruck der maximale Wasserdampfgehalt erreicht wird. In diesem Zustand wird eine relative Luftfeuchtigkeit von 100 % erreicht. Die Aufnahmefähigkeit der Luft für Wasserdampf nimmt mit steigender Temperatur zu und mit steigendem Luftdruck ab. Die relative Luftfeuchtigkeit RH¹⁰ ist das Verhältnis zwischen der aktuellen Wasserdampfmasse und derjenigen im Sättigungszustand (bei gleicher Lufttemperatur und gleichem Luftdruck) und wird in % gemessen. Die Taupunkttemperatur oder der Taupunkt ist die Temperatur bei der die relative Luftfeuchtigkeit 100 % erreicht. Der Wasserdampf kondensiert, wenn die Temperatur unter den Taupunkt sinkt.

Die Messung der relativen Luftfeuchtigkeit spielt eine große Rolle in Meteorologie, Transport und Aufbewahrung von Ware sowie für die menschliche Gesundheit und Behaglichkeit. Die Luftfeuchte kann die Genauigkeit und Zuverlässigkeit von anderen Messsensoren beeinträchtigen, deshalb wird sie als Parameter vieler Messungen vorgegeben. Der Einfluss auf z. B. Schallgeschwindigkeit oder interferometrische Messungen muss sogar kompensiert werden.

Es gibt Materialien, deren Leitfähigkeit oder Dielektrizitätskonstante mit der aus der Luft absorbierten Feuchtigkeit variieren. Diese Eigenschaften werden verwendet um elektronische Sensoren zu bauen, deren Messverfahren für die Messung der relativen Luft-

¹⁰ RH – relative humidity.

feuchtigkeit auf der Messung des elektrischen Widerstandes oder der elektrischen Kapazität beruhen.

6.4.1 Aufbau des SI7021

SI7021 [14, 15] ist ein elektronischer Sensor für die Messung der relativen Luftfeuchtigkeit, dessen Fühler ein Kondensator ist. Durch ein Loch im Gehäuse des Bausteins findet der Feuchtigkeitsaustausch zwischen dem Dielektrikum des Kondensators und der Luft statt. Um die Temperaturabhängigkeit der relativen Luftfeuchtigkeit zu kompensieren, besitzt der SI7021 auch einen Temperaturfühler wie in Abb. 6.18 dargestellt.

Der Sensor muss nicht initialisiert werden, vor der ersten Messung nach dem Hochfahren müssen ca. 15 ms gewartet werden. Über einen D/A-Wandler wird die analoge Ausgangsspannung eines Fühlers umgewandelt, der digitale Wert wird mittels der Kalibrierkoeffizienten in einem arithmetischen Block umgerechnet und das Ergebnis in das Messwert-Register gespeichert. Dieses ist ein 16-Bit Register, das über die serielle Schnittstelle nur gelesen werden kann. Durch die Umrechnung werden die Kennlinien beider Fühler linearisiert und die Abhängigkeit der Luftfeuchtigkeitswerte mit der Temperatur kompensiert. Zusätzlich berechnet der arithmetische Block die Prüfsumme der gemessenen Werte, die bei Bedarf mitübertragen werden können. Mehrere Prüfsummen werden bei der Übertragung der Identifikationsnummer mitübertragen. Die Kalibrierkoeffizienten werden für jeden Sensor werksseitig ermittelt und gespeichert um den Austausch der einzelnen Sensoren in einer Schaltung ohne Software-Änderungen zu ermöglichen.

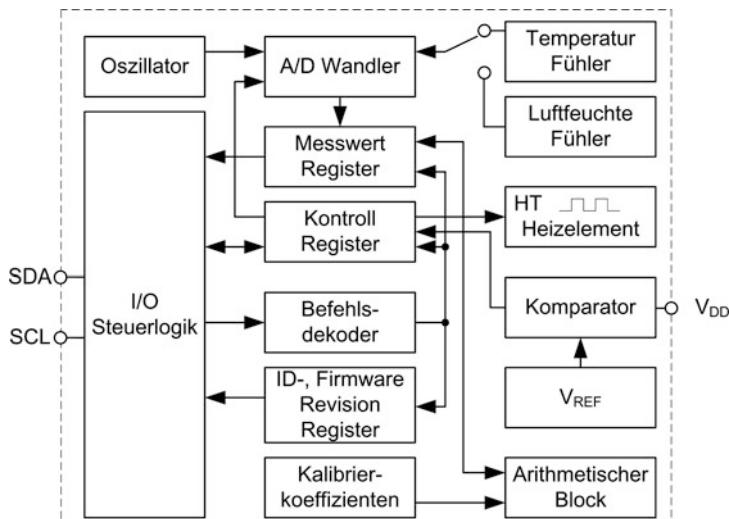


Abb. 6.18 SI7021 Blockschaltbild

Tab. 6.14 SI7021 Bitauflösung und Messdauer der Temperatur und Luftfeuchtigkeit

RES1	RES0	Temperatur		Luftfeuchtigkeit	
		Bitauflösung	Messdauer	Bitauflösung	Messdauer
0	0	14 Bit	10,8 ms	12 Bit	12 ms
0	1	12 Bit	3,8 ms	8 Bit	3,1 ms
1	0	13 Bit	6,2 ms	10 Bit	4,5 ms
1	1	11 Bit	2,4 ms	11 Bit	7 ms

Das 8-Bit Schreib-Lese-Kontroll-Register hat folgende Konfiguration:

- Bit 7, Bit 0**
 - RES1:0 – über diese zwei Bits wird die Bitauflösung des D/A-Wandlers wie in der Tab. 6.14 dargestellt, eingestellt.
- Bit 6**
 - VDDS – ist ein Bit das nur gelesen werden kann und das von der Hardware gesetzt wird wenn die Versorgungsspannung unter der Grenze von 1,9 V liegt. Unter 1,8 V wird die Funktionalität des Sensors nicht mehr gewährleistet.
- Bit 5, Bit 4, Bit 3, Bit 1**
 - sind reserviert.
- Bit 2**
 - HTRE – das interne Heizelement HT des Sensors wird bei „1“ eingeschaltet, sonst aus.

Wenn der Sensor bei Temperaturen unter dem Taupunkt eingesetzt wird, besteht die Gefahr, dass Wasserdampf auf dem Sensor oder Dielektrikum des Fühlers kondensiert, was zu falschen Ergebnissen führt. Ein Offset der Luftfeuchtigkeitswerte kann auftreten, wenn der Sensor lange Zeit hoher Luftfeuchte ausgesetzt wurde. Um diese möglichen Verfälschungen zu vermeiden, kann das interne Heizelement wiederholt ein- und ausgeschaltet werden.

Der Befehlsdekomponierer dekodiert die 1- oder 2-Byte Befehle, die über I²C empfangen wurden, und steuert entsprechend die internen Baugruppen des Sensors. Diese Befehle werden in einer Botschaft gleich nach der Adressierung im Schreibmodus des Slaves platziert, so wie in Abb. 6.19 dargestellt.

Der Sensor speichert in 8 Bytes eine elektronische Identifikationsnummer, die diesen in einem komplexen Netzwerk eindeutig identifiziert und in einem Byte die Firmwareversion, die ihn von der Vorgänger- und Nachfolgerversionen unterscheidet. Diese Informationen kann ein Master über I²C ablesen.

6.4.2 Serielle Kommunikation

Der Sensor ist als I²C-Slave konfiguriert und unterstützt die Kommunikation mit bis zu 400 kBit/s. Seine initiale Device-Typ-Adresse lautet 0x80 (mit dem R/W Bit zurückgesetzt). Er kann mit Spannungen zwischen 1,9 und 3,6 V versorgt werden. Die SDA- und

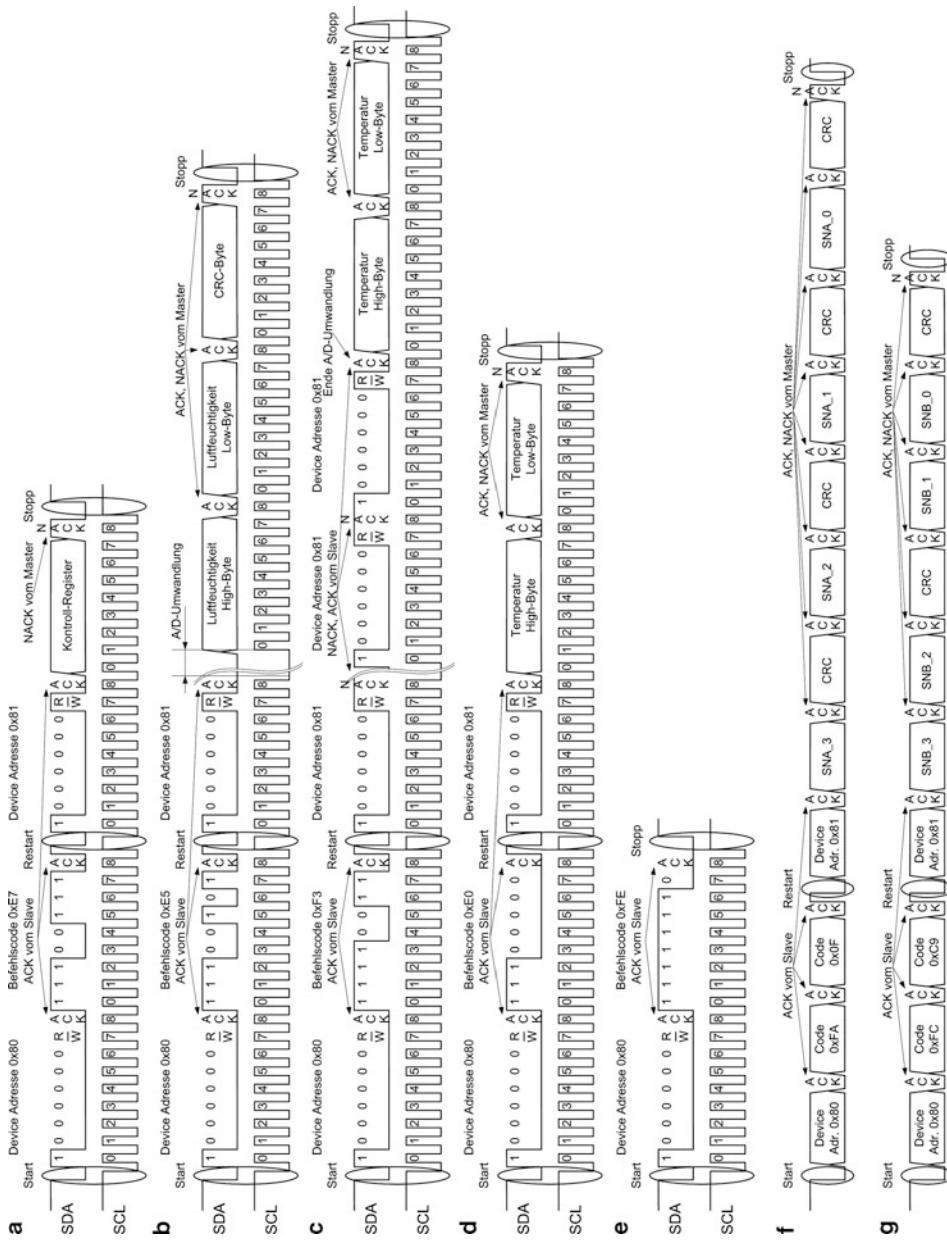


Abb. 6.19 ST7021 I²C-Kommunikation

SCL-Eingänge sind nicht 5 V-tolerant, was bedeutet, dass der Master entweder mit der gleichen Spannung wie der Sensor versorgt werden muss oder den niedrigen Spannungspegel als logische „1“ akzeptieren muss. Außerdem muss zum Schutz gegen elektrische Zerstörung ein Spannungsteiler oder ein Level-Shifter eingebaut werden. Der Slave antwortet auf 1- oder 2-Byte Befehle. Die 16-Bit-Werte werden mit dem höherwertigen Byte zuerst übertragen.

Um Energie sparen zu können funktioniert der Baustein nur im Einzelmess-Modus. Jede weitere Messung einer Größe muss extra gestartet werden.

6.4.2.1 Zugriff auf das Kontroll-Register

Vor der ersten Messung muss über das Kontroll-Register die Messauflösung eingestellt werden. Über die Bits RES1:0 können jeweils vier Kombinationen von Bitauflösungen für die Messung der Temperatur und der relativen Luftfeuchtigkeit gewählt werden (siehe Tab. 6.14). Bei der Wahl der Bitauflösung ist die Umwandlungszeit und die Genauigkeit der Messungen (max. 3 % für die relative Luftfeuchtigkeit zwischen 0 und 80 % und max. 0,4 °C für die Temperatur) zu berücksichtigen. Wenn man die zwei Größen mit Auflösungen messen möchte, deren Kombinationen in der Tabelle nicht vorhanden sind, wie z. B. 12 Bit für beide Messungen, muss die Bitauflösung vor jeder Messung neu eingestellt werden.

In Abb. 6.19a ist das Lesen des Kontroll-Registers dargestellt. Der Master adressiert im Schreib-Modus den Sensor und sendet danach den Befehlscode *0xE7*. Nach einer Restart-Sequenz und Adressierung im Lese-Modus (R/W-Bit = „1“) sendet der Slave den Inhalt des Registers. Der Master quittiert den Empfang des Bytes mit NACK und beendet die Kommunikation mit einer Stopp-Sequenz. Mit dem Lesen des Kontroll-Registers kann z. B. die Versorgungsspannung des Bausteins überwacht werden.

Um in das Kontroll-Register zu schreiben, adressiert der Master den Slave im Schreib-Modus und sendet nach dem Befehlscode *0xE6* ein weiteres Byte mit dem neuen Inhalt des Registers. Der Slave muss den Empfang eines jeden Bytes mit ACK bestätigen. Der Master beendet die Kommunikation mit einer Stopp-Sequenz.

6.4.2.2 Messung der relativen Luftfeuchtigkeit

Eine neue Messung der relativen Luftfeuchtigkeit wird mit dem Code *0xE5* im Hold-Modus (siehe Abb. 6.19b oder mit *0xF5* im No-Hold-Modus gestartet. Nachdem der Slave einen der Codes empfangen und mit ACK bestätigt hat, beginnt die A/D-Umwandlung, deren Dauer von der Bitauflösung abhängig und in Tab. 6.14 zu finden ist. Nach RESTART und erneuter Adressierung streckt der Slave im Hold-Modus das SCL-Signal während der Umwandlung und nach der Freigabe des SCL-Signals sendet er den 16-Bit-Messwert an den Master. Wenn der Master den Empfang des niederwertigen Bytes des Messwertes mit ACK bestätigt, sendet der Slave noch ein CRC¹¹-Byte, das vom Master mit NACK quittiert wird. Unabhängig von der eingestellten Bitauflösung für die Messung ist das

¹¹ CRC –cyclic redundancy check (zyklische Redundanzprüfung).

Bit15 des Messwertes das höchstwertige Bit und die Bit1:0 haben den binären Wert „10“. Der gemessene Wert der relativen Luftfeuchtigkeit, der vom Master eingelesen wird, ist von dem Sensor durch interne Berechnungen temperaturkompensiert worden und kann gemäß der im Abschn. 6.4.3 vorgestellten Formel in % umgerechnet werden.

Die Clock-Streckung ist ein Verfahren, das vom I²C Protokoll als Option vorgesehen ist. Diese Option ermöglicht einem Slave auf Bit-Ebene die Taktfrequenz zu verlangsamen oder auf Byte-Ebene nach dem Empfang eines Bytes das Clock Signal auf Low zu halten solange er intern beschäftigt ist. Dadurch wird die Kommunikation angehalten aber nicht unterbrochen. Die Mikrocontroller der ATmega Familie haben diese Option für die TWI Schnittstelle implementiert.

Die folgende Funktion `SI7021_Read_ValueHoldMode` ermöglicht den Start einer neuen Messung im Hold-Modus und implementiert den zeitlichen Verlauf aus der Abb. 6.19b. Der Befehlscode für eine Temperatur- oder Luftfeuchtigkeitsmessung wird als Parameter übergeben. Weil die Clock-Streckung auf der Data-Link-Schicht des Protokolls implementiert ist, muss sie nicht in der Software berücksichtigt werden. Der Messwert wird in der Variable `ucTempOrHumidValue[2]` gespeichert und das CRC-Byte in der Variable `ucChecksum`. Auf diese Variablen die in dem Modul *SI7021* global deklariert sind, kann man aus dem Hauptprogramm über Schnittstellen Funktionen zugreifen. Die Funktion gibt den Wert `TWI_OK` zurück, wenn die Kommunikation fehlerfrei verlaufen ist, ansonsten `TWI_ERROR`.

```
unsigned char SI7021_Read_ValueHoldMode(uint8_t ucmeasure_cmd)
{
    unsigned char ucAddress;

    ucAddress = SI7021_DEVICE_TYPE_ADDRESS | TWI_WRITE; //Write-Modus
    TWI_Master_Start(); //Start
    TWI_Master_Transmit(ucAddress); //Device Adresse senden
    if((TWI_STATUS_REGISTER) != TWI_MT_SLA_ACK) return TWI_ERROR;
    //Messungsart senden
    TWI_Master_Transmit(ucmeasure_cmd);
    if((TWI_STATUS_REGISTER) != TWI_MT_DATA_ACK) return TWI_ERROR;
    TWI_Master_Start(); //Restart
    if((TWI_STATUS_REGISTER) != TWI_RESTART) return TWI_ERROR;
    ucAddress = SI7021_DEVICE_TYPE_ADDRESS | TWI_READ; //Read-Modus
    TWI_Master_Transmit(ucAddress); //Device Adresse senden
    if((TWI_STATUS_REGISTER) != TWI_MR_SLA_ACK) return TWI_ERROR;
    //Most significant Byte lesen
    ucTempOrHumidValue[0] = TWI_Master_Read_Ack();
    if((TWI_STATUS_REGISTER) != TWI_MR_DATA_ACK) return TWI_ERROR;
    //least significant Byte lesen
    ucTempOrHumidValue[1] = TWI_Master_Read_Ack();
    if((TWI_STATUS_REGISTER) != TWI_MR_DATA_ACK) return TWI_ERROR;
    //Prüfsumme lesen
```

```

    ucChecksum = TWI_Master_Read_NAck();
    if((TWI_STATUS_REGISTER) != TWI_MR_DATA_NACK) return TWI_ERROR;
    TWI_Master_Stop(); //Stop
    return TWI_OK;
}

```

Im No-Hold-Modus beantwortet der Slave die Adressierung im Lese-Modus nach der RESTART-Sequenz mit NACK, solange die A/D-Umwandlung läuft. Am Ende der Umwandlung bestätigt er die Adresse mit ACK und sendet den Messwert und bei Bedarf das CRC-Byte.

6.4.2.3 Messung der Temperatur

Eine neue Temperaturmessung im Hold-Modus wird mit dem Code *0xE3* und im No-Hold-Modus mit *0xF3* (Abb. 6.19c) gestartet. Die Kommunikation verläuft ähnlich wie in Abschn. 6.4.2.2 beschrieben. Bei Bedarf kann auch die Prüfsumme des Temperaturwertes gelesen werden.

Die Temperatur wird auch vor jeder Luftfeuchtigkeitsmessung gemessen um den Temperatureinfluss kompensieren zu können. Dieser Temperaturwert bleibt bis zur nächsten Messung gespeichert und kann mit dem Befehl *0xE0* wie in Abb. 6.19d dargestellt ist, gelesen werden. Für diese Übertragung gibt es keine Prüfsumme. Unabhängig von der gewählten Bitauflösung, ist das Bit15 für die Messung des Messwertes das höchstwertige Bit und die Bit1:0 haben den binären Wert „00“. Mit der Formel aus Abschn. 6.4.3 kann der gemessene Wert in °C umgerechnet werden.

6.4.2.4 Lesen der elektronischen ID und der Firmware Revision

Die elektronische Identifikationsnummer des Sensors wird in zwei Schritten mit Hilfe von 2-Byte Befehlen gelesen, wie in Abb. 6.19 dargestellt. Um das Lesen abzusichern wird für jedes Byte im ersten Schritt und für jeden 2-Byte-Block im zweiten ein CRC-Byte berechnet und angehängt.

Um die Firmwareversion zu lesen sendet der Master in einer I²C-Lese-Botschaft nach der Adressierungsphase den Befehl *0x84* und *0xB8*. In der Slaveantwort folgt dem Byte mit der Firmware Version kein CRC-Byte.

6.4.3 Berechnung der Temperatur und der relativen Luftfeuchtigkeit

Auf Grund des gelesenen und in diesem Beispiel in die Variable *ucTempOrHumidValue[2]* gespeicherten Messwerts wird die relative Luftfeuchtigkeit mit der Formel aus [14] berechnet:

$$\varphi [\%] = \frac{125 \cdot \varphi_{\text{gelesen}}}{2^{16}} - 6 \quad (6.18)$$

Um die Luftfeuchtigkeit mit einer Auflösung von 0,1 % berechnen zu können wird die Gl. 6.18 mit 10 multipliziert:

$$\varphi [0, 1\%] = \frac{1250 \cdot \varphi_{\text{gelesen}}}{2^{16}} - 60 \quad (6.19)$$

Die Funktion `SI7021_Get_Humidity` kann aufgerufen werden, nachdem eine Messung der relativen Luftfeuchtigkeit abgeschlossen ist, und gibt den gemessenen Wert in 0,1 % als Ganzzahl zurück:

```
unsigned int SI7021_Get_Humidity(void)
{
    unsigned int uihumid = 0;
    long lhumid;
    //die 2 ausgelesenen Bytes werden zusammengefasst
    uihumid = (ucTempOrHumidValue[0] << 8) + ucTempOrHumidValue[1];
    //die Formel wird mit 10 multipliziert damit die Auflösung
    //des Ergebnisses 0,1% beträgt
    lhumid = (long)uihumid * 1250;
    lhumid = (lhumid / 65536) - 60;
    if(lhumid < 0) lhumid = 0; //die Werte kleiner 0%, werden auf
                                //0 gesetzt
    else if(lhumid > 1000) lhumid = 1000; //Werte größer 100%
                                                //werden auf 100% gesetzt
    uihumid = lhumid;
    return uihumid;
}
```

Nach einer Temperaturmessung kann der gelesene Wert in die gleiche Variable wie oben mit dem höherwertigen Byte an der Stelle [0] gespeichert und mit der Formel aus [14] die Lufttemperatur berechnet werden:

$$t (\text{°C}) = \frac{175,72 \cdot t_{\text{gelesen}}}{2^{16}} - 46,85 \quad (6.20)$$

Im Folgenden wird die Umsetzung der Gl. 6.20 im Programmcode vorgestellt. Die erste Beispiefunktion `float SI7021_Get_Temperature(void)` berechnet die Temperatur als Festkommazahl und gibt diesen Wert zurück.

```
float SI7021_Get_Temperature(void)
{
    float ftemp;
    unsigned int uitemp;
    //die 2 ausgelesenen Bytes werden zusammengefasst
    uitemp = (ucTempOrHumidValue[0] << 8) + ucTempOrHumidValue[1];
```

```

    ftemp = (((float) uitemp * 1757.2) / ((float) 65356)) -468.5;
    return ftemp;
}

```

Für die zweite Beispielfunktion wurden die Koeffizienten der Gleichung mit dem Faktor 100 multipliziert, was dazu führt, dass alle Operanden und das Ergebnis Ganzzahlen werden. Weil durch die Multiplikation die Temperaturauflösung auf 0,01 °C erhöht wurde, kann das Ergebnis ganzzahlig durch 10 geteilt werden.

```

int SI7021_Get_Temperature(void)
{
    long ltemp;
    //die Temperatur wird in Ganzzahlarithmetik berechnet
    //die 2 ausgelesene Bytes werden zusammengefasst
    ltemp = (ucTempOrHumidValue[0] << 8) + ucTempOrHumidValue[1];
    //durch die Multiplikation mit 100 der Koeffizienten entstehen
    //Ganzzahlen
    ltemp = ltemp * 4393; //der Bruch (temp * 17572) / 65536) wurde
                           //durch 4 gekürzt
    ltemp = ltemp / 16384;
    ltemp = (ltemp - 4685) / 10; //die Teilung durch 10 bewirkt eine
                                //Auflösung von 0,1°C
    return (int)ltemp;
}

```

Ein Testprogramm, das die Funktion mit der Ganzzahlarithmetik benutzt, ist nach der Kompilierung mit AVRStudio Ver.6.2 um mehr als 800 Bytes kürzer als das Testprogramm, das die erste Beispielfunktion benutzt, unabhängig davon, ob die Optimierung nach Codelänge eingeschaltet oder ganz abgeschaltet war. Dies hängt mit der Float Bibliothek des verwendeten Gnu C-Compilers zusammen.

Mit den berechneten Werten der relativen Luftfeuchtigkeit und der Temperatur kann mit Hilfe der in [15] vorgestellten Magnus-Formel die Taupunkttemperatur berechnet werden:

$$\tau = \frac{B_1 \cdot \left(\ln\left(\frac{\varphi}{100}\right) + \frac{A_{1-t}}{B_{1-t}} \right)}{A_1 - \ln\left(\frac{\varphi}{100}\right) - \frac{A_{1-t}}{B_{1+t}}} \quad (6.21)$$

mit τ = Taupunkttemperatur,

φ = relative Luftfeuchtigkeit,

t = gemessene Lufttemperatur,

$A_1 = 17,625$; $B_1 = 243,04$; $C_1 = 610,94$, const.

6.4.4 Testbarkeit

Der SI7021 bietet Mechanismen an, um festzustellen, ob der gelesene Messwert zur gewünschten Messgröße passt oder ob bei der Übertragung von Botschaften Fehler aufgetreten sind. Dies ist in einem komplexen Netzwerk von Vorteil.

Die Ver-UND-ung eines Luftfeuchtigkeitswertes mit 0x0003 ergibt immer 0x0002, die eines Temperaturwertes 0x0000, dadurch können die Messwerte unterschieden werden.

Der arithmetische Block des Sensors berechnet für jeden Messwert, der unmittelbar einem Messstart folgt, eine Prüfsumme in Form eines CRC-Codes, der vom Master gelesen werden kann um die Übertragung zu prüfen. Solche Prüfsummen müssen bei der Übertragung der elektronischen ID, die einen Sensor in einem Netzwerk eindeutig identifiziert, gelesen werden so wie in Abschn. 6.4.2.4 beschrieben ist. Im Allgemeinen beruht die Berechnung der CRC-Codes auf einer Polynomdivision (siehe [16]). Es gibt unterschiedliche Verfahren um einen CRC-Code zu berechnen, für den SI7021 wurde das CRC-8-Verfahren von der Firma Maxim Integrated [17] implementiert, das für die 1-Wire Bauteile benutzt wird. Das Verfahren verwendet das Generatorpolynom $x^8 + x^5 + x^4 + 1$, der Code wird mit 0x00 initialisiert.

Um den CRC-Code eines Bytes zu ermitteln, können im Vorfeld alle 256 möglichen Werte berechnet und in einer lookup-Tabelle (LUT) entsprechend der Reihenfolge der Byte Werte gespeichert werden. In diesem Fall wird das Byte, dessen CRC-Code zu berechnen ist, zur Adresse der Speicherzelle, in die der Code abgelegt wurde. Es werden keine mathematischen oder logischen Operationen benötigt. Für die 2-Byte-Blöcke ist die Ermittlung des CRC-Codes mit dieser Methode schwer implementierbar, weil die Tabelle 64 kByte groß sein muss. In einem solchen Fall muss der CRC-Code berechnet werden, beispielsweise mit folgender Funktion `uint8_t ComputationCRC(uint16_t ucvalue)`, die das oben genannte Verfahren implementiert. Die Funktion berechnet den CRC-Code eines 1- oder 2-Byte-Wertes und gibt diesen als Rückgabewert zurück.

```
uint8_t ComputationCRC(uint16_t ucvalue)
{
    unsigned long ulTwoBytevalue = 0, ulMaske = 0x800000,
               ulPolynomGenerator = 0x988000;
    uint8_t ucIndex = 0, ucCRC;
    //Initialisierung des Codes
    ulTwoBytevalue |= (unsigned long) ucvalue << 8;
    while(ucIndex < 16) //Polynomdivision
    {
        if(ulTwoBytevalue & ulMaske)
        {
            ulTwoBytevalue = ulTwoBytevalue ^ ulPolynomGenerator;
        }
    }
}
```

```

    ucIndex++;
    ulMaske = ulMaske >> 1;
    ulPolynomGenerator = ulPolynomGenerator >> 1;
}
ucCRC = ultwoBytevalue; //CRC-Code als Rest der Polynomdivision
return ucCRC;
}

```

6.5 HMC5883 Magnetfeldsensor

Der Baustein HMC5883 [18] ermöglicht die Messung der Flussdichte magnetischer Felder in der Stärke des erdmagnetischen Feldes. Das Messverfahren basiert auf dem anisotropen magnetoresistiven Effekt (AMR). Dieser zählt neben dem Hall- und Gauß-Effekt zu den galvanomagnetischen Effekten [19]. Der elektrische Widerstand mancher ferromagnetischen Materialien ändert sich abhängig von der Stärke und Richtung des magnetischen Feldes. Damit Magnetfeldsensoren eine höhere Empfindlichkeit erreichen, werden solche anisotrope, magnetoresistive Elemente in einer Wheatstone-Messbrücke eingebaut. Wenn die Feldlinien senkrecht zu der Messbrücke stehen, ist die Messspannung null, bzw. erreicht ein Maximum, wenn sie parallel verlaufen. Um die genaue Richtung des magnetischen Feldes bestimmen zu können wird eine zweite Messbrücke benötigt, deren Orientierung um 90° gegenüber der ersten gedreht ist. Über eine stromdurchflossene, Spule wird ein zusätzliches Magnetfeld erzeugt, das der Offsetkompensation dient. Eine dauerhafte Magnetisierung der Magnetsensoren kann unter dem Einfluss starker magnetischer Pulse auftreten, oder durch ein über längere Zeit wirkendes, konstantes Magnetfeld. Um diesen Effekt zu vermeiden, muss der Sensor regelmäßig entmagnetisiert werden. Das geschieht beim HMC5883 durch richtungswechselnde, starke und sehr kurze magnetische Pulse, die intern über eine zusätzliche Spule erzeugt werden.

Das natürliche Erdmagnetfeld weist in Europa eine magnetische Flussdichte von ca. 40...50 Mikrotesla (μT) auf [20]. Große ferromagnetische Körper sowie Ablagerungen können zu lokalen magnetischen Anomalien von einigen μT führen. Der HMC5883 kann die magnetische Flussdichte nach drei aufeinander, senkrecht stehenden Achsen messen, und zwar in einem Bereich von 1 Milligauss (mG) bis ca. 8 Gauss, was in SI¹² zu 0,1 μT ... 800 μT entspricht. Somit kann der Baustein zum Vermessen von magnetischen Flussdichten, als Kompass oder zur Feststellung lokaler Anomalien des erdmagnetischen Feldes verwendet werden.

¹² SI – Internationales Einheitssystem.

6.5.1 Aufbau des HMC5883

6.5.1.1 Messfühler

Der Messfühlerblock enthält drei Wheatstone-Messbrücken, die jeweils aus vier anisotropen magnetoresistiven Widerständen bestehen. Darüber hinaus enthält er Spulen für Offsetkompensation und Entmagnetisierung. In jedem Messzyklus werden hintereinander die drei Messspannungen mit einem 12-Bit A/D-Wandler digitalisiert und als 16-Bit Ganzzahlen gespeichert. Die umgewandelten Spannungen decken den diskreten Bereich $-2048 \dots +2047$, bzw. $0xF800 \dots 0x7FF$ ab. Bei einer Bereichsüberschreitung wird für die betroffene Achse der Wert -4096 ($0xF000$) gespeichert. Die Messspannungen sind von der Temperatur linear abhängig, intern findet aber keine Temperaturkompensation statt. Die Kompensation kann softwaremäßig realisiert werden wie es im Abschn. 6.5.3 beschrieben wird.

6.5.1.2 Steuerlogik

Die Steuerlogik sorgt für die Initialisierung des Bausteins nach dem Einschalten der Versorgungsspannung und für die Steuerung der gesamten Messabläufe, der Registeradressierung und der seriellen Schnittstelle. Die Steuerung der Schnittstelle ist prioritätär gegenüber den internen Abläufen. Die Steuerlogik steuert die Statusbits entsprechend dem Zustand des Messablaufs und schaltet den DRDY-Pin um dem Master das Speichern eines neuen Messwertes zu signalisieren. Ein externer pull-up Widerstand ist für diesen Pin nicht nötig.

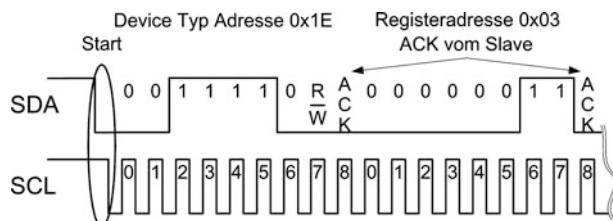
6.5.1.3 Serielle Kommunikation

Der Baustein ist mit einer seriellen Schnittstelle ausgestattet, die das I²C-Protokoll im Standard und Fast Modus implementiert und eine Datenübertragung mit bis zu 400 kBit/s ermöglicht. Die 7Bit Adresse des Slaves lautet 0x1E.

6.5.1.4 Registerblock

Der Registerblock besteht aus dreizehn 8-Bit Registern, die über einen Adresszähler einzeln adressierbar sind und deren Adressen den Bereich 0x00 … 0x0C belegen. Der Inhalt eines Registers ist zugänglich, wenn es vorher adressiert wurde. Beim wahlfreien Zugriff wird der Adresszähler über die serielle Schnittstelle mit dem gewünschten Wert geladen. In der Sequenz aus Abb. 6.20 wird der Zeiger auf die Adresse 0x03 gesetzt.

Abb. 6.20 HMC5883 wahlfreie Registeradressierung



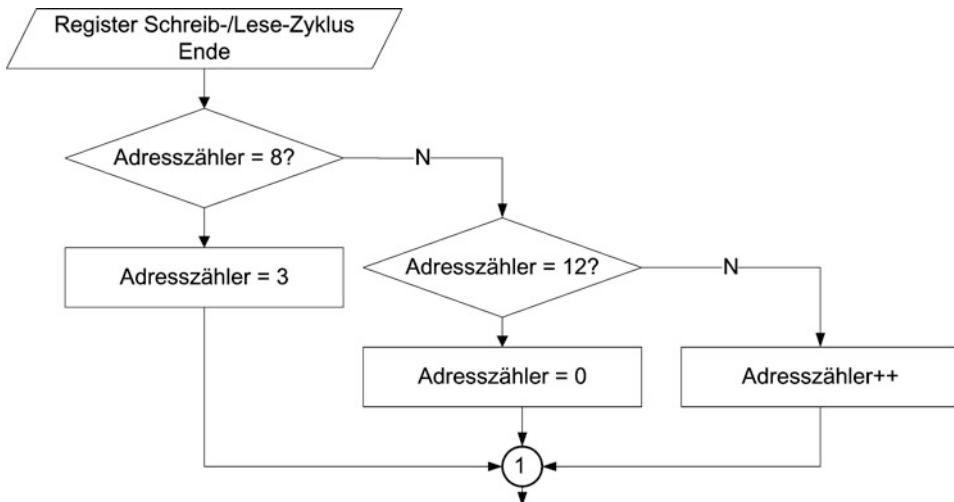


Abb. 6.21 HMC5883 Ansteuerung des internen Adresszählers

Der Master adressiert den Baustein über seine I²C-Adresse im Schreib-Modus und sendet danach die Adresse des gewählten Registers. Nach jedem Schreiben oder Lesen eines Registers wird der Adresszähler so wie im Flussdiagramm Abb. 6.21 geändert. Die Steuerlogik kontrolliert die Registeradressierung und das Auslesen der Messwertregister. Dadurch wird die Software-Konfiguration des Bausteins beschleunigt.

Konfiguration Register

Drei Register dienen der gesamten Konfiguration des Bausteins. Über das Konfigurationsregister A (Adresse 0x00) werden die Messmodi, die Messrate und die Anzahl der gemittelten Messwerte für jeden Messzyklus bestimmt.

Bit 7 – ist immer „0“;

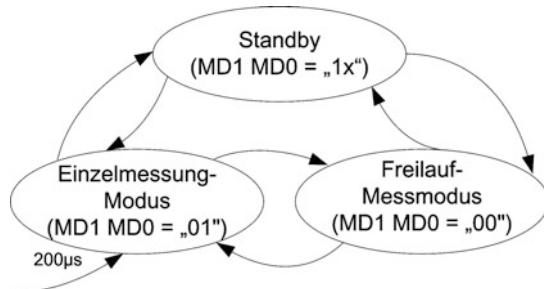
Bit 6:5 – legt die Anzahl der gemittelten Messwerte/den Messzyklus fest: „00“-1, „01“-2, „10“-4 und „11“-8 Messwerte;

Bit 4:2 – mit Werten zwischen „000“ und „110“ wird eine der Messraten: 0,75 Hz, 1,5 Hz, 3 Hz, 7,5 Hz, 15 Hz, 30 Hz, 75 Hz gewählt;

Bit 1:0 – mit diesen zwei Bits wählt man zwischen dem normalen Messmodus „00“ und einem von zwei Testmodi. Bei „01“ wird zu Testzwecken intern ein magnetisches Feld mit einer Flussdichte von 116 µT für die Achsen X und Y erzeugt und von 108 µT für die Achse Z. Bei „10“ wechselt das magnetische Testfeld die Richtung bei gleichbleibender Flussdichte.

Wenn die Bits 4:0 des Konfiguration-Registers B (0x01) auf „0“ gesetzt sind, wird mit den Bits 7:5 einer der Messbereiche zwischen ±88 µT für „000“ und ±810 µT für „111“ ausgewählt.

Abb. 6.22 HMC5883
Betriebsmodi Zustandsüber-
gangsdiagramm



Der Baustein kann über die Bits 1:0 (*MD1* und *MD0*) vom Register *MODE* (0x02) zwischen folgenden Betriebsmodi umschalten: Standby („1x“), Freilauf („00“) und Einzelmessung („01“) wie in Abb. 6.22 dargestellt. Die Bits 7:2 vom gleichen Register müssen „0“ sein. Im Freilauf-Messmodus wird kontinuierlich mit der eingestellten Messrate jeweils eine Messung durchgeführt. Im Einzelmessmodus wird mit dem Setzen des Bit 0 im Register *MODE* eine neue Messung gestartet. Nach dem Speichern der Messwerte schaltet der Baustein in den Standby-Modus um und die Bits *MD1* und *MD0* werden gesetzt.

```

uint8_t HMC5883_Set_Config(uint8_t ucsamples_avg, uint8_t ucout_rate,
                            uint8_t ucmeas_mode, uint8_t ucmeas_range, uint8_t ucop_mode)
{
    uint8_t ucConfigReg[3], ucDeviceAddress, ucI;
    ucConfigReg[0] = ucsamples_avg | ucout_rate | ucmeas_mode;
    ucConfigReg[1] = ucMeasGain[ucmeas_range];
    ucConfigReg[2] = ucop_mode;
    //Adresse des elektronischen Kompasses bilden
    ucDeviceAddress = HMC5883_DEVICE_TYPE_ADDRESS << 1;
    ucDeviceAddress |= TWI_WRITE; //Write-Modus
    TWI_Master_Start(); //Start
    if((TWI_STATUS_REGISTER) != TWI_START) return TWI_ERROR;
    //Device Adresse senden
    TWI_Master_Transmit(ucDeviceAddress);
    if((TWI_STATUS_REGISTER) != TWI_MT_SLA_ACK) return TWI_ERROR;
    //die Adresse des 1. Konfigurationsregisters wird gesendet
    TWI_Master_Transmit(0x00);
    if((TWI_STATUS_REGISTER) != TWI_MT_DATA_ACK) return TWI_ERROR;
    //die Datenbytes werden gesendet
    for(ucI = 0; ucI < 3; ucI++)
    {
        TWI_Master_Transmit(ucConfigReg[ucI]);
        if((TWI_STATUS_REGISTER) != TWI_MT_DATA_ACK) return TWI_ERROR;
    }
    TWI_Master_Stop(); //Stop
  
```

Tab. 6.15 HMC5883 Datenregister

Achse	X	Z	Y
Register	MSByte	LSByte	MSByte
Adresse	0x03	0x04	0x05

```
    return TWI_OK;
}
```

Mit der Funktion `HMC5883_Set_Config()` kann der Baustein neu konfiguriert werden. Ein Beispiel für den Aufruf der Funktion ist im Folgenden erläutert.

```
#define SAMPLES_AVG8      0x60      //der Mittelwert von 8 Messungen
                                //wird gebildet
#define OUT_RATE_1          0x00      //0,75Hz Messrate
#define MEAS_MODE_NORM       0x00      //der normale Messmodus wird gewählt
#define MEAS_GAIN_5          0x04      //±400 µT Messbereich
#define OP_MODE_SINGLE        0x01      //Einzelmessmodus

HMC5883_Set_Config(SAMPLES_AVG8, OUT_RATE_1, MEAS_MODE_NORM,
                    MEAS_GAIN_5, OP_MODE_SINGLE);
```

Messwert Register

Ein neuer Messwert wird als Mittelwert zweier Aufnahmen berechnet. Vor jeder Aufnahme wird je ein Entmagnetisierungsfeld mit wechselnder Richtung erzeugt. Dadurch vermeidet man eine eventuelle Dauermagnetisierung der magnetoresistiven Widerstände. Die gemessenen Werte der magnetischen Flussdichte werden für jede Achse in je zwei Register im „Big-Endian“ Format gespeichert wie in Tab. 6.15 dargestellt.

Statusregister (Adresse 0x09)

Das Status Register liefert Informationen über den Zustand des Messverlaufs und Datenauslesens.

Bit 7:5, 3:2 – die Bits sind immer „0“;

Bit 4 – ist nicht dokumentiert; bei den getesteten Exemplaren wird dieses Bit gesetzt, wenn ungelesene Messwerte überschrieben wurden. Es wird nach dem Lesen eines kompletten Messwertsatzes zurückgesetzt.

Bit 1 – dieses Bit wird gesetzt, wenn das Register *MODE* gelesen wird oder mit dem Beginn des Datenauslesens. Es wird zurückgesetzt: beim Speichern des Konfigurationsregister A, oder des Registers *MODE* oder wenn ein kompletter Messwertsatz ausgelesen wurde. Solange dieses Bit gesetzt ist, werden keine neuen Messwerte in die Datenregister gespeichert.

Bit 0 – ist während der Dauer des Speicherns der Messwerte in den Datenregistern zurückgesetzt (mindestens 250 µs).

Die Änderung der einzelnen Bits des Statusregisters während des Freilauf-Messbetriebs mit asynchronem Datenlesen ist in Abb. 6.23 dargestellt.

Identifikationsregister

Der HMC5883 besitzt drei Nur-Lese-Register mit den Adressen 0x0A, 0x0B und 0x0C die zur Identifikation des Bausteins und zum Testen der seriellen Kommunikation dienen. Die gespeicherten Inhalte der Register lauten: 0x48, 0x34 und 0x33 welche die ASCII-Codes der Zeichen „H“, „4“ und „3“ sind.

6.5.2 HMC5883 Messwerte lesen

Die Dauer einer Messung mit Mittelung von acht Aufnahmen dauert laut [18] 6 ms. Das Auslesen der sechs Datenregister kann beispielsweise mit der Übertragung von 9 Bytes erfolgen:

- 1 Byte für die Adressierung des Sensors im Write-Modus;
- 1 Byte – die Adresse des ersten Datenregisters (0x03);
- 1 Byte für die Adressierung des Sensors im Read-Modus;
- 6 Bytes für die Übertragung der Datenregister.

Im idealen Fall würde die Übertragung der 9x9 Bits, bei einer maximalen Bitrate von 400 kBit/s, 202,5 µs dauern.

6.5.2.1 Messwerte lesen im Einzelmessbetrieb

Im Einzelmessbetrieb wird jede Messung mit dem Setzen des Bits 0 im Register *MODE* gestartet. Weil die Messdauer wesentlich größer ist als die Übertragungsdauer, kann unmittelbar nach dem Start der ($n + 1$) Messung, das Auslesen der n -te Messwerte durchgeführt werden wie im folgenden Codeausschnitt:

```
HMC5883_Write_ByteReg(MODE_REG, 0x01); //Start einer neuen Messung
HMC5883_Read_DataReg(uiData); //Auslesen der 6 Datenregister
```

In diesem Messmodus wird das Bit 1 im Statusregister nach dem Lesen eines oder mehreren Datenregistern gesetzt. Dieses Bit wird aber mit jedem neuen Start zurückgesetzt, so dass auch nur einzelne Register in diesem Modus gelesen werden können. Über ein blockierendes Warten kann mit der Abfrage des Zustandes des Bit 0 vom Statusregister ein Master den richtigen Zeitpunkt ermitteln, an dem die Datenregister unmittelbar nach einem Messvorgang ausgelesen werden können, wie im folgenden Programmausschnitt zu sehen ist.

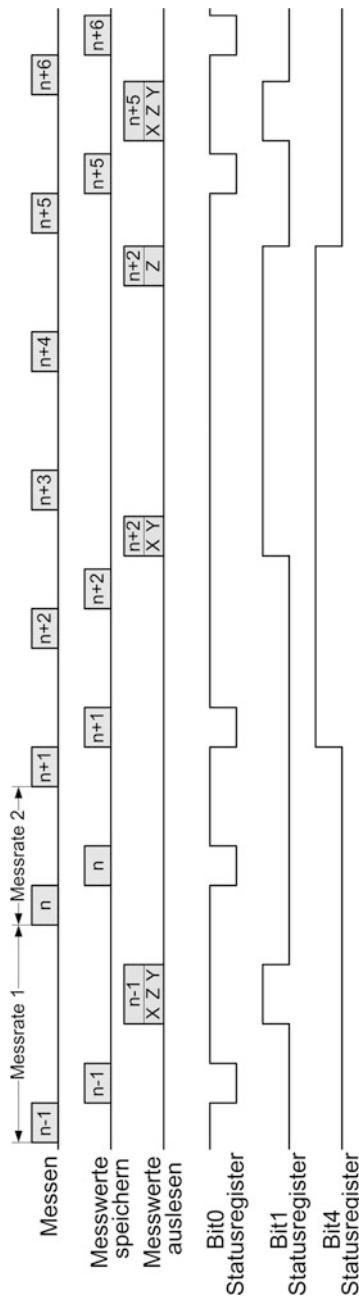


Abb. 6.23 HMC5883 Freilauf-Messbetrieb – Schematische Darstellung mit asynchronem Datenlesen

```

unsigned char ucStatusReg = 0x01;
/*die Funktion HMC5883_Read_ByteReg liest den Inhalt des Registers
 MODE aus und speichert ihn in die Variable ucStatusReg*/
//es wird gewartet bis das Bit 0 im Statusregister auf Low geht
while(ucStatusReg & 0x01) HMC5883_Read_ByteReg(STATUS_REG, ucStatusReg);
//wenn das Bit 0 vom Statusregister wieder auf High geht,
//Ende der Speicherung
while(!ucStatusReg) HMC5883_Read_ByteReg(STATUS_REG, ucStatusReg);

```

6.5.2.2 Messwerte lesen im Freilauf-Messbetrieb

Eine Besonderheit des Sensors besteht darin, dass in diesem Messmodus beim unvollständigen Lesen aller Datenregister neue Messwerte aufgenommen werden, aber nicht in die Datenregister gespeichert werden können. Im Freilauf-Betrieb können die Messwerte im asynchronen oder synchronen Modus gelesen werden.

Asynchrones Lesen im Freilauf-Betrieb

So wie in Abschn. 6.5.1.4, „Messwert Register“ beschrieben wird das Überschreiben ungelesener Messwerte signalisiert, aber nicht blockiert. Über die Dauer des gesamten Lesens der sechs Datenregister wird das Speichern neuer Messwerte verriegelt. Das ermöglicht das asynchrone Lesen zu einem beliebigen Zeitpunkt ohne die Gefahr, dass die Inhalte der Datenregister von unterschiedlichen Messungen stammen.

Synchrones Lesen im Freilauf-Betrieb

Wenn alle Messwerte ausgelesen werden müssen, spricht man vom synchronen Lesen. Eine erste Möglichkeit, die Daten synchron zu lesen, wäre die Ermittlung des Lesezeitpunktes über das Auslesen des Bit 0 vom Statusregister. Besonders bei höheren Datenraten ist wegen des blockierenden Wartens das Verfahren nicht zu empfehlen.

Um das unwirtschaftliche Warten zu vermeiden, verfügt der Baustein über den Anschluss DRDY der gleichzeitig mit dem Bit 0 aus dem Statusregister von der Steuerlogik angesteuert wird. Mit dem Schalten auf Low, kann bei entsprechender Beschaltung dem Master signalisiert werden, dass ein neuer Messwert zum Lesen bereitsteht. Wenn der mit dem DRDY verbundene Pin vom Master interruptfähig ist, kann die entsprechende Interrupt Service Routine rechtzeitig das Auslesen der Messwerte anleiten.

6.5.3 Kalibrierung des Sensors

Ziel der Kalibrierung ist eine genauere Umwandlung der gemessenen Werte in Einheiten der magnetischen Flussdichte. Mit den Bits 7:5 des Konfigurationsregisters B kann einer der acht Messbereiche ausgewählt werden. Für jeden Messbereich gibt der Hersteller einen Verstärkungsfaktor an, welcher dem digitalen Wert bei der Messung einer Flussdichte von 1 G (100 µT) entspricht. Mit den Bits 1:0 des Konfigurationsregisters A können

Testmodi aktiviert werden um interne magnetische Felder bekannter Größe zu erzeugen und damit den Sensor zu kalibrieren.

Beispielsweise wird für den 5. Messbereich (Bit 7:5 = „100“) die Messung einer maximalen Flussdichte von $\pm 400 \mu\text{T}$ empfohlen und ein Verstärkungsfaktor von 440 angegeben. Unter dem Einfluss der Testfelder sind digitale Werte von $\pm 1,16 \times 440 = \pm 510$ für die Achsen X und Y und $\pm 1,08 \times 440 = \pm 475$ für die Achse Z zu erwarten. Die Kalibrierung ist jeweils auf einen Messbereich bezogen und temperaturabhängig. Folgende Schritte sind für die Kalibrierung nötig:

- der Messbereich wird ausgewählt;
- der Testmodus 1 wird aktiviert und eine Einzelmessung gestartet;
- die Messwerte werden ausgelesen;
- der Testmodus 2 wird aktiviert und eine Einzelmessung gestartet;
- die Messwerte werden ausgelesen;
- aus den positiven und negativen Messwerten werden für jede Achse der Offset- und der Korrekturwert des Verstärkungsfaktors berechnet.

Bei der Kalibrierung eines Sensors im 5. Messbereich sind die Werte aus der Tab. 6.16 ausgelesen worden.

Die Offset- und Verstärkungsfaktoren als Ganzzahlen werden am Beispiel der Achse X folgendermaßen berechnet und die Ergebnisse für alle drei Achsen in der Tab. 6.17 gespeichert.

Beispiel

$$X_{\text{Offset}} = (+X_{\text{Mess}} + (-X_{\text{Mess}})) / 2 = (569 - 550) / 2 = +9$$

$$V_X_{\text{Korr}} = X_{\text{Test}} / (+X_{\text{Mess}} - X_{\text{Offset}}) = 510 / 560$$

Um die kalibrierten, digitalen Werte der Flussdichte für diesen Messbereich zu berechnen, werden aus den ausgelesenen Werte zuerst die entsprechenden Offsetwerte subtrahiert und danach die Ergebnisse mit den korrigierten Verstärkungsfaktoren multipliziert. Durch die Teilung der korrigierten Werte durch 4,4 (100 / 440) werden die Messergebnisse in μT ausgedrückt. Bei Änderung des Messbereiches oder der Umgebungstemperatur sollte der Kalibriervorgang wiederholt werden.

Tab. 6.16 Beispiel Kalibrierung: gemessene Werte

+X_Mess	-X_Mess	+Y_Mess	-Y_Mess	+Z_Mess	-Z_Mess
+569	-550	+525	-525	+499	-480

Tab. 6.17 Beispiel Kalibrierung: berechnete Offsetwerte und Verstärkungsfaktoren

X_Offset	Y_Offset	Z_Offset	V_X_Korr	V_Y_Korr	V_Z_Korr
+9	0	+9	510/560	1	475/490

6.5.4 HMC5883 als elektronischer Kompass

Der Sensor kann wegen seiner hohen Empfindlichkeit als elektronischer Kompass verwendet werden. Das erdmagnetische Feld verläuft parallel zur Erdoberfläche und als vektorielle Größe ist es immer auf den magnetischen Nordpol gerichtet [21]. Die Orientierung des Sensors im magnetischen Feld kann deshalb aus den Messwerten der Achsen X (X_Mess) und Y (Y_Mess) ermittelt werden. Die Berechnung des Winkels zwischen der Y-Achse des Sensors und dem Nordpol wird im folgenden Beispiel erläutert. Bei 0° soll die Richtung der positiven Y-Achse auf den magnetischen Nordpol zeigen. Als Voraussetzung für die Winkelbestimmung müssen sich die Punkte $P_i(X_{\text{Mess}_i}/Y_{\text{Mess}_i})$ auf einem Kreis befinden, was in der Regel nicht der Fall ist. Der Radius des Kreises ist bedeutungslos, deshalb kann man für die Kalibrierung des Sensors als elektronischer Kompass ein vereinfachtes Verfahren anwenden, dessen Grundlage in [21] vorgestellt ist. Dieses Verfahren führt zu einem niedrigeren Rechenaufwand. Nach der Auswahl des gewünschten Messbereiches werden folgende Schritte durchgeführt:

1. In einer magnetisch ungestörten Umgebung wird der horizontal platzierte Sensor um seine Z-Achse gedreht. Der kleinste (negative) und der größte (positive) Messwert der Achsen X (X_Min und X_Max) und Y (Y_Min und Y_Max) werden während des Drehens ermittelt und gespeichert.
2. Für die zwei Achsen werden die Offsetwerte als Ganzzahlen berechnet, gespeichert und weiterhin aus den Messwerten subtrahiert.

$$\begin{cases} X_{\text{Offset}} = (X_{\text{Max}} + X_{\text{Min}})/2 \\ Y_{\text{Offset}} = (Y_{\text{Max}} + Y_{\text{Min}})/2 \end{cases} \quad (6.22)$$

Unter Berücksichtigung der Offsetwerte wird der Schritt 1. wiederholt und geprüft, dass $X_{\text{Max}} \approx X_{\text{Min}}$ und $Y_{\text{Max}} \approx Y_{\text{Min}}$. Die aktuellen Maxima X_Max und Y_Max werden gespeichert. Wenn z. B. $X_{\text{Max}} > Y_{\text{Max}}$, dann werden die Messwerte weiterhin folgendermaßen korrigiert:

$$\begin{cases} X_{\text{Korr}} = X_{\text{Mess}} - X_{\text{Offset}} \\ Y_{\text{Korr}} = ((Y_{\text{Mess}} - Y_{\text{Offset}}) \cdot X_{\text{Max}})/Y_{\text{Max}} \end{cases} \quad (6.23)$$

Mit den korrigierten Werten unter Berücksichtigung des Quadranten, in dem sich diese Werte befinden, kann der gesuchte Winkel zwischen der Y-Achse und der Nordpolrichtung mit Hilfe der trigonometrischen Funktion Arkustangens berechnet werden.

$$\beta = \text{atan}\left(\frac{X_{\text{Korr}}}{Y_{\text{Korr}}}\right) \quad (6.24)$$

6.5.5 Winkelberechnung mit dem CORDIC Algorithmus

Um den Winkel aus Gl. 6.24 mit einem Mikrocontroller zu berechnen, müsste zuerst eine Division durchgeführt und danach eine Funktion aufgerufen werden, die den Arkustangens des Quotienten berechnet. Eine Alternative wäre die Berechnung der möglichen Winkelwerte im Vorfeld mit einer akzeptablen Auflösung entsprechend dem Quotienten X_Korr / Y_Korr als Ganzzahl und das Speichern dieser Werte in einer Tabelle (LUT¹³). Der Zugriff auf die gespeicherten Werte würde dann über den Quotienten erfolgen, der als Tabellenindex benutzt wird. Die erste Lösung ist zeitaufwendig, die zweite ist extrem platzintensiv. Als Alternative wird die Anwendung des CORDIC Algorithmus für die Berechnung des Winkels präsentiert [22, 23].

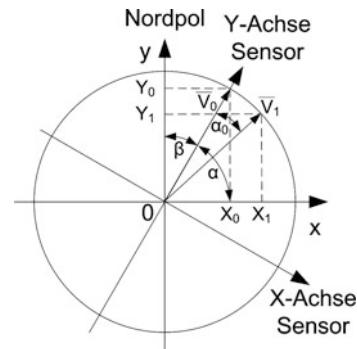
CORDIC ist das Akronym von COordinate Rotation Digital Computer und bezeichnet einen mathematischen Algorithmus, mit dem annähernd trigonometrische Funktionen berechnet werden können. Ein Vektor \vec{V}_0 in der xy-Ebene, dessen Komponente X_0 und Y_0 sind, führt im Sinne des Algorithmus durch Rotation im Uhrzeigersinn mit dem Winkel α_0 zu einem Vektor \vec{V}_1 mit den Komponenten X_1 und Y_1 (siehe Abb. 6.24).

$$\begin{cases} X_1 = X_0 + Y_0 \cdot \tan \alpha_0 \\ Y_1 = Y_0 - X_0 \cdot \tan \alpha_0 \end{cases} \quad (6.25)$$

In einem iterativen Vorgang wird der Vektor \vec{V}_1 weiter in diskreten Schritten rotiert bis die Ordinate des resultierenden Vektors ungefähr Null ist.

$$\begin{cases} X_{i+1} = X_i + Y_i \cdot \sigma_i \cdot \tan \alpha_i \\ Y_{i+1} = Y_i - X_i \cdot \sigma_i \cdot \tan \alpha_i \end{cases} \quad (6.26)$$

Abb. 6.24 HMC5883 – elektronischer Kompass



¹³ Lookup table.

Tab. 6.18 Stützwerte für den CORDIC Algorithmus

$\tan \alpha_i$	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}
α_i	450	265	140	71	36	18	9	4

mit

$$\sigma_i = \begin{cases} +1, & \text{wenn } Y_i > 0 \\ -1, & \text{wenn } Y_i < 0 \end{cases} \quad (6.27)$$

Mit $\tan \alpha_i = 2^{-i}$ lassen sich die Komponenten des rotierenden Vektors lediglich durch die mathematischen Grundfunktionen eines Mikrocontrollers „Addition“ und „Stellenverschiebung nach rechts“ berechnen, was zur Beschleunigung der Winkelberechnung führt. In der Tab. 6.18 sind die dazugehörigen Winkel α_i für acht Iterationsschritte als Ganzzahlen in Zehntelgrad Auflösung aufgelistet.

Wenn das Abbruchkriterium des Algorithmus erfüllt oder die maximale Anzahl n der Schritte erreicht ist, wird der gesamte Rotationswinkel berechnet

$$\alpha = \sum_{i=0}^{n-1} \sigma_i \cdot \alpha_i \quad (6.28)$$

und damit ergibt sich der gesuchte Winkel $\beta = 90^\circ - \alpha$. Die folgende Funktion gibt beim Aufruf mit den korrigierten Messwerten X_Korr und Y_Korr den Winkel zwischen der Y-Achse und dem Nordpol zurück.

```
uint16_t HMC5883_Get_Angle(uint16_t uix_value, uint16_t uiy_value)
{
    uint8_t ucQuadrant = 0, ucI;
    uint16_t uiX_Value, uiY_Value, uiAngle = 0;
    //die Winkel als Stützpunkte in Zehntelgrad
    uint16_t uiPhi[8] = {450, 265, 140, 71, 36, 18, 9, 4};
    if(uiX_value & 0x8000) //wenn wahr, ist der X-Wert negativ
    {
        uix_value = ~uix_value + 1;
        ucQuadrant |= 0x01;
    }
    if(uiy_value & 0x8000) //wenn wahr, ist der Y-Wert negativ
    {
        uiy_value = ~uiy_value + 1;
        ucQuadrant |= 0x02;
    }
    //Annäherung des Winkels im 1. Quadrant in 8 Schritte
    for(ucI = 0; ucI < 8; ucI++)
    {
```

```

        if(uiy_value & 0x8000)
    {
        uiAngle -= uiPhi[ucI];
        uiX_Value = uix_value - ((int)uiy_value >> ucI);
        uiY_Value = uiy_value + (uix_value >> ucI);
    }
    else
    {
        uiAngle += uiPhi[ucI];
        uiX_Value = uix_value + ((int)uiy_value >> ucI);
        uiY_Value = uiy_value - (uix_value >> ucI);
    }
    uix_value = uiX_Value;
    uiy_value = uiY_Value;
}
switch(ucQuadrant) //Berechnung des realen Winkels vom
                    //Quadranten abhängig
{
    case 0: uiAngle = 2700 + uiAngle;      break;
    case 1: uiAngle = 900 - uiAngle;      break;
    case 2: uiAngle = 2700 - uiAngle;      break;
    case 3: uiAngle = 900 + uiAngle;      break;
}
return uiAngle;
}

```

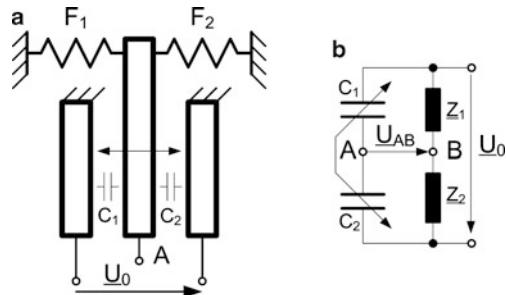
6.6 Beschleunigungssensoren

6.6.1 Beschleunigungssensor ADXL312

ADXL312 ist ein mikroelektromechanischer Sensor (MEMS) [24] der nach einem kapazitiven Messverfahren die Beschleunigung entlang drei aufeinander senkrecht stehenden Achsen im Bereich von $\pm 12 \text{ g}$ misst ($1 \text{ g} \approx 9,81 \text{ m/s}^2$). Das kapazitive Messverfahren ermöglicht eine gute Linearität und einen niedrigen Temperaturkoeffizienten [25]. Mit einer Messrate von bis 3200 Messungen/s wird eine Bandbreite von 0 bis 1600 Hz abgedeckt. Mit der Messung der Erdbeschleunigung kann der Baustein auch als Neigungssensor verwendet werden.

Prinzipiell besteht der Messfühler eines solchen Sensors für eine Messachse aus einer doppelt gefederten seismischen Masse die sich zwischen zwei ortsfesten Elektroden entlang einer Gerade bewegen kann. Zusammen mit den Elektroden bildet die seismische Masse zwei Kondensatoren, deren Kapazitätsänderung proportional zur Bewegung beziehungsweise Beschleunigung des Bausteins ist, wie in Abb. 6.25a. Um die Empfindlichkeit

Abb. 6.25 ADXL312 Prinzipieller Aufbau eines MEMS kapazitiver Beschleunigungssensors



des Messfühlers zu erhöhen, wird eine Kammstruktur verwendet, wie in [25] beschrieben. Der Messfühler wird elektrisch in eine Brückenschaltung (siehe Abb. 6.25b) eingebaut, deren analoge Ausschlagsspannung abgetastet, digitalisiert und nach einer digitalen Filterung gespeichert wird. Abhängig von der Messrate und der Komplexität der Vernetzung können die gemessenen Werte entweder in Datenregister oder in Zwischenpuffer mit 32 Speicherplätzen je Achse gespeichert werden. Die gespeicherten Werte können dann über eine serielle Schnittstelle gelesen werden.

Der Sensor verfügt über einen Registersatz, der die Konfiguration der Messung, die Datenspeicherung, die Kommunikation und die Energieverwaltung flexibel gestalten kann. Ein Master, der mit dem Baustein verbunden ist, kann über den Zugriff auf diesen Registern Einstellungen ändern, Messdaten lesen, den Baustein testen, oder ihn in einem Netzwerk identifizieren. Der Stromverbrauch soll abhängig von den konkret verfolgten Zielen optimiert werden, um den Baustein auch in batteriebetriebenen Geräte einsetzen zu können. Mit einer flexiblen Interruptverwaltung und zwei Interruptausgängen kann der Baustein in komplexen Netzwerkstrukturen eingesetzt werden.

6.6.1.1 Vernetzung des ADXL312

Für die Vernetzung des Sensors, der als Slave konfiguriert ist, sind zwei serielle Schnittstellen vorgesehen: SPI und I²C. Diese Schnittstellen benutzen gemeinsam die Anschlüsse SDI (Data In), CLK (Taktsignal), SDO (Data Out) und CS (Chip Select) (siehe Tab. 6.19). Mit dem Chip Select Signal wird eine der Schnittstellen automatisch aktiviert. Ein Master kann mit dem Sensor im 4-wire oder 3-wire SPI-Modus bei einer maximalen Datenrate von 5 Mbit/s im SPI-Modus 3 kommunizieren. Das höchstwertige Bit eines Bytes wird zuerst gesendet. Standardmäßig ist der 4-wire Modus aktiv. Der 3-wire Modus kann nach dem Hochfahren aktiviert werden indem das Bit 6 vom Register DATA_FORMAT gesetzt wird. Die Elemente der Datenstruktur, die den Sensor für die Benutzung eines globalen SPI-Moduls eindeutig identifizieren, haben folgende Bedeutung:

- 1. Element – die Adresse des DDR-Registers an dem die Slave Select Leitung angeschlossen ist;
- 2. Element – die Adresse des PORT-Registers an dem die Slave Select Leitung angeschlossen ist;

Tab. 6.19 ADXL312 Kommunikationsanschlüsse

Anschluss		I ² C		SPI 4-wire	SPI 3-wire
CS	0	Inaktiv		Aktiv	
	1	Aktiv		Inaktiv	
SCL/SCLK		Clock Eingang			
SDA/SDI/SDIO		Daten Ein/Ausgang		Dateneingang	Daten Ein/Ausgang
SDO	0	0x53	Device Chip Adresse	Datenausgang	Angeschlossen direkt an V _{DD} oder über 10 kΩ an GND
	1	0x31			

- 3. Element – das entsprechende Bit der Slave Select Leitung;
- 4. Element – „1“ wenn die Leitung angesteuert wird („0“ wenn bei fest angeschlossener Leitung).

Nachdem die SPI-Kommunikation mit dem Setzen der CS-Leitung auf Low initiiert wurde, wird ein Startbyte gesendet, das die Adresse des gewünschten Registers beinhaltet. Die Registeradressen befinden sich im Bereich: 0x00 ... 0x3F. Das höchstwertige Bit des Startbytes bestimmt die Richtung des Datenflusses:

- eine „0“ signalisiert das Überschreiben des adressierten Registers (WRITE); auf das Startbyte folgt der neue Inhalt des Registers.
- mit einer „1“ wird ein Lesebefehl codiert; der Master sendet nach dem Startbyte ein beliebiges weiteres Byte (meist 0x00 oder 0xFF) um die Antwort des Sensors zu empfangen.

Wenn das Bit 6 des Startbytes gesetzt ist, wird mit jedem Registerzugriff der interne Adresszähler inkrementiert, was das Lesen oder Schreiben mehrerer zusammenhängender Register in einem Vorgang ermöglicht. Das Startbyte enthält in diesem Fall die Adresse des ersten Registers.

Bei einer Point-to-point Verbindung zwischen einem Master und dem Sensor kann der Master eindeutig die Kommunikationsschnittstelle bestimmen. Im Fall einer SPI-Vernetzung (Abb. 6.26a) wird die SPI-Schnittstelle dadurch angewählt, dass der Master die CS-Leitung auf Low zieht, und der Sensor wertet die ankommen Bytes korrekt als SPI-Nachricht aus. Wenn der Master mit dem Slave1 kommuniziert (CS = High, CS₁ = Low), findet am SDI-Eingang des Sensors ein Datenfluss statt, der zusammen mit dem Clock-Signal vom Sensor als I²C-Botschaft interpretiert werden kann. Um Busstörungen zu vermeiden muss verhindert werden, dass am SDI-Eingang das Signal Low wird während CS High ist [24]. Ein ODER-Gatter (siehe Abb. 6.26b) kann das Problem im Fall der 4-wire Übertragung lösen. Möchte man die hohe Bandbreite des Sensors nutzen und die dadurch entstehende große Datenmenge auslesen, muss die SPI-Schnittstelle verwendet werden.

Wenn die Messfrequenz niedrig ist oder lediglich die Erdbeschleunigung gemessen wird, kann der Sensor über I²C vernetzt werden, wie in Abb. 6.27 dargestellt. In diesem

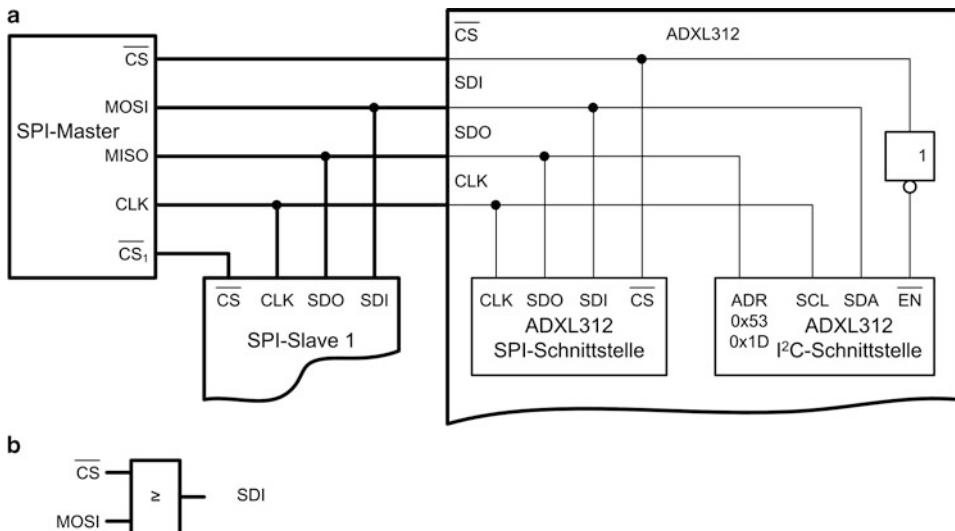


Abb. 6.26 ADXL312 SPI-Vernetzung

Fall wird der ADXL312 dafür benutzt, die horizontale Lage des elektronischen Kompasses HMC5883 festzustellen. Der CS-Anschluss ist fest mit V_{DD} verbunden und die 7-Bit I²C-Adresse ist auf 0x53 fixiert. Weiterhin wird die Kommunikation mit dem Sensor über I²C näher betrachtet die, bei 400 kBit/s stattfinden kann. Über das Register TWBR des Mikrocontrollers wird die Taktfrequenz der Schnittstelle eingestellt. Abhängig von der Taktfrequenz des Masters [Hz] und der Schnittstelle wird der Inhalt des Registers berechnet:

```
#define TWI_SCL_FREQ 400000UL //TWI Clockfrequenz in Hz für den Master
#define TWI_MASTER_CLOCK (((F_CPU / TWI_SCL_FREQ) - 16) / 2 +1)
```

und mit dem Aufruf der Funktion `TWI_Master_Init(TWI_MASTER_CLOCK)` gespeichert.

6.6.1.2 Messdatenerfassung

Der Baustein wurde zur Messung von Beschleunigungen und Neigungswinkeln entwickelt. Wegen seiner hohen Belastbarkeit ($\leq 10.000\text{ g}$) kann er auch für das Detektieren von mechanischen Schocks verwendet werden.

Initialisierung des Bausteins

Der Sensor wird über diverse Register für den konkreten Einsatz initialisiert. Mit der folgenden Funktion wird das Register mit der Adresse `ucreg_address` mit dem Wert `ucdata_byte` geladen. Bei erfolgreicher Ausführung der Funktion wird `TWI_OK` zurückgegeben, ansonsten `TWI_ERROR`.

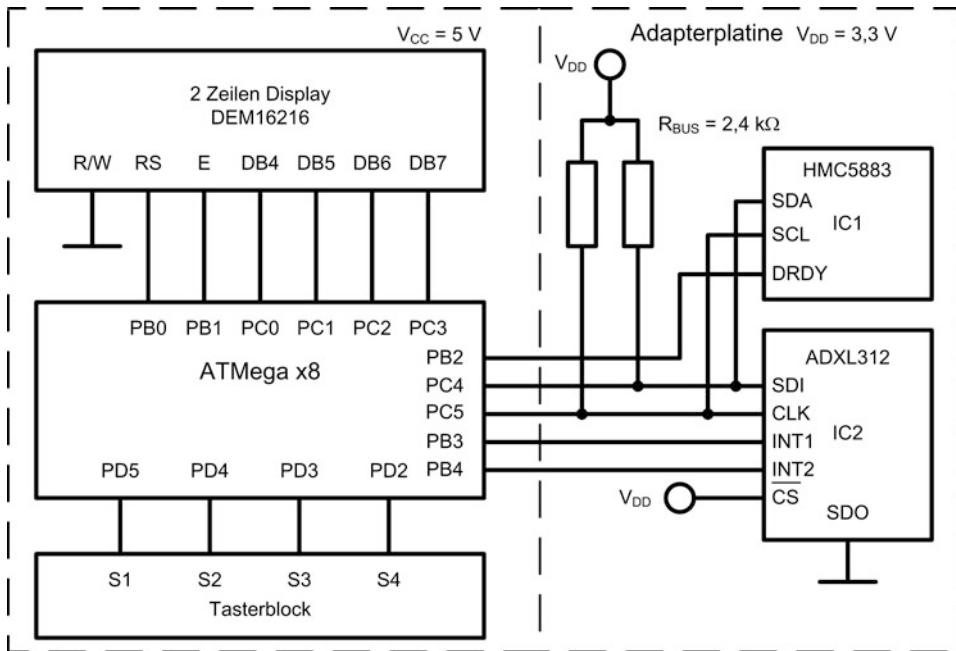


Abb. 6.27 ADXL312 I²C-Vernetzung

```
#define ADXL312_DEVICE_TYPE_ADDRESS 0x53
uint8_t ADXL312_I2C_Write_ByteReg(uint8_t ucreg_address,
                                    uint8_t ucdatal_byte)
{
    uint8_t ucDeviceAddress;
    //Adresse des Beschleunigungssensors bilden
    ucDeviceAddress = ADXL312_DEVICE_TYPE_ADDRESS << 1;
    ucDeviceAddress |= TWI_WRITE; //Write-Modus
    TWI_Master_Start(); //Start
    if((TWI_STATUS_REGISTER) != TWI_START) return TWI_ERROR;
    TWI_Master_Transmit(ucDeviceAddress); //Device Adresse senden
    if((TWI_STATUS_REGISTER) != TWI_MT_SLA_ACK) return TWI_ERROR;
    TWI_Master_Transmit(ucreg_address); //die Adresse des
                                        //Zielregisters wird gesendet
    if((TWI_STATUS_REGISTER) != TWI_MT_DATA_ACK) return TWI_ERROR;
    TWI_Master_Transmit(ucdata_byte); //das Datenbyte wird gesendet
    if((TWI_STATUS_REGISTER) != TWI_MT_DATA_ACK) return TWI_ERROR;
    TWI_Master_Stop(); //Stop
    return TWI_OK;
}
```

Arbeitsmodus

Über die Einstellung des Registers *POWER_CTL* (Adresse 0x0D) kann der Sensor direkt in einen der Modi: Sleep, Standby oder Messmodus versetzt werden. Über die Einstellung des gleichen Registers kann der Sensor im Sleep-Modus bleiben solange die Messwerte kleiner als ein Grenzwert sind. Dieser Grenzwert wird im Register *THRESH_INACT* gespeichert. Der Messmodus ist ein Freilaufmessmodus in dem die analogen Werte der drei Messachsen mit einer eingestellten Frequenz abgetastet, digitalisiert und anschließend gespeichert werden.

Während die anderen Bits „0“ sind, kann man mit dem Bit 3 vom Register *POWER_CTL* zwischen Standby-Modus (Bit 3 = „0“) und Messmodus (Bit 3 = „1“) umschalten. Nach dem Einschalten fährt der Baustein im Standby-Modus hoch um Strom zu sparen.

Messfrequenzeinstellung

Die Bits 3:0 vom Register *BW_RATE* (0x2C) codieren die Messrate. Die kleinste, einstellbare Messfrequenz ist 6,25 Hz, die dem Code „0110“ entspricht. Die Messfrequenz verdoppelt sich mit dem Inkrementieren des Codewertes und erreicht 3200 Hz bei „1111“. Die mögliche Bandbreite der zu messenden Beschleunigungen ist die Hälfte der eingesetzten Messfrequenz. Mit dem Setzen des Bits 4 vom gleichen Register werden die Messungen im Bereich 12,5 Hz ... 400 Hz mit reduziertem Stromverbrauch durchgeführt. In diesem stromsparenden Messmodus kann der Rauschpegel höher sein.

Messdatenformat

Der gesamte Messbereich von $\pm 12\text{ g}$ ist in weitere drei Bereiche unterteilt, um eine höhere Messauflösung zu erreichen. Die A/D-Wandlung erfolgt über die vier Messbereiche entweder mit einer konstanten 10 Bit Auflösung, oder mit einer konstanten Schrittweite von ca. 2,9 mg. Die Einstellungen, die mit dem Register *DATA_FORMAT* (0x31) möglich sind, können folgender Beschreibung entnommen werden:

- Bit 7** – SELF_TEST – mit dem Setzen dieses Bits wird intern eine elektrostatische Kraft erzeugt, die zum Versetzen der seismischen Masse führt. Auf alle drei Achsen wird ein Offset überlagert um den Sensor testen zu können.
- Bit 6** – SPI – der Sensor fährt in den 4-wire SPI-Modus hoch. Mit dem Setzen dieses Bits wird auf den 3-wire SPI-Modus umgeschaltet.
- Bit 5** – INT_INVERT – mit „0“ sind die Interrupt-Ausgänge high-aktiv, mit „1“ werden sie low-aktiv;
- Bit 4** – = „0“;
- Bit 3** – FULL_RES – bestimmt die Schrittweite des A/D-Wandlers, wie in Tab. 6.20 aufgelistet. Wenn dieses Bit gesetzt ist, werden die Messwerte mit der niedrigsten Schrittweite umgewandelt, die Bitauflösung stellt sich automatisch mit der Wahl des Messbereiches um;

Tab. 6.20 ADXL312 Einstellung des Messbereiches, der Bitauflösung und der Schrittweite

DATA_FORMAT	Bit3:Bit1:Bit0							
	000	001	010	011	100	101	110	111
Messbereich [g]	±1,5	±3	±6	±12	±1,5	±3	±6	±12
Bitauflösung [Bit]	10	10	10	10	10	11	12	13
Schrittweite [mg]	2,9	5,8	11,6	23,2	2,9	2,9	2,9	2,9

Bit 2 – JUSTIFY- mit „0“ sind die Messergebnisse als 16 Bit Zahl, rechtsbündig im Zweierkomplement Format in 2 1-Byte Register gespeichert; mit „1“ werden sie linksbündig gespeichert;

Bit 1:0 – bestimmen laut Tab. 6.20 den Messbereich.

Messdaten speichern

Die gemessenen Werte von jeder Messachse werden als 16 Bit Zahl wahlweise entweder in zwei 1-Byte Register oder in einen 32x2 Byte FIFO-Speicherpuffer gespeichert. Vor dem Speichern wird jeder Beschleunigungswert aller Achsen zum Inhalt des entsprechenden Offsetregisters addiert. Dies spart entsprechende Rechenzeit des Masters. Die Offsetregister sind 1-Byte große flüchtige Register und speichern die Werte im Zweierkomplement mit einer Auflösung von 11,6 mg. Der Inhalt dieser Register wird beim Hochfahren auf 0x00 gesetzt. Die Speichermodi des Bausteins werden mit den Bits 7:6 des Registers *FIFO_CTRL* (0x38) folgendermaßen eingestellt:

- **Bit 7:6 = „00“;** im Bypass-Modus werden die Messwerte direkt in die Datenregister im Little-Endian Format gespeichert (siehe Tab. 6.21). Der Sensor erlaubt in diesem Modus das Überschreiben des ungelesenen Datensatzes, bzw. der ungelesenen Datenregister. Mit der Wahl dieses Modus werden alle Speicherzellen des FIFO-Puffers auf „0x00“ gesetzt.
- **Bit 7:6 = „01“;** im FIFO-Modus wird ein neuer Datensatz nur solange in den Puffer gespeichert, solange freie Speicherplätze vorhanden sind. Mit dem Lesen eines Datensatzes in diesem Modus wird ein Speicherplatz frei.
- **Bit 7:6 = „10“;** im Stream-Modus werden die neuen Datensätze kontinuierlich in den Puffer gespeichert. Wenn der Puffer voll ist, wird der älteste Datensatz vom Neusten überschrieben.
- **Bit 7:6 = „11“;** im Trigger-Modus bleiben die letzten „n“-Messwerte vor dem Auslösen eines Interrupts gespeichert. Weitere (32 – n) Datensätze füllen den Puffer voll und anschließend wird ein Interrupt am Pin INT1 ausgelöst, wenn das Bit 5 vom Register *FIFO_CTRL* „0“ ist oder am INT2 wenn das Bit „1“ ist. In diesem Modus können die Ereignisse, die einen Interrupt auslösen, untersucht werden. Neue Datensätze können in diesem Modus erst nach Löschen des Zwischenspeichers gespeichert werden wie im folgenden Codeabschnitt gezeigt:

Tab. 6.21 ADXL312 Daten- und Offsetregister

Messachse	Datenregister		Offsetregister
	Höherwertiges Byte	Niederwertiges Byte	
X	DATA_X_MSB 0x33	DATA_X_LSB 0x32	OFS_X 0x1E
Y	DATA_Y_MSB 0x35	DATA_Y_LSB 0x34	OFS_Y 0x1F
Z	DATA_Z_MSB 0x37	DATA_Z_LSB 0x36	OFS_Z 0x20

```
#define FIFO_CTRL_REG      0x38 //Adresse des Registers
ADXL312_I2C_Write_ByteReg(FIFO_CTRL_REG, 0x00); //Umschalten auf Bypass
ADXL312_I2C_Write_ByteReg(FIFO_CTRL_REG, 0xDF); /*Umschalten auf
Trigger, 15 Messwerte vor dem Ereignis speichern, Interrupt am
INT1 0xDF = 1101 1111*/
```

In den Bits 4:0 des Registers *FIFO_CTRL* wird im Trigger-Modus die Zahl „n“ gespeichert, im FIFO- und Stream-Modus die Füllgrenze des Puffers. Wenn die ungelesenen Datensätze aus dem FIFO-Puffer diese Füllgrenze erreichen, wird ein Interrupt ausgelöst. Die aktuelle Anzahl der ungelesenen Datensätze aus dem Puffer ist in den Bits 5:0 des Registers *FIFO_STATUS* (0x39) gespeichert.

Lesen der Messwerte

Im Bypass-Modus steht der letzte gespeicherte Datensatz in den Datenregistern zum Lesen bereit. In diesem Modus können auch nur einzelne Datenregister gelesen werden, es gibt aber keinen Schutzmechanismus, der das Überschreiben eines ungelesenen Datensatzes verhindert. Bei den Speichermodi, die den FIFO-Puffer benutzen, steht der älteste, ungelesene Datensatz in den Datenregistern. Nach dem Auslesen der 6 Datenregister, mit dem Sprung des internen Adresszählers auf die Adresse 0x38 wird innerhalb von 5 µs der nächste Datensatz in die Datenregister verschoben. Diese Verzögerungszeit muss beim Datenlesen aus dem FIFO-Puffer eingehalten werden um das Lesen eines konsistenten Datensatzes sichern zu können. Mit dem Aufruf folgender Funktion wird über I²C ein kompletter Datensatz ausgelesen und die Messwerte in einen Vektor vom Typ *unsigned int* gespeichert.

```
uint8_t ADXL312_I2C_Read_DataReg(uint16_t* uidata_word)
{
    uint8_t ucDeviceAddress, ucDataLSByte, ucDataMSByte, ucI;
    //Adresse des Beschleunigungssensors bilden
    ucDeviceAddress = ADXL312_DEVICE_TYPE_ADDRESS << 1;
    ucDeviceAddress |= TWI_WRITE; //Write-Modus
    TWI_Master_Start(); //Start
    if((TWI_STATUS_REGISTER) != TWI_START) return TWI_ERROR;
    TWI_Master_Transmit(ucDeviceAddress); //Device Adresse senden
    if((TWI_STATUS_REGISTER) != TWI_MT_SLA_ACK) return TWI_ERROR;
```

```

//die Anfangsadresse des Registerbereiches wird gesendet
TWI_Master_Transmit(DATA_X_MSB_REG);
if((TWI_STATUS_REGISTER) != TWI_MT_DATA_ACK) return TWI_ERROR;
TWI_Master_Start(); //Restart
if((TWI_STATUS_REGISTER) != TWI_RESTART) return TWI_ERROR;
ucDeviceAddress = (ADXL312_DEVICE_TYPE_ADDRESS << 1) | TWI_READ;
//Read-Modus
TWI_Master_Transmit(ucDeviceAddress); //Device Adresse im Read-
                                         //Modus senden
if((TWI_STATUS_REGISTER) != TWI_MR_SLA_ACK) return TWI_ERROR;
//Inhalt der adressierten Register wird eingelesen
for(ucI = 0; ucI < 3; ucI++)
{
    ucDataLSByte = TWI_Master_Read_Ack(); //das niedwertige Byte
                                         //wird gelesen
    if((TWI_STATUS_REGISTER) != TWI_MR_DATA_ACK) return TWI_ERROR;
    if(ucI < 2)
    {
        ucDataMSByte = TWI_Master_Read_Ack(); //höherwertiges Byte
                                         //wird gelesen
        if((TWI_STATUS_REGISTER) != TWI_MR_DATA_ACK) return TWI_ERROR;
        uidata_word[ucI] = (ucDataMSByte << 8) + ucDataLSByte;
    }
    else
    {
        ucDataMSByte = TWI_Master_Read_NAck(); //letztes Register
                                         //wird gelesen
        if((TWI_STATUS_REGISTER) != TWI_MR_DATA_NACK) return TWI_ERROR;
        uidata_word[ucI] = (ucDataMSByte << 8) + ucDataLSByte;
    }
}
TWI_Master_Stop(); //Stop
return TWI_OK;
}

```

6.6.1.3 Offset Ermittlung

Wenn die absoluten Werte der Beschleunigung benötigt werden, wie beispielsweise für die Berechnung des Neigungswinkels, müssen die Offsetwerte berücksichtigt werden. Für die Bestimmung des Offsets liest der Master die gemessenen Beschleunigungen und bildet den arithmetischen Mittelwert über mindestens zwei oder mehr Messwerte, um eventuelle Spitzen (Ausreißer) herauszufiltern (gleitender Durchschnitt). Mit einer Drehbewegung um jede Achse werden die Maxima (positive Werte) $iValueMax_i$ und die Minima (negativen Werte) $iValueMin_i$ erfasst und gespeichert. Der Offsetwert für die i.-Achse wird mit

der Gleichung:

$$iOffset_i = \frac{iValueMax_i + iValueMin_i}{2} \quad (6.29)$$

entsprechend dem gewählten Messbereich berechnet. Die berechneten Offsetwerte müssen abhängig von der gewählten Schrittweite (siehe Tab. 6.21) mit einem Faktor korrigiert werden, bevor sie in die Offset-Register gespeichert werden:

$$iOffsetKorr_i = -\left(iOffset_i \cdot \frac{\text{OffsetReg_i Auflösung}}{\text{Messbereich Auflösung}} \right) \quad (6.30)$$

Für einen gewählten Messbereich von $\pm 1,5$ g und eine Auflösung des Offsetregisters von 11,6 mg kann der korrigierte Offsetwert der X-Achse mit folgendem Programmausschnitt berechnet und gespeichert werden:

```
#define OFFSET_X_REG      0x1E //Adresse des Offsetregisters
int iValueMax_X, iValueMin_X, iOffset_X; //Deklaration der Variablen
iOffset_X = (iValueMax_X + iValueMin_X) / 8; //Teilen durch (11,6/2,9)x2
iOffset_X = ~iOffset_X + 1; //Multiplikation mit (-1)
ADXL312_I2C_Write_ByteReg(OFFSET_X_REG, iOffset_X); //Speichern des
//Offsets
```

Die gemessenen Werte werden mit den Inhalten der Offsetregister intern addiert bevor sie in die Datenregister gespeichert werden. Werte um die 0 für die X- und die Y-Achse zeigen die horizontale Lage des Sensors.

6.6.1.4 Interruptmodus

ADXL312 verfügt über eine komplexe Interrupt Struktur, die zur Entlastung des Masters führen soll. Die frei konfigurierbaren Interrupts steuern beim Auslösen zwei push-pull Ausgänge. Das Erfüllen einer einen Interrupt auslösenden Bedingung wird von der internen Logik durch das Setzen des entsprechenden Bits in das Register *INT_SOURCE* (0x30) signalisiert, das im Folgenden beschrieben ist.

- Bit 7** – DATA_READY – ist dieses Bit gesetzt, dann zeigt es an, dass ein neuer Datensatz vorhanden ist. Diese Interruptquelle kann im Bypass-Modus benutzt werden, um alle Messwerte ohne blockierendes Warten lesen zu können.
- Bit 6 5, 2** – nicht benutzt
- Bit 4** – ACTIVITY – dieses Bit wird gesetzt, wenn ein gemessener Wert den Grenzwert überschreitet, der im Register *TRESH_ACT* (0x24) gespeichert ist.
- BIT 3** – INACTIVITY – wenn die gemessenen Werte den Wert, der im Register *TRESH_INACT* (0x25) abgelegt ist, über die Zeit *TIME_INACT* (0x26) unterschreiten, wird dieses Bit gesetzt. Die Zeit wird in das Register in Sekunden gespeichert.

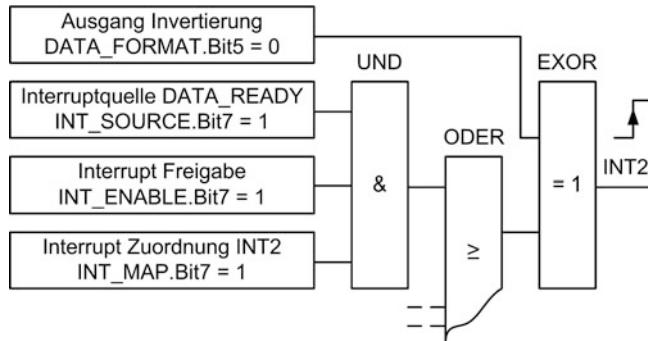


Abb. 6.28 ADXL312 Interrupt Konfiguration und Pin Zuordnung

- Bit 1** – WATERMARK – das Bit kann im FIFO- oder Stream-Modus gesetzt werden, wenn die Anzahl der ungelesenen Datensätze aus dem FIFO-Puffer gleich mit der Füllgrenze ist.
- Bit 0** – OVERFLOW – das gesetzte Bit zeigt an, dass ungelesene Datensätze überschrieben wurden.

Die Register *TRESH_ACT* und *TRESH_INACT* sind 8 Bit Register, die die Grenzwerte ohne Vorzeichen mit einer Auflösung von 46,4 mg speichern. Die gleiche Bitzuordnung wie das Register *INT_SOURCE* haben auch die Register *INT_ENABLE* und *INT_MAP*. Das Register *INT_ENABLE* (0x2E) ermöglicht die Freigabe jeder Interruptquelle durch das Setzen des entsprechenden Bits. Die freigegebenen Interrupts können über das Register *INT_MAP* (0x2F) dem Ausgang INT1 oder INT2 zugeordnet werden. Wird in diesem Register ein Bit gesetzt, so wird der entsprechende freigegebene Interrupt dem Ausgang INT2 zugeordnet, ansonsten dem INT1. Mehrere Interrupts können dem gleichen Ausgang über eine ODER-Verknüpfung zugeordnet werden wie in Abb. 6.28. Weitere Details bezüglich Interrupts können dem Datenblatt des Bausteins [24] entnommen werden.

Mit dem folgenden Programmausschnitt wird der *DATA_READY* Interrupt zusätzlich zu den vorhandenen Interrupts eingestellt (siehe Abb. 6.28).

```

uint8_t ucRegByte;
ADXL312_I2C_Read_ByteReg(INT_SOURCE_REG, &ucRegByte);
ucRegByte |= 0x80; //das Bit 7 wird gesetzt
ADXL312_I2C_Write_ByteReg(INT_SOURCE_REG, ucRegByte); //Interruptquelle
//wird gewählt

ADXL312_I2C_Read_ByteReg(INT_MAP_REG, &ucRegByte);
ucRegByte |= 0x80;
ADXL312_I2C_Write_ByteReg(INT_MAP_REG, ucRegByte); //Interrupt Zuordnung
ADXL312_I2C_Read_ByteReg(INT_ENABLE_REG, &ucRegByte);
ucRegByte |= 0x80;
ADXL312_I2C_Write_ByteReg(INT_ENABLE_REG, ucRegByte); //Interrupt Freigabe
  
```

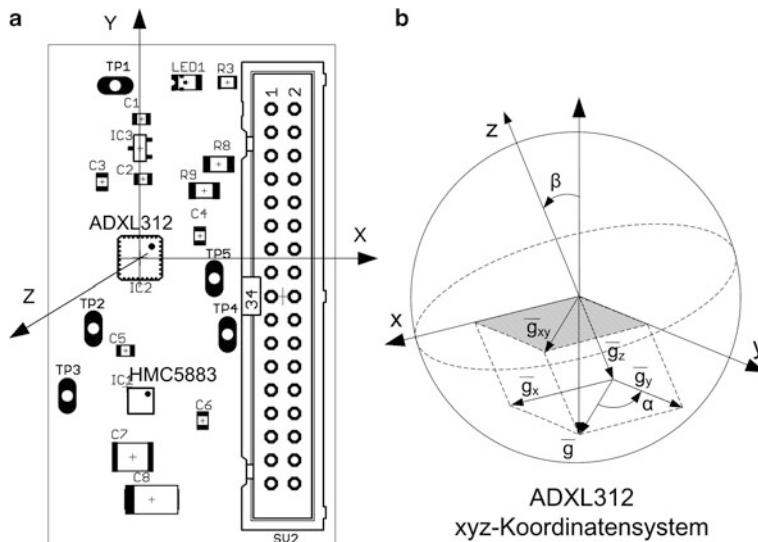


Abb. 6.29 ADXL312 als Neigungssensor

6.6.1.5 ADXL312 als Neigungssensor

Im Folgenden wird beispielhaft eine Möglichkeit vorgestellt, den ADXL312 als Neigungssensor zu benutzen. Die Abb. 6.29a zeigt die Achsenanordnung des Sensors, der wie in Abb. 6.27 beschaltet ist. Wenn der Winkel zwischen der positiven Richtung einer Achse und dem Vektor der Gravitationsbeschleunigung 180° beträgt, wird an jener Achse $+1\text{ g}$ gemessen. Abhängig von der Sensorlage teilt sich die Erdbeschleunigung \bar{g} in die Komponenten g_x , g_y und g_z auf, die von dem Sensor gemessen werden, wobei

$$g^2 = g_x^2 + g_y^2 + g_z^2 \quad (6.31)$$

Theoretischer Ansatz

Das Ziel ist die Berechnung des Neigungswinkels β (siehe Abb. 6.29b zwischen der Vertikale und der eigenen Z-Achse des Sensors und die Neigungsrichtung über den Winkel α . Der Neigungswinkel nimmt Werte zwischen 0° und 180° an und kann mit Hilfe des COR-DIC-Algorithmus berechnet werden, wenn die Komponenten g_z und g_{xy} bekannt sind. Die Komponente g_z wird direkt gemessen und g_{xy} kann mit dem Satz des Pythagoras berechnet werden:

$$g_{xy} = \sqrt{g_x^2 + g_y^2}, \quad (6.32)$$

was aber für einen 8 Bit-Mikrocontroller zeitaufwendig ist. Die Neigungsrichtung wird über den Winkel α zwischen dem Vektor \bar{g}_{xy} und der Y-Achse bestimmt. Bei sehr kleinen Neigungswinkeln beeinflusst das Rauschen die Genauigkeit der Richtungsberechnung, weil die X- und Y-Werte sehr klein sind. Dieser Winkel nimmt Werte von 0° bis 360° an

und kann mit dem in Gl. 6.26 vorgestellten CORDIC-Algorithmus berechnet werden. Am Ende des iterativen Verfahrens für die Winkelberechnung liegt der Endvektor $\overline{g'_{xy}}$ nahezu auf der X-Achse und liefert ein Maß für den Betrag g_{xy} . Laut [22] stehen die Vektorbeträge g_{xy} und g'_{xy} im folgenden Verhältnis:

$$\begin{aligned} g'_{xy} &= k \cdot g_{xy}, \quad k > 1 \\ k &= \prod_{i=0}^{n-1} \sqrt{1 + 2^{-2i}} \end{aligned} \quad (6.33)$$

wobei n die Anzahl der Iterationen bedeutet. Für $n=8$ ist $k \approx 1,646473506$. Durch die Teilung der Endvektorlänge durch den Faktor k erhält man den gewünschten Betrag g_{xy} . Die im Abschn. 6.5.5 vorgestellte Funktion für die Winkelberechnung mit der Ergänzung:

```
uint16_t uiLength; //Vektorbetrag
uiLength = uiValue_X * 128; //der Betrag des Endvektors wird
                           //durch 1,648 geteilt
uiLength = uiLength / 211;
```

kann neben dem Winkel α auch den Betrag $uiLength$ des Vektors $\overline{g_{xy}}$ liefern. Die Division durch eine Gleitkommazahl wurde mit einer Verschiebung und einer Division durch eine Ganzzahl ersetzt. Der relative Fehler der dadurch entsteht, liegt bei ca. 0,1 %. Mit dem ermittelten g_{xy} ist die Voraussetzung für die Berechnung des Neigungswinkels mit dem CORDIC-Algorithmus erfüllt.

Korrektur der Messwerte

Bei der Berechnung des Neigungswinkels und der Neigungsrichtung mit einem 8 Bit Mikrocontroller muss ein Kompromiss zwischen Rechenaufwand, Genauigkeit und Messrate getroffen werden. Eine höhere Genauigkeit wird mit der Wahl eines Messbereiches mit einer niedrigen Schrittweite erreicht (siehe Tab. 6.20). Nach der Initialisierung des Beschleunigungssensors müssen die Offsetwerte wie etwa in Abschn. 6.6.1.3 beschrieben berechnet werden. Mit einer Schrittweite von 2,9 mg und der festen Auflösung der Offset-Register von 11,6 mg ergibt sich nach der Korrektur mit dem beschriebenen Verfahren ein maximales Offset von $\pm 7 \cdot 2,9$ mg wenn:

$$iValueMax_i + iValueMin_i = n \cdot 8 - 1 \quad n \in \mathbb{N} \quad (6.34)$$

was die Genauigkeit der Berechnungen beeinträchtigt. Bessere Ergebnisse können erzielt werden, wenn die mit der Gl. 6.29 berechneten Offsetwerte direkt zu den ausgelesenen Messwerten addiert werden. Mit zusätzlichen drei Additionen ergibt sich ein maximaler Offset von ± 1 .

Die Korrektur der Messwerte hat die Berechnung der Korrekturfaktoren zum Ziel, mit denen die Messwerte multipliziert werden damit Gl. 6.35 erfüllt ist.

$$g_{x_korr}^2 + g_{y_korr}^2 + g_{z_korr}^2 = \text{const.} \quad (6.35)$$

- Es werden die maximalen, positiven Werte für alle drei Achsen mit Berücksichtigung des Offsets berechnet:
 $iValueMax_i = iValueMax_i + iOffset_i$
- der höchste Wert aller drei Werte wird ermittelt:
 $iValueMax = \max(iValueMax_i)$
- die Korrekturfaktoren für die drei Achsen werden berechnet:
 $iValueKorr_i = iValueMax / iValueMax_i$.

Bevor die gemessenen Beschleunigungswerte für die Berechnung des Neigungswinkels und der Neigungsrichtung eingesetzt werden, werden die entsprechenden Offsetwerte dazu addiert und die Ergebnisse mit den Korrekturfaktoren multipliziert wie im folgenden Codeausschnitt für die X-Achse:

```
iValue_X = iValue_X + iOffset_X;
iValue_X = (iValue_X * iValueMax) / iValueMax_X;
```

Berechnung des Neigungswinkels und der Neigungsrichtung

Mit den korrigierten Messwerten der X- und Y-Achse werden der Winkel α und der Betrag g_{xy} wie beschrieben berechnet. Mit dem Aufruf der Funktion `ADXL312_Get_TiltAngle()` mit dem Betrag `uixy_value` und dem korrigierten Wert der Z-Achse `uiz_value` als Parameter wird auch der Neigungswinkel in Zehntelgrad berechnet.

```
uint16_t ADXL312_Get_TiltAngle(uint16_t uixy_value, uint16_t uiz_value)
{
    uint8_t ucQuadrant = 0, ucI;
    uint16_t uiXY_Value, uiZ_Value, uiAngle = 0;
    //die Winkel als Stützpunkte in Zehntelgrad
    uint16_t uiPhi[8] = {450, 265, 140, 71, 36, 18, 9, 4};
    if(uixy_value & 0x8000) uixy_value = ~uixy_value + 1;
    //wenn wahr, ist X negativ
    if(uiz_value & 0x8000) //wenn wahr, ist der Y-Wert negativ
    {
        uiz_value = ~uiz_value + 1;
        ucQuadrant |= 0x01;
    }
    for(ucI = 0; ucI < 8; ucI++) //Annäherung des Winkels im
                                //1. Quadrant in 8 Schritte
    {
        if(uiz_value & 0x8000)
        {
            uiAngle -= uiPhi[ucI];
            uiXY_Value = uixy_value - ((int)uiz_value >> ucI);
            uiZ_Value = uiz_value + (uixy_value >> ucI);
        }
    }
}
```

```
    else
    {
        uiAngle += uiPhi[ucI];
        uiXY_Value = uixy_value + ((int)uiz_value >> ucI);
        uiZ_Value = uiz_value - (uixy_value >> ucI);
    }
    uixy_value = uiXY_Value;
    uiz_value = uiZ_Value;
}
switch(ucQuadrant) //Berechnung des realen Winkels
{
    case 0: uiAngle = 900 - uiAngle;
    break;
    case 1: uiAngle = 900 + uiAngle;
    break;
}
return uiAngle;
}
```

6.6.2 MMA6525

MMA6525 ist ein 2-Achsen Beschleunigungssensor, der Beschleunigungen im Bereich $\pm 105,5\text{ g}$ mit einer Auflösung von $0,055\text{ g}$ messen kann [26]. Er wurde für die Ausstattung von Airbagsteuergeräten im Automotive Bereich entwickelt, kann aber auch für andere Anwendungen verwendet werden, in denen hohe Beschleunigungen zu erwarten sind.

6.6.2.1 Sensoraufbau

Das Blockschaltbild des Sensors ist in Abb. 6.30 dargestellt. Um die hohen Sicherheitsanforderungen eines Airbag Sensors zu erfüllen, ist für die Ansteuerung und Datenverarbeitung jeder Achse ein DSP vorgesehen. Die Beschleunigungen werden kontinuierlich im Freilaufmodus gemessen. Die Messfühler liefern eine analoge Spannung, die proportional zur Beschleunigung ist. Diese Spannung wird verstärkt, mit einem 12 Bit A/D-Wandler digitalisiert und danach digital gefiltert. Die auswählbare Grenzfrequenz des Filters liegt im Bereich $50\text{ Hz} \dots 1000\text{ Hz}$. Die digitalen Beschleunigungswerte können mit oder ohne Vorzeichen ausgelesen werden. Der Messbereich des Sensors ist mit vorzeichenlosen Werten im Bereich $+128 \dots +3968$ und mit vorzeichenbehafteten Werten im Bereich $-1920 \dots +1920$ abgedeckt. Die Mitte eines Bereiches (2048 bzw. 0) bildet den Wert 0 g ab. Jeder DSP steuert den gesamten digitalen Informationsfluss einer Messachse, tauscht Daten mit dem internen Speicher aus und ist über eine 4-Leitung SPI-Schnittstelle mit dem steuernden Mikrocontroller verbunden. Der DSP kann den Messfühler in einen Testmodus versetzen, indem er das elektrostatische Messprinzip ausnutzt. Die Überschreitung eines einstellbaren Beschleunigungsbereiches kann in zeitkritischen Anwendungen über

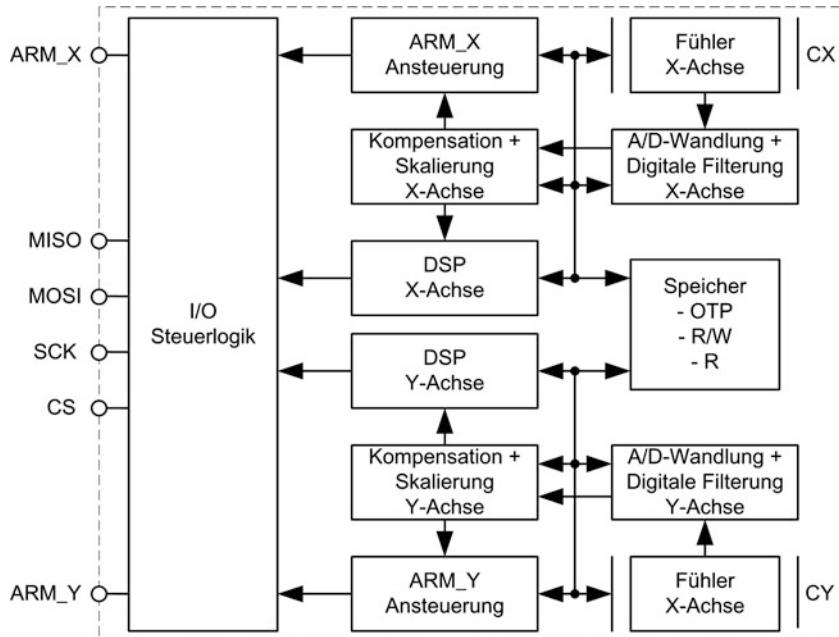


Abb. 6.30 MMA6525 Blockschaltbild

die Anschlüsse ARM_X und ARM_Y asynchron signalisiert werden, beispielsweise um einen Airbag zu entriegeln. Diese Anschlüsse können natürlich auch einen Interrupt auslösen.

6.6.2.2 Registerblock

Der Benutzer kann auf die Register im Adressbereich $0x00 \dots 0x1D$ zugreifen.

OTP¹⁴-Register

Die Register im Adressbereich $0x00 \dots 0x08$ sind vom Hersteller einmalig programmiert und können nicht umprogrammiert werden. Die Adresse $0x08$ speichert den Wert $0xE1$ zur Identifikation des Messbereiches $\pm 105,5\text{ g}$. Jedem Baustein wird eine einmalige 32-Bit große Identifikationsnummer zugewiesen, die an den Adressen $0x00 \dots 0x03$ gespeichert wird. Beim Hersteller wird der Beschleunigungswert im Testmodus gelesen und für die X-Achse an der Adresse $0x04$ und für die Y-Achse an der Adresse $0x05$ gespeichert. Bei der Wiederholung der Messungen im Testmodus dürfen die gemessenen Werte von den gespeicherten um nicht mehr als $\pm 10\%$ abweichen. Für diese neun Speicherzellen wurde eine Prüfsumme berechnet und gespeichert. Im normalen Betrieb wird die Prüfsumme neu berechnet und mit der gespeicherten verglichen. Bei Unstimmigkeit wird ein persistenter Fehler signalisiert.

¹⁴ OTP – one time programmable (einmalig programmierbar).

Schreib-Lese-Register

Die gespeicherten Werte im Adressbereich 0x0A...0x13 dienen zur Konfiguration der Achsen und können geschrieben und gelesen werden. Die Register mit Adressen von 0x0E bis 0x13 dienen zur Einstellung der interruptfähigen Ausgänge *ARM_X* und *ARM_Y* und zur Speicherung der positiven und negativen Beschleunigungsgrenzen für jede Achse. Über das Kontrollregister *DEVCTL* (Adresse 0x0A) kann der Sensor zurückgesetzt werden und über die Konfigurationsregister *DEVCFG_X* (0x0C) und *DEVCFG_Y* (0x0D) kann der Testmodus für die entsprechende Achse aktiviert und die Grenzfrequenz des digitalen Filters ausgewählt werden. Die allgemeine Konfiguration des Sensors erfolgt über das Register *DEVCFG* (0x0B) das im Folgenden beschrieben wird:

- Bit 7** – OC – mit dem Rücksetzen dieses Bits liefert der Sensor dynamische Beschleunigungswerte;
- Bit 6** – reserviert;
- Bit 5** – ENDINIT – der Master setzt dieses Bit auf „1“ um die Initialisierungsphase zu beenden und den normalen Betriebsmodus zu starten; dadurch wird das Schreiben des Speicherbereichs bis auf das Register *DEVCTL* gesperrt und eine Prüfsumme dieses Registerbereichs wird berechnet und gespeichert.
- Bit 4** – SD – wenn dieses Bit „1“ ist, werden die gemessenen Werte vorzeichenlos ausgegeben;
- Bit 3** – OFMON – wenn gesetzt, überwacht die Sensorlogik das Überschreiten der gespeicherten Beschleunigungsgrenzen;
- Bit 2...0** – mit diesen drei Bits auf „0“ werden die ARM-Ausgänge deaktiviert.

Nur-Lese-Register

Die Registerwerte aus dem Adressbereich 0x14 bis 0x17 werden intern aktualisiert und können nur gelesen werden. Die von der Sensorlogik ermittelten Offsetwerte für die X- und Y-Achse werden in das Register *OFFCORR_X* (0x16) bzw. *OFFCORR_Y* (0x17) gespeichert. Der aktuelle Status des Sensors ist von dem Register *DEVSTAT* (0x14) abgebildet.

- Bit 7** – reserviert;
- Bit 6** – IDE – wird bei Unstimmigkeit der Prüfsummen gesetzt;
- Bit 5** – reserviert;
- Bit 4** – DEVINIT – wird nur während der internen Initialisierungsphase gesetzt;
- Bit 3** – MISOERR – signalisiert gesetzt einen SPI-Übertragungsfehler;
- Bit 2...1** – OFF_X, OFF_Y – die Bits werden gesetzt, wenn die Offsetgrenzen erreicht wurden;
- Bit 0** – DEVRES – wird während der Initialisierung als Folge eines Reset gesetzt.

Die von dem Sensor gesetzten Fehlerbits (bis auf *DEVINIT*) können durch das Lesen des Statusregisters gelöscht werden.

6.6.2.3 SPI-Kommunikation

Der Sensor MMA6525 ist als SPI-Slave im Modus 0 konfiguriert. Die Datenstruktur eines Sensors, dessen Chip Select Eingang mit dem Anschluss PB0 eines Mikrocontrollers vom Typ ATmega168 verbunden ist, sieht wie folgt aus (Abschn. 6.1.5):

```
MMA65XX_pins MMA65XX_1 = {{/*CS_DDR*/      &DDRB,
                           /*CS_PORT*/     &PORTB,
                           /*CS_pin*/      PB0,
                           /*CS_state*/    ON}}; //ON = 1
```

Eine SPI-Nachricht an den Sensor besteht immer aus zwei Bytes, das höherwertige Bit wird zuerst übertragen. Während des Empfangs eines Befehls sendet der Sensor eine Rückmeldung auf die vorige Nachricht. Jeder Befehl ist mit einem Paritätsbit gesichert, das vom Sensor geprüft wird. Bei Unstimmigkeit wird ein SPI-Fehler signalisiert. Die vom Sensor gesendete Rückmeldung ist ebenfalls mit einem Paritätsbit versehen, das vom Mikrocontroller geprüft werden kann. Es wird mit ungerader Parität berechnet.

Initialisierung des Sensors

Die Initialisierung des Bausteins soll frühestens 10 ms nachdem er versorgt wurde, stattfinden. Während der Initialisierung wird der Sensor konfiguriert und sein Fehlerstatus überprüft. Zusätzlich kann seine Identifikationsnummer geprüft und im Testmodus die Messfühler und -kette getestet werden. Eventuelle auftretende Fehler müssen von der Software abgefangen und behandelt werden. Die fehlerfreie Initialisierung wird mit dem Setzen des Bits DEVINIT in das Register DEVCFG beendet und damit wird der normale Arbeitsmodus des Sensors gestartet. Im folgenden Programmcode wird analog zu Abschn. 6.1.5 zunächst eine Datenstruktur MMA65XX_pins definiert, die die Pindefinitionen inklusive des SPI SlaveSelect Eingangs enthält. Im Fall des MMA65XX mag dies übertrieben aussehen, da der Chip keine zusätzlichen Pins enthält, im Sinne der Einheitlichkeit macht eine solche Vorgehensweise jedoch Sinn.

```
typedef struct {
    tspiHandle MMA65XXspi;
} MMA65XX_pins;
```

Bei allen folgenden Bausteinen mit SPI Schnittstelle wird dies analog gehandhabt, ohne dass darauf näher eingegangen wird.

```
uint8_t MMA65XX_Init(MMA65XX_pins sdevice_pins)
{
    #define NO_ERROR 0
    #define DEVINIT 0x20
    #define ERROR_MASK 0x5F
    volatile uint8_t ucAnswer[4];
```

```

    uint8_t ucReg, ucError;
//der Slave Select Anschluss des Bausteins wird initialisiert
//(Ausgang auf High gesetzt)
    SPI_MasterInit_CS(sdevice_pins.MMA65XXspi);
//die Error-flags werden zurückgesetzt
    MMA65XX_Read_Reg(sdevice_pins, DEVICE_STATUS_REG, ucAnswer);
//sind die Error-flags zurückgesetzt?
    MMA65XX_Read_Reg(sdevice_pins, DEVICE_STATUS_REG, ucAnswer);
    ucError = ucAnswer[3] & ERROR_MASK;
    if(ucError) return ucError;
//Speicherung des Device Config Reg. in ucReg
    MMA65XX_Read_Reg(sdevice_pins, DEVICE_CONFIG_REG, ucAnswer);
    ucReg = ucAnswer[3];
    ucReg |= DEVINIT;
    MMA65XX_Write_Reg(sdevice_pins, DEVICE_CONFIG_REG, ucReg, ucAnswer);
    return NO_ERROR;
}

```

Die Funktion `MMA65XX_Init()` führt eine vereinfachte Initialisierung des Sensors durch. Mit dem ersten Lesen des Statusregisters werden die während des Einschaltens gesetzten Fehlerbits gelöscht. Mit dem zweiten Lesen des Registers wird geprüft, ob alle Fehlerbits gelöscht wurden, ansonsten wird die Funktion mit einem Fehlercode quittiert. Wenn keine persistenten Fehler vorhanden sind, wird das Bit `ENDINIT` gesetzt und die Funktion gibt den Wert `NO_ERROR` zurück.

Lesen eines Registers

Die Zusammensetzung eines Lesebefehls und die Antwort des Sensors, die mit dem Senden einer weiteren Nachricht folgt, ist in Tab. 6.22 dargestellt. Der Paritätsbit P hängt von der Adresse, bzw. dem Inhalt des Registers ab.

Der Inhalt eines Registers kann mit der folgenden Beispielfunktion realisiert werden. Beim Aufruf der Funktion `MMA65XX_Read_Reg()` muss die SPI-Datenstruktur des Sensors `sdevice_pins`, die Adresse des zu lesenden Registers `ucaddress` und die Adresse eines 4 Byte Vektors `ucarray2read` als Parameter übergeben werden. In der Funktion wird das Paritätsbit berechnet, das die Nachricht begleitet. In den ersten zwei Bytes des Vektors

Tab. 6.22 MMA6525 Lesebefehl eines Registers und Antwort

Tab. 6.23 MMA6525 Schreibbefehl eines Registers und Antwort

	MSB	Bit															LSB
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Befehl	P	1	0	A4	A3	A2	A1	A0	D7	D6	D5	D4	D3	D2	D1	D0	
	Adresse des Registers								Neuer Inhalt des Registers								
Antwort	0	0	1	P	1	1	1	0	D7	D6	D5	D4	D3	D2	D1	D0	
	Neuer Inhalt des Registers								Neuer Inhalt des Registers								

wird die Antwort des Sensors auf die vorige Nachricht gespeichert und in den letzten zwei der Inhalt des Registers entsprechend Tab. 6.22, oder eine Fehlermeldung.

```
void MMA65XX_Read_Reg(MMA65XX_pins sdevice_pins, uint8_t ucaddress,
                      volatile uint8_t* ucarray2read)
{
    uint8_t ucDummy = 0x00, ucI, ucMask = 0x01, ucCnt = 0x00;
    for(ucI = 0; ucI < 8; ucI++)
    {
        if(ucaddress & ucMask) ucCnt++;
        ucMask = ucMask << 1;
    }
    if(!(ucCnt % 2)) ucaddress |= 0x80;
    //die Chip Select-Leitung wird auf Low gesetzt
    SPI_Master_Start(sdevice_pins.MMA65XXspi);
    //die Adresse des Registers wird übertragen
    ucarray2read[0] = SPI_Master_Write(ucaddress);
    ucarray2read[1] = SPI_Master_Write(ucDummy);
    SPI_Master_Stop(sdevice_pins.MMA65XXspi);
    //die Antwort des Sensors auf den Lesebefehl wird ausgelesen
    SPI_Master_Start(sdevice_pins.MMA65XXspi);
    ucarray2read[2] = SPI_Master_Write(ucDummy);
    ucarray2read[3] = SPI_Master_Write(ucDummy);
    SPI_Master_Stop(sdevice_pins.MMA65XXspi);
}
```

Schreiben eines Registers

Der Schreibbefehl eines Registers und die zu erwartende Antwort ist nach [26] in Tab. 6.23 zusammengefasst. Bit 13 des Befehls codiert eine Registeroperation, während Bit 14 das Schreiben codiert. Das niederwertige Byte des Befehls bildet den neuen Inhalt des Registers ab.

Tab. 6.24 Lesebefehl der Messwerte und Antwort des Sensors

	MSB	Bit																	LSB
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
Befehl	0	AX	1	OC	0	0	0	0	0	0	0	0	1	SD	ARM	P			
Antwort	D1	D0	AX	P	S1	S0	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2			

Auslesen der Beschleunigungswerte

Die Abfrage eines Beschleunigungswertes speichert auf der steigenden Flanke des Chip Select Signals einen neuen Wert in das Messregister der entsprechenden Achse. Dieser Wert kann aber erst mit der nächsten Nachricht (Abfrage) gelesen werden. Die Abfrage besteht aus einem 2-Byte Befehl, der über SPI an den Sensor übertragen wird. Mit dem Bit 13 auf „1“ wird der Lesebefehl eines Messwertes codiert. Folgende Optionen können gewählt werden:

Bit 14 – AX – codiert die Achse: „0“ für die X-Achse, „1“ für die Y-Achse;

Bit 12 – OC – bei „0“ wird die dynamische Beschleunigung gemessen, ansonsten die statische:

Bit 2 – SD – mit „0“ werden die Werte vorzeichenbehaftet, mit „1“ vorzeichenlos ausgetragen:

Bit 1 = ARM = mit „0“ wird die ARM-Funktion deaktiviert.

Die Bits 11 und 10 S1:S0 aus der Antwort geben Folgendes an:

- „00“ – die Abfrage erfolgt in der Initialisierungsphase;
 - „01“ – die Abfrage erfolgt im normalen Betrieb;
 - „10“ – die Beschleunigung wurde im Testbetrieb gemessen;
 - „11“ – ein Fehler ist aufgetreten.

Die Bits 9:0, 15 und 14 aus der Antwort bilden den gemessenen Beschleunigungswert. Die gemessene Beschleunigung kann mit der Funktion MMA65XX_Get_Acceleration ausgelesen werden. Der Parameter `ucop_code` codiert den Lesebefehl entsprechend Tab. 6.24. An der Adresse `uiaccel_data` wird der Beschleunigungswert gespeichert. Um den dynamischen Wert der Beschleunigung in X-Richtung bei deaktivierter ARM-Funktion zu messen und vorzeichenlos zu lesen, lautet der Befehlscode 0x200C (b0010 0000 0000 1100). Mit dem Aufruf: `MMA65XX_Get_Acceleration(MMA65XX_1, 0x200C, &uiAccelData)` wird die Funktion ausgeführt und bei fehlerfreier Ausführung wird der zusammengefasste Wert der Beschleunigung in die Variable `uiAccelData` gespeichert.

```
#define ERROR_CODE 0x0C00
#define NO_ERROR 0
uint16_t uiData, uiMask = 0x01;
uint8_t ucI, ucCnt = 0x00, ucDummy = 0x00;
uint8_t ucCodeHigh, ucCodeLow, ucData;

ucCodeLow = uiop_code;
ucCodeHigh = uiop_code >> 8;
SPI_Master_Start(sdevice_pins.MMA65XXspi);
//der Lesebefehl wird übertragen
SPI_Master_Write(ucCodeHigh);
SPI_Master_Write(ucCodeLow);
SPI_Master_Stop(sdevice_pins.MMA65XXspi);
//die Antwort auf die vorige Nachricht wird gelesen
SPI_Master_Start(sdevice_pins.MMA65XXspi);
ucData = SPI_Master_Write(ucDummy);
uiData = (uint16_t) ucData << 8;
uiData |= SPI_Master_Write(ucDummy);
SPI_Master_Stop(sdevice_pins.MMA65XXspi);
if((uiData & ERROR_CODE) == ERROR_CODE) return 1; //interner Fehler
for(ucI = 0; ucI < 16; ucI++) //Paritätsprüfung
{
    if(uiData & uiMask) ucCnt++;
    uiMask = uiMask << 1;
}
if(!(ucCnt % 2)) return 2; //Paritätsfehler
uiData = uiData << 2;
uiData |= ucData >> 6;
*uiaccel_data = uiData; //vorzeichenloser 12Bit Beschleunigungswert
return NO_ERROR;
}
```

Im ersten Schritt wird über SPI der Befehlscode übertragen; im zweiten wird mit der Übertragung zweier Dummy-Bytes die Antwort des Sensors auf die vorige Anforderung gelesen und in die Variable uiData zusammengefasst. Die Funktion testet danach, ob die Bits 10 und 11 dieser Variable beide „1“ sind und wenn ja, gibt sie den Wert 1 zurück, um einen internen Fehler des Sensors zu signalisieren. Anschließend wird die Parität der Antwort geprüft. Wenn diese nicht stimmt, gibt die Funktion den Wert 2 zurück, was Paritätsfehler bedeuten soll. Wenn die Antwort fehlerfrei ist, wird der Beschleunigungswert zusammengefasst und zurückgegeben.

6.7 Näherungssensoren

Näherungssensoren sind Sensoren, die ein Objekt ohne direkten Kontakt detektieren können. Nach Messprinzip unterscheidet man Ultraschall-, kapazitive, induktive und optische Detektoren.

6.7.1 Ultraschall-Näherungssensoren

Ultraschall-Näherungssensoren detektieren berührungslos Objekte nach dem Impuls-Echo-Prinzip. Der Frequenzbereich des Ultraschalls liegt oberhalb des menschlichen Hörbereiches ($> 20\text{ kHz}$). Für das Umwandeln der elektrischen Pulse in mechanischen Pulse und umgekehrt benutzen die Sensoren Ultraschallwandler, die meist auf dem piezoelektrischen Effekt beruhen. Ein piezoelektrischer Kristall verformt sich unter dem Einfluss eines elektrischen Feldes. Umgekehrt entsteht bei einer von außen verursachten elastischen Verformung an seiner Oberfläche eine elektrische Spannung, die proportional zum Druck ist. Ein solcher Sensor strahlt eine Schallpulsfolge ab, die sich als Longitudinalwelle¹⁵ ausbreitet. Beim Auftreffen auf ein Objekt werden die Schallpulse als Echo reflektiert. Durch die Messung der Laufzeit der gesendeten Schallpulse und des empfangenen Echos kann, wenn die Ausbreitungsgeschwindigkeit des Schalls bekannt ist, der Abstand zwischen dem Sensor und dem reflektierenden Objekt berechnet werden. Die Schallgeschwindigkeit in Luft beträgt ca. 340 m/s [1] und ist von der Temperatur, der relativen Luftfeuchtigkeit und dem Druck abhängig. Die Gleichung Gl. 6.36 [27] beschreibt die Temperaturabhängigkeit der Schallgeschwindigkeit c von der Lufttemperatur t [$^{\circ}\text{C}$]

$$c = c_0 \cdot \sqrt{1 + \frac{t}{273,15}} \quad (6.36)$$

wobei c_0 die Ausbreitungsgeschwindigkeit des Schalls bei $0\text{ }^{\circ}\text{C}$ ist. Die Schallgeschwindigkeit erhöht sich mit 5 m/s bei einer Zunahme des Lufdrucks von 30 hPa [27] und gleichbleibender Lufttemperatur. Eine Änderung der relativen Luftfeuchte von 0 auf 100% führt bei einer Lufttemperatur von $20\text{ }^{\circ}\text{C}$ zu einer Erhöhung der Schallgeschwindigkeit mit ca. 1 m/s [27]. Die Schallintensität bei konstanter Frequenz nimmt mit der Entfernung exponentiell ab. Die Schalldämpfung ist direkt proportional mit dem Quadrat der Schallfrequenz. Höhere Frequenzen können deshalb nur für die Messung kürzeren Abstände eingesetzt, ermöglichen aber eine höhere Messrate.

6.7.1.1 Messprinzip

Ein Ultraschallsensor misst die Zeit Δt zwischen dem Ausstrahlen einer Schallpulsfolge und dem Empfang der reflektierten Schallwelle. Der Abstand d vom Sensor bis zum

¹⁵ Longitudinalwelle ist eine Welle bei der die Richtung der Druckveränderung dieselbe ist wie die Richtung der Wellenausbreitung [1].

Objekt wird folgendermaßen berechnet:

$$d = c \cdot \frac{\Delta t}{2} \quad (6.37)$$

Nach der Anzahl der benutzten Ultraschallwandler unterscheidet man zwischen Einkopf- und Zweikopfsysteme. Ein Einkopfsystem besitzt einen Schallwandler, der den Schall sowohl senden als auch empfangen kann. Der Öffnungswinkel der Schallkeule eines solchen Sensors liegt bei ca. 5° [19]. In der Sendephase wird der Wandler von der steuernden Elektronik zum Schwingen angeregt und erzeugt eine Schallpulsfolge mit einer Gesamtdauer, die von der Schallfrequenz abhängig ist. Der Schallwandler kann nach dem Senden nicht gleich empfangen, weil er nach der Anregung ausschwingen muss. In dieser Zwischenphase werden die von nahliegenden Objekten reflektierten Schallwellen nicht wahrgenommen. Abhängig von der Ausschwingzeit kann die Größe der Blindzone berechnet werden. Die Ausschwingzeit ist um ein Vielfaches größer als die Dauer der Pulsfolge. Nach der Ausschwingzeit wird der Schallwandler auf Empfang geschaltet und die Elektronik führt die Messungen innerhalb eines einstellbaren Zeitfensters durch. Es muss geprüft werden ob die empfangenen Schwingungen mit den gesendeten korreliert sind um Messfehler auszuschließen. Um der Schalldämpfung entgegen zu wirken und die Reichweite der Messung zu erhöhen, werden die empfangenen Signale unterschiedlich verstärkt. Die Signale die später empfangen wurden, werden stärker verstärkt. Ein solches System ist einfach und kompakt zu realisieren, kann aber nahliegende Objekte nicht detektieren. Der zeitliche Abstand zwischen zwei Messungen muss groß genug sein damit die Amplitude der bei der vorigen Messung verursachten Echos abgeklungen ist.

Ein Zweikopfsystem besitzt einen Schallwandler für das Senden und einen zweiten für den Empfang. Ein solcher Sensor kann gleich nach der Sendephase ankommende Signale empfangen und auswerten. Die Wartezeit wegen der Ausschwingzeit und die dadurch entstandene Blindzone entfallen. Eine Blindzone entsteht auch bei diesem System wegen der Begrenzung des Öffnungswinkels der Schallkeule α , ist aber wesentlich kleiner als beim Einkopfsystem. Der kleinste Abstand d zu einem Objekt, der mit diesem System erfasst werden kann, ist (siehe Abb. 6.31):

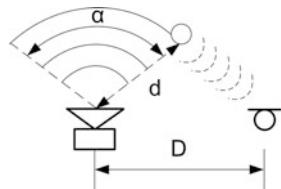
$$d = \frac{D/2}{\sin \frac{\alpha}{2}} \quad (6.38)$$

wobei D der Abstand zwischen zwei Schallwandlern ist. Dieses System ermöglicht wegen des größeren Öffnungswinkels die Detektion mehreren Objekten.

Die Amplitude der reflektierten Schallpulse ist von der Entfernung zum detektierten Objekt, von der Größe und der Oberfläche des Objektes abhängig. Ein großes Objekt kann die Schallwellen besser als ein kleines reflektieren, eine glatte Oberfläche besser als eine raue. Es gibt Materialien, die die Schallwelle gut reflektieren, andere, die sie gut absorbieren.

Ultraschallsensoren werden für Abstandsmessung, für Muster- und Objekterkennung, als Bewegungsmelder oder Näherungssensoren eingesetzt. Wenn mehrere Ultraschallsen-

Abb. 6.31 Ultraschall-Zwei-kopfsystem



soren in einem Raum aktiv sind, können sie im Synchron- oder Multiplexbetrieb arbeiten. Im Synchronbetrieb senden alle Sensoren gleichzeitig ihre Schallpulse, vorausgesetzt sie stören sich nicht gegenseitig. Im Multiplexbetrieb werden die Sensoren nacheinander aktiviert und ausgewertet.

6.7.1.2 SRF08 – Ultraschall Messmodul

SRF08 ist ein diskret aufgebauter, Mikrocontroller gesteuerter Ultraschallsensor. Er nutzt Schallwandler mit einem Öffnungswinkel von ca. 60° und erzeugt für die Detektion, bzw. Messung Schallpulse mit einer Frequenz von 40 kHz (Abb. 6.32a). Der Abstand zu bis 6 m entfernte Objekte wird mit einer Auflösung von 1 cm gemessen und die Messergebnisse können in Zentimeter, Zoll oder Mikrosekunden umgerechnet werden. Konfiguration des Sensors und Auslesen der Messwerte erfolgen über den I²C-Bus.

Aufbau

Ein Blockschaltbild des Ultraschallsensors SRF08 ist in Abb. 6.33 dargestellt. Der Mikrocontroller regt den Schallsender über einen Pegelwandler und einen Verstärker mit einer elektrischen Pulsofolge zum Schwingen an (siehe Abb. 6.32a). Die empfangenen Schallpulse werden mit dem Schallempfänger in elektrischen Pulse umgewandelt (Abb. 6.32b), die zur Verbesserung der Reichweite vor der Digitalisierung verstärkt werden können.

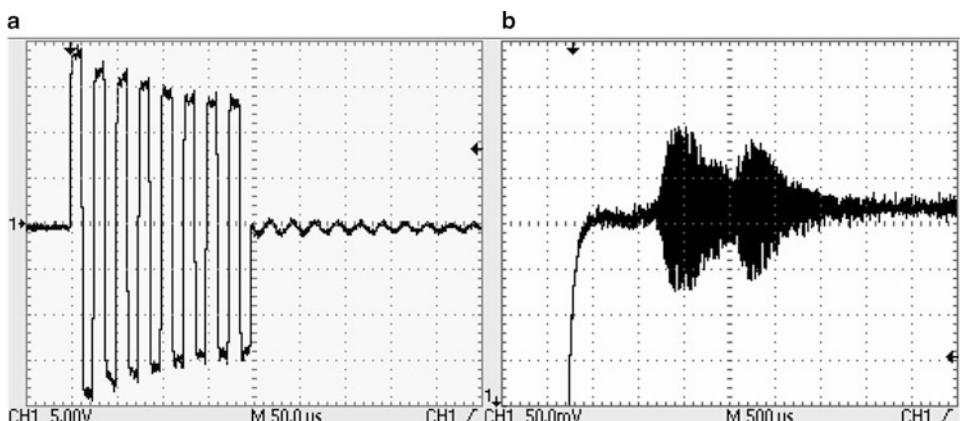


Abb. 6.32 SRF08 gesendete Pulse (a) und empfangene Pulse (b)

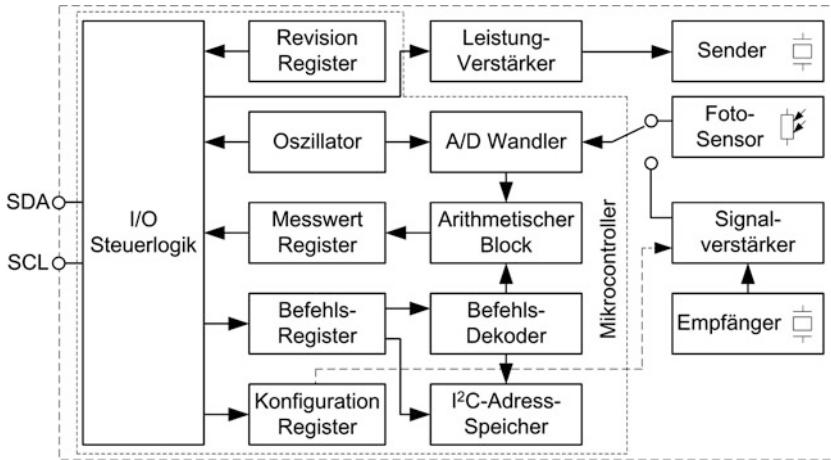


Abb. 6.33 SRF08 Blockschaltbild

Wegen des großen Öffnungswinkels und der großen Reichweite kann der Sensor mehrere Objekte detektieren. Die gemessenen Laufzeiten können vor dem Speichern in den Messwertregistern mit der Gl. 6.37 in Längen umgerechnet werden. Der Sensor besitzt auch einen Fotowiderstand, mit dem die Lichthelligkeit geschätzt werden kann. Die Abstandsmessungen werden über das Reichweiten- und Verstärkungs-Register konfiguriert. Mit dem Speichern eines Wertes aus dem Bereich [0, 31] in das Verstärkungs-Register (Adresse 0x01) wird das analoge Signal vom Schallempfänger um einen Faktor aus dem Bereich [94, 1025] verstärkt, um optimale Messergebnisse zu erzielen. Die Echos werden innerhalb eines einstellbaren Zeitfensters, das einem Vielfachen von $256 \mu\text{s}$ entspricht, ausgewertet. Mit dem Speichern von Werten aus dem Bereich [0, 140] in das Reichweiten-Register (0x02) wird die Reichweite auf Werte zwischen 43 mm und ca. 6 m in einem Raster von 43 mm begrenzt. Die im Befehls-Register 0x00 gespeicherten Codes führen zum Start eines neuen Messvorgangs, bzw. zum Ändern der I²C-Adresse.

Serielle Kommunikation

Die Konfiguration des Sensors, der Start neuer Messungen, das Auslesen der Messwerte, und die Änderung der Adresse erfolgen über den I²C-Bus. Der Sensor ist als Slave konfiguriert und antwortet nach der Übertragung auf die I²C-7 Bit-Adresse 0x70, die vom Benutzer auf eine von weiteren 15 Adressen aus dem Bereich [0x71, 0x7F] eingestellt werden kann, damit über einen Bus ein Multiplexbetrieb möglich ist. Alle an einem Bus angeschlossenen Sensoren können über die Adresse 0x00 gleichzeitig angesprochen werden. Damit ist es beispielsweise möglich, alle Messungen zeitgleich zu starten. Das Schreiben eines Wertes in ein Register beginnt mit einer START-Sequenz und findet in einer einzigen I²C-Botschaft statt. Der Master überträgt die Sensoradresse gefolgt von der Registeradresse und dem zu speichernden Wert. Jedes Byte muss vom Slave mit ACK

quittiert werden, der Master beendet die Übertragung mit einer STOP-Sequenz. Um die Adresse des Slaves zu ändern speichert der Master die Bytes 0xA0, 0xAA und 0xA5 gefolgt von der neuen Adresse in das Befehlsregister. Zwischen zwei Übertragungen ist eine Pause von mindestens 50 ms einzuhalten:

```
void SRF08_Set_NewAddress(uint8_t ucold_address, uint8_t ucnew_address)
{
    #define COMMAND_REG      0
    uint8_t ucTime, ucAddress_Sequence[4] = {0xA0, 0xAA, 0xA5, 0x00}, ucI;
    ucAddress_Sequence[3] = ucnew_address;
    for(ucI = 0; ucI < 4; ucI++)
    {
        SRF08_Write_ByteReg(ucold_address, COMMAND_REG,
                            ucAddress_Sequence[ucI]);
        ucTime = 0;
        while(ucTime < 6)
        {
            //nach jedem gesendeten Byte muss eine 50 ms Pause
            //eingehalten werden
            if(Timer1_get_10msState() == TIMER_TRIGGERED)
            {
                ucTime++;
            }
        }
    }
}
```

Die Funktion `Timer1_get_10msState()` gibt nach jeden 10 ms den Wert `TIMER_TRIGGERED` zurück. Beim Aufrufen der Funktion muss die alte und die neue Adresse im Write-Modus eingegeben werden. Angenommen, die aktuelle 7-Bit-Adresse lautet 0x70 und die zukünftige 0x74, dann sieht der Aufruf der Funktion wie folgt aus:

```
SRF08_Set_NewAddress(0xE0, 0xE8);
```

Mit dem Lesen eines Registers wird der interne Adresszähler inkrementiert damit man in einer I²C-Übertragung mehrere Register lesen kann.

Messwerterfassung

Der Sensor SRF08 arbeitet im Einzelmessmodus; jede Messung wird durch das Speichern eines entsprechenden Befehlscodes in das Befehlsregister gestartet und innerhalb des eingestellten Zeitfensters durchgeführt. Das Auslesen der Messwerte kann am Ende dieses Zeitfensters geschehen. Ein Timerinterrupt, der entsprechend konfiguriert ist und einmal mit der Messung gestartet ist, kann ohne den Mikrocontroller zu belasten den Zeitpunkt für das Auslesen angeben. Alternativ kann das `SOFTWARE_REVISION` Register (Adresse 0x00) ausgelesen werden. Dieses Register speichert während der Messung den Wert 0xFF,

ansonsten einen Wert der, der Softwarerevision entspricht. Diese Methode beschäftigt aber zusätzlich den Mikrocontroller.

Abstand-Messmodus

Im Abstand-Messmodus wird entweder die Laufzeit (Befehlscode 0x52) oder der Abstand in Zoll (0x50) oder in Zentimeter (0x51) gemessen. Die Messungen beginnen in diesem Modus mit dem kleinsten Verstärkungsfaktor aus dem genannten Bereich. Dieser wird jede 70 µs erhöht, bis er den eingestellten Wert erreicht. Durch Testen können diese Parameter abhängig von den verfolgten Zielen optimiert werden. In einem Messvorgang können bis zu 17 Echosignale ausgewertet werden. Die 16 Bit Ergebnisse werden in steigender Reihenfolge in die Messwertregister im Adressbereich [0x02, 0x23] gespeichert. Die Messung des nächsten Objektes wird an den Adressen 0x02 und 0x03 im Big-Endian-Format gespeichert. Die Messwerte können einzeln oder als Ganzen wie folgt gelesen werden. Als Parameter werden die Moduladresse im Write-Modus `ucdevice_address` und die Vektoradresse `uidata_word` übergeben, an der die gemessenen Werte gespeichert werden.

```
uint8_t SRF08_Read_DataWordReg(uint8_t ucdevice_address,
                                uint16_t* uidata_word)
{
    uint8_t ucDeviceAddress, ucDataLSByte, ucDataMSByte, ucI;
    TWI_Master_Start(); //Start
    if((TWI_STATUS_REGISTER) != TWI_START) return TWI_ERROR;
    TWI_Master_Transmit(ucdevice_address); //Modul Adresse senden
    if((TWI_STATUS_REGISTER) != TWI_MT_SLA_ACK) return TWI_ERROR;
    //die Anfangsadresse des Registerbereiches wird gesendet
    TWI_Master_Transmit(DATA1_H_ADDR);
    if((TWI_STATUS_REGISTER) != TWI_MT_DATA_ACK) return TWI_ERROR;
    TWI_Master_Start(); //Restart
    if((TWI_STATUS_REGISTER) != TWI_RESTART) return TWI_ERROR;
    ucDeviceAddress = ucdevice_address | TWI_READ; //Read-Modus
    //Moduladresse im Read-Modus senden
    TWI_Master_Transmit(ucDeviceAddress);
    if((TWI_STATUS_REGISTER) != TWI_MR_SLA_ACK) return TWI_ERROR;
    //Inhalt der Register wird eingelesen
    for(ucI = 0; ucI < 17; ucI++)
    {
        ucDataMSByte = TWI_Master_Read_Ack();
        if((TWI_STATUS_REGISTER) != TWI_MR_DATA_ACK) return TWI_ERROR;
        if(ucI < 16)
        {
            ucDataLSByte = TWI_Master_Read_Ack();
            if((TWI_STATUS_REGISTER) != TWI_MR_DATA_ACK) return TWI_ERROR;
            uidata_word[ucI] = (ucDataMSByte << 8) + ucDataLSByte;
        }
    }
}
```

```

    }
    else
    {
        ucDataMSByte = TWI_Master_Read_NAck();
        uidata_word[ucI] = (ucDataMSByte << 8) + ucDataLSByte;
        if((TWI_STATUS_REGISTER) != TWI_MR_DATA_NACK) return TWI_ERROR;
    }
}
TWI_Master_Stop(); //Stop
return TWI_OK;
}

```

Artificial Neural network¹⁶-Messmodus

Die Messungen im Artificial Neural Network-Modus werden mit einem der Befehlscodes 0x53, 0x54 oder 0x55 gestartet. Die Messungen finden unabhängig von den Einstellungen mit dem höchsten Verstärkungsfaktor statt. Das gesamte Mess-Zeitfenster wird in $32 \times 2048 \mu\text{s}$ großen Zeitintervalle unterteilt, die je einem Register ab der Adresse 0x04 zugeordnet sind. Wenn in einem Zeitintervall kein reflektiertes Signal empfangen oder erkannt wurde, so wird in das entsprechende Register eine 0 gespeichert, ansonsten ein Wert ungleich 0. Es entsteht dadurch ein Muster, das zum Teil den reellen Abstand zu den reflektierenden Objekten aus der Nähe abbildet. Eine genauere Abbildung kann mit Hilfe weiteren Ultraschallsensoren erreicht werden. Durch die Auswertung dieses Musters mit neuronalen Netzen können Objektumrisse, bzw. Bewegung von Objekten erkannt werden. Mit dem Befehl 0x53 wird gleichzeitig mit den oben erwähnten Messungen der Abstand zum nächsten Objekt in Zoll, mit 0x54 der Abstand in Zentimeter und mit 0x55 die Laufzeit in Mikrosekunden gemessen und das Ergebnis in die Register mit den Adressen 0x02 und 0x03 gespeichert. Für das Auslesen der Ergebnisse kann die oben aufgelistete Funktion verwendet werden.

Lichthelligkeitsmessung

Mit jeder Abstandsmessung wird von dem Sensor auch die Lichthelligkeit gemessen und das Ergebnis in das Register mit der Adresse 0x01 gespeichert. Dieses Register kann wie alle weiteren 8 Bit Register mit folgender Funktion ausgelesen werden:

```

uint8_t SRF08_Read_BytReg(uint8_t ucdevice_address,
                           uint8_t ucreg_address,
                           uint8_t* ucdata_byte)
{
    uint8_t ucDeviceAddress;
    TWI_Master_Start(); //Start
    if((TWI_STATUS_REGISTER) != TWI_START) return TWI_ERROR;
    TWI_Master_Transmit(ucdevice_address); //Modul Adresse im Write-

```

¹⁶ Artificial Neural Network – künstliche neuronale Netzwerke.

```

        //Modus senden
    if((TWI_STATUS_REGISTER) != TWI_MT_SLA_ACK) return TWI_ERROR;
    //die Adresse des gewünschten Registers wird gesendet
    TWI_Master_Transmit(ucreg_address);
    if((TWI_STATUS_REGISTER) != TWI_MT_DATA_ACK) return TWI_ERROR;
    TWI_Master_Start(); //Restart
    if((TWI_STATUS_REGISTER) != TWI_RESTART) return TWI_ERROR;
    ucDeviceAddress = ucdevice_address | TWI_READ; //Read-Modus
    TWI_Master_Transmit(ucDeviceAddress); //Modul Adresse im Read
                                            //Modus senden
    if((TWI_STATUS_REGISTER) != TWI_MR_SLA_ACK) return TWI_ERROR;
    *ucdata_byte = TWI_Master_Read_NAck(); //das adressierte Register
                                            //wird eingelesen
    if((TWI_STATUS_REGISTER) != TWI_MR_DATA_NACK) return TWI_ERROR;
    TWI_Master_Stop(); //Stop
    return TWI_OK;
}

```

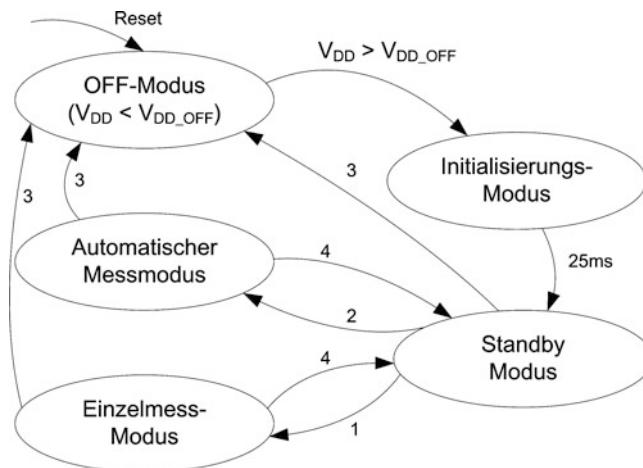
6.7.2 SI114x – optischer Näherungssensor

Die Bausteine SI1141, SI1142 und SI1143 besitzen einen Fotosensor für sichtbares und zwei für infrarotes Licht [28–30]. Sie ermöglichen eine Näherungsmessung (PS = Proximity Sensing) zu einem Objekt, das bis zu 50 cm vom Sensor entfernt ist und die direkte Messung des Umgebungslichtes (ALS = Ambient Light Sensing). Ein Baustein aus dieser Familie steuert abhängig vom Typ bis zu drei infrarote Leuchtdioden mit Spannungspulsen und misst die reflektierten Strahlen um ein Objekt detektieren zu können. Die Beleuchtungsstärke des weißen und infraroten Lichtes können gemessen werden, um den Einfluss des Umgebungslichtes auf die Näherungsmessung zu minimieren.

6.7.2.1 SI114x Arbeitsmodi

Die Sensoren aus der Reihe SI114x besitzen keinen Reset-Pin. Ein interner Komparator sorgt dafür, dass bei Versorgungsspannungen unter dem Pegel V_{DD_OFF} (ca. 1 V) der gesamte Baustein abgeschaltet ist. Sobald diese Grenze überschritten wird, findet die Hardware-Initialisierung statt, die nach ca. 25 ms abgeschlossen ist. Der Sensor befindet sich dann im Standby-Modus. In diesem Modus ist die Kommunikationsschnittstelle aktiv und Befehle können empfangen und ausgeführt werden. Ein RESET-Befehl (Code 0x01) ermöglicht eine vollständige Reinitialisierung des Bausteins, so wie in Abb. 6.34 dargestellt.

Um die Initialisierung zu vervollständigen muss der steuernde Master zuerst in das Register *HW-KEY* (Adresse 0x07) den Wert 0x17 speichern. Der Standby-Modus wird verlassen um Messungen im Einzelmess-Modus, oder im automatischen Messbetrieb durchzuführen. Nachdem der gewählte Messungssatz beendet wurde und die Messwerte ge-



- 1) $(MEAS_RATE = 0) \& (PS_FORCE | ALS_FORCE | PSALS_FORCE)$
- 2) $(MEAS_RATE = 1) \& (PS_AUTO | ALS_AUTO | PSALS_AUTO)$
- 3) RESET Befehl
- 4) Ende einer Messung

Abb. 6.34 SI114x Zustandsübergangsdiagramm

speichert wurden, kehrt der Sensor in den Standby-Modus zurück um Strom zu sparen. Über das Register *MEAS_RATE* (0x08) kann einer der Messmodi gewählt werden. Nach der Initialisierungsphase wird dieses Register auf „0“ gesetzt und somit der Einzelmess-Modus gewählt. In diesem Modus können einzelne Näherungsmessungen mit dem Befehl *PS_FORCE* (0x05), Umgebungslicht-Messungen mit dem Befehl *ALS_FORCE* (0x06) oder eine komplette Messung mit dem Befehl *PSALS_FORCE* (0x07) gestartet werden.

Im automatischen Messmodus befindet sich der Sensor jeweils nachdem ein von Null verschiedener komprimierter¹⁷ Wert in das 8-Bit Register *MEAS_RATE* (Adresse 0x08) gespeichert und eine der Messarten gestartet wurden. Der Start erfolgt für die Näherungsmessungen mit dem Befehl *PS_AUTO* (Code 0x0D), für die Messung des Umgebungslichtes mit dem Befehl *ALS_AUTO* (0x0E). Alle Messungen finden automatisch nach dem Befehl *PSALS_AUTO* (0x0F) statt. Eine automatische Messungsart kann gestoppt werden ohne den automatischen Messmodus zu verlassen mit einem der Befehle: *PS_PAUSE* (0x09), *ALS_PAUSE* (0x0A) oder *PSALS_PAUSE* (0x0B).

Mit dem im Register *MEAS_RATE* gespeicherten Wert wird die Abtastfrequenz eingestellt, mit der der Sensor aus dem Standby aufwacht, um bei Bedarf Messungen zu starten. Der unkomprimierte Wert k multipliziert mit $31,25 \mu\text{s}$ ($1/32 \text{ kHz}$) gibt die Zeit T_{mess} zwischen zwei möglichen Messungen an. Die Abtastperiode für die Näherungsmessung $n * T_{\text{mess}}$ und für die Messung des Umgebungslichtes $m * T_{\text{mess}}$ sind ein Vielfaches der Grundabtastperiode T_{mess} , wobei n und m 16-Bit-Ganzahlen sind. Wenn n oder m

¹⁷ Die Kompression wird später erläutert.

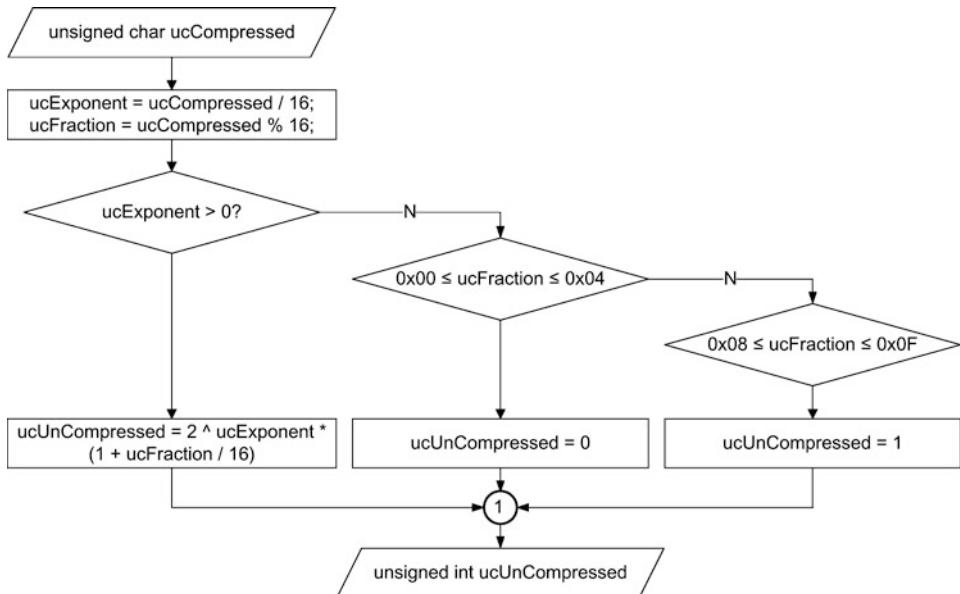


Abb. 6.35 SI114x – Berechnung der unkomprimierten Werte

gleich Null ist, wird die Messung der entsprechenden Größe im automatischen Messmodus nicht durchgeführt. Weil die Register *PS_RATE* (0xA) für das Speichern von n und *ALS_RATE* (0x09) für das Speichern von m 8-Bit groß sind, müssen die Werte komprimiert werden. Die Abb. 6.35 stellt den Rechenweg dar, um aus einem komprimierten Wert (gespeichert in der Variable *ucCompressed*) den unkomprimierten berechnen zu können.

Die folgende Funktion berechnet und gibt den komprimierten Wert einer positiven, 16-Bit großen Zahl zurück, die als Parameter beim Aufrufen der Funktion übergeben wurde.

```

uint8_t SI114x_Set_CompressedValue(unsigned int uiuncompressed)
{
    uint8_t ucExponent = 0xFF, ucFraction, ucCompressed;
    unsigned int uiUnCompressed = uiuncompressed;
    //dem unkomprimierten Wert 0 wird 0x04 und dem 1 wird 0x08 zugewiesen
    if(!uiuncompressed)    ucCompressed = 0x04;
    else if (uiuncompressed == 1)    ucCompressed = 0x08;
    else
    {   //es wird nach der führenden 1 in der binären Darstellung der
        //Zahl gesucht und das höherwertige Nibble der komprimierten
        //Zahl gebildet
        while(uiUnCompressed)
        {
            uiUnCompressed = uiUnCompressed >> 1;
            ucExponent++;
        }
    }
}
  
```

```

//das niedrigerwertige Nibble der komprimierten Zahl wird berechnet
if(ucExponent >= 4)    ucFraction = uiuncompressed >>
    (ucExponent - 4);
else    ucFraction = uiuncompressed << (4 - ucExponent);
ucFraction &= 0x0F;
//die komprimierte Zahl wird berechnet
ucCompressed = ucExponent * 16 + ucFraction;
}
return ucCompressed;
}

```

Diese Art der Datenkompression spart Speicherplatz, bildet aber keine bijektive Zuordnung zwischen den Mengen der komprimierten und der unkomprimierten Zahlen ab. Zum Beispiel, lautet der komprimierte Wert aller binären Zahlen der Form *1111 1xxx xxxx xxxx* in hexadezimaler Form 0xFF. Durch die Dekodierung dieses Wertes wird lediglich die Zahl 0xF800 wiederhergestellt und somit einen maximalen Fehler von ca. 3,2 % verursacht.

6.7.2.2 SI114x Aufbau

Das Blockschaltbild eines Proximity-Sensors vom Typ SI1143 ist in der Abb. 6.36 dargestellt. Der Baustein besitzt einen Sensorblock, der aus einem Weißlichtsensor, zwei

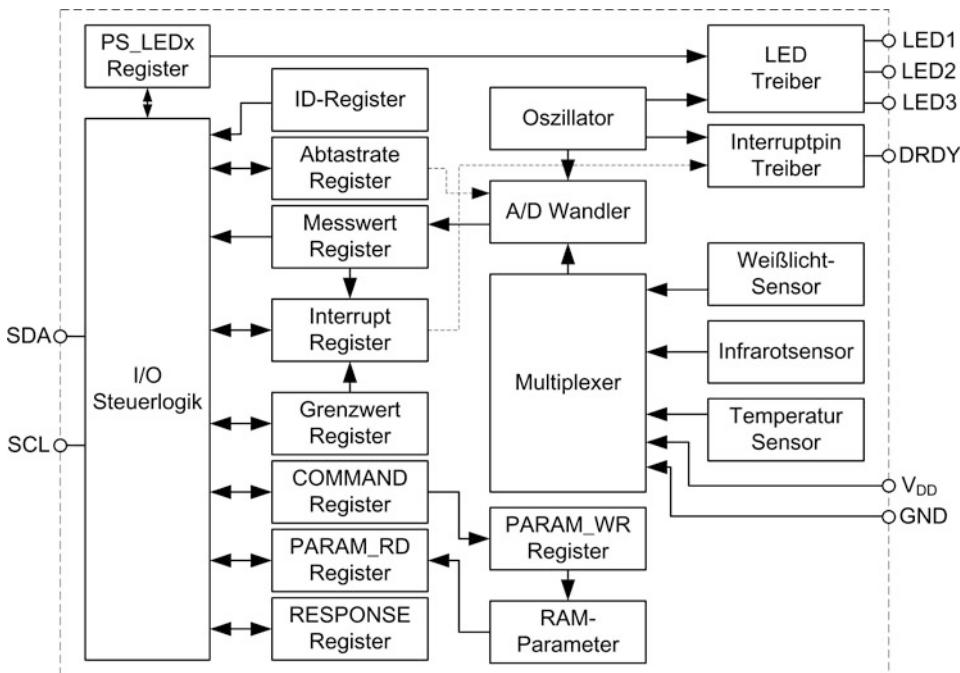


Abb. 6.36 SI1143 Blockschaltbild

Infrarotsensoren und einem Temperatursensor besteht. Mit den Lichtsensoren kann die Beleuchtungsstärke des weißen und des infraroten Lichtes gemessen werden. Der Infrarotsensor wird zusätzlich bei der Näherungsmessung, bzw. beim Detektieren von Objekten verwendet. Der Temperatursensor wird für die Messung der Temperaturdifferenz zwischen zwei Messungen empfohlen. Die von den Sensoren gelieferten analogen Spannungen und zusätzlich die Versorgung- und die Massespannung können mit einem 17-Bit A/D-Wandler digitalisiert werden.

Für die Näherungsmessung besitzt der Baustein SI1143 einen LED-Treiber, mit dem bis zu drei infrarote Sendedioden angesteuert werden können. Die geschieht durch Spannungspulse, deren Dauer und Amplitude einstellbar sind. Für jede Sendediode kann jeweils eine Näherungsmessung ausgewählt und ausgeführt werden. Durch diese dreifache Messung und Auswertung der Messwerte kann sogar die räumliche Position eines detektierten Objektes gegenüber dem Baustein festgestellt werden.

Der Baustein besitzt einen umfangreichen Registerblock mit 8-Bit großen Registern, die einzeln adressierbar sind. Das Inkrementieren des internen Register-Adresszählers wird abgeschaltet, wenn das Bit 6 einer Registeradresse bei einem Schreib- oder Lesezugriff gesetzt ist. So ist es möglich, den Inhalt eines Registers in einer einzigen I²C-Übertragung (zwischen einer START- und STOP-Sequenz) wiederholt zu lesen oder zu schreiben ohne die Adressierung des Bausteins und des Registers zu wiederholen. Die 16-Bit-Messergebnisse werden in jeweils zwei Register im Little-Endian-Format gespeichert, wie in der Tab. 6.25 aufgelistet. Weiterhin besitzt der Sensor eine Registergruppe für Mess- und Interrupt-Einstellungen, eine mit Grenzwerten für die Interrupt-Auslösung, Identifikations- und Statusregister. Alle Register werden über die serielle Schnittstelle direkt angesprochen. Einstellungen der Messbereiche, der Verstärkungsfaktoren, des DC-Offsets, der Freigabe der Messungsart oder der Grenzwert-Hysterese können im Bereich der flüchtigen Parameter gespeichert werden. Der Zugriff auf den Parameter-Bereich erfolgt durch die Übertragung von Befehlen. Die codierten Befehle, die von dem Baustein akzeptiert werden, sind in [28] aufgelistet und werden zwecks Ausführung in das Register *COMMAND* (0x18) geladen. Um einen Parameter über die serielle Schnittstelle zu lesen, wird der Befehl *PARAM_QUERY* verwendet, dessen Code *100aaaaa* ist. Die fünf niederwertigen Bits des Codes bilden die Adresse des auszulesenden Parameters. Nach dem Ausführen des Befehls steht der gewünschte Parameter im Register *PARAM_RD* (0x2E) zum Ablesen bereit. Um einen Parameter zu ändern, wird zuerst der neue Wert in das Register *PARAM_WR* (0x17) geladen und danach der Befehlscode *PARAM_SET* (*101aaaaa*) in das Register *COMMAND*. Ein erfolgreich ausgeführter Befehl inkrementiert einen Zähler im Register *RESPONSE* (0x20), bzw. setzt den Zähler auf 0x01 wenn das Register vorher mit dem Befehl *NOP* (0x00) auf „0“ gesetzt wurde. Wegen der internen Vorgänge sind die Befehle nach dem Speichern der Befehlscodes im Register *COMMAND* nicht sofort wirksam. Die Ausführungszeit der Befehle ist unterschiedlich, deshalb soll das Register *RESPONSE* wie im folgenden Programmcode wiederholt gelesen werden, aber nicht länger als 25 ms. Bei Zeitüberschreitung sollte das ganze Verfahren wiederholt werden.

Tab. 6.25 SI1143 Messwert Register

Name	Adresse	Beschreibung
ALS_VIS_DATA0	0x22	Speicherregister für das Ergebnis einer Messung des sichtbaren Lichtes
ALS_VIS_DATA1	0x23	
ALS_IR_DATA0	0x24	Speicherregister für das Ergebnis einer Messung des infraroten Lichtes
ALS_IR_DATA1	0x25	
PS1_DATA0	0x26	Speicherregister für eine Näherungsmessung mit der 1. Sendediode
PS1_DATA1	0x27	
PS2_DATA0	0x28	Speicherregister für eine Näherungsmessung mit der 2. Sendediode
PS2_DATA1	0x29	
PS3_DATA0	0x2A	Speicherregister für eine Näherungsmessung mit der 3. Sendediode
PS3_DATA1	0x2B	
AUX_DATA0	0x2C	Speicherregister für die Messung der Temperatur oder der Versorgungs- oder Massespannung
AUX_DATA1	0x2D	

```

#define COMMAND_REG    0x18 //Adresse des Registers COMMAND
#define RESPONSE_REG   0x20 //Adresse des Registers RESPONSE
#define NOP            0x00 //NOP-Befehl; setzt das Register RESPONSE
                      //auf 0x00
uint8_t ucResponse; //global definierte Variable im Modul SI114x

uint8_t SI114x_Write_CommandReg(uint8_t ucdata_byte)
{
    uint8_t ucErrorFlag = 0, ucRepeatCnt = 0;
    ucResponse = 0x01;
    //das Response Register wird zurückgesetzt
    SI114x_Write_ByteReg(COMMAND_REG, NOP);
    //das Response Register wird gelesen
    SI114x_Read_ByteReg(RESPONSE_REG, &ucResponse);
    //wenn das Zurücksetzen nicht funktioniert hat,
    //wird das Verfahren wiederholt
    while (ucResponse != 0) SI114x_Read_ByteReg(RESPONSE_REG, &ucResponse);
    //in das Command Register wird ucdatabyte geschrieben; ein Befehl soll
    //ausgeführt werden
    SI114x_Write_ByteReg(COMMAND_REG, ucdata_byte);
    //die Antwort des Sensors auf den Befehl wird in die Variable
    //ucResponse gespeichert
    SI114x_Read_ByteReg(RESPONSE_REG, &ucResponse);
    while(!ucResponse && !ucErrorFlag)
    { //die Abfrage wird wiederholt bis wann der Sensor auf den
      //Befehl antwortet oder 25 ms verstrichen sind
      SI114x_Read_ByteReg(RESPONSE_REG, &ucResponse);
      ucRepeatCnt++;
}

```

```

    if(ucRepeatCnt == 100)
    {
        ucErrorFlag = 1;
        return TWI_ERROR;
    }
}
return ucResponse;
}

```

6.7.2.3 Serielle Kommunikation

Die Proximity-Sensoren der Reihe SI114x sind als I²C-Slaves gebaut und können von einem Master über die 7-Bit Adresse 0x5A angesprochen werden. Sie reagieren auch auf die globale Adresse 0x00 und auf den globalen I²C Reset-Befehl 0x06, erlauben aber nicht die 10-Bit-Adressierung. Der globale Reset-Befehl führt zur Initialisierung des Bausteins, ähnlich wie nach dem *RESET*-Befehl. In Abb. 6.37 ist der zeitliche Verlauf dieses Befehls dargestellt. Die serielle Schnittstelle ist auch im Standby-Modus aktiv und erlaubt eine maximale Übertragungsrate von 3,4 MBit/s. Die I²C-Adresse dieser Bausteine kann softwaremäßig geändert werden. Zuerst wird die neue Adresse als *I²C-ADDR* Parameter (Adresse 0x00) gespeichert. Nach dem Ausführen des Befehls *BUSADDR* (Code 0x02) wird die neue Adresse wirksam und bleibt bis zur nächsten Initialisierung des Bausteins erhalten.

Mit der im Folgenden aufgelistete Funktion `SI114x_Set_NewAddress()` wird der übergebene Aufrufparameter `ucnew_address` als neue Slaveadresse gespeichert. Die Funktion gibt den Wert 0x00 bei erfolgreicher Änderung zurück, ansonsten 0x01.

```

//Registeradressen
#define PARAM_WR_REG      0x17
#define COMMAND_REG       0x18
//Befehle
#define PARAM_SET          0xA0
#define BUSADDR            0x02
//Parameteradressen

```

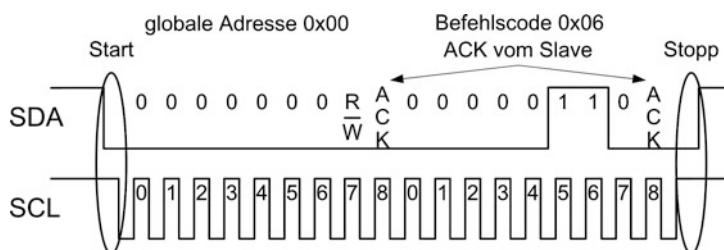


Abb. 6.37 I²C globaler Reset-Befehl

```

#define I2C_ADDR_PARAM      0x00
//Variablen
uint8_t ucSI114xAddress = 0x5A;
//Funktionsbeginn
uint8_t SI114x_Set_NewAddress(uint8_t ucnew_address)
{
    //in das PARAM_WR Register wird ucnew_address gespeichert das an der
    //Parameter Adresse I2C_ADDR übertragen werden soll
    SI114x_Write_ByteReg(PARAM_WR_REG, ucnew_address);
    //in das Command Register wird die Adresse des Parameters im Write-
    //Modus geladen und dadurch das Übertragen der neuen Adresse in den RAM
    //an der Adresse 0x00 ausgeführt
    if(!SI114x_Write_CommandReg(PARAM_SET | I2C_ADDR_PARAM))
        return TWI_ERROR;
    //mit dem Befehl BUSADDR wird die neue Adresse wirksam
    SI114x_Write_ByteReg(COMMAND_REG, BUSADDR);
    ucSI114xAddress = ucnew_address;
    return TWI_OK;
}

```

Um den Inhalt eines Registers zu ändern, muss der Master die Kommunikation mit dem Slave mit einer START-Sequenz initiieren, den Slave über die aktuelle Adresse im Write-Modus ansprechen, die Adresse des gewünschten Registers senden und schließlich den neuen Wert übertragen. Wenn der Slave jedes empfangene Byte mit ACK quittiert hat, war der Schreibvorgang erfolgreich und der Master beendet die Kommunikation mit einer STOP-Sequenz.

6.7.2.4 Näherungsmessen mit dem SI114x

Ein Proximity-Sensor der Reihe SI114x ermöglicht die Messung der Näherung und/oder des Umgebungslichtes. Die Näherungsmessungen finden auf bis zu drei Messkanäle *PS1*, *PS2* und *PS3* statt. Bei der gewählten Umgebungslicht-Messung können sowohl die Lichtstärken des sichtbaren und des infraroten Lichtes, als auch die Temperatur, die Versorgungsspannung oder die Massespannung gemessen werden. Jeder Messkanal kann an- und abgewählt werden, was zu einer hohen Flexibilität des Messvorgangs führt. Die Anzahl der gewählten Messungen beeinflusst die gesamte Messdauer und den gesamten Energieverbrauch des Bausteins. Die angewählten Messungen werden hintereinander durchgeführt und die Messergebnisse eines jeden Messkanals werden in ein Registerpaar (siehe Tab. 6.25) gespeichert.

Die Bits 3:0 des Registers *RESPONSE* bilden einen Zähler, der mit jedem erfolgreich ausgeführten Befehl inkrementiert wird. Dieses Register speichert einen Fehlercode, wenn bei einer Messung ein Überlauf stattgefunden hat. In einem solchen Fall müssen die Einstellungen an den Umgebungsbedingungen angepasst und die Messungen wiederholt werden. Einige Ergebnisse entstehen durch die Subtraktion zweier Messwerte, was zu ei-

nem negativen Wert führen kann, der fälschlicherweise als Überlauf signalisiert würde. Um das zu vermeiden, werden alle Messwerte mit einem rechnerischen Offset versehen. Dieser einstellbare, globale Offset ist im Register *ADC_OFFSET* (0x1A) in komprimierter Form gespeichert und muss bei der Auswertung der Ergebnisse berücksichtigt werden.

Die Näherungsmessung (Detektion eines Objektes) kann auch für die Gestenerkennung benutzt werden [30]. Die Bausteine besitzen 1..3 Messkanäle *PS1*, *PS2* und *PS3*. Über diese können die angeschlossenen Infrarotdioden mit Spannungspulsen angesteuert werden, die Lichtstärke der reflektierten Strahlen gemessen und die Messwerte in die entsprechenden Register gespeichert werden. Jeder Baustein besitzt einen kleinen und einen großen Infrarotsensor. Der kleine kann eingesetzt werden, wenn die Außenbeleuchtung hoch ist (in der Regel bei Sonnenlicht), der große wird unter normalen Lichtverhältnissen eingesetzt um eine höhere Empfindlichkeit zu erreichen. Der Einfluss des Umgebungslichtes wird durch eine zweifache Messung kompensiert. Zuerst wird das infrarote Umgebungslicht mit abgeschalteter Sendediode gemessen und dann wird die Näherungsmessung durchgeführt. Die Differenz der zwei Messungen wird gespeichert. Um genauere Messungen zu erzielen, die Entfernung zu den detektierten Objekten zu erhöhen und um den Einfluss des Umgebungslichtes (Lichtstärke und pulsierendes Umgebungslicht) zu minimieren gibt es zahlreiche Einstellmöglichkeiten. Als Beispiel wird die Näherungsmessung im Einzelmessmodus des SI1141 näher betrachtet, der nur den Messkanal *PS1* besitzt und nur eine Leuchtdiode ansteuert.

Messmodus Einstellung und Auswahl des Messkanals

Mit dem Setzen des Registers *MEAS_RATE* auf 0x00 können nur noch die einzelnen Messungen gestartet werden die im Parameter *CHLIST* (0x01) vorher freigegeben wurden. Die Konfiguration des Parameters sieht folgendermaßen aus:

Bits 7,3 – sind reserviert;

- Bit 6** – EN_AUX = 1 gibt die Messung des zusätzlichen Kanals (Temperatur-/Versorgungsspannung-/Massespannung-Messung) frei;
- Bit 5** – EN_ALS_IR – Freigabe der Infrarotlicht Messung;
- Bit 4** – EN_ALS_VIS – Freigabe der Weißlichtmessung;
- Bit 2** – EN_PS3 – Freigabe Näherungsmessung am Kanal PS3;
- Bit 1** – EN_PS2 – Freigabe Näherungsmessung am Kanal PS2;
- Bit 0** – EN_PS1 – Freigabe Näherungsmessung am Kanal PS1;

Über weitere Parameter *PSLED12_SELECT* (0x02) und *PSLED3_SELECT* (0x03) werden die Messkanäle den angeschlossenen Infrarotdioden zugeordnet. Mit *PSLED12_SELECT*=0x01 und *PSLED3_SELECT*=0x00 findet die Näherungsmessung am Kanal *PS1* statt und die Messwerte werden in das Registerpaar *PS1_DATA0/PS2_DATA1* gespeichert.

Tab. 6.26 SI1141 Empfindlichkeitsauswahl und Anpassung an der Umgebungsbeleuchtung

PS_ADC_MISC	
Umgebungsbeleuchtung	
0x04	0x24
Normal	Stark
PS1_ADCMUX	
Empfindlichkeit	
0x00	0x03
Niedrig	Hoch

Einstellung der infraroten Lichtpulse

Die zum Detektieren von Objekten gesendeten infraroten Lichtpulse können eine höhere Lichtstärke erreichen, wenn die Stromstärke durch die Sendediode größer ist. Mit Gl. 6.39 kann die Stromstärke durch die LED1 und SI1141 berechnet werden, die mit den Bits 3:0 (*LED1_I*) vom Register *PS_LED21* (0x0F) eingestellt werden kann. Wenn *LED1_I* = „000“ ist, dann wird der Strom durch die Leuchtdiode abgeschaltet.

$$I [\text{mA}] = \begin{cases} \text{LED1_I} \cdot 5,6 & | 0 < \text{LED1_I} < 3 \\ (\text{LED1_I} - 2) \cdot 22,4 & | 2 < \text{LED1_I} < 13 \\ (\text{LED1_I} - 12) \cdot 22,4 & | \quad \text{LED1_I} > 12 \end{cases} \quad (6.39)$$

Die Dauer der Lichtpulse t_p wird über den Parameter *PS_ADC_GAIN* (0x0B) eingestellt und ist von der Gl. 6.40 gegeben. Der Einstellbereich des Parameters ist 0 ... 7.

$$t_p = 25,6 \mu\text{s} \cdot 2^{\text{PS_ADC_GAIN}} \quad (6.40)$$

Die Periodendauer des ADC-Taktes wird auch mit dem Faktor $2^{\text{PS_ADC_GAIN}}$ erhöht.

Auswahl des Messsensors und der Messeinstellungen

Zwei unterschiedlich große Infrarotsensoren erlauben Messungen der Näherung mit unterschiedlicher Empfindlichkeit, die über den Parameter *PS1_ADCMUX* (0x07) laut Tab. 6.26 auswählbar sind. Eine Korrektur der Lichtpulsdauer bei Messungen unter direktem Sonnenlicht kann über den Parameter *PS_ADC_MISC* (0x0C) erfolgen.

Um das Messrauschen bei der Quantisierung zu reduzieren wird vor jeder Näherungsmessung eine Erholungszeit t_R für den A/D-Wandler eingestellt. Diese Erholungszeit, die mit den Bits 6:4 (*PS_ADC_REC*) des Parameters *PS_ADC_COUNTERS* (0x0A) mit Gl. 6.41 einzustellen ist, soll groß genug sein um das Messrauschen zu minimieren. Bei Näherungsmessungen, die unter pulsierendem Umgebungslicht stattfinden, muss dafür gesorgt werden, dass die zwei aufeinanderfolgenden Messungen zeitnah geschehen um die Messungenauigkeiten klein zu halten, was kleine Erholungszeiten bedeutet. Bei der Wahl der Erholungszeit muss auch die Empfehlung des Herstellers, dass *PS_ADC_REC* das

Einerkomplement von PS_ADC_GAIN abbildet, berücksichtigt werden.

$$t_R[\text{ns}] = \begin{cases} 50 \cdot 2^{\text{PS_ADC_GAIN}} & | \text{PS_ADC_REC} = 0 \\ 50 \cdot 2^{\text{PS_ADC_GAIN}} \cdot (2^{2+\text{PS_ADC_REC}} - 1) & | \text{PS_ADC_REC} > 0 \end{cases} \quad (6.41)$$

Initialisierung des Bausteins

Der folgende Programmcode stellt die Initialisierung eines Sensors vom Typ SI1141 für die Näherungsmessung im Einzelmessmodus dar. Die Stromstärke durch die Sendediode ist auf 11,2 mA eingestellt und die Dauer der Lichtpulse wird um den Faktor 16 ($\text{PS_ADC_GAIN} = 4$) vervielfacht. Daraus erfolgt $\text{PS_ADC_REC} = 0x03$ für die Berechnung der Erholungszeit. Der Sensor soll unter normalem Umgebungslicht arbeiten und die Messungen sollen mit hoher Empfindlichkeit stattfinden.

```
uint8_t SI114x_Init(void)
{
    //Initialisierung wird vervollständigt
    if(SI114x_Write_ByteReg(HW_KEY_REG, HARDWARE_INIT)) return TWI_ERROR;
    //Einzelmessmodus
    if(SI114x_Write_ByteReg(MEAS_RATE_REG, 0x00)) return TWI_ERROR;
    //Freigabe Messkanal PS1
    if(!SI114x_Write_Param(CHLIST_PARAM, 0x01)) return TWI_ERROR;
    //Dem Messkanal PS1 wird LED1 zugewiesen, LED2 deaktiviert
    if(!SI114x_Write_Param(PSLED12_SEL_PARAM, 0x01)) return TWI_ERROR;
    //LED3 wird deaktiviert
    if(!SI114x_Write_Param(PSLED3_SEL_PARAM, 0x00)) return TWI_ERROR;
    //der Strom durch LED1 wird auf 11,2mA eingestellt,
    //durch LED2 abgeschaltet
    if(SI114x_Write_ByteReg(PS_LED21_REG, 0x02)) return TWI_ERROR;
    //der Strom durch LED3 wird abgeschaltet
    if(SI114x_Write_ByteReg(PS_LED3_REG, 0x00)) return TWI_ERROR;
    //die Pulsbreite wird um Faktor 16 vergrößert
    if(!SI114x_Write_Param(PS_ADC_GAIN_PARAM, 0x04)) return TWI_ERROR;
    //die Messungen finden unter normaler Beleuchtung statt
    if(!SI114x_Write_Param(PS_ADC_MISC_PARAM, 0x04)) return TWI_ERROR;
    //der große Infrarotsensor wird ausgewählt
    if(!SI114x_Write_Param(PS1_ADCMUX_PARAM, 0x03)) return TWI_ERROR;
    //Einstellung der Erholungszeit
    if(!SI114x_Write_Param(PS_ADC_COUNTER_PARAM, 0x30)) return TWI_ERROR;
    return TWI_OK;
}
```

Die Funktion `SI114x_Init()` gibt den Wert `TWI_OK` („0“) zurück wenn sie fehlerfrei ausgeführt wurde. Die Einstellungen müssen an die konkrete Anwendung angepasst werden.

Messungsstart

Nachdem die gewünschten Einstellungen gespeichert wurden, kann eine Näherungsmessung mit dem Befehl *PS_FORCE* gestartet werden. Die Dauer einer Messung beträgt laut [28] für die Standardeinstellungen 155 µs. Das Ende einer Messung kann über einen Interrupt signalisiert werden.

Lesen der Messwerte

Das Messergebnis wird in den Registern *PSI_DATA0* und *PSI_DATA1* gespeichert und kann mit der Funktion *SI114x_Read_WordReg()* ausgelesen werden. Diese Funktion, die im Folgenden aufgelistet ist, liest die Inhalte zweier Register, deren Adresse aufeinander folgen, fasst die Inhalte zu einem 16-Bit-Wert zusammen und speichert diesen Wert in eine Variable.

```
uint8_t SI114x_Read_WordReg(uint8_t ucreg_address, uint16_t* uidata_word)
{
    uint8_t ucDeviceAddress, ucDataLSByte, ucDataMSByte;

    ucDeviceAddress = ucSI114xAddress << 1; //Adresse des Proximity-
                                                //Sensors
    ucDeviceAddress |= TWI_WRITE; //Write-Modus
    TWI_Master_Start(); //Start
    if((TWI_STATUS_REGISTER) != TWI_START) return TWI_ERROR;
    TWI_Master_Transmit(ucDeviceAddress); //Device Adresse senden
    if((TWI_STATUS_REGISTER) != TWI_MT_SLA_ACK) return TWI_ERROR;
    //die Adresse des gewünschten Registers wird gesendet
    TWI_Master_Transmit(ucreg_address);
    if((TWI_STATUS_REGISTER) != TWI_MT_DATA_ACK) return TWI_ERROR;
    TWI_Master_Start(); //Restart
    if((TWI_STATUS_REGISTER) != TWI_RESTART) return TWI_ERROR;
    ucDeviceAddress = (ucSI114xAddress << 1) | TWI_READ; //Read-Modus
    TWI_Master_Transmit(ucDeviceAddress); //Device Adresse senden
    if((TWI_STATUS_REGISTER) != TWI_MR_SLA_ACK) return TWI_ERROR;
    //Inhalt des adressierten Registers wird eingelesen
    ucDataLSByte = TWI_Master_Read_Ack();
    if((TWI_STATUS_REGISTER) != TWI_MR_DATA_ACK) return TWI_ERROR;
    ucDataMSByte = TWI_Master_Read_NAck();
    *uidata_word = (ucDataMSByte << 8) + ucDataLSByte;
    if((TWI_STATUS_REGISTER) != TWI_MR_DATA_NACK) return TWI_ERROR;
    TWI_Master_Stop(); //Stop
    return TWI_OK;
}
```

Die Adresse der Variable, in der das Messergebnis gespeichert wird, zusammen mit der Adresse des Registers *PSI_DATA0* werden als Parameter bei dem Aufruf der Funk-

tion übergeben. Nach dem Auslesen des ersten Registers wird der interne Adresszähler inkrementiert und das zweite Register kann im gleichen I²C-Vorgang gelesen werden.

```
unsigned int uiPSValue;
SI114x_Read_WordReg(PS1_DATA0_REG, &uiPSValue);
```

6.7.2.5 Interrupts

Ein SI114x verfügt über einen open-drain-Ausgang der als Interruptauslöser für einen Master dienen kann. Zwischen diesem Ausgang und der Versorgungsspannung des Bausteins muss ein pull-up Widerstand angeschlossen sein. Er wird mit dem Speichern von 0x01 in das Konfigurationsregister *INT_CFG* (0x03) aktiviert. Die Messkanäle, die ein Interrupt auslösen können, werden im Register *IRQ_ENABLE* (0x04) freigeschaltet. Nach einer Näherungsmessung über den Kanal *PS1* kann ein Interrupt ausgelöst werden, wenn eine Messung abgeschlossen ist oder wenn das Messergebnis einen gesetzten Grenzwert durchquert, also von oben oder unten überschreitet. Damit ein Interrupt nach jeder abgeschlossenen *PS1* Messung ausgelöst wird, müssen in das Register *IRQ_ENABLE* der Wert 0x04 und in das Register *IRQ_MODE1* (0x05) der Wert 0x00 gespeichert werden.

Durch die Erfüllung einer interruptauslösenden Bedingung wird das entsprechende Bit (Bit 2 für den Kanal *PS1*) im Statusregister *IRQ_STATUS* (0x21) gesetzt und der INT-Ausgang auf Low geschaltet. Dieser Zustand bleibt gespeichert bis er vom Master durch das Überschreiben mit „1“ der gesetzten Bits im Register *IRQ_STATUS* zurückgesetzt wird.

Die im Folgenden aufgelistete Interruptinitialisierung kann die Funktion *SI114x_Init()* ergänzen.

```
//Interrupt-Einstellungen
//die Interrupts über den Kanal PS1 werden freigeschaltet
if(SI114x_Write_ByteReg(IRQ_ENABLE_REG, 0x04)) return TWI_ERROR;
//eine abgeschlossene Messung am Kanal PS1 löst ein Interrupt aus
if(SI114x_Write_ByteReg(IRQ_MODE1_REG, 0x00)) return TWI_ERROR;
//INT-Ausgang des Bausteins wird aktiviert
if(SI114x_Write_ByteReg(INT_CFG_REG, 0x01)) return TWI_ERROR;
//das Interrupt Statusregister wird ausgelesen
if(SI114x_Read_ByteReg(IRQ_STATUS_REG, &ucResponse)) return TWI_ERROR;
//durch das Überschreiben des Registers mit dem gleichen Wert
//werden alle Interrupts zurückgesetzt
if(SI114x_Write_ByteReg(IRQ_STATUS_REG, ucResponse)) return TWI_ERROR;
```

6.7.2.6 SI114x Netzwerkidentifikation

Die Bausteine der Serie lassen sich über die Nur-Lese-Register *PART_ID* (0x00), *REV_ID* (0x01) und *SEQ_ID* (0x02) in einem Netzwerk identifizieren. Im Register *PART_ID* ist der Wert 0x41 für einen Baustein vom Typ SI1141 gespeichert, der Wert 0x42 für SI1142 bzw. 0x43 für SI1143. Das Lesen dieses Registers liefert den Bausteintyp, so dass die passende Initialisierung und korrekte Ansteuerung vorgenommen werden können.

Die Revisionsnummer der internen Ablaufsteuerung, die im Register *SEQ_ID* gespeichert ist, kann eine wichtige Rolle spielen. Zum Beispiel wird der 16 Bit große Grenzwert, dessen Überschreitung bei der Messung im automatischen Messmodus mit dem Kanal *PS1* zu einem Interrupt führen kann, ab der Revision A11 (Speichercode 0x09) in das Registerpaar *PS1_TH0* (0x11) und *PS1_TH1* (0x12) gespeichert und bis zu dieser Revision (Speichercode kleiner 0x09) wird er in komprimierten Form im Register *PS1_TH0* gespeichert. Weitere Unterschiede sind dem Datenblatt [28] zu entnehmen. Im Register *REV_ID* ist immer 0x00 gespeichert.

6.8 Strommessung mit dem LMP92064

LMP92064 ist ein über SPI vernetzbarer Baustein, der für die gleichzeitige Messung eines Gleichstroms und einer Gleichspannung entwickelt wurde [31]. Der Baustein besitzt zwei 12 Bit A/D-Wandler, mit denen die Spannungen an externen Widerständen gemessen werden. Mit einer Umwandlungsrate von 125 kS/s können schnelle Änderungen der elektrischen Größen erfasst werden. Der Hersteller gibt eine Bandbreite für die Strommessung von 70 kHz und für die Spannungsmessung von 100 kHz an. Weil die zwei Größen gleichzeitig abgetastet werden, kann auch die elektrische Leistung und durch Integrieren die elektrische Energie berechnet werden. Der Baustein nimmt die Werte im Freilaufmodus auf, eine interne Logik verhindert das Überschreiben eines alten Datensatzes während der Datenübertragung, die bis zu 20 Mbit/s erreichen kann.

6.8.1 LMP92064 Aufbau

Für die Steuerung der A/D-Wandler, die Konfiguration der seriellen Schnittstelle, die Identifizierung des Bausteins in einem Netzwerk und das Speichern der Messwerte, besitzt LM92064 einen Registersatz, dessen 8 Bit große Register mit einem 16 Bit Adresszähler adressierbar sind. Eine Besonderheit des Bausteins besteht darin, dass mit dem Lesen oder Schreiben eines Registers der Adresszähler dekrementiert wird. Das automatische Dekrementieren des Adresszählers führt zum Adressensprung von 0x0000 auf 0xFFFF. Das höherwertige Byte eines 16 Bit Registers besitzt die höhere, und das niederwertige Byte die niedrigere Adresse. Der Baustein kann hardwaremäßig mit dem Setzen des RESET-Eingangs auf High zurückgesetzt werden. Das Gleiche kann softwaremäßig durch das Setzen des Bit 7 im Konfigurationsregister A mit der Adresse 0x0000 erreicht werden. Nach dem Hochfahren wird dieses Bit automatisch gelöscht. Die weiteren Bits der Konfigurationsregister können nur gelesen werden, deshalb werden sie weiter explizit nicht beschrieben. Für die Identifizierung des Bausteins und das Testen der Kommunikation gibt es mehrere Register die in der Tab. 6.27 aufgelistet sind.

Eine interne Referenzspannung von 2,048 V für beide Messkanäle vereinfacht die Beschaltung des Bausteins. Zwei Operationsverstärker mit einer Eingangsimpedanz von

Tab. 6.27 LMP92064 Identifikationsregister

Register		Adresse	Inhalt
Baustein Typ		0x0003	0x07
Baustein ID	Low Byte	0x0004	0x00
	High Byte	0x0005	0x04
Baustein Revision		0x0006	0x01
Hersteller ID	Low Byte	0x000C	0x51
	High Byte	0x000D	0x04

100 GΩ sorgen dafür, dass die zu messende Spannungen unbelastet dem jeweiligen A/D-Wandler zugeführt werden. Für die Messung der Spannung wird ein Impedanzwandler mit dem Verstärkungsfaktor 1 verwendet. Eine positive Spannung gegenüber dem Masseanschluss des Bausteins bis zur Höhe der Referenzspannung kann mit einer Schrittweite von $2048 \text{ mV} / 2^{12} = 0,5 \text{ mV}$ gemessen werden. Der Spannungsmessbereich wird mit einem Spannungsteiler bis zu U_{\max} erweitert (siehe Abb. 6.39) was auch zur Erhöhung der Schrittweite um den Faktor $(R_1 + R_2) / R_2$ führt:

$$U_{\max} = U_{\text{ref}} \cdot \frac{R_1 + R_2}{R_2} \quad (6.42)$$

An einem externen Shunt-Widerstand wird die Spannung proportional zum Strom abgegriffen und mit dem Faktor 25 (typischer Wert) verstärkt, bevor sie digitalisiert wird. Diese Spannung kann maximal $2048 \text{ mV} / 25 = 81,92 \text{ mV}$ groß sein, was einer Schrittweite von $81,92 \text{ mV} / 2^{12} = 20 \mu\text{V}$ entspricht. Der digitale Wertebereich des A/D-Wandlers für die Strommessung ist durch die interne Logik des Bausteins auf 0 ... 3840 begrenzt, was zu einer maximalen Messspannung von 76,8 mV führt. Höhere Eingangsspannungen im zulässigen Spannungsbereich führen zur Anzeige des digitalen Wertes 3840 (oder 0x0F00). Die Offset-Spannung dieses Verstärkers beträgt $\pm 15 \mu\text{V}$. Die digitalisierten Werte der Messspannungen werden in den 16 Bit Datenregister abgelegt deren Adressen sich in der Tab. 6.28 befinden.

Ein unvollständiges Lesen der Datenregister soll das Überschreiben der alten Messwerte verhindern. Bei den getesteten Exemplaren wurde festgestellt, dass die Messwerte aktualisiert werden auch wenn nur die Spannungswerte ohne die Stromwerte gelesen werden.

Tab. 6.28 LMP92064 Messdatenregister

Datenregister		Adresse
Spannung	Low Byte	0x0200
	High Byte	0x0201
Strom	Low Byte	0x0202
	High Byte	0x0203

6.8.2 Serielle Kommunikation

LMP92064 ist als Slave konfiguriert und verfügt über eine Vierdraht SPI-Schnittstelle, die eine bidirektionale Datenübertragung im Modus 0 oder 3 mit dem höchstwertigen Bit eines Bytes zuerst ermöglicht. Das einfache Übertragungsprotokoll und eine Übertragungsrate von bis zu 20 Mbit/s ermöglichen das Ausnutzen der hohen Umwandlungsrate des Bausteins. Um in einer Schaltung mehrere Messeinheiten mit dem gleichen Softwaremodul anzusteuern ist es sinnvoll, die einzelnen Bausteine mit jeweils einem Definitionsarray zu identifizieren. Für den Messsensor aus Abb. 6.39 wird in der Hauptdatei folgende SPI-Datenstruktur deklariert (Abschn. 6.1.5):

```
LMP92064_pins LMP92064_1 = {{/*CS_DDR*/      &DDRB,
                                /*CS_PORT*/     &PORTB,
                                /*CS_pin*/      PBO,
                                /*CS_state*/    ON}}; //ON = 1
```

Nachdem der dedizierte Slave Select Anschluss der SPI-Schnittstelle auf Ausgang deklariert wurde, wird mit dem Aufruf der Funktion:

```
SPI_Master_Init(SPI_INTERRUPT_DISABLE, SPI_MSB_FIRST, SPI_MODE_3,
                 SPI_FOSC_DIV_4);
```

der Mikrocontroller als Master eingestellt und für die Kommunikation mit dem LMP92064 konfiguriert. Mit einer SPI-Nachricht kann der Master einzelne Register oder einen Registerblock ansprechen. Eine Nachricht beginnt mit dem Setzen der Slave Select Leitung auf Low gefolgt von der Übertragung der Adresse des gewünschten Registers, bzw. des ersten Registers aus dem Registerblock. Mit dem Überschreiben des höherwertigen Bits der zu sendenden Adresse kann die auszuführende Operation kodiert werden: „0“ bedeutet Schreiben, „1“ Lesen. Die Abb. 6.38 stellt die zeitlichen Verläufe beim Auslesen der Messwerte dar. Die höchste Adresse aus dem Registerbereich (0x0203) wird übertragen, für jedes auszulesende Byte überträgt der Master ein Dummy-Byte.

6.8.3 Messen mit dem LMP92064

Dank der hohen Eingangsimpedanz der Messeingänge, der niedrigen Offset-Spannung der Operationsverstärker, der hohen Auflösung der A/D-Wandler und der hohen Abtastrate, können simultane, präzise Messungen der elektrischen Größen in Gleichspannungsnetzwerken realisiert werden. Diese Messungen können für die Überwachung oder für die Berechnung von Leistung und Energie durchgeführt werden. Die Abb. 6.39 zeigt beispielhaft den Anschluss eines LMP92064 an einem Mikrocontroller, sowie die Mess- und Stromkalibrierschaltung.

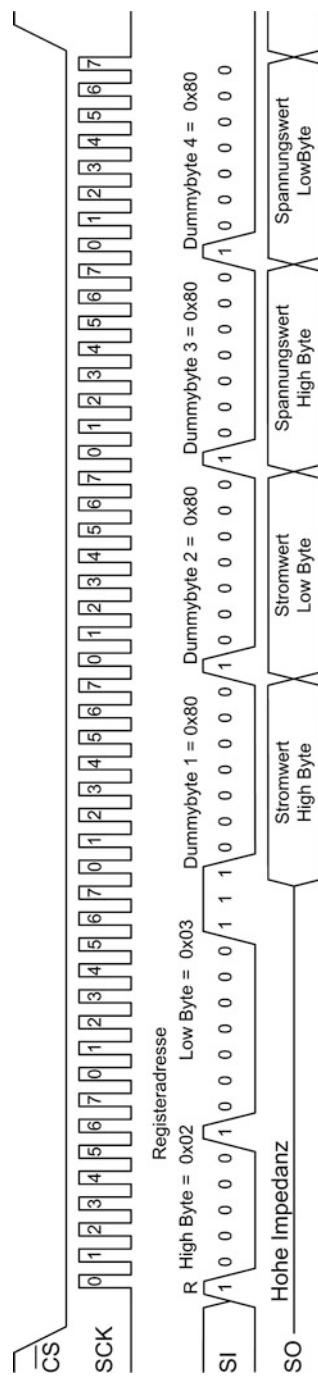


Abb. 6.38 LMP92064 Auslesen Datenregister

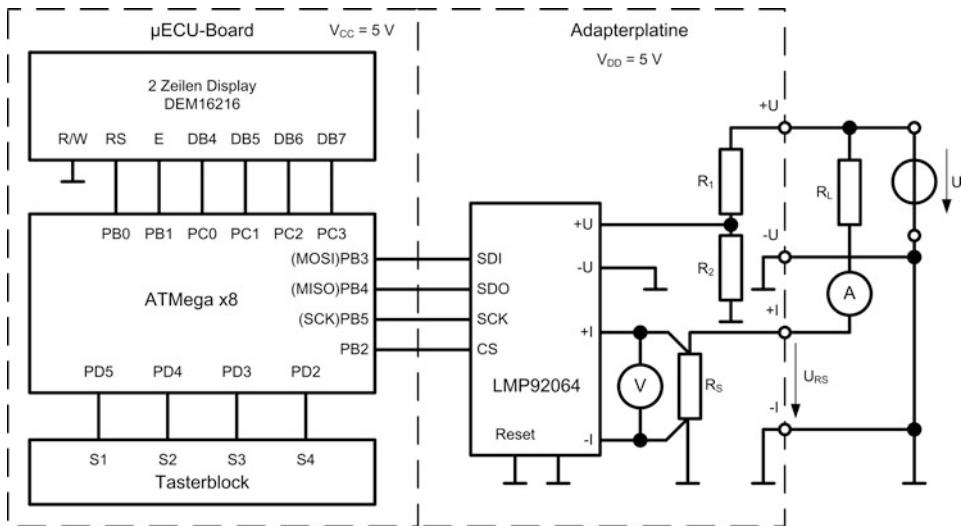


Abb. 6.39 LMP92064 Beschaltung

6.8.3.1 Spannungsmessung

Für die Messung von Spannungen, die größer als die Referenzspannung sind, wird ein Spannungsteiler benötigt, der unter der Berücksichtigung folgender Schritte dimensioniert werden kann:

- es wird eine maximale Stromstärke durch den Spannungsteiler I_D gewählt, die die Verluste minimiert und die zu messende Schaltung nicht belastet
- ausgehend von der maximalen Messspannung U_{\max} , die auftreten kann, wird der Gesamtwiderstand $R_1 + R_2$ berechnet so, dass:

$$I = \frac{U_{\max}}{R_1 + R_2} \leq I_D \quad (6.43)$$

- Aus Gl. 6.42 wird das Verhältnis R_1 / R_2 berechnet. Mit der Summe und dem Verhältnis der Widerstände werden R_1 und R_2 berechnet. Die Genauigkeit der Widerstände soll kleiner 1 % sein um präzise Spannungsmessungen realisieren zu können.

Die Gestaltung folgender Funktion entspricht diesen Überlegungen, mit denen die beste Spannungsauflösung erzielt wird. Die Funktion, die mit den Widerstandswerten in $[\Omega]$ aufgerufen wird, liest das Messregister *DATA_V_OUT_REG*, berechnet und gibt den Spannungswert in [mV] zurück. Bei einer Taktfrequenz des Mikrocontrollers von 18,432 MHz, beträgt die Rechenzeit des Spannungswertes ca. 41 μ s.

```

uint16_t LMP92064_Get_VoltageValue(LMP92064_pins sdevice_pins, long lr2,
                                    long lr1)
{
    unsigned long ulVoltage;
    ulVoltage = LMP92064_Read_WordReg(sdevice_pins, DATA_V_OUT_REG);
    ulVoltage = (ulVoltage * (lr1 + lr2)) / 2; //0,5mV = 1/2 Auflösung
                                                //des A/D-Wandlers
    ulVoltage = ulVoltage / lr2;    //lr1 + lr2 Gesamtwiderstand
    return ulVoltage;
}

```

Wenn das Verhältnis R_1 / R_2 so gewählt wird, dass:

$$\frac{R_1}{R_2} = \begin{cases} (2 \cdot n) - 1 & \text{oder} \\ 2^n - 1 & \end{cases} \quad \text{mit } n \in \mathbb{N} \quad (6.44)$$

dann kann der Spannungswert in [mV] folgendermaßen berechnet werden:

```

ulVoltage = ulVoltage * n; //für R1/R2 = 2n-1
ulVoltage = ulVoltage << 2n-1; //für R1/R2 = 2n-1

```

was zu einer Rechenzeit unter 2,5 µs führt.

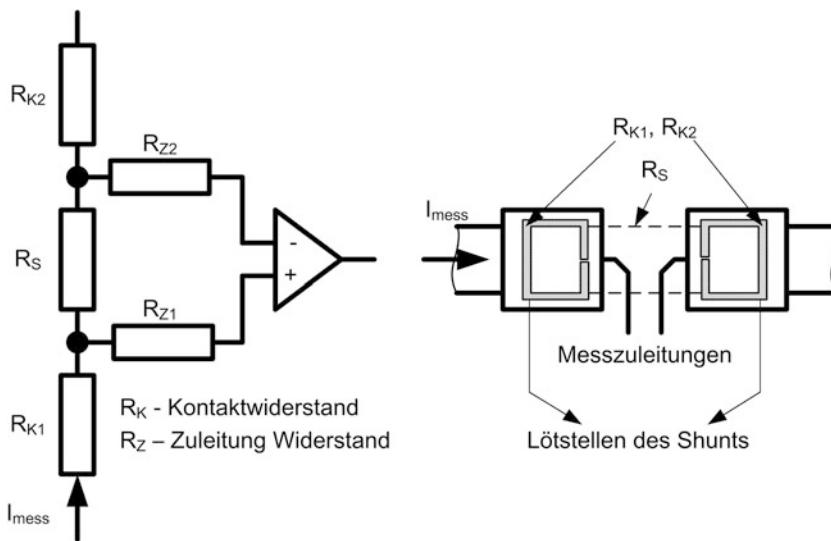
6.8.3.2 Strommessung

Der zu messende Strom verursacht am Strommesswiderstand einen Spannungsabfall, der verstärkt und digitalisiert als Grundlage für die Berechnung der Stromstärke dient. Um den Einfluss der Kontaktwiderstände R_{K1} und R_{K2} auf die Strommessung zu reduzieren wird das Vierleiter-Messverfahren verwendet [32] wie in Abb. 6.40a. Diese Kontaktwiderstände entstehen beim Löten des Messwiderstandes. Das Layout im Fall eines SMD-Shunts, das für die Strommessung nach dem Vierleiter-Verfahren eingesetzt wird, ist in Abb. 6.40b dargestellt.

Eine präzise Messung wird nur gewährleistet, wenn die gesamte Messkette kalibriert ist. In der Schaltung aus Abb. 6.39 wurde ein 150 mΩ großer SMD-Shunt verbaut. Entsprechend den Stromstärken aus der Tab. 6.29 sind die Spannungsabfälle am Shunt gemessen. Abgesehen von den Messungenauigkeiten ergibt sich aus dieser Messreihe ein Wert von ca. 160 mΩ, mit dem weiterhin gerechnet wird.

Tab. 6.29 Messung des Strommesswiderstandes

I _{mess} [mA]	25,5	50,3	100	200	300	400
U _{RS} [mV]	4	8	16	32,1	48	64,1

**Abb. 6.40** Vierleiter-Messverfahren**Tab. 6.30** Kennlinie Messstrom

I _{soll} [mA]	10	25	50	100	200	300	400	500
I _{ist} [mA]	9,5	24,5	49,5	99,3	199	298,6	398	498

Entsprechend einem eingestellten Strom von 200 mA wird aus dem Datenregister des Bausteins der digitale Wert 1504 gelesen. Mit diesem Wert errechnet sich eine Stromschrittweite von ca. 0,133 mA; der maximale Strom der mit diesem Shunt gemessen werden kann, ist:

$$I_{\text{mess max}} = 0,133 \text{ mA} \cdot 3840 \approx 510 \text{ mA.}$$

Ein Stromwert wird aus dem digitalen Wert `ulCurrent` mit der Anweisung:

```
ulCurrent = (ulCurrent * 133) / 1000;
//die Stromschrittweite beträgt 0,133 mA
```

in ca. 35 µs berechnet. Zum Vergleich stehen in der Tab. 6.30 eine Reihe von eingestellten Stromwerten und die Mittelwerte von über 100 Messungen gegenüber.

6.9 Temperaturmessung mit dem TMP75

TMP75 ist ein digitaler Temperatursensor, der die Temperatur im spezifizierten Bereich von -40 bis $+125^{\circ}\text{C}$ mit einer typischen Genauigkeit von $\pm 0,5^{\circ}\text{C}$ misst [33]. Er kann als Slave über I²C oder SMBus im Fast- oder High-Speed-Modus konfiguriert und ausgelesen

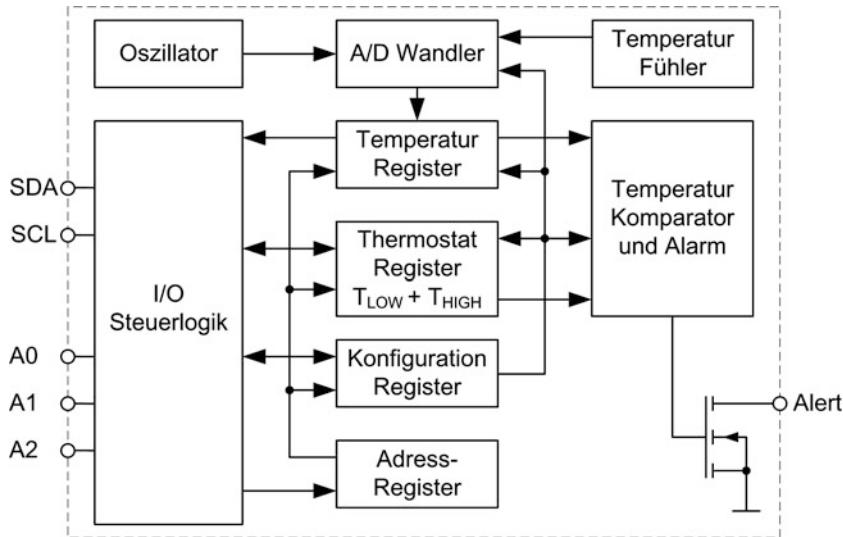


Abb. 6.41 TMP75 Blockschaltbild des Sensors

werden. Bis zu acht gleiche Sensoren können dank der drei Adresseingänge am gleichen Bus betrieben werden. Ein grobes Blockschaltbild des Sensors ist in Abb. 6.41 dargestellt. Als Temperaturfühler wird eine Diode verwendet. Die zur Temperatur proportionale analoge Spannung wird mit einem A/D-Wandler mit einstellbarer Bitauflösung digitalisiert und in einem 12 Bit Temperaturregister gespeichert. Dieses Register kann über die serielle Schnittstelle nur gelesen werden. Der TMP75 kann im Freilauf- oder Einzelmessung-Betrieb arbeiten. Im Freilauf-Betrieb wird die Temperatur kontinuierlich gemessen. Um Energie zu sparen und dadurch auch die eigene Erwärmung, die zu Messabweichungen führt, zu reduzieren, kann der Sensor in einen Energiesparmodus geschaltet werden. In diesem Modus bleibt nur noch die serielle Schnittstelle aktiv. Über die eingebaute Thermostatfunktion kann der Sensor die Überschreitung einer maximalen oder die Unterschreitung einer minimalen Temperatur über einen open-drain Ausgang signalisieren. Dieser Ausgang benötigt einen externen pull-up Widerstand. Die zwei Temperaturgrenzen werden in zwei 16 Bit Register gespeichert und sind über die Software jederzeit einstellbar. Ein weiteres Register sorgt für die Konfiguration des Sensors und die Steuerung der Messung. Der Zugriff auf die einzelnen Register wird über das Adressregister realisiert.

6.9.1 Sensorkonfigurierung

Das 8Bit Konfigurationsregister mit der Adresse 0x01 kann gelesen oder geschrieben werden. Die Bedeutung der einzelnen Bits ist im Folgenden erläutert:

Tab. 6.31 TMP75 Auflösung und Messdauer

R1	R0	Bitauflösung	Temperaturschritt	Max. Messdauer
0	0	9	0,5 °C	37,5 ms
0	1	10	0,25 °C	75 ms
1	0	11	0,125 °C	150 ms
1	1	12	0,0625 °C	300 ms

Tab. 6.32 TMP75 Codierung der Fehlerzahl

F1	F0	Aufeinanderfolgende Fehler
0	0	1
0	1	2
1	0	4
1	1	6

Bit7 – OS – Das Setzen dieses Bits durch die Ansteuersoftware, während sich der Sensor im Energiesparmodus befindet, startet eine einzelne Temperaturmessung. Nach dem Beenden der Messung und Speichern des Temperaturwertes kehrt der Sensor in den Energiesparmodus zurück. Dieses Bit wird immer als „0“ gelesen.

Bit6:5 – R1:R0 – Mit diesen zwei Bits wird die Temperaturaufösung eingestellt (Tab. 6.31).

Bit4:3 – F1:F0 – Über diese Bits wird die Anzahl der aufeinanderfolgenden Über- und Unterschreitungen der Temperaturgrenzen eingestellt um einen Temperaturalarm auszulösen (Tab. 6.32).

Bit2 – POL – Dieses Bit bestimmt die Polarität des Alarmsignals. Wenn das Bit „0“ ist, wird das Alarmsignal im Alarmfall auf Low gesetzt.

Bit1 – TM – Wenn dieses Bit „1“ ist, arbeitet der Thermostat im Komparator Modus, ansonsten im Interrupt Modus.

Bit0 – SD – Mit dem Setzen dieses Bits durch die Ansteuersoftware wird der Sensor in den Energiesparmodus versetzt.

6.9.2 Serielle Schnittstelle

Der TMP75 implementiert für die Kommunikation mit einem Master die I²C und SMBus Protokolle. Die maximale Übertragungsrate kann 400 kBit/s im Fast Modus und 3,4 Mbit/s im High-speed Modus erreichen. Die minimale Grenze für die Frequenz des Bustaktes liegt bei 1 kHz und dadurch weicht sie von der entsprechenden Anforderung der I²C Spezifikation (0 Hz) leicht ab. Über die drei Adresseingänge können die niederwertigen Bits der Adresse eingestellt werden. Diese Eingänge können entweder an GND („0“) oder an Vcc („1“) angeschlossen werden und dadurch bis zu acht unterschiedliche Adressen realisieren. Über die drei Adresseingänge des Sensors TMP175, der ähnlich wie der TMP75

aufgebaut ist und die gleichen Eigenschaften besitzt, können bis zu 27 unterschiedlichen Adressen realisiert werden. Diese Eingänge können an Vcc oder GND angeschlossen werden. Die genauen einstellbaren Adressen sind dem Datenblatt [33] zu entnehmen.

Die Grundadresse des TMP75, wenn alle Adresseingänge mit GND verbunden sind, lautet 0x90 und ist gleich mit der des A/D- und D/A-Wandler Bausteins PCF8591. Schaltungstechnisch muss das bei der Vernetzung der zwei Bausteine an einem Bus berücksichtigt werden damit alle Busteilnehmer eindeutig identifizierbar sind. Für das Beispiel in Abb. 6.42 lautet die Adresse von IC1 0x90 (wie die Adresse des Echtzeituhr Bausteins MAX31629) und die von IC2 0x9C.

Die Temperaturwerte und das Konfigurationsbyte sind in flüchtigen Registern gespeichert. Die gewünschte Konfiguration und eventuell die Temperaturgrenzen müssen vom Master in der Initialisierungsphase des Busses geladen werden. Um auf den Inhalt eines Registers zuzugreifen muss der Master nach der Initiierung der Kommunikation und Adressierung des Slaves die Adresse des Zielregisters senden. Dabei ist das Read/Write Bit zurückgesetzt. Während eines Schreibvorgangs sendet der Master weiterhin ein oder zwei Bytes und beendet dann die Kommunikation. Der Master prüft nach jedem gesendeten Byte, ob der Sensor mit ACK den Empfang bestätigt hat. Um Daten auszulesen muss der Master nach der Adressierung des Zielregisters eine RESTART Sequenz erzeugen und die Slave Adresse mit gesetztem Read/Write Bit senden. Weiterhin erzeugt er den Takt für

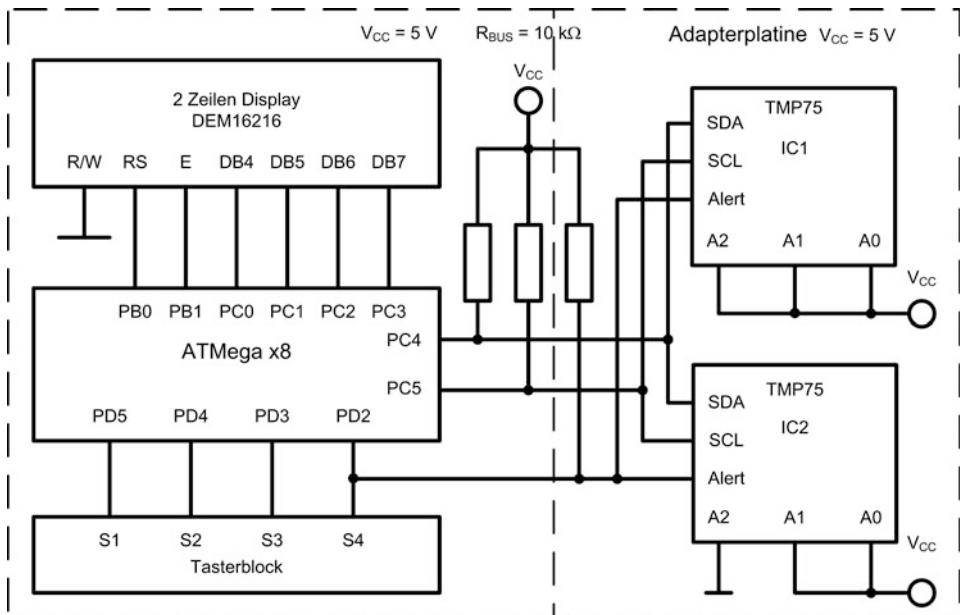


Abb. 6.42 TMP75 Beschaltung zweier Temperatursensoren

die Übertragung und empfängt die Daten. Das letzte empfangene Byte wird mit NACK quittiert, alle anderen mit ACK.

Die Sensoren TMP75 und TMP175 reagieren auf die reservierte Adresse 0x00 (genereller Aufruf), bestätigen deren Empfang und falls das folgende Byte 0x06 ist, laden sie die Register mit den Anfangswerten nach Reset und speichern den Zustand der Adress-eingänge.

Falls sich einer der Teilnehmer in einem I²C Bus „aufhängt“ und eine der Busleitungen dauerhaft auf Low zieht, bricht die gesamte Kommunikation zusammen. Daher implementiert der Sensor eine Timeout Funktion, die automatisch eingreift, wenn zwischen einer START und einer STOP Sequenz eine der Busleitungen für länger als 54 ms auf Low bleibt. In einem solchen Fall führt die Funktion zu einem Reset der eigenen seriellen Schnittstelle.

6.9.3 Temperaturmessung

Nach dem Hochfahren des Sensors müssen die Bitauflösung und der Messbetrieb über das Konfigurationsregister eingestellt werden. Wenn im Vordergrund die Messung der Temperatur und der Energieverbrauch stehen, wird durch das Setzen des Bit0 im Konfigurationsregister der Energiesparmodus gewählt, in dem der Einzelmessung-Betrieb möglich ist. Jede Temperaturmessung wird vom Master gestartet, indem das Bit OS vom gleichen Register gesetzt wird. Das Ende der Messung wird nicht signalisiert, die maximale Messdauer muss berücksichtigt werden. Die Bitauflösung wirkt sich auf die Messdauer aus, ohne jedoch die Genauigkeit zu beeinflussen. Das Temperaturregister des Sensors ist über die Adresse 0x00 erreichbar, ist 12 Bit groß (T11:0) und speichert den gemessenen Temperaturwert in einer Zweierkomplement-Darstellung mit einem Temperaturschritt von 0,0625 °C. Diese Darstellung ermöglicht das Speichern von positiven und negativen Fest-kommazahlen als Ganzzahlen. Der Inhalt des Temperaturregisters wird auf zwei Bytes aufgeteilt, wie in Tab. 6.33 ausgeführt, um ihn über die serielle Schnittstelle übertragen zu können. Zuerst wird das höherwertige Byte und dann das niederwertige ausgelesen.

Das Bit T11 ist „0“ für positive und „1“ für negative Temperaturen. Für eine ausgelesene positive Temperatur erhält man den Temperaturwert als reelle Zahl durch Teilung der 16 Bit Zahl durch 256. Für eine negative Temperatur wird:

Tab. 6.33 TMP75 Temperaturwert Codierung

Temperatur [°C]	Höchstwertiges Byte								Niederwertiges Byte							
	T11	T10	T9	T8	T7	T6	T5	T4	T3	T2	T1	T0	0	0	0	0
+23,75	0	0	0	1	0	1	1	1	1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
-17,25	1	1	1	0	1	1	1	0	1	1	0	0	0	0	0	0

- das Vorzeichen gemerkt, dann
- die 16 Bit Zahl bitweise invertiert und eine Eins dazu addiert um das Zweierkomplement zu bilden
- und das Ergebnis durch 256 geteilt um den Temperaturbetrag als reelle Zahl zu erhalten.

Eine Funktion, die ein Temperaturregister ausliest, lautet:

```
uint8_t TMP75_Read_Temperature(uint8_t ucdevice_address,
                                uint8_t uctemp2read)
{
    uint8_t ucDeviceAddress, ucTempHigh, ucTempLow;
    //Adresse des TMP75-Temperatursensors bilden
    ucDeviceAddress = (ucdevice_address << 1) | TMP75_DEVICE_TYPE_ADDRESS;
    ucDeviceAddress |= TWI_WRITE;      //Write-Modus
    TWI_Master_Start();              //Start
    if((TWI_STATUS_REGISTER) != TWI_START) return TWI_ERROR;
    TWI_Master_Transmit(ucDeviceAddress); //Device Adresse senden
    if((TWI_STATUS_REGISTER) != TWI_MT_SLA_ACK) return TWI_ERROR;
    //die Adresse des gewünschten Temperaturregisters wird gesendet
    TWI_Master_Transmit(uctemp2read);
    if((TWI_STATUS_REGISTER) != TWI_MT_DATA_ACK) return TWI_ERROR;
    TWI_Master_Start(); //Restart
    if((TWI_STATUS_REGISTER) != TWI_RESTART) return TWI_ERROR;
    ucDeviceAddress = (ucdevice_address << 1);
    ucDeviceAddress |= TMP75_DEVICE_TYPE_ADDRESS | TWI_READ;
    TWI_Master_Transmit(ucDeviceAddress); //Device Adresse im Read-
                                         //Modus senden
    if((TWI_STATUS_REGISTER) != TWI_MR_SLA_ACK) return TWI_ERROR;
    //Inhalt des adressierten Registers wird eingelesen
    ucTempHigh = TWI_Master_Read_Ack();
    if((TWI_STATUS_REGISTER) != TWI_MR_DATA_ACK) return TWI_ERROR;
    ucTempLow = TWI_Master_Read_NAck();
    if((TWI_STATUS_REGISTER) != TWI_MR_DATA_NACK) return TWI_ERROR;
    TWI_Master_Stop(); //Stop
    iTemperature = (ucTempHigh << 8) + ucTempLow;
    return TWI_OK;
}
```

Beim Aufruf der Funktion werden als Parameter die Device Chip Adresse des Sensors `ucdevice_address` und die Adresse des auszulesenden Registers `uctemp2read` übergeben. Der Temperaturwert wird in Zweierkomplement-Darstellung in die Variable:

```
int iTemperature;
```

gespeichert. Aus dem Hauptprogramm greift man auf diese im Softwaremodul TMP75.c deklarierte Variable über die folgende Aufrufsschnittstelle zu:

```
int TMP75_Get_Temperature(void)
{
    return iTemperature;
}
```

6.9.4 Thermostatfunktion

Der Sensor vergleicht im Hintergrund den gespeicherten Temperaturwert mit dem Inhalt zweier Thermostatregister. Beide sind Schreib-Lese-Register, haben die gleiche Größe wie das Temperaturregister und benutzen für das Speichern der Grenztemperaturen die Zweierkomplement-Darstellung. Um einen vorgegebenen Temperaturwert zu codieren, muss folgendes beachtet werden:

- wenn der Wert positiv ist, wird er mit 256 multipliziert;
- wenn er negativ ist, wird der Temperaturbetrag mit 256 multipliziert, das 2-Byte Ergebnis bitweise invertiert und eine Eins dazu addiert (Zweierkomplement).

Das Thermostatregister mit der Adresse 0x02 speichert die untere Grenztemperatur, das Register mit der Adresse 0x03 die obere. Das Überschreiten der oberen Grenze beziehungsweise das Unterschreiten der unteren führt zu einem Alarmzustand, der am Alert Ausgang (siehe Abb. 6.43) signalisiert wird. Wenn der Sensor als Zwei-Punkt-Regler benutzt wird, somit die Thermostat Funktion im Vordergrund steht und eine schnelle Reaktion im Alarmfall erwünscht ist, wird der Freilauf-Messmodus gewählt, indem das Bit SD zurückgesetzt wird. In diesem Modus findet die Temperaturmessung kontinuierlich statt. Über das Bit TM kann einer der zwei Thermostat Modi und über das Bit POL die Polarität des Alert Signals im Alarmfall bestimmt werden. Im Komparator Modus ($TM = „0“$) wird der Alert Ausgang auf High geschaltet (für $POL = „1“$) wenn der Temperaturwert die obere Grenztemperatur überschreitet und schaltet ihn auf Low beim nächsten Unterschreiten der unteren Grenze. Im Interrupt Modus ($TM = „1“$) mit $POL = „1“$ wird der Alert Ausgang sowohl beim Überschreiten der oberen Grenze als auch beim Unterschreiten der unteren auf High geschaltet. Nach einer Messperiode oder nach dem Aufruf einer SMBus Alert Response Funktion wird der Ausgang wieder auf Low geschaltet. Über die Bits F1:0 kann die Anzahl der aufeinanderfolgenden Über-/Unterschreitungen, die einen Alarm auslösen nach Tab. 6.32 eingestellt werden. Dadurch kann das Auslösen eines Alarms durch Temperaturrauschen vermieden werden. In Abb. 6.43 ist die Thermosstatfunktion für die Einstellungen $POL = „1“$, $F1 = „0“$ und $F0 = „1“$ graphisch dargestellt.

In einer Schaltung mit mehreren Sensoren, wie in Abb. 6.42 dargestellt, werden die Alert Ausgänge zu einer UND-Verdrahtung zusammengeschaltet. Weil im Komparator

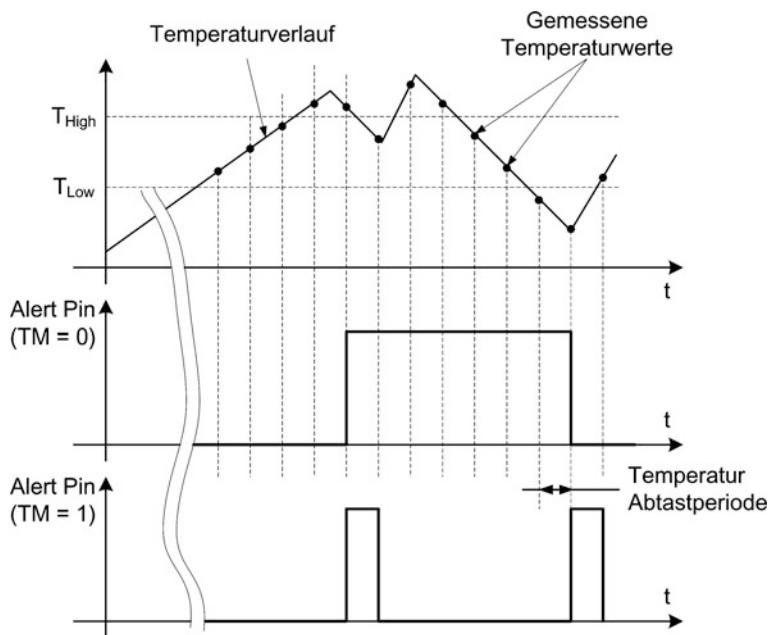


Abb. 6.43 TMP75 Die Thermostat Funktion

Modus aus dem resultierenden Signal der Alarmauslöser nicht identifizierbar ist, muss der Interrupt Modus mit $POL = „0“$ gewählt werden. Mit einer SMBus Alert Response Funktion, deren zeitlichen Verlauf in Abb. 6.44 für IC1 als Auslöser dargestellt ist, kann der Master den Slave als auch das Ereignis, das den Alarm ausgelöst hat, ermitteln. Mit den vorigen Einstellungen wird der Alert Pin beim Eintreten eines einen Alarm auslösenden Ereignisses auf Low geschaltet. Der Master reagiert und initiiert eine I²C Kommunikation. Weiterhin sendet er die reservierte Adresse 000110 und setzt das Read/Write Bit. Diese Alert Response Adresse die im SMBus Protokoll implementiert ist, wird von den TMP75 Sensoren mit ACK bestätigt. Der Slave der den Alarm ausgelöst hat, sendet seine eigene 7 Bit Adresse und mit dem achten Bit codiert das Alarm Ereignis folgendermaßen: mit „0“ das Unterschreiten der unteren Grenztemperatur und mit „1“ das Erreichen, bzw. das Überschreiten der oberen Grenze. Der Master quittiert die Antwort des Slaves mit NACK, beendet die Kommunikation und der Slave deaktiviert seinen Alarmausgang. Wenn zwei oder mehrere Sensoren gleichzeitig einen Alarm auslösen, werden sie versuchen auch gleichzeitig auf die SMBus Alert Response Adresse zu antworten. Der Slave mit der kleinsten Adresse gewinnt die Arbitration, platziert seine Adresse und das auszulösende Ereignis in codierter Form auf der Datenleitung und deaktiviert seinen Alert Ausgang. Das Alert Signal bleibt aber weiter aktiv und der Master muss das Verfahren wiederholen bis alle Sensoren die den Alarm auslösten, sich identifiziert haben.

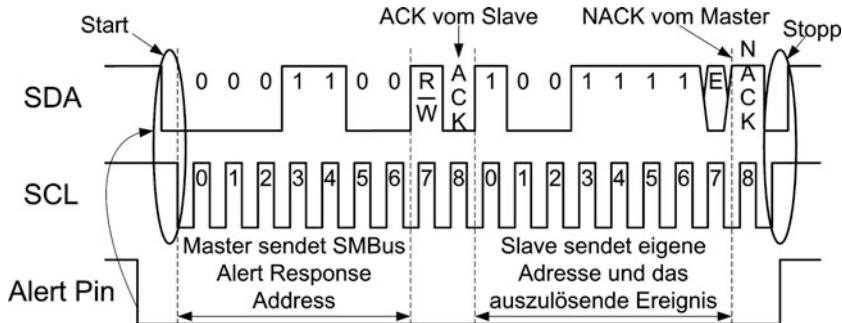


Abb. 6.44 TMP75 SMBus Alert Response Funktion

Folgender Code zeigt die Implementierung des zeitlichen Verlaufs aus der Abb. 6.44. Die Funktion `TMP75_Read_AlarmAddress()` speichert die Antwort eines Sensors in die Variable `ucAlarmAddress`. Über die Schnittstellenfunktion `TMP75_Get_AlarmAddress()` greift das Hauptprogramm auf diese Antwort zu, ohne globale Variablen zu benutzen.

```

uint8_t ucAlarmAddress;
#define SMBUS_ALERT_RESPONSE_ADDRESS 0x19

uint8_t TMP75_Read_AlarmAddress(void)
{
    TWI_Master_Start(); //Start
    if((TWI_STATUS_REGISTER) != TWI_START) return TWI_ERROR;
    // SMBus Alert Response Adresse senden
    TWI_Master_Transmit(SMBUS_ALERT_RESPONSE_ADDRESS);
    if((TWI_STATUS_REGISTER) != TWI_MT_SLA_ACK) return TWI_ERROR;
    //Adresse des Alarm auslösenden Sensors wird eingelesen
    ucAlarmAddress = TWI_Master_Read_NAck();
    if((TWI_STATUS_REGISTER) != TWI_MR_DATA_NACK) return TWI_ERROR;
    TWI_Master_Stop(); //Stop
    return TWI_OK;
}

uint8_t TMP75_Get_AlarmAddress(void)
{
    return ucAlarmAddress;
}

```

Literatur

1. Meroth, A., Tolg, B.: Infotainmentsysteme im Kraftfahrzeug. Vieweg, GWV Fachverlage, Wiesbaden (2008)
2. Roppel, C.: Grundlagen der digitalen Kommunikation. Hanser, München (2006)
3. Borucki, L.: Grundlagen der Digitaltechnik. Teubner, Wiesbaden (1985)
4. Werner, M.: Nachrichten-Übertragungstechnik. Vieweg, GWV Fachverlage, Wiesbaden (2006)
5. von Grüningen, D. Ch : Digitale Signalverarbeitung. Hanser, München (2008)
6. Microchip Technology Inc.: 8-bit AVR® Microcontroller with 4/8/16k bytes in-system programmable flash (2015). www.microchip.com, Zugegriffen: 16. April 2018
7. Microchip Technology Inc.: 8-bit AVR® Instruction set. <http://www.microchip.com>, Zugegriffen: 2. April 2016
8. STMicroelectronics: L3GD20 – MEMS motion sensor: three-axis digital output gyroscope (2015). www.st.com, Zugegriffen: 1. März 2015
9. STMicroelectronics: AN4505 Application Note. L3GD20: 3-axis digital output gyroscope (2015). www.st.com, Zugegriffen: 10. Oktober 2015
10. Freescale Semiconductor. MPL3115A2, I²C Precision Altimeter. Data Sheet Rev 4.0, 09/2015
11. Frescale Semiconductor. Miguel Salhuana Application Note AN4519 – Data Manipulation and Basic Settings of the MPL3115A2 Command Line Interface Driver Code. Rev 0.1 08/2012
12. Freescale Semiconductor. Miguel Salhuana Application Note AN4481 – Sensor I²C Setup and FAQ. Rev 0.1, 07/2012
13. Roedel, W., Wagner, T.: Physik unserer Umwelt: Die Atmosphäre. Springer, Berlin Heidelberg (2011)
14. Silicon Labs. SI7021 – I2C Humidity and temperature sensor (2014). www.silabs.com, Zugegriffen: 30. Dezember 2014
15. Silicon Labs: AN607 – SI70xx – Humindity sensor designer's guide (2014). www.silabs.com, Zugegriffen: 6. Dezember 2014
16. Werner, M.: Information und Codierung. Vieweg + Teubner, Wiesbaden (2008)
17. Maxim Integrated: AN27 – Understanding and using redundancy checks with Maxim 1-wire and iButton Products (2015). www.maximintegrated.com, Zugegriffen: 2. Oktober 2015
18. Honeywell International Inc: HMC5883L – three-axis digital compass IC (2015). www.honeywell.com, Zugegriffen: 20. November 2015
19. Tränkler, H.-R., Reindl, L.M. (Hrsg.): Sensortechnik. Springer Vieweg, Wiesbaden (2014)
20. Knödel, K., Krummel, H., Lange, G. (Hrsg.): Geophysik. Springer, Heidelberg New York (2005)
21. Honeywell International: Michael Caruso application of magnetoresistive sensors in navigation systems (2015). www.honeywell.com, Zugegriffen: 22. Dezember 2015
22. Volder, J.: The CORDIC computing technique. IRE Trans. Electron. Comput. **8**(3), 330–334 (1959)
23. Kuhlmann, M., Kerhab, K.: Parhi P-CORDIC: a precomputation based rotation CORDIC algorithm. EURASIP J. Appl. Signal. Processing **9**, 936–943 (2002)
24. Analog Devices: ADXL312 – digital accelerometer (2015). www.analog.com, Zugegriffen: 27. Oktober 2015
25. Glück, M.: MEMS in der Mikrosystemtechnik. Springer, Berlin (2005)
26. NXP Semiconductors: MMA65xx, dual-axis SPI inertial Sensor (2017). www.nxp.com, , Zugegriffen: 21. Februar 2017
27. Hering, E., Schönfelder, G.: Sensoren in Wissenschaft und Technik. Vieweg +Teubner, Wiesbaden (2012)
28. Silicon Labs: SI1141/42/43 Proximity / Ambient Light Sensor IC with I²C Interface (2016). www.silabs.com, Zugegriffen: 29. März 2016

29. Silicon Labs: AN498 – SI114x Designer’s Guide (2014). www.silabs.com, Zugegriffen: 18. November 2014
30. Silicon Labs: AN580 – infrared gesture sending (2015). www.silabs.com, Zugegriffen: 21. November 2015
31. Texas Instruments: LMP92064 precision low-side, 125-kSps simultaneous sampling, current sensor and voltage monitor with SPI (2015). www.ti.com, Zugegriffen: 2. Juli 2015
32. Parthier, R.: Messtechnik. Vieweg, GWV Fachverlage, Wiesbaden (2008)
33. Texas Instruments: TMP175 – digital temperature sensor with two-wire interface (2014). www.ti.com, Zugegriffen: 27. Oktober 2014

Weiterführende Literatur

34. Bosch Sensortec: BMP180 Digital pressure sensor (2015). www.bosch-sensortec.com
35. Hesse, S., Schell, G.: Sensoren für die Prozess- und Fabrikautomation. Springer Vieweg, Wiesbaden (2014)
36. Honeywell International: AN203 – compass heading using magnetometers (2015). www.honeywell.com, Zugegriffen: 22. Dezember 2015
37. Ultraschall-Modul SRF08. Datenblatt. www.roboter-teile.de/datasheets/srf08. Zugegriffen: 23. Februar 2016
38. Hoffmann, D.W.: Grundlagen der technischen Informatik. Hanser, München (2007)
39. Büchel, G.: Praktische Informatik – eine Einführung. Vieweg + Teubner, Springer, Wiesbaden (2012)
40. Brinkschulte, U., Ungerer, T.: Mikrocontroller und Mikroprozessoren. Springer, Heidelberg Dordrecht London Ney York (2010)

Vernetzbare integrierte Schaltkreise

7

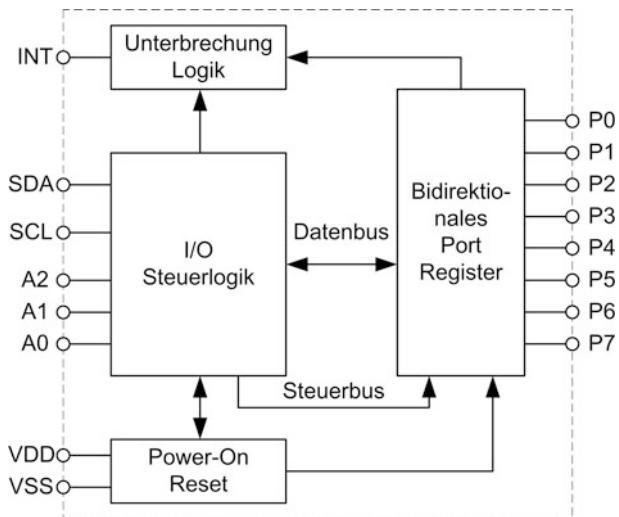
Zusammenfassung

Oftmals reichen die auf einem Chip vorhandenen Ressourcen für eine komplexere Schaltung nicht aus. Dies gilt insbesondere für Flash- und EPROM-Speicher und für die Genauigkeit integrierter DA-Wandler. Das Kapitel stellt wichtige Bausteine vor, die hier über serielle Schnittstellen Abhilfe schaffen. Zum Aufbau spezieller Schaltungen, zum Beispiel analoger Regler, sind digitale Regelwiderstände nötig. Weiterhin macht es bisweilen Sinn, über eine unabhängige Uhr zu verfügen. Auf beides wird in der Folge eingegangen. Schließlich wird mit der Beschreibung der Anbindung eines Radio-IC eine weitere interessante Anwendung vorgestellt.

7.1 PCF8574 Port Expander

In manchen Anwendungen besitzt der steuernde Mikrocontroller zu wenige I/O-Pins um alle Elemente der Schaltung zu bedienen. Eine Möglichkeit, dieses Problem zu lösen, ist die Benutzung eines so genannten Port-Expanders. Der Baustein PCF8574 [1] ist ein solcher Port-Expander Baustein, der die Peripherie eines Mikrocontrollers um einen 8-Bit Port erweitern kann, und benötigt für die Kommunikation lediglich die zwei Leitungen eines I²C-Busses. Der Baustein wird in zwei Ausführungen hergestellt, die bis auf die Device-Typ-Adresse: 0x40 für PCF8574 und 0x70 für PCF8574A identisch sind. Dank der drei Adressanschlüsse können jeweils bis zu 8 gleiche Bausteine von jeder Ausführung am gleichen Bus angeschlossen werden, was einer Peripherieerweiterung eines Mikrocontrollers von bis zu 16 Ports entspricht. Die innere Power-On-Reset Schaltung (siehe Abb. 7.1) sorgt dafür, dass nach dem Hochfahren des Bausteins alle I/O-Pins auf High geschaltet werden. Dieses Verhalten unterscheidet sich von dem der Mikrocontroller aus der ATmega Familie. Für eine passende Ansteuerung der I/O Pins entsprechend der konkreten Beschaltung muss der Mikrocontroller während der Initialisierungsphase sorgen.

Abb. 7.1 PCF8574 Block-schaltbild

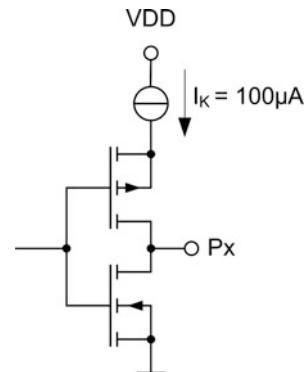


Die Kommunikation mit dem Baustein erfolgt über I²C im Standard-Modus mit bis zu 100 kBit/s.

7.1.1 Endstufe eines I/O Pins

Wenn ein I/O-Pin auf Low geschaltet ist, kann der Transistor T2 in Abb. 7.2 einen Strom von 25 mA zur Masse leiten. Ist der Pin auf High geschaltet, so ist die Stromstärke von der Konstantstromquelle bestimmt (zwischen 30..300 µA). Diese Beschaltung ermöglicht, dass jeder I/O Pin unabhängig von den anderen Pins unter folgenden Voraussetzungen als Ausgang oder Eingang verwendet werden kann:

Abb. 7.2 PCF8574 Endstufe eines I/O Pins



- der minimale Ausgangsstrom im High Zustand muss ausreichend für die Ansteuerung eines Aktors sein;
- bevor ein Pin als Eingang benutzt wird, muss er zuerst auf High geschaltet werden. Bei maximaler Stromstärke der Stromquelle soll der Pegel der Eingangsspannung mit einem angeschlossenen Bauelement am Eingang noch in dem Bereich liegen, in dem er als Low erkannt wird.

7.1.2 Ausgang-Port-Modus

In der Abb. 7.3a ist beispielhaft das Schalten aller I/O-Pins eines PCF8574 Bausteins von Low auf High dargestellt. Der Baustein mit der Device-Typ-Adresse 0x40 hat den Adresseingang A0 auf High geschaltet und die anderen zwei auf Low. Der steuernde Mikrocontroller, der als Master konfiguriert ist, initiiert die Kommunikation mit dem Slave

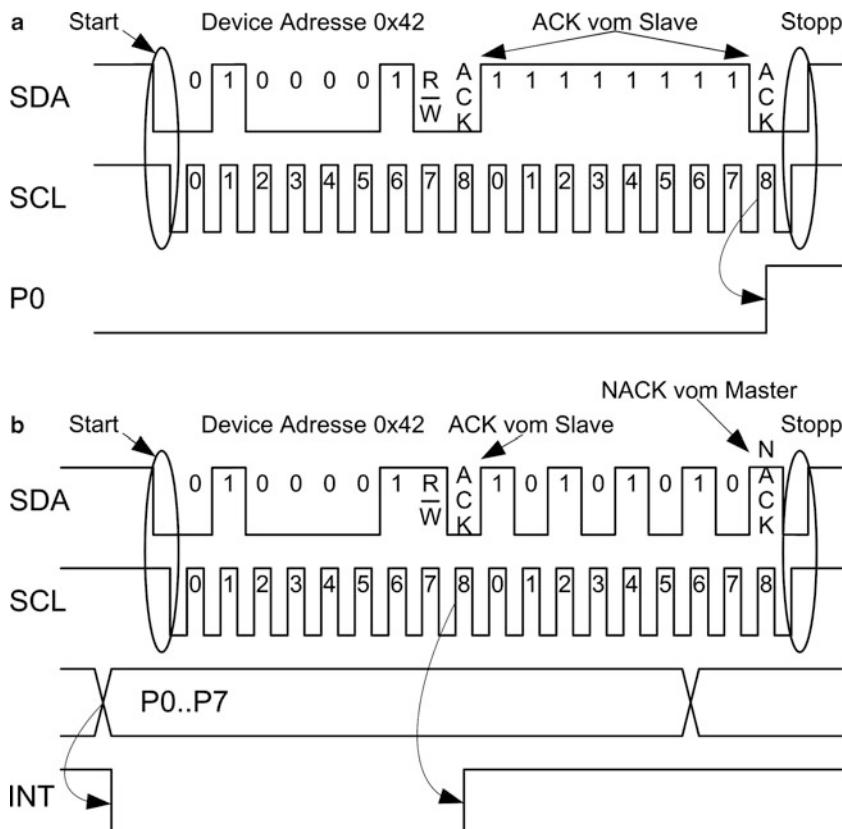


Abb. 7.3 PCF8574 Ein-/Ausgabe eines Bytes

mit einer START-Sequenz und prüft, ob der Bus frei ist. Wenn ja, adressiert er den Slave mit seiner Adresse und dem R/W-Bit auf 0. Die Slave Adresse setzt sich aus der ODER-Verknüpfung der Device-Typ-Adresse und der nach links um eine Stelle verschobenen Device-Chip-Adresse zusammen:

```
ucSlaveAddress = ucDeviceTypeAddress | (ucDeviceChipAddress << 1);
```

Wenn der Slave die empfangene Adresse mit ACK bestätigt, sendet der Master das steuernde Byte 0xFF. Während des ACK-Bits, mit dem der Slave dieses Byte quittiert, wird auf der steigenden Flanke des Taktcs das Byte in das bidirektionale Port-Register gespeichert. Über dieses Register werden die Endstufen der Pins geschaltet. Der Master kann jetzt die Kommunikation mit einer STOP-Sequenz beenden, muss aber nicht und kann anschließend weitere Bytes senden.

Im Folgenden wird eine C-Funktion vorgestellt, deren Ausführung die Ausgabe eines Bytes an einen Port-Expander bewirkt.

```
uint8_t PCF8574_Write_Byte(uint8_t ucdevice_type_address,
                           uint8_t ucdevice_chip_address,
                           uint8_t ucbyte2write)
{
    uint8_t ucDeviceAddress;
    ucDeviceAddress = ucdevice_type_address
                      | (ucdevice_chip_address << 1);
    ucDeviceAddress |= TWI_WRITE;           //Write-Modus
    //Start
    TWI_Master_Start();
    if((TWI_STATUS_REGISTER) != TWI_START)      return TWI_ERROR;
    // Device Adresse senden
    TWI_Master_Transmit(ucDeviceAddress);
    if((TWI_STATUS_REGISTER) != TWI_MT_SLA_ACK) return TWI_ERROR;
    TWI_Master_Transmit(ucbyte2write);
    if((TWI_STATUS_REGISTER) != TWI_MT_SLA_ACK) return TWI_ERROR;
    // Stop
    TWI_Master_Stop();
    return TWI_OK;
}
```

Beim Aufruf der Funktion werden die Device-Typ- und Device-Chip-Adresse, sowie das steuernde Byte als Parameter übergeben. Die Funktion gibt TWI_OK zurück, wenn die Kommunikation fehlerfrei stattgefunden hat. Um die Aufgabe aus dem Beispiel zu lösen, wird die Funktion aufgerufen:

```
PCF8574_Write_Byte(0x40, 0x01, 0xFF);
```

7.1.3 Eingang-Port-Modus

In der Abb. 7.3b sind die zeitlichen Verläufe dargestellt, die beim einmaligen Auslesen des gesamten Ports eines PCF8574-Bausteins entstehen, der wie oben beschrieben beschaltet ist und an dessen Pins P7..P0 die logischen Pegel 10101010 anliegen. Der Slave wird diesmal mit dem R/W-Bit auf 1 adressiert. Während des ACK-Bits, mit dem der Slave die empfangene Adresse bestätigt, werden die logischen Eingangsspegel auf der steigenden Flanke des Taktes im bidirektionalen Port-Register gespeichert und auf der seriellen Datenleitung ausgegeben. Der Master empfängt das Byte, quittiert es mit NACK und beendet die Kommunikation mit einer STOP-Sequenz oder kann mit ACK antworten und ein weiteres Byte empfangen. Mit dem Beenden der Kommunikation wird der Inhalt des Datenregisters gelöscht. Die folgende C-Funktion verdeutlicht das Auslesen eines Erweiterung-Ports, der mit den Adressen *ucdevice_type_address* und *ucdevice_chip_address* angesprochen wird und das ausgelesene Byte an der Adresse *ucbyte2read* speichert.

```
uint8_t PCF8574_Read_Byte(uint8_t ucdevice_type_address,
                           uint8_t ucdevice_chip_address, uint8_t* ucbyte2read)
{
    uint8_t ucDeviceAddress;
    ucDeviceAddress = ucdevice_type_address
                      | (ucdevice_chip_address << 1);
    ucDeviceAddress |= TWI_READ; //Write-Modus
    //Start
    TWI_Master_Start();
    if((TWI_STATUS_REGISTER) != TWI_START) return TWI_ERROR;
    // Device Adresse senden
    TWI_Master_Transmit(ucDeviceAddress);
    if((TWI_STATUS_REGISTER) != TWI_MT_SLA_ACK) return TWI_ERROR;
    *ucbyte2write = TWI_Master_Read_NAck();
    if((TWI_STATUS_REGISTER) != TWI_MR_DATA_NACK) return TWI_ERROR;
    // Stop
    TWI_Master_Stop();
    return TWI_OK;
}
```

Die Bitdauer bei maximaler Taktfrequenz beträgt:

$$t_{\text{Bit}} = \frac{1}{f_{\text{max}}} = \frac{1}{100.000} = 10 \mu\text{s}$$

was bedeutet, dass der zeitliche Abstand zwischen zwei aufeinander folgenden ACK-Bits 90 μs ist. Pegeländerungen, die kürzer als 90 μs sind, und in der Zeit zwischen zwei ACK-Bits geschehen, können nicht erfasst werden, auch wenn der Mikrocontroller kontinuierlich den Port ausliest.

7.1.4 Interrupt-Modus

In einem Mikrocontrollersystem muss die Erfassung der Änderungen aller Eingangspegel gewährleistet werden. Der Baustein PCF8574 besitzt eine digitale Logik, die die Aktivierung eines Ausgangs bei jeder logischen Pegeländerung an einem I/O-Pin ermöglicht. Diese Aktivierung kann dem Mikrocontroller über einen Interrupt signalisieren, dass die Eingangskonstellation sich geändert hat und dieser kann gezielt einen Lesevorgang starten. Dieses Prinzip ähnelt dem Pin-Change-Interrupt eines Mikrocontrollers mit dem Unterschied, dass beim PCF8574 nur durch Vergleich mit dem alten Wert festgestellt werden kann, welcher Eingang sich geändert hat. In Abb. 7.3b ist die Änderung der Eingänge P7...P0, die das Schalten des INT-Ausgangs auf Low verursacht hat, zu sehen. Der Mikrocontroller reagiert und startet das Lesen eines neuen Wertes. Während des ACK-Bits, mit dem der Slave die eigene Adresse bestätigt, wird auf der steigenden Flanke des Taktes der Eingangswert in das bidirektionale Port-Register gespeichert und der Interrupt deaktiviert. Der neue Wert muss am Eingang des Bausteins mindestens 90 µs unverändert anliegen, damit er vom Mikrocontroller erfasst werden kann.

Der INT-Ausgang benutzt als Ausgangsendstufe einen open-drain Transistor, der einen externen Drain-Widerstand (pull-up) benötigt. Mehrere INT-Ausgänge, die parallelgeschaltet sind, benutzen den gleichen Widerstand und ermöglichen einem Mikrocontroller die Überwachung von mehreren Eingangssports. Bei zusammengesetzten INT-Ausgängen vervielfacht sich die Zeit in der die Eingangswerte unverändert bleiben müssen, um jede Änderung erfassen zu können. Um den Auslöser eines Interrupts festzustellen, muss der Mikrocontroller hintereinander alle Bausteine auslesen (pro Baustein 1 Adressbyte + 1 Datenbyte). Ein ausgelöster Interrupt kann nur durch den Zugriff auf den auslösenden Baustein deaktiviert werden. Kurze Impulse können Interrupts auslösen, werden aber nicht immer erfasst.

7.1.5 PCA9534

PCA9534 [2] ist eine Weiterentwicklung des Port Expander PCF8574 mit derselben Pinbelegung. Der Baustein besitzt sogar die gleiche Device-Typ-Adresse 0x40, erfüllt aber die Anforderungen des I²C Fast-Modus und kann Daten mit bis zu 400 kBit/s senden und empfangen. In diesem Fall rechnet man mit einer Bytedauer von ca. 22,5 µs. Intern ist er ähnlich aufgebaut, besitzt aber zusätzlich vier Register, die den Datenfluss steuern.

Das Konfigurationsregister wird über den Befehlscode 0x03 erreicht und ähnlich wie das DDR-Register des Mikrocontrollers ATmegax8 bestimmt es die Datenflussrichtung eines jeden Portanschlusses. Das Zurücksetzen eines Bits in diesem Register schaltet den entsprechenden Portpin auf Ausgang. Beim Hochfahren des Bausteins wird der gesamte Port auf Eingang initialisiert. Über das Output-Register (Befehlscode 0x01) werden die einzelnen Anschlüsse mit „1“ auf High und mit „0“ auf Low gesetzt. Das Setzen eines Bits auf „1“ im Polarity Inversion Register (Befehlscode 0x02) verursacht die bitweise

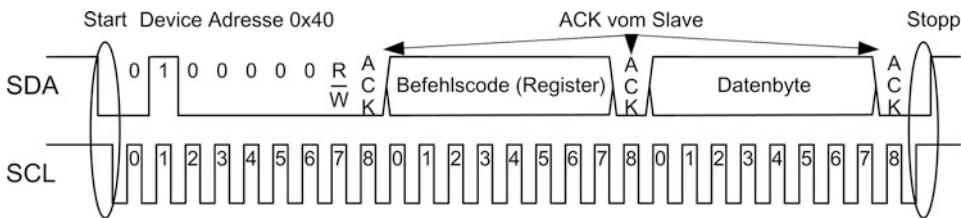


Abb. 7.4 PCA9534 Register schreiben

Invertierung beim Einlesen des entsprechenden Eingangs. Der Inhalt dieser drei Register kann mit einem I²C-Schreibbefehl wie in der Abb. 7.4 geändert, oder mit einem Lesebefehl gelesen werden. Mit dem ACK-Bit wird das Datenbyte in das gewählte Register gespeichert. Weitere Datenbytes können nach dem ersten übertragen werden. Achtung: das Auslesen des Output-Registers liefert den Zustand der Ausgangs-Flip-Flops und nicht die Ausgangspegel! Wenn zuvor in einem Schreibzyklus das Input Port Register ange wählt wurde, verläuft das Einlesen von Daten genau wie beim PCF8574.

7.2 Festwertspeicher

Festwertspeicher sind nichtflüchtige Speicher, in denen Daten zur Laufzeit gespeichert werden, die nach Abschalten oder Reset des Systems persistent gespeichert bleiben. Man unterscheidet auch hier (Kap. 3) zwischen EEPROMs und Flashspeichern. Insbesondere, wenn man größere Datenmengen zur Laufzeit speichern will, beispielsweise in offline-Datenloggern, werden Festwertspeicher als Erweiterung des begrenzten Speicherplatzes eines Mikrocontrollers benötigt.

7.2.1 Parallele Festwertspeicher

Ein parallel ansteuerbarer Festwertspeicher zeichnet sich durch eine hohe Datenrate und eine einfache Ansteuerung aus. Die Beschaltung dieser Speicher ist komplex und die große Anzahl von Pins macht sie teuer und für Mikrocontrollerprojekte kaum anwendbar (siehe Abb. 7.5). In diesem Beispiel wird innerhalb eines Taktzyklus ein gesamtes Datenbyte (8 Datenbits) vom Mikrocontroller zum Speicher oder umgekehrt übertragen. Für die Adressierung einer Speicherzelle werden neben den acht Datenleitungen zusätzlich zehn Adressleitungen und für die Ansteuerung des Bausteins bis zu vier Steuerleitungen benötigt. Bei der Verdoppelung der Speicherkapazität erhöht sich die Anzahl der Adressleitungen um eins.

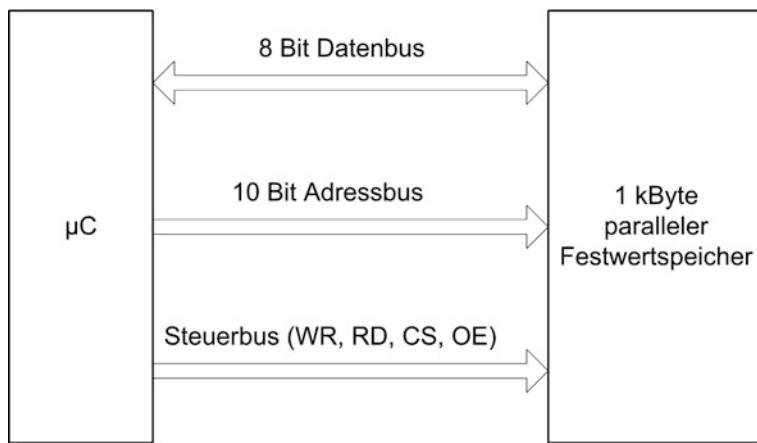


Abb. 7.5 Externer, paralleler Speicher angeschlossen an einem Mikrocontroller

7.2.2 Serielle Festwertspeicher

Die seriellen Festwertspeicher sind im Innern ähnlich den parallelen Speichern aufgebaut, nach Außen aber erfolgt der gesamte Informationsfluss (Daten, Adressierung und Steuerung) über eine serielle Schnittstelle, wie es in Abb. 7.6 zu sehen ist. Die für die

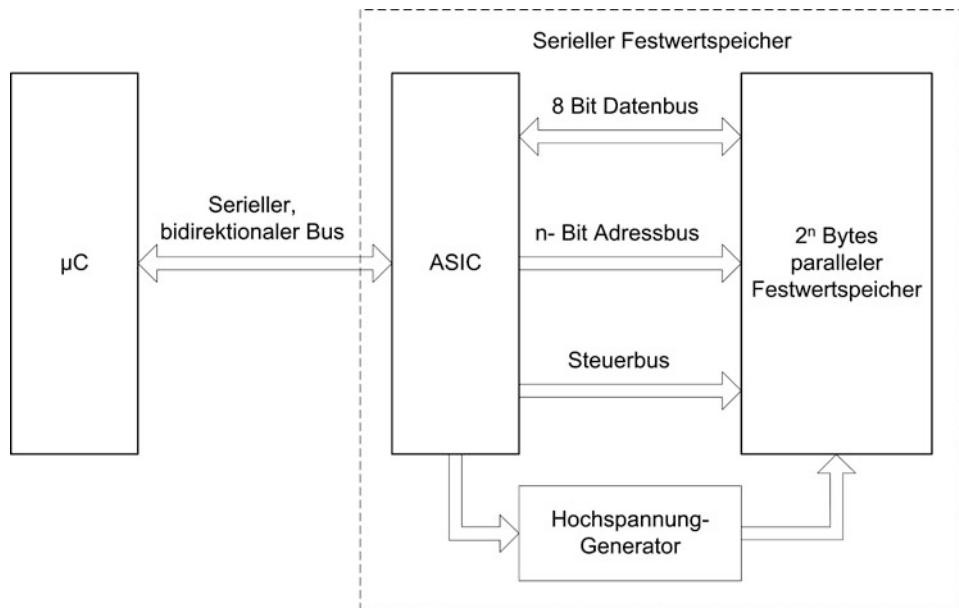


Abb. 7.6 Externer serieller Speicher angeschlossen an einem μC

Ansteuerung der seriellen Speicher meist verwendeten seriellen Schnittstellen sind: SPI, I²C, Microwire und UNI/O (1-Draht Schnittstelle der Fa. Microchip). Der Datenaustausch zwischen Mikrocontroller und einem seriellen Speicher findet bitweise statt. Aus diesem Grund können zu parallelen Speichern vergleichbare Datenraten nur bei viel höheren Taktfrequenzen erreicht werden. Die niedrige Anzahl der für die Ansteuerung notwendigen Pins eines solchen Speichers führt zu einer kleineren Baugröße, zu einem niedrigeren Energieverbrauch und niedrigerem Preis. Deshalb werden sie als externe Festwertspeicher für Mikrocontrolleranwendungen bevorzugt.

7.2.2.1 Serielle EEPROM Speicher

M24C64 – I²C-angesteuerter EEPROM

Als Beispiel für I²C-angesteuerte EEPROMs wird im Folgenden die 24xx Reihe vorgestellt und der Baustein M24C64-R (siehe [3]) näher beschrieben. Diese Speicherreihe wird von den meisten Speicherherstellern mit unterschiedlichen Speicherkapazitäten produziert. Die Bausteine werden über einen I²C-Bus mit Taktfrequenzen von 100 kHz, 400 kHz oder 1 MHz (herstellerabhängig) angesteuert. Ein Baustein vom Typ 24xx00 hat eine Speicherkapazität von 128 Bit (16 Byte), während einer vom Typ 24xx102 1 Mbit (128 kByte) speichern kann. Ab 1 kBit sind die Speicher in so genannten Pages (Speicherseiten) organisiert, die 8, 16 oder 32 Byte groß sein können, abhängig von der Größe des jeweiligen Speichers. Eine Speicherseite ist ein Speicherbereich von aufeinanderfolgenden Bytes, dessen Anfangsadresse durch die Größe der Speicherseite teilbar ist. Die Bausteine dieser Familie können mit Spannungen zwischen 1,7 und 5,5 V versorgt werden, und der Datenerhalt beträgt laut Hersteller mindestens 100 Jahre.

Der Baustein M24C64-R hat eine Kapazität von 64 kBit (8 kByte) und weist einen 8-Bit großen Datenbus auf. Er braucht für die Adressierung der einzelnen Speicherzellen (Bytes) einen internen 13 Bit Adressbus und besitzt eine digitale Logik, die die Speichervorgänge steuert und das I²C-Protokoll implementiert. Die Versorgungsspannung kann im Temperaturbereich von –40 bis 85 °C zwischen 2,5 und 5,5 V variieren, bei den -F und -DF Ausführungen zwischen 1,7 und 5,5 V. Die Ausführung M24C64-D besitzt eine zusätzliche Page (32 Bytes), deren Inhalt dauerhaft schreibgeschützt werden kann.

In Abb. 7.7 ist eine mögliche Schaltung solcher Speicherbausteine dargestellt; in diesem Beispiel bieten die EEPROMs dem Mikrocontroller zusätzlichen nichtflüchtigen Speicher an, um Einstellungen, Kompensationswerte oder umfangreiche Look-Up-Tabellen (LUT) zu speichern. Wegen der hohen Anzahl der Lösch/Programmzyklen (>4 Mio.) können diese Speicher auch in Datenlogger bei relativ niedriger Abtastrate verwendet werden. In dieser Beispielschaltung sind sowohl der Mikrocontroller als auch die Speicherbausteine mit +5 V versorgt, deshalb braucht man keine Pegelanpassung für die I²C-Bus-Signale. Der Write-Control-Eingang ist dauerhaft auf Low geschaltet, so dass der Schreibschutz deaktiviert ist.

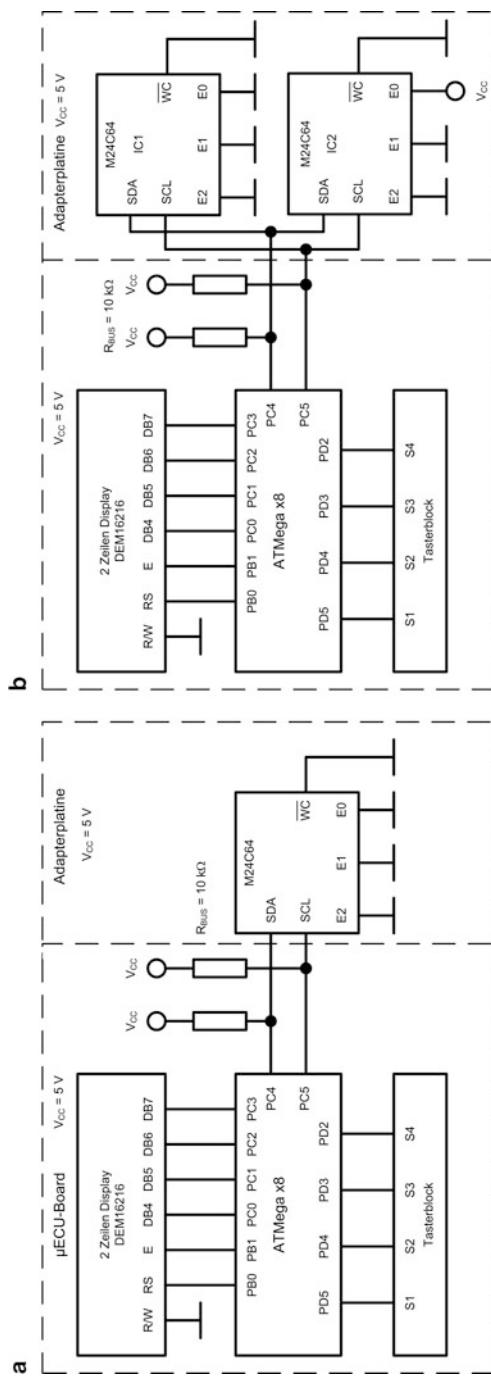


Abb. 7.7 Speichererweiterung eines Mikrocontrollers mit seriellen EEPROMs

I²C Kommunikation

Der Speicherbaustein ist als Slave konfiguriert und wartet im Stromsparmodus auf eine Kommunikation mit einem Master. Die Kommunikation kann im Standard-Modus (100 kHz), Fast-Modus (400 kHz) oder Fast-Modus-Plus (1 MHz) stattfinden. Die 7-Bit Device-Typ-Adresse, mit der die Speicher der Reihe 24C64 adressierbar sind, lautet 0xA0. Für die Ansteuerung besitzt der Baustein einen Anschluss für die bidirektionale Datenübertragung (SDA), einen Takteingang (SCL), einen Write-Control-Eingang, der den gesamten Speicherbereich schreibschützen kann, wenn er auf High geschaltet ist und drei Device-Chip-Adresseingänge (E0, E1 und E2). Über diese drei Adresseingänge können bis zu 8 Speicher von diesem Typ an dem gleichen Bus angeschlossen werden. Der Mikrocontroller unterscheidet zwei oder mehrere Speicher, die am gleichen Bus angeschlossen sind über ihre Device-Chip-Adresse. Während IC1 (Abb. 7.7) alle Adresseingänge auf Low geschaltet hat und somit die Adresse 0xA0 besitzt, ist der Eingang E0 vom IC2 auf High geschaltet und das führt zur Adresse 0xA2. Der große Taktfrequenzbereich und die Adressauswahl ermöglichen eine leichte Vernetzung mit anderen I²C-Bausteinen.

Eine externe Initialisierung des Bausteins ist nicht vorgesehen. Bevor der Master die Kommunikation mit dem Slave initiiert, müssen die Device-Chip-Adresseingänge des Speichers auf den gewünschten Pegel geschaltet sein, falls sie nicht fest verdrahtet sind.

Der Mikrocontroller beginnt die Kommunikation mit einer I²C-START-Sequenz, gefolgt von der Adressierungsphase. In dieser Phase wird die Adresse des Bausteins übertragen. Wenn die empfangene Adresse mit der eigenen übereinstimmt, antwortet der Slave auf dem 9. Takt mit ACK, ansonsten schaltet der Baustein in den Standby-Modus und wird erst nach einer weiteren START-Sequenz wieder aktiv. Nach der Adressierungsphase folgt die Kommunikationsphase, in der Daten in einer Richtung übertragen werden. Der Master beendet die Kommunikation mit einer I²C-STOP-Sequenz, gibt den Bus wieder frei und der Slave schaltet in den Stromsparmodus.

Lesen

Der interne Adresszähler wird nach dem Einschalten zurückgesetzt. Vor einem Schreib-/Lesevorgang wird er auf die gewünschte Anfangsadresse gesetzt und mit jedem gelesenen/gespeicherten Byte inkrementiert. Die Speicherorganisation ermöglicht einen direkten (wahlfreien) oder sequentiellen Zugriff auf die Speicherzellen. Die Lesefunktionen können auch dann ausgeführt werden, wenn der Write-Control-Eingang auf High geschaltet ist. Folgende Lesefunktionen sind implementiert:

Sequentielles Lesen ab der aktuellen Adresse

Um ein oder mehrere Bytes ab der aktuellen Adresse zu lesen, werden folgende Schritte durchgeführt:

- Schritt 1: der Master initiiert die I²C Kommunikation mit einer START-Sequenz;
- Schritt 2: Wenn der Bus frei ist, sendet der Master die Slave Adresse mit dem R/W Bit auf 1;

- Schritt 3: der Slave bestätigt mit ACK den Empfang der eigenen Adresse;
- Schritt 4: der Slave sendet mit den nächsten 8 Takten das aktuell adressierte Byte und inkrementiert den Adresszähler;
- Schritt 5: wenn der Master den Empfang des Bytes mit ACK bestätigt, wird der Slave ein weiteres Byte senden und Schritt 4 wird ausgeführt. Wenn der Master keine weiteren Bytes fordert, antwortet er mit NACK und es wird der Schritt 6 durchgeführt.
- Schritt 6: der Master beendet die Kommunikation mit einer STOP-Sequenz.

Sequentielles Lesen mit direktem Zugriff

Ein oder mehrere Bytes können auch ab einer frei gewählten Adresse gelesen werden. Dafür muss der Master die Kommunikation starten und wenn der Bus frei ist, die Slave Adresse senden, wobei das R/W Bit auf „Schreiben“ gesetzt ist. Wenn der Slave den Empfang der Adresse mit ACK bestätigt, sendet der Master die gewünschte 16-Bit-Adresse, mit dem höherwertigen Byte zuerst. Die zwei Bytes, die vom Slave mit ACK quittiert wurden, werden in den Adresszähler gespeichert und der Lesevorgang kann beginnen. Mit einer START-Sequenz findet ein Neustart der Kommunikation statt. Ab jetzt wird das Lesen mit den Schritten 2 bis 6 vom „Lesen ab der aktuellen Adresse“.

Das sequentielle Lesen ermöglicht das Lesen einer beliebigen Zahl von Bytes. Wenn der Adresszähler das Speicherende erreicht hat, springt er nach Inkrementieren auf die erste Adresse.

Speichern

Mit dem Write Control Eingang kann der Speicher gegen unbeabsichtigtes Speichern geschützt werden, wenn er auf High geschaltet ist. Wenn dieser Eingang nicht fest verdrahtet ist, muss er vor einem Schreibvorgang auf Low geschaltet werden. Es können bis zu 32 Bytes in einem Schreibzyklus gespeichert werden, vorausgesetzt sie befinden sich alle auf der gleichen Speicherseite. Die Anfangsadresse kann frei gewählt werden. Das Speichern wird in mehreren Schritten durchgeführt:

- Schritt 1: der Master initiiert mit einer START-Sequenz die Kommunikation mit dem Slave;
- Schritt 2: wenn der Bus frei ist, sendet der Master die Slave Adresse mit dem R/W-Bit auf Low gesetzt;
- Schritt 3: wenn ein interner Speichervorgang noch läuft, antwortet der Slave mit NACK; ansonsten quittiert er den Empfang der Adresse mit ACK;
- Schritt 4: der Master sendet das höherwertige gefolgt vom niederwertigen Adressbyte; der Slave bestätigt die zwei Bytes mit ACK und speichert sie in den Adresszähler;
- Schritt 5: der Master sendet das Byte, das gespeichert werden soll;
- Schritt 6: wenn der Write-Control-Eingang auf High geschaltet ist, antwortet der Slave mit NACK, ansonsten mit ACK und inkrementiert den Adresszähler der Speicherseite; wenn das Ende der Speicherseite erreicht wurde, so springt dieser Adresszähler zur ersten Adresse der Seite;

- Schritt 7: wenn ein weiteres Byte gespeichert werden soll, dann weiter mit Schritt 5 ansonsten mit Schritt 8;
- Schritt 8: der Master beendet den Vorgang mit einer STOP-Sequenz.

Die empfangenen Bytes werden in einen 32 Byte großen, flüchtigen Speicher und von dort, nach der I²C-STOP-Sequenz, in den EEPROM-Hauptspeicher gespeichert. Während des internen zeitgesteuerten Schreibvorgangs, der unabhängig von der zu speichernden Anzahl von Bytes bis zu 5 ms dauert, schaltet sich der Baustein vom Bus ab und ein Adressierungsversuch wird mit NACK quittiert.

Ein Flussdiagramm des oben beschriebenen Speichervorgangs ist in der Abb. 7.8 zu sehen und der Funktionscode ist weiter unten aufgelistet.

Die Funktion soll eine Zahl *ucbyte_number* von Bytes aus einem RAM-Array, dessen Adresse *ucbyte_array* ist, an den mit der Device-Chip-Adresse *ucdevice_address* identifizierten 24C64 Speicherbaustein übertragen und angefangen mit der *uibyte_address* Adresse abspeichern. Um eine gesamte Seite zu speichern, muss die Anfangsadresse der Seite berechnet und als *uibyte_address* übertragen werden. Die Funktion gibt zurück:

- bei fehlerfreier Durchführung: TWI_OK (=0x00);
- bei fehlerhafter Datenübertragung: _24C64_WRITE_PROTECTED (=0x80);
- bei fehlerhafter Übertragung der Adresse oder der Anfangsadresse des Speicherbereichs: TWI_ERROR (=0x01).

Die aufrufende Stelle der Funktion muss entsprechend des Rückgabewertes entsprechend reagieren.

```
uint8_t _24C64_Write_Page(uint8_t ucdevice_address,
                           uint16_t uibyte_address,
                           volatile uint8_t* ucbyte_array,
                           uint8_t ucbyte_number)
{
    uint8_t ucDeviceAddress, ucAddressByteHigh, ucAddressByteLow, ucI;
    //die Byteadresse wird in seinem High- und Lowbyte zerlegt
    ucAddressByteLow = uibyte_address;
    ucAddressByteHigh = uibyte_address >> 8;
    //die Adresse des 24C64-Speicherbausteins wird gebildet
    ucDeviceAddress = (ucdevice_address << 1) | _24C64_ADDRESS;
    ucDeviceAddress |= TWI_WRITE; //Write-Modus
    TWI_Master_Start(); //Start
    if((TWI_STATUS_REGISTER) != TWI_START) return TWI_ERROR;
    // Device Adresse senden
    TWI_Master_Transmit(ucDeviceAddress);
    if((TWI_STATUS_REGISTER) != TWI_MT_SLA_ACK) return TWI_ERROR;
```

```

//das höherwertige Byte der Byteadresse wird gesendet
TWI_Master_Transmit(ucAddressByteHigh);
if((TWI_STATUS_REGISTER) != TWI_MT_DATA_ACK) return TWI_ERROR;
//das niederwertige Byte der Byteadresse wird gesendet
TWI_Master_Transmit(ucAddressByteLow);
if((TWI_STATUS_REGISTER) != TWI_MT_DATA_ACK) return TWI_ERROR;
for(ucI = 0; ucI < ucbyte_number; ucI++)
{
    //ein Datenbyte wird gesendet
    TWI_Master_Transmit(ucbyte_array[ucI]);
    if(TWI_STATUS_REGISTER == TWI_MT_DATA_NACK)
        return _24C64_WRITE_PROTECTED;
}
TWI_Master_Stop(); // Stop
return TWI_OK;
}

```

Folgender Funktionsaufruf speichert 32 Bytes aus dem Array ucBuffer[32] auf der 32. Speicherseite (Byte 1024:1055) von IC1 (s. Abb. 7.7):

```
_24C64_Write_Page(0x00, 1024, ucBuffer, 32);
```

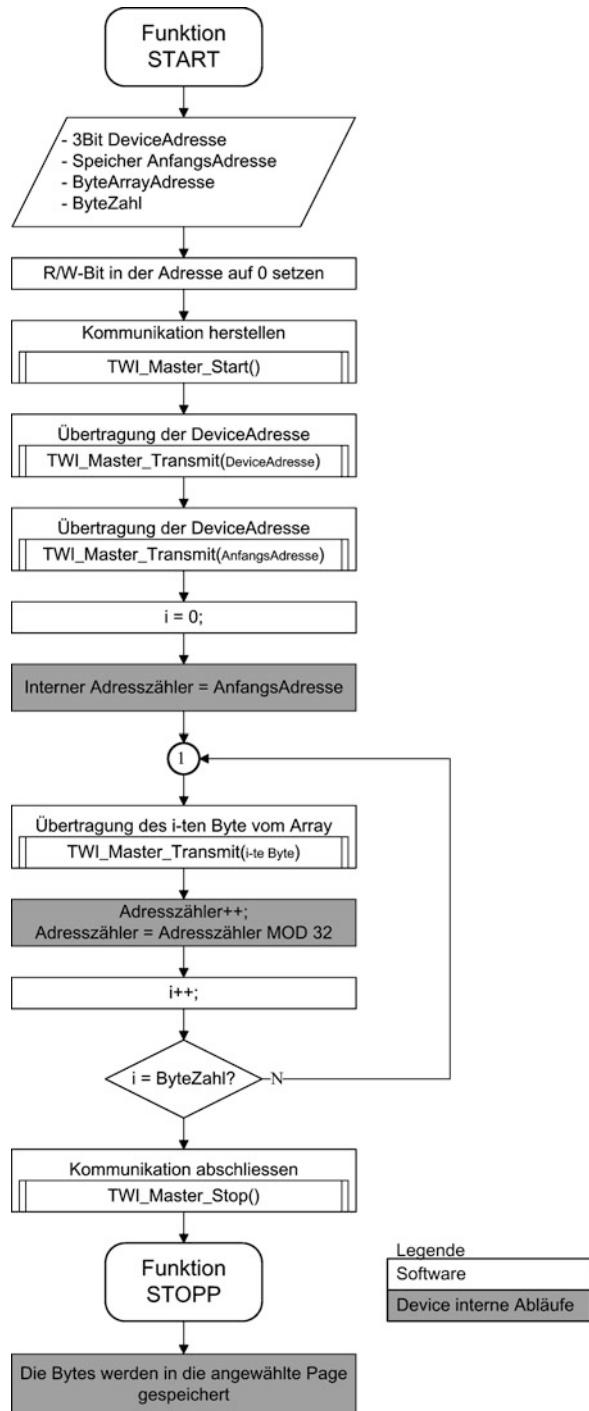
Testbarkeit

Die elektrische Verbindung zwischen dem Mikrocontroller und dem Speicherbaustein wird schon in der Adressierungsphase getestet. Wenn der Baustein beim Empfang seiner Adresse nicht mit Acknowledge antwortet, dann ist entweder die Adresse falsch, oder die elektrische Verbindung nicht in Ordnung, oder er speichert gerade Daten in EEPROM. Das Ergebnis eines Schreibvorgangs kann getestet werden, indem man den Speicher ausliest und die gesendeten mit den ausgelesenen Daten vergleicht.

Ein Schreibzyklus dauert maximal 5 ms. Während dieser Zeit ist der Baustein nicht adressierbar. Um ein blockierendes Warten nach einem Schreibzyklus zu umgehen, muss man den nächsten Schreibversuch um die maximale Speicherzeit verschieben oder durch Polling den Speicherfortschritt prüfen. Ein Flussdiagramm einer Funktion, die den Speicherfortschritt prüft, ist in Abb. 7.9 dargestellt und der Code weiter unten aufgelistet.

Um zu prüfen, ob der letzte Schreibvorgang abgeschlossen ist, initiiert der Master die I²C Kommunikation mit einer START-Sequenz und sendet die Adresse des zu testenden Speichers. Wenn die Adresse mit ACK quittiert wird, kann ein weiterer Schreibvorgang gestartet werden, ansonsten gibt der Master mit einer START-STOP-Sequenz den Bus wieder frei.

Abb. 7.8 Flussdiagramm
Funktion PageWrite für den
M24C64 Speicherbaustein



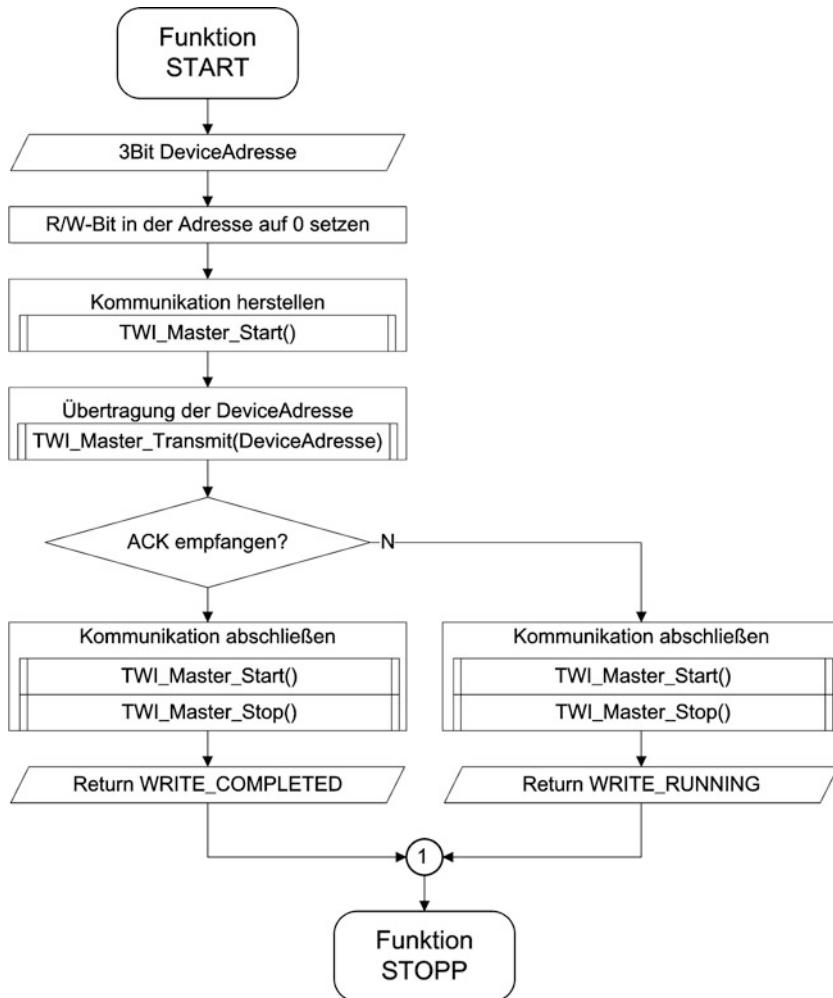


Abb. 7.9 Flussdiagramm Schreibzustand Abfrage für den M24C64 Speicherbaustein

```

uint8_t M24C64_Get_WriteState(uint8_t ucdevice_address)
{
    uint8_t ucDeviceAddress;
    //Adresse des 24C64-Speicherbausteins bilden
    ucDeviceAddress = (ucdevice_address << 1) | _24C64_ADDRESS;
    ucDeviceAddress |= TWI_WRITE; //Write-Modus
    //Start
    TWI_Master_Start();
    if((TWI_STATUS_REGISTER) != TWI_START) return TWI_ERROR;
    // Device Adresse senden
  
```

```

TWI_Master_Transmit(ucDeviceAddress);
if((TWI_STATUS_REGISTER) != TWI_MR_SLA_ACK)
{
    TWI_Master_Start();
    TWI_Master_Stop();
    return WRITE_RUNNING;
}
else if((TWI_STATUS_REGISTER) == TWI_MR_SLA_ACK)
{
    //Slave hat mit ACK geantwortet und ist bereit weitere Bytes
    //zu empfangen, der Master beendet die Kommunikation
    TWI_Master_Start();
    TWI_Master_Stop();
    return WRITE_COMPLETED;
}
}

```

Der folgende Programmcode zeigt für die Beispielschaltung aus Abb. 7.7b das byteweise Speichern im Polling-Betrieb. Die Bytes werden, angefangen mit der Adresse uiAddress, gespeichert. Der Mikrocontroller liest je ein Byte aus dem Array ucBuffer[], überprüft die Schreiberlaubnis und speichert das Byte in den Speicher IC2 an der Zieladresse:

```

if(M24C64_Get_WriteState(0x01) == WRITE_COMPLETED)
{
    _24C64_Write_Byte(0x01, uiAddress+ucIndex, ucBuffer[ucIndex] )
    ucIndex++;
}

```

25LC256 SPI-angesteuerte EEPROMs

SPI-angesteuerte EEPROMs werden von den meisten Herstellern unter der Reihe 25xx (oder 95xx [4]) mit Speicherkapazitäten ab 1 kBit (128 Byte) bis 1 Mbit (128 kByte) angeboten. Das Blockschaltbild eines solchen Speichers ist in Abb. 7.10 zu sehen. Als Beispiel wird im Folgenden der Speicherbaustein 25LC256 [5] vorgestellt. Der Baustein, der eine Speicherkapazität von 265 kBit (32 kByte) besitzt, ist in 64 Byte große Seiten (Pages) organisiert und kann mit Spannungen zwischen 2,5 und 5,5 V (bei einigen Ausführungen wie beispielsweise 25AA256 ab 1,8 V) versorgt werden. Die serielle Kommunikation erfolgt über einen SPI-Bus, der im Modus 0 oder 3 angesteuert wird. Der Bus kann mit bis zu 10 MHz (oder 20 MHz [4]) getaktet werden. Das höherwertige Bit eines Bytes wird zuerst übertragen. Der Speichervorgang eines Bytes oder einer gesamten Page dauert maximal 5 ms, der Datenerhalt beträgt 100 bis 200 Jahre. Die Endurance¹ liegt bei 1.000.000 Lösch-/Schreibzyklen (bei [4] bis zu 4.000.000).

¹ Frei übersetzt: Lebensdauer.

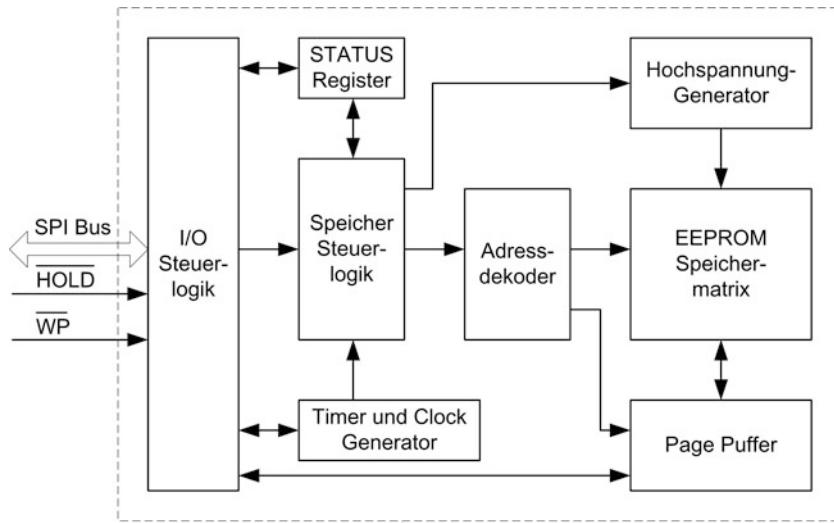


Abb. 7.10 25LC256 SPI-Speicher Blockschaubild

Schreibschutz des Speichers

Über das nichtflüchtige Status-Register des Speichers kann man:

- den Schreibschutz unterschiedlicher Speicherbereiche ändern,
- die Schreibschutzkonfiguration auslesen,
- feststellen ob ein Schreibvorgang läuft.

Die Konfiguration des Status-Registers sieht folgendermaßen aus:

Bit 7 – WPEN (Write Protect Enable) – zusammen mit dem Write-Protect-Eingang dient dem Schreibschutz des Status-Registers (siehe Tab. 7.2);

Bit 3:2 – BP1:BP0 (Block Protection) – diese 2 Bits zeigen an, welcher Speicherbereich gerade schreibgeschützt ist (siehe Tab. 7.1);

Bit 1 – BP1:BP0 (Write Enable Latch) – so lange dieses Bit auf Low steht, ist das Speichern gesperrt; das Speichern wird mit dem Aufruf der Funktion Write Latch Enable freigegeben. Dieses Bit wird immer auf Low gesetzt:

Tab. 7.1 25LC256 Schreibgeschützte Speicherbereiche

BP1	BP0	Schreibgeschützter Speicherbereich
0	0	Keiner
0	1	0x6000 – 0x7FFF
1	0	0x4000 – 0x7FFF
1	1	0x0000 – 0x7FFF

Tab. 7.2 25LC256 Schreibschutz Mechanismen (x = keine Bedeutung)

WEL	WPEN	WP-Pin	Schreibgeschützte Blocks	Ungeschützte Blocks	Status Register
0	X	X	Geschützt	Geschützt	Geschützt
1	0	X	Geschützt	Ungeschützt	Ungeschützt
1	1	Low	Geschützt	Ungeschützt	Geschützt
1	1	High	Geschützt	Ungeschützt	Ungeschützt

- während der Einschaltphase,
- nach dem erfolgreichen Ausführen aller Schreibfunktionen: Write Byte Array, Write Latch Disable oder Write Status Register. Dieses Bit muss vor jedem Schreibvorgang softwaremäßig auf High gesetzt werden um das Schreiben freizugeben.

Bit 0 – BP1:BP0 (Write-In-Proces) – dieses Bit ist von der internen Logik auf High gesetzt, solange ein interner Speicherzyklus aktiv ist;

Die Bits 4, 5 und 6 sind nicht belegt.

Die interne Logik des Bausteins zusammen mit dem Write-Protect-Pin implementieren unterschiedliche Schreibschutz-Mechanismen gegen das unbeabsichtigte Ändern des Speicherinhalts, so wie in Tab. 7.2 dargestellt.

Wenn das Bit WEL im Status Register auf Low geschaltet ist, oder die Bits WEL und WPEN auf High geschaltet sind und der Write Protect Pin auf Low, dann ist das gesamte Status-Register schreibgeschützt und keine weiteren Änderungen können vorgenommen werden. Während man im ersten Fall den Schreibschutz softwaremäßig durch den Aufruf der Funktion Write Latch Enable aufheben kann, ist dies im zweiten Fall nur möglich, wenn man den WP-Pin auf High schaltet.

Schreib-Lese-Funktionen

Die interne Logik des Bausteins implementiert folgende Funktionen (siehe Abb. 7.11):

- **Read Byte Array** (siehe Abb. 7.11e): Lesen einer beliebigen Anzahl von Bytes ab einer wahlfreien Adresse; nach dem Lesen eines Bytes wird der interne Adresszähler inkrementiert. Wenn das letzte Byte ausgelesen wurde (Adresse 0x7FFF), springt der Adresszähler auf die erste Adresse (0x0000).
- **Write Byte Array** (siehe Abb. 7.11f): Speichern eines Bytearrays ab einer wahlfreien Adresse; die Byte-Zahl ist auf die Pagegröße begrenzt. Nach jedem gespeicherten Byte wird der interne Adresszähler inkrementiert. Wenn die oberste Adresse der Page erreicht ist, werden die übrigen Bytes ab der ersten Adresse der Page gespeichert.
- **Write Latch Enable** (siehe Abb. 7.11a): der Write Enable Latch wird auf High gesetzt; dadurch wird das Speichern freigegeben (siehe Tab. 7.2).
- **Write Latch Disable** (siehe Abb. 7.11b): Rücksetzen des Write Enable Latches; das Speichern wird dadurch gesperrt.

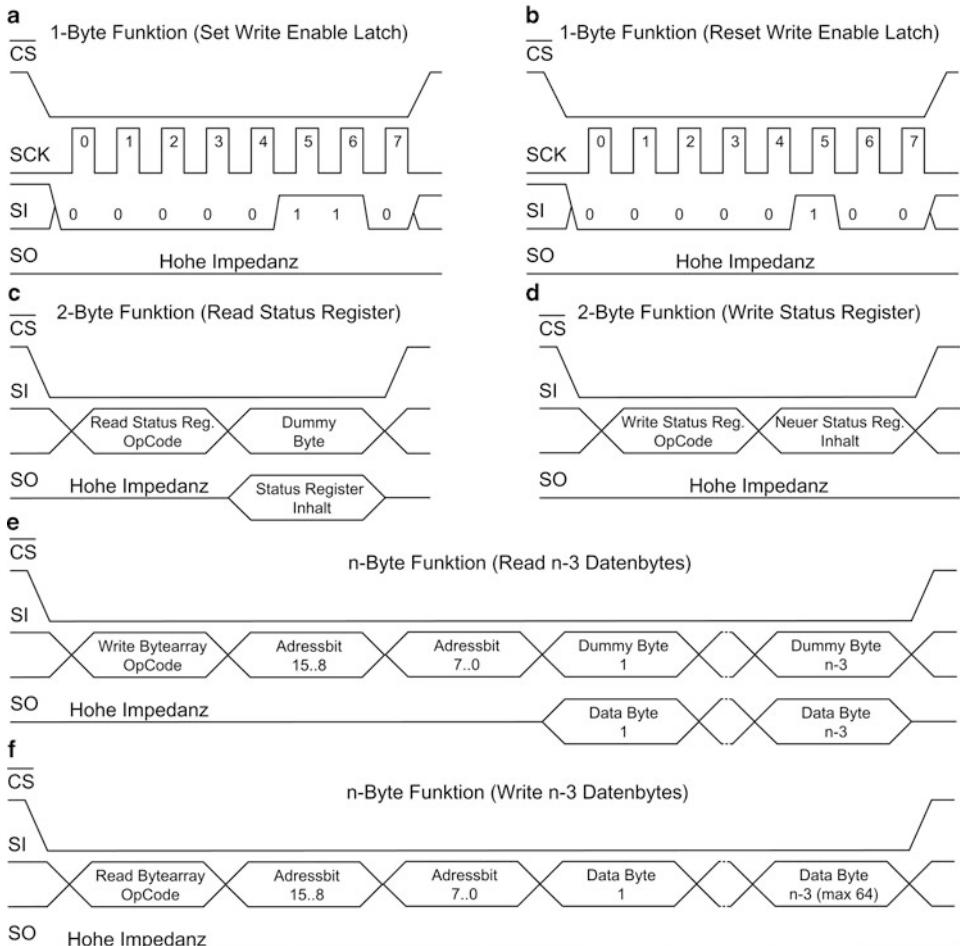


Abb. 7.11 Überblick über die Funktionen des 25LC256

- **Read Status Register** (siehe Abb. 7.11c): Das Status-Register wird ausgelesen.
- **Write Status Register** (siehe Abb. 7.11d): Der Inhalt des Status-Registers wird geändert (nicht die WIP und WEL Bits).

Speicher-Ansteuerung

Eine mögliche Beschaltung des 25LC256 Speichers ist in Abb. 7.12 zu sehen. Über die dedizierten SPI-Anschlüsse MOSI (SI), MISO (SO), Clock (SCK) und Chip Select (CS) findet die Kommunikation zwischen dem Mikrocontroller als Master und dem Speicher als Slave statt. Der Write-Protect-Eingang kann für den Schreibschutz des Status-Registers eingesetzt werden. Wenn dieser Schreibschutz nicht genutzt wird, um einen I/O-Pin des Mikrocontrollers zu sparen, wird dieser Eingang mit der Versorgungsspannung V_{CC} ver-

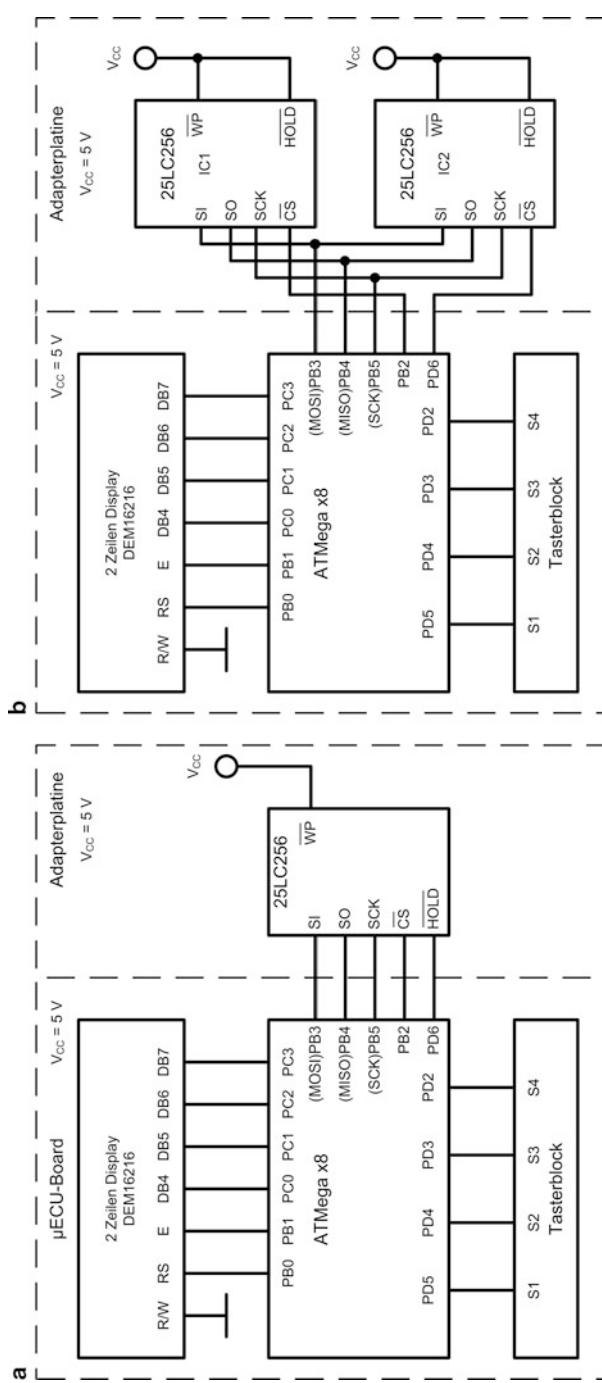


Abb. 7.12 Schaltung eines 25LC256 Speichers

bunden. Mit dem Schalten des HOLD-Eingangs auf Low wird die Datenkommunikation nach der vollständigen Übertragung eines Bytes vorübergehend unterbrochen. Während dieser Unterbrechung wird von dem Baustein weder die Bitfolge über die SI-Leitung noch der Takt über die SCK-Leitung wahrgenommen (auch wenn der Chip-Select-Eingang weiterhin auf Low liegt). Im zweiten Beispiel (Abb. 7.12b) werden zwei Speicherbausteine vom Typ 25LC256 am gleichen SPI-Bus eines Mikrocontrollers angeschlossen. Für diese Bustopologie wird für jeden Busteilnehmer eine Chip-Select-Leitung gebraucht.

Initialisierung der SPI-Schnittstelle des Mikrocontrollers

Bei der Initialisierung der SPI-Schnittstelle des Mikrocontrollers muss man die Hinweise aus [6] und [7] beachten. Für die Mikrocontroller der Reihe ATmegax8 ist PB2 der dedizierte Slave Select Anschluss der SPI-Schnittstelle. Wenn der Mikrocontroller als Master und PB2 auf Eingang konfiguriert sind, schaltet die Schnittstelle intern auf den Slave-Modus um, sobald dieser Pin auf Low steht. Um das zu vermeiden, soll die Initialisierung folgendermaßen realisiert werden:

- PB2 auf Ausgang schalten und auf High setzen.
- Über das Register SPCR wird die SPI-Schnittstelle für die Kommunikation mit dem Speicherbaustein 25LC256 folgendermaßen konfiguriert:
 - der Master-Modus wird gewählt (Bit MSTR auf „1“ gesetzt);
 - die SPI-Schnittstelle wird freigegeben (Bit SPE auf „1“ gesetzt);
 - mit CPOL und CPHA auf „0“, wird der Übertragungsmodus 0 gewählt;
 - das Bit DORD wird zurückgesetzt, um das höherwertige Byte zuerst zu übertragen
 - über die Bits SPR0, SPR1 und SPI2X aus dem Register SPSR wird die Bitrate bestimmt;
- wenn man die Kommunikation im Interrupt-Modus steuern möchte, so ist auch das Bit SPIE im Register SPCR auf „1“ zu setzen, bevor das Interrupt-Flag gelöscht wird (Auslesen des Registers SPCR gefolgt vom Auslesen des Registers SPDR).

Die Gestaltung eines Programms nach dem Polling-Verfahren ist einfacher, führt aber zum blockierenden Warten des Mikrocontrollers. Für die Ansteuerung der Bausteine nach dem Interrupt-Verfahren, müssen für alle Funktionen Zustandsautomaten implementiert werden. Das Programm für die Ansteuerung der in Abb. 7.12 dargestellten Speicherbausteine könnte mit einer Mischung der zwei Verfahren realisiert werden. Abhängig von der konkreten Anwendung bestimmt man, welche Funktionen nach welchem Verfahren aufgerufen werden.

Ein Flussdiagramm für die Gestaltung der Endlosschleife nach beiden Verfahren ist in Abb. 7.13 zu sehen. Die Verwaltung der Interrupt-Funktionen wird mit Hilfe eines Zustandsautomaten in der Interrupt Service Routine der SPI-Schnittstelle implementiert.

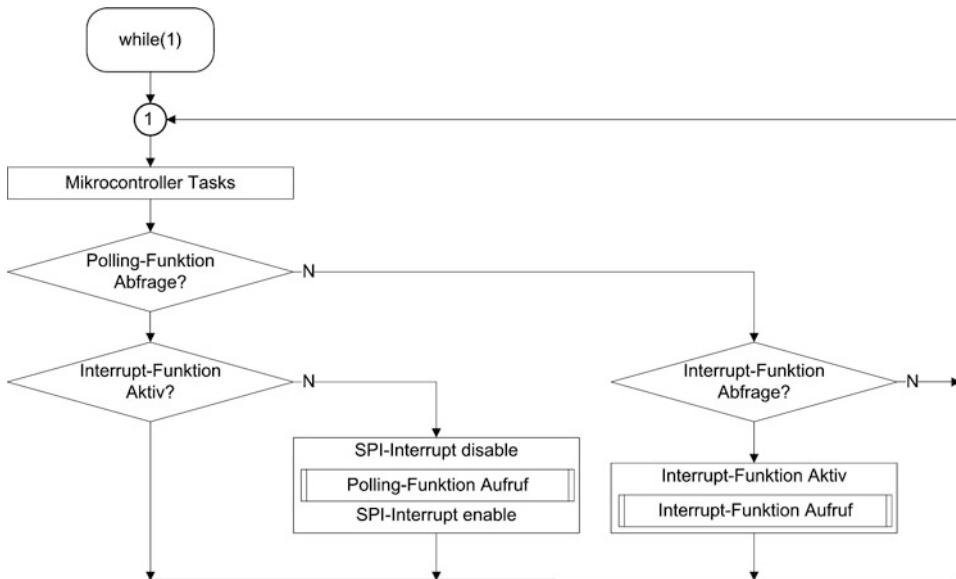


Abb. 7.13 Flussdiagramm: Ansteuerung von 25LC256 Speicherbausteine nach einer Mischung von Polling- und Interrupt-Verfahren

Softwarebeispiel

Als Beispiel wird eine Read-Bytarray-Funktion vorgeschlagen die nach dem Polling-Verfahren implementiert ist. Die Funktion ist Teil eines Softwaremoduls, das die Ansteuerfunktionen für die Speicherbausteine vom Typ 25LC256 beinhaltet. Die Funktionen des Moduls können aus der *main*-Funktion aufgerufen werden, sie greifen auf Funktionen eines SPI-Moduls zu, die im Kap. 5 aufgelistet sind. Die Datenstruktur *_25LC256_pins* ist analog zu den Strukturen in Abschn. 6.1.5 aufgebaut.

```

void _25LC256_Read_ByteArray(_25LC256_pins sdevice_pins, uint16_t
uubyte_address, volatile uint8_t* ucbyte_array, uint16_t uibyte_number)
{
    unsigned char ucDummy;
    unsigned int uiIndex = 0;

    //Chip Select Leitung wird auf Low gesetzt, SPI-Kommunikation
    //wird gestartet
    SPI_Master_Start(sdevice_pins._25LC256spi);
    //der Befehlscode wird übertragen: READ_BYTE_ARRAY_OPCODE = 0x03
    SPI_Master_Write(READ_BYTE_ARRAY_OPCODE);
    //das höherwertige Byte der Adresse wird ermittelt
    ucDummy = uibyte_address >> 8;
    //das höherwertige Byte der Adresse wird übertragen
    SPI_Master_Write(ucDummy);
}

```

```

//das niedwertige Byte der Adresse wird ermittelt
ucDummy = uibyte_address;
//das niedwertige Byte der Adresse wird übertragen
SPI_Master_Write(ucDummy);
//uibyte_number Bytes aus dem adressierten Speicher werden zum
//Zielpuffer ucbyte_array übertragen
for(uiIndex = 0; uiIndex < uibyte_number; uiIndex++)
{
    ucbyte_array[uiIndex] = SPI_Master_Write(ucDummy);
}
//SPI-Kommunikation wird beendet, Chip Select Leitung wird auf
//High gesetzt
SPI_Master_Stop(sdevice_pins._25LC256spi);
}

```

Die Gestaltung des Moduls erlaubt seine unveränderte Benutzung (Portierung) bei unterschiedlichen Anschlusskonfigurationen der Bausteine bzw. bei der Ansteuerung mit unterschiedlichen Mikrocontrollern. Damit das korrekt funktioniert, muss die Datenstruktur `_25LC256_pins` (Abschn. 6.1.5) mit der Pinkonfiguration in der *main*-Datei angepasst werden. Im Folgenden werden die Datenstrukturen mit der Pinkonfiguration der in der Abb. 7.12 vorgestellten Speicherbausteine erläutert:

```

/*typedef struct
{
    tspiHandle _25LC256spi;

    volatile uint8_t* WP_DDR;
    volatile uint8_t* WP_PORT;
    uint8_t WP_pin;
    uint8_t WP_state;

    volatile uint8_t* HOLD_DDR;
    volatile uint8_t* HOLD_PORT;
    uint8_t HOLD_pin;
    uint8_t HOLD_state;
} _25LC256_pins;*/ //diese Struktur wird in der Headerdatei
                    //des Moduls definiert

```

```
#define ON      1
#define OFF     0
```

Für den IC1 rechts im Bild lautet diese Datenstruktur:
`_25LC256_pins _25LC256_1 = {{&DDRB, &PORTB, PB2, ON},`
`OFF, OFF, OFF, OFF,`
`OFF, OFF, OFF, OFF};`

Und für den IC2 rechts im Bild:

```
_25LC256_pins _25LC256_2 = {{&DDRD, &PORTD, PD6, ON},
                                OFF, OFF, OFF, OFF,
                                OFF, OFF, OFF};
```

7.2.2.2 AT45DB161 serieller Flashspeicher

Die Firma Adesto Technologies stellt Flash-Speicher mit einer Speicherkapazität ab 2 Mbit bis 64 Mbit her. Diese Speicherbausteine werden über einen seriellen SPI-Bus bei Taktfrequenzen von bis zu 104 MHz angesteuert und können in Applikationen mit niedriger Versorgungsspannung, wo Platz gespart und auf den Energieverbrauch geachtet werden muss, eingesetzt werden. Sie weisen eine Wortbreite von 8 Bit (1 Byte) auf und werden mit Spannungen ab 2,3 bis 3,6 V versorgt. Aus dieser Spannung wird mit einer sogenannten „Ladungspumpe“ die Hochspannung erzeugt, mit der man den Speicher programmieren oder löschen kann. Als Beispiel für diese Speicherreihe wird ein AT45DB161 [8] näher betrachtet, dessen Blockschaltbild in Abb. 7.14 zu sehen ist. Der Speicher ist vom Werk in 4096 Speicherseiten (engl. pages) mit einer Größe von 528 Byte organisiert und besitzt zwei 528 Byte große SRAM Zwischenspeicher. Die Seiten- und Zwischen-

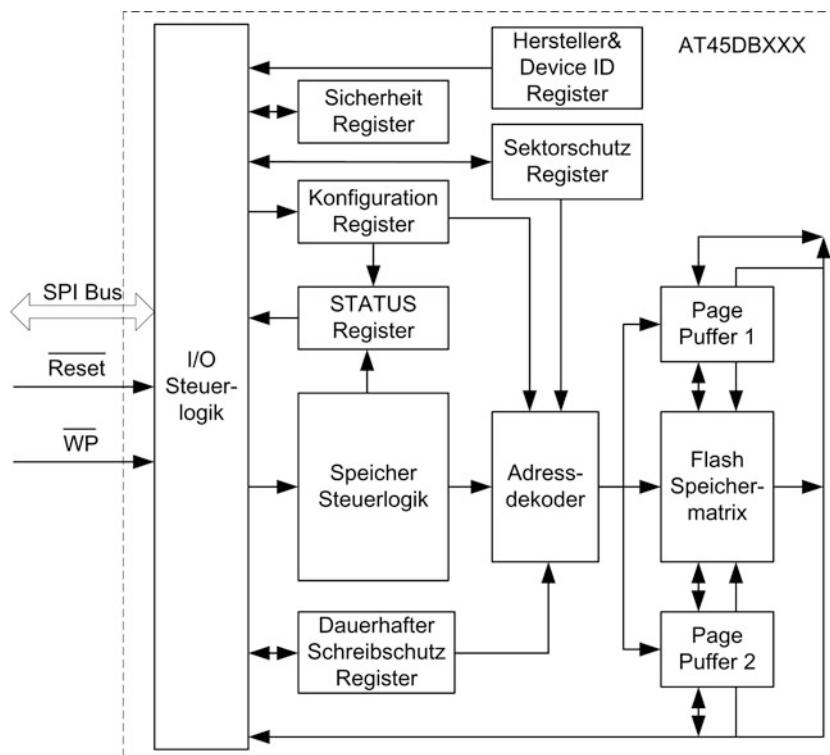


Abb. 7.14 AT45DBXXX-Blockschaltbild

speichergröße können über das Konfigurationsregister auf 512 (= 2^9) Bytes umgestellt werden. Es gibt unterschiedliche Lesefunktionen des Haupt- und Zwischenspeichers bei niedrigen (50 MHz) beziehungsweise hohen (85...104 MHz) Taktfrequenzen. Im Folgenden werden nur die Funktionen bei niedrigen Taktfrequenzen für eine Seitengröße von 512 Byte erwähnt.

SPI-Kommunikation

Der Mikrocontroller muss als Master konfiguriert werden, die Bytes werden im Modus 0 oder 3 mit dem höchstwertigen Bit zuerst übertragen. Weil der Master mit 5 V und der Speicher mit 3,3 V versorgt ist, muss der Pegel der Steuersignale angepasst werden. In diesem Beispiel ist die Pegelanpassung mit Spannungsteilern realisiert, was eine zuverlässige Übertragung bei einer Busfrequenz von ca. 4,5 MHz ermöglicht. Für den Dateneingang vom Mikrocontroller wird keine Pegelanpassung benötigt.

Die SPI-Datenstruktur des Bausteins AT45DB161_pins kodiert in die Steueranschlüsse Chip Select, Write Protect und Reset. Sie setzt sich analog zur Datenstruktur aus dem vorhergehenden Softwarebeispiel zum 25LC256 zusammen und ist für die Beschaltung aus Abb. 7.15 folgendermaßen definiert (Abschn. 6.1.5):

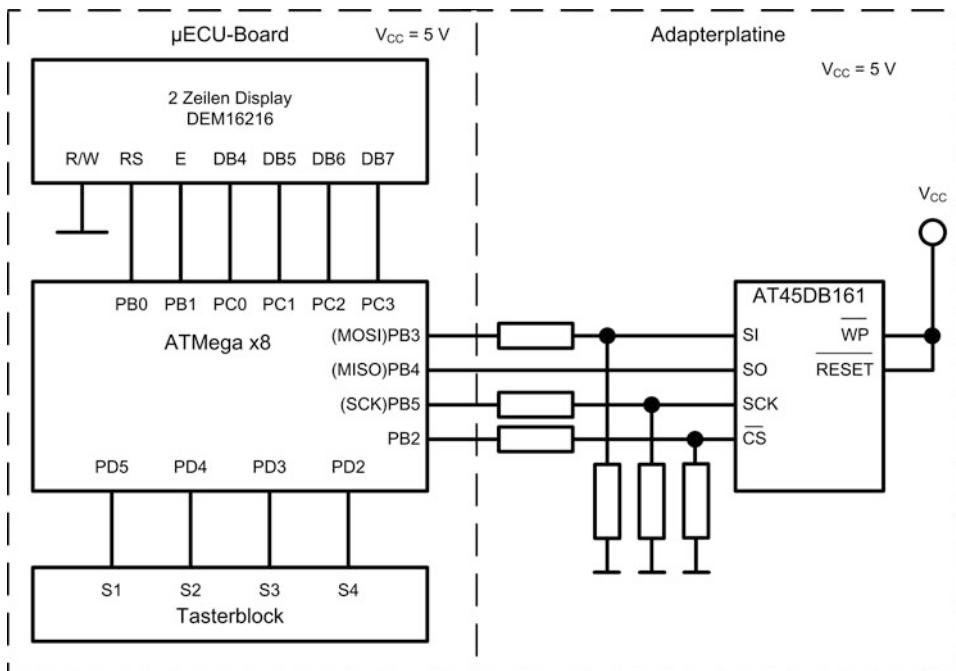


Abb. 7.15 AT45DB161 Beschaltung

```
AT45DB161_pins AT45DB161_1 ={{/*CS_DDR*/
    &DDRB,
/*CS_PORT*/
    &PORTB,
/*CS_pin*/
    PB2,
/*CS_state*/
    ON},
/*WP_DDR*/
    OFF,
/*WP_PORT*/
    OFF,
/*WP_pin*/
    OFF,
/*WP_state*/
    OFF,
/*RESET_DDR*/
    OFF,
/*RESET_PORT*/
    OFF,
/*RESET_pin*/
    OFF,
/*RESET_state*/
    OFF};
```

SRAM-Zwischenspeicher

Die Zwischenspeicher des Bausteins können vom ansteuernden Mikrocontroller direkt angesprochen werden, um Daten zu lesen oder zu speichern und sie können auch als RAM-Erweiterung dienen. Die Anfangsadresse beider Speicher ist 0. Die einfach aufgebauten Zugriffsfunktionen sowie die Architektur des Bausteins ermöglichen die Übertragung einer unbegrenzten Anzahl von Bytes innerhalb einer SPI-Nachricht in beiden Richtungen. Nachdem der steuernde Mikrocontroller das Chip Select Signal auf Low geschaltet hat, überträgt er den Befehlscode, mit dem sowohl ein Zwischenspeicher als auch die Übertragungsrichtung (Tab. 7.3) gewählt wird. Weiterhin wird die Anfangsadresse des Speicherbereiches als 24-Bit-Zahl übertragen, die im internen Adresszähler gespeichert wird. Mit jedem übertragenen Byte wird der interne Adresszähler inkrementiert. Nach der letzten Adresse springt der Adresszähler ohne Verzögerung auf die erste Adresse zurück.

Tab. 7.3 Zugriffsfunktionen auf den Zwischenspeicher

Funktion	Befehlscode
Zwischenspeicher 1 Lesen (50 MHz)	0xD1
Zwischenspeicher 2 Lesen (50 MHz)	0xD3
Zwischenspeicher 1 Speichern	0x84
Zwischenspeicher 2 Speichern	0x87

Um `uibyte_number` Byte aus dem RAM-Puffer `ucssource_buffer` des Mikrocontrollers in einen SRAM-Zwischenspeicher ab der Anfangsadresse `uibuffer_address` speichern zu können, kann folgende Funktion verwendet werden:

```
void AT45DB161_Write_Buffer(AT45DB161_pins sdevice_pins,
                           uint8_t ucop_code,
                           uint16_t uibuffer_address,
                           uint16_t uibyte_number,
                           uint8_t* ucsource_buffer)
{
    uint8_t ucAddressHighByte = 0x00, ucAddressMiddleByte,
    ucAddressLowByte;
```

```

//die 32-Bit Adresse wird in einzelnen Bytes zerlegt
ucAddressLowByte = uibuffer_address;
uibuffer_address = uibuffer_address >> 8;
ucAddressMiddleByte = uibuffer_address;
SPI_Master_Start(sdevice_pins.AT45DB161spi); //die Kommunikation
                                                //wird gestartet
SPI_Master_Write(ucop_code); //der Befehlscode wird übermittelt
//die 24 Bit Adresse wird übertragen
SPI_Master_Write(ucAddressHighByte);
SPI_Master_Write(ucAddressMiddleByte);
SPI_Master_Write(ucAddressLowByte);
//uibyte_number Bytes werden übertragen
for(uiI = 0; uiI < uibyte_number; uiI++)
{
    SPI_Master_Write(ucsouce_buffer[uiI]);
}
SPI_Master_Stop(sdevice_pins.AT45DB161spi);
}

```

Mit dem Aufruf:

```
AT45DB161_Write_Buffer(AT45DB161_1, 0x84, 0, 10, ucBuffer);
```

würde man für die Beispielbeschaltung aus Abb. 7.15, 10 Bytes aus der Variable ucBuffer[] in den ersten SRAM-Zwischenspeicher ab Adresse 0 speichern.

Flash-Hauptspeicher

Der Baustein AT45DB161 ist ein 16-Mbit (2-Mbyte) großer Speicher mit einem Datenerhalt von 20 Jahren und 100.000 Schreib-/Löschenzyklen für jede Seite. Die komplexe Speicherarchitektur des Bausteins ist in Abb. 7.16 dargestellt. 8 Seiten bilden einen Block, 32 Blöcke bilden einen Sektor. Insgesamt sind es 17 Sektoren, weil Sektor 0 in 0a und 0b eingeteilt ist. Bei der Verwendung einer Seitengröße von 528 Byte stehen 16 Byte/Seite zusätzlich zur Verfügung, die Berechnung der logischen Adresse für manche Funktionen wird aber aufwändiger. Die Umschaltung auf die binäre $2^9 = 512$ Bytes Größe erfolgt mit der Übertragung des Befehlscodes 0x3D 2A 80 A6. Die Wiederherstellung der Werkseinstellung (528 Bytes/Seite) wird mit der Übertragung des Codes 0x3D 2A 80 A7 realisiert.

Lesen

Der Mikrocontroller als Master kann Daten aus dem Speicherbereich direkt (ohne Zwischenspeichern) oder indirekt (mit Zwischenspeichern) auslesen.

Lesen aus dem gesamten Speicherbereich

Der Master startet die Kommunikation mit dem Slave durch das Umschalten des Chip Select Signals von High auf Low. Mit dem Übertragen des Befehlscodes 0x03 gefolgt

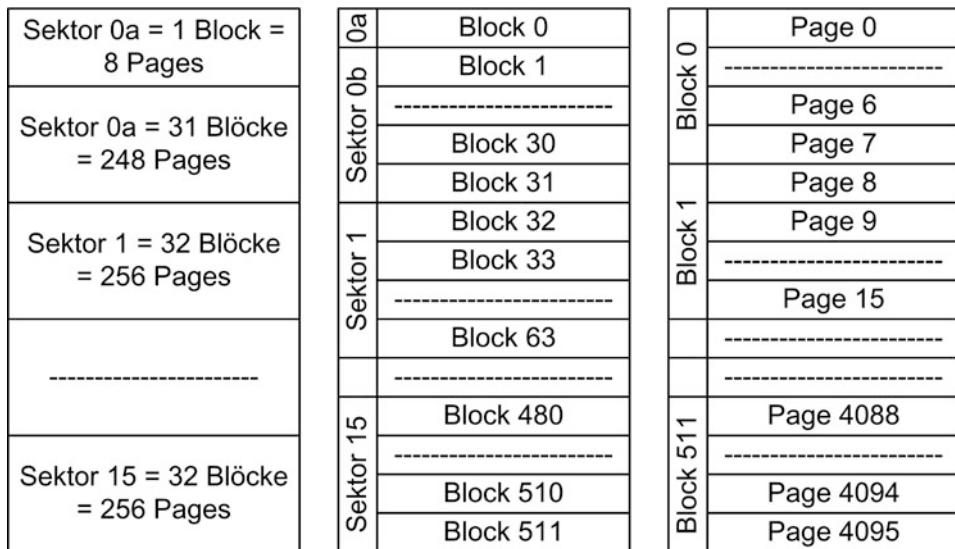


Abb. 7.16 AT45DB161 Speicherarchitektur

von einer gültigen 24-Bit Adresse aus dem Adressbereich des Bausteins, wird der interne Adresszähler des Hauptspeichers mit der Adresse geladen. Diese Funktion kann bei einer Taktfrequenz von bis zu 50 MHz stattfinden. Mit den nächsten 8 Taktten schiebt der Slave den Inhalt der adressierten Speicherzelle über den Datenausgang heraus. Der Adresszähler wird mit jedem ausgelesenen Byte inkrementiert. Solange der Master den Takt weiterhin generiert, wird der Baustein die gespeicherten Daten ausgeben. Wenn der Adresszähler das Ende einer Speicherseite erreicht, so springt er ohne Verzögerung zur ersten Adresse der nächsten Seite. Wird die letzte Adresse des gesamten Speichers erreicht, so springt der Adresszähler zur ersten Adresse. Mit dem Schalten des Chip Select Signals auf High wird das Lesen beendet. Die Inhalte der zwei Zwischenspeicher werden während des Lesevorgangs nicht geändert.

Mit dem Befehlscode *0x01* (Lesen aus dem gesamten Speicherbereich im Low-Power-Modus) werden die Daten ähnlich wie bei der vorigen Funktion bei einem Bustakt von bis zu 15 MHz gelesen.

Mit der folgenden Funktion können aus einem Speicherbaustein ab der Adresse *laddress_begin*, *uibyte_number* Bytes im Low Power Modus ausgelesen und in das Array *ucbuffer* im RAM-Speicher des Mikrocontrollers gespeichert werden.

```

void AT45DB161_Read_MemoryArray(AT45DB161_pins sdevice_pins,
                                 long laddress_begin, uint16_t uibyte_number, uint8_t* ucbuffer)
{
    uint8_t ucAddressHighByte, ucAddressMiddleByte, ucAddressLowByte;
    uint8_t ucDummy = 0;

```

```

//die 32-Bit Adresse wird in einzelnen Bytes zerlegt
ucAddressLowByte = laddress_begin;
laddress_begin = laddress_begin >> 8;
ucAddressMiddleByte = laddress_begin;
laddress_begin = laddress_begin >> 8;
ucAddressHighByte = laddress_begin;
SPI_Master_Start(sdevice_pins.AT45DB161spi); //Start der
                                              //Kommunikation
SPI_Master_Write(0x03); //Übertragung des Befehlscodes
//die 24 Bit Adresse wird übertragen
SPI_Master_Write(ucAddressHighByte);
SPI_Master_Write(ucAddressMiddleByte);
SPI_Master_Write(ucAddressLowByte);
//uibyte_number Bytes werden ausgelesen
for((long)lI = 0; lI < uibyte_number; lI++)
{
    ucbuffer[lI] = SPI_Master_Write(ucDummy);
}
SPI_Master_Stop(sdevice_pins.AT45DB161spi); //die Kommunikation
                                              //wird beendet
}

```

Mit dem folgenden Aufruf würde man aus dem Speicher aus der Beispielschaltung ab Adresse 1024 insgesamt 256 Bytes auslesen und in das Array ucBuffer[] speichern.

```
AT45DB161_Read_MemoryArray(AT45DB161_1, 1024, 0x100, ucBuffer);
```

Bei der Übertragung eines Bytes über die SPI-Schnittstelle findet einen Byteaustausch zwischen dem Master und dem Slave statt. Diese Besonderheit ermöglicht, dass die Lese- und Schreib-Funktionen ähnlich aufgebaut werden.

Lesen innerhalb einer Speicherseite

Ähnlich wie beim Auslesen aus dem gesamten Speicherbereich muss der Master die Kommunikation starten. Mit der Übertragung des BefehlsCodes *0xD2*, gefolgt von einer gültigen 24-Bit-Adresse, wird der interne Adresszähler geladen; vier folgende Dummybytes sorgen für die Initialisierung des Vorgangs und ab dieser Folge werden die Daten ausgegeben solange der Master den Takt generiert. Mit jedem übertragenen Byte wird der Adresszähler inkrementiert, jedoch innerhalb der adressierten Seite. Wenn das Ende der adressierten Seite erreicht wird, springt der Adresszähler zur ersten Adresse der gleichen Seite. Mit dem Schalten des Chip Select Signals auf High durch den Master wird die Übertragung beendet. Die Inhalte der Zwischenspeicher werden dadurch nicht geändert.

Laden einer Speicherseite in einem Zwischenspeicher

Dieser Vorgang ermöglicht das Laden einer gesamten Speicherseite in den über den Befehlscode ausgewählten Zwischenspeicher. Der Mikrocontroller kann auf diese Daten zugreifen, indem er den Zwischenspeicher ausliest (indirektes Lesen). Nach dem Start der Kommunikation sendet der Master den Befehlscode $0x53$ für den ersten Zwischenspeicher oder $0x55$ für den zweiten, gefolgt von der 24-Bit-Anfangsadresse der gewünschten Seite. Mit dem Schalten des Chip Select Signals auf High wird die SPI-Kommunikation beendet und es beginnt die bausteinintern zeitgesteuerte Übertragung von der Speicherseite zum Zwischenspeicher, die nach $200\text{ }\mu\text{s}$ beendet ist. Der Mikrocontroller kann nur indirekt auf die gespeicherten Daten zugreifen. Im ersten Schritt wird der Zwischenspeicher mit den Daten aus der gewünschten Seite geladen und im zweiten werden sie daraus mit der Zwischenspeicher-Lesefunktion ausgelesen. Während der internen Übertragung kann der Mikrocontroller Daten in den anderen Zwischenspeicher speichern.

Speichern

Das fehlerfreie Speichern von Daten kann nur in zuvor gelöschten Bereichen stattfinden. Beim AT45DB161 können im Flash nur Daten aus den Zwischenspeichern gespeichert werden. Das Speichern ist ein intern zeitgesteuerter Vorgang mit wechselnder Dauer (Tab. 7.4), währenddessen das Bit RDY/BUSY aus dem STATUS Register auf „0“ geschaltet ist. Während des Speicherns kann der andere Puffer Daten vom Mikrocontroller empfangen. Falls beim Speichern Fehler auftreten, wird das Bit EPE im STATUS Register auf „1“ gesetzt. Es sind unterschiedliche Speichervorgänge implementiert, die im Folgenden kurz erläutert werden.

Puffer in Flashseite speichern mit Löschen

Bei diesem Vorgang wird der gesamte Inhalt eines Puffers in eine Speicherseite gespeichert. Die zu speichernde Daten müssen vorher in den Puffer übertragen werden. Der ansteuernde Mikrocontroller startet die SPI-Kommunikation und überträgt den Befehlscode $0x83$ um den Inhalt des Puffers 1 ($0x86$ für den Puffer 2) zu speichern, gefolgt von der 24-Bit Anfangsadresse der gewünschten Seite. Mit dem Beenden der Kommunikation beginnt der interne Vorgang, die adressierte Seite wird zuerst gelöscht und danach werden die Daten aus dem Puffer gespeichert.

Tab. 7.4 AT45DB161 Speicherzeiten

Speicherfunktion	Speicherdauer
Puffer in Flashseite speichern mit Löschen	< 25 ms
Puffer in Flashseite speichern ohne Löschen	< 4 ms
Daten über den Puffer in Flashseite speichern mit Löschen	< 25 ms
Daten über den Puffer in Flashseite speichern ohne Löschen	$8\text{ }\mu\text{s}/\text{Byte}$

Puffer in Flashseite speichern ohne Löschen

Dieser Vorgang ist ähnlich dem oben Beschriebenen, der Puffer 1 wird über den Code `0x88` und der Puffer 2 über `0x89` adressiert. In diesem Fall muss die Speicherseite vorher gelöscht sein. Im Folgenden wird eine Funktion, die den Inhalt eines Puffers in eine Speicherseite speichert, vorgestellt. Der Puffer wird über den Befehlscode ausgewählt und die Speicherseite über den Parameter `uipage_address` adressiert. Abhängig von dem gewählten Befehlscode soll die Speicherseite vorher gelöscht werden. Eine gültige Seitenadresse ist eine Zahl zwischen 0 und 4095.

```
void AT45DB161_Write_MemoryPageFromBuffer(AT45DB161_pins sdevice_pins,
                                            uint8_t ucop_code,
                                            uint16_t uipage_address)
{
    uint8_t ucAddressHighByte, ucAddressMiddleByte,
    ucAddressLowByte = 0x00;
    //die 32-Bit Adresse wird in einzelnen Bytes zerlegt
    uipage_address = uipage_address << 1;
    ucAddressMiddleByte = uipage_address;
    uipage_address = uipage_address >> 8;
    ucAddressHighByte = uipage_address;
    SPI_Master_Start(sdevice_pins.AT45DB161spi); //SPI-Kommunikation
                                                    //wird gestartet
    SPI_Master_Write(ucop_code); //der Befehlskode wird übertragen
    //die Anfangsadresse des gewünschten Speicherbereiches
    //wird übertragen
    SPI_Master_Write(ucAddressHighByte);
    SPI_Master_Write(ucAddressMiddleByte);
    SPI_Master_Write(ucAddressLowByte);
    //die SPI-Kommunikation wird gestoppt
    SPI_Master_Stop(sdevice_pins.AT45DB161spi);
}
```

Für die Beispielschaltung würde man die nachfolgende Funktion aufrufen um den Inhalt von Zwischenspeicher 1 in Seite 5 (Speicherbereich 2048...2559) ohne Löschen zu speichern:

```
AT45DB161_Write_MemoryPageFromBuffer(AT45DB161_1, 0x88, 5);
```

Vergleich zwischen gespeicherte Seite und Quellpuffer

Um zu prüfen ob das Speichern eines gesamten Puffers im Flash fehlerfrei stattgefunden hat, kann man nach dem Vorgangsabschluss die zwei Inhalte vergleichen. Wenn die zwei Inhalte übereinstimmen, dann wird das Bit COMP im Status-Register auf „0“ gesetzt. Dafür muss der Master über den SPI-Bus den Befehlscode `0x60`, gefolgt von einer 24-Bit-Anfangsadresse der gewünschten Seite senden, um den Inhalt dieser Seite mit dem Puffer 1 zu vergleichen (der Puffer 2 wird über den Code `0x61` adressiert).

Daten über den Puffer in Flashseite speichern mit Löschen

Dieser Vorgang ist eine Kombination zwischen dem Schreibvorgang eines Zwischenspeichers und der Speicherfunktion „Puffer in Flashseite speichern mit Löschen“. Der Mikrocontroller startet die SPI-Kommunikation und wählt einen der Zwischenspeicher durch das Übertragen eines Befehlscodes (0x82 für den Puffer 1 bzw. 0x85 für den Puffer 2). Danach sendet er eine 24-Bit-Adresse, die folgendermaßen aufgebaut ist: die Bits 23 bis 21 sind Dummybits, mit den folgenden 12 wird eine Seite adressiert und mit den letzten 9 die Adresse im Puffer, ab welcher die folgenden Datenbytes abgelegt werden sollen. Wenn das Ende des Puffers erreicht wird und weitere Daten übertragen werden, so werden diese ab der Adresse 0 des Puffers gespeichert. Mit dem Schalten des Chip Select Signals auf High beendet der Mikrocontroller die Kommunikation und startet den internen Speicherzugang. Zuerst wird die adressierte Seite gelöscht und anschließend der gesamte Inhalt des ausgewählten Puffers in die Seite gespeichert.

Flash Speichern mit direkten Zugriff über Puffer 1

Mit dem Senden des Befehlscodes 0x02 gefolgt von einer 24-Bit-Adresse wird mit den Bits 20:9 die Seite adressiert, in der die Daten gespeichert werden und mit den Bits 8:0 die Anfangsadresse des Puffers gesetzt (siehe Abb. 7.17). Der Speicherbaustein wartet auf weitere bis zu 512 Bytes, die in den Puffer ab der Anfangsadresse zwischengespeichert werden. Alle Bytes die übertragen wurden, werden nach Kommunikationsende in die vorher gelöschte Seite gespeichert. Wenn weniger als 512 Bytes übertragen wurden, bleiben die übrigen Bytes der Seite unverändert.

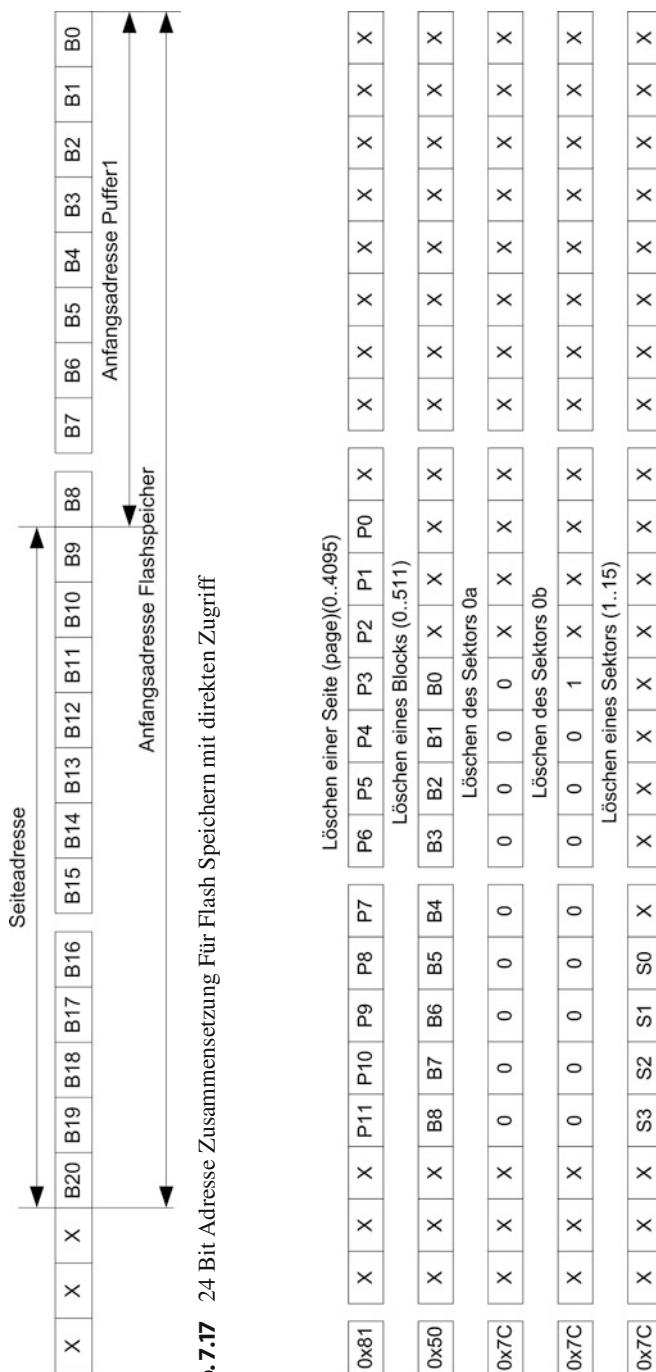
Löschen

Wie bei jedem Flash-Speicher ist das Programmieren nur im gelöschten Zustand möglich (alle Bytes 0xFF). Das Löschen des Bausteins kann nur bereichsweise durchgeführt werden. Um die Arbeit mit dem Baustein flexibel zu gestalten, gibt es die Möglichkeit einzelne Seiten, Blöcke, Sektoren oder den gesamten Chip auf einmal zu löschen. Die Löszeiten sind in der Tab. 7.5 zusammengefasst.

Um einen der ersten drei Speicherbereiche zu löschen, sendet der Master über den SPI-Bus den entsprechenden Befehlscode gefolgt von einer auf 24 Bit kodierten Adresse des Speicherbereichs, so wie in Abb. 7.18 dargestellt. Um den gesamten Speicher zu löschen, wird nur der 4 Byte große Befehlscode übertragen. Am Ende der Übertragung schaltet der Master das Chip Select Signal auf High, was den internen Löschkvorgang startet, währenddessen das Bit RDY/BUSY aus dem STATUS Register auf 0 gesetzt ist. Das bedeutet,

Tab. 7.5 AT45DB161 Löschzeiten

Speicherbereich	Löszeit	Befehlscode
Seite	12...35 ms	0x81
Block	45...100 ms	0x50
Sektor	1,4...2 s	0x7C
Gesamter Speicher	22...40 s	0xC7 94 80 84



dass der Speicher besetzt ist. In dieser Zeit sind das Auslesen der STATUS und Device ID Register und das Schreiben in die Zwischenspeicher erlaubt.

Specherschutz

Für den Schreibschutz unterschiedlicher Speicherbereiche sind mehrere Schreibschutzmechanismen implementiert.

Temporärer Schreibschutz (Sector Protection)

Der Baustein besitzt ein nichtflüchtiges, 16 Byte großes Schreibschutz-Register, in dem man auf kodierte Weise festlegen kann, welche der 17 Sektoren gegen zufälliges Schreiben oder Löschen geschützt werden sollen. Dieses Register kann gelesen oder mit einer Konfiguration der geschützten Sektoren nach vorherigem Löschen neu beschrieben werden. Die Zahl der Schreib-/Löschenzyklen dieses Registers ist auf ca. 10.000 begrenzt. Die Aktivierung/Deaktivierung des Schutzes kann entweder über die Hardware oder die Software realisiert werden. Mit dem Write Protect Pin des Bausteins auf Low geschaltet wird der Schreibschutz der ausgewählten Sektoren aktiviert und das Register selbst wird schreibgeschützt. Mit dem WP-Pin auf High wird mit dem Befehlscode *0x3D 2A 7F A9* der Schreibschutz aktiviert und mit *0x3D 2A 7F 9A* deaktiviert.

Dauerhafter Schreibschutz (Sector Lockdown)

Es besteht die Möglichkeit, einen ganzen Sektor dauerhaft gegen Schreiben/Löschen zu schützen. Dafür muss ein Mikrocontroller als Master über SPI den Befehlscode *0x3D 2A 7F 30* gefolgt von einer 24-Bit-Adresse eines Bytes innerhalb des zu schützenden Sektors zu übertragen. Der Vorgang kann nicht rückgängig gemacht werden. In ein nichtflüchtiges, 16-Byte großes Read Only Register wird die Konfiguration der dauerhaft geschützten Sektoren aktualisiert.

Testbarkeit

Wird der Baustein in aufwändigen Schaltungen eingebaut, sind wegen seiner Komplexität und Speichergröße umfangreiche Testmöglichkeiten gefordert, die Auskunft über den Zustand und Identität des Bausteins geben.

Status Register

Über das 2-Byte große, Read-Only Register kann das Ergebnis einiger Funktionen oder der Stand interner Vorgänge geprüft werden. Das Register ist folgendermaßen strukturiert:

- **Byte 1:**
 - Bit 7 – RDY/BUSY** – ist „0“ solange ein interner Vorgang nicht abgeschlossen ist
 - Bit 6 – COMP** – ist „0“ wenn die Daten beim Vergleich zwischen Puffer und gewählte Seite übereinstimmen
 - Bit 5:2 – DENSITY** – kodiert die Speichergröße des Bausteins (1011 für 16 Mbit)
 - Bit 1 – PROTECT** – bei „1“ zeigt an, dass der Sektorenschreibschutz aktiv ist
 - Bit 0 – PAGE SIZE** – zeigt die Seitengröße an; eine „1“ bedeutet 512 Bytes

- **Byte 2:**

Bit 7 – RDY/BUSY – wie Bit 7 vom Byte 1

Bit 6 – reserviert

Bit 5 – EPE – zeigt bei „0“ an, wenn ein Lösch- oder Schreibvorgang erfolgreich war

Bit 4 – reserviert

Bit 3 – SLE – bei „0“ können weitere Sektoren nicht mehr dauerhaft schreibgeschützt werden

Bit 2 – PS2 – zeigt bei „1“ an, dass ein Schreibvorgang über den Puffer 2 vorübergehend ausgesetzt wurde

Bit 1 – PS1 – zeigt bei „1“ an, dass ein Schreibvorgang über den Puffer 1 vorübergehend ausgesetzt wurde

Bit 0 – ES – zeigt an, dass das Löschen eines Sektors vorübergehend ausgesetzt wurde

Sicherheitsregister (Security Register)

Das Sicherheitsregister ist ein 128-Byte großes Register, dessen zweite Hälfte (Byte 64:127) vom Hersteller mit einer Kennzeichensummer versehen ist, die jeden Baustein eindeutig identifiziert. Dieser Teil kann weder gelöscht noch umprogrammiert werden. Der erste Teil (Byte 0:63) kann vom Anwender einmal programmiert und danach nur noch gelesen werden.

Hersteller und Chip ID-Register

Dieses Read-Only Register speichert Informationen über den Hersteller und den Baustein im JEDEC Standard, die im System zur Identifikation des Speichers dienen können.

7.2.2.3 SST25WF0808 serieller Flashspeicher

Der Baustein SST25WF0808 ist ein serieller Flashspeicher der Familie SST25XXYYYY der Firma Microchip Technology [9]. Die Speicher dieser Familie haben eine Speicherkapazität von 512 kBit (64 kByte) bis 64 Mbit (8 Mbyte) und zeichnen sich durch einen niedrigen Energieverbrauch und eine einfache Ansteuerung aus. Ein Blockschaltbild des 8 Mbit Bausteins SST25WF0808 ist in Abb. 7.19 dargestellt. Jede Speicherzelle ist adressierbar und der Inhalt kann gelesen oder geändert werden. Der gesamte Speicher ist in 256x4 kByte große Speichersektoren, beziehungsweise 16x64 kByte große Speicherblocks unterteilt. Diese Speichereinheiten können einzeln gelöscht werden. Um die unerlaubte Änderung von Speicherbereichen zu vermeiden können diese geschützt werden. Der Baustein wird über eine 4-Leitung SPI-Schnittstelle angesteuert, die mit bis zu 40 MHz getaktet werden kann. Um die Bitrate beim Lesen zu verdoppeln kann eine spezielle, serielle Zweileitungsübertragung verwendet werden.

SPI-Kommunikation

Der Baustein ist als SPI-Slave fest eingestellt. Die Kommunikation kann im Modus 0 oder 3 mit der Übertragung des höherwertigen Bits zuerst stattfinden. Ein steuernder Mikrocontroller wird als Master mit dem Aufruf initialisiert:

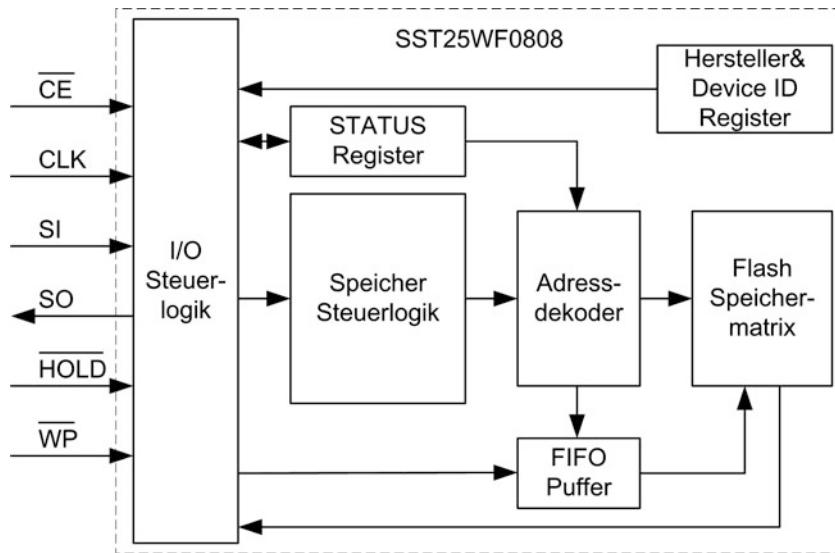


Abb. 7.19 SST25WF0808 Blockschaltbild

```
SPI_Master_Init(SPI_INTERRUPT_DISABLE, SPI_MSB_FIRST, SPI_MODE_0,
                 SPI_FOSC_DIV_16);
```

Die Übertragung eines Bytes beginnt nachdem der Eingang CE (Chip Enable) auf Low geschaltet wird, während der Eingang HOLD auf High ist. Eine Datenübertragung an den Speicher kann der Mikrocontroller durch das Setzen des Eingangs HOLD auf Low unterbrechen und die SPI-Kommunikation mit einem anderen Slave herstellen. Mit dem Setzen des Eingangs HOLD zurück auf High wird die Kommunikation mit dem Speicher dort fortgesetzt, wo sie unterbrochen wurde. Während der Unterbrechung muss der Eingang CE weiterhin auf Low bleiben. Die SPI-Datenstruktur eines Speichers SST25WF0808, dessen Eingänge WP und HOLD inaktiv sind (an der Versorgungsspannung fest angeschlossen) und dessen Eingang CE mit dem Pin PC1 eines Mikrocontrollers ATmega88 verbunden ist (Abschn. 6.1.5), sieht folgendermaßen aus:

```
SST25PFXX_pins SST25PFXX_1 ={{/*CS_DDR*/      &DDRC,
                                /*CS_PORT*/     &PORTC,
                                /*CS_pin*/      PC1,
                                /*CS_state*/    ON},
                                /*WP_DDR*/      OFF,
                                /*WP_PORT*/    OFF,
                                /*WP_pin*/     OFF,
```

```

/*WP_state*/    OFF,
/*HOLD_DDR*/    OFF,
/*HOLD_PORT*/   OFF,
/*HOLD_pin*/    OFF,
/*HOLD_state*/  OFF};

```

Statusregister

Das Statusregister ist ein 8 Bit Register, das den Zustand von intern laufenden Vorgängen liefert und den Schutz gegen die versehentliche Änderung der gespeicherten Informationen steuert.

Bit 7 – BPL – dieses Bit zusammen mit dem Eingang WP steuert den Schreibschutz des Status Registers und des gesamten Speichers (siehe Tab. 7.6);

Bit 6 – reserviert;

Bit 5:2 – diese Bits können über die Software geändert werden und bestimmen die Speicherbereiche, die schreibgeschützt werden sollen (für Details, siehe [9]); wenn Bit 2 = Bit 3 = Bit 4 = 0 ist der gesamte Speicherbereich schreibbar;

Bit 1 – WEL – wird intern gesteuert und zeigt, wenn es „1“ ist, dass Inhalte des Speichers geändert werden können;

Bit 0 – BUSY – das Bit zeigt, wenn es „1“ ist, dass ein interner Schreibvorgang noch nicht beendet ist.

Bei der Wiederholung von Schreib- und/oder Löschoperationen ist auf die Dauer dieser Vorgänge zu achten (Tab. 7.7). Ein Timer kann mit der maximalen Dauer eines Änderungsvorgangs eingestellt werden, um das Ende der Operation über einen Interrupt zu signalisieren. Eine weitere Möglichkeit, das Ende einer Operation festzustellen, besteht darin, in der Endlosschleife der main-Funktion regelmäßig den Wert des Bits BUSY im Status Register zu prüfen, wie im folgenden Codeausschnitt zu sehen ist:

Tab. 7.6 SST25WF0808 Schreibschutz

WP-Pin	BPL-Bit	Status Register	Speicher
Low	1	R	R
Low	0	R/W	Schutzfreie Bereiche sind R/W
High	x	R/W	Schutzfreie Bereiche sind R/W

Tab. 7.7 SST25WF0808 Dauer der internen Schreibzyklen

	Speicherbereich	Operation	Dauer
Status Register	Schreiben	< 10 ms	
Speichersektor	Löschen	40...150 ms	
Speicherblock	Löschen	80...250 ms	
Gesamter Speicher	Löschen	0,5...6 s	
n-Byte	Schreiben	<(0,2 + n · 0,8 / 256) ms	

```
#define BUSY 7
while(1)
{
    if(!(SST25_Read_StatusRegister(SST25PFXX_1) & (1 << BUSY)))
    {
        //Schreibzyklus ist beendet
    }
}
```

Funktionen

Lese-Funktionen

Unabhängig vom Zustand des Pins WP und des Bits BPL können der gesamte Speicher und das Statusregister gelesen werden. Um das Statusregister zu lesen, sendet der Mikrocontroller den Code *0x05*. Die Speicherlogik dekodiert den Befehl und legt den Inhalt des Registers in den Sendepuffer. Mit dem Senden eines weiteren Bytes liest der Mikrocontroller diesen Sendepuffer aus.

```
uint8_t SSTPFXX_Read_StatusRegister(SST25PFXX_pins sdevice_pins)
{
    //sdevice_pins Datenstruktur mit der Definition der
    //ansteuerbaren Pins
    uint8_t ucDataByte;
    SPI_Master_Start(sdevice_pins.SST25PFXXspi); //CS auf Low
    SPI_Master_Write(0x05); //Lesebefehl des Status Registers wird
                           //übertragen
    ucDataByte = SPI_Master_Write(0xFF); /*es wird ein Dummy-Byte
                                       übertragen um den Inhalt des Registers auslesen zu können*/
    SPI_Master_Stop(sdevice_pins.SST25PFXXspi); //CS auf High
    return ucDataByte;
}
```

Folgendes Beispiel stellt eine Funktion für das Lesen eines gespeicherten Bytes dar. Nach dem Senden des Lesecodes *0x03* (Lesen bei maximaler Taktfrequenz von 30 MHz) folgen die 24 Bit Adresse der gewünschten Speicherzelle und ein Dummy-Byte, um deren Inhalt auszulesen. Nach dem Lesen eines Bytes wird der interne Adresszähler inkrementiert. So ist es möglich, einen ganzen Bereich von direkt aneinander liegenden Speicherzellen in einem SPI-Vorgang zu lesen, wenn man die Adresse der ersten Speicherzelle vorgibt. Nach dem Lesen der Speicherzelle mit der höchsten Adresse wird der Adresszähler mit dem Wert *0x000000* geladen.

```

uint8_t SST25_Read_Byte(SST25PFXX_pins sdevice_pins,
                        unsigned long* uladdress)
{
    //sdevice_pins Datenstruktur mit der Definition der
    //ansteuerbaren Pins
    unsigned char ucAddressHigh, ucAddressMiddle,
    ucAddressLow ucDataByte;
    SPI_Master_Start(sdevice_pins.SST25PFXXspi); //CS auf Low
    SPI_Master_Write(0x03); //Operation Code wird übertragen
    ucAddressLow = uladdress;
    ucAddressMiddle = uladdress >> 8;
    ucAddressHigh = uladdress >> 16;
    SPI_Master_Write(ucAddressHigh); //High-Byte der Adresse wird
                                    //übertragen
    SPI_Master_Write(ucAddressMiddle); //Middle-Byte der Adresse
                                    //wird übertragen
    SPI_Master_Write(ucAddressLow); //Low-Byte der Adresse
                                    //wird übertragen
    ucDataByte = SPI_Master_Write(0xFF); /*es wird ein dummy-Byte
    übertragen um ein Byte auslesen zu können*/
    SPI_Master_Stop(sdevice_pins.SST25PFXXspi); //CS auf High
    return ucDataByte;
}

```

Lösch-Funktionen

Nach dem Löschen werden alle Bits des gewählten Speicherbereichs auf „1“ gesetzt. Ein Speicherbereich kann nur gelöscht werden, wenn er nicht schreibgeschützt ist. Vor dem Löschen muss das Schreiben freigegeben werden. Das geschieht mit dem Übertragen des Codes *0x06*, der das Bit WEL im Statusregister auf „1“ setzt. Für das Löschen eines Speicherblocks muss der Befehlscode *0xD8* übertragen werden und danach eine beliebige Adresse aus dem Zielblock. Nach dem Setzen der Leitung CE auf High wird die SPI-Übertragung beendet und es beginnt der intern gesteuerte Löschkvorgang. Wenn der Vorgang beendet ist, werden die Bits BUSY und WEL auf „0“ gesetzt. Mit dem folgenden Codebeispiel kann der gesamte Inhalt des Speichers gelöscht werden.

```

void SST25_ChipErase(SST25PFXX_pins sdevice_pins)
{
    //sdevice_pins Datenstruktur mit der Definition der
    //ansteuerbaren Schreibfreigabe
    Pins SST25_WriteEnable(sdevice_pins);
    SPI_Master_Start(sdevice_pins.SST25PFXXspi); //CS auf Low
    SPI_Master_Write(0x60); //der gesamte Inhalt des Speichers
                            //wird gelöscht
    SPI_Master_Stop(sdevice_pins.SST25PFXXspi); //CS auf High
}

```

Schreib-Funktionen

Der Programmervorgang ermöglicht nur das Schalten der Bits von „1“ auf „0“, deshalb muss der Speicherbereich vorher gelöscht werden. Wie beim Löschen darf der Speicher nicht geschützt werden und das Schreiben muss freigegeben werden. Der Master sendet den Befehlscode *0x02* gefolgt von der Adresse des ersten Bytes, das geändert werden soll. Weitere n-Datenbyte werden übertragen und in einem 256 Byte großen FIFO-Puffer auf dem Speicherbaustein zwischengespeichert. Von den n-Datenbytes werden nur die letzten 256 programmiert. Mit dem Schalten des Anschlusses CE auf High wird die SPI-Übertragung beendet und es beginnt die Programmierung. Am Ende der Programmierung werden die Bits BUSY und WEL auf „0“ gesetzt. Der Schreibvorgang ist für 256 Byte große Seiten optimiert. Die Programmierung eines einzelnen Bytes kann 203,125 µs dauern (siehe Tab. 7.7), während die Programmierung einer ganzen Seite ca. 1 ms dauert, was einer Dauer von knapp 3,9 µs/Byte entspricht.

Die Anfangsadresse der Speicherseiten ist ein Vielfaches von 256. Nach dem Speichern eines Bytes wird der interne Adresszähler innerhalb der gewählten Speicherseite inkrementiert. Wenn der Adresszähler die letzte Adresse der Seite erreicht, springt er auf die erste Adresse dieser Seite, was beim Speichern von Datensätzen größer als 1 Byte berücksichtigt werden muss. Ein Beispiel einer Funktion, die 1 Byte speichern soll, ist im folgenden Programmcode dargestellt.

```
void SST25_Write_Byte(SST25PFXX_pins sdevice_pins,
                      unsigned long* uladresse,
                      uint8_t ucbytetowrite)
{
    unsigned char ucAddressHigh, ucAddressMiddle,
                ucAddressLow ucDataByte;
    SST25_WriteEnable(sdevice_pins); //Schreibfreigabe
    SPI_Master_Start(sdevice_pins.SST25PFXXspi); //CS auf Low
    SPI_Master_Write(0x02); //Operation Code wird übertragen
    ucAddressLow = uladdress;
    ucAddressMiddle = uladdress >> 8;
    ucAddressLow = uladdress >> 16;
    SPI_Master_Write(ucAddress_High); //High-Byte der Adresse
                                      //wird übertragen
    SPI_Master_Write(ucAddress_Middle); //Middle-Byte der Adresse
                                      //wird übertragen
    SPI_Master_Write(ucAddress_Low); //Low-Byte der Adresse wird
                                      //übertragen
    SPI_Master_Write(ucbytetowrite); //das zu schreibende Byte wird
                                      //übertragen
    SPI_Master_Stop(sdevice_pins.SST25PFXXspi); //CS auf High
}
```

2-Leitung serielle Schnittstelle

Diese Schnittstelle benutzt die gleichen Anschlüsse wie die SPI Schnittstelle, ermöglicht aber die parallele Übertragung von 2 Datenbit und eine Bitrate von 80 Mbit/s bei einer Taktfrequenz von 40 MHz. Der Master setzt die Leitung CE auf Low und überträgt im SPI-Modus für das Lesen den Code `0x3B` gefolgt von der Adresse. Während der Übertragung eines weiteren Dummy-Bytes wird der Befehl dekodiert und der Adresszähler geladen. Die Speicherlogik schaltet nun auch den Anschluss DI auf Ausgang und schiebt über die Anschlüsse DO und DI bei jedem Takt jeweils ein Bit heraus. Der Master muss alle 4 Takte aus den empfangenen Bits ein Byte zusammenstellen. Wegen des hohen Rechenaufwands kann die Übertragung in diesem Modus bei höheren Bitraten mit Mikrocontrollern der Familie ATmega nicht realisiert werden. Die 2-Leitung Schnittstelle kann dagegen sehr gut in programmierbaren Logikbausteinen (CPLD und FPGA) implementiert werden.

7.3 Digital-Analog-Wandler

7.3.1 MCP48XX SPI-angesteuerte Digital-Analog-Wandler

Die Bausteine der Reihe MCP48XX [10, 11] sind 1- oder 2-Kanal, serielle, unipolare D/A-Wandler mit einer Bitauflösung von 8, 10 oder 12 Bits (siehe Tab. 7.8). Sie zeichnen sich durch eine einfache Außenbeschaltung aus, dank einer einzigen Versorgungsspannung V_{DD} zwischen 2,7 und 5,5 V und der internen, präzisen Referenzspannung von 2,048 V. Die D/A-Wandler werden über eine serielle, unidirektionale (SDO-Leitung fehlt) SPI-Schnittstelle angesteuert, die mit einer Frequenz von bis zu 20 MHz getaktet werden kann und haben eine Einschwingzeit von 4,5 μ s.

Die Bausteine beinhalten wie in Abb. 7.20 dargestellt ein doppeltes Datenregister, ein D/A-Wandler und einen analogen Verstärker.

Tab. 7.8 MCP48xx – SPI angesteuerte D/A-Wandler

Baustein MCP	Bitauflösung	Kanäle	Verstärkungsfaktor	
			1	2
			Schrittweite [mV] / U_{out} [mV]	Schrittweite [mV] / U_{out} [mV]
4801	8	1	8/0..2040	16/0..4080
4802	8	2	8/0..2040	16/0..4080
4811	10	1	2/0..2046	4/0.. 4092
4812	10	2	2/0..2046	4/0.. 4092
4821	12	1	0,5/0..2047,5	1/0..4095
4822	12	2	0,5/0..2047,5	1/0..4095

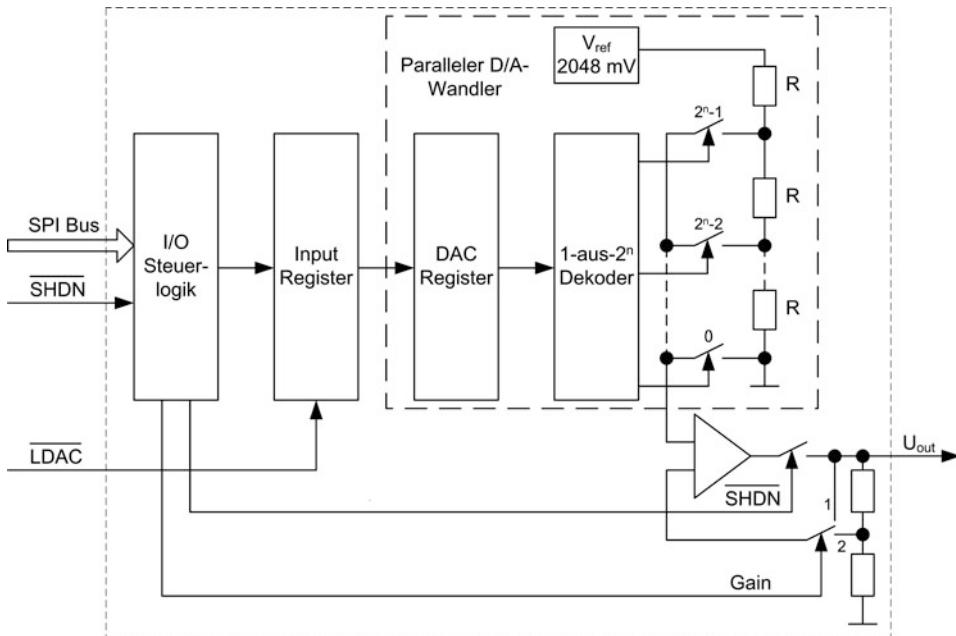


Abb. 7.20 MCP48x1 – Blockschaltbild

7.3.1.1 Die SPI-Schnittstelle

In der I/O Steuerlogik ist ein SPI-konformes Protokoll implementiert, das für die Kommunikation mit einem Mikrocontroller sorgt. Der Baustein ist als Slave vorkonfiguriert und ein Master muss die Bytes im SPI-Modus 0 oder 3 senden, mit dem höchstwertigen Bit zuerst. Es handelt sich um eine unidirektionale Übertragung, weil der Baustein kein Sendeanschluss besitzt (es gibt kein SDO-Pin). Die Datenübertragung beginnt mit der Aktivierung der Chip Select (oder Slave Select) Leitung, gefolgt von der Übertragung von 2 Bytes, die benötigt werden um einen analogen Wert zu erzeugen. Das zuerst übertragene Byte wird die Bits 15:8 und das zweite Byte die Bits 7:0 vom Eingangsregister belegen. Auf der steigenden Flanke vom Chip Select werden die zwei Bytes ins Eingangsregister gespeichert.

Damit man ein Softwaremodul für den MCP48XX allgemein gestalten kann, wird in der Hauptdatei eine Datenstruktur definiert und für jeden am SPI-Bus angeschlossenen Baustein, abhängig von der konkreten Beschaltung, initialisiert. Für jeden ansteuerbaren Anschluss (in diesem Fall Slave Select, Latch DAC und Shutdown) müssen die entsprechenden DDR- und PORT-Register, sowie der Anschluss Pin angegeben werden und vermerken, ob sie in der Schaltung angeschlossen sind. Im Softwaremodul wird eine Datenstruktur folgender Form deklariert (Abschn. 6.1.5):

```

typedef struct
{
    tspiHandle MCP48XXspi;

    volatile uint8_t* LDAC_DDR;
    volatile uint8_t* LDAC_PORT;
    uint8_t LDAC_pin;
    uint8_t LDAC_state;

    volatile uint8_t* SHDN_DDR;
    volatile uint8_t* SHDN_PORT;
    uint8_t SHDN_pin;
    uint8_t SHDN_state;
} MCP48XX_pins;

```

7.3.1.2 Das Eingangsregister

Das 16 Bit große Eingangsregister beinhaltet neben den Datenbits auch Steuerbits und ist folgendermaßen konfiguriert:

- Bit 15** – ist bei MCP48x1 immer „0“, während beim MCP48x2 „0“ für Kanal A und „1“ für Kanal B steht;
- Bit 14** – reserviert;
- Bit 13** – Gain – wenn dieses Bit den Wert „0“ hat, wird das analoge Signal um den Faktor 2 verstärkt;
- Bit 12** – Shutdown – bei „1“ wird die analoge Spannung auf den Ausgang zugeschaltet; die 1-Kanal Wandler besitzen zusätzlich einen Shutdown-Eingang;
- Bit 11:0** – Datenbits für MCP482x (Bit 11 = MSB, Bit 0 = LSB);
- Bit 11:2** – Datenbits für MCP481x (Bit 11 = MSB, Bit 2 = LSB, Bit 1:0 ohne Bedeutung);
- Bit 11:4** – Datenbits für MCP480x (Bit 11 = MSB, Bit 4 = LSB, Bit 3:0 ohne Bedeutung);

7.3.1.3 Der D/A-Wandler

Der D/A-Wandler besteht hauptsächlich aus einem Spannungsteiler mit 2^n gleich großen Widerständen, wobei n die Bitauflösung ist. Der Spannungsteiler ist an der internen Referenzspannung und an der Masse des Bausteins angeschlossen. An jeder gemeinsamen Verbindung zwischen zwei Widerständen und an der Masse ist jeweils ein analoger, elektronischer Schalter angeschlossen. Durch die Verbindung des zweiten Anschlusses aller 2^n Schalter wird der Ausgang des D/A-Wandlers gebildet. Die im DAC-Register gespeicherte Zahl steuert über den 1-aus- 2^n -Decoder einen einzigen Schalter an. Die Eingangsspannung dieses Schalters wird auf den gemeinsamen Ausgang durchgeschaltet. Wenn der Steuereingang LDAC auf Low geschaltet wird, dann werden die Datenbits aus dem Eingangsregister in das DAC-Register gespeichert und somit die D/A-Wandlung gestartet, die nach der Einschwingzeit beendet ist.

Wenn U_{REF} die Referenzspannung des Wandlers ist, dann definiert man die Schrittweite U_{LSB} als die analoge Spannung, die einem LSB entspricht:

$$U_{LSB} = \frac{U_{REF}}{2^n} \quad (7.1)$$

Mit der Gl. 7.1 lässt sich die Spannung vor dem i -ten Widerstand berechnen:

$$U_i = (2^i - 1) \cdot U_{LSB} \quad (7.2)$$

Der interne D/A-Wandler wandelt positive Binärzahlen im Bereich $0 \dots 2^n - 1$ in eine positive Spannung, theoretisch zwischen $0 \text{ V} \dots (U_{REF} - U_{LSB}) \text{ V}$ um. Der relative Unterschied zwischen den Widerstandswerten ist gering und somit wird eine gute Linearität der Ausgangsspannung über den gesamten Temperaturbereich gewährleistet.

7.3.1.4 Der analoge Ausgangsverstärker

Damit der Energieverbrauch des Bausteins klein gehalten wird, muss der Widerstand des Spannungsteilers groß sein, was aber bedeutet, dass bei unterschiedlichen Belastungen die reale Ausgangsspannung von der berechneten abweicht. Die Ausgangsspannung wird unempfindlicher gegenüber der Außenbelastung, wenn ein analoger Verstärker mit niedrigem Ausgangswiderstand verwendet wird. Mit einem Verstärkungsfaktor von 2, der über das Bit Gain des Eingangsregisters einstellbar ist, lassen sich Ausgangsspannungen zwischen $0 \text{ V} \dots 2 \cdot (U_{REF} - U_{LSB}) \text{ V}$ realisieren, jedoch nicht größer als V_{DD} . Dadurch verdoppelt sich auch die Schrittweite. Dank der Rail-to-Rail Technologie kann die analoge Ausgangsspannung praktisch Werte im Bereich $10 \text{ mV} \dots (V_{DD} - 40 \text{ mV})$ annehmen.

Über das Bit 12 (Shutdown) des Eingangsregisters, beziehungsweise den Shutdown-Pin bei den 1-Kanal D/A-Wandlern, kann der Ausgangspin von dem Verstärkerausgang abgekoppelt und auf hohe Impedanz geschaltet werden.

In Anwendungen, in denen die Ausgabe der analogen Spannung nicht zeitkritisch ist (zum Beispiel die Erzeugung einer variablen Referenzspannung oder Offsetspannung oder die Kalibrierung eines Sensors), kann die LDAC-Leitung die ganze Zeit auf Low geschaltet bleiben. Auf der steigenden Flanke des Chip Select Signals werden die Datenbytes in das Eingangsregister und gleichzeitig die Datenbits in das DAC-Register gespeichert und die Umwandlung gestartet.

7.3.1.5 Synchrone Ansteuerung zweier D/A-Wandler

Wenn man ein PAM-Signal² mit fester Abtastrate oder zwei analoge Signale mit definierter Phasenverschiebung erzeugen will, müssen die analogen Spannungen zeitgesteuert ausgegeben werden. In diesem Fall ist der Zeitpunkt des Umwandlungsstarts über das LDAC-Signal zu bestimmen, indem man am Ende der SPI-Übertragung das Steuersignal von High auf Low schaltet. Die Zeitverläufe der SPI-Übertragung für die Erzeugung von synchronen Spannungen mit der Schaltung in Abb. 7.21b ist in Abb. 7.22 dargestellt. Die

² PAM-Signal – Puls Amplitude Moduliertes Signal.

Reihenfolge, in der man die D/A-Wandler anspricht, ist ohne Bedeutung. Nachdem die Datenbytes an die zwei D/A-Wandler gesendet wurden, startet der Master synchron die Umwandlungen durch das Umschalten des LDAC-Signals.

Ein Beispielcode für die in Abb. 7.22 dargestellte Ansteuerung für die Beschaltung aus Abb. 7.21 ist im Folgenden angegeben. Mit:

```
#define ON      1
#define OFF     0
```

werden die SPI-Datenstrukturen der zwei Bausteine in der Hauptdatei initialisiert:

```
//Deklaration für IC1
MCP48XX_pins  MCP48XX_1 = {{ &DDRD,  &PORTD,  PD6,  ON},
                           &DDRB,  &PORTB,  PB2,  ON,
                           OFF,  OFF,  OFF,  OFF};

//Deklaration für IC2
MCP48XX_pins  MCP48XX_2 = {{ &DDRC,  &PORTC,  PC4,  ON},
                           &DDRB,  &PORTB,  PB2,  ON,
                           OFF,  OFF,  OFF,  OFF};
```

Im ersten Schritt werden die zuvor berechneten Bytes an den ersten D/A-Wandler übertragen,

```
//Ausschnitt aus der main-Datei
//die Slave Select-Leitung von IC1 wird auf Low gesetzt
SPI_Master_Start(MCP48XX_1.MCP48XXspi);
//das höchstwertige Byte für IC1 wird übertragen
SPI_Master_Write(ucMSByte_IC1);
//das niederwertige Byte für IC1 wird übertragen
SPI_Master_Write(ucLSByte_IC1);
//die Slave Select-Leitung wird auf High gesetzt und somit wird
//die SPI-Übertragung beendet
SPI_Master_Stop(MCP48XX_1.MCP48XXspi);
```

und dann an den zweiten:

```
//die Slave Select-Leitung von IC2 wird auf Low gesetzt
SPI_Master_Start(MCP48XX_2.MCP48XXspi);
//das höchstwertige Byte für IC2 wird übertragen
SPI_Master_Write(ucMSByte_IC2);
//das niederwertige Byte für IC2 wird übertragen
SPI_Master_Write(ucLSByte_IC2);
//die Slave Select-Leitung wird auf High gesetzt und somit wird die
//SPI-Übertragung beendet
SPI_Master_Stop(MCP48XX_2.MCP48XXspi);
```

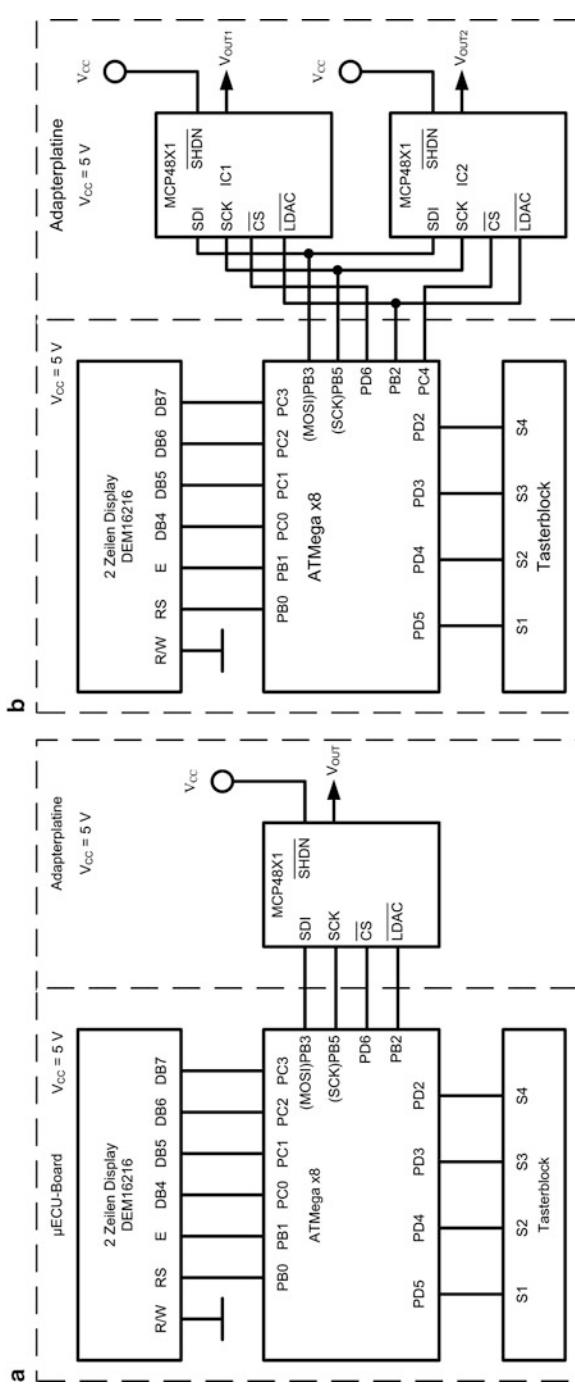


Abb. 7.21 Anschluss eines SPI-angesteuerten D/A-Wandlers aus der Reihe MCP48x1 an einem Mikrocontroller (a) und Vernetzung zweier D/A-Wandler (b)

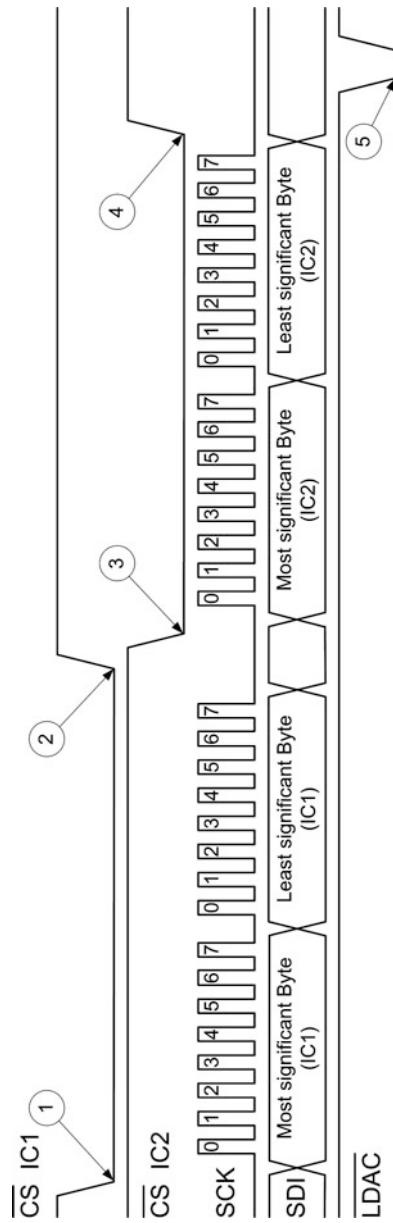


Abb. 7.22 Zeitbereichsdarstellung der SPI-Übertragung für die Erzeugung eines analogen Wertepaars mit zwei A/D-Wandler der Reihe MCP48x1.

1 Beginn der SPI-Kommunikation mit IC1. 2 Ende der SPI-Kommunikation mit IC1. Die 2 Bytes werden in das Eingangsregister von IC1 gespeichert.

3 Beginn der SPI Kommunikation mit IC2. 4 Ende der SPI-Kommunikation mit IC2. Die 2 Bytes werden in das Eingangsregister von IC2 gespeichert.

5 Die Datenbits der zwei Schnittstellen werden gleichzeitig in das entsprechende Dac-Register transferiert. Beginn der D/A-Wandlung, die nach der Einschwingzeit beendet ist.

und schließlich wird die Umwandlung durch das Umschalten des LDAC-Signals gestartet:

```
/*die gemeinsame LDAC-Leitung der D/A-Wandler wird auf Low und dann auf
High gesetzt (die umgewandelten Werte werden gleichzeitig ausgegeben) */
MCP48XX_Set_LDACLow(MCP48XX_1);
MCP48XX_Set_LDACHigh(MCP48XX_1);
```

7.3.1.6 Softwarebeispiel

Bei der Wahl eines bestimmten D/A-Wandlers spielen die erreichbare Schrittweite und die Aussteuergrenze eine entscheidende Rolle. Die vorgestellte MCP48XX-Reihe bietet eine bessere Schrittweite (Gl. 7.1) bei Ausgangsspannungen kleiner als 2,048 V. Mit Hilfe des Ausgangsverstärkers kann die Aussteuergrenze erweitert werden, dadurch verschlechtert sich aber die Schrittweite. Der analoge Zielwert wird oft mit Hilfe einer Umrechnungsfunktion berechnet. Für die Gestaltung einer allgemeinen Ansteuerfunktion für alle MCP48XX-D/A-Wandler muss mit einem Gleitkomma-Ergebnis gerechnet werden, wenn die beste Schrittweite erzielt werden soll. (siehe Tab. 7.8). Ausgehend von dem Zielwert und unter Berücksichtigung der Bitauflösung, müssen der binäre Wert DAC_n , der in das DAC-Register gespeichert wird und die 16-Bit-Zahl, die in das Eingangsregister übertragen wird, berechnet werden. Die analoge Ausgangsspannung U_{OUT} eines D/A-Wandlers dieser Familie ist durch Gl. 7.3 bestimmt:

$$U_{OUT} = DAC_n \cdot U_{LSB} \cdot Gain \quad (7.3)$$

wobei U_{LSB} die Schrittweite angibt und Gain der Verstärkungsfaktor ist. Löst man Gl. 7.3 nach DAC_n auf, so erhält man:

$$DAC_n = \frac{V_{out}}{U_{LSB} \cdot Gain} \quad (7.4)$$

Mit der Wahl des Verstärkungsfaktors $Gain = 1$ für Ausgangsspannungen kleiner als die Referenzspannung kann die beste Schrittweite erreicht werden.

Im Folgenden wird eine Funktion vorgestellt, die für die Ansteuerung eines beliebigen D/A-Wandlers aus der Reihe benutzt werden kann. Die Funktion berechnet auf Grund der gewünschten Ausgangsspannung, `fvalue` (in mV mit einer Auflösung von 0,5 mV), und des benutzten Bausteins, `ucdevice` (0 für MCP480X, 1 für MCP481X und 2 für MCP482X) den binären Wert DAC_n und den Verstärkungsfaktor `ucGain`. Dieser binäre Wert zusammen mit dem gewählten Kanal `ucchannel` und der Freischaltung der Ausgangsspannung `ucout` bilden die Variable `uiInputRegister`, die in das Eingangsregister des D/A-Wandlers übertragen wird. Mit einer logischen „0“ wird der erste Kanal und mit einer logischen „1“ der zweite Kanal selektiert. Die Ausgangsspannung wird mit einer logischen „1“ freigeschaltet. Die konkrete Beschaltung des Bausteins ist im Definition-Array berücksichtigt. Die Funktion überprüft die als Parameter eingegebene analoge Ausgangsspannung und begrenzt sie im Fall einer Bereichsüberschreitung. Der berechnete

Wert für das Eingangsregister wird über SPI übertragen während das LDAC-Signal auf Low bleibt. Mit der steigenden Flanke des Chip Select Signals wird die Umwandlung gestartet.

```
void MCP48XX_Set_OutputVoltage(MCP48XX_pins sdevice_pins,
                                uint8_t ucdevice,
                                uint8_t ucchannel,
                                uint8_t ucout,
                                float fvalue)
{
    uint8_t ucStepSize, ucLSByte, ucMSByte, ucGain = 2;
    uint16_t uiVoltageOut, uiInputRegister = 0x0000,
    uiCommandBits = 0x0000;
    //Spannungsbegrenzung auf den maximal erreichbaren Wert
    if(fvalue >= (VOLTAGE_REFERENCE * 2)) fvalue
        = (VOLTAGE_REFERENCE * 2) - 1;
    //wenn Kanal 1 gewählt wurde, dann wird Bit 15 gesetzt
    if(ucchannel) uiCommandBits = 0x8000;
    /*wenn die Ausgangsspannung kleiner als die Referenzspannung ist,
    dann wird Bit 13 gesetzt*/
    if(fvalue < VOLTAGE_REFERENCE)
    {
        uiCommandBits |= 0x2000;
        ucGain = 1;
    }
    //wenn Ausgangsfreigabe, dann wird Bit 12 gesetzt
    if(ucout) uiCommandBits |= 0x1000;
    switch(ucdevice)
    {
        case MCP480X:
            uiVoltageOut = fvalue;    //Ausgangsspannung als Ganzzahl
            ucStepSize = (VOLTAGE_REFERENCE * ucGain) / 256; //Berechnung
                                                //der Schrittweite
            uiInputRegister = (uiVoltageOut / ucStepSize) << 4; //Binärwert
                                                //wird ermittelt
            break;
        case MCP481X:
            uiVoltageOut = fvalue;
            ucStepSize = (VOLTAGE_REFERENCE * ucGain) / 1024;
            uiInputRegister = (uiVoltageOut / ucStepSize) << 2;
            break;
        case MCP482X:
            uiVoltageOut = fvalue * 2;
            ucStepSize = (2 * VOLTAGE_REFERENCE * ucGain) / 4096;
```

```

        uiInputRegister = uiVoltageOut / ucStepSize;
        break;
    }
    uiInputRegister |= uiCommandBits; //Berechnung des Eingangsregisters
    //das niederwertigste Byte des Eingangsregisters wird ermittelt
    ucLSByte = uiInputRegister;
    //das höchstwertige Byte des Eingangsregisters wird ermittelt
    ucMSByte = uiInputRegister >> 8;
    //die Slave Select-Leitung wird auf Low gesetzt
    SPI_Master_Start(sdevice_pins.MCP48XXspi);
    SPI_Master_Write(ucMSByte); //das MSB des Eingangsregisters
                                //wird übertragen
    SPI_Master_Write(ucLSByte); //das LSB des Eingangsregisters
                                //wird übertragen
/*die Slave Select-Leitung wird auf High gesetzt und somit wird die
SPI-Übertragung beendet*/
    SPI_Master_Stop(sdevice_pins.MCP48XXspi);
}

```

7.3.2 PCF8591 I²C-angesteuerter D/A- und A/D-Wandler

Der Baustein PCF8591 integriert auf einem einzigen Siliziumchip sowohl einen D/A- als auch einen A/D-Wandler und wird mit einer einzigen Spannung versorgt, so wie in der Abb. 7.23 dargestellt. Die Umwandlungsrate ist für beide Richtungen gleich und ist von der Busfrequenz bestimmt. Der Baustein hat einen analogen Ausgang und vier analoge Eingänge und wird über einen I²C-Bus angesteuert. Dank der drei Adresseingänge können bis zu acht gleiche Bausteine an einem einzigen Zweidraht-Bus angeschlossen werden. Somit könnte die analoge Peripherie eines Mikrocontrollers mit acht analogen Ausgängen oder mit bis zu 32 analogen unsymmetrischen Eingängen erweitert werden.

7.3.2.1 I²C-Kommunikation

Die Kommunikation mit einem Mikrocontroller, der als Master konfiguriert ist, erfolgt bei einer Übertragungsrate von max. 100 kBit/s. Die Übertragungsrate spielt eine aktive Rolle für die Umwandlungsvorgänge, weil mit der Übertragung neue Umwandlungen gestartet werden. Der Master startet die Kommunikation mit der Übertragung der Device-Adresse des Bausteins. Das R/W-Bit, als niederwertigstes Bit dieser Adresse, steuert den internen Datenfluss des Bausteins. Wenn dieses Bit 0 ist (Schreib-Befehl), dann wird das nächste Byte in das Control-Register gespeichert, um die folgende Umwandlung vorzubereiten. Die nächsten 1 bis n Bytes werden in das DAC-Register gespeichert und am Ende eines Bytes (nach 9 Takten) als analoger Wert ausgegeben. Dieser Burst-Modus ermöglicht die höchste Umwandlungsrate: 100 kBit/s / 9 Bit/Sample = 11,1 kS/s (kS/s – kilo Samples/Second).

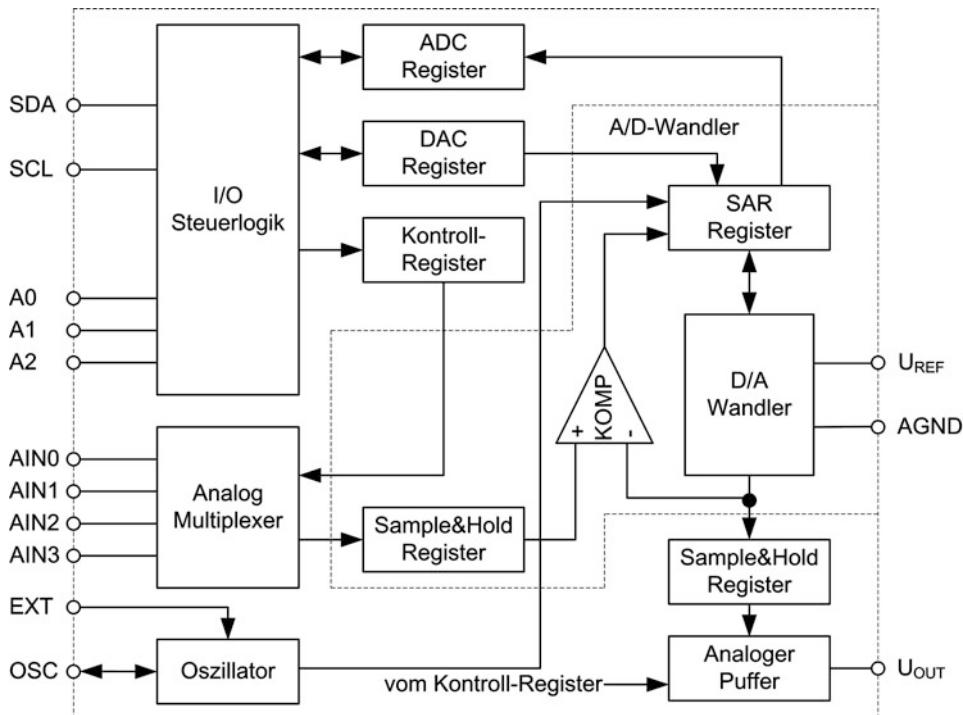


Abb. 7.23 PCF8591 Blockschaltbild

Wenn das R/W-Bit „1“ ist (Lese-Befehl), passiert Folgendes:

- der Slave startet eine neue A/D-Wandlung mit der Bestätigung der eigenen Adresse mit ACK. Die Umwandlung berücksichtigt die vorhandenen Einstellungen aus dem Control-Register (Eingangskonfiguration und Eingangskanal)
- mit den folgenden 8 Takten schiebt der Slave bitweise den Inhalt des ADC-Registers auf die Datenleitung (das Ergebnis der vorigen Umwandlung). Wenn der Master mit ACK antwortet, wird eine neue Umwandlung gestartet und zwar entweder vom gleichen Eingangskanal oder vom nächsten, falls das Autoinkrement-Bit im Control-Register gesetzt ist. Mit einem NACK signalisiert der Master dem Slave das Ende der Kommunikation.

Nach dem Einschalten liest der Mikrocontroller als erstes Byte aus dem ADC-Register den Wert 0x80.

7.3.2.2 Der D/A-Wandler

Der 8-Bit-D/A-Wandler des Bausteins funktioniert nach demselben Parallelverfahren, wie es in Abschn. 7.3.1.3 beschrieben wurde. Der interne Spannungsteiler, der für die

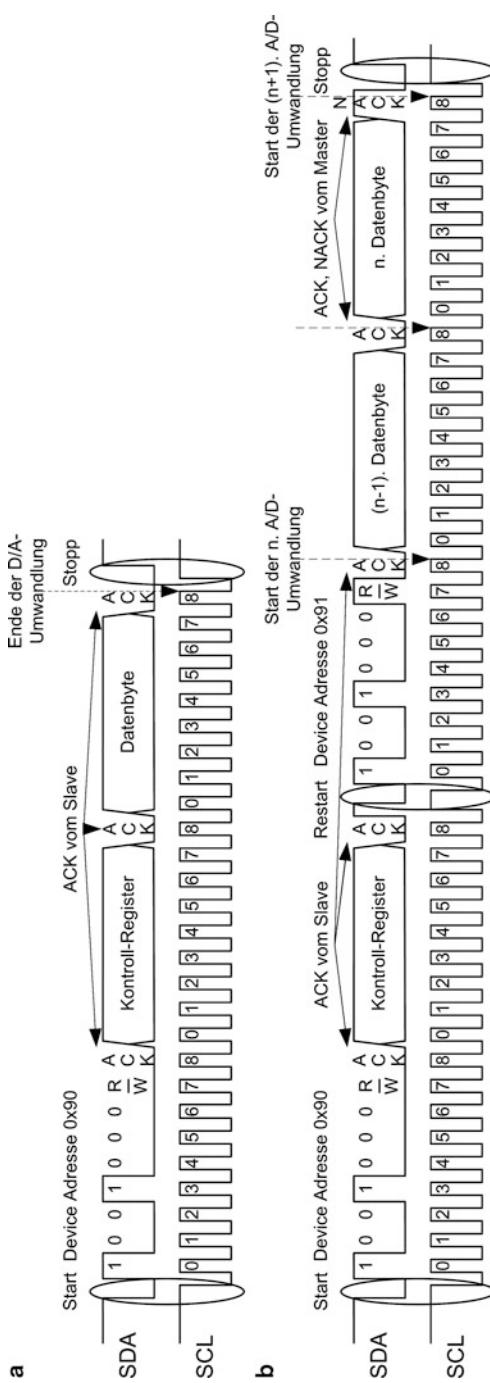


Abb. 7.24 PCF8591 I²C-Kommunikation für die Ansteuerung des D/A- und A/D-Wandlers

D/A-Wandlung sorgt, wird an einer Referenzspannung und an der internen analogen Masse, die von der digitalen galvanisch getrennt ist, angeschlossen. Somit werden der Aussteuerbereich des Wandlers $U_{REF} - U_{AGND}$, die Schrittweite U_{LSB}

$$U_{LSB} = \frac{U_{REF} - U_{AGND}}{256} \quad (7.5)$$

und die Ausgangsspannung

$$U_{OUT} = U_{AGND} + U_{LSB} \cdot \text{OutputByte} \quad (7.6)$$

berechnet. Wenn man die analoge Masse mit der digitalen verbindet, dann wird der Aussteuerbereich gleich U_{REF} , was bedeutet, dass eine analoge, positive Spannung, die kleiner U_{REF} ist, in einen binären Wert kleiner 256 umgewandelt werden kann. Die Einschwingzeit beträgt 90 µs und die Umwandlungsrate kann 11,1 kS/s erreichen. Nach der Umwandlung wird der analoge Wert in einer Sample&Hold-Schaltung gespeichert und über einen analogen, abschaltbaren Puffer am Pin U_{OUT} ausgegeben. Das Control-Register steuert diesen analogen Puffer. Im aktiven Zustand bleibt dessen Wert bis zu einer neuen Umwandlung erhalten. Der analoge Ausgang kann im unbelasteten Zustand 100 %, bei einer Belastung mit 10 kΩ nur noch 90 % der Versorgungsspannung erreichen. Mit folgenden Definitionen:

```
#define PCF8591_DEVICE_TYPE_ADDRESS      0x90
#define REFERENCE_VOLTAGE                5000 //mV
#define ANALOG_GROUND_VOLTAGE           0 //mV
#define OUTPUT_STEP_SIZE                 ((REFERENCE_VOLTAGE - ANALOG_GROUND_VOLTAGE) / 256)
#define D_A_CONVERSION_OPCODE            0x40
```

könnte der Programmausschnitt für die Umwandlung eines, in Millivolt eingegebenen, Spannungswerts $iout_value$ folgendermaßen aussehen:

```
uint8_t ucDeviceAddress, ucOut_Byte;
/*der für die Ansteuerung des D/A-Wandlers benötigte Wert wird unter
Berücksichtigung der Referenzspannung und der Massespannung berechnet*/
ucOut_Byte = (iout_value - ANALOG_GROUND_VOLTAGE) / OUTPUT_STEP_SIZE;
/*Adresse des PCF8591 bilden (es können bis zu 8 Bausteine an einem Bus
angeschlossen sein)*/
ucDeviceAddress = (ucdevice_address << 1) | PCF8591_DEVICE_TYPE_ADDRESS;
ucDeviceAddress |= TWI_WRITE; //Write-Modus
//der Master initiiert die I²C Kommunikation
TWI_Master_Start();
if((TWI_STATUS_REGISTER) != TWI_START) return TWI_ERROR;
//Device Adresse senden
```

```

TWI_Master_Transmit(ucDeviceAddress);
if((TWI_STATUS_REGISTER) != TWI_MT_SLA_ACK) return TWI_ERROR;
TWI_Master_Transmit(D_A_CONVERSION_OPCODE);
if((TWI_STATUS_REGISTER) != TWI_MT_DATA_ACK) return TWI_ERROR;
//das Datenbyte wird gesendet
TWI_Master_Transmit(ucOut_Byte);
if((TWI_STATUS_REGISTER) != TWI_MT_DATA_ACK) return TWI_ERROR;
// der Master beendet die Kommunikation
TWI_Master_Stop();
return TWI_OK;

```

Der zeitliche Verlauf der Kommunikation ist in Abb. 7.24a zu sehen. Wenn man einen analogen Wert erzeugen will und die Referenz- und Offsetspannung der analogen Masse bekannt sind, dann kann Gl. 7.6 nach dem binären Wert aufgelöst werden und man erhält:

$$\text{OutputByte} = \frac{U_{\text{OUT}} - U_{\text{AGND}}}{U_{\text{LSB}}}. \quad (7.7)$$

Um eine einzige D/A-Wandlung zu realisieren startet der Master die Kommunikation und sendet danach die Adresse des Slaves mit dem R/W-Bit auf „0“ (Schreib-Befehl), ein weiteres Byte, das in das Control-Register gespeichert wird und schließlich das umzuwandelnde Byte. Mit einem Stopp-Befehl beendet der Master die Kommunikation. Der Slave muss jedes empfangene Byte mit ACK bestätigen, ansonsten unterbricht der Master die Kommunikation. Sobald der Slave das Datenbyte mit ACK quittiert hat, wird die analoge Spannung in der Sample&Hold-Schaltung gespeichert und falls das Bit 6 vom Control-Register auf „1“ ist, wird der analoge Ausgangspuffer aktiviert.

7.3.2.3 Der A/D-Wandler

Die vier analogen Eingänge können wie in Abb. 7.25 beschaltet werden und die so entstandenen Kanäle über den analogen Multiplexer an den Eingang des A/D-Wandlers geleitet werden. Die Beschaltung der Eingänge und die Kanalauswahl werden vom Control-Register gesteuert. Die analogen Kanäle können entweder unsymmetrisch (die Spannung wird gegen die analoge Masse gemessen) oder symmetrisch (es wird die Differenz der an den symmetrischen Eingängen angelegten Spannungen gemessen) sein. Bei der Umwandlung eines unsymmetrischen Kanals liegt die untere Aussteuergrenze bei U_{AGND} und die obere bei U_{REF} und das Ergebnis wird als positive Ganzzahl in das ADC-Register gespeichert. Bei der Umwandlung eines symmetrischen Kanals liegt die untere Aussteuergrenze bei $-(U_{\text{REF}} - U_{\text{AGND}})/2$ und die obere bei $+(U_{\text{REF}} - U_{\text{AGND}})/2$ und das Ergebnis wird mit Vorzeichen gespeichert.

Der A/D-Wandler funktioniert nach dem Prinzip der sukzessiven Annäherung (successive approximation)³ und besteht aus dem oben beschriebenen D/A-Wandler, der vorüber-

³ Siehe auch Abschn. 3.8.3.

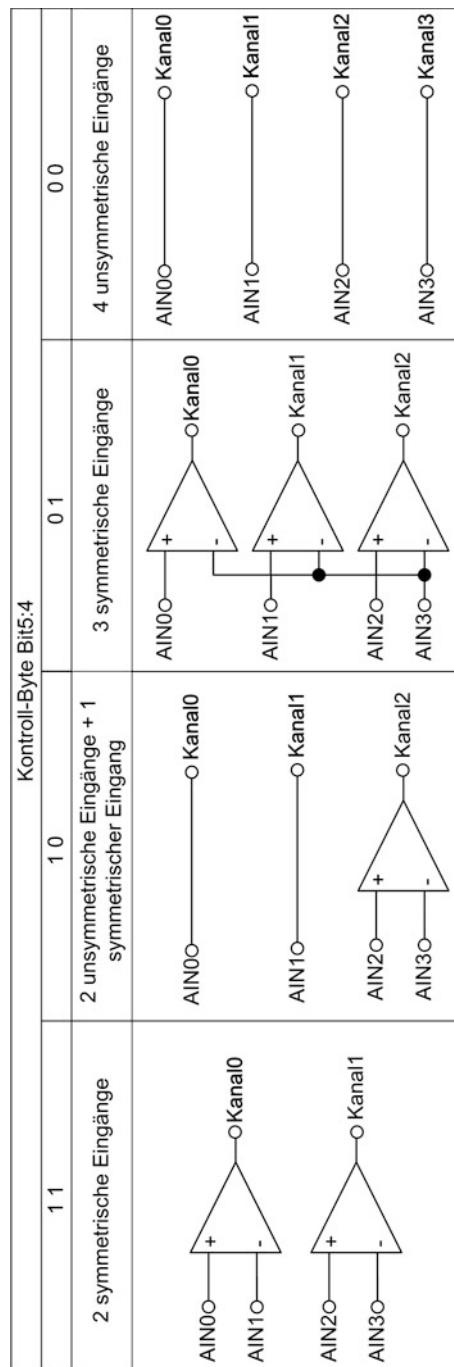


Abb. 7.25 PCF8591 Beschaltung analoger Eingänge

gehend für die A/D-Wandlung eingesetzt wird, einem analogen Komparator und einem SAR⁴-Register. Bei diesem A/D-Wandler ist die Anzahl der Schritte gleich der Bitauflösung. Schrittweise werden die einzelnen Bits beginnend mit Bit 7 getestet, ob sie „1“ oder „0“ sind. Im ersten Schritt wird das Bit 7 (MSB) des SAR-Registers auf „1“ gesetzt, was zu einer analogen Spannung am Ausgang des D/A-Wandlers gleich $V_{REF}/2$ führt. Der Komparator vergleicht die umzuwandelnde Spannung mit der am Ausgang des D/A-Wandlers. Ist sie größer, bleibt das Bit 7 auf „1“, ansonsten wird es auf „0“ gesetzt. Im zweiten Schritt wird das Bit 6 getestet und auf „1“ gesetzt. Die Spannung am Ausgang des D/A-Wandlers beträgt jetzt $Bit\ 7 \cdot U_{REF}/2 + U_{REF}/4$. Nach dem Vergleich mit der Eingangsspannung bleibt Bit 6 gesetzt oder es wird zurückgesetzt. Mit jedem weiteren Schritt nähert man sich der Spannung immer mehr an, nach dem 8. Schritt ist die Annäherung kleiner als U_{LSB} und im SAR-Register steht der binäre Wert der im ADC-Register gespeichert wird.

Im folgenden Codeausschnitt wird ein Beispiel einer Lesefunktion präsentiert, die über eine parametrierte Eingabe der Eingangskonfiguration (`ucinput_mode`), des Eingangskanals (`ucchannel`) und des Zustands des analogen Ausgangspuffers (`ucanalog_out`), den Start einer neuen A/D-Wandlung und das Auslesen des entsprechenden Binärwerts ermöglicht. Zusätzlich wird noch die Device Chip Adresse (`ucdevice_Address`) übergeben, sowie die Adresse der Variable die den umgewandelten Wert (`*ucdigital_out`) speichern soll. Der zeitliche Verlauf beim Aufruf dieser Funktion ist in der Abb. 7.24b dargestellt.

In den Variablen `ucControlByte` und `ucDeviceAddress` werden die Parameter zusammengesetzt, um die neue Konfiguration des Control-Registers und die Gesamtadresse zu gestalten.

```
uint8_t ucDeviceAddress, ucControlByte;
ucControlByte = ucanalog_out | ucchannel | ucinput_mode;
ucDeviceAddress = (ucdevice_address << 1) | PCF8591_DEVICE_TYPE_ADDRESS;
ucDeviceAddress |= TWI_WRITE;
```

Nachdem der Master die Kommunikation initiiert hat, muss er prüfen ob der Bus frei ist, wenn nicht, wird die Funktion mit einem Fehlercode abgebrochen.

```
TWI_Master_Start();
if((TWI_STATUS_REGISTER) != TWI_START) return TWI_ERROR;
```

Wenn der Bus frei ist, sendet der Master die Adresse des Bausteins mit dem R/W-Bit auf „0“ um den Inhalt des Control-Registers ändern zu können. Wenn der Slave den Empfang der Adresse mit ACK bestätigt, sendet der Master auch den neuen Inhalt des Control-Registers. Die Konfiguration der Eingänge und ein neuer Eingangskanal können dadurch mit jedem Aufruf der Funktion geändert werden.

⁴ SAR – Successive Approximation Register.

```

TWI_Master_Transmit(ucDeviceAddress);
if((TWI_STATUS_REGISTER) != TWI_MT_SLA_ACK) return TWI_ERROR;
TWI_Master_Transmit(ucControlByte);
if((TWI_STATUS_REGISTER) != TWI_MT_DATA_ACK) return TWI_ERROR;

```

Nach einem neuen Start der Kommunikation sendet der Master erneut die Adresse des Bausteins, diesmal mit dem R/W-Bit auf „1“ gesetzt, um den Inhalt des ADC-Registers auslesen zu können. Auf der fallenden Flanke des ACK-Bits, mit dem der Slave den Empfang der Adresse bestätigt, wird eine neue A/D-Wandlung gestartet. Während der nächsten acht Clock-Takte findet die Umwandlung statt und das Ergebnis der vorigen Umwandlung, das im ADC-Register gespeichert blieb, wird übertragen. Dieses Byte wird aber von der Funktion verworfen, weil es von einem anderen Eingangskanal oder einer anderen Eingangskonfiguration stammt. Der Master antwortet mit ACK um das gewünschte Ergebnis empfangen zu können und eine neue Umwandlung wird gestartet.

```

TWI_Master_Start();
if((TWI_STATUS_REGISTER) != TWI_RESTART) return TWI_ERROR;
ucDeviceAddress = (ucdevice_address << 1) | PCF8591_DEVICE_TYPE_ADDRESS;
ucDeviceAddress |= TWI_READ; //Read-Modus
TWI_Master_Transmit(ucDeviceAddress);
if((TWI_STATUS_REGISTER) != TWI_MR_SLA_ACK) return TWI_ERROR;
TWI_Master_Read_Ack();
if((TWI_STATUS_REGISTER) != TWI_MR_DATA_ACK) return TWI_ERROR;

```

Das Ergebnis wird an der Adresse der Variable `ucanalog_in` gespeichert und somit steht es in der Hauptdatei zur Verfügung. Der Master wird veranlasst, mit NACK zu antworten und danach die Kommunikation zu beenden. Die Funktion meldet einen fehlerfreien Ausgang.

```

*ucanalog_in = TWI_Master_Read_NAck();
if((TWI_STATUS_REGISTER) != TWI_MR_DATA_NACK) return TWI_ERROR;
TWI_Master_Stop();
return TWI_OK;

```

7.3.2.4 Das Control-Register

Das Control-Register steuert die Beschaltung der analogen Eingänge, den analogen Multiplexer und den analogen Puffer und hat folgende Konfiguration:

- Bit 7** – ist für weitere Entwicklungen reserviert und soll stets auf „0“ gesetzt werden
- Bit 6** – schaltet den analogen Ausgang in den aktiven Zustand mit einer logischen „1“; eine „0“ schaltet diesen Ausgang ab und somit wird Strom gespart
- Bit 5:4** – sind Steuerbits für die Beschaltung der analogen Eingänge (siehe Abb. 7.25)
- Bit 3** – wie Bit 7
- Bit 2** – Autoinkrement Bit; wenn Bit 2 = „1“ ist, werden die analogen Kanäle hintereinander mit der Übertragung jedes Bytes automatisch umgeschaltet. Die zuerst umgewandelte Spannung ist diejenige von Kanal 0. Damit der Vorgang fehlerfrei funktioniert, muss das Bit 6 auf „1“ gesetzt werden
- Bit 1:0** – sind Steuerbits für den analogen Multiplexer: die Bits kodieren den ausgewählten Kanal (beispielsweise bedeutet 00 Kanal 0 usw.)

7.3.2.5 Der Oszillatior

Der Baustein benötigt für die A/D-Umwandlung einen Takt. Wenn Pin EXT mit Masse verbunden ist, wird dieser Takt vom internen Oszillatior generiert und ist am Pin OSC von außen zugänglich. Wenn Pin EXT an V_{CC} angeschlossen ist, dann muss ein externer Oszillatior am Pin OSC den benötigten Takt bereitstellen.

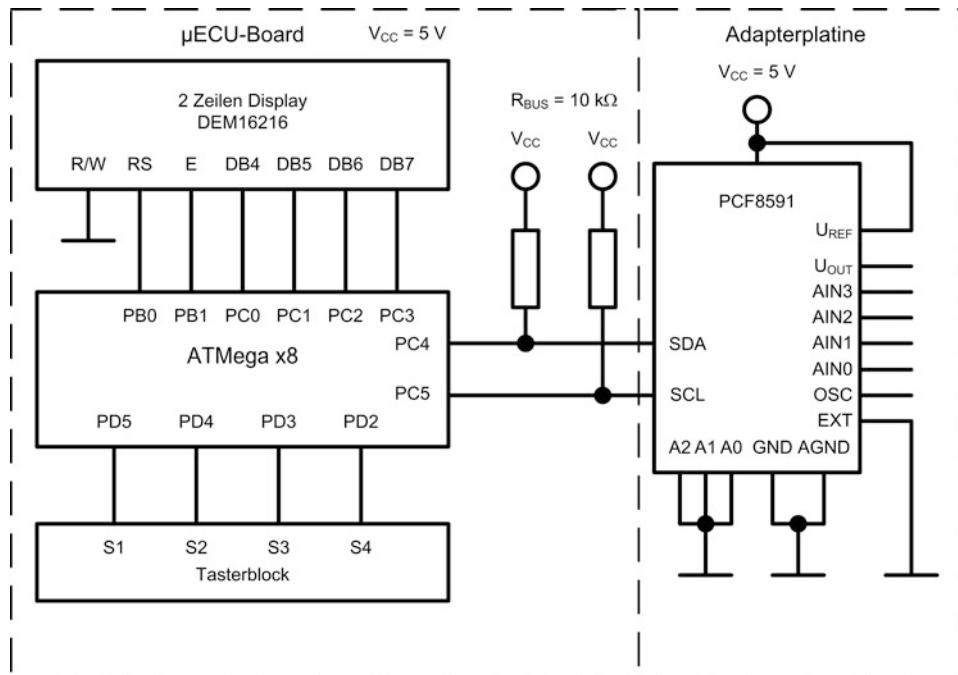


Abb. 7.26 Anschluss eines PCF8591 an einem Mikrocontroller

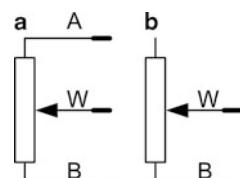
In der Beispielschaltung Abb. 7.26 sind alle drei externe Adresspins mit der Masse verbunden, somit wird die Gesamtadresse des Bausteins 0x90. Der interne Oszillator wird über den EXT-Pin aktiviert und der Aussteuerbereich des D/A-Wandlers ist gleich V_{CC} . Der Pin für die Referenzspannung wird an V_{CC} und der analoge Masseanschluss AGND an der digitalen Masse angeschlossen.

7.4 MCP41X1 digitale Regelwiderstände

Digitale Regelwiderstände sind integrierte Schaltkreise, die einen Spannungsteiler beinhalten, der aus 2^n gleich großen Widerständen besteht, wobei n die Auflösung in Bit ist. Sie werden als Potentiometer oder als Rheostat⁵ hergestellt (siehe Abb. 7.27). Der elektrische Widerstand zwischen dem Ende B und dem Schleifer kann nur stufenweise geändert werden. Die Ansteuerung der Bausteine kann über I²C oder SPI erfolgen. Die Schleiferposition kann entweder in einem RAM oder in einem EEPROM gespeichert werden. Im letzten Fall bleibt die Position auch im stromlosen Zustand erhalten. Namhafte Hersteller wie Microchip Technology Inc., Xicor, Analog Device, Maxim Integrated oder Dallas Semiconductor produzieren digitale Widerstände mit unterschiedlichen Werten und Bitauflösungen. Weiterhin wird beispielhaft die Reihe der digitalen Potentiometer MCP41X1 näher betrachtet.

MCP41X1 ist eine Reihe von einzelnen digitalen Potentiometern, die eine 7/8 Bitauflösung haben und über SPI angesteuert werden. Diese können mit Spannungen zwischen 2,7 und 5,5 V versorgt werden, während die anderen Anschlüsse Spannungsspegel von bis zu 12,5 V vertragen. Die digitalen Potentiometer können als Spannungsteiler mit einstellbarem Teilverhältnis verwendet werden, um analoge Potentiometer in Audioverstärkern zu ersetzen oder für die Einstellung einer Offset-Spannung. Sie können auch in Filter mit einstellbarem Amplitudengang eingesetzt werden. In diesem Fall muss auf die Bandbreite der Bausteine geachtet werden. Sie weisen eine –3 dB Bandbreite von 2 MHz bei den 5 k Ω - und von 100 kHz bei den 100 k Ω -Potentiometern auf. Ein Blockschaltbild eines MCP4151-Bausteins ist in Abb. 7.28 dargestellt.

Abb. 7.27 Regelbare Widerstände: **a** Potentiometer, **b** Rheostat



⁵ Aus dem Griechischen wörtlich übersetzt etwa Flussversteller: Ein einstellbarer Widerstand.

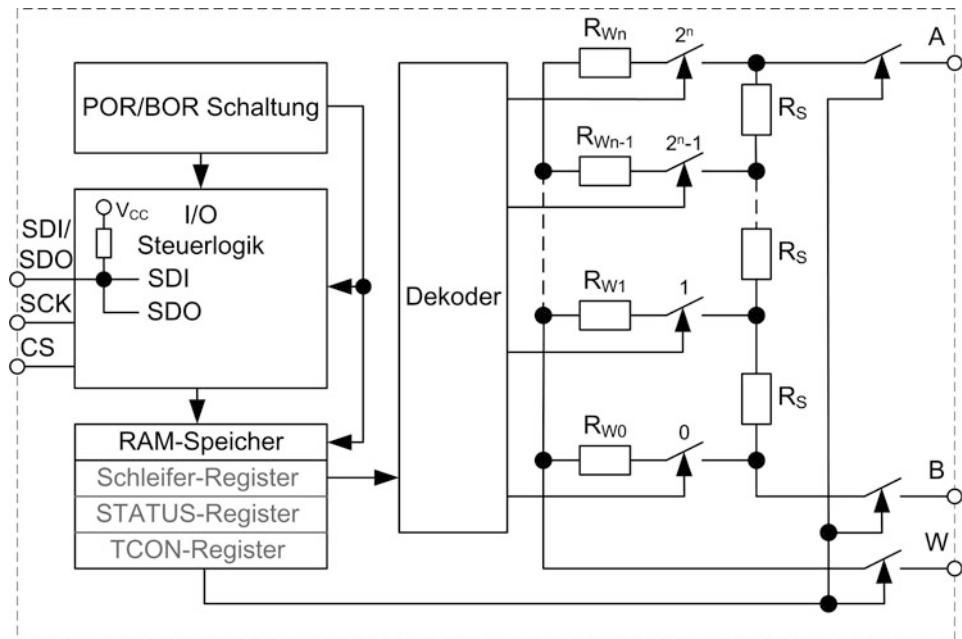


Abb. 7.28 MCP4151 Blockschaltbild

7.4.1 Power-On-/Brown-Out-Reset Schaltung

Die POR/BOR-Schaltung im MCP4151 sorgt dafür, dass der Baustein nach dem Einschalten sowie beim Unterschreiten einer kritischen Schwelle der Versorgungsspannung (Brown Out) einen definierten Zustand einnimmt. Dies geschieht auf folgende Weise:

1. die digitale Kommunikation wird freigegeben und die internen Register werden mit den Standardwerten geladen sobald die Versorgungsspannung einen Pegel V_{BOR} überschreitet;
2. beim Unterschreiten dieses Spannungspegels wird die digitale Kommunikation gesperrt. Der Inhalt der internen Register bleibt auch unterhalb dieses Pegels erhalten bis wann die Versorgungsspannung den Wert V_{RAM} erreicht. Unter dem Pegel V_{RAM} können die Inhalte der einzelnen Register beschädigt werden.

7.4.2 Elektrischer Widerstand

Der elektrische Widerstand des Bausteins MCP4151 besteht aus einer Reihenschaltung von 2^n gleich großen Teilwiderständen R_S , wobei in diesem Fall $n = 8$.

$$R_S = \frac{R_{AB}}{256} \quad (7.8)$$

Die beiden Endanschlüsse des Potentiometers sowie der Schleifer werden an den Pins A, B bzw. W über analoge Schalter ausgeführt. Laut Angaben des Herstellers kann der Stromfluss an den Pins A und W bidirektional stattfinden. An jeder Verbindungsstelle zweier Teilwiderstände ist jeweils ein analoger Schalter angeschlossen. Die Ausgänge aller Schalter sind zusammengeschaltet und bilden den Schleifer des Potentiometers. Außer diesen $2^n - 1$ Stellen soll der Schleifer direkt auch mit den Endanschlüssen A und B verbunden werden können. Dadurch ergeben sich $2^n + 1 = 257$ Schleiferstellungen. Die analogen Schalter sind mit FET⁶-Transistoren realisiert und durch die Reihenschaltung eines elektrischen Kontaktes und eines Widerstandes im Blockschaltbild symbolisiert. Die Teilwiderstände und der Gesamtwiderstand R_{AB} ändern ihren Wert nur geringfügig mit der Temperatur und der Versorgungsspannung. Der Schleifer-Widerstand ist von der Stellung und stark von der Temperatur und Versorgungsspannung abhängig. Er hat einen größeren Einfluss auf die Linearität der Spannung bei kleineren R_{AB} Widerständen.

7.4.3 Potentiometer-Register

Der Baustein besitzt einen RAM-Speicher, in dem die Schleifer Stellungen und die Konfiguration des Bausteins gespeichert sind. Nach dem Einschalten müssen die gewünschten Einstellungen in den Speicher übertragen werden. Er besteht aus 3 Registern, die 9-Bit groß sind um die 257 Schleiferstellungen speichern zu können. Die Adressen der Register und die Standardwerte, die gleich nach dem Einschalten geladen werden, sind Tab. 7.9 zu entnehmen.

Das **Schleifer**-Register speichert und steuert über einen Dekoder die Schleiferstellung. Nach dem Einschalten wird dieses Register mit dem Wert 0x80 (dezimal 128) bei den 8-Bit, bzw. mit 0x40 (64) bei den 7-Bit Potentiometern initialisiert, was der Mitte des Gesamtwiderstandes entspricht. Wenn das Register mit 0x00 geladen wird, so wird der

Tab. 7.9 MCP4151 – Register

Registername	Registeradresse	Standardwert
Schleifer-Register	0x00	0x80
TCON-Register	0x04	0x1FF
Status-Register	0x05	0xE0

⁶ FET – Field Effect Transistor.

Schleifer auf das B-Ende und bei allen Werten größer 0xFF wird der Schleifer auf das A-Ende des Potentiometers geschaltet.

Das **Status**-Register kann nur gelesen werden.

Bit 8:5 – sind reserviert, dauernd auf „1“ geschaltet;

Bit 4:2 – sind reserviert;

Bit 1 – SHDN; dieses Bit gibt an, ob sich der Baustein im Hardware-Shutdown-Modus befindet, wenn es auf „1“ geschaltet ist (nur bei den Bausteinen die einen SHDN-Pin haben). In diesem Modus ist der Pin A des Bausteins vom Endanschluss des Potentiometers getrennt und der Schleifer mit dem Pin B direkt verbunden (Abb. 7.29).

Bit 0 – ist reserviert;

Das **Terminal Control Register (TCON)** steuert die Anschlüsse des digitalen Potentiometers. Nach dem Einschalten werden alle Bits auf 1 gesetzt.

Bit 8 – ist reserviert;

Bit 7:4 – diese Bits sind relevant für die Bausteine mit 2 Potentiometern

Bit 3 – Modus Steuerbit (1 – aktiv-Modus, 0 – Shutdown-Modus)

Bit 2 – Schaltersteuerung des A-Pins (1 – der A-Pin ist mit dem Endanschluss verbunden, 0 – der Pin A ist vom Endanschluss getrennt)

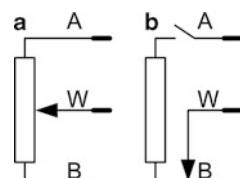
Bit 1 – Schaltersteuerung des W-Pins (1 – der W-Pin ist mit dem Schleifer verbunden, 0 – der Pin W ist vom Schleifer getrennt)

Bit 0 – Schaltersteuerung des B-Pins (1 – der B-Pin ist mit dem Endanschluss verbunden, 0 – der Pin B ist vom Endanschluss getrennt).

Wenn das Bit 3 im Register TCON auf 1 gesetzt wird, wird der Baustein in den Shutdown-Modus versetzt. Das ändert aber weder die Bits dieses Registers noch die der anderen Register. Nach dem Zurücksetzen dieses Bits wird der alte Zustand wiederhergestellt.

Die I/O-Steuerlogik implementiert eine 3-Leitung SPI-konforme Schnittstelle, die den Datenaustausch zwischen einem SPI-Master und dem internen Speicher ermöglicht. Bei dieser Schnittstelle benutzen die internen Signale SDI⁷ und SDO⁸ eine einzige externe

Abb. 7.29 MCP41X1 im aktiven Modus (**a**) und im Shutdown-Modus (**b**)



⁷ SDI – Slave Data In.

⁸ SDO – Slave Data Out.

Leitung, was einem „wired and“ entspricht. Damit diese Beschaltung problemlos funktioniert, muss der SDI/SDO Anschluss mit dem MOS- Anschluss des Mikrocontrollers über einen Widerstand verbunden werden (siehe Abb. 7.30), damit beim Datenauslesen nicht 2 Ausgänge zusammen geschaltet werden. Der interne Pull-up Widerstand wird bei der Datenausgabe vom SDO-Signal aktiviert und ermöglicht die bidirektionale Datenkommunikation, begrenzt aber die Taktfrequenz der Schnittstelle. Diese Taktfrequenz kann durch das Zuschalten eines externen Pull-up Widerstandes erhöht werden.

Dank der Verträglichkeit von Spannungspegeln über der Versorgungsspannung an allen SPI-Pins kann der Baustein ohne zusätzliche Pegelumschalter in Schaltungen funktionieren, in denen der Master und der Slave mit unterschiedlichen Spannungen versorgt werden (Split-Rail-Anwendungen).

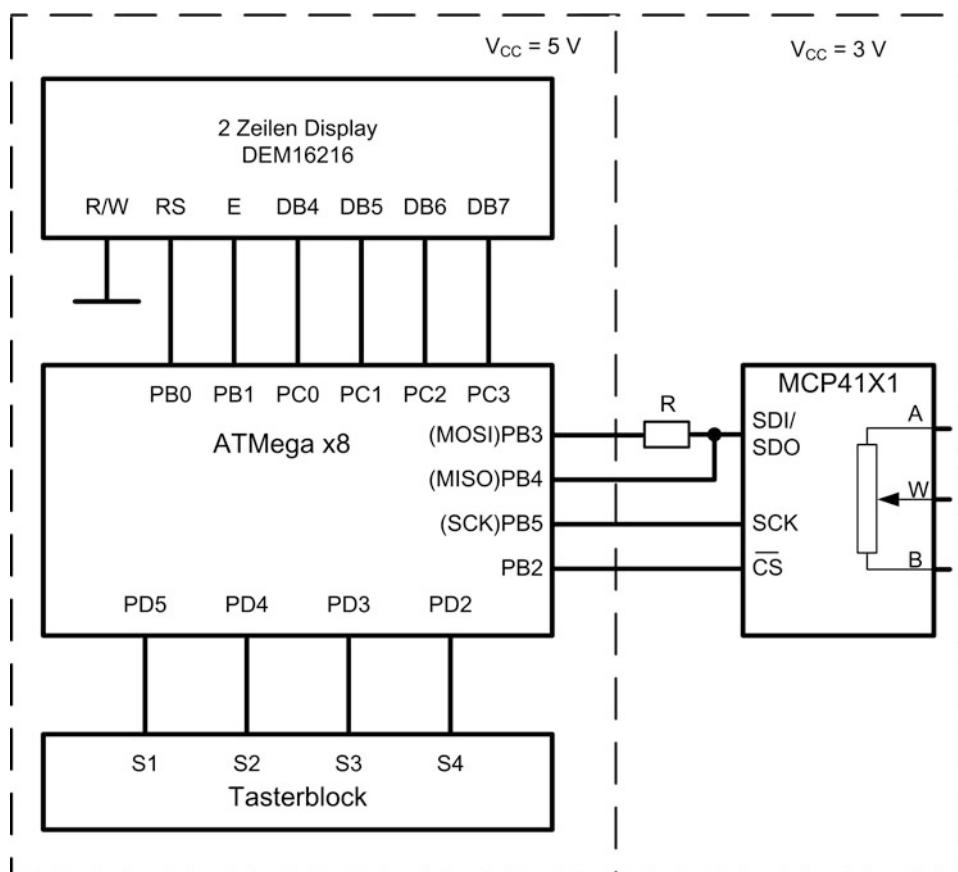


Abb. 7.30 MCP41X1 – Anschluss an einem Mikrocontroller

7.4.4 Ansteuerfunktionen des Bausteins MCP4151

Um den Datenfluss zum und vom internen Speicher zu gewährleisten, stellt der Baustein vier allgemeine Funktionen zur Verfügung, die in Tab. 7.10 zusammengefasst sind.

Das erste Byte wird als Command-Byte bezeichnet, das zweite, wenn es existiert, als Datenbyte. Die Struktur des Command-Bytes sieht folgendermaßen aus:

Bit 7:4 – diese Bits bilden die Adresse des internen Registers (Tab. 7.9);

Bit 3:2 – die Bits codieren den Befehlscode (Tab. 7.10);

Bit 1 – wird als Datenbit 9 bewertet und als ERROR-Bit verwendet;

Bit 0 – wird als Datenbit 8 verwendet um alle Schalter steuern zu können.

Das Datenbyte beinhaltet die Datenbit 7:0. Die erlaubten Kombinationen der ersten 6 Bits des Command-Bytes auf die der Baustein reagiert sind in der Tab. 7.11 aufgelistet.

Tab. 7.10 Befehle der digitalen Potentiometer MCP41X1

Befehl	2-Bit Befehlscode	
Read	11	2-Byte Befehle
Write	00	
Increment	01	1-Byte Befehle; sie beziehen sich auf das Schleifer-Register
Decrement	10	

Tab. 7.11 Erlaubte Kombinationen der ersten 6 Bits des Command-Bytes

Adresse	Befehlscode/ Funktion	Aktion
0000	00 / Write	Das Schleifer-Register wird mit den nachfolgenden Datenbits geladen; ein neues Widerstandsverhältnis an dem Schleifer wird bestimmt
	11 / Read	Der Inhalt des Schleifer-Registers wird ausgelesen
	01 / Inkrement	Wenn der Inhalt des Schleifers-Registers kleiner als 0x100 ist, dann wird er inkrementiert
	10 / Dekrement	Wenn der Inhalt des Schleifer-Registers größer 0 ist, dann wird er dekrementiert
0100	00 / Write	Das flüchtige TCON-Register wird mit den nachfolgenden Datenbits geladen
	11 / Read	Der Inhalt des TCON Registers wird ausgelesen
0101	11 / Read	Der Inhalt des Status-Registers wird ausgelesen

7.4.5 SPI-Kommunikation

Die MCP41X1 Bausteine werden über eine SPI-Schnittstelle im Modus 0 oder 3 mit einer Taktfrequenz von bis zu 10 MHz (bis 250 kHz beim Ausführen eines Read-Befehls) angesteuert. Das höherwertige Bit eines Bytes wird zuerst übertragen. Für die in Abb. 7.30 dargestellte Beschaltung lautet die SPI-Datenstruktur des Bausteins (s. Abschn. 6.1.5):

```
MCP41X1_pins MCP41X1_1 = {{/*CS_DDR*/      &DDRB,
                           /*CS_PORT*/     &PORTB,
                           /*CS_pin*/      PB2,
                           /*CS_state*/    ON}; //ON = 1
```

Um die Kompatibilität der Ansteuerung zu anderen Familien von digitalen Potentiometern zu gewährleisten, kann ein Master die Kommunikation initiieren, und zwar entweder:

- durch das Umschalten der Chip Select Leitung von High (+5 V) auf Low (0 V), oder
- durch das Umschalten der Leitung von High (+5 V) auf einen höheren Spannungspegel V_{IHH} (+8,5 V ... +12 V) (siehe Abb. 7.31).

Am Anfang der Kommunikation ist der Pin SDI/SDO auf Eingang geschaltet und empfängt die ankommenden Bits. Der Zustandsautomat des Bausteins wertet die ersten sechs Bits des Command-Bytes aus und vergleicht sie mit den in Tab. 7.11 aufgelisteten möglichen Kombinationen. Während dieser ersten sechs Bits der Übertragung ist das Signal SDO intern auf „1“ geschaltet. Am Ende der sechsten Taktperiode wird der Pin SDI/SDO auf Ausgang geschaltet und falls keine erlaubte Kombination erkannt wurde, wird das Signal SDO intern auf Low umgeschaltet. Damit wird auch der physikalische Ausgang auf Low geschaltet. Der Ausgang bewahrt diesen Zustand, bis die Kommunikation durch das Umschalten der Chip Select Leitung in den inaktiven Zustand beendet wird (+5 V). Wenn die 6-Bit-Kombination des Command-Byte als korrekt gewertet wird:

- bleibt der SDI/SDO Pin auf Ausgang, falls ein ankommender Read-Befehl erkannt wurde, oder
- der Pin wird auf Eingang geschaltet, um die Datenbits zu empfangen.

Die einzelnen Befehle werden, während die Chip Select Leitung im aktiven Zustand ist, am Ende der achten Taktperiode bei den 1-Byte Befehlen bzw. der 16. Taktperiode bei den 2-Byte Befehlen ausgeführt und nicht – wie man glauben könnte – nach dem Umschalten der Chip Select Leitung vom aktiven in den inaktiven Zustand. Somit besteht die Möglichkeit, die Funktionen bei Bedarf zu verketteten (Abb. 7.31). In diesem Beispiel werden die Funktion *Increment* zweimal hintereinander und die Funktion *Read* einmal ausgeführt.

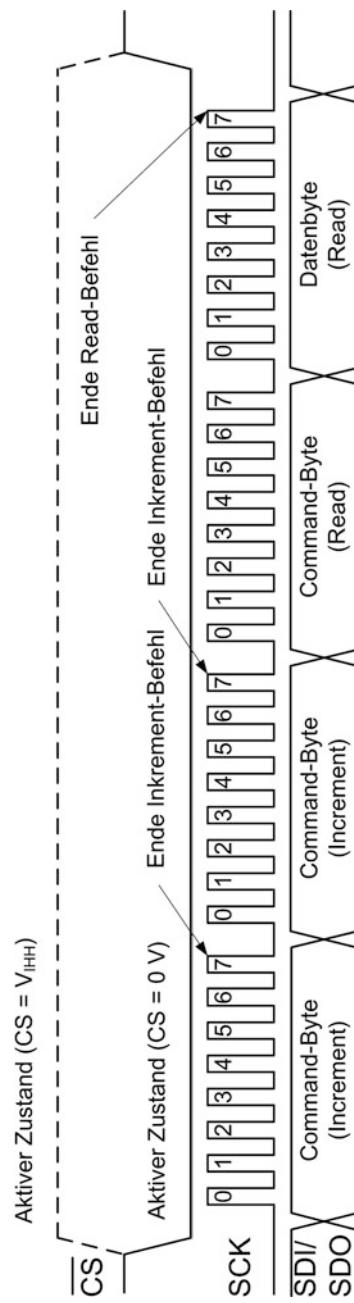


Abb. 7.31 MCP41X1 Verketteten Funktionen

7.4.6 Softwarebeispiel

Im Folgenden werden beispielhaft die Funktionen `Read()` und `Increment()` vorgestellt.

```
uint16_t MCP41X1_Read_Reg(MCP41X1_pins sdevice_pins, uint8_t ucreg_address)
{
    uint8_t ucCommandByte, ucDataIn;
    uint16_t uiDataIn = 0;

    ucCommandByte = ucreg_address | READ_DATA_OPCODE;
    //Gestaltung des Command-Bytes
    SPI_Master_Start(sdevice_pins.MCP41X1spi);
    //die SPI-Übertragung wird gestartet
    //das Command-Byte wird übertragen, das 9. Datenbit des Registers
    //wird eingelesen
    ucDataIn = SPI_Master_Write(ucCommandByte);
    uiDataIn = ucDataIn << 8;
    //die Datenbits 7:0 werden eingelesen; die Übertragung des
    //dummy-Bytes 0xFF ermöglicht
    //die Erkennung eines eventuellen Übertragungsfehler
    ucDataIn = SPI_Master_Write(0xFF);
    uiDataIn |= ucDataIn; //der Inhalt des eingelesenen Registers wird
    //zusammengefasst
    SPI_Master_Stop(sdevice_pins.MCP41X1spi); //die SPI-Übertragung
                                                //wird beendet
    return uiDataIn;
}

uint8_t MCP41X1_Increment_Reg(MCP41X1_pins sdevice_pins,
                               uint8_t ucreg_address)
{
    uint8_t ucCommandByte, ucDataIn;

    ucCommandByte = ucreg_address | INCREMENT_OPCODE; //Gestaltung des
                                                       //Command-Bytes
    SPI_Master_Start(sdevice_pins.MCP41X1spi); //die SPI-Übertragung
                                                //wird gestartet
    ucDataIn = SPI_Master_Write(ucCommandByte); //die Rückmeldung
                                                //wird eingelesen
    SPI_Master_Stop(sdevice_pins.MCP41X1spi); //die SPI-Übertragung
                                                //wird beendet
    return ucDataIn;
}
```

Aus diesen allgemein gehaltenen Funktionen können weitere spezifische Funktionen erstellt werden, wie beispielsweise eine Funktion, die überprüft ob sich der Baustein im Shutdown-Modus befindet. Diese Funktion könnte folgendermaßen aussehen:

```

uint8_t MCP41X1_Get_Status(MCP41X1_pins sdevice_pins)
{
    uint8_t uiDataIn;
    uiDataIn = MCP41X1_Read_Reg(sdevice_pins, STATUS_REGISTER);
    if(uiDataIn & 0x0002)    return SHUTDOWN_ON;
    else return SHUTDOWN_OFF;
}

```

wobei die Konstanten *STATUS_REGISTER*, *SHUTDOWN_ON* und *SHUTDOWN_OFF* natürlich im .h-File als unterscheidbare Ganzzahlkonstanten definiert werden müssen.

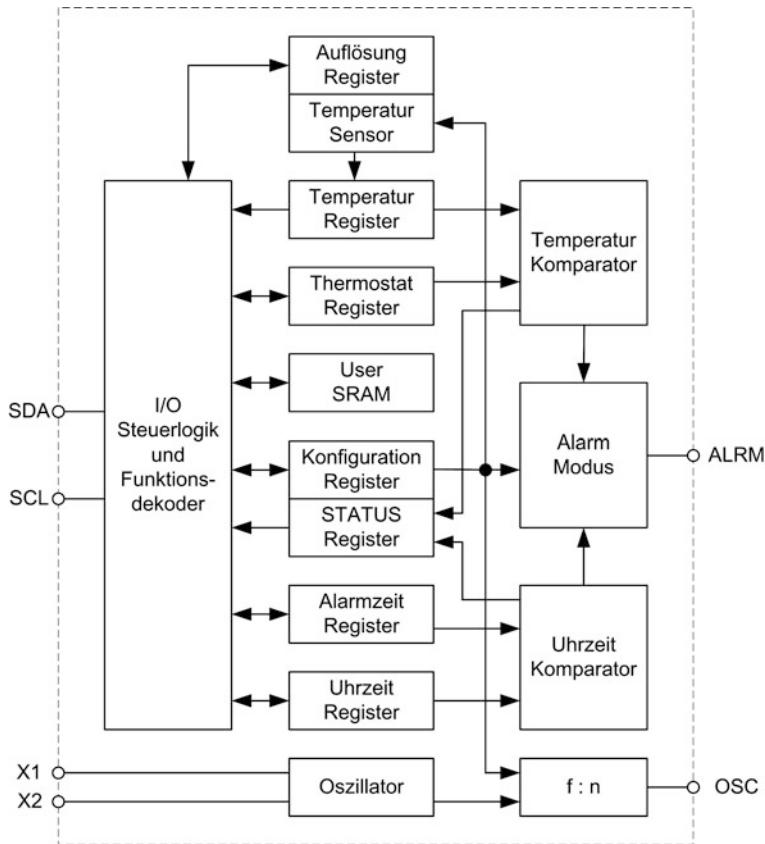
7.5 MAX31629 Real Time Clock (RTC)

Eine Real Time Clock (eine Echtzeituhr) ist ein integrierter Baustein, der eine digitale Uhr implementiert, die die Zeit in menschenlesbaren Einheiten misst und die Informationen für den Abruf bereit stellt. Die Uhr wird von außen eingestellt und läuft, solange sie mit Energie versorgt wird. Beim Ausschalten der Hauptenergiequelle muss die Uhr weiter über eine Batterie, einen Akku oder einen Superkondensator (Supercap) versorgt werden, weil die Uhrzeit in einem SRAM-Speicher gespeichert ist. Aus diesem Grund muss der Energiebedarf solcher Bausteine gering sein. Manche Bausteine haben eine zusätzliche Energiequelle im Gehäuse integriert. Hersteller wie Maxim Integrated, NXP Semiconductors (früher Philips), Texas Instruments u. a. produzieren RTC-Bausteine. Als Taktquelle wird ein Uhrenquarz mit einer Frequenz von 32.768 Hz verwendet. Diese Zahl ist eine Zweierpotenz, so dass durch wiederholtes Halbieren der Grundfrequenz (15 Mal) eine Frequenz von 1 Hz erreicht wird, die einer Periodendauer von einer Sekunde entspricht. Mit dieser Frequenz wird die Uhr angeregt. Für eine angegebene Zeitspanne (meist ein Jahrhundert) werden beim Berechnen vom Datum die Schaltjahre und die unterschiedlichen Monatslängen berücksichtigt. Beispielhaft für solche Bausteine wird im Folgenden ein MAX31629 [12] betrachtet, der die Funktionen einer Echtzeituhr mit Kalender bis zum Jahr 2100 implementiert. Zusätzlich besitzt er einen Temperatursensor (siehe Abb. 7.32), mit dem die Temperatur von -55 bis $+125$ °C mit einer Genauigkeit von ± 2 °C gemessen werden kann und einen 32-Byte großen SRAM-Speicher, der dem Benutzer frei zur Verfügung steht.

Ein Oszillator regt die Echtzeituhr an. Über einen einstellbaren Frequenzteiler kann der Oszillator einen open-drain Ausgang ansteuern, der als Takteingang für einen Mikrocontroller⁹ dienen kann. Der Frequenzteiler wird über das Konfiguration-Register eingestellt, wie in Tab. 7.12 dargestellt ist.

Wenn der Taktausgang nicht gebraucht wird, soll er um Strom zu sparen abgeschaltet werden.

⁹ Beispielsweise für den erniedrigten Takt im Schlafmodus, siehe dazu Abschn. 3.3.2.

**Abb. 7.32** MAX31629 Blockschaltbild**Tab. 7.12** Frequenzteller
Clock

Konfiguration-Register		Clock Ausgang
Bit 7	Bit 6	
0	0	Ausgeschaltet
0	1	:8
1	0	:4
1	1	:1

7.5.1 Zeitmessung

Die Elemente der Uhrzeit: Sekunden, Minuten, Stunden, Wochentage, Tage, Monate und Jahre belegen byteweise in einem flüchtigen Speicherbereich logische Adressen in steigender Reihenfolge angefangen mit 0x00 (Tab. 7.13) und bilden zusammen den Speicherbereich der Uhrzeit-Register. Über den Befehlskode 0xC0 können die Werte gelesen oder geändert werden. Die sieben Werte sind alle zweistellig, weil die Jahre von 0 bis 99 ge-

Tab. 7.13 Speicherorganisation des MAX31629 Bausteins

Register Bezeichnung	Register Elemente	Logische Adresse	Befehlscode	Speicherart	Zugriff
Uhrzeit-Register	Sekunden	0x00	0xC0	SRAM	Schreiben-Lesen
	Minuten	0x01			
	Stunden	0x02			
	Wochentage	0x03			
	Tage	0x04			
	Monate	0x05			
	Jahre	0x06			
Alarmzeit-Register	Sekunden	0x00	0xC7	SRAM	Schreiben-Lesen
	Minuten	0x01			
	Stunden	0x02			
	Wochentage	0x03			
Thermometer-Register	–	2 Byte	0xAA	SRAM	Nur Lesen
Auflösung-Register	–	1 Byte	0xAD	EEPROM	Schreiben-Lesen
Thermostat-TL	–	2 Byte	0xA1	EEPROM	Schreiben-Lesen
Konfiguration/STATUS-Register	–	2 Byte	0xA2	EEPROM	Schreiben-Lesen
	–	1 Byte	0xAC	EEPROM	Schreiben-Lesen
User SRAM	–	32 Bytes	0x17	SRAM	Schreiben-Lesen

zählt werden (entspricht den Jahren 2000 bis 2099) und werden in BCD¹⁰-kodierter Form gespeichert. Um eine Dezimalstelle binär zu kodieren, benötigt man vier Binärstellen oder ein Nibble, für eine zweistellige Zahl werden somit acht Bits oder ein Byte benötigt. Im niederwertigsten Nibble wird die Einerstelle und im höherwertigen Nibble die Zehnerstelle der Dezimalzahl gespeichert. Der folgende Programmcode-Ausschnitt soll die BCD-Dezimal bzw. die Dezimal-BCD Umwandlung erläutern:

```
//Dezimal - BCD Umwandlung
uint8_t ucDecimalNumber, ucBCDNumber;
/*die Einerstelle der Dezimalzahl wird in das niederwertigste Nibble
der Variable ucBCDNumber gespeichert*/
ucBCDNumber = ucDecimalNumber % 10;
//die Zehnerstelle der Dezimalzahl wird berechnet
ucDecimalNumber = ucDecimalNumber / 10;
```

¹⁰ BCD – Binary Coded Decimal.

```

/*die dezimale, binär kodierte Zahl wird mit dem höherwertigen
 Nibble ergänzt*/
ucBCDNumber = ucBCDNumber + (ucDecimalnumber << 4);

//BCD - dezimal Umwandlung
/*das niederwertigste Nibble (die Einerstelle) wird der Dezimalzahl
 zugewiesen*/
ucDecimalNumber = ucBCDNumber & 0x0F;
//die Zehnerstelle wird berechnet
ucBCDNumber = ucBCDNumber >> 4;
//die Dezimalzahl wird mit der Zehnerstelle ergänzt
ucDecimalnumber = ucDecimalNumber + (ucdecimalNumber * 10);

```

Das Sekunden-Byte zählt die Sekunden von 0 bis 59 (binär 0101 1001). Diese Zahlen im BCD-Format benötigen nur sieben Bits. Das höherwertige Bit dieses Bytes, als Clock-Halt-Bit bezeichnet, schaltet den Oszillatator bei „0“ an bzw. bei „1“ aus. Die Stunden werden im 12- oder 24-Stundenformat gezählt. Wenn das Bit 6 vom Stunden-Byte auf „1“ gesetzt ist, dann läuft die Uhr im 12-Stundenformat, die Stundenzahl nimmt Werte zwischen 1 und 12 an und das Bit 5 (AM/PM¹¹) zeigt, wenn es „0“ ist, dass es Vormittag ist. Ist das Bit 6 vom Stunden-Byte „0“, dann läuft die Uhr im 24-Stundenformat und die Stundenzahl nimmt Werte zwischen 0 und 23 an. Die Wochentage werden mit Zahlen von 1 bis 7 kodiert, wobei „1“ für Sonntag steht.

7.5.2 Alarmzeit

Der Baustein MAX31629 bietet auch eine Alarmzeit-Funktion an. Die Alarmzeit, bestehend aus Sekunden, Minuten, Stunden und Wochentag wird in einem flüchtigen Teil des Speichers ab der logischen Adresse 0x00 abgelegt. Die voreingestellte Alarmzeit ist Sonntag, 12:00:00 AM. Die Stundenformate der Uhrzeit und der Alarmzeit müssen gleich sein. Der Alarm kann über das Konfigurationsregister aktiviert, bzw. deaktiviert werden (siehe Tab. 7.14). Ein Alarm wird ausgelöst, wenn die erreichte Uhrzeit gleich der eingestellten Alarmzeit ist. Das führt dazu, dass das Bit 7 (CAF = Clock Alarm Flag) vom Status-Register auf „1“ gesetzt wird und falls im Konfiguration-Register der Alarmmodus 3 oder 4 gewählt ist, dann wird der Alarmausgang aktiv. Der ausgelöste Alarm kann als Folge eines Zugriffs (Lesen oder Speichern) auf die Uhr- oder Alarmzeit-Register abgeschaltet werden. Das Bit 5 (CAL = Clock Alarm Latch) vom Status-Register wird beim ersten Auslösen des Zeitalarms auf „1“ gesetzt und wird mit dem nächsten Einschalten zurückgesetzt.

¹¹ AM – ante meridiem (Vormittag), PM – post meridiem (Nachmittag).

Tab. 7.14 MAX31629 Alarm
Modi

Konfiguration-Register		Alarm-Modus
Bit 5	Bit 4	
0	0	1 – deaktiviert
0	1	2 – nur Temperatur
1	0	3 – nur Zeit
1	1	4 – beide

7.5.3 Temperaturmessung

Der interne Temperatursensor misst die Temperatur mit einer über das Auflösung-Register einstellbaren Bitauflösung von 9 bis 12 Bits, so wie es in Tab. 7.15 aufgelistet ist.

Die Erhöhung der Bitauflösung um 1 führt zu einer Verdoppelung der Messdauer. Diese Dauer muss beim Auslesen des Temperaturregisters berücksichtigt werden, weil das Beenden einer Temperaturmessung vom Baustein nicht signalisiert wird. Der gemessene Temperaturwert wird als Ganzzahl in ein zwei Byte großes Register im Zweierkomplement gespeichert mit den (16-n) niederwertigen Bits auf „0“ gesetzt, wobei n die eingestellte Bitauflösung ist. Das höherwertige Byte des Temperaturregisters beinhaltet den gemessenen Temperaturwert im Zweierkomplement mit einer Auflösung von 1 °C, das niederwertige Byte den Nachkommaanteil. Auf diese Art können Festkommazahlen als Ganzzahlen gespeichert werden.

Die Temperaturmessung kann im Freilauf- oder Einzelmessung-Modus betrieben werden. Der Messmodus wird über das Konfiguration-Register bestimmt (Tab. 7.16).

Im Freilauf-Modus steht im Temperaturregister der letzte gemessene Temperaturwert zum Lesen bereit, ein Lese-Zugriff auf dieses Register beeinflusst eine laufende Messung

Tab. 7.15 MAX31629 – Temperaturauflösung des Thermometers

Bitauflösung [Bit]	Temperaturschritte [°C]	Max. Messdauer [ms]	Auflösung-Register	
			Bit 1	Bit 0
9	0,5	25	0	0
10	0,25	50	0	1
11	0,125	100	1	0
12	0,0625	200	1	1

Tab. 7.16 Temperatur Messmodus

Konfiguration-Register		Messmodus nach Einschalten
Bit 2	Bit 0	
0	0	Freilauf
0	1	Eine Messung; danach Einzelmessung
1	0	Standby; nach dem Start einer Temperaturmessung, Freilauf-Betrieb
1	1	Einzelmessung

nicht. Mit einer Stopp-Messung Funktion (Befehlskode *0x22*) wird der Freilauf-Modus vorübergehend gestoppt, nachdem die aktuelle Umwandlung vollständig beendet ist und der Messwert wird gespeichert. Mit einer Start-Messung Funktion (Befehlskode *0xEE*) werden die gestoppten Messungen im Freilauf-Modus fortgesetzt, bzw. eine neue Messung im Einzelmessung-Modus gestartet. Mit dem folgenden Programmausschnitt kann eine Temperaturmessung gestartet werden:

```
char MAX31629_Start_TempMeasure(void)
{
    #define MAX31629_DEVICE_TYPE_ADDRESS          0x9E
    #define TEMP_MEASURE_START_OPCODE            0xEE

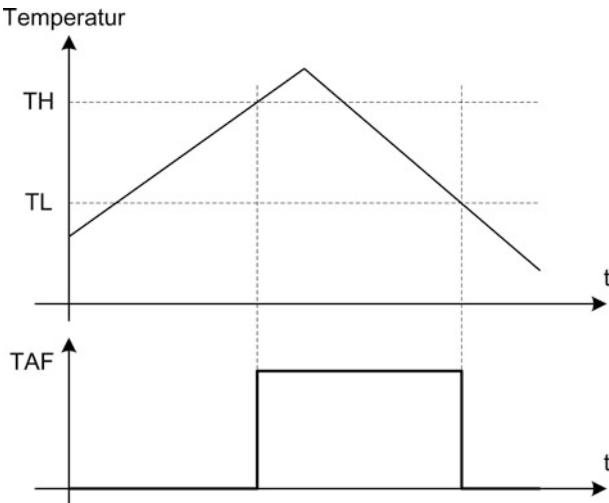
    uint8_t ucDeviceAddress;

    ucDeviceAddress = MAX31629_DEVICE_TYPE_ADDRESS | TWI_WRITE;
    //Write-Modus
    //I2C Master initiiert die Kommunikation
    TWI_Master_Start();
    if((TWI_STATUS_REGISTER) != TWI_START) return TWI_ERROR;
    //Device Adresse senden
    TWI_Master_Transmit(ucDeviceAddress);
    if((TWI_STATUS_REGISTER) != TWI_MT_SLA_ACK) return TWI_ERROR;
    //der Befehlscode der Operation wird gesendet
    TWI_Master_Transmit(TEMP_MEASURE_START_OPCODE);
    if((TWI_STATUS_REGISTER) != TWI_MT_DATA_ACK) return TWI_ERROR;
    // I2C Master beendet die Kommunikation
    TWI_Master_Stop();
    return TWI_OK;
}
```

7.5.4 Thermostat mit Alarmfunktion

Um eine Thermostat-Funktion implementieren zu können, besitzt der MAX 31629 zwei 16-Bit-Register, die die obere und die untere Temperaturgrenze speichern. Es werden zwei Werte gebraucht um eine einstellbare Hysterese zu gewährleisten. Das Speicherformat dieser zwei Werte ist gleich dem Format des Thermometers. Sobald die gemessene Temperatur die eingestellte obere Temperaturgrenze erreicht oder überschreitet, schaltet der Temperaturkomparator (Abb. 7.32) um und das Bit 6 (TAF – Thermal Alarm Flag) im Status-Register wird auf „1“ gesetzt. Falls der Alarmmodus 1 oder 3 (Tab. 7.14) gewählt ist, wird der Alarmausgang aktiv(Abb. 7.34). Wenn das TAF-Bit zum ersten Mal gesetzt wird, wird auch das Bit 4 (TAL – Thermal Alarm Latch) im Status-Register gesetzt und erst beim nächsten Einschalten zurückgesetzt. Dieses Bit zeigt an, wenn es auf „1“ gesetzt ist, dass die Temperatur wenigstens einmal die obere Temperaturgrenze erreicht oder überschritten hat. Das TAF-Bit wird zurückgesetzt, sobald die gemessene Temperatur die

Temperatur Register																	
MSB								LSB									
±	2^6	2^5	2^4	2^3	2^2	2^1	2^0	,	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	°C

Abb. 7.33 MAX31629 Temperatur Kodierung**Abb. 7.34** MAX 31629 Temperaturalarm

untere Temperaturgrenze erreicht oder unterschreitet. Die Temperaturauflösung des Thermometers und des Thermostats müssen nicht gleich sein.

Die Temperaturgrenzwerte müssen vor der Übertragung entsprechend Abb. 7.33 kodiert werden. Dafür kann der folgende Programmcode verwendet werden.

```
uint16_t MAX31629_Set_FloatToIntTemp(float ftemp)
{
    uint8_t ucInteger, ucDecimalPlace, ucSign = 0;
    uint16_t uiValue;

    if(ftemp < 0)
    {
        ucSign = 1; //die Variable ucSign speichert das Vorzeichen
                    //des Temperaturwertes
        ftemp = ftemp * (float)(-1.); //der Betrag des negativen
                                      //Wertes wird berechnet
    }
    ucInteger = ftemp; //ucInteger speichert die Ganzzahl
                      //des Temperaturwertes
    ftemp = ftemp - ucInteger; //die Nachkommastelle wird berechnet
    ucDecimalPlace = ftemp * 16; //Abrundung auf die 12 Bit Auflösung
```

```

ucDecimalPlace = ucDecimalPlace << 4;
//das Low-Byte des Temperaturwerts wird erstellt
//der Temperaturwert als positive 2-Byte Zahl wird erstellt
uiValue = (ucInteger << 8) + ucDecimalPlace;
if(ucSign)
{
    //falls der Temperaturwert negativ ist, wird der
    //Zweierkomplement gebildet
    uiValue = (~uiValue) + 1;
}
return uiValue;
}

```

Beim Aufruf der Funktion *MAX31629_Set_FloatToIntTemp* wird als Parameter der Temperaturwert als Festkommazahl übergeben, die Funktion rechnet diesen Wert um und beschränkt das Ergebnis auf eine 12-Bit-Auflösung und gibt dann den kodierten Wert als 2-Byte-Zahl zurück. Die Funktion kann sowohl positive, als auch negative Temperaturwerte codieren.

7.5.5 I²C Kommunikation

Die Kommunikation zwischen einem Mikrocontroller, der als Master konfiguriert ist und dem Baustein MAX31629 als Slave erfolgt über I²C bei einer Bitrate von max. 400 kBit/s. Die Device Typ Adresse des Bausteins ist 0x9E. Die einzelnen Speicherbereiche des Bausteins (Tab. 7.13) können über Befehlscodes adressiert und gelesen, bzw. geändert werden. Unterschiedliche Zugriffs-Vorgänge werden in Abb. 7.35 dargestellt.

- Das Speichern in einem einzigen 2-Byte-Register wie unter anderem beim Thermometer oder Thermostat ist in a) dargestellt. Nachdem der Master mit einer Start-Sequenz die Kommunikation initiiert hat und feststellt, dass der Bus frei ist, sendet er die Device Adresse des Bausteins mit dem R/W-Bit auf „0“ (Schreib-Zugriff). Weiterhin wird der Befehlscode für das gewünschte Register übertragen und anschließend zwei Datenbytes mit dem höherwertigen Byte zuerst. Mit einer Stopp-Sequenz beendet der Master die Kommunikation und gibt den Bus wieder frei. Der Slave antwortet nach jedem empfangenen Byte mit ACK. Das Speichern in das Auflösung-oder Konfiguration-Register erfolgt ähnlich, nur, dass der Master die Kommunikation nach dem Senden des 1. Datenbytes beendet.
- Das Lesen eines 2-Byte-Registers wie die Thermometer-, Thermostat- oder Konfiguration-und Status-Register erfolgt nach dem Vorgang b), der bis zum Senden des Befehlscodes gleich mit dem bei a) beschriebenen Vorgang ist. Mit einer Restart-Sequenz (eine erneute Start-Sequenz) nach dem Senden des Befehlscodes und der Antwort des Slaves sendet der Master die Bausteinadresse mit dem R/W-Bit auf „1“.

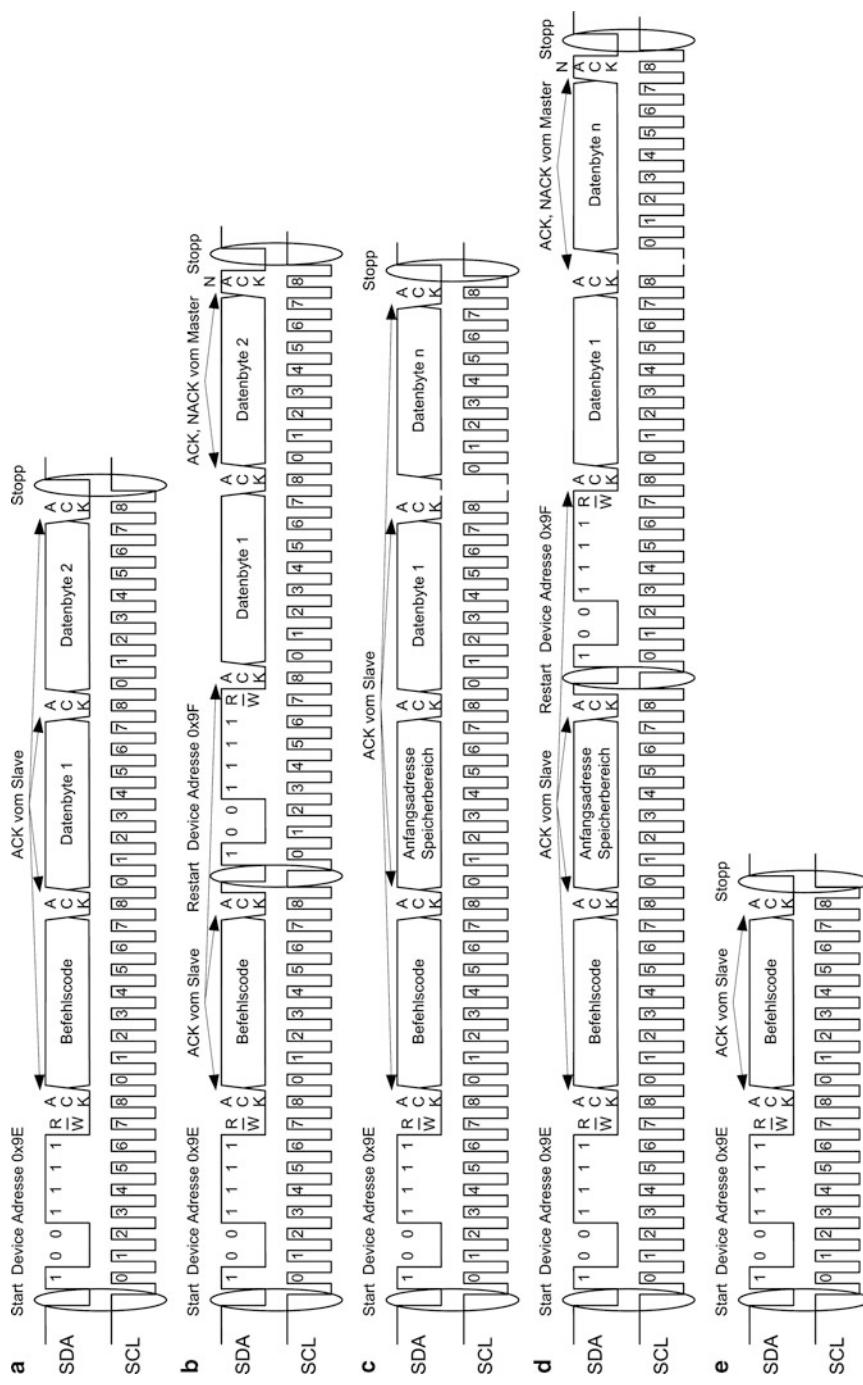


Abb. 7.35 MAX31629 I²C-Kommunikation

Damit wird dem Slave signalisiert, dass er mit den nächsten acht Clock-Takten das höherwertige Byte des gewünschten Registers auf der Datenleitung ausgibt. Der Master bestätigt den Empfang des Bytes mit ACK und der Slave sendet auch das zweite Byte, das von dem Master mit einem NACK quittiert wird. Das signalisiert dem Slave, dass er keine weiteren Bytes senden soll und der Master beendet die Kommunikation mit einer Stopp-Sequenz. Wenn der Master nur ein Byte empfangen will, wie beim Lesen der Temperatur als Ganzzahl, dann antwortet er mit NACK nach dem ersten Byte und anschließend beendet er die Kommunikation.

- c) Der Vorgang c) zeigt den Verlauf eines Schreib-Zugriffs auf einem Speicherbereich, dessen Bytes einzeln adressierbar sind wie beispielsweise die Uhrzeit-, Alarmzeit-Register oder der Benutzer-SRAM. Der Unterschied zu a) besteht darin, dass der Master nach dem Senden des Befehlscodes, die logische Anfangsadresse des Speicherbereichs gefolgt von den Datenbytes senden muss. Nach dem Empfang eines Datenbytes wird der interne Adresszähler des MAX31629 inkrementiert. Der Benutzer kann innerhalb eines Speicherbereichs die Anfangsadresse und die zu übertragende Anzahl von Bytes frei wählen. So ist es möglich, gezielt Bytes zu ändern, beispielsweise bei der Uhrzeitstellung nur die Minuten und die Sekunden zu nullen.
- d) Um einen ganzen oder nur ein Teil eines adressierbaren Speicherbereichs auszulesen verwendet man den Vorgang d). Zusätzlich zum Vorgang b) sendet der Master nach dem Befehlscode die Anfangsadresse des gewünschten Speicherbereichs. Das letzte empfangene Byte wird von dem Master mit NACK, alle anderen mit ACK quittiert.
- e) Mit diesem Vorgang wird die Temperaturmessung gestoppt (Befehlscode 0x22), oder gestartet (Befehlscode 0xEE).

7.6 SI4840 Radio-IC

Der Baustein SI4840 [13–15] von der Fa. Silicon Laboratories ist ein AM¹²/FM¹³-Empfänger mit einem Stereo-Decoder so wie es in Abb. 7.36 dargestellt ist. Er gehört zu einer Reihe von ATDD¹⁴-Bausteinen, die einen analogen Tuner besitzen, was bedeutet, dass die Frequenzabstimmung über eine externe analoge Spannung realisiert wird und die Informationen über die Einstellungen in digitaler Form vom Baustein übermittelt werden. Der Baustein integriert alle benötigten Baugruppen für den Empfang der Mittelwelle- und Ultrakurzwelle-Frequenzbänder (siehe Tab. 7.15) mit einer sehr einfachen Außenbeschaltung, muss aber von einem Mikrocontroller angesteuert werden. Der Baustein ermöglicht die Anzeige eines Stereo-Empfangs, den Empfang eines gültigen Senders, sowie eine digitale Einstellung der Tonhöheregelung und der Lautstärke. Um eine gute Audioqualität unter unterschiedlichen Empfangsbedingungen zu gewährleisten, wird abhängig von der

¹² AM – Amplitudenmodulation.

¹³ FM – Frequenzmodulation.

¹⁴ ATDD – Analog Tune Digital Display.

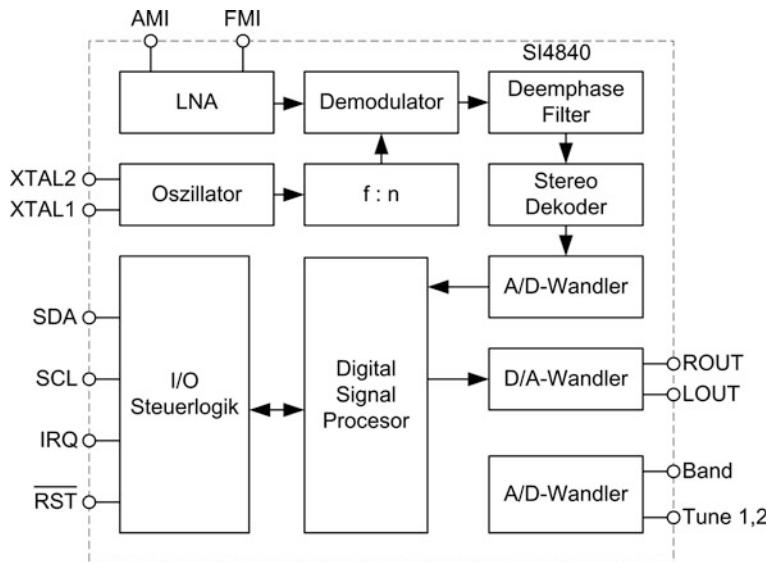


Abb. 7.36 SI4840-Blockschaltbild

Empfangsqualität automatisch zwischen Monophonie und Stereophonie umgeschaltet. Es werden dafür drei Empfangsqualitätsparameter überwacht:

- die Empfangsfeldstärke – RSSI¹⁵;
- der Signal-Rausch-Abstand (oder Störabstand) – SNR¹⁶;
- Intersymbol-Interferenzen wegen dem Mehr-Wege-Empfang.

Sobald einer dieser drei Parameter unter einer digital einstellbaren Grenze fällt, wird auf Mono-Betrieb umgeschaltet.

7.6.1 Bausteinbeschreibung

Das Rundfunksignal wird über eine passende Antenne (siehe [15]) empfangen, die an Pin FMI (für UKW¹⁷) oder AMI (für MW¹⁸) angeschlossen ist und mit einem rauscharmen Verstärker LNA¹⁹ verstärkt. Der Demodulator benötigt ein Taktsignal mit einer Frequenz von 32.768 Hz. Dieses Taktsignal kann mit Hilfe eines Uhrenquarzes mit ei-

¹⁵ RSSI – Received Signal Strength Indication.

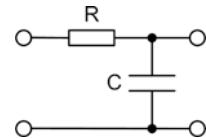
¹⁶ SNR – Signal to Noise Ratio.

¹⁷ UKW – Ultrakurzwelle.

¹⁸ MW – Mittelwelle.

¹⁹ LNA – Low Noise Amplifier.

Abb. 7.37 SI4840 Deemphaser-Filter



ner Genauigkeit von $\pm 100 \text{ ppm}$ ²⁰ generiert, oder mit einem externen Oszillatoren erzeugt. Die Frequenz des externen Oszillators wird mit dem internen einstellbaren Frequenzteiler ($:1 \dots 4095$) auf die Frequenz $32.768 \pm 5\%$ Hz eingestellt. Bei der Übertragung des modulierten Audiosignals werden die höheren Frequenzanteile stärker gestört, was zu einer Verschlechterung des Signal-Rausch-Abstandes führt. Eine Verbesserung schafft man, indem auf der Senderseite vor der Modulation die hochfrequenten Anteile des Signals mit einem so genannten Preemphase-Filter (Hochpassfilter 1. Ordnung) angehoben werden. Weil nur das Nutzsignal verstärkt wird, während die Rauschleistung von der Übertragungsstrecke konstant bleibt, erhöht sich der Signal-Rausch-Abstand. Um auf der Empfängerseite einen natürlichen Klang zu erreichen, muss das Signal mit einem Deemphaser-Filter (Tiefpassfilter 1. Ordnung Abb. 7.37) wieder hergestellt werden, was zur Absenkung der höheren Frequenzanteile führt. In Europa wird ein Tiefpassfilter mit einer -3 dB -Grenzfrequenz von 3,18 kHz verwendet

$$f_{-3\text{dB}} = \frac{1}{2\pi RC} \quad (7.9)$$

und einer Zeitkonstante von

$$\tau = RC = 50 \mu\text{s.} \quad (7.10)$$

Mit diesem Rauschunterdrückungsverfahren ist eine Verbesserung des Rausch-Signal-Abstandes von bis zu 7,8 dB möglich [16]. In den USA wird ein Filter mit einer Zeitkonstante von 75 μs verwendet, deshalb muss auf die Wahl der Zeitkonstante geachtet werden.

Am Anschluss TUNE1 wird eine analoge Spannung erzeugt, die zur Auswahl des Frequenzbands, bzw. der Frequenzabstimmung dient. An den Anschlüssen ROUT und LOUT kann ein Kopfhörer oder ein Audioverstärker angeschlossen werden. Der Anschluss RST dient zum sicheren Rücksetzen des Bausteins. Für die Kommunikation mit einem Mikrocontroller stehen die Anschlüsse SDA, SCL, und IRQ zur Verfügung.

7.6.2 Auswahl des Frequenzbandes und Frequenzabstimmung

Der Baustein kann das Frequenzband selbst detektieren. Er erzeugt am Pin TUNE1 die analoge Spannung, die für die Frequenzabstimmung benötigt wird. Um das zu realisieren

²⁰ ppm – parts per million; 1 ppm = 0,000001.

Tab. 7.17 SI4840 empfangene Frequenzbänder

	Modulationsart	Frequenzbereich
FM		87–108 MHz
		86,5–109 MHz
		87,3–108,25 MHz
		76–90 MHz
		64–87 MHz
AM		520–1710 kHz
		522–1620 kHz
		504–1665 kHz
		520–1730 kHz
		510–1750 kHz

muss zwischen dem Anschluss TUNE1 und Masse ein präziser Spannungsteiler mit einem gesamten Widerstand von 500 kOhm angeschlossen werden. Über einen Schalter, der am Pin BAND angeschlossen ist, kann ein Frequenzband ausgewählt werden. Beispiele für Spannungsteiler findet man in [15]. Ein Potentiometer, das zwischen TUNE1 und Masse angeschlossen ist und dessen Schleifer mit dem Pin TUNE2 verbunden ist, sorgt für die Frequenzabstimmung. Um den Vorgang zu erleichtern bietet der Baustein Informationen über die eingestellte Frequenz und Qualität des Empfangs in digitaler Form an.

In Abb. 7.38 wird eine Schaltung für die Frequenzband-Auswahl und Frequenzabstimmung vorgestellt, die schaltungstechnisch einfach ist. Wenn der Anschluss BAND des Bausteins SI4840 auf High geschaltet ist, dann erwartet dieser, dass der Mikrocontroller das Frequenzband auswählt und es seriell übermittelt. Mit dem LNA_EN Pin direkt auf High (ohne Pull-up-Widerstand) können die Grenzen der Frequenzbänder frei gewählt werden, ansonsten gelten die Standardwerte aus der Tab. 7.17. Die für die Frequenzabstimmung benötigte analoge Spannung wird mit Hilfe eines D/A-Wandlers vom Typ MCP4911 erzeugt, der als Referenzspannung die Spannung am Pin TUNE1 verwendet. Der MCP4911 ist ähnlich wie die D/A-Wandler der Reihe MCP48XX (siehe Abschn. 7.3.1) aufgebaut und bis auf den VREF Pin mit einem MCP4811 pinkompatibel. Für seine Ansteuerung können die gleichen Funktionen wie für die MCP48XX-Bausteine verwendet werden. Bei der Wahl eines anderen D/A-Wandlers muss auf die Begrenzung seiner Ausgangsspannung auf den Spannungspegel vom Pin TUNE1 geachtet werden.

7.6.3 Initialisieren des Bausteins

Nachdem die Versorgungsspannung des Bausteins ihren stabilen Zustand erreicht hat, muss der Baustein zunächst initialisiert werden. Dafür muss der steuernde Mikrocontroller den Pin RST des Bausteins auf „0“ setzen und ihn für mindestens 100 µs auf diesem Pegel halten. Der Baustein schaltet als Antwort das Signal IRQ auf High und spätestens nach 2 ms wieder auf Low. Nach der fallenden Flanke des IRQ-Signals befindet sich der Baustein im Standby-Modus und akzeptiert die Befehle Get_Status und Power_Up. In die-

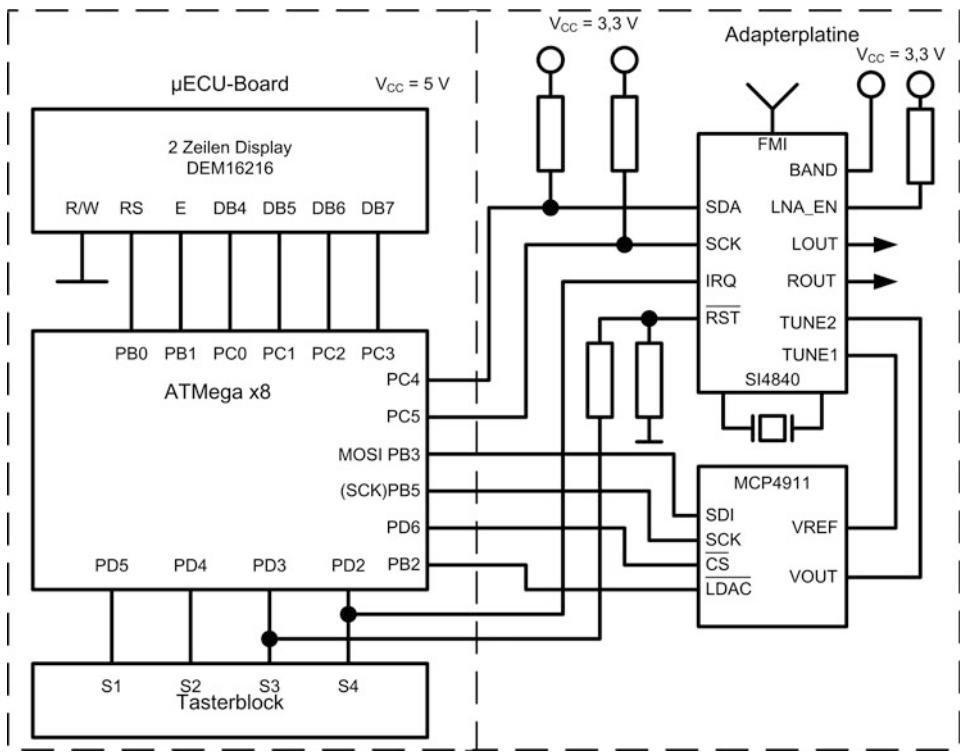


Abb. 7.38 SI4840 Fequenzband-Auswahl und Frequenzabstimmung über einen Mikrocontroller

ser Phase, nach dem Übertragen eines Befehls, muss der Mikrocontroller noch 2 ms auf die Antwort des Bausteins warten oder im Polling testen, wann das Bit CTS im Status-Byte „1“ ist.

7.6.4 Kommunikation mit dem Baustein

Mit dem Baustein kann man mit einer Ansteuerung von einem Mikrocontroller einen kompletten Rundfunkempfänger aufbauen. Der Baustein implementiert ein I²C- (mit den Anschlüssen SCL und SDA) und ein SMB- (SCL, SDA und IRQ) konformes Protokoll und ist als Slave mit der Device Typ Adresse 0x22 vorkonfiguriert. Die serielle Schnittstelle kann mit bis zu 400 kHz getaktet werden.

Als Elemente der Kommunikation unterscheidet man zwischen Befehlen, dazugehörigen Parametern, Einstellungen und Antworten. Auch wenn der Baustein als MW-Empfänger arbeiten kann, werden hier nur die Befehle und Einstellungen für den FM-Empfänger beschrieben, MW spielt technisch keine Rolle mehr. Der Übertragungsverlauf eines Befehls ist in der Abb. 7.39 zu sehen. Nach einer I²C-Startsequenz sendet der

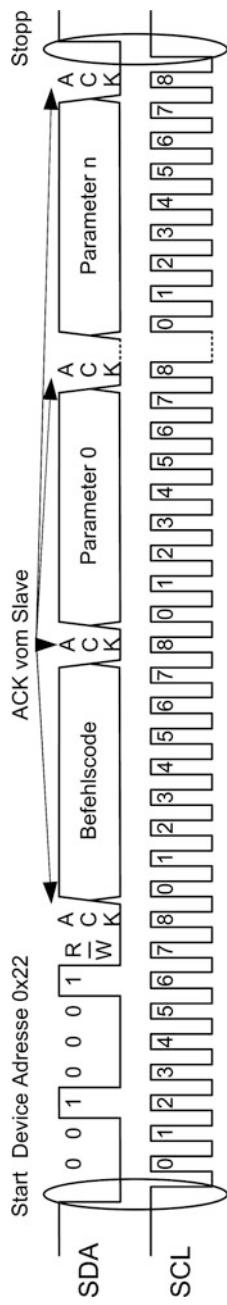


Abb. 7.39 SI4840 Befehlaufbau

Master die Device Adresse mit dem R/W-Bit auf „0“ (Schreib-Vorgang), gefolgt vom gewünschten Befehlscode und 0...n Parametern. Den erfolgreichen Empfang eines jeden Bytes bestätigt der Slave mit ACK. Nach dem Senden der gesamten Botschaft beendet der Master mit einer Stopp-Sequenz die Kommunikation. Während des internen Umsetzens des Befehls reagiert der Baustein nicht auf weitere Befehle. Wenn der interne Vorgang abgeschlossen ist, kann der Master die Antwort des Bausteins anfordern. Das geschieht, indem nach der Start-Sequenz die Device Adresse mit dem R/W-Bit auf „1“ gesendet wird. Mit den nächsten acht Taktzyklen empfängt der Master das erste Byte, das so genannte Status-Byte, das Informationen in kodierter Form über den letzten Befehl liefert. Wenn der Master keine weiteren Bytes benötigt, quittiert er das empfangene Byte mit NACK, ansonsten mit ACK. Nach dem letzten empfangenen Byte der Antwort beendet der Master mit einer Stopp-Sequenz die Kommunikation. Die Antwort des Empfängers auf einem Befehl besteht aus mindestens einem Byte. Das Bit 7 des Status-Bytes hat immer die gleiche Bedeutung, CTS²¹, und zeigt mit „1“ an, dass der Baustein für den Empfang eines neuen Befehls bereit ist.

Die folgende Funktion liest das Statusbyte aus um das CTS-Bit auszuwerten:

```
uint8_t SI4840_Get_StatusByte(void)
{
    uint8_t ucAddress = 0x23, ucStatusByte;
    TWI_Master_Start();
    if((TWI_STATUS_REGISTER) != TWI_START) return TWI_ERROR;
    TWI_Master_Transmit(ucAddress); //Device-Adresse senden
    if((TWI_STATUS_REGISTER) != TWI_MR_SLA_ACK) return TWI_ERROR;
    ucStatusByte = TWI_Master_Read_NACK();
    if((TWI_STATUS_REGISTER) != TWI_MR_DATA_NACK) return TWI_ERROR;
    TWI_Master_Stop();
    return ucStatusByte;
}
```

Die Funktion gibt nach dem Aufruf entweder einen Fehlercode oder das Statusbyte des Empfängers zurück. Durch das wiederholte Aufrufen der Funktion und Auswertung des CTS-Bits kann festgestellt werden, wann der Baustein bereit ist, einen neuen Befehl auszuführen.

7.6.4.1 Power_Up Befehl

Eine Power_Up-Funktion kann aufgerufen werden um den Baustein aus dem Standby- in den aktiven Zustand zu versetzen oder um das Frequenzband oder deren Eigenschaften zu ändern. Der Befehl wird vom Baustein erst nach einer erfolgreichen Reset-Sequenz angenommen. Der Master kann nach der Device Adresse und dem Power_Up-Befehlscode 0x91 bis zu sechs Parameter senden, die folgende Bedeutung haben:

²¹ CTS – Clear to Send (Empfangsbereitschaft).

- Parameter 1
 - Bit 7** – wird auf „1“ gesetzt, wenn der Empfänger einen 32.768 kHz Quarz verwendet
 - Bit 6** – codiert die Wartezeit für die Stabilisierung der Schwingungen des Quarzes (0 für 600 ms)
 - Bit 5:0** – Frequenzbandindex; für FM ist eine Zahl zwischen 0 bis 19, die das Frequenzband, die Zeitkonstante des Deemphase-Filters, den Abstand zwischen den Audiokanälen und die Grenze der Empfangsfeldstärke kodiert.
- Parameter 2:3 entsprechen zusammen der unteren Frequenzgrenze des Frequenzbands in 10 kHz
- Parameter 4:5 entsprechen zusammen der oberen Frequenzgrenze des Frequenzbands in 10 kHz
- Parameter 6 kodiert den Frequenzkanalabstand; für FM ist der Parameter 6 = 10

Der Befehl Power_Up könnte folgendermaßen implementiert werden:

```
uint8_t SI4840_Set_ATDDPowerUp(uint8_t ucband_index, uint16_t
                                 uibottom_freq, uint16_t uitop_freq, uint8_t ucchannel_spacing)
{
    uint8_t ucAddress, ucDataToSend[7];
    ucDataToSend[0] = SI4840_ATDD_POWER_UP; //0xE1
    ucDataToSend[1] = ucband_index; //Frequenzbandindex
    ucDataToSend[2] = uibottom_freq >> 8; //Highbyte der unteren
                                              //Frequenzgrenze
    ucDataToSend[3] = uibottom_freq; //Lowbyte der unteren Frequenzgrenze
    ucDataToSend[4] = uitop_freq >> 8; //Highbyte der oberen
                                              //Frequenzgrenze
    ucDataToSend[5] = uitop_freq; //Lowbyte der unteren Frequenzgrenze
    ucDataToSend[6] = ucchannel_spacing; //Frequenzkanalabstand
    ucAddress = SI4840_DEVICE_TYPE_ADDRESS | TWI_WRITE; //Write-Modus
    TWI_Master_Start();

    TWI_Master_Transmit(ucAddress); //Device Adresse senden
    if((TWI_STATUS_REGISTER) != TWI_MT_SLA_ACK) return TWI_ERROR;
    //Befehlscode, Frequenzband, untere und obere Frequenz und
    //Kanalabstand senden
    for(uint8_t ucI = 0; ucI < 7; ucI++)
    {
        TWI_Master_Transmit(ucDataToSend[ucI]);
        if((TWI_STATUS_REGISTER) != TWI_MT_DATA_ACK) return TWI_ERROR;
    }
    TWI_Master_Stop();
    return TWI_OK;
}
```

Mit dem Aufruf:

```
SI4840_Set_ATDDPowerUp(0x03, 8800, 10800, 10);
```

wird der Baustein auf das Frequenzband 88...108 MHz mit 50 µs Zeitkonstante, 12 dB Audiokanalabstand und 28 dB Empfangsfeldstärke-Grenze eingestellt.

Auf den Power_Up-Befehl antwortet der Baustein nur mit dem Status-Byte. Wenn das Bit 6 (ERR) des Status-Bytes gesetzt ist, ist ein Fehler beim Ausführen des Befehls aufgetreten. Die Bits 3:0 speichern den Fehlercode.

7.6.4.2 Power_Down Befehl

Mit dem Befehl Power-Down (Befehlscode *0x11*) wird der Baustein in den Standby-Modus versetzt. Die Stromaufnahme beträgt in diesem Modus ca. 10 µA. Für diesen Befehl werden keine Parameter benötigt. Während sich der Baustein im Standby Modus befindet akzeptiert er nur die Befehle Power_Up und Get_Status. Der Baustein steuert die Bits 7 (CTS) und 6 (ERR) im Status-Register als Antwort auf diesen Befehl.

7.6.4.3 Get_Status Befehl

Der Master fordert mit dem Befehl Get_Status (Befehlscode *0xE0*) von dem Baustein Informationen über den allgemeinen Status und die Konfiguration des Empfängers an. Es werden keine zusätzlichen Parameter benötigt. Die Antwort auf diesen Befehl besteht aus vier Bytes, die folgende Informationen liefern:

- Status-Byte (Byte 1)

Bit 7 – CTS

Bit 6 – HOSTRST – wenn es „1“ ist, muss der Baustein zurückgesetzt werden

Bit 5 – HOSTPWRUP – wenn es „1“ ist, muss der Befehl Power_Up ausgeführt werden

Bit 4 – INFORDY – zeigt bei „1“ an, dass Informationen über den Tuner vorhanden sind

Bit 3 – STATION – wenn es „1“ ist, dann wird ein stabiler Sender empfangen

Bit 2 – STEREO – zeigt bei „1“ an, dass eine Stereo Sendung empfangen wird

Bit 1 – BCFG1 – zeigt bei „1“ an, dass der Baustein Einstellungen des Frequenzbands abweichend von den Standardwerten akzeptiert

Bit 0 – BCFG2 – zeigt bei „0“ an, dass der Baustein das Frequenzband detektiert

- Byte 2

Bit 7:6 – BANDMODE – für FM = 0, für AM = 1;

Bit 5:0 – diese Bits kodieren den detektierten Frequenzbandindex.

- Byte 3:4 – diese 2 Bytes kodieren im BCD-Format den Frequenzkanal als vierstellige Zahl; eine Codesequenz für die Umrechnung der Frequenz aus dem BCD-Code ist im Abschn. [7.6.5](#) vorgestellt.

7.6.4.4 Audio-Mode Befehl

Mit diesem Befehl können grundlegende Audioeinstellungen geändert bzw. abgefragt werden. Der Master muss nach dem Befehlscode *0xE2* einen Parameter übertragen, dessen Bits folgende Bedeutung haben:

Bit 7 – wenn dieses Bit „1“ ist, werden die Einstellungen abgefragt und folgende Bits haben keine Bedeutung

Bit 6:5 – sind reserviert

Bit 4 – bei „0“ wird der Empfang einer Stereosendung angezeigt

Bit 3 – bei „0“ wird die Lautstärke rund um die Senderfrequenz um 2 dB gedämpft

Bit 2 – bei „0“ wird der Stereo-Modus gewählt

Bit 1:0 – kodieren den Audiomodus: **0** – erlaubt eine digitale Lautstärke-Einstellung zwischen 0 und 63, ohne Tonhöheregelung; **1** – der Lautstärke-Pegel ist auf 59 fixiert, die Tonhöheregelung ist erlaubt; **2** – die Einstellung der Lautstärke von 0 bis 59 und die Tonhöheregelung sind erlaubt; **3** – die Einstellung der Lautstärke von 0 bis 63 und die Tonhöheregelung sind erlaubt.

Die Antwort des Bausteins auf diesem Befehl besteht aus einem Status-Byte mit dem Bit 7 (CTS), Bit 6 (ERR) und die Bits 4:0 mit der gleichen Bedeutung, wie oben beschrieben. Das Bit 5 ist reserviert.

7.6.4.5 Set_Property-Befehl

Eigenschaften des Bausteins wie Taktfrequenz, Frequenzteiler, Lautstärke, Hoch- und Tieftonregelung und Ein- und Abschalten eines Audiokanals können mit Hilfe des Befehls Set_Property geändert werden. Um das zu realisieren, sendet der Master nach dem Befehlscode *0x12* ein Dummy-Byte das gleich 0 ist, gefolgt von dem 2-Byte großen Eigenschaftscode und dem 2-Byte großen Eigenschaftswert. Das höherwertige Byte wird immer zuerst übertragen. Auf diesen Befehl liefert der Baustein keine Antwort.

Eine genaue Beschreibung aller Eigenschaften findet man in [14].

7.6.4.6 Get_Property-Befehl

Die mit dem Befehl Set_Property geänderten Eigenschaften, können mit Hilfe des Befehls Get_Property abgefragt werden. Der Master muss dafür den Befehlscode *0x13* senden, gefolgt von einem Dummy-Byte das gleich 0 ist und dem 2-Byte großen Eigenschaftscode. Auf diesen Befehl antwortet der Baustein mit dem Status-Byte (CTS und ERR aktiv), gefolgt von einem Byte das immer 0 ist und die weiteren 2 Bytes kodieren den gefragten Eigenschaftswert.

7.6.4.7 Get_Rev-Befehl

Beim Aufrufen der Funktion Get_Rev (Befehlscode *0x10*) werden keine zusätzlichen Parameter benötigt. Die Antwort des Bausteins auf den Befehl besteht aus dem Status-Byte mit den aktiven Bits CTS und ERR sowie weitere 8 Bytes, die Informationen über

den Baustein liefern. Diese Informationen können in einer komplexen Schaltung zu seiner Identifizierung dienen.

7.6.5 Sendersuche mit dem SI4840

In der Abb. 7.38 steuert der Mikrocontroller die analoge Abstimmspannung über den D/A-Wandler. Der SI4840 liefert Informationen über die erreichte Frequenz, den Sender- oder Stereo-Empfang. Aufgrund dieser Tatsachen ist die Überlegung nahe, bequem per Tastendruck von Frequenzkanal zu Frequenzkanal zu schalten, nach dem nächsten Sender, der nächsten Stereo-Sendung oder gezielt nach einer Frequenz zu suchen. Beispielhaft wird die Suche nach einem Frequenzkanal, dessen Frequenz als Vielfaches von 100 kHz in der Variable uiFrequency_Old gespeichert ist, vorgestellt. Die SPI-Datenstruktur des D/A-Wandlers ist ähnlich mit dem vom MCP48XX aufgebaut (siehe Abschn. 7.3.1.1) und lautet:

```
MCP48XX_pins  MCP491X_1 = {{  /*CS_DDR*/      &DDRD,
                                /*CS_PORT*/     &PORTD,
                                /*CS_pin*/      PD6,
                                /*CS_state*/    ON},
                                /*LDAC_DDR*/    &DDRB,
                                /*LDAC_PORT*/   &PORTB,
                                /*LDAC_pin*/    PB2,
                                /*LDAC_state*/  ON,
                                /*SHDN_DDR*/    OFF,
                                /*SHDN_PORT*/   OFF,
                                /*SHDN_pin*/    OFF,
                                /*SHDN_state*/  OFF};
```

Die Funktion MCP48XX_Set_Output mit dem Prototyp:

```
void MCP48XX_Set_Output(MCP48XX_pins sdevice_pins, uint8_t ucdevice,
                        uint8_t ucchannel, uint8_t ucout, uint8_t ucgain,
                        uint16_t uidigital_value)
```

wandelt den binären Wert uidigital_value in eine analoge Spannung bei festgelegtem Verstärkungsfaktor um. Dieser Wert wird bei jedem Durchlauf der Schleife inkrementiert und auf den Wert 1023 begrenzt. Im nächsten Schritt wird der Status des Empfängers mit der Funktion SI4840_Get_ATDDStatus abgefragt und die Antwort des Bausteins mit der Funktion SI4840_Get_Response in das Array ucStatusVector[] gespeichert. Die daraus resultierende Frequenz wird berechnet und mit der Zielfrequenz verglichen. Wenn die zwei Frequenzen gleich sind, ist die Suche beendet. Während der Suche ist die Lautstärke abgeschaltet.

```
while(uiFrequencyNew != uiFrequencyOld)
{
    uiVoltage++; //die Abstimmspannung wird erhöht
    if(uiVoltage > 1023)
    {
        uiVoltage = 1023; //die Abstimmspannung hat die obere Grenze
                           //erreicht
        return VOLTAGE_ERROR; //die Funktion wird verlassen mit einem
                           //Fehlercode
    }
    MCP48XX_Set_Output(MCP491X_1, MCP491X, CHANNEL0, OUTPUT_ENABLE,
                        GAINX1, uiVoltage);
    if(SI4840_Get_ATDDStatus()) //das Status-Register wird abgefragt
    {
        TWI_Master_Stop(); //die Kommunikation wird beendet
        return TWI_ERROR; //ein Übertragungsfehler hat stattgefunden
    }
    if(SI4840_Get_Response(ucStatusVector, STATUS_RESPONSE_BYTE))
    { //die Antwort des Bausteins wird gelesen
        TWI_Master_Stop();
        return TWI_ERROR;
    }
    uiFrequencyNew = (ucStatusVector[3] & 0x0F) +
                    (ucStatusVector[3] >> 4) * 10 + (ucStatusVector[2] &
                    0x0F) * 100 + (ucStatusVector[2] >> 4) * 1000;
    for(uint16_t li = 0; li < 50000; li++); //Wartezeit
}
```

Literatur

1. Semiconductors, N.X.P.: PCF8574 – Remote 8-bit I/O expander for I²C-bus (2014). www.nxp.com, Zugegriffen: 17. Dezember 2014
2. Semiconductors, N.X.P.: PCA9534 – 8-bit I²C-bus and SMBus low power I/O port with interrupt (2015). www.nxp.com, Zugegriffen: 30. Juli 2015
3. STMicroelectronics: M24C24R – 64 kBit I²C-Bus serieller EEPROM (2015). www.st.com, Zugegriffen: 20. Januar 2015
4. STMicroelectronics: M95256-xx SPI – serielle EEPROMs (2018). www.st.com, Zugegriffen: 13. April 2018
5. Microchip Technology: 25LC256 – 256 kBit SPI-Bus serieller EEPROM (2015). www.microchip.com, Zugegriffen: 30. Juli 2015
6. Microchip Technology: ATmega88 – 8 Bit Microcontroller (2018). www.microchip.com, Zugegriffen: 14. April 2018
7. Microchip Technology: AN2665 – Interfacing AVR serial memories (2018). www.microchip.com, Zugegriffen: 13. April 2018

8. Adesto Technologies: AT45DB161 – 16 Mbit SPI serial flash memory (2015). www.adestotech.com, Zugegriffen: 14. Juni 2015
9. Microchip Technology: SST25WF080B – 8 Mbit 1,8 V SPI serial flash (2016). www.microchip.com, Zugegriffen: 23. Februar 2016
10. Microchip Technology: MCP4801/4811/4821 1 Kanal SPI-D/A-Wandler (2018). www.microchip.com, Zugegriffen: 13. April 2018
11. Microchip Technology: MCP4802/4812/4822 2 Kanal SPI-D/A-Wandler (2018). www.microchip.com, Zugegriffen: 13. April 2018
12. Maxim Integrated: MAX31629 – I2C digital thermometer and real-time-clock (2014). www.maximintegrated.com, Zugegriffen: 31. Oktober 2014
13. Silicon Labs: SI4840/44 – broadcast analog tuning digital display AM/SW/FM radio receiver (2018). www.silabs.com, Zugegriffen: 13. April 2018
14. Silicon Labs: AN610 – SI48XX ATDD programming guide V3.0/2013 (2014). www.silabs.com, Zugegriffen: 27. April 2014
15. Silicon Labs: AN602 – SI4822/26/27/40/44 Antenna, Schematic, Layout and Guidelines (2014). www.silabs.com, Zugegriffen: 27. April 2014
16. Werner, M.: Nachrichten-Übertragungstechnik. Vieweg, Wiesbaden (2006)

Weiterführende Literatur

17. Kesel, F., Bartholomä, R.: Entwurf von digitalen Schaltungen und Systemen mit HDLs und FPGAs. Oldenbourg, München (2013)
18. Woitowitz, R., Urbanski, K., Gehrke, W.: Digitaltechnik. Springer, Berlin Heidelberg (2012)
19. Ohm, J.-R., Lüke, H.D.: Signalübertragung. Springer, Berlin Heidelberg New York (2007)
20. Microchip Technology: AT25256 – 256 kBit serieller SPI-BUS EEPROM (2015). www.microchip.com, Zugegriffen: 14. April 2018
21. Microchip Technology: MCP413X/415X/423X/425X – digitale Potentiometer mit flüchtigen Speicher (2015). www.microchip.com
22. NXP Semiconductors: PCF8591 – 8-Bit A/D and D/A converter (2015). www.nxp.com

Anzeigen

8

Zusammenfassung

Die Mensch-Maschine-Schnittstelle ist in eingebetteten Systemen nicht immer sichtbar oder notwendig. Dennoch gibt es immer wieder Fälle, in denen der Systemzustand visualisiert werden muss. Das Kapitel gibt deshalb eine Einführung in die Anzeigetechnik und stellt eine Familie von graphischen und textuellen Displays vor, die sich ohne Aufwand in die Umgebung eines AVR-Controllers integrieren lassen.

Anzeigen stellen eine Schnittstelle zwischen einem technischen System und Benutzern dieses Systems dar. Sie liefern Informationen über den Zustand des Systems, stellen Messwerte dar und geben Rückmeldung auf benutzergesteuerte Aktionen wieder. Die Anzeigen unterscheiden sich durch Informationsmenge, Informationsdichte, Energieverbrauch, Steuerkomplexität, Ablesbarkeit und Preis.

8.1 Einführung

8.1.1 Displaylayout

Alphanumerische Anzeigen sind entwickelt worden, um m-Zeichen mit Hilfe einer festen Anordnung von n-Bildelementen darzustellen. Die Siebensegment- und Vierzehnsegment-Anzeigen zählen zu den bekanntesten Anordnungen und dienen zur Darstellung der Ziffern und bedingt auch der Buchstaben. Die Bildsegmente werden direkt von einem Displaycontroller angesteuert. Um die Anzahl der Steuerpins zu reduzieren, werden die gleichen Segmente aller Zeichen parallelgeschaltet und die Segmente im Multiplexbetrieb angesteuert.

Bei den Punktmatrix-Anzeigen sind die Bildelemente als Bildpunkte (Pixel) matrixförmig mit m-Zeilen und n-Spalten angeordnet. Durch die Platzierung dreier Bildelemente

mit den Farben rot, grün und blau an jedem Matrixknoten dicht beieinander, entsteht eine voll farbige Anzeige. Mit der Ansteuerung im Multiplexbetrieb kann die Helligkeit jedes der $3 \times m \times n$ Bildelemente geändert werden. Durch die Überlagerung der drei Farben nimmt das menschliche Auge die Pixelfarbe wahr. Mit einer Punktmatrix-Anzeige können Informationen in alphanumerischer Form so wie Bilder dargestellt werden.

8.1.2 Emissive und nicht emissive Anzeigen

Man unterscheidet zwischen emissiven und nicht emissiven Anzeigen. Eine nicht emissive Anzeige benötigt eine Hinterleuchtung, die entweder mit einer künstlichen Lichtquelle oder durch die Reflexion des Tageslichtes realisiert wird. Die LCD¹-Anzeigen gehören zu den nicht emissiven Anzeigen und können als passive oder aktive Matrix gebaut werden.

8.1.2.1 Flüssigkristallanzeigen (LCD)

Kristalle besitzen die Eigenschaft der Anisotropie, das heißt, bestimmte Eigenschaften des Materials sind richtungsabhängig. Dazu gehören unter anderem die Lichtgeschwindigkeit und damit auch der Brechungsindex. Diese hängen mit der Polarisationsrichtung des Lichtes zusammen. Die Polarisationsrichtung ist die Richtung der Schwingungsebene einer transversalen Welle, im Fall von Licht die Richtung der elektrischen Feldstärke. Unpolarisiertes Licht kann man in eine vertikal und eine horizontal polarisierte Komponente verteilen. Tritt es durch den Kristall hindurch, werden die zwei Komponenten unterschiedlich stark gebrochen, aus dem Kristall treten zwei Strahlen polarisierten Lichts aus. Die Überlagerung ergibt eine gegenüber der Einfallspolarisation geänderte Polarisierung. Diesen Effekt nennt man *Doppelbrechung*. Optisch anisotrope Medien können also die Polarisationsrichtung von Licht beeinflussen. Normalerweise gehen Kristalle bei Erhitzen von der festen, einer Fernordnung unterliegenden, anisotropen Phase in die flüssige Phase über, die durch das Fehlen einer Fernordnung und Isotropie charakterisiert ist. Spezielle organische Kristalle haben jedoch die merkwürdige Eigenschaft, dass sie bereits flüssig sind, jedoch einer (elastischen) Fernordnung unterliegen und in dieser Phase anisotrop bleiben. Man spricht von einer Orientierungsfernordnung, während die Positionsfernordnung verloren geht (Flüssigkeit). Diese Kristalle wurden 1888 vom Biologen Friedrich Reinitzer entdeckt und heißen Flüssigkristalle (LC – liquid crystal).

Seit Ende der 60er-Jahre macht man sich diesen Effekt in der Anzeigetechnik zunutze. Im Prinzip befüllt man den Hohlraum (Zelle) zwischen zwei Glasplatten mit Flüssigkristallen, die diese Eigenschaft bei Raumtemperatur haben. Die meisten heute gebräuchlichen Anzeigen nutzen dabei den *Schadt-Helfrich-Effekt*, der in Abb. 8.1a schematisch dargestellt ist. Durch entsprechende Oberflächenbehandlung (Reiben), wird dafür gesorgt, dass die Vorzugsrichtungen (Direktor) der Moleküle direkt auf den beiden Glasplatten um 90° gegeneinander verdreht sind. Aufgrund der Wechselwirkungen zwischen den Mole-

¹ LCD – Liquid Cristal Display (deutsch Flüssigkristallanzeige).

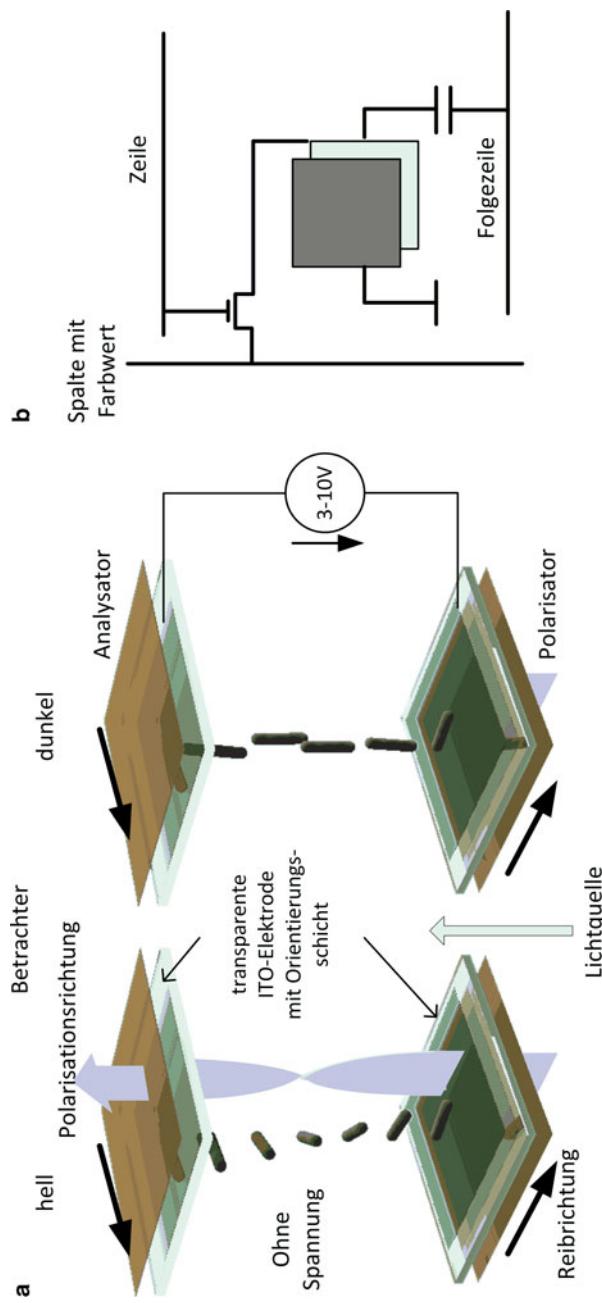


Abb. 8.1 Prinzip einer TN-Zelle (**a** hell, **b** dunkel), daneben das Schaltprinzip einer aktiven TFT-Zelle

külen drehen sich die dazwischenliegenden Schichten schraubenförmig auf und drehen dabei die Polarisationsrichtung von durchtretendem Licht um 90°. Man spricht von einer *twisted nematic (TN) Zelle*.

Beschichtet man die Glasplatten mit einer durchsichtigen aber leitfähigen ITO (Indium-Zinn-Oxid) Schicht und legt an diese eine Spannung an, so bewirkt das elektrische Feld in der Zelle, dass sich die Flüssigkristalle aufrichten und das Licht unbeeinflusst passieren lassen. Durch zwei Polarisationsfolien, deren Polarisationsrichtungen senkrecht aufeinander stehen, kann offensichtlich das Licht im spannungslosen Zustand ungehindert durchtreten, da es selbst in seiner Polarisationsrichtung gedreht wird. Bei eingeschalteter Spannung wird die Polarisationsrichtung nicht gedreht und das Licht wird vom jenseits der Zelle liegenden Polfilter nicht durchgelassen. Die Zelle ist „aktiv dunkel“. Bei parallelen Filtern ist die Zelle „aktiv hell“.

Bei der Herstellung eines (passiven) LC-Displays werden die Elektroden aus ITO auf die Gläser aufgesputtert und in einem photochemischen Verfahren strukturiert (geätzt). Dabei muss man darauf achten, dass sich gegenüberliegende Elektroden nur an sichtbaren Stellen kreuzen. Anschließend wird eine Polymerschicht aufgetragen, die durch Reiben konditioniert wird. Zwischen die Scheiben werden kleine Kugelchen (Spacer) eingefüllt, die einen konstanten Abstand gewährleisten. Die Scheiben werden verklebt, durch eine Öffnung wird der Flüssigkristall eingefüllt. Viele LC-Displays besitzen bereits fest strukturierte Symbole oder 7- bzw. 13-Segment-Ziffern.

Diese passiven Zellen ermöglichen nur eine niedrige Bildwiederholungsrate bei einem eingeschränkten Blickwinkel. LCD-Anzeigen mit IPS² - und VA³ -Zellen verbessern den Kontrast bei erhöhtem Blickwinkel [1]. Die Weiterentwicklung der TN-Zelle um verschiedene Hintergrundfarben und eine höhere Multiplexrate zu ermöglichen, hat zu den Enhanced-, Modulated-, Super-, Double-Super- und Enhanced-Super-Twisted-Nematic Zellen geführt [2]. Hohe Schaltzeiten und niedrige Speicherzeiten einer passiven TN-Anzeigematrix reduzieren den Kontrast bei wachsender Anzeigegröße, weil die Ansteuerungszeit pro Bildelement und Einzelbild bei konstanter Multiplexrate kleiner wird.

Aktivmatrix LCD-Anzeigen beheben diesen Nachteil, indem sie für jedes Bildelement einen Kondensator für die Erhaltung des elektrischen Feldes nutzen. Dieser Kondensator kann über einen Dünnschicht-Transistor (englisch TFT – Thin Film Transistor) nachgeladen werden. Diese TFT-Technologie ermöglicht den Bau vollfarbiger Displays. Man spricht in dem Zusammenhang auch von Aktivmatrix LC-Anzeigen (AMLCD). In Abb. 8.1b ist das Prinzipschaltbild dargestellt, in Abb. 8.2 ist eine Mikroskop Aufnahme gezeigt, die der Autor mit einem klassischen Lichtmikroskop im Drauflicht mit einem TFT-Monitor erstellt hat.

Alle LCD Anzeigen benötigen eine Hinterleuchtung. Diese kann aus einer einfachen Kaltkathodenröhre oder einer weißen LED-Zeile bestehen, deren Licht über einen Lichtleiter gleichmäßig verteilt wird. Manche Displays nutzen auch Elektronlumineszenzfolien

² IPS – In plane switching.

³ VA – vertical alignment.

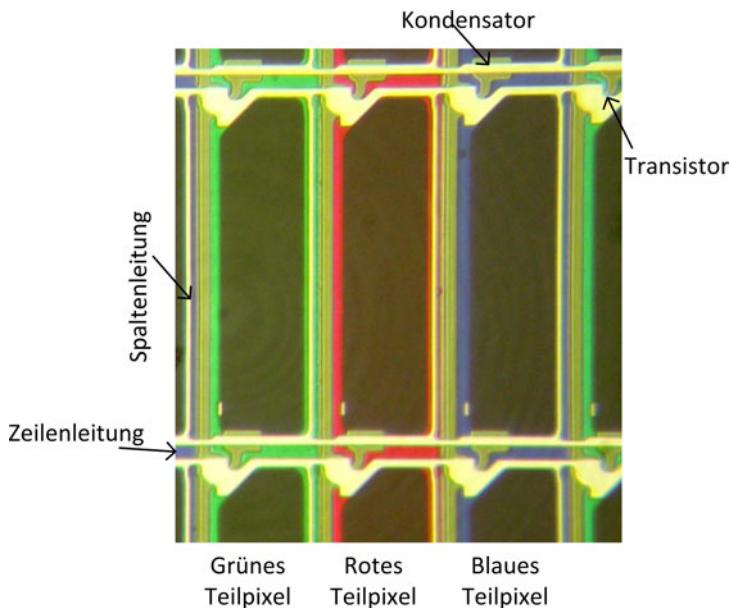


Abb. 8.2 Mikroskopische Aufnahme eines Pixels aus einem TFT-Monitor

als Hinterleuchtung, die allerdings mit höherer Spannung angesteuert werden müssen. Größere Anzeigen besitzen LED-Matrizen, die hinter der gesamten Displayfläche angeordnet sind und diese bereichsweise hinterleuchten, wobei dunklere Bildpartien schwächer beleuchtet werden und heller stärker. Dieses Verfahren erhöht den Kontrast der Anzeige und spart deutlich Energie (Full-LED-Backlight). Die Hinterleuchtung ist der größte Energieverbraucher einer LCD-Anzeige und benötigt umso mehr Energie, je heller das Umgebungslicht ist. Für Displays, die im hellen Sonnenlicht abgelesen werden sollen, werden oftmals Spiegelfolien statt einer Hinterleuchtung eingesetzt, das Sonnenlicht passiert dann die Zelle zweimal (so genannte reflektive Anzeigen). Nutzt man halbdurchlässige Spiegel und eine aktive Lichtquelle zusätzlich, spricht man von transflektiven Anzeigen, die im Dunkeln und in der Sonne arbeiten können.

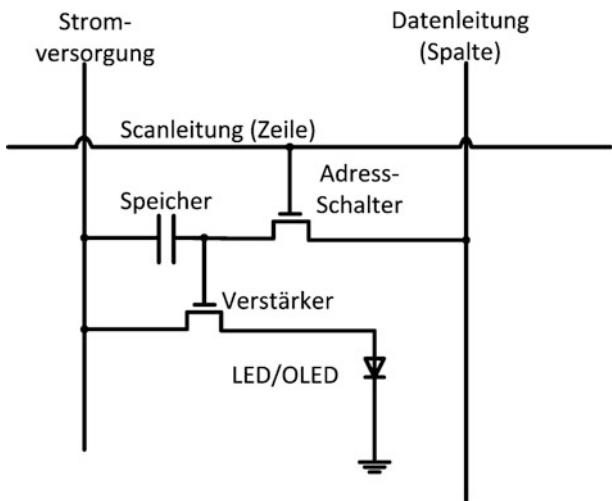
8.1.2.2 LED-Anzeigen

Als emissive Anzeigen werden hier nur die mit LEDs⁴ gebauten, Anzeigen erwähnt. Heutzutage können großflächige Displays aus anorganischen Leuchtdioden in Halbleiter-Technologie hergestellt werden.

Organische LEDs (OLED) basieren auf der Eigenschaft einiger organischen Materialien, Licht zu erzeugen, wenn sie von elektrischem Strom durchflossen werden. Die mit OLEDs hergestellten vollfarbigen Anzeigen können als passive oder aktive Matrix gebaut

⁴ LED – Light Emitting Diode (deutsch Leuchtdiode).

Abb. 8.3 Ansteuerung eines Pixels in einer aktiv leuchtenden (LED oder OLED) Anzeige



werden. Sie können extrem dünn und biegsam sein, bieten einen weiten Blickwinkel, haben aber derzeit noch einen höheren Energieverbrauch als LCD-Anzeigen. Aktive OLED-Matrixanzeigen haben einen etwas komplexeren Aufbau als aktive LCD-Anzeigen, da neben der Bildinformation auch die Energie für die Erzeugung des Bildpunktes an jeden einzelnen Pixel herangeführt werden muss. In Abb. 8.3 ist eine solche Schaltung für einen einzelnen Bildpunkt schematisch dargestellt. Dafür ist jedoch keine Hinterleuchtung notwendig.

8.1.3 Bildaufbau

Der ansteuernde Controller einer Punktmatrix-Anzeige empfängt von einem Mikrocontroller die zur Bilddarstellung nötige Informationen und frischt das Bild regelmäßig auf. Der Zustand, bzw. die Farbe jedes Pixels ist in einem flüchtigen Speicher abgebildet. Durch den direkten Zugriff auf den Speicher können Änderungen am Bild vorgenommen werden. Der Mikrocontroller überträgt die Koordinaten (Zeile und Spalte) und den Zustand, bzw. die Farbe des zu ändernden Pixels.

Um den Datenfluss zwischen dem Mikrocontroller und dem Display zu reduzieren und die Bilder leicht zu skalieren, wird oftmals Vektorgraphik verwendet. In der Vektorgraphik werden die Zeichensätze und Graphiken durch Linienzüge, Kreisbögen, Polynomkurven und Flächenelemente vektoriell definiert. Der Mikrocontroller überträgt nur noch die Vektorelemente, die von dem Display-Controller verwendet werden, um das Bild aufzubauen.

8.1.4 Display Ansteuerung

Einige Displays mit einem hohen Informationsgehalt stellen für die Übertragung der Befehlscodes und der Daten einen Parallelbus zur Verfügung. Dieser Bus besteht aus 4, 8, 16 oder mehr Leitungen. Auch ältere Display-Controller besitzen einen Parallelbus wegen der einfachen Ansteuerung.

Moderne Displays mit niedrigem bis mittlerem Informationsgehalt für embedded Systeme benutzen für den Informationsfluss eine schnelle, serielle Schnittstelle wie zum Beispiel SPI. Eine serielle Schnittstelle kann auch Displays mit integrierter Vektorgrafik benutzen. Hochauflösende Displays werden nur noch zum Teil analog (RGB, VGA) angesteuert, die meisten über LVDS⁵ basierte Bussysteme wie DisplayPort oder TMDS⁶ basierte Anschlüsse wie DVI oder HDMI. Hier werden die Farbwerte und der Pixeltakt teils getrennt und teils kombiniert seriell übertragen, meist kombiniert mit Audiosignalen und mit Signalen zum Digitalen Rechtemanagement. In diesem Kapitel sollen jedoch die Displayansteuerungen vorgestellt werden, die mit geringstmöglichen Prozessorressourcen auskommen und daher direkt von einem 8-Bit Prozessor angesteuert werden können.

8.2 Punktmatrix LCD-Display mit paralleler Ansteuerung

In Geräten mit Mikrocontrollern der ATmega Klasse werden Punktmatrix LCD-Displays wegen der einfachen parallelen Ansteuerung oft eingesetzt. Diese Displays benutzen meist einen Controller, der mit dem Hitachi HD44780 kompatibel ist. Es handelt sich um STN⁷ - Displays, also monochromatische LCD-Displays mit grünem oder blauem Hintergrund und reduziertem Energieverbrauch. Sie können auch eine Hintergrundbeleuchtung eingebaut haben, um die Lesbarkeit bei Sonnenlicht zu verbessern. Sie werden als 1-, 2- oder 4-zeilige Displays hergestellt.

8.2.1 Struktur eines Displays mit einem KS0070B Controller

Ein grobes Blockschaltbild eines Displays, das mit einem Controller KS0070B, der mit dem HD44780 kompatibel ist und von der Fa. Samsung [3] hergestellt wird, ist in Abb. 8.4 dargestellt. Der Datenaustausch zwischen dem steuernden Mikrocontroller und dem Display-Controller findet über einem 8-Bit-Datenbus (DB7...DB0) oder alternativ über einem 4-Bit-Datenbus, bestehend aus den Pins DB7...DB4, statt. Im Folgenden wird näher die alternative Ansteuerung betrachtet, weil dadurch I/O Pins des Mikrocontrollers gespart werden. Die doppelte Zeit für die Übertragung eines Bytes wird in diesem Fall in

⁵ Low Voltage Differential Signaling.

⁶ Transition Minimized Differential Signaling.

⁷ STN – Super-Twisted Nematic.

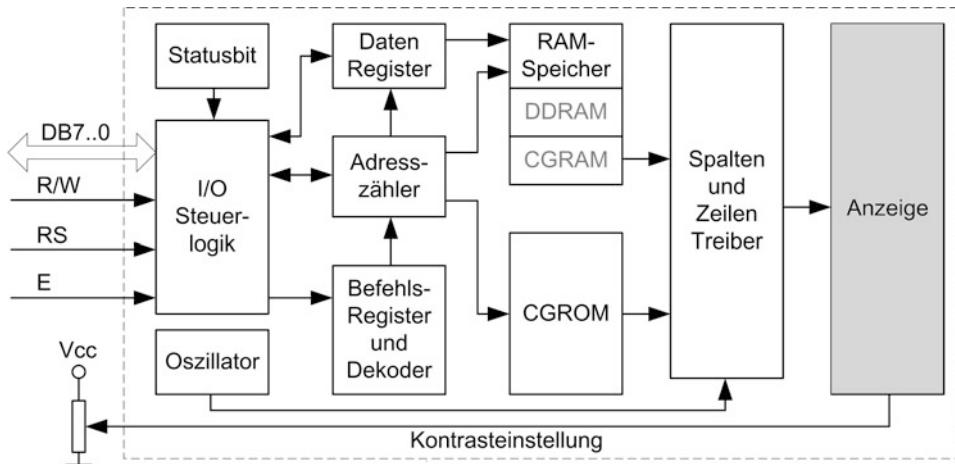


Abb. 8.4 LCD-Display mit Controller KS0070B – Blockschaltbild

Kauf genommen. Der Controller besitzt neben den Datenpins auch 3 Steuerpins um den Datenfluss zu steuern:

- **RS** (Register Select) ist ein Eingang, der das ankommende Byte bei High in einen der RAM-Speicher und bei Low in den Funktionsdekoder steuert.
- **E** (Enable) über eine Low-High-Low an diesem Eingang werden die Bits vom Datenbus in einen Datenpuffer gespeichert. Der Datenpuffet wird mit dem Eingang RS selektiert.
- **R/W** (Read/Write) bestimmt die Richtung des Datentransfers. Aus Sicht des Mikrocontrollers bedeutet High Datenauslesen, Low Datenschreiben.

Der Zeichensatz des Displays, der aus 192 5×7 Punkte und 32 5×10 Punkte große Zeichen besteht, ist im CGROM⁸ hinterlegt. Dieser Zeichensatz kann dem Datenblatt entnommen werden. Der Anwender kann selbst 5×8 Punkte große Zeichen definieren und bis zu acht davon gleichzeitig verwenden.

Der RAM-Speicher ist in zwei Bereiche unterteilt:

- DDRAM⁹ ist ein 80 Byte großer Speicher, der die Display-Positionen abbildet. Die entsprechenden DDRAM-Adressen im hexadezimalen Zahlenformat eines 4zeiligen Displays wie zum Beispiel DEM 16481 [4] von der Fa. Display Elektronik sind in Tab. 8.1 zu sehen. Im DDRAM werden die ASCII-Werte derjenigen Zeichen gespeichert, die dargestellt werden sollen. Der folgende Programmausschnitt:

```
Display_Set_DDRAMAddress(0x04);
Display_Write('A'); //Übertragung des Zeichens A
```

⁸ CGROM – Character Generator ROM.

⁹ DDRAM – Display Data RAM.

Tab. 8.1 Display-Positionen im DDRAM

Display-Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
1. Zeile	DDRAM Adresse	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
2. Zeile		40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F
3. Zeile		10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
4. Zeile		50	51	52	53	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F

führt zum Setzen des Cursors auf die 5. Position der ersten Zeile und zur Anzeige des Zeichens A an dieser Stelle. Beim Schreiben oder Lesen eines Datenbytes wird der Adresszähler entsprechend den Cursoreinstellungen inkrementiert oder dekrementiert.

- CGRAM¹⁰ ist ein 64 Byte großer RAM-Speicher, in dem die Bitmuster von bis zu 8 neu definierten Zeichen in codierter Form gespeichert werden können. Im Abschn. 8.2.4 wird die Generierung eines neuen Zeichens beschrieben.

Der Kontrast eines solchen Displays wird über eine analoge Spannung eingestellt.

8.2.2 Befehlssatz

Der Befehlssatz eines KS0070B Displays besteht aus der Kombination eines Datenbytes und der Steuerbits **RS** und **R/W**. Eine Zusammenfassung dieser Befehle mit den entsprechenden Befehlscodes und den Ausführungszeiten befindet sich in Tab. 8.2.

- **Display löschen** füllt den gesamten DDRAM Speicher mit Leerzeichen (0x20), setzt die DDRAM-Adresse auf 0x00 (erste Position von links der ersten Zeile des Displays) und setzt den Eingangsmodus auf Inkrementieren; mit jedem Datenbyte wird der Cursor nach rechts geschoben und der Adresszähler inkrementiert.
- **Display-Grundeinstellung** setzt die DDRAM-Adresse auf 0x00, der DDRAM Inhalt bleibt unverändert und eine eventuelle Displayschiebung wird rückgängig gemacht;
- **Display-Eingangsmodus** legt abhängig von den Bits **I/D** (Inkrement/Dekrement) und **SH** (Shift) die Bewegungsrichtung des Cursors und die Schieberichtung des Displays bei der Übertragung eines Datenbytes fest; beim Lesen des RAM-Speichers bzw. beim Schreiben in das CGRAM findet keine Displayschiebung statt. Ansonsten ist die Auswirkung der zwei Bits der Tab. 8.3 zu entnehmen.
- **Display/Cursor an/aus**, wenn **D = 1** wird das Display eingeschaltet, bei **C = 1** wird der Cursor eingebendet und **B = 1** führt zum Blinken des Cursors;

¹⁰ CGRAM – Character Generator RAM.

Tab. 8.2 Befehle des KS0070B (HD44780) Controllers

Befehl	Befehlscode											Ausführungszeit
	RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0		
Display löschen	0	0	0	0	0	0	0	0	0	1		1,53 ms
Display-Grundeinstellung	0	0	0	0	0	0	0	0	1	x		1,53 ms
Display-Eingangsmodus	0	0	0	0	0	0	0	1	I/D	SH		39 µs
Display/Cursor an/aus	0	0	0	0	0	0	1	D	C	B		39 µs
Cursor/Display schieben	0	0	0	0	0	1	S/C	R/L	x	x		39 µs
Funktion setzen	0	0	0	0	1	DL	N	F	x	x		39 µs
CGRAM-Adresse setzen	0	0	0	1	A5	A4	A3	A2	A1	A0		39 µs
DDRAM-Adresse setzen	0	0	1	A6	A5	A4	A3	A2	A1	A0		39 µs
Statusbit/Adresse lesen	0	1	BF	A6	A5	A4	A3	A2	A1	A0	0	
Byte in RAM schreiben	1	0	D7	D6	D5	D4	D3	D2	D1	D0		43 µs
Byte aus RAM lesen	1	1	D7	D6	D5	D4	D3	D2	D1	D0		43 µs

- **Cursor/Display schieben** – der Befehl wirkt sich auf das Schieben des Cursors, bzw. des Displays abhängig von den Bits **S/C** (Displayshift/Cursor) und **R/L** (Right/Left) (Tab. 8.4) in Abwesenheit eines Datenbytetransfers.
- **Funktion setzen** – Der Befehl legt fest, ob die Kommunikation mit dem Display über einen 8 Bit (**DL** = 1) oder über einen 4 Bit (**DL** = 0) Datenbus stattfindet, ob das Display als einzeiliges (**N** = 0) oder zweizeiliges (**N** = 1) Display initialisiert wird und ob 5 × 7 Punkte (**F** = 0) oder 5 × 10 Punkte (**F** = 1) große Zeichen verwendet werden.
- **CGRAM-Adresse setzen** – Mit diesem Befehl wird der Cursor auf die gewünschte Adresse im CGRAM gesetzt;
- **DDRAM-Adresse setzen** – Mit dem Befehl wird die Display Position festgelegt, an der das nächste Zeichen angezeigt werden soll.
- **Statusbit/Adresse lesen** – Beim Ausführen dieses Befehls liest der Mikrocontroller den Inhalt des Adresszählers und das Statusbit, das bei **BF** = 1 zeigt, dass der Display Controller beschäftigt ist.
- **Byte im RAM schreiben** – Mit dem Befehl wird ein Byte an die aktuelle RAM-Adresse gespeichert (DDRAM oder CGRAM).
- **Byte aus RAM lesen** – Mit diesem Befehl liest der Mikrocontroller ein Byte von der aktuellen RAM-Adresse (DDRAM oder CGRAM).

Tab. 8.3 Auswirkung der I/D und SH Bits

I/D	SH	Beschreibung
0	0	Cursor wird nach links geschoben, der Adresszähler dekrementiert, keine Displayschiebung
0	1	Cursor wird nach links geschoben, der Adresszähler dekrementiert, Displayschiebung nach rechts
1	0	Cursor wird nach rechts geschoben, der Adresszähler inkrementiert, keine Displayschiebung
1	1	Cursor wird nach rechts geschoben, der Adresszähler inkrementiert, Displayschiebung nach links

Tab. 8.4 Auswirkung der S/C und R/L Bits

S/C	R/L	Beschreibung
0	0	Cursor wird nach links geschoben, der Adresszähler dekrementiert; keine Displayschiebung
0	1	Cursor wird nach rechts geschoben, der Adresszähler inkrementiert; keine Displayschiebung
1	0	Gesamtes Display mit Cursor werden nach links geschoben
1	1	Gesamtes Display mit Cursor werden nach rechts geschoben

Wenn der Display-Controller ein Byte empfangen hat, beginnt ein intern gesteuerter Ablauf, dessen Dauer vom Befehl abhängig ist. Während dieser Zeit wird das Statusbit **BF** gesetzt, um dem Mikrocontroller zu signalisieren, dass der Controller keine weiteren Bytes aufnehmen kann. Über ein blockierendes Warten kann der Mikrocontroller den Zustand des Bits **BF** abfragen, bevor er ein weiteres Byte sendet. Über einen passend eingestellten Timerinterrupt, der mit dem Speichern eines neuen Bytes gestartet wird, kann der richtige Zeitpunkt für eine neue Übertragung bestimmt werden. Die Ausführungszeit der Befehle ist von der Betriebsspannung, dem benutzten Controller und der internen Taktfrequenz des Controllers abhängig und kann dem entsprechenden Datenblatt entnommen werden.

8.2.3 Bit Kommunikation

Um I/O-Pins des Mikrocontrollers zu sparen, kann die Kommunikation mit dem Display-Controller auf 4 Bits eingestellt werden. In diesem Fall werden nur die Eingänge DB7...4 verwendet und ein Byte muss in zwei Schritten übertragen und gespeichert werden: zuerst die höherwertige Bytehälfte D7...4 und danach die niederwertige Bytehälfte D3...0, so wie in Abb. 8.5 dargestellt. Nach dem Hochfahren des Controllers wird der gesamte Inhalt des DDRAM-Speichers mit Leerzeichen gefüllt, die Kommunikation auf 8 Bit gestellt, das Display wird abgeschaltet und der Cursor ausgeblendet.

Abb. 8.5 Display Kommunikation über einen 4 Bit Datenbus

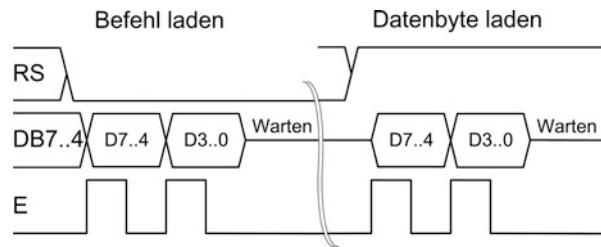
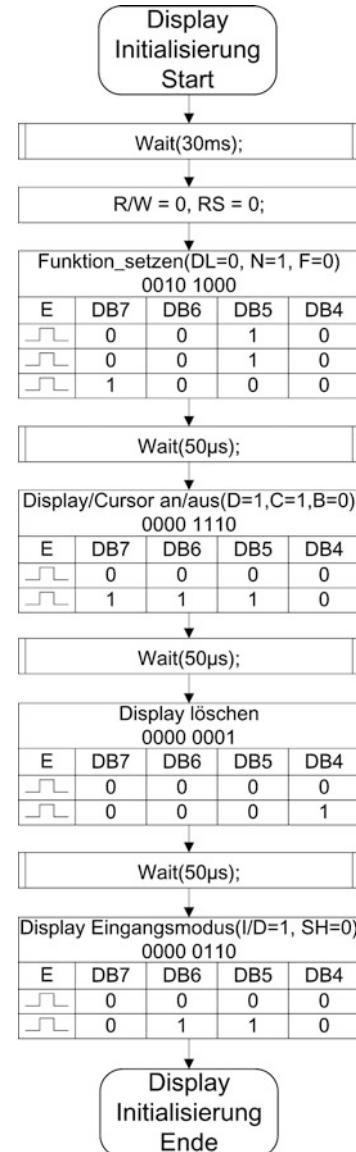


Abb. 8.6 Display Initialisierung im 4 Bit Modus



Das Display kann softwaremäßig mit den Befehlen: Funktion setzen, Display/Cursor an/aus, Display löschen und Display Eingangsmodus initialisiert werden. Die Wahl-Bits dieser Befehle werden entsprechend der konkreten Aufgabe ausgewählt. Das Flussdiagramm für die Display-Initialisierung im 4-Bit-Modus ($DL = 0$) ist in Abb. 8.6 dargestellt und hat folgende Einstellungen:

- zweizeiliges Display ($N = 1$),
- 5×7 Punkte große Zeichen ($F = 0$),
- Display an ($D = 1$),
- Cursor an ($C = 1$),
- Blinken ausgeschaltet ($B = 0$),
- mit einem neuen Datenbyte wird der Adresszähler inkrementiert ($I/D = 1$) und
- das Display soll nicht geschoben werden ($SH = 0$)

Jede übertragene Bytehälfte wird mit der fallenden Flanke des **E**-Signals in das Eingangsregister geladen.

8.2.4 Generierung eines neuen Zeichens

Dem Anwender stehen die Zeichenadressen 0...7 zur Verfügung, um neue 5×8 Punkte große Zeichen zu definieren. Ein neu definiertes Zeichen belegt einen acht Byte großen Speicherbereich und muss vor dem Benutzen in den CGRAM geladen werden. Es wird in der Folge als Beispiel die Generierung des Zeichens „ö“ und das Speichern des entstandenen Bitmusters an der Zeichenadresse 0 erläutert. Das Bitmuster des neuen Zeichens ist in der Tab. 8.5 zu sehen.

Das Bitmuster eines zweiten Zeichens müsste man im CGRAM ab der Adresse 0x08 bis 0x0F speichern.

Um den Programmcode zu vereinfachen, werden die für ein neues Zeichen berechneten CGRAM-Werte in ein Array gespeichert. Der Adresszähler des CGRAM-Speichers wird

Tab. 8.5 Bitmuster des Zeichens „ö“

Zeichenadresse	CGRAM Adresse	2^4	2^3	2^2	2^1	2^0	CGRAM Wert
0x00	0x00	0	1	0	1	0	$2^1 + 2^3 = 0xA$
	0x01	0	0	0	0	0	0x00
	0x02	0	1	1	1	0	$2^3 + 2^2 + 2^1 = 0xE$
	0x03	1	0	0	0	1	$2^4 + 2^0 = 0x11$
	0x04	1	0	0	0	1	0x11
	0x05	1	0	0	0	1	0x11
	0x06	0	1	1	1	0	0x0E
	0x07	0	0	0	0	0	0x00

auf die erste Adresse gesetzt und anschließend werden die acht Datenbytes übertragen um sie im CGRAM zu speichern, wie im folgenden Codeausschnitt:

```
uint8_t ucOeZeichen[8] = {0x0A, 0x00, 0x0E, 0x11, 0x11, 0x11, 0x0E, 0x00};
...
Display_Set_CGRAMAddress(0x00); //0x00 = Adresse des 1. Datenbytes des
                                //Bitmusters im CGRAM
for(uint8_t ucI = 0; ucI < 8; ucI++)
{
    Display_Write(ucOeZeichen[ucI]);
}
```

Nachdem die DDRAM-Adresse (Cursor-Position) im Hauptprogramm mit dem Aufruf:

```
Display_Write(0x00);      //0x00 = die Zeichenadresse des neu
                        //definierten Zeichens
```

gesetzt wurde, wird an der aktuellen Cursor-Position das Zeichen „ö“ dargestellt.

8.2.5 Ausführen der Display Befehle ohne blockierendes Warten

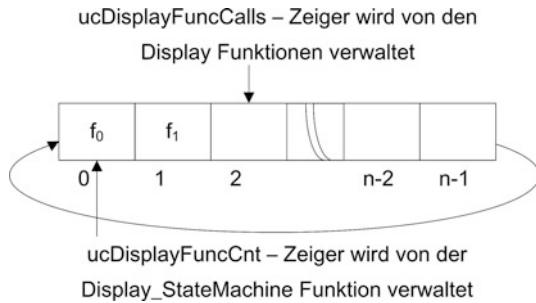
Das weitere Beispiel bezieht sich auf ein Display, dessen R/W-Eingang fest mit GND (=0 V) verbunden ist, um einen weiteren I/O Pin des Mikrocontrollers zu sparen. In diesem Fall können keine Lesebefehle ausgeführt werden und die einzige Möglichkeit, das Ende eines Befehls festzustellen, bleibt die Messung der Ausführungszeit. Um diese Zeit genau zu bestimmen, wird ein Timerinterrupt verwendet, der auf 50 µs initialisiert wird. Das Ausführen eines Displaybefehls bedeutet:

- das Ausgeben eines Bytes (Befehlscode oder Datenbyte) auf den 4-Bit Datenbus. Das erfolgt in zwei Schritten gemäß der Abb. 8.5,
- das entsprechende Steuern der RS- und E-Eingänge des Displays und
- das Ausführen der weiteren Tasks des Hauptprogramms während das Display mit dem internen Ablauf beschäftigt ist.

Um ein blockierendes Warten des Mikrocontrollers nach dem Aufruf einer Display-Funktion zu vermeiden und gleichzeitig sicherzustellen, dass kein Befehl wirkungslos ist, kann die Ansteuerung des Displays als Zustandsautomat realisiert werden. Es wird ein n-Stellen großer Ringpuffer definiert wie in Abb. 8.7, auf den zwei Zeiger gerichtet sind, beide mit 0 initialisiert. Mit dem Aufruf einer Display-Funktion:

Abb. 8.7 Displayfunktionen

Puffer



- wird das zu übertragende Byte (Befehlscode oder Datenbyte) in das Array `ucDisplayCommandByte` gespeichert zusammen mit einem zweiten Byte in das Array `ucDisplayDataByte` das den Zustand des RS-Eingangs (0/1) und die entsprechende Wartezeit (als Vielfache von 50 µs) codiert; der Zeiger `ucDisplayFuncCalls` zeigt auf die Speicherstelle des Ringpuffers;
- wird der Zeiger `ucDisplayFuncCalls` anschließend inkrementiert und falls er den Wert `n` erreicht, wird er auf 0 gesetzt;
- wird der Zähler `ucDisplayFuncOpen` inkrementiert. Dieser zählt, wie viele nicht ausgeführte Funktionen in der Warteschleife stehen.

Der zweite Zeiger `ucDisplayFuncCnt` wird von einer Funktion `Display_StateMachine` verwaltet, deren Aufbau und Aufruf in der Abb. 8.8 verdeutlicht ist. Dieser Zeiger zeigt auf die nächste Funktion, die ausgeführt wird. Mit dem Aufruf einer Display-Funktion:

- wird der Zeiger `ucDisplayFuncCnt` inkrementiert (wenn er den Wert `n` erreicht, wird er auf 0 gesetzt),
- wird der Zähler `ucDisplayFuncOpen` dekrementiert. Wenn dieser Zähler der am Anfang mit 0 initialisiert wurde, den Wert 0 erreicht, dann sind keine Funktionen mehr in der Warteschleife.
- wird der Timerinterrupt mit der neuen Wartezeit gestartet. Beim Erreichen der eingestellten Wartezeit wird der Timer gestoppt und der eventuelle Aufruf einer weiteren Funktion freigegeben.

Die Größe `n` des Ringpuffers soll abhängig von der konkreten Anwendung so klein wie möglich gewählt werden, um wenig Speicherplatz zu verbrauchen, muss aber groß genug sein, damit die Textanzeige einwandfrei funktioniert.

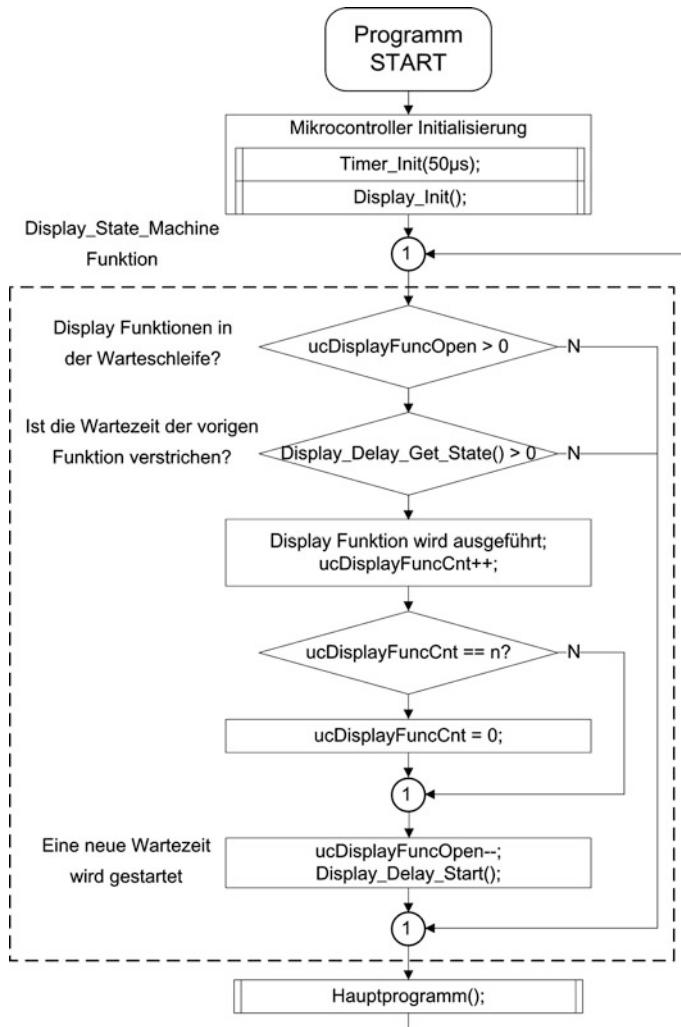


Abb. 8.8 Zustandsautomat Displayfunktionen

8.3 Serielle Ansteuerung eines parallelen LCD-Displays

Ein LCD-Display (siehe Abschn. 8.2), angeschlossen an einem im Abschn. 7.1 beschriebenen Port Expander Baustein PCF8574, kann über I²C angesteuert werden, so wie in Abb. 8.9 dargestellt. Der Mikrocontroller benötigt für die Ansteuerung lediglich den I²C-Bus, an dem weitere Geräte oder Displays angeschlossen werden können.

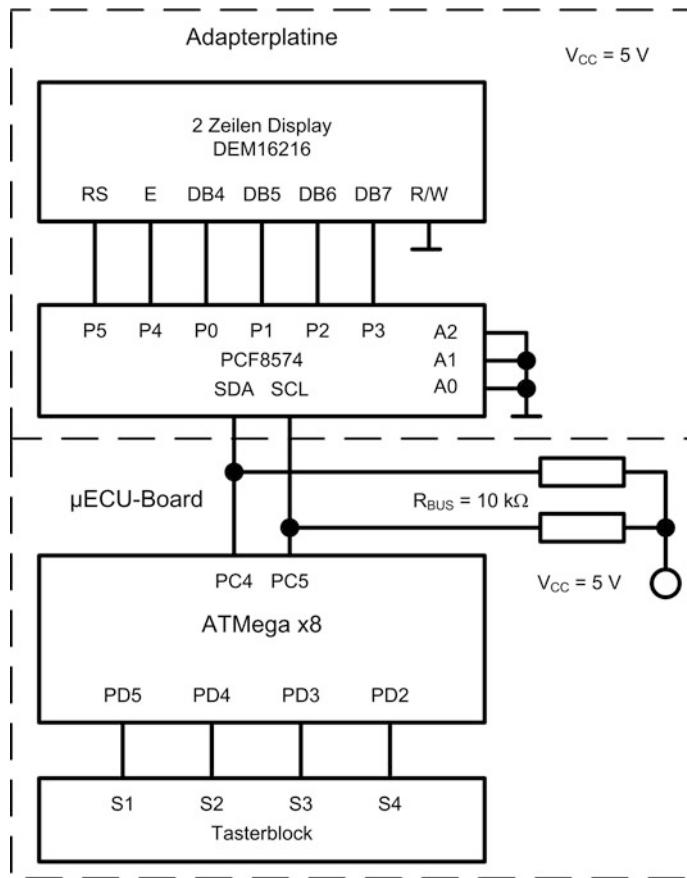


Abb. 8.9 Serielle Ansteuerung eines parallelen LCD-Displays

8.3.1 Display Ansteuerung über I²C

Der zeitliche Ablauf der Display-Ansteuerung und die Kommunikation mit dem Port Expander wurden im Abschn. 8.2.3 beziehungsweise in Abschn. 7.1 beschrieben. Im Folgenden wird als Beispiel die Anzeige an der aktuellen Cursorposition des Zeichens „A“ (ASCII Wert 0x41) vorgestellt. Der Ausgangsverlauf des Port Expanders zur Ansteuerung des Displays ist in der Tab. 8.6 dargestellt.

Der Baustein PCF8574 muss vor der Übertragung der fünf Bytes adressiert werden. Der gesamte Zeitverlauf der I²C Kommunikation ist in der Abb. 8.10 dargestellt. Die Taktfrequenz des Busses beträgt in diesem Fall max. 100 kHz, was bedeutet, dass die Übertragung eines Bytes mindestens 90 µs dauert. Für das betrachtete Beispiel werden mit dem Adressbyte des Port Expander insgesamt sechs Bytes übertragen, was bedeutet, dass die Anzeige eines ersten Zeichens ca. 540 µs dauert. Um anschließend ein weiteres

Tab. 8.6 PCF8574 Ausgangsverlauf um das Datenbyte „A“ anzuzeigen

Beschreibung	P7	P6	P5 RS	P4 E	P3 DB7	P2 DB6	P1 DB5	P0 DB4
Das RS-Bit wird auf 1 gesetzt, es folgt die Übertragung eines Datenbytes	0	0	1	0	0	0	0	0
Das E-Bit wird auf 1 gesetzt, die höherwertige Bytehälfte wird übertragen	0	0	1	1	0	1	0	0
Das E-Bit wird auf 0 gesetzt um die höherwertige Bytehälfte in das Eingangsregister vom Display zu speichern	0	0	1	0	0	1	0	0
Das E-Bit wird auf 1 gesetzt, die niederwertige Bytehälfte wird übertragen	0	0	1	1	0	0	0	1
Die niederwertige Bytehälfte wird in das Eingangsregister vom Display übernommen	0	0	1	0	0	0	0	1

Zeichen anzuzeigen, werden in der gleichen I²C-Botschaft nur noch 4 Bytes übertragen, weil die Adresse und das Setzen des RS-Bits entfallen. Die Übertragung eines Display-Befehls sieht ähnlich aus mit dem Unterschied, dass das RS-Bit auf 0 gesetzt wird. Eine Wartezeit wird bei dieser Ansteuerung nur noch für die Befehle „Display löschen“ und „Display Grundeinstellung“ benötigt.

8.3.2 Software Beispiel: Übertragung eines Datenbytes

Aus Portabilitätsgründen der Software werden die Verbindungen zwischen dem Display und dem Port Expander wie folgt definiert:

```
#define DISP_TWI_RS_BIT          0x20
#define DISP_TWI_E_BIT            0x10
#define DISP_TWI_DATA_DB7_BIT    0x08
#define DISP_TWI_DATA_DB6_BIT    0x04
#define DISP_TWI_DATA_DB5_BIT    0x02
#define DISP_TWI_DATA_DB4_BIT    0x01

#define DISP_TWI_RS_SET_ON        DISP_TWI_RS_BIT
#define DISP_TWI_RS_SET_OFF       (~DISP_TWI_RS_SET_ON)
#define DISP_TWI_E_SET_ON         DISP_TWI_E_BIT
#define DISP_TWI_E_SET_OFF        (~DISP_TWI_E_SET_ON)

#define DISP_TWI_DATA_MASK  (~(DISP_TWI_DATA_DB7_BIT
                           | DISP_TWI_DATA_DB6_BIT
                           | DISP_TWI_DATA_DB5_BIT
                           | DISP_TWI_DATA_DB4_BIT))
```

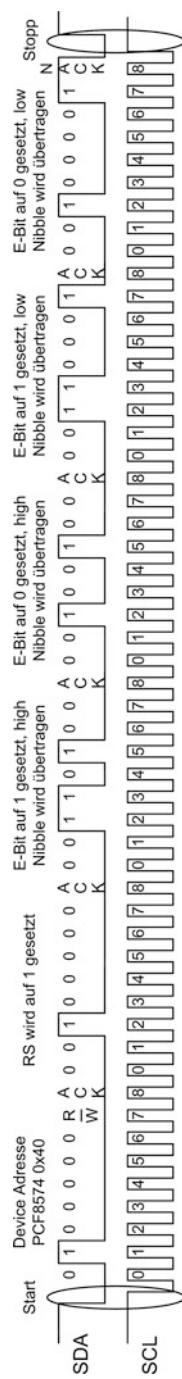


Abb. 8.10 I²C Kommunikation – Anzeige des Zeichens „A“ am Display

Es wurde berücksichtigt, dass das Bit P7 den höchsten und P0 den niedrigsten Stellenwert hat. Dieser Teil muss abhängig von der konkreten Beschaltung eventuell geändert werden um den Code unverändert verwenden zu können.

Die Funktion `DispTWI_Write_Data`, deren Code weiter aufgelistet wird, dient dazu, das an dem Port Expander angeschlossene Display anzusteuern, um ein Zeichen anzuzeigen. Als Parameter werden die Device Chip Adresse des Bausteins PCF8574 und der ASCII-Wert des anzuzeigenden Zeichens übergeben. Die Funktion gibt den Wert `TWI_OK` bei fehlerfreier I²C-Übertragung zurück, ansonsten `TWI_ERROR`. Die Variable `ucDataBuffer` speichert die zwei Bytehälften des Zeichens. Mit dem fett gedruckten Teil der Funktion wird eine flexible Pin-Verbindung zwischen Display und Port Expander ermöglicht. Die in der Funktion berechneten Bytes werden entsprechend Tab. 8.6 übertragen.

```
uint8_t DispTWI_Write_Data(uint8_t ucdevice_address,
                           uint8_t ucascii_of_char)
{
    uint8_t ucDispDataBit[4] = {DISP_TWI_DATA_DB7_BIT,
                               DISP_TWI_DATA_DB6_BIT,
                               DISP_TWI_DATA_DB5_BIT,
                               DISP_TWI_DATA_DB4_BIT};
    uint8_t ucDeviceAddress, ucDispTWIData;
    uint8_t ucMask, ucI, ucJ;
    uint8_t ucDataBuffer[2];
    ucDataBuffer[0] = ucascii_of_char >> 4;
    ucDataBuffer[1] = ucascii_of_char & 0x0F;
    ucDeviceAddress = (ucdevice_address << 1)
                      | PCF8574_DEVICE_TYP_ADDRESS;
    ucDeviceAddress |= TWI_WRITE; //Write-Modus
    TWI_Master_Start(); //Start
    if((TWI_STATUS_REGISTER) != TWI_START) return TWI_ERROR;
    TWI_Master_Transmit(ucDeviceAddress); //Device Adresse senden
    if((TWI_STATUS_REGISTER) != TWI_MT_SLA_ACK) return TWI_ERROR;
    ucDispTWIData = DISP_TWI_RS_SET_ON; //RS-Bit wird auf 1 gesetzt
    TWI_Master_Transmit(ucDispTWIData); //RS-Leitung wird auf
                                         //High gesetzt
    if((TWI_STATUS_REGISTER) != TWI_MT_DATA_ACK) return TWI_ERROR;
    for(ucJ = 0; ucJ < 2; ucJ++)
    {
        ucDispTWIData |= DISP_TWI_E_SET_ON; //EN wird auf High gesetzt
        ucMask = 0x08;
        ucDispTWIData &= DISP_TWI_DATA_MASK; //die Datenbits werden
                                             //auf 0 gesetzt
        for(ucI = 0; ucI < 4; ucI++)
        {
            if(ucDataBuffer[ucJ] & ucMaske) ucDispTWIData
```

```

        |= ucDispDataBit[ucI];
        ucMask = ucMask >> 1;
    }
    TWI_Master_Transmit(ucDispTWIData); //das Datennibble
                                         //wird gesendet
    if((TWI_STATUS_REGISTER) != TWI_MT_DATA_ACK) return TWI_ERROR;
    ucDispTWIData &= DISP_TWI_E_SET_OFF; //EN wird auf Low gesetzt
    TWI_Master_Transmit(ucDispTWIData);
    if((TWI_STATUS_REGISTER) != TWI_MT_DATA_ACK) return TWI_ERROR;
}
TWI_Master_Stop(); //Stop
return TWI_OK;
}

```

8.4 DOGS102-6 Graphisches Display mit serieller Ansteuerung

Die Displays der Reihe DOGS102-6 sind passive STN Punktmatrix LCD-Displays [5]. Die Größe der Displays, der niedrige Energieverbrauch und die gute Ablesbarkeit auch bei Sonnenlicht ermöglichen ihren Einsatz in hand-held Geräten. Sie werden mit einem Controller vom Typ UC1701x [6] angesteuert. Der Controller hat für die Datenkommunikation mit dem ansteuernden Mikrocontroller eine 8-Bit parallele Schnittstelle und eine serielle SPI-Schnittstelle implementiert. Er ermöglicht die Ansteuerung eines Displays mit bis zu 65×132 Punkten. Eine Beispielschaltung ist in der Abb. 8.11 dargestellt.

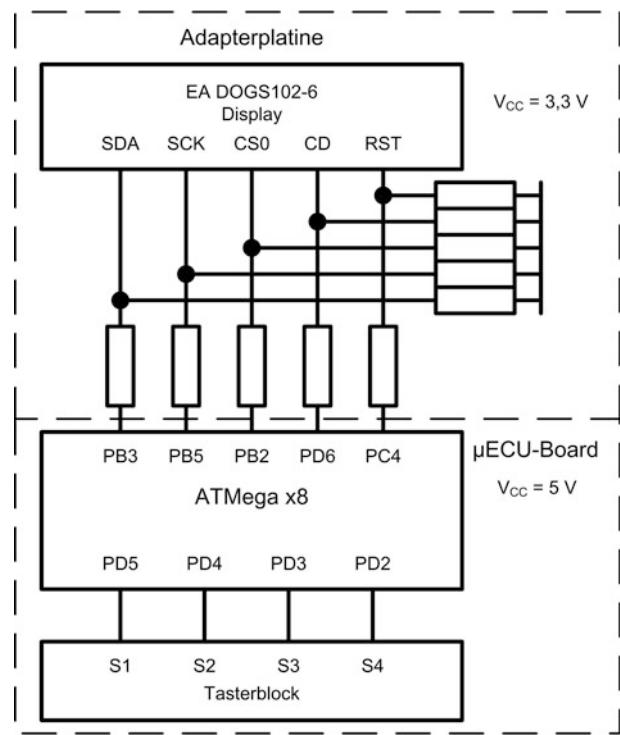
8.4.1 Struktur des graphischen Displays DOGS 102-6

Ein grobes Blockschaltbild des Displays ist in der Abb. 8.12 zu sehen. Das Display kann mit einer einzigen 3,3 V Spannung versorgt werden, die über eine interne Ladungspumpe die Kontrastspannung erzeugt. Die Kontrastspannung kann einmalig eingestellt werden, um die Herstellungsdifferenzen auszugleichen. Intern findet eine Temperaturkompensation statt, deren Koeffizient einstellbar ist. Über den RST-Eingang wird der Display Controller zurückgesetzt, die Ladungspumpe abgeschaltet, die Treiber für die Spalten- und die Zeilelektroden werden deaktiviert und die Register mit den Werkseinstellungen geladen. Um das zu realisieren, muss der Eingang, nachdem die Versorgungsspannung eine voreingestellte Schwelle überschritten hat, für 1 ms auf Low geschaltet werden und danach wieder auf High. Nach weiteren 5 ms kann der Controller Befehle empfangen und verarbeiten.

Für die Kommunikation mit einem Mikrocontroller gibt es eine unidirektionale SPI-Schnittstelle bestehend aus den Anschlüssen SDA (MOSI), SCK und CS0. Die übertragenen Bytes werden entweder wenn der Eingang CD¹¹ auf Low ist, in den Befehlsdekor

¹¹ CD-Control Data.

Abb. 8.11 DOGS Display
Ansteuerung



geladen und als Befehl dekodiert oder, wenn CD auf High ist in den Display Data RAM-Speicher gespeichert.

Der DDRAM-Speicher ist in 8 Seiten mit jeweils 102 Bytes organisiert. Sein Inhalt wird vom Display abgebildet, so wie in der Tab. 8.7 dargestellt ist. 8 Displayzeilen bilden

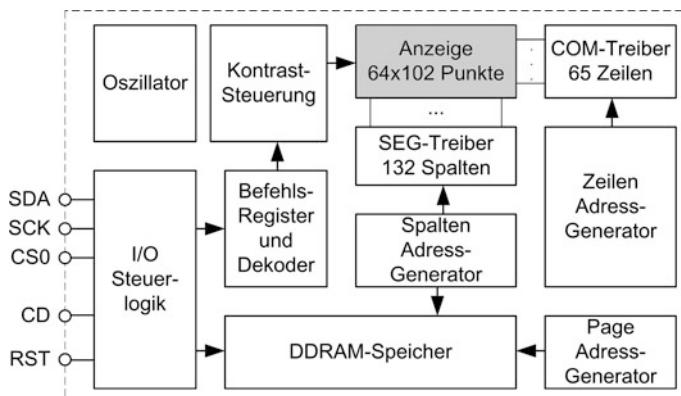


Abb. 8.12 Display DOGS102-6 Blockschaltbild

Tab. 8.7 Zusammenhang zwischen DDRAM-Inhalt und Pixel Darstellung. Dieser Zusammenhang gilt bei normalen Anzeige (keine Invertierung) und Startzeile gleich 0

DDRAM		Display	
Page	Datenbyte m 0xD9	Spalte m	Zeilennummer
n	D0 = 1		8 * n
	D1 = 0		8 * n + 1
	D2 = 0		8 * n + 2
	D3 = 1		8 * n + 3
	D4 = 1		8 * n + 4
	D5 = 0		8 * n + 5
	D6 = 1		8 * n + 6
	D7 = 1		8 * n + 7

eine Speicherseite. Mit dem Page- und Spaltenzähler wird eine Speicherzelle adressiert, über dessen Inhalt acht Pixel angesteuert werden. In der normalen Darstellung wird ein Pixel eingeblendet, wenn das entsprechende Bit „1“ ist. Der Displaycontroller ermöglicht die Invertierung der Reihenfolge bei der Ansteuerung der Zeilen und Spalten, was zu einer vertikalen oder horizontalen Spiegelung der Darstellung führt, ohne den Inhalt des DDRAM-Speichers ändern zu müssen. Zusätzlich ist das Einblenden aller Pixel oder eine Invertierung der Anzeige möglich.

Die Ansteuerung vom Display macht sowohl eine 6:00 Uhr als auch eine 12:00 Uhr Einbaulage möglich, wie in Abb. 8.13 dargestellt. Die Page-Nummerierung beginnt immer von oben bei 0. In Abb. 8.13a werden die Spalten beginnend von links von 0 bis 101 nummeriert, in Abb. 8.13b von 30 bis 131.

8.4.2 SPI-Kommunikation

Das Display ist als SPI-Slave voreingestellt und kann im SPI-Modus 3, das höchstwertige Bit zuerst, bei einer Taktfrequenz von bis zu 33 MHz Daten empfangen. Ein steuernder

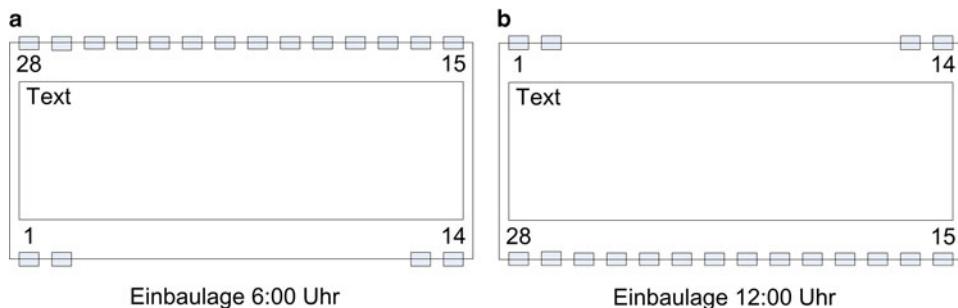


Abb. 8.13 DOGS Display Einbaulage

Mikrocontroller muss entsprechend als Master eingestellt werden. Bei unterschiedlichen Versorgungsspannungen des Displays und Mikrocontrollers muss der Spannungspiegel am Display-Eingang angepasst werden, wie in Abb. 8.11 gezeigt wird.

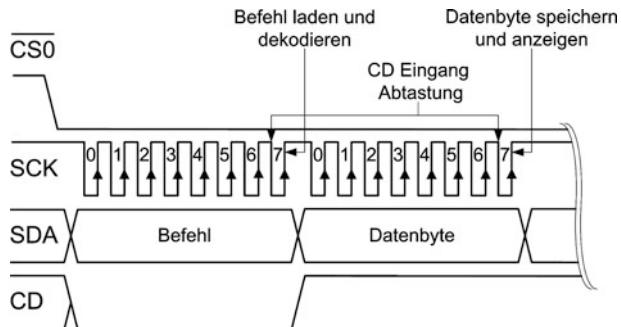
Die SPI-Datenstruktur kodiert folgende Steuereingänge: Chip Select, Control Data und Reset und sieht für die Beispielschaltung so aus:

```
DispDOGS_pins DispDOGS_1 ={{/*CS_DDR*/      &DDRB,
                            /*CS_PORT*/     &PORTB,
                            /*CS_pin*/      PB2,
                            /*CS_state*/    ON},
                            /*CD_DDR*/      &DDRD,
                            /*CD_PORT*/     &PORTD,
                            /*CD_pin*/      PD6,
                            /*CD_state*/    ON,
                            /*RST_DDR*/     &DDRC,
                            /*RST_PORT*/    &PORTC,
                            /*RST_pin*/     PC4,
                            /*RST_state*/   ON};
```

In Abb. 8.14 ist prinzipiell die SPI-Kommunikation mit dem Display dargestellt, die vom Mikrocontroller mit dem Schalten des Chip-Select Signals auf Low initiiert wird. Auf der fallenden Flanke jedes achten Taktes eines Bytes wird intern der Zustand des CD-Eingangs abgetastet und auf der steigenden Flanke dieses Taktes wird das empfangene Byte entweder in den Befehlsdecoder geladen und dekodiert oder in das DDRAM gespeichert. Der dekodierte Befehl wird gleich ausgeführt, das gespeicherte Byte gleich dargestellt, es wird nicht auf das Ende der Kommunikation gewartet. Es ist zwischen der Übertragung einzelnen Bytes keine Wartezeit erforderlich.

Ein Beispiel für die Umsetzung des in Abb. 8.14 dargestellten Ablaufs zeigt folgender Codeausschnitt. In diesem Beispiel wird das gesamte Display durch das Speichern einer „0“ an jeder Adresse gelöscht.

Abb. 8.14 DOGS Display
SPI-Kommunikation



```

SPI_Master_Start(sdevice_pins.DispDOGSSpi); //die CS-Leitung wird auf
                                              //Low gesetzt
for(ucPage = 0; ucPage < 8; ucPage++)
{ //mit dem Setzen der CD-Leitung auf Low, wird das folgende Byte als
  //Befehl interpretiert
  DispDOGS_Set_CDLow(sdevice_pins);
  SPI_Master_Write(SET_PAGE_ADDRESS | ucPage); //die Seitenadresse
                                                //wird gesetzt
  DispDOGS_Set_Column(sdevice_pins, 0x00); //die Anfangsspalte
                                             //wird gesetzt
  DispDOGS_Set_CDHigh(sdevice_pins); //die folgenden Bytes
                                             //sind Datenbytes
  for(ucColumn = 0; ucColumn < 102; ucColumn++)
  {
    SPI_Master_Write(0x00);
  }
}
SPI_Master_Stop(sdevice_pins.DispDOGSSpi); //Ende der Übertragung

```

8.4.3 Befehlssatz

Die Befehlscodes (siehe Tab. 8.8) werden über den SPI-Bus übertragen und der Zustand des CD-Eingangs bestimmt, ob das empfangene Byte in den DDRAM-Speicher oder in den Befehlsdekompler gespeichert wird. Die Serie DOGS102-6 implementiert aus dem gesamten Befehlssatz des Display Controllers UC1701x folgende Funktionen:

- **Pageadresse wählen** (Set Page Address, PA – Page Address) mit den vier Bits PA0:3 wird die Seite (0 die oberste, 7 die unterste Seite) für das Speichern des nächsten Bytes gewählt;
- **Spaltenadresse wählen** (Set Column Address, CA – Column Adress) ist ein 2 Byte Befehl um die Spalte für das Speichern des nächsten Bytes zu wählen. In der Ansicht von unten beginnt die Nummerierung der Spalten von links bei 0 bis 101, bei der Ansicht von oben beginnt sie von links bei 30 bis 131;
- **Datenbyte speichern** (Write Data Byte) mit dem CD-Eingang auf „1“ wird das Byte in den DDRAM-Speicher in der aktuell adressierten Speicherzelle gespeichert. Die Spaltenadresse wird inkrementiert;
- **Display zurücksetzen** (System Reset) der Befehl setzt die interne Register auf die voreingestellten Werte;
- **Display Modus wählen** (Set Display Enable, DC – Display Control) versetzt das Display in den aktiven Modus für DC2 = 1 oder in den Sleep-Modus für DC2 = 0;

- **Ladungspumpe steuern** (Set Power Control, PC – Power Control) schaltet die Ladungspumpe für die Erzeugung einer LCD-Hochspannung bei PC_{2...0}=111 ein oder bei PC_{2...0}=000 aus;
- **Kontrast Grobeinstellung** (Set VLCD Resistor Ratio) stellt die Hochspannung für den Kontrast grob ein;
- **Kontrast Feineinstellung** (Set Electronic Volume) stellt die Hochspannung für den Kontrast fein ein. Um eine gute Zuverlässigkeit vom Display zu gewährleisten, soll die VLCD Kontrastspannung den Wert 11,5 V nicht übersteigen (siehe Tab. 8.9);
- **LCD Vorspannung wählen** (Set LCD Bias Ratio, BR – Bias Ratio) stellt das Verhältnis zwischen der LCD-Spannung und Vorspannung auf 1/9 für BR=0 oder 1/7 für BR=1 ein;
- **Startzeile wählen** (Set Scroll Line, SL – Scroll Line) wählt über die Bits SL_{5...0} die Startzeile des Displays aus. Das Ausführen dieses Befehls führt zu einem Bildschirmrollen nach oben um SL Displayzeilen;
- **Alle Pixel einblenden** (Set All Pixel On) blendet alle Pixel ein für DC₁=1, ändert aber den Inhalt des DDRAM-Speichers nicht; für DC₁=0 wird das Display angesteuert um den RAM-Inhalt anzuzeigen;
- **Display Invertierung** (Set Inverse Display) blendet für DC₀=0 ein Pixel ein, wenn das entsprechende Bit aus dem Speicher „1“ ist, oder für DC₀=1 wenn das Bit „0“ ist;
- **Vertikale Spiegelung** (Set COM Direction, MY – Mirror Y-axle) legt die Reihenfolge der Zeilen fest; für MY=0 ist die Reihenfolge 0...63 während sie für MY=1 63...0 ist, was zu einer vertikalen Spiegelung der Darstellung führt;
- **Horizontale Spiegelung** (Set SEG Direction, MX – Mirror X-axle) legt die Reihenfolge der Spalten fest; für MX=0 ist die Reihenfolge 0...131 während sie für MX=1 131...0 ist, was zu einer horizontalen Spiegelung der Darstellung führt;
- **Sondereinstellungen** (Set Advanced Program Control 0); der Befehl stellt den Temperaturkompensierungskoeffizient der LCD-Vorspannung auf -0,05 %/°C für TC=0 oder auf -0,11 %/°C für TC=1; beim Überschreiten der letzten Spalte wird auf die erste gesprungen, wenn WP=1, ähnlich bei den Speicherseiten für WA=1.

Die Displayeinstellungen werden in einen RAM-Speicher gespeichert. Deshalb muss das Display nach dem Hochfahren neu initialisiert werden. Für eine 6:00 Uhr Einbaulage kann die Initialisierungen durch folgende Schritte realisiert werden:

- Startzeile auf 0 setzen (SL_{5...0}=000000b);
- Horizontale Spiegelung an (MX=1);
- Vertikale Spiegelung aus (MY=0);
- das Einblenden aller Pixel ausschalten (DC₁=0);
- Display Invertierung aus (DC₀=0);
- LCD Vorspannung wählen (BR=0);
- Ladungspumpe einschalten (PC_{2...0}=111b);
- Kontrast grobeinstellen (PC_{5...3}=111b);

Tab. 8.8 Befehlssatz eines Displays vom Typ DOGS102-6

Befehl	Befehlscode									
	CD	D7	D6	D5	D4	D3	D2	D1	D0	
Pageadresse wählen	0	1	0	1	1	PA3	PA2	PA1	PA0	
Spaltenadresse wählen	0	0	0	0	0	CA3	CA2	CA1	CA0	
		0	0	0	1	CA7	CA6	CA5	CA4	
Datenbyte speichern	1	Datenbyte								
Display zurücksetzen	0	1	1	1	0	0	0	1	0	
Display Modus wählen	0	1	0	1	0	1	1	1	DC2	
Ladungspumpe steuern	0	0	0	1	0	1	PC2	PC1	PC0	
Kontrast Grobeinstellung	0	0	0	1	0	0	PC5	PC4	PC3	
Kontrast Feineinstellung	0	1	0	0	0	0	0	0	1	
		0	0	PM5	PM4	PM3	PM2	PM1	PM0	
LCD-Vorspannung wählen	0	1	0	1	0	0	0	1	BR	
Startzeile wählen	0	0	1	SL5	SL4	SL3	SL2	SL1	SL0	
Alle Pixel einblenden	0	1	0	1	0	0	1	0	DC1	
Display-Invertierung	0	1	0	1	0	0	1	1	DC0	
Vertikale Spiegelung	0	1	1	0	0	MY	0	0	0	
Horizontale Spiegelung	0	1	0	1	0	0	0	0	MX	
Sondereinstellungen	0	1	1	1	1	1	0	1	0	
		TC	0	0	1	0	0	WC	WP	

Tab. 8.9 Einstellbare Kontrastspannungsbereiche

PC5..3	PM5..0	VLCD Spannungsbereich [V]
000	0..63	3,87..6,33
001	0..63	4,51..7,38
010	0..63	5,15..8,43
011	0..63	5,79..9,48
100	0..63	6,43..10,53
101	0..63	7,08..11,51
110	0..63	7,72..11,46
111	0..63	8,36..11,48

- Kontrast feineinstellen (PM5...0 = 010000b);
- Sondereinstellungen (TC = 1, WC = WP = 0);
- Display aktiv Modus wählen (DC2 = 1);

Um diese Schritte umzusetzen, müssen die in den Klammern vorgeschlagenen Parameter in den entsprechenden Befehlscodes eingesetzt werden und über SPI übertragen werden, während die CD-Leitung auf Low ist. Das Zusammenfassen der Codes in einem Array führt zu einem kompakteren Programmcode wie im folgenden Beispiel.

```

#define SET_SCROLL_LINE           0x40 //Startzeile setzen 0x40
#define SET_SEG_DIRECTION_MIRROR 0xA1 //horizontale Spiegelung an
#define SET_COM_DIRECTION_NORMAL 0xC0 //vertikale Spiegelung aus
#define SET_ALL_PIXEL_ON_DISABLE 0xA4 //Einblenden aller Pixel aus
#define SET_INVERSE_DISPLAY_OFF   0xA6 //Display Invertierung aus
#define SET_LCD_BIAS_RATIO_1_9    0xA2 //LCD Vorspannung wählen
#define SET_POWER_CONTROL_ALL_ON  0x2F //Ladungspumpe einschalten
#define SET_VLCD_RESISTOR_RATIO   0x27 //Kontrast Grobeinstellung
#define SET ELECTRONIC_VOLUME_1    0x81 //1. Byte der Kontrast
                                         //Feineinstellung
#define SET ELECTRONIC_VOLUME_2    0x07 //2. Byte der Kontrast
                                         //Feineinstellung
#define SET_ADVANCED_PROGRAM_CONTROL_0 0xFA //Temperaturkoeffizient
                                         //Einstellung (1. Byte)
#define TEMP_COMP_011              0x91 //Auswahl Temperaturkoeffizient
                                         //(2. Byte)
#define SET DISPLAY_ON              0xAF //Display in aktiv Modus
                                         //versetzen

uint8_t ucDispDOGSInitValue[13] ={SET_SCROLL_LINE,
                                  SET_SEG_DIRECTION_MIRROR,
                                  SET_COM_DIRECTIONNORMAL, SET_ALL_PIXEL_ON_DISABLE,
                                  SET_INVERSE_DISPLAY_OFF, SET_LCD_BIAS_RATIO_1_9,
                                  SET_POWER_CONTROL_ALL_ON, SET_VLCD_RESISTOR_RATIO,
                                  SET_ELECTRONIC_VOLUME_1, SET_ELECTRONIC_VOLUME_2,
                                  SET_ADVANCED_PROGRAM_CONTROL_0,
                                  TEMP_COMP_011, SET_DISPLAY_ON};

void DispDOGS_Init(DispDOGS_pins sdevice_pins)
{
    //der Slave Select Anschluss des Displays wird initialisiert
    // (Ausgang auf High gesetzt)
    SPI_Master_SlaveSelectInit(sdevice_pins.DispDOGSspi);
    DispDOGS_Init_RST(sdevice_pins); //Initialisierung des RST-Anschlusses
    DispDOGS_Init_CD(sdevice_pins); //Initialisierung des CD-Anschlusses
    DispDOGS_Set_RSTHigh(sdevice_pins); //RST Leitung auf High setzen
    //20ms Wartezeit
    while(!Timer1_get_10msState());
    while(!Timer1_get_10msState());
    DispDOGS_Set_CDLow(sdevice_pins);
    //die Übertragung der Initialisierungswerte wird gestartet
    SPI_Master_Start(sdevice_pins.DispDOGSspi);
    //die Initialisierungswerte werden übertragen
    for(uint8_t ucI = 0; ucI < 13; ucI++)

```

```

SPI_Master_Write(ucDispDOGSInitValue[ucI]) ;
SPI_Master_Stopp(sdevice_pins.DispDOGSspi) ;
DispDOGS_Clear_Memory(sdevice_pins); //der gesamte Inhalt des
//Speichers wird gelöscht
}

```

8.4.4 Generierung eines Zeichens

Um Texte auf einen DOGS-Display darzustellen, müssen alle Zeichen softwaremäßig generiert werden, weil das Display selber keinen Zeichengenerator hat. Es gibt keine Einschränkungen, was die Größe des Zeichensatzes betrifft. Wenn man den Aufbau des Displays berücksichtigt, ist ein 5×7 Pixel großer Zeichensatz bequem zu implementieren. Diese Zeichensatzgröße passt auf die Höhe einer Speicherseite (1 Byte) und man kann das unterste oder oberste Pixel zur Trennung zwischen zwei Textzeilen benutzen. Beispielhaft wird weiterhin die Generierung des Zeichens „A“ präsentiert, dessen Bitmuster in Tab. 8.10 zu sehen ist. Für die Trennung zwischen den Textzeilen wird der unterste Bildpunkt des Zeichensatzes benutzt. Das sechste Generierungsbyte ist nicht unbedingt nötig, dient aber zur Trennung zwischen zwei benachbarten Zeichen und bietet den Vorteil, dass für die Anzeige eines Textes nur die Anfangsadresse der Darstellung gesetzt werden muss. Mit diesen Überlegungen ist der Zeichensatz 6×8 Pixel groß und auf dem gesamten Display können 8×17 Zeichen dieser Größe dargestellt werden.

Die entstandenen Generierungsbytes für das Zeichen „A“ werden in ein Array zusammengefasst, um den Programmcode kompakter zu gestalten:

```
uint8_t ucZeichen_A[6] = {0x7E, 0x11, 0x11, 0x11, 0x7E, 0x00};
```

Mit dem folgenden Programmausschnitt aus der main-Funktion wird das Zeichen „A“ auf einem Display vom Typ DOGS102-6 mit der SPI-Datenstruktur DispDOGS_1 in der

Tab. 8.10 Bitmuster des Zeichens „A“

		Spalte					
Page		m	m + 1	m + 2	m + 3	m + 4	m + 5
n	2^0	0	1	1	1	0	0
	2^1	1	0	0	0	1	0
	2^2	1	0	0	0	1	0
	2^3	1	0	0	0	1	0
	2^4	1	1	1	1	1	0
	2^5	1	0	0	0	1	0
	2^6	1	0	0	0	1	0
	2^7	0	0	0	0	0	0
		0x7E	0x11	0x11	0x11	0x7E	0x00

Seite „n“, anfangend mit der Spalte „m“ dargestellt. Die Seitenzahl „n“ muss eine Zahl zwischen 0 und 7 sein, die Spaltenzahl, abhängig von der Orientierung des Displays, eine Zahl zwischen 0 und 101 oder zwischen 30 und 131.

```
//Set Page Address
#define SET_PAGE_ADDRESS          0xB0
//Set Column Address
#define SET_COLUMN_ADDRESS_LSB    0x00
#define SET_COLUMN_ADDRESS_MSB    0x10

SPI_Master_Start(DispDOGS_1.DispDOGSSpi); //SPI-Kommunikation wird
                                            //gestartet
DispDOGS_Set_CDLow(DispDOGS_1); //der CD-Eingang wird auf Low gesetzt
/* Seitenwahl*/
SPI_Master_Write(SET_PAGE_ADDRESS | n);
/*Spaltenwahl - 2Byte Befehl*/
SPI_Master_Write(SET_COLUMN_ADDRESS_LSB | (m % 16));
SPI_Master_Write(SET_COLUMN_ADDRESS_MSB | (m / 16));
DispDOGS_Set_CDHigh(DispDOGS_1); //der CD-Eingang wird auf High gesetzt
/*die sechs Generierungsbytes des Zeichens A werden übertragen*/
for(uint8_t ucI = 0; ucI < 6; ucI++)
{
    SPI_Master_Write(ucZeichen_A[ucI]);
}
SPI_Master_Stop(DispDOGS_1.DispDOGSSpi); //SPI-Kommunikation wird beendet
```

Entsprechend diesem Beispielcode werden für die Darstellung eines Zeichens dieser Größe 9 Bytes seriell an das Display übertragen: drei für die Auswahl der Textzeile und der Anfangsspalte und sechs für die Visualisierung des Bitmusters. Bei einer Taktfrequenz von 4 MHz ergibt sich eine Übertragungsdauer der 9×8 Bits von 18 µs. Für die Darstellung eines weiteren Zeichens rechts zum Vorigen, werden nur noch 12 µs gebraucht (Übertragungsdauer von 6 Datenbytes) weil die Adressen der Zeile, bzw. der Anfangsspalte nicht explizit gesetzt werden müssen. Am einfachsten ist es, wenn man die einzelnen Zeichen (Buchstaben, Ziffern und darstellbare Sonderzeichen) über Ihren ASCII-Wert adressieren kann. Ein Zeichensatz aller Zeichen der ASCII-Tabelle mit Werten zwischen 32 und 127 enthält 96 Elemente a 6 Bytes. Die Adresse eines Zeichens errechnet sich aus dem ASCII-Wert des Zeichens minus 32 (Offset der neuen Tabelle). Ein beliebiges Zeichen ucZeichen aus dem wie oben beschriebenen Zeichensatz, der in der Tabelle ucChar6x8 [96] [6] gespeichert ist, kann mit dem Beispielcode dargestellt werden, indem der Befehl SPI_Master_Write(ucZeichen_A[ucI]) mit dem SPI_Master_Write(ucChar6x8 [ucZeichen - 32] [ucI]) getauscht wird.

Literatur

1. Meroth, A., Tolg, B.: Infotainmentsysteme im Kraftfahrzeug. Vieweg, GWV Fachverlage, Wiesbaden (2008)
2. Gevatter, J.H., Grünhaupt, U.: Handbuch der Mess- und Automatisierungstechnik im Automobil. Springer, Berlin, Heidelberg (2006)
3. Samsung Electronics: KS0070B – driver & controller for dot matrix LCD (2015). www.datasheetcatalog.com, Zugegriffen: 13. April 2018
4. Display Elektronik: LCD module DEM 16481 SYH-LY (2015). www.display-elektronik.de, Zugriffen: 28. Juli 2015
5. Electronic Assembly: DOGS Graphic Series 102x64 Dots (2015). www.lcd-module.de, Zugriffen: 18. Mai 2015
6. UltraChip: UC1701x 65x132 STN controller driver (2015). www.ultrachip.com, Zugriffen: 21. Mai 2015

Weiterführende Literatur

7. Reisch, M.: Elektronische Bauelemente. Springer, Berlin Heidelberg New York (2006)

Beispielprojekt: Datenlogger

9

Zusammenfassung

In diesem Kapitel wird ein einfacher Datenlogger vorgestellt, der im Modellflug oder bei der Transportsicherung verwendet werden kann. In der vorgestellten Ausführung schreibt der Datenlogger die Beschleunigung in zwei Achsen, den Luftdruck und die Temperatur, sowie daraus abgeleitet die barometrische Höhe in einen Flash-Speicher und liest diese über einen Mini-USB-Anschluss mit einem virtuellen seriellen Port wieder aus. Im vorliegenden Projekt wurde er für die Messung der Flugeigenschaften von Modellraketen eingesetzt.

9.1 Aufbau der Modellrakete

Modellraketen sind kleine und leichte Flugkörper, die mit einem Festbrennstoff angetrieben werden. Unter einer bestimmten Größe, beziehungsweise einem bestimmten Gewicht und mit zugelassenen Motoren dürfen sie genehmigungsfrei geflogen werden, wenn bestimmte Auflagen erfüllt sind. In Deutschland darf das Gewicht des Treibstoffs 20 g bei einem genehmigungsfreien Modell nicht übersteigen. Der Betrieb ist nur durch Personen über 14 Jahre und unter Zustimmung von Erziehungsberechtigten gestattet, die Abgabe von Motoren nur an Personen über 18 Jahren. Für größere Flugkörper benötigt man ggf. eine Zulassung und eine Lizenz als Betreiber. Die Rakete wurde vom Schoolab des DLR Campus Lampoldshausen (Deutsche Gesellschaft für Luft- und Raumfahrttechnik) gebaut. Sie wiegt inklusive des Motors ca. 155 g, hat einen Durchmesser von 4,1 cm und ist 91,7 cm lang. Ihre Steighöhe wurde im Vorfeld mit ca. 168 m bei einer Maximalbeschleunigung von 124 m/s und einer Maximalgeschwindigkeit von 69 m berechnet. Die Berechnung wie auch die Zeichnung entstammen der Software open rocket [1].

In Abb. 9.1 oben ist ein Modell der Rakete zu sehen. Deutlich erkennbar sind im Schnitt der Motor und eines der beiden Fallschirmpakete. Die beiden Hauptrohre werden durch

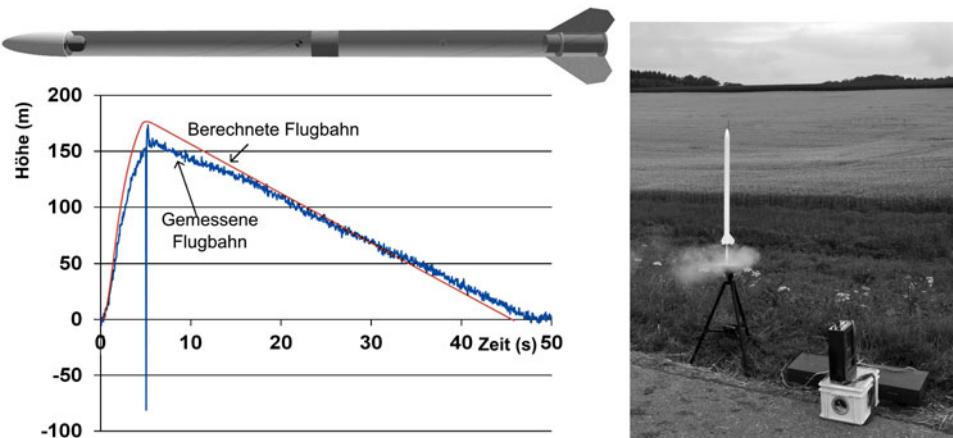


Abb. 9.1 Modellrakete mit Flugverlauf

eine Muffe zusammengehalten. Als Motor wurde ein im Modellbauhandel beziehbarer Treibsatz der Klasse D7-3 von WECO eingebaut. Dieser entwickelt einen Gesamtimpuls von 13,9 Ns und einen maximalen Schub von 20,4 N während einer Brenndauer von 1,48 s. Nach dem Abbrennen der Ladung fliegt die Rakete im freien ballistischen Flug noch 3 s, anschließend wird die Spitze mit der Nutzlast durch eine Treibladung ausgestoßen, Raketenkörper und Spitze fallen an Fallschirmen wieder zu Boden.

Da die Steighöhe mit Hilfe der barometrischen Höhenformel aus dem Luftdruck und der Temperatur ermittelt wurde, wird der Druckstoß beim Ausstoßen der Spitze als Höhenimpuls wahrgenommen. In Abschn. 9.4 werden auch die Beschleunigungsverläufe diskutiert.

9.2 Beschreibung des Projekts

Das Projekt wurde mit einem ATmega88 im Gehäuse TQFP realisiert. Als Drucksensor wurde der in Abschn. 6.3 beschriebene MPL3115 eingesetzt. Als Beschleunigungssensor wurde ein MMA6525 und als externer Flashspeicher ein SST25PF (Kap. 7) verwendet. Die Umsetzung auf die USB-Schnittstelle erfolgte durch den Baustein FT232RL von FTDIchip (Abschn. 5.2.4).

Die gesamte Schaltung wird mit 8 MHz getaktet und mit einer Lithiumbatterie betrieben, die wegen der hohen Beschleunigungen auf das Board aufgelötet ist. Um beim Auslesen die Schaltung über die USB-Schnittstelle zu versorgen, setzt ein Spannungsregler vom Typ TS5204 die 5 V Spannung der USB-Schnittstelle auf die 3,6 V Versorgungsspannung des Boards herunter. Bei der Bauteileauswahl spielte insbesondere die Baugröße eine entscheidende Rolle, auf die dritte Beschleunigungsachse konnte verzichtet werden.

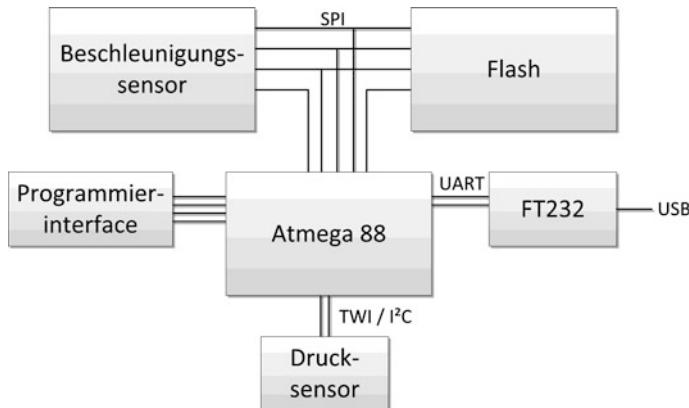


Abb. 9.2 Blockschaltbild des gesamten Projektes

Abb. 9.2 zeigt das Blockschaltbild mit den verschiedenen eingesetzten Bussystemen. Ein ausführlicher Schaltplan ist im Zusatzmaterial des Buches enthalten.

Das Board wurde auf eine Größe von $30\text{ mm} \times 71\text{ mm}$ ausgelegt, damit es in den meisten Raketenspitzen Platz hat. Abb. 9.3 zeigt das Layout der doppelseitigen Platine, die in SMD-Technik von Hand beidseitig bestückt wurde, sowie die fertige Platine mit der Raketen spitze und dem Gehäuse. Die Öffnungen für den USB-Anschluss im Gehäuse und die Bohrung für den Druckausgleich in der Spitze sind gut zu erkennen.

9.3 Beschreibung der Software

Die Software des Boards hat zwei Aufgaben:

- Zunächst soll nach dem Einschalten der Batterieversorgung eine Wartezeit von einer Minute realisiert werden. In dieser Zeit wird die Spitze auf die Rakete aufgesetzt und die Rakete durch einen Zünder gestartet. Anschließend sollen alle 100 ms die Werte für Druck, Temperatur und Beschleunigung ausgelesen und in den zuvor gelöschten Flash geschrieben werden.
- Nach dem Anschluss an die USB-Schnittstelle soll durch Terminaleingabe die Datenübertragung gestartet werden. Die Daten sollen als Semikolon separierte Werte in ASCII ausgegeben und als .csv-Datei auf dem Rechner gespeichert werden.

9.3.1 Der Zustandsautomat

Die gesamte Software ist als Zustandsautomat realisiert. Dabei existieren folgende Zustände:

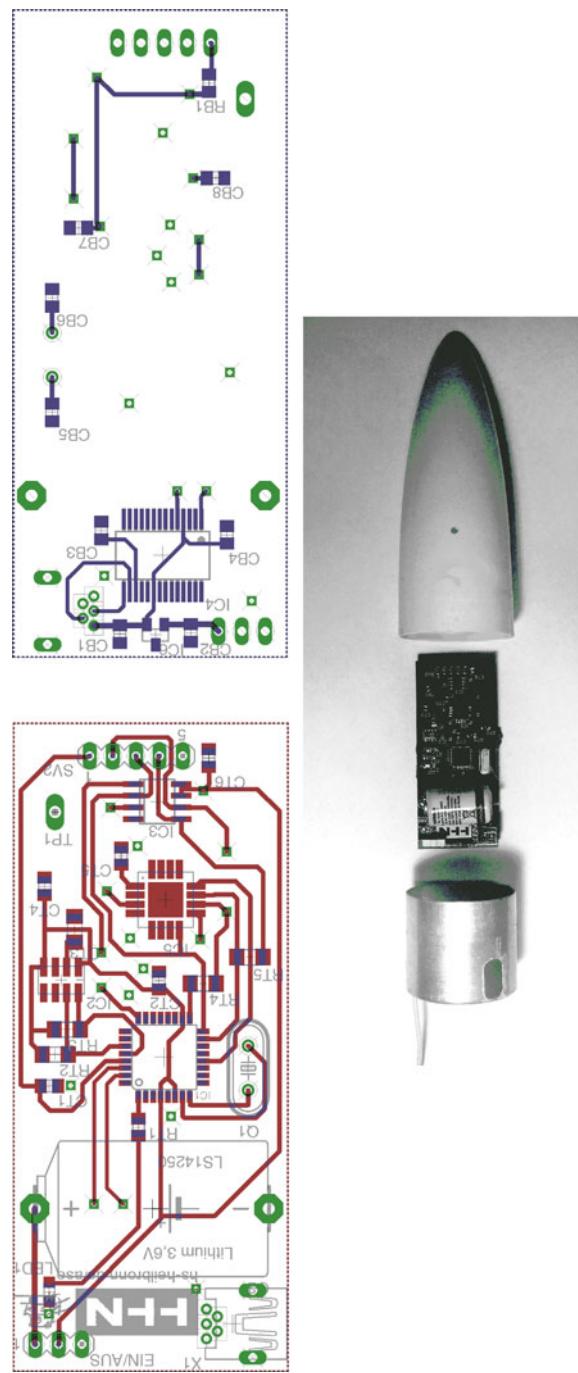
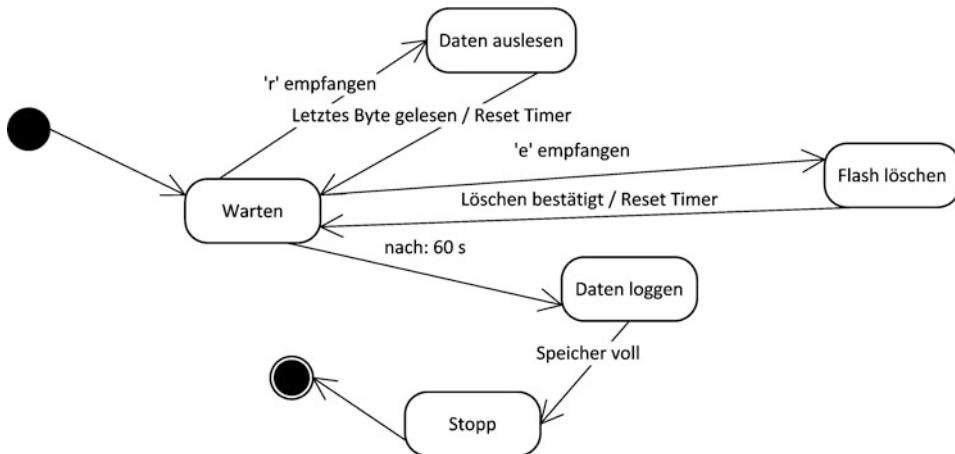


Abb. 9.3 Boardlayout und fertiges Board mit Raketen spitze

**Abb. 9.4** Zustandsautomat des Raketenloggers

- Warten
- Flash löschen
- Daten loggen
- Daten auslesen
- Stop

Der Zustandsautomat wird in einem 100 ms Task geschaltet. Folgende Ereignisse werden ausgewertet:

- 60 s abgelaufen
- Byte zum Flash löschen empfangen
- Byte zum Daten auslesen empfangen
- Flash voll (FLASH_FULL)
- Flash komplett ausgelesen (SEND_COMPLETE)

Damit kann der Automat wie folgt realisiert werden (Abb. 9.4)

9.3.2 Loggen auf dem Flash

Das Loggen auf dem Flash geschieht regelmäßig in einem Task, dessen Frequenz im Timer-Interrupt eingestellt werden kann. Dies können beispielsweise 100 ms sein.

Zunächst werden die Daten aus dem MPL3115 ausgelesen und in den Flashspeicher geschrieben. Druck und Temperatur sind in einem Datensatz von 5 Byte abgelegt, davon sind 3 Byte Druck und 2 Byte Temperatur:

```

MPL3115_Write_BitReg(CTRL1_REG, maske_TD); /*setzen des OST Bits im
Control1 Reg. um den Messwert zu aktualisieren */
MPL3115_Read_DataReg(OUT_P_MSB_REG,drucktempdata,5); /*auslesen der
ersten 5 Bytes (Datenrelevant) des OUT_P_MSB_REG in drucktempdata */
for (int i=0; i<5; i++){
    SST25PFXX_Write_Byte(SST25PFxx_1, OP_CODE_WRITE_BYTE,
        ulAddress_pointer, drucktempdata[i]);
    ulAddress++;
}

```

9.3.3 Daten auslesen

Beim Auslesen der Daten muss ein Zähler die aktuelle Flashadresse global mitzählen. Gleichzeitig werden insgesamt neun Byte aus dem Flash ausgelesen:

- 3 Byte für den Druck
- 2 Byte für die Temperatur
- 2 Byte für die Beschleunigung in x-Richtung
- 2 Byte für die Beschleunigung in y-Richtung

```

CntFF=0;
for (i=0;i<9;i++)
{
    fbuf [i]=SST25PFXX_Read_Byte(    SST25PFxx_1, OP_CODE_READ_BYTE,
                                    ulAddress_pointer );
    if (fbuf [i]==0xFF) CntFF++;
    ulAddress++;
}

```

`ulAddress_pointer` zeigt dabei auf die Adresse im Flash `ulAddress`. Werden im Flash hintereinander mehr als achtmal 0xFF erkannt, so ist dies ein Zeichen, dass der unbeschriebene Flashbereich erreicht wurde und die Übertragung abgebrochen werden kann. Dazu ist die Variable `CntFF` zuständig, die am Ende das Ereignis `SEND_COMPLETE` auslöst.

Das Auslesen geschieht derart, dass die Daten semikolonsepariert als ASCII-Werte über die serielle Schnittstelle übertragen werden. Dadurch kann ein vorhandenes Terminalprogramm direkt eine .csv Datei schreiben, die mit Excel gelesen werden kann.

Der Druck wird zunächst durch 64 geteilt (um 6 Stellen nach rechts geschoben), danach liegt er in 0,01 mBar vor.

```

pres = (long)fbuf[0]<<16;
pres |= (long)fbuf[1]<<8;
pres |= (long)fbuf[2];
pres = pres >> 6;
len = Convert2ASCII(pres,tbuf);
TransmitNumber(tbuf,len);
uartTransmitChar(';');

```

Hier wurde eine eigene Routine `Convert2ASCII(pres,tbuf)` wie in Abschn. 2.9.1 beschrieben realisiert, anstelle von `itoa()`. Gleicher geschieht mit der Temperatur, die in °C * 256 vorliegt sowie den Beschleunigungen, die in 2 m/s^2 vorliegen.

9.4 Auswertung

Die geflogene Höhe ergibt sich aus der bekannten und in jedem Physikbuch nachzuschlagenden barometrischen Höhenformel, die unter der Annahme einer isothermen Atmosphäre hergeleitet wurde. Bei den zu erwartenden Flughöhen ist diese Annahme noch zulässig und erweist sich auch bei der Temperaturmessung als gültig. Mit

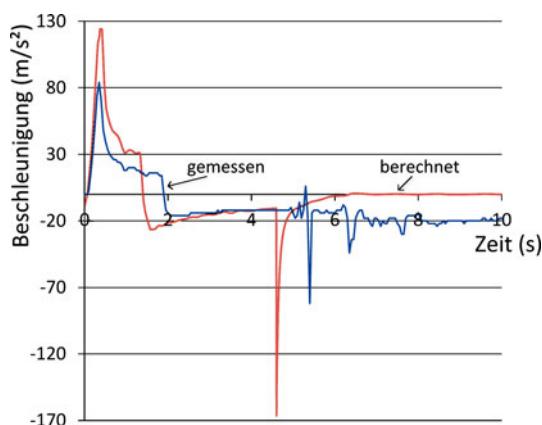
- M als mittlere molare Masse der Atmosphärengase ($0,02896 \text{ kg mol}^{-1}$),
- g als Schwerkraftbeschleunigung ($9,807 \text{ m s}^{-2}$),
- R als die universelle Gaskonstante ($8,314 \text{ J K}^{-1} \text{ mol}^{-1}$) und
- T als die absolute Temperatur

ist die geflogene Höhe $\Delta h = h_1 - h_0$

$$\Delta h = -\frac{R \cdot T}{M \cdot g} \ln \left(\frac{p(h_1)}{p(h_0)} \right) \quad (9.1)$$

Abb. 9.5

Beschleunigungsverlauf in Flugrichtung



Dies lässt sich in Excel aus den gemessenen Werten t (Temperatur in °C) und p bzw. $p(h_0)$ am Boden gut berechnen, wobei für p nur das Verhältnis eine Rolle spielt. Durch Anwendung der barometrischen Höhenformel berechnen sich Flugverläufe wie in Abb. 9.1 gezeigt.

Abb. 9.5 zeigt den Beschleunigungsverlauf in Flugrichtung. Der Schubaufbau des Brennstoffs erreicht nicht die volle berechnete Höhe, dafür findet der Brennschluss im realen Flug etwa 0,5 s nach dem berechneten Zeitpunkt bei 2 s statt. Im nächsten Moment ist die Rakete der Erdbeschleunigung ausgesetzt, wobei sie weiterhin nach oben fliegt. Bei 5 s (gemessen) wird der Fallschirm ausgestoßen, was zu einer negativen Beschleunigung führt und im Höhenverlauf Abb. 9.1 zu einem Druckstoß, der als Höhe natürlich falsch interpretiert ist. Ab hier fällt die Rakete am Fallschirm nach unten, wobei die Beschleunigungswerte wegen der Lage der Raketenspitze nicht mehr plausibel sind.

Literatur

1. Open rocket. <http://openrocket.sourceforge.net/>, Zugegriffen: November 2016

Sachverzeichnis

- #define, 16
#endif, 32
#ifndef, 32
#include, 32
- A**
Abtasttheorem, 220
Nyquist-Shannon, 86
ADC, 88
AD-Wandler, 88, 90
Alarmzeit, 414
Aliasing, 223
ALU, 52
Ambient Light Sensing, 312
AMLCD, 436
Analogkomparator, 88, 89
Analogmultiplexer, 88
Analog-Digitalwandler, 88
Annäherung, sukzessive, 397
Anpassung, 136
Anweisung, 9
Approximation, sukzessive, 90, 397
Arbeitsspeicher, 54, 58
Architektur, 107
Array, 35
ASCII, 37
ASIC, 226
AtmelStudio, 48
AT-Befehl, 212
Ausdruck, 9
Automatic Repeat Request, 151
AVR Libc, 61
- Beschleunigungssensor, 283
Bezeichner, 8
BLE, 205
BlueTooth, 152
Bluetooth, 204
Board Abstraction Layer, 231
Bootloader, 58
Brown-Out-Detection, 55
Brown-Out-Reset, 403
Bus, 140
Bytestuffing, 146
- C**
call-by-value, 28, 45
CAN Controller, 191
CAN Timing, 189
CAN-Frame, 187
Central Processing Unit, 51
CISC, 53
Clock-Streckung, 266
Codemultiplex, 202
Compiler, 48
Controller Area Network, 186
Coordinator, ZigBee, 208
CORDIC, 281
Counter, 73
CPU, 51
CRC, 265
crosstalk, 135
CSI, 160
CSMA, 148
CSMA/CA, 186
- D**
D/A-Wandler, 386
Datenrichtungs-Register, 62

Datenstruktur, dynamisch, 118

Datentyp, 11

Debugger, 49

Definition, von Funktionen, 29

Deklaration, 14

Digitaler Input/Output, 62

Digital-Analog-Wandler, 384

dominant, 138

do-while, 26

Doxygen, 10

DSP, 226

Duplex

Vollduplex, 153

E

Echtzeituhr, 411

EEPROM, 54, 100, 359

Eingangsschnittstellen, analog, 88

Einkopfsystem, 306

else, 22

Endian, 243, 253, 275, 289, 316

Endurance, 359

End-Device, 208

Energie sparen, 89, 98, 100

Ereignis, 122

extern, 33, 110

F

Fehlerbehandlung, 150

Fehlererkennung, 150

Fehlerkorrektur, 151

Fehlerspeicher, 100

Festwertspeicher, 349

 serieller, 350

FFD, 208

FIFO, 116

Filter, 195

FIR-Filter, 225

Flashspeicher

 seriell, 368

Flussdiagramm, 22, 26

for, 27

Forward Error Correction, 151

free, 44, 104

Frequenzmultiplex, 202

Funktionen, 28

Funktionsprototyp, 29

Funkübertragung, 201

Fuse, 56

G

Ganzzahlarithmetik, 232

Ganzzahltypen, 11

Gateway, 208

getter, 110

Gleitkommatypen, 11

goto, 28

Gyroskop, 236

H

.h, 32

Hamming, 226

Handle, 121

Hanning, 226

Hardware-Abstraktion, 109

Harvard-Architektur, 54

Header, 32

Heißleiter, 96

Hinterleuchtung, 436

Historyzustand, 127

I

I2C-Bus, 173

IDE, 6, 48

if, 21

IIR-Filter, 228

Index, 35

Inkrementoperator, 17

Inline-Dokumentation, 9

Integrated Development Environment, 6

Interrupt, 68

 extern, 72

 nested, 71

Interrupt Service Routine, 68

ISM, 201

isochron, 115

J

Jitter, 149

K

Kanalcodierung, 151

Kommentar, 10

Komponentensicht, 109

Konkurrenzphase, 187

Konstanten, 15

Konstantstromquelle, 344

Kontextsicht, 108

- L**
- Latenzzeit, 149
 - LCD, 434
 - Leitung, 135
 - Leitwerk, 51
 - Linker, 48
 - Liste, verkettete, 116, 118
 - Little-Endian, 243
 - Lookup table, 281
 - Lufldrucksensor, 250
 - Luftfeuchtigkeit, 261
 - LUT, 270, 281
- M**
- Magnetfeldsensor, 271
 - main(), 28
 - Makro, 16, 30
 - malloc, 43, 103
 - Maschinencode, 6
 - Master-Slave, 174
 - Master-Slave-Protokoll, 147
 - Mehrfauswahl, 24
 - MEMS, 219
 - Mesh, 140
 - MISO, 161
 - MODBUS, 196
 - MOSI, 161
 - Multipoint, 138
 - Multitasking, 114
- N**
- Näherungssensoren, 305
 - Neigungssensor, 294
 - NTC, 96
 - Nyquist, 140
 - Nyquist-Kriterium, 220
- O**
- OLED, 437
 - Operatoren
 - arithmetisch, 17
 - bitweise, 18
 - logisch, 18
 - Speicher, 19
 - weitere, 20
 - optischer Näherungssensor, 312
 - Output Compare Register, 76
- P**
- Page, 359
- Pairing, 207
- PAM, 221
- Paritätsbit, 153
- PC, 52
- PCI, 69
- Pin Change Interrupt, 69
- Pointer, 43
- Point-to-Point, 138
- Polling, 68, 176, 364
- polling, 245
- Port-Expander, 343
- Potentiometer, digital, 402
- Power Mangement, 98
- Power reduction register, 100
- Prescaler, 73
- Profil, Bluetooth, 207
- Programmzähler, 52
- Protokolle, 132
- Prozess, asynchron, 115
- Pseudozustand, 124
- Puffer, 116
- Pullup, 62–64, 67
- Pulsweitenmodulation, 79
- Q**
- Quantisierung, 223
 - Quantisierungsrauschen, 224
 - Quellcode, 48
- R**
- Radio, 421
 - Real Time Clock, 411
 - Receive-Puffer, 191
 - Rechenwerk, 52
 - Redundanz, 150
 - Referenz, 43
 - Referenzspannung
 - ADC, 92
 - interne, 88
 - Reflexionsfaktor, 136
 - Regelwiderstand, digital, 402
 - RESET, 56
 - Reset, 57
 - rezessiv, 138
 - RFD, 208
 - Ring, 140
 - Ringpuffer, 116
 - RISC, 53
 - Roboterclub Aachen, 190

- Roundtrip-Zeit, 149
Router, 208
RTU, 198
- S**
Sample and Hold, 91, 221
Sättigung, 236
Scheduler, 114
Schleife
 fußgesteuert, 26
 kopfgesteuert, 25
Schlüsselworte, 8
Sequenzdiagramm, 138
Serial Peripheral Interface, 160
Service ID, 41
setter, 110
Skalierung, 236
Sleep, 98
Sleep-Mode, 98
Speicher, 57
 dynamisch, 103
SPI, 160
SPP, 211
Startzustand, 124
state chart, 124
State Machine, 122
Status-Register, 232
Stern, 140
Steuerwerk, 51
Störabstand, 224
String, 37
 nullterminiert, 37
Strommessung, 325
Struktur, 38
Supercap, 411
switch, 24
Symbole, 9
- T**
Task, 112, 114
TDMA, 147
Temperaturmessung, 332
TFT, 436
Thermometer, analog, 96
Thermostat, 337
Tiefpassfilter, 220
- Time Quantum, 189
Timer, 73
Transition, 123
Transmit-Puffer, 191
Trust Center, 208
TWI, 179
Typumwandlung
 explizite, 21
 implizite, 21
- U**
UART, 152, 211
Ultraschall-Näherungssensor, 305
UML, 124
- V**
Variable
 global, 31
 lokal, 31
Versorgung, 55
Verteilungssicht, 108
Verzweigung, 21
„von Neumann“-Architektur, 53
- W**
Warteschlange, 115
Watchdog, 57
Wellenwiderstand, 136
while, 25
wired-AND, 138
wired-OR, 138
wrapping, 109
- Z**
Zählschleife, 27
Zeichenkette, 37
Zeiger, 43
 auf Funktionen, 46
Zeitmultiplex, 202
ZigBee, 207
Zustandsautomat, endlich, 122
Zustandsdiagramm, 124
Zustandsübergangstabelle, 124
Zuweisungsoperator, 17
Zweikopfsystem, 306
Zyklus, 119