



Ansgar Meroth  
Petre Sora

# Sensornetzwerke in Theorie und Praxis

Embedded Systems-Projekte erfolgreich  
realisieren

*2. Auflage*



Springer Vieweg

---

# Sensornetzwerke in Theorie und Praxis

---

Ansgar Meroth · Petre Sora

# Sensornetzwerke in Theorie und Praxis

Embedded Systems-Projekte erfolgreich  
realisieren

2. Auflage



Springer Vieweg

Ansgar Meroth  
Fakultät für Mechanik und Elektronik  
Hochschule Heilbronn  
Heilbronn, Deutschland

Petre Sora  
Fakultät für Mechanik und Elektronik  
Hochschule Heilbronn  
Heilbronn, Deutschland

ISBN 978-3-658-31708-9

ISBN 978-3-658-31709-6 (eBook)

<https://doi.org/10.1007/978-3-658-31709-6>

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

© Springer Fachmedien Wiesbaden GmbH, ein Teil von Springer Nature 2018, 2021, korrigierte Publikation 2022  
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 allgemein beschreibenden Bezeichnungen, Marken, Unternehmensnamen etc. in diesem Werk bedeutet nicht, dass diese frei durch jedermann benutzt werden dürfen. Die Berechtigung zur Benutzung unterliegt, auch ohne gesonderten Hinweis hierzu, den Regeln des Markenrechts. Die Rechte des jeweiligen Zeicheninhabers sind zu beachten.

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.

Planung/Lektorat: Reinhard Dapper

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 zur zweiten Auflage

Die Entwicklung des IoT schreitet dynamisch voran und lässt Inhalte sehr schnell altern. Zudem haben wir einige wertvolle Hinweise für die Weiterentwicklung des Buches erhalten. Auch die Zeit bis zur Drucklegung der ersten Auflage war begrenzt und viele Ideen konnten nicht mehr umgesetzt werden. Daher kommen wir dem Wunsch nach einer zweiten, erheblich erweiterten und überarbeiteten Auflage gerne nach. In diesem Zug wurde das gesamte Buch neu strukturiert, etliche Bausteine und auch einige Algorithmen sind dazugekommen. Dem Wunsch der Leser folgend haben wir Hinweise zur Funktionsweise der Bausteine ergänzt. Zur Drucklegung sind alle Bausteine im Handel erhältlich.

Wir haben feststellen müssen, dass Leser mit dem Einführungskapitel zur Programmiersprache C überfordert sind. Dieses Buch ersetzt kein Anfängerbuch in der Programmierung. Vielmehr ist eine gute Kenntnis in C Voraussetzung für das Verständnis, Kapitel 2 dient nur dem schnellen Nachschauen. Wir verweisen auf die Einführung, in der weiterhin ausgeführt ist, welche Voraussetzungen zum Verständnis notwendig sind und was von der Lektüre dieses Buchs zu erwarten ist.

Heilbronn  
im April 2021

Ansgar Meroth  
Petre Sora

---

## Vorwort

Das „Internet of things“ ist in aller Munde. Eine wichtige Voraussetzung für die digitale Revolution im Alltag ist die Verfügbarkeit kleiner 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 November 2017

Ansgar Meroth  
Petre Sora

---

Die Originalversion des Buchs wurde revidiert. Ein Erratum ist verfügbar unter  
[https://doi.org/10.1007/978-3-658-31709-6\\_27](https://doi.org/10.1007/978-3-658-31709-6_27)

---

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	1
1.1	Was finden Sie in diesem Buch?	1
1.2	Für wen ist dieses Buch geschrieben?	3
1.3	Welche Kenntnisse setzt das Buch voraus?	4
1.4	Warum beschreibt das Buch keinen Arduino?	4
1.5	Zusatzmaterialien	4
1.6	Disclaimer/Haftungsausschluss	4
1.7	Dank	5
	Literatur	5
<b>2</b>	<b>Einführung in die Programmiersprache C</b>	7
2.1	Die Sprache C: Hintergrund und Aufbau	7
2.2	Bezeichner, Schlüsselworte und Symbole in C	10
2.2.1	Bezeichner	10
2.2.2	Schlüsselworte	10
2.2.3	Symbole	11
2.2.4	Anweisungen	11
2.3	Kommentare	12
2.3.1	Einzeiliger Kommentar	12
2.3.2	Mehrzeiliger Kommentar	12
2.3.3	Dxygen-Kommentare	12
2.4	Typen, Variablen und Konstanten	13
2.4.1	Fundamentale Datentypen	15
2.4.2	Deklaration von Variablen	17
2.4.3	Konstanten	18
2.5	Operatoren	20
2.5.1	Arithmetische Operatoren	20
2.5.2	Logische Operatoren	21
2.5.3	Bitoperatoren	21
2.5.4	Operatoren für Speicherzugriffe	21

2.5.5	Weitere Operatoren .....	22
2.5.6	Assoziativitt und Prioritt von Operatoren.....	22
2.5.7	Typumwandlung.....	23
2.6	Kontrollstrukturen .....	25
2.6.1	Verzweigung (Auswahl).....	25
2.6.2	Fallstricke.....	27
2.6.3	Mehrfachverzweigung .....	27
2.7	Schleifen.....	29
2.7.1	Kopfgesteuerte Schleifen.....	29
2.7.2	Fußgesteuerte Schleifen.....	30
2.7.3	Zahlschleifen .....	30
2.7.4	Sprnge .....	31
2.8	Funktionen .....	32
2.8.1	int main() .....	32
2.8.2	Definition und Deklaration.....	33
2.8.3	Sichtbarkeit und Lebensdauer von Variablen in Funktionen ..	35
2.8.4	Header .....	36
2.8.5	Die Schlsselworte extern, volatile und static .....	37
2.9	Komplexe Datentypen .....	40
2.9.1	Arrays, Felder und Zeichenketten .....	40
2.9.2	Struktur.....	44
2.9.3	Unions .....	47
2.9.4	Aufzhlungstypen .....	49
2.9.5	Pointer .....	50
2.10	Aufbau eines Embedded-C-Programms.....	54
2.11	Arbeiten mit dem Procompiler.....	56
2.11.1	#define und Arbeiten mit Makros.....	56
2.11.2	#pragma .....	58
2.11.3	Zusammenfassung der Procompiler-Befehle .....	59
2.12	Ubersetzen und Binden .....	59
	Literatur.....	61
3	<b>Programmierung von AVR Mikrocontrollern .....</b>	63
3.1	Architektur der AVR-Familie .....	64
3.2	Gehuse und Anschlussbelegungen .....	67
3.3	Versorgung, Takt und Reset-Logik .....	67
3.3.1	Versorgung .....	67
3.3.2	Takt .....	67
3.3.3	Reset-Logik .....	70
3.3.4	Speicher .....	70
3.4	Umgang mit Registern .....	72
3.5	Digitaler Input/Output .....	75

3.5.1	Grundsätzlicher Aufbau . . . . .	75
3.5.2	Programmierung. . . . .	78
3.6	Interrupts . . . . .	81
3.6.1	Einstieg in den Umgang mit Interrupts . . . . .	82
3.6.2	Interrupt-Programmierung am Beispiel des Pin Change Interrupt . . . . .	82
3.6.3	Die externen Interrupts INTx. . . . .	86
3.7	Timer . . . . .	87
3.7.1	Timer-Grundlagen . . . . .	87
3.7.2	Programmierung der Timer/Counter . . . . .	89
3.8	Analoge Schnittstelle . . . . .	108
3.8.1	Analogmultiplexer . . . . .	108
3.8.2	Analogkomparator . . . . .	109
3.8.3	AD-Wandler (ADC). . . . .	110
3.8.4	Beispiel: Thermometer. . . . .	117
3.8.5	Beispiel: Effektivwertmessung an einer Sinusspannung . . . . .	119
3.9	Power Management . . . . .	120
3.10	Internes EEPROM . . . . .	123
3.10.1	Deklaration einer Variablen im EEPROM . . . . .	123
3.10.2	Lesen aus dem EEPROM . . . . .	123
3.10.3	Schreiben ins EEPROM . . . . .	124
3.11	Dynamische Speichernutzung . . . . .	125
3.12	Verlagerung von Daten in den Programmspeicher. . . . .	127
Literatur . . . . .		128
<b>4</b>	<b>Software Framework</b> . . . . .	129
4.1	Sichten . . . . .	130
4.2	Hardware-Abstraktion . . . . .	131
4.3	Modularisierung und Zugriff auf Module . . . . .	132
4.4	Zeitsteuerung . . . . .	134
Literatur . . . . .		138
<b>5</b>	<b>Speicherkonzepte und Algorithmen</b> . . . . .	139
5.1	Wichtige Speicherkonzepte . . . . .	139
5.1.1	Warteschlangen und Ringpuffer (FIFO). . . . .	140
5.1.2	Warteschlange mit dynamischen Datenstrukturen . . . . .	143
5.1.3	Mehrere Warteschlangen in einem Programm . . . . .	145
5.1.4	Mehrere Warteschlangen mit unterschiedlichen Typen . . . . .	147
5.2	Zustandsautomaten (Statemachine) . . . . .	149
5.2.1	Allgemeine Betrachtung. . . . .	149
5.2.2	Beschreibung von Zustandsautomaten. . . . .	151
5.2.3	Umsetzung von Zustandsautomaten auf Mikrocontrollern . . . . .	154
Literatur . . . . .		157

<b>6</b>	<b>Theoretische Überlegungen zu IoT Netzwerken</b>	159
6.1	Das ISO/OSI Schichtenmodell.	160
6.1.1	Schicht 1: Physikalische Bitübertragung	162
6.1.2	Schicht 2: Sicherungsschicht	173
6.1.3	Schicht 3: Vermittlungsschicht	183
6.1.4	Schicht 4: Transportschicht	186
6.1.5	Anwendungsprotokolle für IoT Netzwerke	188
6.2	Anforderungen an IoT Netzwerke	190
Literatur		193
<b>7</b>	<b>Asynchrone serielle Schnittstellen</b>	195
7.1	Universal Asynchronous Receiver/Transmitter (UART)	195
7.1.1	Hardwareanbindung in der AVR-Familie	197
7.1.2	UART-Register beim ATmega 88	198
7.1.3	Initialisieren der UART Schnittstelle beim ATmega88	200
7.1.4	Empfangen von Daten	201
7.1.5	Senden von Daten	203
7.1.6	Implementierung von uartWriteBuffer()	203
7.1.7	UART-Multiprozessor-Modus	204
7.2	Anbindung der seriellen Schnittstelle an USB	211
7.3	Ein einfaches serielles Protokoll	211
7.3.1	Herstellung von Codetransparenz durch Bytestuffing	216
Literatur		218
<b>8</b>	<b>Serial Peripheral Interface (SPI)</b>	219
8.1	Aufbau und Funktionsweise	220
8.2	Konfiguration der SPI-Schnittstelle	221
8.3	SPI-Schnittstelle im Slavebetrieb	225
8.4	SPI-Schnittstelle in einem Sensornetzwerk	228
8.5	3-wire-SPI-Kommunikation	233
8.6	SPI-Master über USART	235
Literatur		239
<b>9</b>	<b>Die I<sup>2</sup>C/TWI-Schnittstelle</b>	241
9.1	I <sup>2</sup> C-Bus-Konfiguration	243
9.2	Bus-Erweiterung	245
9.2.1	I <sup>2</sup> C-Repeater	246
9.2.2	I <sup>2</sup> C-Hub	246
9.2.3	I <sup>2</sup> C-Multiplexer	246
9.2.4	I <sup>2</sup> C-Switch	247
9.3	TWI in der AVR-Familie	247
9.3.1	TWI-Register beim ATmega 88	248
9.3.2	Initialisieren der TWI-Schnittstelle	249

9.3.3	TWI-Kommunikation . . . . .	251
9.3.4	Der Mikrocontroller ATmega als TWI-Master . . . . .	251
9.3.5	Der Mikrocontroller ATmega als TWI-Slave . . . . .	257
	Literatur . . . . .	261
<b>10</b>	<b>CAN Bus . . . . .</b>	<b>263</b>
10.1	CAN-Grundlagen kompakt . . . . .	264
10.2	CAN-Timing . . . . .	266
10.3	Nutzung von CAN mit Prozessoren der AVR-Familie . . . . .	267
10.3.1	CAN-Controller MCP2515 . . . . .	268
10.3.2	AT90CANxx . . . . .	274
10.3.3	Implementierung mit der CAN-Bibliothek des Roboterclub Aachen . . . . .	278
10.4	CAN-Transportprotokoll . . . . .	281
10.5	CANopen in der Industriesteuerungstechnik . . . . .	284
	Literatur . . . . .	285
<b>11</b>	<b>Der Modbus . . . . .</b>	<b>287</b>
11.1	TIA/EIA-485 als Bitübertragungsschicht für MODBUS . . . . .	288
11.2	MODBUS-Kommunikation . . . . .	290
11.2.1	Remote-Terminal-Unit-Übertragung . . . . .	290
11.2.2	ASCII-Übertragung . . . . .	293
	Literatur . . . . .	296
<b>12</b>	<b>Eindrahtbussysteme . . . . .</b>	<b>297</b>
12.1	1- Wire <sup>®</sup> -BUS . . . . .	298
12.1.1	Netzwerktopologie . . . . .	298
12.1.2	Initialisierung des Busses . . . . .	300
12.1.3	1-wire-Bitübertragung . . . . .	301
12.1.4	Kommunikationsitzung . . . . .	303
12.1.5	Softwareaufbau der 1-wire-Buskommunikation . . . . .	307
12.1.6	Ansteuerung eines 1-wire-Temperatursensors vom Typ DS18B20 . . . . .	309
12.2	UNI/O <sup>®</sup> -Bus . . . . .	312
12.2.1	Netzwerktopologie . . . . .	312
12.2.2	Bitcodierung . . . . .	313
12.2.3	UNI/O-Pulsrahmen . . . . .	313
12.2.4	Kommunikationsitzung . . . . .	313
12.2.5	Softwareaufbau einer UNI/O-Buskommunikation . . . . .	315
12.2.6	Ansteuerung eines 11XXYYZ-EEPROM . . . . .	319
12.3	LIN-Bus . . . . .	326
	Literatur . . . . .	331

<b>13 Drahtlose Netzwerke</b>	333
13.1 Grundlagen der Funkschnittstellen	334
13.1.1 Multiplexverfahren	334
13.1.2 Sensorknoten	335
13.2 Funkübertragung im 433 MHz und 868 MHz ISM-Band	337
13.2.1 Aufbau des RFM12B	337
13.2.2 Beschaltung des RFM12-Funkmoduls	338
13.2.3 Die SPI-Kommunikation	339
13.2.4 Der Befehlssatz	340
13.2.5 Das Statusregister	345
13.2.6 Initialisierung des Transceivers	346
13.2.7 Daten senden	348
13.2.8 Lesen der empfangenen Daten	350
13.3 Funkprotokolle im 2,4-GHz-ISM-Band	351
13.3.1 Bluetooth	351
13.3.2 ZigBee	357
13.4 Bluetooth® Kommunikation mit dem serial profile	361
13.4.1 Betriebsmodi	362
13.4.2 Befehlssatz	362
13.4.3 Initialisierung des Funkmoduls	363
Literatur	365
<b>14 Sensortechnik Systemtechnische Überlegungen</b>	367
14.1 Abtastung	368
14.2 Quantisierung	371
14.3 Digitale Filterung	373
14.3.1 Finite-Impulse-Response-Filter (FIR)	373
14.3.2 Infinite-Impulse-Response-Filter (IIR)	376
14.3.3 Filterung am Beispiel eines FIR-Filters	377
14.4 I/O-Steuerlogik	378
14.5 Abstraktion der I/O-Pins	379
14.6 Ganz Zahlarithmetik	380
14.6.1 Mikrocontrollerinterne Zahlenformate	380
14.6.2 Vorzeichenlose Ganzzahltypen	382
14.6.3 Vorzeichenbehaftete Ganzzahltypen	383
14.6.4 Erkennung und Verhinderung eines Überlaufs	384
Literatur	385
<b>15 Umweltsensoren</b>	387
15.1 MPL3115 digitaler Luftdrucksensor	388
15.1.1 Funktionsweise	388
15.1.2 Aufbau des MPL3115	390
15.1.3 Serielle Kommunikation	397

15.1.4	Power-Modi . . . . .	397
15.1.5	Mess- und Lesemodi . . . . .	398
15.1.6	Initialisierung des MPL3115-Sensors . . . . .	399
15.2	Luftfeuchtigkeit SI7021 . . . . .	399
15.2.1	Aufbau des SI7021 . . . . .	400
15.2.2	Serielle Kommunikation . . . . .	402
15.2.3	Berechnung der Temperatur und der relativen Luftfeuchtigkeit . . . . .	406
15.2.4	Testbarkeit . . . . .	408
15.3	Temperaturmessung mit dem TMP75 . . . . .	409
15.3.1	Sensorkonfigurierung . . . . .	410
15.3.2	Serielle Schnittstelle . . . . .	411
15.3.3	Temperaturmessung . . . . .	413
15.3.4	Thermostatkktion . . . . .	414
15.4	Feinstaubsensor SDS011 . . . . .	418
15.4.1	Messprinzip . . . . .	419
15.4.2	Ansteuerung und serielle Kommunikation . . . . .	420
Literatur . . . . .		424
<b>16</b>	<b>Beschleunigungssensoren . . . . .</b>	<b>427</b>
16.1	Beschleunigungssensor ADXL312 . . . . .	428
16.1.1	Vernetzung des ADXL312 . . . . .	429
16.1.2	Messdatenerfassung . . . . .	431
16.1.3	Offset-Ermittlung . . . . .	437
16.1.4	Interruptmodus . . . . .	437
16.1.5	ADXL312 als Neigungssensor . . . . .	439
16.2	MMA6525 . . . . .	443
16.2.1	Sensoraufbau . . . . .	443
16.2.2	Registerblock . . . . .	444
16.2.3	SPI-Kommunikation . . . . .	445
Literatur . . . . .		453
<b>17</b>	<b>Drehratensensoren . . . . .</b>	<b>455</b>
17.1	Gyroskop . . . . .	456
17.1.1	Beschaltung des L3GD20 . . . . .	457
17.1.2	Kommunikationsschnittstellen . . . . .	457
17.1.3	Arbeitsmodi . . . . .	461
Literatur . . . . .		471
<b>18</b>	<b>Magnetfeldsensoren . . . . .</b>	<b>473</b>
18.1	HMC5883-Magnetfeldsensor . . . . .	475
18.1.1	Aufbau des HMC5883 . . . . .	475
18.1.2	HMC5883 Messwerte lesen . . . . .	481
18.1.3	Kalibrierung des Sensors . . . . .	482

18.1.4	HMC5883 als elektronischer Kompass . . . . .	484
18.1.5	Winkelberechnung mit dem CORDIC-Algorithmus . . . . .	485
Literatur . . . . .		487
<b>19</b>	<b>Näherungssensoren . . . . .</b>	<b>489</b>
19.1	Ultraschall-Näherungssensoren . . . . .	489
19.1.1	Messprinzip . . . . .	490
19.1.2	SRF08 – Ultraschall-Messmodul . . . . .	492
19.2	SI114x – optischer Näherungssensor . . . . .	497
19.2.1	SI114x – Arbeitsmodi . . . . .	497
19.2.2	SI114x – Aufbau . . . . .	500
19.2.3	Serielle Kommunikation . . . . .	504
19.2.4	Messungen mit dem SI114x . . . . .	505
19.2.5	Interrupts . . . . .	510
19.2.6	SI114x – Netzwerkidentifikation . . . . .	511
Literatur . . . . .		511
<b>20</b>	<b>Digital-Analog- und Analog-Digital-Wandler . . . . .</b>	<b>513</b>
20.1	MCP48XX SPI-angesteuerte Digital-Analog-Wandler . . . . .	513
20.1.1	Die SPI-Schnittstelle . . . . .	515
20.1.2	Das Eingangsregister . . . . .	515
20.1.3	Der D/A-Wandler . . . . .	516
20.1.4	Der analoge Ausgangsverstärker . . . . .	516
20.1.5	Synchrone Ansteuerung zweier D/A-Wandler . . . . .	517
20.1.6	Softwarebeispiel . . . . .	520
20.2	PCF8591 I <sup>2</sup> C-angesteuerter D/A- und A/D-Wandler . . . . .	522
20.2.1	I <sup>2</sup> C-Kommunikation . . . . .	523
20.2.2	Der D/A-Wandler . . . . .	524
20.2.3	Der A/D-Wandler . . . . .	526
20.2.4	Das Control-Register . . . . .	529
20.2.5	Der Oszillator . . . . .	529
20.3	Strommessung mit dem LMP92064 . . . . .	530
20.3.1	LMP92064 Aufbau . . . . .	531
20.3.2	Serielle Kommunikation . . . . .	532
20.3.3	Messen mit dem LMP92064 . . . . .	533
Literatur . . . . .		536
<b>21</b>	<b>Serielle EEPROMs . . . . .</b>	<b>539</b>
21.1	Parallele Festwertspeicher . . . . .	541
21.2	Serielle EEPROM-Speicher . . . . .	542
21.2.1	M24C64 – I <sup>2</sup> C-angesteuerter EEPROM . . . . .	542
21.2.2	25LC256 – SPI-angesteuerte EEPROMs . . . . .	551
Literatur . . . . .		559

<b>22</b>	<b>Serielle Flash-Speicher</b>	561
22.1	AT45DB161 serieller Flashspeicher	564
22.1.1	SPI-Kommunikation.	565
22.1.2	SRAM-Zwischenspeicher	566
22.1.3	Flash-Hauptspeicher.	567
22.1.4	Lesen	568
22.1.5	Speichern	570
22.1.6	Löschen	573
22.1.7	Speicherschutz	573
22.1.8	Testbarkeit	574
22.2	SST25WF0808 serieller Flashspeicher	575
22.2.1	SPI-Kommunikation.	576
22.2.2	Statusregister	577
22.2.3	Lese-Funktionen.	578
22.2.4	Lösch-Funktionen	579
22.2.5	Schreib-Funktionen	580
22.2.6	2-Leitung-serielle-Schnittstelle	581
	Literatur.	581
<b>23</b>	<b>Bausteine für die Audiotechnik</b>	583
23.1	SI4840 Radio-IC	584
23.1.1	Bausteinbeschreibung	585
23.1.2	Auswahl des Frequenzbandes und Frequenzabstimmung	586
23.1.3	Initialisieren des Bausteins.	588
23.1.4	Kommunikation mit dem Baustein	588
23.1.5	Sendersuche mit dem SI4840	593
23.2	LM48100Q Verstärker-Baustein	597
	Literatur.	602
<b>24</b>	<b>Vernetzbare integrierte Schaltkreise</b>	603
24.1	PCF8574 – Port-Expander	603
24.1.1	Endstufe eines I/O-Pins	604
24.1.2	Ausgang-Port-Modus	606
24.1.3	Eingang-Port-Modus	607
24.1.4	Interrupt-Modus	608
24.1.5	PCA9534	608
24.2	MCP41X1 digitale Regelwiderstände	609
24.2.1	Power-On-/Brown-Out-Reset-Schaltung	610
24.2.2	Elektrischer Widerstand	611
24.2.3	Potentiometer-Register	611
24.2.4	Ansteuerfunktionen des Bausteins MCP4151	614
24.2.5	SPI-Kommunikation.	614
24.2.6	Softwarebeispiel	616

24.3	MAX31629 – Real Time Clock (RTC) . . . . .	618
24.3.1	Zeitmessung . . . . .	618
24.3.2	Alarmzeit . . . . .	621
24.3.3	Temperaturmessung . . . . .	622
24.3.4	Thermostat mit Alarmfunktion. . . . .	623
24.3.5	I <sup>2</sup> C-Kommunikation. . . . .	625
	Literatur. . . . .	627
<b>25</b>	<b>Anzeigen. . . . .</b>	<b>629</b>
25.1	Einführung . . . . .	630
25.1.1	Displaylayout . . . . .	630
25.1.2	Emissive und nicht emissive Anzeigen . . . . .	630
25.1.3	Bildaufbau . . . . .	634
25.1.4	Display Ansteuerung . . . . .	635
25.2	Punktmatrix-LCD-Display mit paralleler Ansteuerung . . . . .	635
25.2.1	Struktur eines Displays mit einem KS0070B-Controller. . . . .	635
25.2.2	Befehlssatz . . . . .	637
25.2.3	4-Bit-Kommunikation . . . . .	639
25.2.4	Generierung eines neuen Zeichens. . . . .	640
25.2.5	Ausführen der Display-Befehle ohne blockierendes Warten . . . . .	642
25.3	Serielle Ansteuerung eines parallelen LC-Displays. . . . .	645
25.3.1	Display-Ansteuerung über I <sup>2</sup> C . . . . .	646
25.3.2	Software-Beispiel: Übertragung eines Datenbytes . . . . .	646
25.4	DOGS102-6 – Graphisches Display mit serieller Ansteuerung . . . . .	649
25.4.1	Struktur des graphischen Displays DOGS 102–6 . . . . .	650
25.4.2	SPI-Kommunikation. . . . .	651
25.4.3	Befehlssatz . . . . .	653
25.4.4	Generierung eines Zeichens. . . . .	657
	Literatur. . . . .	659
<b>26</b>	<b>Beispielprojekte. . . . .</b>	<b>661</b>
26.1	Datenlogger . . . . .	661
26.1.1	Aufbau der Modellrakete . . . . .	661
26.1.2	Beschreibung des Projekts . . . . .	663
26.1.3	Beschreibung der Software . . . . .	665
26.1.4	Auswertung . . . . .	667
26.2	Smart Home mit CAN . . . . .	668
26.2.1	Aufbau . . . . .	669
	Literatur. . . . .	673
	<b>Erratum zu: Sensornetzwerke in Theorie und Praxis. . . . .</b>	<b>E1</b>
	<b>Stichwortverzeichnis. . . . .</b>	<b>675</b>



# Einleitung

1

## Zusammenfassung

Eine kurze Übersicht, was in diesem Buch zu finden und wie es zu benutzen ist.

### 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 wird<sup>1</sup>, 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

<sup>1</sup> So auch in der Arduino® -Familie.

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 zunächst einmal C üben und verstehen
- 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. Die Firma Atmel hat ein hervorragendes, instruktives Handbuch herausgegeben, das nach der Übernahme durch Microchip weiter gepflegt wird, außerdem beschreiben zahlreiche Autoren diese Familie, speziell auf der Seite [www.mikrocontroller.net](http://www.mikrocontroller.net) finden sich ausführliche Anleitungen und Antworten, mit denen die Leser das hier geschriebene vertiefen können
- Eine Beschreibung wichtiger Softwaremechanismen, die in einem embedded Programm – unabhängig vom verwendeten Prozessor – eingesetzt werden können. Diese sind hardwareunabhängig beschrieben.
- Kapitel über Bussysteme und Netzwerke, in denen sowohl die Kommunikations-schnittstellen UART, SPI und I<sup>2</sup>C (TWI<sup>2</sup>) des Prozessors als auch, abstrakter, die Kommunikation über ausgewählte Bussysteme (CAN, Modbus, LIN, OneWire, PSI5) und Funkschnittstellen (ISM Bänder, Bluetooth, ZigBee, LoRa) beschrieben sind. Hiermit lässt sich eine Vielzahl von Sensornetzwerken aufbauen.

Der Kern des Buches besteht aus verschiedenen Kapiteln, die ausgewählte Sensoren, Anzeigen und weitere vernetzbare Bausteine beschreiben, beispielsweise Flash-Daten-speicher, 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. Tab. 1.1 zeigt den grundsätzlichen Aufbau des Buchs für das schnelle Auffinden der Einzelkapitel.

---

<sup>2</sup>Im Rahmen der AVR Familie wird der Begriff Two-wire-interface (TWI) verwendet. Im beschriebenen Zusammenhang ist TWI mit I<sup>2</sup>C gleichzusetzen.

**Tab. 1.1** Aufbau des Buchs

Teil	Detail
1: Einleitung und Einführung in C	1 Einleitung
	2 Kurzeinführung in C
2: Programmieren in der AVR Familie	3 Struktur und Programmierung von AVR Controllern
3: Hardwareunabhängige Mechanismen	4 Software Framework
	5 Speicherkonzepte und Algorithmen
4: Kommunikation	6 Kommunikation – Theorie
4.1: Mikrocontroller Schnittstellen	7 Asynchrone Serielle Schnittstellen
	8 Die SPI Schnittstelle
	9 Die I2C Schnittstelle
4.2: Bussysteme	10 Der CAN Bus
	11 Der Modbus
	12 Eindrahtbussysteme
	13 Drahtlose Netzwerke
5: Sensoren	14 Sensorik – Systemtechnische Überlegungen
	15 Umweltsensoren
	16 Beschleunigungssensoren
	17 Drehratensensoren
	18 Magnetfeldsensoren
	19 Näherungssensoren
6: Vernetzbare Bausteine in Sensornetzwerken	20 DA und AD Wandler
	21 Serielle EEPROMS
	22 Serielle Flash-Speicher
	23 Bausteine für die Audiotechnik
	24 Weitere vernetzbare Bausteine
7: Weitere Überlegungen	25 Anzeigen
	26 Beispielprojekte

---

## 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, I<sup>2</sup>C (TWI) und der Port-Pins an [3]. Da dieses Buch jedoch auch zum Verständnis der rudimentären Vorgänge in einem Mikrocontroller beitragen soll, würde es keinen Sinn machen, diese durch die Arduino-Bibliothek zu verbergen.

---

### **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 unter der Detailbeschreibung des Buches auch 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 potenziell 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.

Dieses Buch wurde von zwei Personen geschrieben und von verschiedenen anderen quergelesen. Dennoch können sich bei der Komplexität der Materie Fehler einschleichen, die niemand bemerkt. Wir bitten, diese den Autoren zu melden, damit sie in der nächsten Auflage behoben werden können. Ein Erratum nach Drucklegung wird auf der Webseite des Verlags zur Verfügung gestellt.

---

## 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, sowie unserem geschätzten Kollegen em. Prof. Dr.-Ing. Wolfgang Wehl, 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. (Hrsg.). (2018). *Sensoren in Wissenschaft und Technik* (2. Aufl.). Vieweg & Teubner.
2. Küveler, G., & Schwoch, D. (2006). *Informatik für Ingenieure und Naturwissenschaftler I – Grundlagen Programmierung mit C/C++ – Großes C/C++ Praktikum* (5. vollständig überarbeitete und aktualisierte Aufl.). Vieweg.
3. Arduino: Sprachreferenz. <https://www.arduino.cc/en/Reference/HomePage>. Zugegriffen: 19. Aug. 2020.



# Einführung in die Programmiersprache C

2

## Zusammenfassung

In diesem Kapitel wird ein kurzer Überblick über die Programmiersprache C gegeben, soweit es zum Verständnis des weiteren Buchs und insbesondere der Programmierbeispiele notwendig ist.

Dieses Kapitel ist bewusst stichwortartig gehalten und ersetzt für Programmieranfänger nicht das Studium eines C-Kurses. Vielmehr dient es 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 2018 (C18) weiterentwickelt wurde [2]. Bjarne Stroustrup entwickelte C zu einer objektorientierten Programmiersprache C++ weiter, die seit 1998 in einer ISO-Norm festgeschrieben wurde (ISO/IEC 14.882:2014) [2]. 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  $\times 86$  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 sogenannte Kontrollstrukturen (Verzweigungen, Schleifen) verwendet, absolute Sprunganweisungen (`goto` Abschn. 2.7.4) sind zwar in C grundsätzlich erlaubt, sollten aber nach Möglichkeit vermieden werden. Der strukturierte/prozedurale Programmierstil (Paradigma) der Sprache C 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:

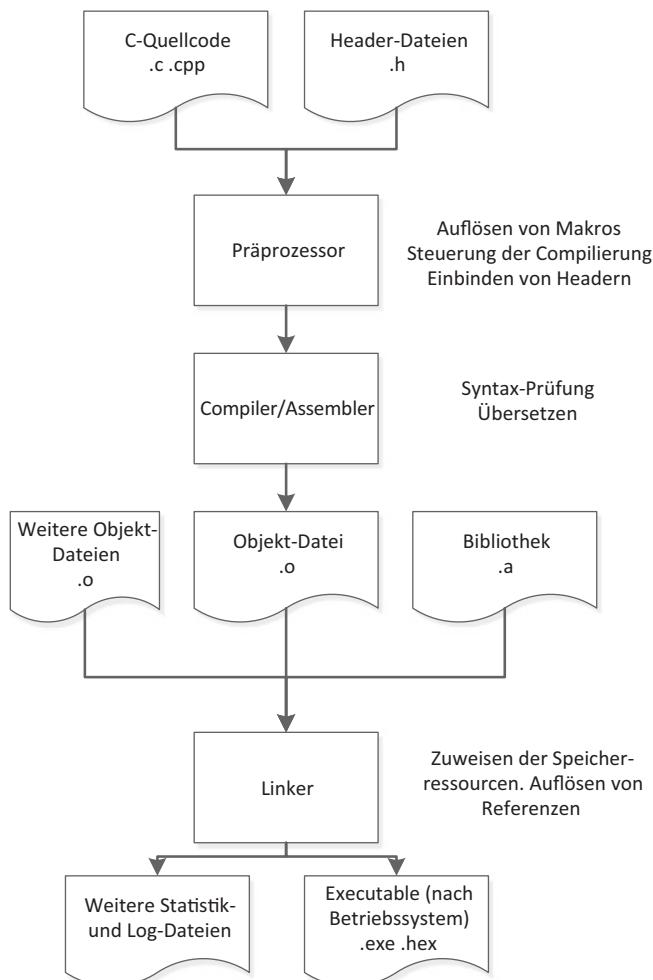
<code>ldd</code>	<code>r24, Y+1</code>	<code>; Lade aus dem Speicher das niedrige Byte von c</code>
<code>ldd</code>	<code>r25, Y+2</code>	<code>; Lade aus dem Speicher das höhere Byte von c</code>
<code>adiw</code>	<code>r24, 0x07</code>	<code>; Addiere zum niedrigeren Byte von c die Zahl 7</code>

ldd	r24, Y+1	; Lade aus dem Speicher das niedrige Byte von c
std	Y+6, r25	; Speichere Ergebnis (hohes Byte) an die Speicherstelle von a
std	Y+5, r24	; Speichere Ergebnis (niedriges Byte) an die Speicherstelle 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 [4]) oder Bücher [5–10] 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 Klein-schreibung unterscheiden, also „case-sensitive“ sind. Der Bezeichner „Wert“ ist also von „WERT“ und „wert“ verschieden und bezeichnet eine andere Variable oder Funktion.

<b>Gültige Bezeichner:</b>	umu_foo	Nummer_5	n_nummer
<b>Ungültiger Bezeichner:</b>	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. 2.9ff)
5. Weitere Schlüsselworte: `void` (Abschn. 2.8), `typedef` (Abschn. 2.9.2), `sizeof` (Abschn. 2.9)

### 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 - 1.10.2020 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  
*****
```

Doxygen erzeugt aus diesen Kommentaren und aus der Analyse des Codes sehr gut lesbare Dateien in Rich Text Format (RTF – Word), HTML, LaTex, XML, UNIX manpages etc. 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 folgende Abb. 2.2 zeigt, wie Doxygen aus den oben gezeigten Kommentarbeispielen eine Rich-text-Datei (MS Word) generiert.

Mit Doxygen, das beim Entwickler erhältlich ist [11] und der ebenfalls kostenlosen GraphViz Bibliothek mit dem Tool dot [12] erzeugt die Software auch Aufrufdiagramme und Include-Hiearchien, sodass die erstellte Software elegant und vollständig dokumentiert ist. In C++ werden außerdem weitere UML-Diagramme, wie Klassendiagramme, generiert.

---

## 2.4 Typen, Variablen und Konstanten

Das Konzept von Variablen ist in jeder Programmiersprache fundamental. Variablen sind symbolische Platzhalter für Daten. In ihrer Benutzung unterscheiden sich Konstanten (unveränderlich) und Variablen (veränderlich) nur dadurch, dass einer Variablen ein Wert zugewiesen werden kann, indem sie links vor ein Gleichheitszeichen (entspricht dem Zuweisungsoperator, Abschn. 2.5) gesetzt wird. Damit der Compiler den entsprechenden Speicherplatz organisieren kann, benötigen Variablen und Konstanten einen Typ.

## 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 TEMPSEN0 0
2 #define TEMPSEN1 1
3 unsigned char result;
4 result = ucTemperaturGet(TEMPSEN0);
```

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

## 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 *Gleitkommazahlen*. Ein logischer Datentyp existiert seit dem ANSI-Standard C99 als `_Bool`. In diesem Buch verwenden wir aus Gründen der Abwärtskompatibilität jedoch für logische Ausdrücke stets `unsigned char`.

Bei den Ganzzahltypen kann der Vorsatz `unsigned` (ohne Vorzeichen) verwendet werden. Dadurch verschiebt sich der Wertebereich in den Zahlenbereich  $\geq 0$  (siehe Tab. 2.1). Die Basis dafür ist die Zweierkomplement-Darstellung, die sicherstellt, dass jede mathematische Operation innerhalb des Wertebereichs abgebildet werden kann. Dabei bildet sich die negative Zahl aus der bitweisen Negierung der positiven Zahl plus einer binären 1. Das heißt, in 8-Bit Darstellung wäre die  $-1$  durch 0000 0001 komplementär 1111 1110 + 1 = 1111 1111 darzustellen. Die  $-128$  ist 1000 0000 komplementär 0111 1111 plus 1 = 1000 0000. Die Null bleibt damit eine Null und  $-1 + 1$  ergibt auch eine Null, da ein Überlauf über die 8-Bit Grenze nicht stattfindet.

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.

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  $b = r \cdot b^e$  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. Beispielsweise werden für `float` ein Bit für das Vorzeichen, acht Bit für den Exponenten zur Basis 2 in *signed*-Darstellung (Zweierkomplement) und 23 Bit für die Mantisse

**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>	4Byte (32 Bit)	$-2.147.483.648 \dots +2.147.483.647$	$0 \dots 4.294.967.295$
<code>long long</code>	8 Byte (64 Bit)	$-2^{63} \dots +2^{63}-1$ $-9.223.372.036.854.775.808 \dots$ $9.223.372.036.854.775.807$	$0 \dots 2^{64}-1$

**Tab. 2.2** Gleitkommatypen

Name	Größe	Typische Wertebereiche
float	4 Byte (32 Bit)	$\pm 1.175494351 \cdot 10^{-38} \dots 3.402823466 \cdot 10^{38}$
double	8 Byte (64 Bit)	$\pm 2.2250738585072014 \cdot 10^{-308} \dots 1.7976931348623158 \cdot 10^{308}$
long double	8...16 Byte (128 Bit)	$\pm 3.362103143112093506262677817321752602598 \cdot 10^{-4932} \dots 1.189731495357231765021263853030970205169 \cdot 10^{4932}$

als Binärzahl vorgehalten. In einem Format mit sehr geringer Genauigkeit wäre 1 also  $1.000 \cdot 2^0$ , 2 wäre  $1.000 \cdot 2^1$  und 0,375 wäre  $1.100 \cdot 2^{-2}$ , wobei die erste 1 nach dem Dezimalpunkt als 1/2 zählt, die zweite als 1/4 und so fort. Für eine richtig skalierte (oder normalisierte) Gleitkommazahl auf der Basis 2 ist die Ziffer vor dem Dezimalpunkt immer 1. Aus diesem Grund wird sie normalerweise weggelassen (obwohl dies eine spezielle Darstellung für die 0 erfordert). Daraus ergeben sich dann etwas „krumme“ Wertebereiche.

Generell sind die Gleitkommatypen in ihrer Größe implementierungsabhängig, die Compiler stellen Macros zur Verfügung, durch die die Grenzen der Wertebereiche abgerufen werden können (z. B. `LLONG_MIN`, `LLONG_MAX` oder `DBL_MIN`, `DBL_MAX`). Diese sind den Compilerdokumentationen zu entnehmen. Die in diesem Buch vorgestellte Software kommt gänzlich ohne Gleitkommazahlen aus. Sie 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-Code<sup>1</sup>), 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.

---

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

**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.

```
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 `zah11` und `zah12`.

```
int zah11, zah12;
```

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.

**Beispiel**

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:

- Dezimale Ganzzahlkonstanten 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.
- Binäre Ganzzahlkonstanten `0b01101010` (entspricht der Dezimalzahl 106 oder der Hexadezimalzahl  $0 \times 6A$ )

- 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)
- Spezialfall von Zeichenkonstanten sind Steuersequenzen, die mit einem \ (Backslash) angeführt werden, wie '\n' für Zeilenumbruch, '\t' für einen Tabulatorsprung oder '\\\' für einen Backslash. Man beachte, dass dies keine Zeichenketten sind, sondern Einzelzeichen!

Literalen Konstanten kann man durch ein Postfix Typen zuweisen, beispielsweise:

- 129u (unsigned)
- 123987 L (long)
- 987ul (unsigned long)
- –24.8E10 (double)
- 3.14159 f. (float)
- 1.414213562373095 L (long double)

Literele Konstanten, die eine Zahl repräsentieren, nennt man auch numerische Konstanten.

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 sogenannte *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 vor dem Compilerlauf 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. Zum Thema Makros finden Sie mehr in Abschn. [2.11](#).

Neuere Compiler nutzen das Schlüsselwort (type qualifier) `const` um dem Compiler anzuseigen, dass einer Variablen zur Laufzeit kein Wert mehr zugewiesen werden darf.

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

Bei der Deklaration einer Konstanten mit `const` muss unmittelbar die Initialisierung erfolgen. Es ist wichtig, zu verstehen, dass einer mit `const` deklarierte Variable zwar im Lauf des Programms kein Wert mehr zugewiesen werden kann, dennoch kann sie in einigen Fällen ihren Wert ändern, beispielsweise wenn es sich um ein Register handelt oder der Speicherplatz über eine Referenz überschrieben wird.

- ▶ 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 ein oder zwei *Operanden* miteinander verknüpft. Operatoren mit einem Operanden nennt man *unäre Operatoren*, solche mit zwei Operanden heißen *binäre Operatoren* (das 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 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. 2.3** Zuweisungsoperator

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

**Tab. 2.4** Arithmetische Operatoren

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

## 2.5.2 Logische Operatoren

Logische Operatoren werden auf boolsche 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 `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. 2.5** Logische Operatoren

Operator	Beschreibung	Anmerkung
$! a$	Logisches Nicht	liefert 1 <sup>2</sup> bei $a = 0$ oder 0 bei $a \neq 0$
$a \& \& b$	UND-Verknüpfung logisch	liefert 1 wenn $a$ und $b$ von 0 verschieden, ansonsten 0
$a     b$	ODER-Verknüpfung logisch	liefert 1, wenn $a$ oder $b$ von 0 verschieden, ansonsten 0
$a < b$	Kleiner	liefert 1 wenn $a$ kleiner $b$ sonst 0
$a > b$	Größer	liefert 1 wenn $a$ größer $b$ , sonst 0
$a <= b$	Kleiner oder gleich	liefert 1 wenn $a$ kleiner oder gleich $b$ sonst 0
$a >= b$	Größer oder gleich	liefert 1 wenn $a$ größer oder gleich $b$ sonst 0
$a == b$	Gleich (Vergleichsoperator, Achtung! Zwei Gleicheitszeichen)	liefert 1 wenn $a$ gleich $b$ , sonst 0
$a != b$	Ungleich	liefert 1 wenn $a$ ungleich $b$ , sonst 0

**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 $2^n$ )
$\sim a$	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 $b$

## 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.

---

<sup>2</sup>Die Eins ist im Folgenden im Sinne von „ungleich Null“ zu verstehen.

**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 Unions 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

**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 Ausdruck1 ausgewertet ansonsten Ausdruck 2 – einziger ternärer Operator in C	Abschn. 2.6.1

## 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 ;
```

**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)

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 Ganzzahloperationen 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. Eine **Bedingung** steht für einen Ausdruck vom Datentypen Integer. Der Ausdruck wird berechnet 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;
}
```

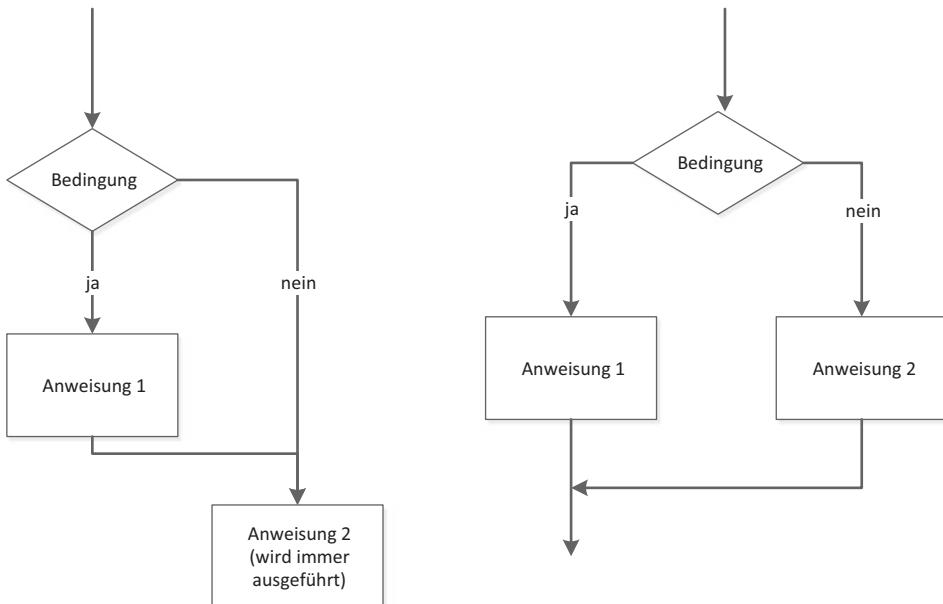
Der erste Anweisungsblock wird nur durchlaufen, wenn die Bedingung erfüllt ist. Der untere Block wird nur durchlaufen, wenn die Bedingung NICHT erfüllt ist. Dieser zweite Block, der mit `else` eröffnet wird, kann auch weggelassen werden. Als Bedingungen können beliebige Ausdrücke verwendet werden, die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



**Abb. 2.3** Programmablaufplan für eine Verzweigung ohne (links) und mit else (rechts)

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;
```

## 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 zu 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 boolschen Verknüpfungen dieser Ausdrücke zusammengesetzt sind. Kopf-

gesteuerte 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 muss, 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

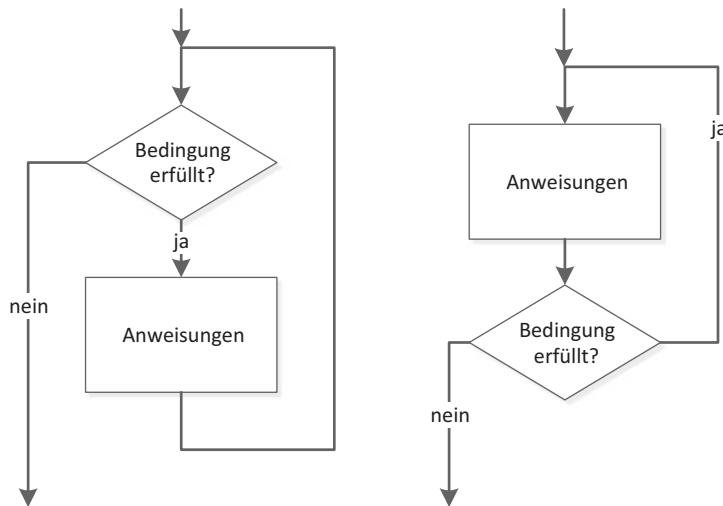
Fußgesteuerte Schleifen oder *do-while*-Schleifen werden immer dann eingesetzt, wenn der Anweisungsblock (Schleifenkörper) mindestens einmal durchlaufen werden muss, beispielsweise um eine Bedingung abzufragen. Die Anweisungen nach einmaliger Ausführung solange wiederholt, wie die Bedingung, die im Fuß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

## 2.7.3 Zählschleifen

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



**Abb. 2.4** Kopf-(links) und fußgesteuerte (rechts) Schleifen im Flussdiagramm

```
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  $y=f(x)$ . Funktionen bilden jedes Element aus einem Raum, der von einer oder mehreren Veränderlichen aufgespannt wird, auf ein Element aus einem anderen Raum ab, das Eingangselement wird gemäß der Funktion verändert z. B.  $(x^2)$ . Das Ergebnis der Berechnung wird dann  $y$  zugewiesen. Funktionen mehrerer Veränderlicher lauten dann beispielsweise  $y=f(a,b)$ ;

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) ausgewertet, sodass er beispielsweise einer anderen Variablen zugewiesen werden kann. 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 einer 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.

C macht es zwar generell möglich, eine Funktion gleichzeitig zu deklarieren und zu definieren, indem man die Funktion erst verwendet, nachdem sie definiert wurde. Eine Definition deklariert aber die Funktion immer auch gleich. Man sollte sich aber von vorneherein angewöhnen, die Deklaration und die Definition zu trennen, da die im Folgenden verwendeten Konzepte nur dann Sinn machen.

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) 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 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. Siehe dazu auch Abschn. 2.11.1

### 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 eines Programms. Diese werden über Bibliotheksfiles im Bindeprozess geladen (siehe später) während die Funktionsdeklaration über Header-Dateien (Datei-Endung .h) eingebunden werden.

C-Kompiler liefern in der Regel einige Standardbibliotheken mit Grundfunktionen mit, 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, oder die Ausgabe von Daten auf Streams. Generell werden Header verwendet, um die Schnittstellen von Modulen (d. h. deren Funktionsaufrufe und Makros) anderen Modulen zur Verfügung zu stellen, ohne dass man den Quellcode des Moduls offenlegen muss. 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. Siehe dazu auch Abschn. 2.11.1

- ▶ Man beachte, dass im Header selbst Funktionen nicht definiert sondern nur deklariert werden dürfen! Syntaktisch ist es zwar vordergründig möglich, kann aber ins Auge gehen, wenn der Header mehrfach innerhalb eines Programms eingebunden wird. Dann meldet der Linker einen Fehler.

### 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:

#### Beispiel

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 sogenannte „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();
void setStatus(unsigned char s);
#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 function 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 Zeichenketten

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 zehn 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  $0 \times 46$ , das Zeichen „Z“ ist  $0 \times 5A$ .

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  $0 \times 00$ , da diese im ASCII-Code nicht druckbar ist – eine ASCII-Null, also die druckbare Ziffer 0, ist im ASCII-Code der Zahl  $0 \times 30$  (dezimal 48) zugewiesen.

**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	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	

**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

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)

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. Das nötige Wissen können Sie sich 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 ist dies bei Displays und Terminals in der Regel nicht sinnvoll. 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 [13]. 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.

Strukturen kann man auch verschachteln:

```
struct time {
    unsigned int hr;
    unsigned int min;
    unsigned int sec;
};

struct date {
    unsigned int day;
    unsigned int month;
    int year;
};

struct appointment {
    struct date d;
    struct time t;
};

struct appointment a = { {19, 12, 2017}, {20, 15, 0} };
```

Der Zugriff erfolgt dann analog:

```
int stunde = a.time.hr ;
```

Über das Schlüsselwort `typedef` lässt sich ein eigener komplexer Datentyp definieren, sodass man bei weiterer Verwendung auf `struct` verzichten kann.

```
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. Man sollte beachten, dass Rechner, deren Verarbeitungsbreite größer als ein Byte ist (z. B. 16, 32 oder 64 Byte) in der Regel auch die Strukturen entsprechend aufzblähen. Ist eine Komponente auf einem 64-Bit-Compiler beispielsweise vom Typ `char` wird für sie in der Regel ein 64-Bit-Feld (= 8 Byte) angelegt. Man kann dieses Verhalten bei manchen Compilern mit der Option `#pragma pack(1)` optimieren.

Bei der folgenden Definition:

```
struct store {
    char x;
    int z;
} ;
#pragma pack(1)
struct store2 {
    char x;
    int z;
} ;
```

geben die Befehle `sizeof(store)` und `sizeof(store2)` bei einem 32 Bit GCC Compiler unterschiedliche Werte aus, nämlich 8 und 5. Die ungepackte Struktur besteht aus zwei mal vier Bytes. Man nennt dieses Verhalten Alignment. Damit wird der Zugriff

schneller, der Arbeitsspeicher wird aber voller. In aller Regel spielt letzteres eine untergeordnete Rolle, daher verbieten viele Firmen das Packen von Strukturen in ihrem Code. In der Pointerarithmetik Abschn. 2.9.5.2 sind diese Verhältnisse des Alignment bereits berücksichtigt. Mehr zum Schlüsselwort `#pragma` erfahren Sie in Abschn. 2.11.2.

### 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; } vzyll;
    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 acht Bytes gegeben. Das erste Byte wird als Service ID (SID) interpretiert, das den Rest der Botschaft wie folgt festlegt:

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)`<sup>3</sup> 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;
}
```

---

<sup>3</sup>Zur Verwendung des `char*`-Zeigers siehe Abschn. [2.9.5](#).

## 2.9.4 Aufzählungstypen

Aufzählungstypen erleichtern den Zugriff auf feste Werte über symbolische Konstanten, ersetzen also aufwendige `#defines` oder `const` Variablen.

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
```

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!

Das Schlüsselwort `enum` zusammen mit `typedef` lässt sich elegant als Typ mit kontrolliertem Wertebereich verwenden:

```
typedef enum { monday=1, tuesday, wednesday, thursday, friday,
saturday, sunday} tDay;

typedef enum {jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct,
nov, dec} tMonth;

void printDate(tDay day, tMonth month);
```

Der Aufruf gestaltet sich dann so:

```
printDate(monday, apr);
```

und ist der Lesbarkeit und Typsicherheit halber der reinen Verwendung von Konstanten vorzuziehen.

## 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 bei einem komplexen Mikroprozessor 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 (in Bytes) im sogenannten *Heap* und gibt einen Zeiger auf diesen Speicher zurück, den man in einer Zeigervariablen ablegt. `void* calloc (size_t num, size_t size)` liefert wiederum einen solchen Zeiger auf ein Feld, das `num` Objekte der Größe `size` enthält. Nutzt man die Variable im Lauf des Programms anderweitig, so muss der Speicher mit `free()` wieder freigegeben werden. Im Moment verzichten wir auf eine weitergehende Erläuterung.

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 //Inhalt der Variablen, auf die b zeigt
b=&a //setzt die Pointervariable b auf die Adresse der Variablen a
a=*b //kopiert den Inhalt der Variablen, auf die b zeigt, in a
*b=a; //weist der Variablen, auf die b zeigt, den Wert von a zu
```

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;
    tmp = *x;
    *x = *y;
    *y = tmp;
}
```

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

### 2.9.5.2 Zeigerarithmetik

Grundsätzlich kann man 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){return ++a;}
int Dekrement(int a){return --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.

### 2.9.5.4 Const Zeiger

Oftmals sieht man das Schlüsselwort `const` im Zusammenhang mit Zeigern. Hier sind drei Beispiele:

```
const char * myPtr = &char_A;
char * const myPtr = &char_A;
const char * const myPtr = &char_A;
```

Die erste Zeile mit `const` deklariert einen Zeiger auf ein konstantes Zeichen. Sie können diesen Zeiger nicht verwenden, um den Wert zu ändern, auf den er zeigt. Natürlich könnte aber ein anderer Zeiger, der auf dieselbe Speicherstelle zeigt, das Zeichen dennoch verändern.

Die zweite Zeile mit `const` deklariert einen konstanten Zeiger auf ein Zeichen. Die im Zeiger gespeicherte Position kann nicht geändert werden. Sie können also die Adresse, auf die dieser Zeiger zeigt, nicht ändern.

Die dritte Zeile mit `const` deklariert einen Zeiger auf ein Zeichen, weder der Zeigerwert (also die Adresse) noch der Wert, auf den mit diesem Zeiger referenziert wird, kann geändert werden.

---

## 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 Prozessteil (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 auf-
        gerufen
    }
    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, ver sehentliches doppeltes Einbinden verhindert, indem nach dem Makro `PROGRAMM NAME_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.

In den folgenden Kap. 3 und 4 wird dieser Rahmen weiter ausgebaut, bis hin zu einem kleinen Echtzeitbetriebssystem.

## 2.11 Arbeiten mit dem Präompiler

Der Präompiler oder Präprozessor ist ein mächtiges Werkzeug, um den Compilierablauf zu steuern. Mit ihm kann man gezielt Compileroptionen ein- und ausschalten oder ganze Codeblöcke modifizieren beziehungsweise an- und ausschalten um damit zur Laufzeit effizienteren, flexibleren und lesbaren Code zu generieren. Allerdings verbergen sich hinter den Befehlen oftmals Fallstricke. Aus Platzgründen werden hier nur die wichtigsten Eigenschaften der Präompiler-Befehle genannt. In der Literaturliste finden sich Bücher, die dem Thema weit mehr Platz einräumen.

### 2.11.1 #define und Arbeiten mit Makros

In Abschn. 2.4.3 wurde bereits der Präompiler Befehl `#define` als Methode zur Definition von Makros eingeführt. In Abschn. 2.8.2 und 2.8.3 wurde ein Makro mit Parameterliste eingeführt. An dieser Stelle sollen noch weitere Eigenschaften von Makros besprochen werden.

- a) Da Makros auch per Kommandozeile an den Compiler übertragen werden können, ist das Makro ein beliebtes Mittel um Code für verschiedene Zielhardware zu übersetzen:

```
#ifdef TARGET_1
    #include <target1.h>
#elif TARGET_2
    #include <target2.h>
#endif
#define INCFILE "global_include.h"
#include INCFILE
```

Ist das Makro `TARGET_1` definiert, dann wird ein anderes Headerfile eingefügt, als wenn `TARGET_2` definiert ist. Gesteuert wird dies durch `#ifdef` (= if defined), `#elif` (=else if defined) und `#endif` (als Begrenzer). Auch Code kann an dieser Stelle eingesetzt werden, der dann beispielsweise unterschiedliche Hardwareabstraktionen enthält (Kap. 4).

```
#ifdef _DEBUG_
//hier debugging/simulation code einsetzen
    Get_simulated_event();
#elif _TARGET_
```

```
// hier target code einsetzen
Get_event();
#else
#error No target defined
#endif
```

Das Makro **#error** bricht den Vorgang ab und macht mit einer Fehlermeldung des Compilers auf sich aufmerksam.

- b) Üblicherweise werden Headerfiles mit `#ifdef ... #endif` umschlossen, um irrtümliche Mehrfachdeklarationen zu verhindern, viele IDEs machen dies automatisch, wenn ein neues Headerfile angelegt wird (dies wird auch in Abschn. 2.8.4 beschrieben):

```
#ifndef PROGRAMMNAME_H
#define PROGRAMMNAME_H
/* Hier kommen die Funktionsdeklarationen bzw. struct und Konstanten */
void Init(void);
#endif
```

Alternativ kann man bei GCC die `#include` Direktive durch ein `#import` ersetzen, das den Vorteil hat, sich selbst auf Mehrfachnutzung zu überprüfen und diese zu verhindern. Aus Kompatibilitätsgründen empfehlen wir das jedoch nicht.

- c) Es gibt eine Reihe von Standardmakros, die der Compiler durch Code ersetzt, Beispiele sind in Tab. 2.12 zu sehen. Die Liste ist bewusst nicht vollständig um Sie nicht zu verwirren. Viele Compiler bringen Dutzende eigener Standardmakros mit, so auch der AVR GCC, den wir hier einsetzen
- d) Wie in Abschn. 2.8.2 schon erwähnt, kann man Makros auch mit Parametern versehen:

**Tab. 2.12** Standardmakros

<code>__DATE__</code>	Das aktuelle Compilier-Datum im „MMM DD YYYY“ Format
<code>__TIME__</code>	Die Compilierzeit im „HH:MM:SS“ Format
<code>__FILE__</code>	Den Namen des aktuellen Files als literale Zeichenkettenkonstante
<code>__LINE__</code>	Die Zeilennummer im aktuellen File als numerische Konstante
<code>__func__</code>	Der Name der Funktion, in der das Makro steht
<code>NULL</code>	Der Nullpointer
<code>offsetof(type, member-designator)</code>	Gibt den Offset, also die Position einer Komponenten in einer Struktur zurück

```
#define square(x) ((x) * (x))
#define MAX(x,y) ((x) > (y) ? (x) : (y))
```

Dieses Werkzeug ist sehr mächtig und wir möchten an der Stelle darauf hinweisen, dass Anfänger sehr vorsichtig damit umgehen sollen! Da die Auflösung der Makros im Quelltext erfolgt, können sich beispielsweise bei der Verwendung von Semikolons üble Fehler einschleichen. Weitere interessante Eigenschaften finden sich in der Online Dokumentation des GCC Compilers (CPP Manual) [14] in der jeweiligen Version in Kap. 3, beispielsweise die Verwendung von Makros mit einer variablen Anzahl von Parametern oder die Verkettung von Parametern.

## 2.11.2 #pragma

Im Zusammenhang mit dem GCC ist es manchmal notwendig, einzelne Compileroptionen stellenweise umzuschalten. Ein ganz wichtiges Beispiel ist die Verwendung des Optimizers. Dieser generiert nach semantischen Regeln erheblich effizienteren Code als eine reine Eins-zu-Eins-Übersetzung. Allerdings gehen dabei oftmals bewusst eingebrachte Befehle verloren, denen der Optimizer keine Bedeutung zumisst. Mit anderen Worten: Manchmal muss man den Optimizer ausschalten, damit der Code noch läuft. Normalerweise geschieht das in den Optionseinstellungen des Compilers (in der IDE oder über Kommandozeilenparameter). Andererseits stößt man damit recht häufig an die Grenzen des Flash-Speichers. Optimal wäre, nur an den entscheidenden Stellen den Optimierer abzustellen. Hier kommt `#pragma` ins Spiel. Es ist ein mächtiges Werkzeug, das über Dutzende von Einstellungsmöglichkeiten verfügt und praktisch den gesamten Compiler aus dem Coder heraus steuern kann (siehe [14] GCC Manual Kap. 6). Wir nutzen bisweilen:

```
#pragma GCC optimize (string, ...)
```

und setzen `string` zur jeweiligen Optimizer-Option, beispielsweise.

```
#pragma GCC push_options
#pragma GCC optimize("O0")
```

Schaltet den Optimizer aus. Am Ende des Codeblocks, für den die Optionen gelten sollen schreiben wir dann:

```
#pragma GCC pop_options
```

**Tab. 2.13** Präcompiler-Befehle

Schlüsselwort	Bedeutung
#define	Definiert ein Makro
#include	Kopiert eine Headerdatei an die aktuelle Stelle in Quelltext
#undef	Löscht ein existierendes Makro
#ifdef	Liefert “true”, wenn das genannte Makro existiert, der Code im folgenden Bedingungsblock wird dann vom Compiler übersetzt (ansonsten übersprungen)
#ifndef	Das Gegenteil: liefert “true” wenn das Makro nicht existiert
#if	Überprüft, ob eine Compilerbedingung erfüllt ist
#else	Dieser Zweig wird übersetzt, wenn die vorangegangene #if oder #ifdef Bedingung nicht erfüllt war (Alternative)
#elif	#else und #if in einem Statement, also eine Alternative mit weiterer Bedingung
#endif	Ende eines Bedingungsblocks
#error	Stoppt den Compiliervorgang und gibt eine Fehlermeldung auf dem Kanal stderr aus
#pragma	Weist den Compiler an, das folgende Compilerkommando auszuführen

#pragma GCC push\_options speichert dabei die bestehenden Compiler-Optionen ab und #pragma GCC pop\_options holt sie aus dem Speicher zurück, sodass der alte Zustand wiederhergestellt ist.

### 2.11.3 Zusammenfassung der Präcompiler-Befehle

Die Tab. 2.13 fasst die gebräuchlichen Präcompiler-Befehle nochmals zusammen.

---

## 2.12 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 Microchip Studio<sup>4</sup> benutzt, das Microchip unter [www.microchip.com](http://www.microchip.com) kostenlos verteilt und das auf dem bekannten Visual Studio basiert. Grundsätzlich kann man aber auch kommando-

---

<sup>4</sup>Früher AtmelStudio.

zeilenorientierte Tools verwenden, beispielsweise die bekannte GNU-Toolkette, die auch den C-Compiler für Microchip Studio 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 Präcompiler 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 den 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 Atmel-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 sogenannte 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 Microchip Studio 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 sogenannte 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](http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=57853). Zugegriffen: 20. Aug. 2020.
2. ISO/IEC 14882:2014. Standard for programming language C++. <https://isocpp.org/std/the-standard>. Zugegriffen: 20. Aug. 2020
3. Mathworks. Embedded coder. <http://de.mathworks.com/products/embedded-coder/features.html>. Zugegriffen: 20. Aug. 2020
4. Wikibooks. C-Programmierung. <https://de.wikibooks.org/wiki/C-Programmierung>. Zugegriffen: 20. Aug. 2020
5. Erlenkötter, H. (1999). *C: Programmieren von Anfang an* (23. Aufl.). Rowohlt Taschenbuch.
6. Kernighan, B. W., & Ritchie, D. M. (1990). *Programmieren in C: Mit dem C-Reference Manual in deutscher Sprache*. Hanser.
7. Kernighan, B. W., & Ritchie, D. M. (2010) *The C Programming Language*, 2. Auflage, Prentice Hall (englisch)
8. Gookin, D. (2010). *C für Dummies*, 2. Wiley-VCH.
9. Goll, J. (2014). *C als erste Programmiersprache- Mit den Konzepten von C11* (8. Aufl.). Springer
10. Wiegelmann, J. (2017). *Softwareentwicklung in C für Mikroprozessoren und Mikrocontroller* (7., neu bearbeitete und erweiterte Aufl.). VDE.
11. Dimitri van Heesch. Doxygen. <https://www.doxygen.nl/download.html>. Zugegriffen: 11. März 2021.
12. Emden Gansner GraphVIZ dot (online). <https://graphviz.org>. Zugegriffen: 11. März 2021.
13. C-Programmierung: Komplexe Datentypen. (27. Juli 2015). Wikibooks, Die freie Bibliothek. [https://de.wikibooks.org/w/index.php?title=C-Programmierung:\\_Komplexe\\_Datentypen&oldid=764916](https://de.wikibooks.org/w/index.php?title=C-Programmierung:_Komplexe_Datentypen&oldid=764916). Zugegriffen: 11. März. 2021
14. GCC Online Dokumentation. (2020). GCC10.2. <http://gcc.gnu.org/onlinedocs/>. Zugegriffen: 20. Aug. 2020



# Programmierung von AVR Mikrocontrollern

3

## Zusammenfassung

Das Kapitel beschreibt die wichtigsten Peripherieelemente der AVR-Familie, also grundsätzlichen Funktionen, mit denen ein Mikrocontroller der AVR-Familie mit der Außenwelt kommuniziert. Beschrieben wird der Aufbau von Peripherieschaltungen, die sich natürlich auch auf andere Vertreter der AVR-Familie oder andere Mikrocontrollerfamilien anwenden lassen. Interrupts, Digitale Ein- und Ausgänge, AD-Wandler, Timer, PWM, der EEPROM-Speicher und Methoden des Power Managements sind die wichtigsten Themen des Kapitels. Die ihnen zugrundeliegenden Ideen sind auch auf andere Prozessoren übertragbar.

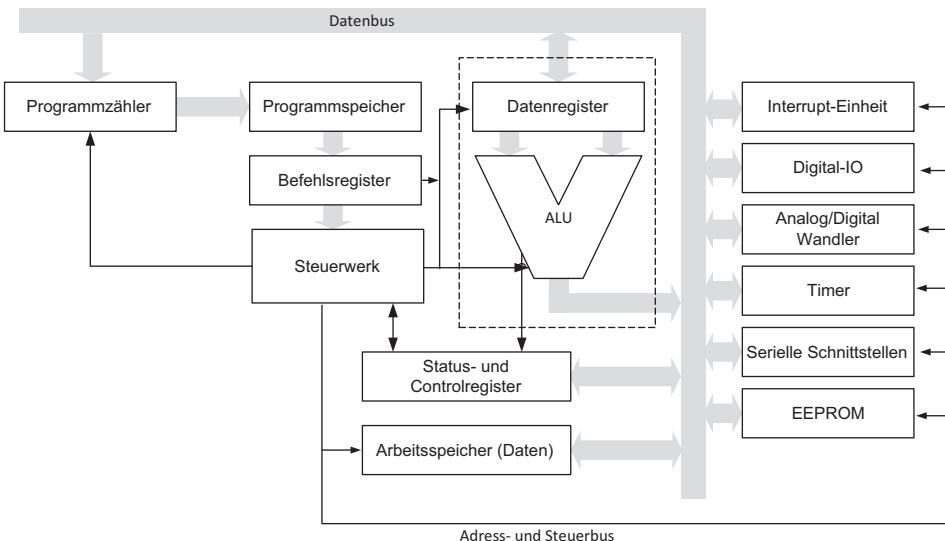
Am Ende dieses Kapitels sind Sie in der Lage, einfache, digitale I/O Anschlüsse einzulesen und zu schalten, analoge Sensoren auszulesen, einen Motor mit einer Vollbrücke anzusteuern, das integrierte EEPROM zu beschreiben und auszulesen und zeitabhängige Abläufe und Signale aus dem Timer zu generieren. Auch das Powermanagement wird kurz besprochen. Alle Beispiele in diesem und den folgenden Kapiteln sind getestet und mit dem MicrochipStudio<sup>1</sup> (Version 6 und höher) kompiliert worden.

<sup>1</sup> Bisher: AtmelStudio.

Kapitel 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 ausführlichen und gut geschriebenen Datenblattes und der umfangreichen Dokumentation [1] empfohlen. Allein das Datenblatt hat über 660 Seiten und wird ständig gepflegt! Daher ist es in einer konkreten Entwicklung immer als Referenz zu befragen, was dieses Buch nicht leisten kann. 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 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:



**Abb. 3.1** Grundlegende Architektur von AVR-Mikrocontrollern

- 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 Oszillator oder einen externen Quarzoszillator 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 festverdrahteten 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, einfache Architekturen begnügen sich mit zweien (Fetch/Decode/Execute und Write Back). Zur Beschleunigung können sogenannte 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, boole'sche 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 sequenziell („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 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 sogenannte 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:

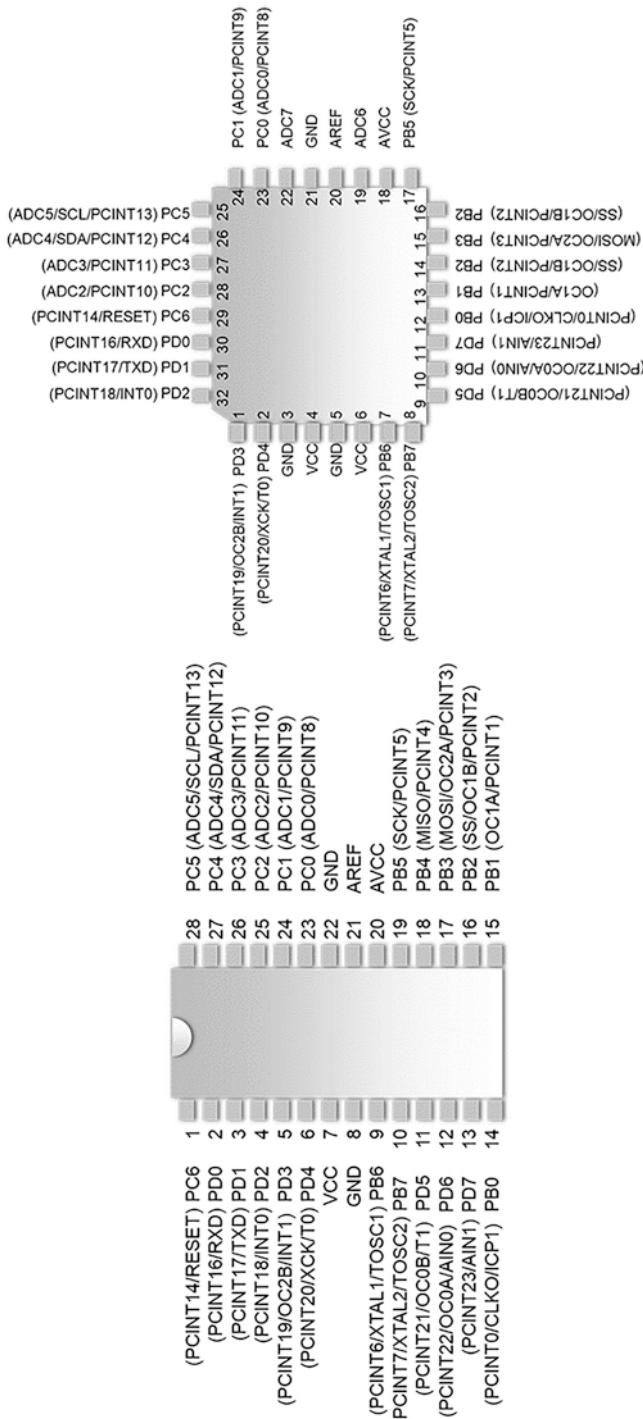
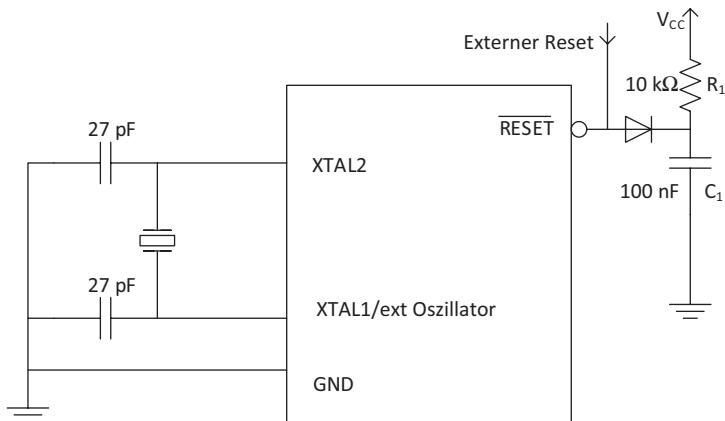


Abb. 3.2 Pinbelegung des ATMega8 im PDIP Gehäuse (links) und im TQFP Gehäuse (rechts) nach [1]



**Abb. 3.3** Beschaltung des Oszillators und des RESET Eingangs

- 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 Oszillatoren eingänge 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 sogenannten 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).

### 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 heißtt, das Programm startet dort.

Der Prozessor kann über verschiedene Mechanismen extern zurückgesetzt werden:

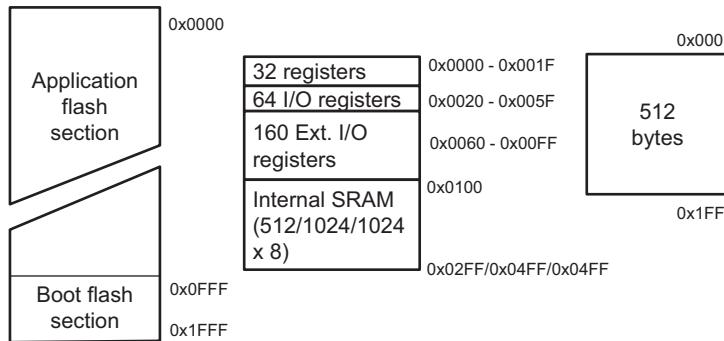
- Beim Einschalten (Power-On-Reset) bildet der ungeladene Kondensator am RESET-Pin einen Kurzschluss gegen Masse. Dadurch wird der Reset-Pin 1 kurz auf 0 gelegt bis sich C1 über R1 aufgeladen hat.
- Eine RESET-Taste am RESET-Pin bewirkt einen manuellen Reset.
- Am RESET-Pin ist ein sogenannter Watchdog 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 Microchip, 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.

Abb. 3.4a ist der Flash-Speicher zu sehen, der beim ATmega88 8 kByte groß ist. Im Sinne der Harvard-Architektur stellt er den Programmspeicher (Firmware) dar. Rechts im Bild ist der EEPROM-Speicher zu sehen, dieser hält persistente (also über das Ausschalten hinaus gültige) Daten, z. B. Fehlerspeichereinträge oder Parametrierungen, die



**Abb. 3.4** Speicher der AVR-Familie am Beispiel ATmega88. Links: Programmspeicher, Mitte: Datenspeicher, Rechts: EEPROM. (Nach [1])

man unabhängig vom Quellcode einstellen will. So lässt sich beispielsweise das Verhalten der Software zur Laufzeit anpassen oder kalibrieren.

Der Flash und EEPROM wird 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.
- 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 Programmcode liest.
- Manche Vertreter der AVR-Familie besitzen eine eigene Boot-Flash-Section. Diese hält die Bootloaderroutine und kann in ihrer Größe variiert und optional verriegelt werden, sodass 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 der Bildmitte von Abb. 3.4 sind die Register und der Arbeitsspeicher angedeutet. Im Fall des ATmega88 besitzt dieser 1 kByte freien Speicherplatz<sup>2</sup> 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 Variable benötigt. In der Abbildung sind die 32 Arbeitsregister skizziert, die direkt von

<sup>2</sup>Im Bild sind auch die Speicherausdehnungen des ATmega48 und des ATmega168 mit 512 Byte bzw. 1 KByte dargestellt.

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 sogenannten IO-Register. Diese steuern die Funktionen des Mikrocontrollers. In den folgenden Abschnitten werden die wichtigsten davon beschrieben. Eine ausführliche Referenz findet sich in [1].

---

### 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 Mitte).

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 0x005 F) 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, in dem die Variablen abgelegt werden können. Der Compiler legt automatisch fest, wo Variablen abgelegt werden, lokale Variablen sind üblicherweise in Registern verteilt, während globale und static Variablen im Hauptspeicher Platz finden, ebenso wie Felder.

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 sind<sup>3</sup>.

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)
```

---

<sup>3</sup>Um genau zu sein, bindet `io.h` das dem Prozessortyp entsprechende File ein, z. B. `iomx8.h` für die ATmegax8 Serie.

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
- `<<` 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.

```
PORTB = PORTB | 0x10;
```

oder in der Kurzschrifweise:

```
PORTB |= 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

```
PORTB |= (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 PORTB ver-ODERt. Da die Null das neutrale Element der ODER-Operation darstellt, wird nur das vierte Bit von PORTB 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:

```
PORTB &= ~0x10; //oder in Schiebeschreibweise
PORTB &= ~(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 (gezählt ab 0!) 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. Wenn man also einzelne Registerbits auf 0 oder 1 setzen will, benötigt man zwei Zeilen, eine für die Einsen und eine für die Nullen.

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

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

Das Setzen eines einzelnen Bits kann man natürlich schöner schreiben, indem man sich zwei Makros definiert

```
#define setbit(reg, bit) reg |= (1<<bit)
#define clrbit(reg, bit) reg &=~(1<<bit)
//Aufruf:
setbit(PORTB, PB0);
clrbit(PORTB, PB0);
```

und dann die Bits einzeln setzt oder zurücksetzt.

An dieser Stelle sei darauf hingewiesen, dass eine umfangreiche Funktionsbibliothek AVR Libc existiert, die beispielsweise mit dem Microchip 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 an wenigen Stellen auch diese Funktionen in den weiteren Teilen des Buches. Eine ausführliche Beschreibung ist in [3] und [4] zu finden.

## 3.5 Digitaler Input/Output

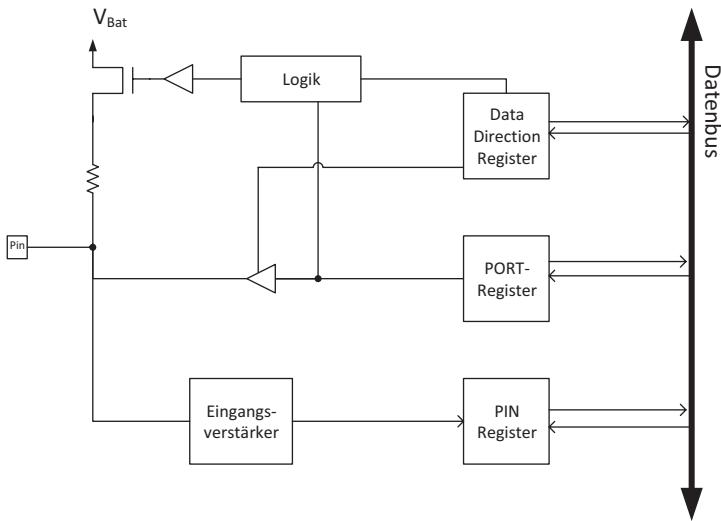
### 3.5.1 Grundsätzlicher Aufbau

Nahezu jeder Pin eines AVR-Prozessors, 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.

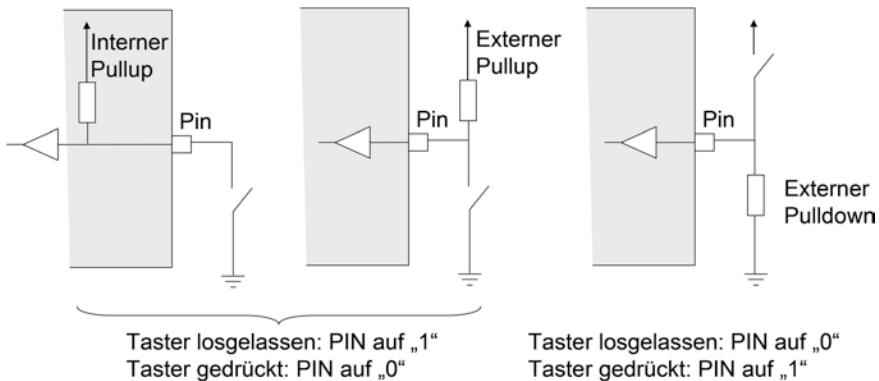
Abb. 3.6 zeigt mögliche Eingangsbeschaltungen. Statt eines Tasters können natürlich auch die Endstufe einer digitalen Schaltung oder der Fototransistor eines Optokopplers angenommen werden.

Das Datenrichtungs-Register (DDRxn<sup>4</sup>) 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.

<sup>4</sup>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. 3.5** Vereinfachtes Prinzipbild der digitalen Ein- und Ausgänge. (Nach [1])



**Abb. 3.6** Beschaltungsmöglichkeiten für digitale Eingänge

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);
```

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 Eingangspiegel 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.

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-Betrieb (ähnlich Open-Collector in TTL-Logik).

In Wirklichkeit ist die Beschaltung nach Abb. 3.5 etwas komplizierter, denn 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:

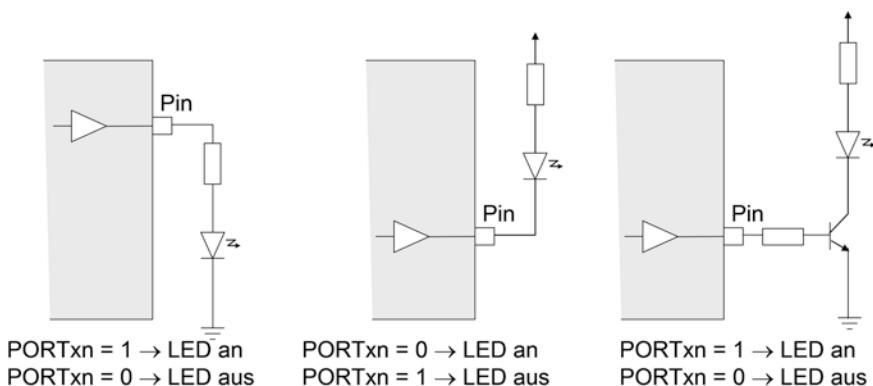


Abb. 3.7 Beschaltungsmöglichkeiten für digitale Ausgänge

**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

### 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. 3.7 links 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. Gewöhnt 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 Bauenteile nach Abb. 3.6 Links oder Mitte 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

Ein Interrupt ist eine Unterbrechung des normalen Programmablaufs<sup>5</sup>. Sobald ein Interrupt angefordert wird (das auslösende Ereignis wird Unterbrechungsanforderung oder Interrupt Request, IRQ genannt), pausiert das normale Programm und eine Interruptfunktion (Interrupt-Serviceroutine ISR) wird ausgeführt. Ist diese abgearbeitet, wird das normale Programm fortgesetzt.

---

<sup>5</sup>Aus dem Englischen: to interrupt für unterbrechen, Lateinisch: interrumpere.

### 3.6.1 Einstieg in den Umgang mit Interrupts

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-Serviceroutine (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.

### 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

**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

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 PCMSK<sub>n</sub> (Pin Change Mask Register). Dabei kann jeder einzelne Pin eines der drei Ports einen Interrupt auslösen, sobald sich sein Wert ändert (positive und negative Flanke). Jeder Port bildet eine eigene Gruppe von Interrupts (PORTB: PCINT

**Tab. 3.3** Namen der ISR für den ATmega88

Vektor Nummer	Ausgelöst durch	Interrupt-Serviceroutine 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

0..7, PORTC: PCINT 8..14<sup>6</sup>, 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.

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

---

<sup>6</sup>Port C hat nur sieben Eingänge.

**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

**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

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-Serviceroutine 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-Serviceroutinen 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 in späteren Kapiteln.

### 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 steuern die Bits ISC00 bis ISC11 das Verhalten gemäß Tab. 3.9, wobei die Tabelle, so zu lesen ist, dass das „x“ durch die Nummer des Interrupts zu ersetzen ist. Alle vier Bit auf 1 gesetzt würde also bei steigender Flanke auf beiden Anschlüssen einen Interrupt auslösen.

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.

**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

---

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

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

```
void INTO_Init()
{
    EICRA &= ~(1 << ISC00) & ~(1 << ISC01);
    EIMSK |= (1 << 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 drei 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 triggern sie einen Interrupt oder setzen einen Ausgangsport. Im PWM Modus wird das Setzen und Rücksetzen der Ausgänge so gesteuert, dass ein variables Tastverhältnis bei gleichbleibender Frequenz erzielt wird.

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. 3.8 links 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  $f_{\text{CLK\_I/O}}/8$ ,  $f_{\text{CLK\_I/O}}/64$ ,  $f_{\text{CLK\_I/O}}/256$  oder  $f_{\text{CLK\_I/O}}/1024$  setzen. Das Ganze geschieht über einen Multiplexer.

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

Die Timer 0 und 1 können statt über den Prescale 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.

Der Timer 2 kennt keine externen Taktquellen dafür ist der Prescaler feiner abgestuft einstellbar, gemäß Tab. 3.12.

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 abgeleitet 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.

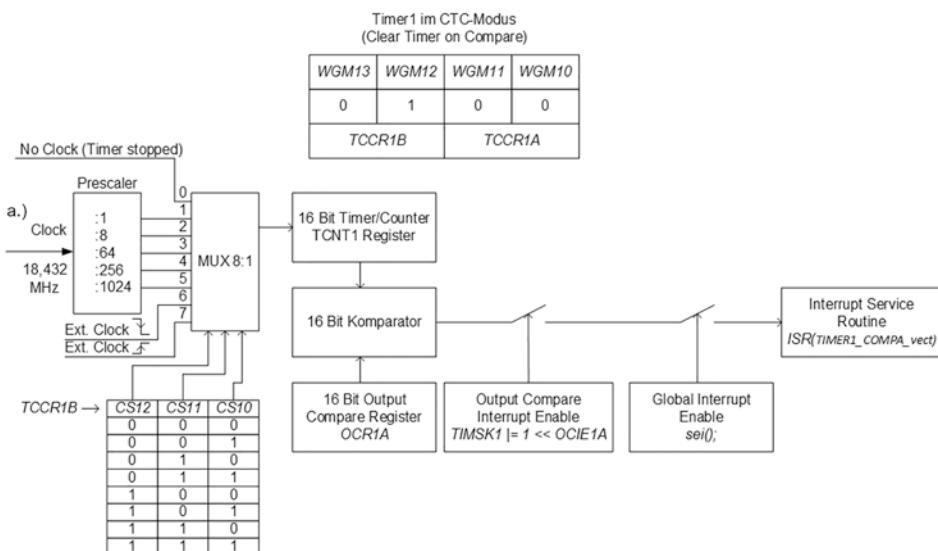


Abb. 3.8 Prinzipieller Aufbau des 16-Bit-Timers 1

**Tab. 3.11** Auswahl der Taktquelle im Register TCCR1B (Timer 1, analog zu Timer 0)

CS <sub>12</sub>	CS <sub>11</sub>	CS <sub>10</sub>	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

**Tab. 3.12** Auswahl der Taktquelle im Register TCCR2B (Timer 2)

CS <sub>12</sub>	CS <sub>11</sub>	CS <sub>10</sub>	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	$f_{CLK\_I/O}/32$ (vom Prescaler)
1	0	0	$f_{CLK\_I/O}/64$ (vom Prescaler)
1	0	1	$f_{CLK\_I/O}/128$ (vom Prescaler)
1	1	0	$f_{CLK\_I/O}/256$ (vom Prescaler)
1	1	1	$f_{CLK\_I/O}/1024$ (vom Prescaler)

Alternativ kann man den Interrupt auslösen, wenn der Timer überläuft (d. h. von 0xFF auf 0x00 bzw. 0xFFFF auf 0x0000). 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.

### 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 ähnlich, die Zähl- und Vergleichsregister beim 16-Bit-Timer 1 sind allerdings zwei Byte lang, außerdem beherrscht er doppelt so viele Modi

und besitzt deshalb mehr Mode-Bits. Bei Timer 2 gibt es außerdem keine externen Taktquellen. Daher sind die Register ein wenig unterschiedlich belegt. 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. 3.13 und 3.14 sind die Register TCCR0A und TCCR0B des Timer 0 beschrieben. Sie 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. 3.15)

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

OCIEkB: Output Compare Interrupt Enable Output Compare Match B

OCIEkA: Output Compare Interrupt Enable Output Compare Match A

TOIE: Timer Counter Overflow Interrupt Enable

### 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

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

**Tab. 3.13** 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. 3.14** 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. 3.15** Timer Interrupt Mask Register TIMSKn

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
–	–	–	–	–	OCIEkB	OCIEkA	TOIE

ergibt sich damit eine Periodendauer  $T$  von  $55,555\dots \mu\text{s}$ . Dies bedeutet, dass der Interrupt alle ca.  $55,5 \mu\text{s}$  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-Serviceroutine aufgerufen wird (Bitte beachten: 0–179 sind 180 Durchläufe). Dadurch wird die ISR alle  $180 \cdot 55,5 \mu\text{s} = 10 \text{ ms}$  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 einer Sekunde. 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. 3.16), 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.

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

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

**Tab. 3.16** Einige Waveform-Generation-Modes für Timer 0

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

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

```
volatile unsigned char ucFlag10ms = 0;
volatile unsigned char ucFlag1s = 0;
volatile 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. Das Schlüsselwort `volatile` verhindert hier, dass der Optimierer im folgenden Code die Verzweigung wegoptimiert, da die Variable ucFlag1s im Programmfluss nirgends gesetzt wird, sondern in einer (zum Compilierzeitpunkt unbekannten) Interrupt-Serviceroutine.

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

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

```
extern volatile unsigned char ucFlag10ms;
extern volatile 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<sub>xy</sub> (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 Vorverteilung
    (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 Vorteilerverhältnis, 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. 3.17** 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 folgenden Tab. 3.17 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 Tastverhältnis<sup>7</sup>  $D$  (mit  $0 < D < 1$ ) auszugeben. Der Mittelwert der Signalamplitude eines solchen Signals ist:

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

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

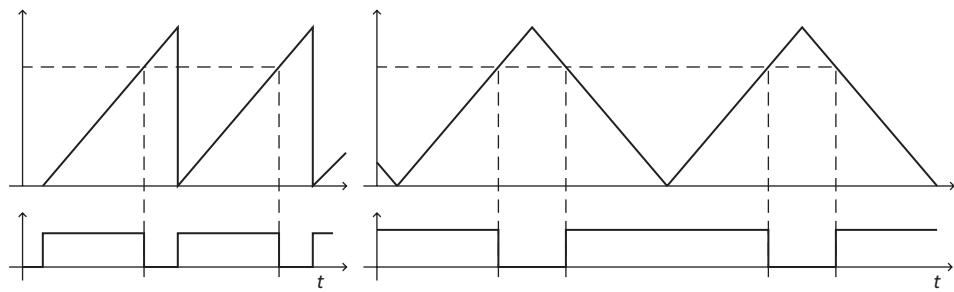
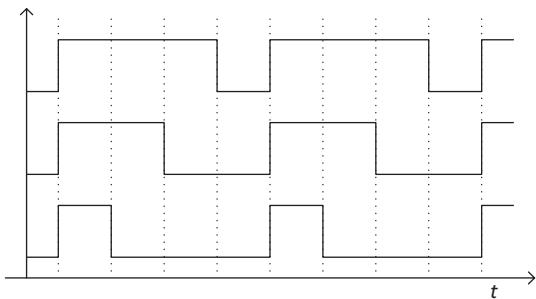
$$\bar{y} = D \cdot y_{max} \quad (3.5)$$

und damit proportional zum Tastverhältnis.

---

<sup>7</sup>Verhältnis zwischen Einschaltzeit und Gesamtzeit eines Rechteckssignals.

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



**Abb. 3.10** Pulsweitenmodulation (links (a) schnell, rechts (b) phasen- und frequenzrichtig)

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.

Im linken Bild von Abb. 3.10 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 oder 255), 0x01FF (9 Bit oder 511), 0x03FF (10 Bit oder 1023) oder 0xFFFF (16 Bit oder 65535) betragen oder durch den Wert im Register OCR1A oder auf weitere Arten festgelegt werden<sup>8</sup>. 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 Timerüberlauf und Rücksetzen auf 0 wird der Ausgang wieder gesetzt (Nichtinvertierender Modus).

<sup>8</sup> Im Sinne der Lesbarkeit wird hier auf eine vollständige Darstellung verzichtet. In [1] und [7] sind ausführliche Beschreibungen.

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

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

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. 3.10 rechts). 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:

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

Der folgende Code zeigt die Initialisierung von Timer 1 für ein phasenrichtiges PWM-Signal an Ausgang OC1A mit acht 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 schnellem (englisch *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 Kommutierungsströme 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)“.

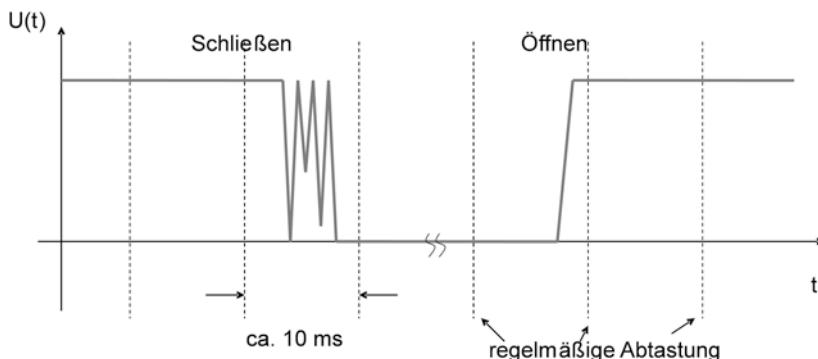
Diese Ereignisse lassen sich nicht einfach durch einen „pin change“ erfassen, denn Tasten „prellen“. Das heißt, nach dem mechanischen Auslösen kann es geschehen, dass das Potenzial am angeschlossenen Pin mehrfach um die Auslöseschwelle schwankt und damit der binäre Wert des Pins zwischen den Werten „0“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

Um die Ereignisse „pressed“ und „released“ sicher zu detektieren, kann also kein Pin change interrupt herangezogen werden, vielmehr müssen im Abstand von ca. 5 bis 10 ms die Tastenzustände überprüft werden und dann aufgrund der Änderung bestimmt werden, ob eine Taste aktuell gedrückt wurde oder nicht. Auf diese Weise können auch leitungs- oder feldbedingte Störimpulse ausgeblendet werden.

```
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 folgenden Tab. 3.18 sind die Timer-Modi in Abhängigkeit der Bits WGM13 und WGM12 im Register TCCR1B und WGM11 bzw. WGM10 im Register TCCR1A zusammengefasst.

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 einmal mehr auf das Datenblatt verwiesen ([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 Ganzzahltyp den Verlauf des (integrierten) Ausgangssignals darstellt. Ein Quadrant eines Sinussignals ( $0\dots90^\circ$  bzw.  $0\dots\pi/2$ ) mit einer Auflösung von 64 Zeitschritten und 128 Werteschritten sieht beispielsweise so aus:

**Tab. 3.18** Timer-Modi des Timers 1

WGM13	WGM12	WGM11	WGM10	Modus	
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

```
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, 122,
    123, 124, 125, 125, 126, 126, 126, 127, 127, 127};
```

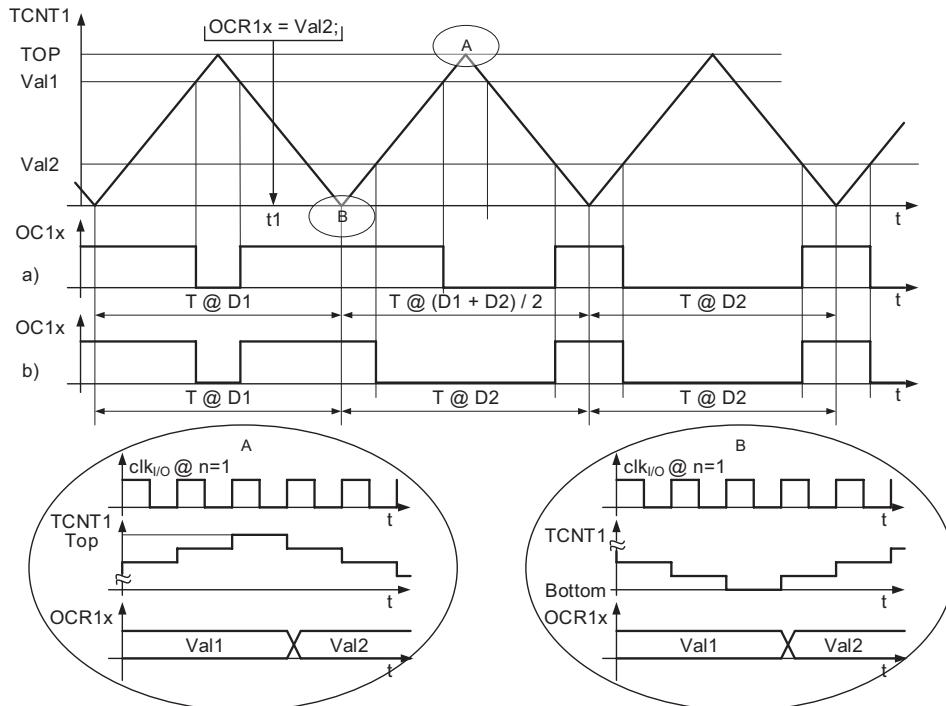
Um ein nach Integration vollständiges Sinussignal zu erhalten, addiert man für den ersten Quadranten mit jedem Ablauf von Timer 0 den Wert am dem Index der Tabelle, der dem jeweiligen Zeitpunkt entspricht, auf 128 (entspricht 50 % Tastverhältnis) auf. Im zweiten Quadranten zählt man die Tabelle rückwärts. 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:

$$f_{\sin} = \frac{f_{\text{timer}0}}{256} \quad (3.8)$$

In der Interrupt-Serviceroutine von Timer 0 (dem Abtasttimer) steht folgender Code<sup>9</sup>:

---

<sup>9</sup>Wir nehmen zunächst an, dass die Variable ucMult = 1 ist.



a) Phasenrichtiger PWM-Modus  
b) Phasen- und frequenzrichtiger PWM-Modus

**Abb. 3.12** Unterschied zwischen phasenkorrektem und phasen- und frequenzkorrektem PWM bei AVR

```

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
    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:

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

Damit das PWM-Signal vollständig erzeugt wird, muss Timer 2 theoretisch mindestens einen vollständigen PWM-Impuls pro Abtastschritt ausgeben. Praktisch sollten es etwa drei bis fünf 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  $f_{\text{max}}$  bandbegrenztes Signal mit mindestens  $2 \cdot f_{\text{max}}$  abgetastet werden muss, um es aus dem zeitdiskreten

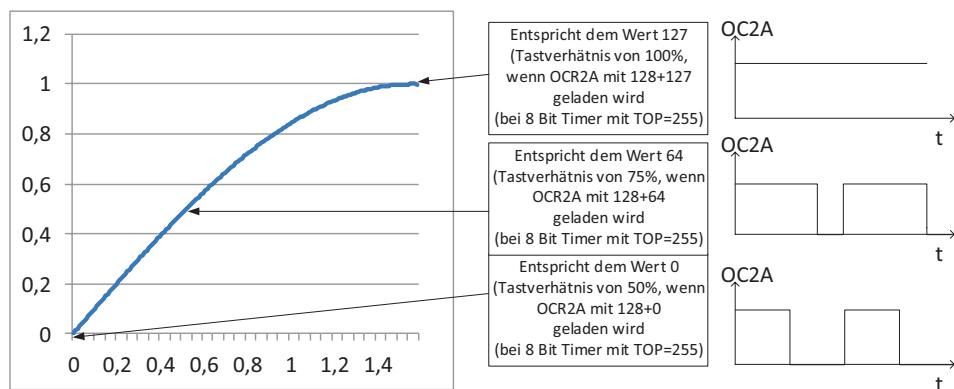


Abb. 3.13 Sinus-Viertelwelle aus einem PWM-Signal

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  $128 \cdot 3 \cdot 4 \text{ kHz} = 1,536 \text{ MHz}$  zu takten. Aus der Diskretisierung mit 7 Bit ergibt sich ein theoretischer Signal-Rauschabstand von  $7 \cdot 6 \text{ dB} = 42 \text{ dB}$  (also ein Verhältnis von Signal zu Diskretisierungsrauschen von 1:2<sup>14</sup>). Damit sind die Grenzen des Verfahrens durch den begrenzten Prozessortakt schnell erreicht, daher werden in der Praxis andere Modulationsarten (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. 22) lässt sich so beispielsweise ein Anrufbeantworter oder eine Sprachausgabe realisieren.

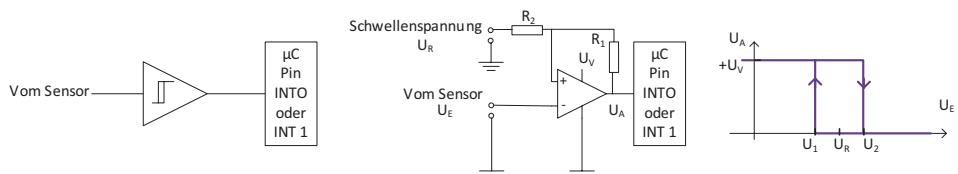
### 3.7.2.6 Laufende Zeitmessung zwischen Eingangsimpulsen

Eine weitere interessante Anwendung ist die laufende Zeitmessung zwischen zwei Eingangsimpulsen. Hiermit kann man beispielsweise einen Tachometer realisieren, eine Frequenz messen oder auch die Phase zwischen zwei Spannungsverläufen. Grundsätzlich sollte das Signal geformt sein, das heißt, in der Hardware ist ein Schmitt-Trigger vorzusehen, sofern der Sensor keine sauberen Pegel ausgibt. Ein Beispiel für einen invertierenden Schmitt-Trigger ist in Abb. 3.14 zu sehen. Die Schwellspannung  $U_R$  wird beispielsweise über ein Poti eingestellt oder fest auf die halbe Versorgungsspannung gelegt. Die Schalten hat einen vergleichsweise hohen Eingangswiderstand, benötigt aber einen zusätzlichen Inverter. Nähert sich die Eingangsspannung  $U_E$  dem oberen Hysteresewert  $U_2$  von unten, springt die Ausgangsspannung  $U_A$  von der Versorgungsspannung  $U_V$ , also 5 V oder 3,3 V auf 0 V. Nähert sie sich dem unteren Hysteresewert  $U_1$  von oben, springt  $U_A$  wieder auf die Versorgungsspannung.  $U_R$  wird wie folgt ausgelegt:

$$U_R = U_V \frac{U_1 + U_2}{2U_V + U_2 - U_1} \quad (3.10)$$

und  $R_1$  und  $R_2$  sind wie folgt ins Verhältnis zu setzen:

$$R_1 = \frac{U_V - U_1}{U_1 - U_R} R_2 \quad (3.11)$$



**Abb. 3.14** Schmitt-Trigger zur Signalformung vor einem Pin mit Flankenerkennung

Möchte man die Hysterese nicht selbst einstellen, kann ein Schmitt-Trigger auch mit einem IC aus der Familie 74x14 realisiert werden, der sechs dieser Schaltungen enthält.

Die Idee des folgenden Programmbausteins ist, die Zeitmessung mit der fallenden (oder steigenden, je nach Belieben) Flanke eines Ereignisses zu starten und mit der nächsten entsprechenden Flanke auszulesen und wieder neu zu starten. Die Zeitmessung selbst wird durch einen Timer realisiert. Hier ist eine sorgfältige Planung vorab notwendig. Man kann beispielsweise durch den Vorteiler eine passende Frequenz einstellen und die verstrichene Zeit direkt aus dem Zählerregister TCNTx ablesen. Je nach eingesetztem Quarz muss man anschließend noch die Zeit umrechnen. Die Zeit in  $\mu\text{s}$  lässt sich beispielsweise mit einem 8 MHz Quarz und einem Vorteilverhältnis von 8 einstellen. Bei längeren Zeiten empfiehlt es sich, die Zeit mit einer Variablen über den Timerinterrupt zu messen. Im Folgenden sind zwei Beispiele realisiert. Die Messung der Phasenlage zweier 50 Hz Sinussignale und die Messung zwischen zwei Impulsen in Millisekunden.

### 3.7.2.6.1 Zeitmessung im Millisekundenbereich

Hierfür benötigen wir einen Timerinterrupt pro Millisekunde. Im Interrupt wird dabei eine globale Variable hochgezählt (Softtimer). Bei jedem externen Interrupt (Pinchange) wird der Wert dieser Variable umgespeichert und sie wird zurückgesetzt.

Hier wird Timer 1 verwendet und so eingestellt, dass pro Millisekunde ein Interrupt erfolgt:

```
void Timer1_Init(void)
{
    TCCR1B |= 1<<CS10 | 1<<CS12 | 1<<WGM12; //Prescaler 1024 ->
    18 kHz free running
    OCR1A = 17;
    TIMSK1 |= 1<<OCIE1A;
}
```

Im Timerinterrupt wird dann einfach eine Variable hochgezählt

```
ISR(TIMER1_COMPA_vect)
{
    cntval++;
}
```

In unserem Fall wird das zu messende Signal an den Pin des INT0 gelegt und ein Interrupt ausgelöst:

```
ISR (INT0_vect)
{
    Timer1_Reset();
    sssSlopeDetected=1;
}
```

Dieser Interrupt setzt die Zählvariable `cntval` auf 0 zurück und setzt zugleich ein Flag, das im Hauptprogramm ausgelesen werden kann.

```
void Timer1_Reset(void)
{
    TCNT1=0;
    zaehler=cntval;
    cntval=0;
}
```

Auf das Flag `SlopeDetected` kann im Programm in der Hauptschleife die Variable `zaehler` ausgelesen werden. Das Timerregister `TCNT1` wird an der Stelle auf 0 gesetzt, da der Interrupt ja asynchron ausgelöst wird und das Register dazwischen ja einen Wert zwischen 0 und 17 annehmen kann, sodass sich ein kleiner Messfehler ergeben könnte.

### 3.7.2.6.2 Messung der Phase zweier Sinussignale (Wirk- und Blindleistungsmessung)

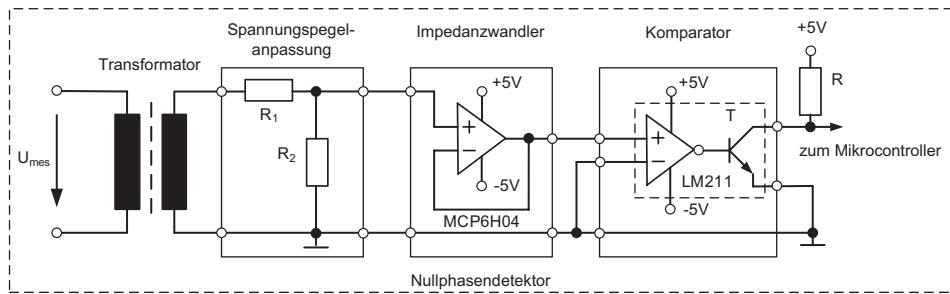
Die Phasenlage zweier 50 Hz Sinussignale ist dann von Interesse, wenn man bei induktiven Verbrauchern die Wirk- und die Blindleistung trennen will, beispielsweise um nur die Wirkleistung abzurechnen. Sind nämlich Strom- und Spannung phasenverschoben, setzt sich die mit den Effektivwerten gemessene Scheinleistung  $S$  aus der Wirkleistung  $P$  und der lediglich zwischen Verbraucher und Erzeuger hin- und herpendelnden Blindleistung  $Q$  zusammen, wobei gilt:

$$P = U \cdot I \cos \varphi \quad (3.12)$$

wobei  $\varphi$  der Phasenwinkel ist. Man muss also lediglich die Spannung auf ein normiertes Maß transformieren und den Strom idealerweise mit einer Wandlerspule in eine Spannung umwandeln, beide auf elektronischem Wege in Effektivwerte umwandeln und analog messen (siehe Abschn. 3.8.5). In diesem Abschnitt besprechen wir die Phasenmessung, die sich dadurch ergibt, dass man den Nulldurchgang der Spannung zum Start eines Timers und den Nulldurchgang des Stroms zum Stoppen der Zeit und damit des Phasenwinkels nutzt.

Bei 50 Hz ( $T=20$  ms) entspricht  $1^\circ$  des Phasenwinkels genau  $55,5 \mu\text{s}$  also 18 kHz. Mit wenig Aufwand kann man also den Phasenwinkel auf  $1^\circ$  genau messen, was bei kleinen Phasenwinkeln einer Genauigkeit beim Cosinus von unter 0,1 % bis maximal 1,7 % entspricht.

Setzt man einen 18,432 MHz Quarz ein, benötigt man gar keinen Softwarezähler sondern konfiguriert den Vorteiler des Timers auf 1024, sodass der Timer mit 18 kHz angesteuert wird. Mit einem Pin löst der Nulldurchgang der Spannung einen Interrupt aus, der das `TCNTx`-Register auf null setzt, mit einem zweiten Pin löst der Nulldurch-



**Abb. 3.15** Nullphasendetektor

gang des Stroms einen Interrupt aus, der das TCNTx-Register ausliest. Dieser Wert ist dann exakt der Phasenwinkel in Grad (DEG).

Um den Nulldurchgang eines Sinussignals zu detektieren, kann eine Schaltung verwendet werden wie in Abb. 3.15 dargestellt.

Die zu messende Spannung  $U_{\text{mess}}$ , die in der Regel größere Amplituden als die Versorgungsspannung der Schaltung hat, wird mit einem Transformator heruntertransformiert; bei Bedarf wird der Spannungspegel zusätzlich angepasst. Der Transformator gewährleistet auch die galvanische Trennung zwischen der Messspannung und der Mikrocontroller-Schaltung. Durch die Spannungspegelanpassung darf die Phase der Spannung nicht geändert werden, deshalb muss ein Spannungsteiler bestehend aus  $R_1$  und  $R_2$  verwendet werden um das gewünschte Verhältnis zwischen der Originalspannung und dem durch den Mikrocontroller gemessenen Wert einzustellen.

Der folgende Impedanzwandler, der mit einem Rail-to-Rail-Operationsverstärker realisiert ist, soll die Belastung des Spannungsteilers und weitere Spannungsabfälle verhindern.

Der Nulldurchgang der Spannung wird mit einem schnellen Komparator detektiert, dessen invertierter Ausgang einen Open-Collector Transistor steuert. Der Zugriff auf den Emitter und Kollektor des Ausgangstransistors T ermöglicht über die vorgestellte Beschaltung die Erzeugung von TTL-kompatibel Spannungspegeln, die direkt auf den Mikrocontrollereingang geschaltet werden können. Das Signal wird beispielsweise auf den Eingang INT0 gegeben. Mit dem Strom verfährt man analog: Die Praxis hat gezeigt, dass ein induktiver Stromwandler für eine Messschaltung dieser Art am besten geeignet ist, die Messspule liefert eine kleine Spannung, die proportional (also auch sinusförmig) zum zu messenden Strom verläuft. Man kann diese Spannung hochtransformieren, was zusätzlich eine galvanische Trennung sicherstellt. Danach muss sie komplett analog zum Spannungsmesskreis behandelt werden. Die Phasendetektion wird dann auf den Eingang INT1 gelegt.

Softwareseitig wird der Timer, der die Zeit zwischen den Nulldurchgängen zählt, per Prescaler auf 18 kHz gestellt, bei der ISR von INT0 zurückgesetzt und bei der ISR von INT1 ausgelesen.

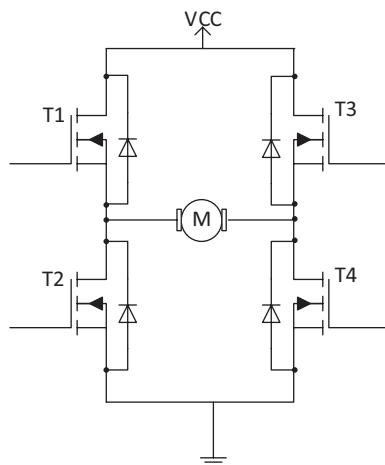
```
ISR (INT0_vect)
{
    TCNT1=0;
}

ISR (INT1_vect)
{
    zaehler=TCNT1;
    MeasurementComplete=1;
}
```

Auf diese Weise erhält man bei 50 Hz direkt den Phasenwinkel in Grad in der Variable `zaehler`. Auf das Flag `MeasurementComplete` muss dieser nur noch ausgewertet werden, was in der Hauptschleife erfolgen kann. Den Cosinus bestimmt man am effektivsten über eine Tabelle (siehe Abschn. 3.7.2.5 und 3.8.4).

### 3.7.2.7 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. 3.16 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

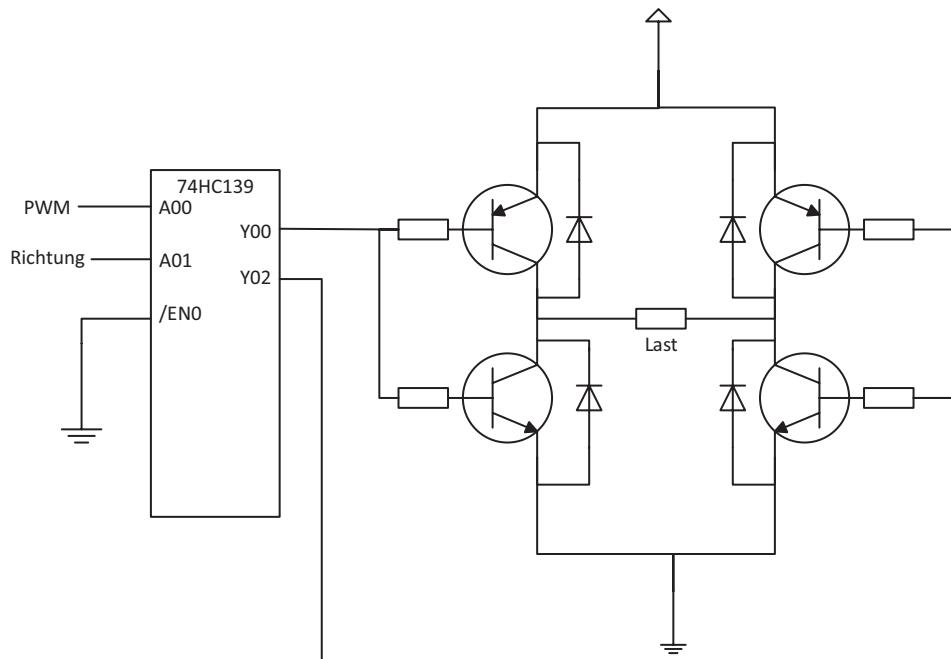


**Abb. 3.16** Vierquadrantensteller

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 sogenannten 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 Atmel/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.

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



**Abb. 3.17** H-Brücke mit Bipolartransistoren, bei induktiven Lasten mit Freilaufdioden

er mit der PWM-Frequenz umgepolzt 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). In Abbildung Abb. 3.17 ist eine einfache Schaltung mit Bipolartransistoren dargestellt. Die Freilaufdioden dienen auch hier dem Schutz der Transistoren vor Überspannungen beim Schalten von Motoren oder induktiven Lasten. Für einfache Modellbau-Minimotoren eignen sich Transistoren vom Typ BC327 (PNP) und BC337 (NPN), in diesem Fall sind Basiswiderstände um 1 K $\Omega$  und Freilaufdiode vom Typ 1N4148 verwendbar. Der 2–4-Dekoder 74HC139 dient der Absicherung der Brücke gegen Kurzschlüsse. An die Eingänge werden das PWM-Signal des Mikrocontrollers und ein digitaler Ausgang geschaltet, der die Richtung stellt.

## 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. 3.19):

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. 3.20 sind die entsprechenden Eingänge aufgelistet.

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

**Tab. 3.19** 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. 3.20** Bedeutung der MUXn

MUX3..0	Eingang	Pin beim ATmega88 im PDIP-Gehäuse	Pin beim ATmega88 im TQFP32-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. 3.21).

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-Serviceroutine des Analogkomparators ISR (ANALOG\_COMP\_vect) ausgeführt, dabei geht das Bit ACI solange auf 1 bis die Interrupt-Serviceroutine gestartet wurde. Der Modus der Ausführung wird von ACIS1 und ACIS0 gemäß Tab. 3.22 gesteuert.

**Tab. 3.21** 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. 3.22** 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  $\pm 2$  LSB<sup>10</sup> angibt, eine 8-Bit-Messung schöpft damit die Genauigkeit des Analogeingangs bereits hinreichend aus.

Eine vereinfachte Übersicht über den Aufbau des AD-Wandlers findet sich in Abb. 3.18.

Die Messtechnik basiert auf dem Prinzip der sukzessiven Approximation (schrittweisen Näherung), indem zunächst die halbe Referenzspannung<sup>11</sup>  $V_c = V_{REF}/2$  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 nächsten Schritt beträgt die Veränderung ein Achtel der Referenzspannung bis schließlich Vergleichsspannung und Eingangsspannung übereinstimmen<sup>12</sup>.

Wenn  $V_{in} < V_c \rightarrow V_c = V_c - V_{REF}/4$   
 sonst  $\rightarrow V_c = V_c + V_{REF}/4$   
 Wenn  $V_{in} < V_c \rightarrow V_c = V_c - V_{REF}/8$   
 sonst  $\rightarrow V_c = V_c + V_{REF}/8$   
 usw.

<sup>10</sup>LSB heißt „least significant bit“, also die niederwertigsten Bit.

<sup>11</sup>Hier wird aus Gründen der Kompatibilität mit dem Handbuch der Buchstabe V für die Spannung (Voltage) verwendet.

<sup>12</sup>Siehe auch Abschn. 20.2.3.

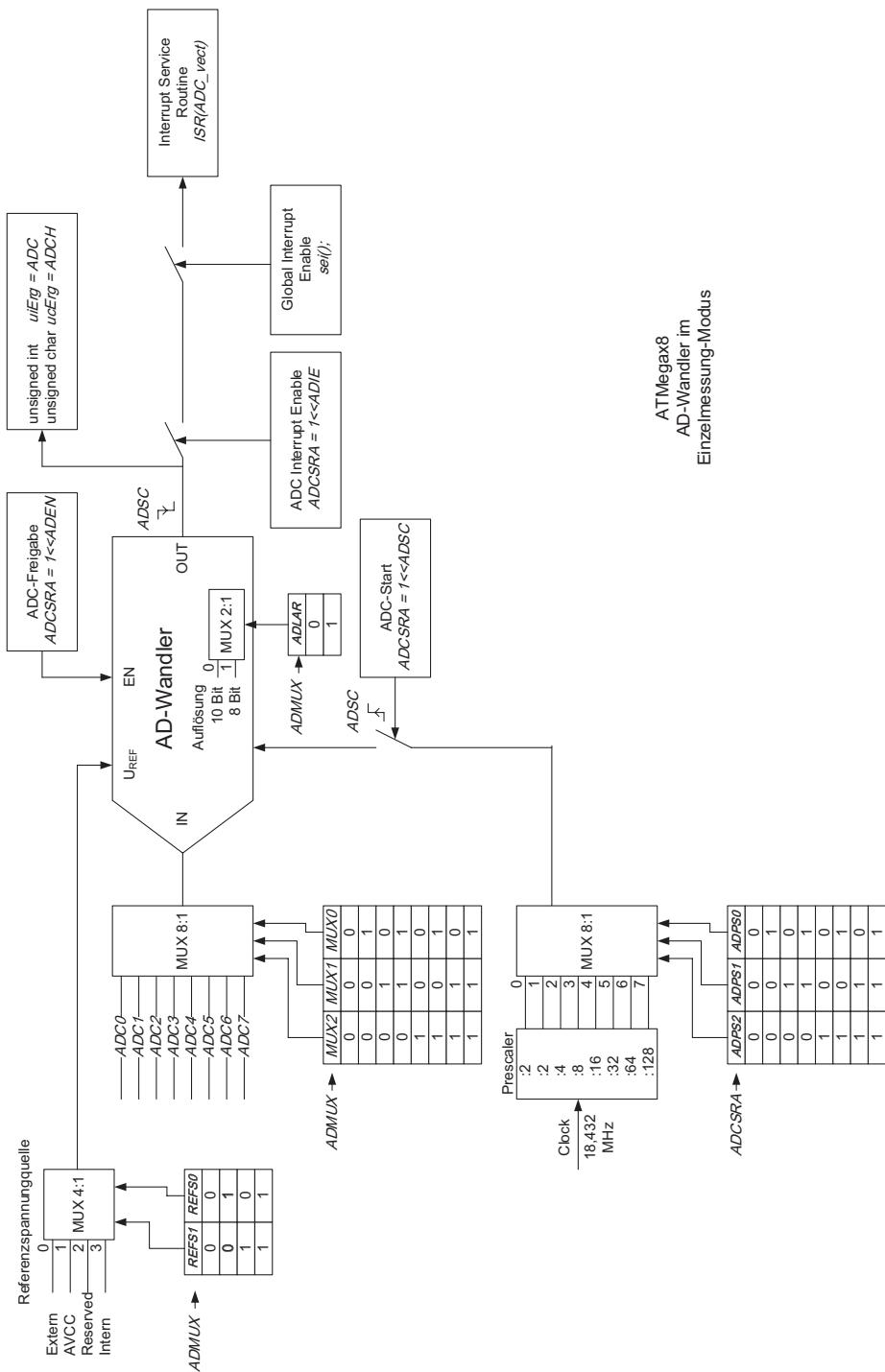


Abb. 3.18 Vereinfachter Aufbau des AD-Wandlers

Dies ist ein schrittweiser Prozess, der eine **Taktung** und damit auch **Zeit** benötigt. Der Takt wird aus einem eigenen Vorteiler generiert (s. u.) und sollte laut Hersteller zwischen etwa 50 kHz 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 (Vorteiler) auf 128 →  $f_{ADC} = 144 \text{ kHz}$ .

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

$$t_{Sample} = \frac{13}{144 \cdot 10^3} s = 9.027 \cdot 10^{-5} s \approx 90 \mu\text{s} \text{ bzw.} \quad (3.13)$$

$$f_{Sample} = \frac{144 \cdot 10^3}{13} \text{ Hz} \approx 11 \text{ kHz} \quad (3.14)$$

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

$$ADC = 1024 \frac{V_{in}}{V_{ref}} \quad (3.15)$$

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.

### 3.8.3.2 Triggern der Messung

Die Messung kann auf verschiedene Weisen gestartet werden. Im einfachsten Fall wird das ADC Start Conversion bit (ADSC) im ADC control and status register A (ADCSRA) gesetzt. Daraufhin wird unmittelbar eine Messung gestartet. Das Bit wird automatisch zurückgesetzt, wenn die Messung beendet ist. Durch Setzen des ADATE- Bits (ADC auto trigger enable) kann man eine automatische Triggerung der Messung durch verschiedenste Quellen erreichen. Im so genannten „Free running“-Modus wird eine neue Messung durch das ADC Interrupt Flag getriggert, das bei Beendigung einer Messung („conversion complete“) durch die Hardware gesetzt wird. Damit erreicht man, dass nach Beendigung sofort eine neue Messung gestartet wird. Weitere Quellen sind der externe Interrupeingang 0, der Analogkomparator und verschiedene Ereignisse der Timer 0 und 1. Diese finden sich in Tab. 3.25.

### 3.8.3.3 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 folgender Tab. 3.23 eingestellt:

Eine interessante Variante bietet ADLAR (ADC Left Adjust Result) im selben Register (Tab. 3.19): Ist das Bit auf 1 gesetzt, wird das Ergebnis (10 Bit) im 16 Bit Register links ausgerichtet, d. h. das MSB<sup>13</sup> 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. 3.19 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:

ADCSRA (siehe Tab. 3.24):

- 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.

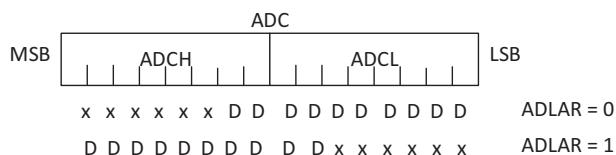
---

<sup>13</sup>MSB: Most significant bit, also die höherwertigsten Bits.

**Tab. 3.23** Auswahl der Referenzspannungsquelle

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

**Abb. 3.19** Bedeutung des left alignment im AD Wandler



**Tab. 3.24** 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. 3.25** 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 Match A
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

- 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-Serviceroutine so aus:

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

**Tab. 3.26** 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

- ADPS2, ADPS1, ADPS0 (ADC Prescaler Select Bits): Legen das Teilverhältnis des Verteilers und damit die Messgeschwindigkeit fest: 000 ist halber Quarztakt, 010 viertel, 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. 3.26) 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. 3.25).

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. 3.27). Der korrespondierende Digitaleingang wird durch Setzen einer 1 abgeschaltet (disable).

### 3.8.3.4 ADC-Initialisierung

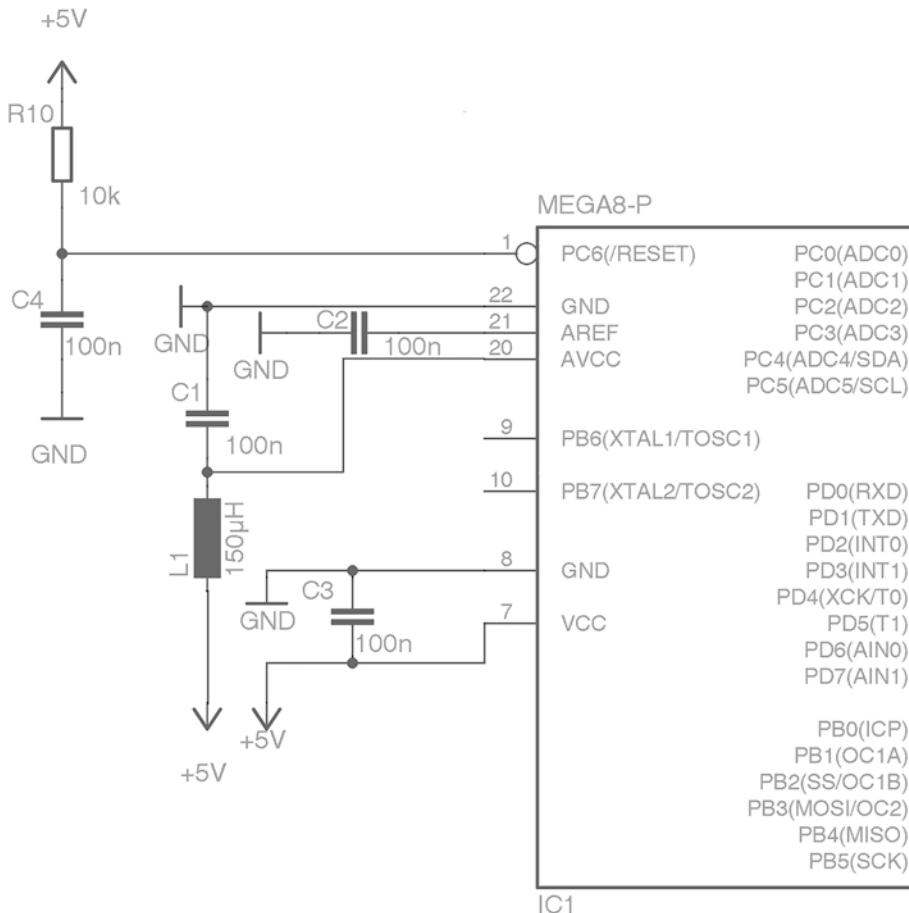
Um den ADC des µECU zu initialisieren, nehmen wir zunächst das Szenario an, am Eingang ADC4 eine freilaufende Messung zu starten.

Zudem soll die Versorgungsspannung nach Abb. 3.20 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 Verteilverhältnis von 1/128 ( $144\ 000 * 128 = 18\ 432\ 000$ ). Damit ist die Initialisierung:

**Tab. 3.27** 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



**Abb. 3.20** Beispiel für die Verwendung der Versorgungsspannung als externe Referenzspannung

```

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) und man muss sie verwerfen.

In diesem Code wird sofort nach dem Start der Messung der Interrupt „scharf“ geschaltet. Damit liefert natürlich auch der erste Aufruf der Interrupt-Serviceroutine einen falschen Messwert. In der Regel ist dies nicht weiter von Bedeutung, falls dennoch, sollte der ADC-Interrupt erst freigegeben werden, wenn die erste Messung sicher beendet ist. Dies kann in der Initialisierungsphase beispielsweise durch eine Verzögerungsschleife passieren.

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.

### 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. 3.21 gezeigte einfache Schaltung aufbauen. Selbst der Quarz wird eingespart, über Fuses, wird der interne Oszillatator 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 kHz und 200 kHz liegt, gemäß Gl. 3.13 und 3.14 beträgt sie dann 125 kHz.

Heißleiter haben eine Kennlinie, die der folgenden Funktion genügt:

$$R_T = R_R \cdot e^{B \cdot \left( \frac{1}{T} - \frac{1}{T_R} \right)} \quad (3.16)$$

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:

$$B = \frac{T \cdot T_R}{T_R - T} \ln \frac{R_T}{R_R} \quad (3.17)$$

In der gezeigten Schaltung wird der NTC in einem Spannungsteiler verwendet, der bei  $25^\circ\text{C}$  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. 3.15 ausgegeben wird.

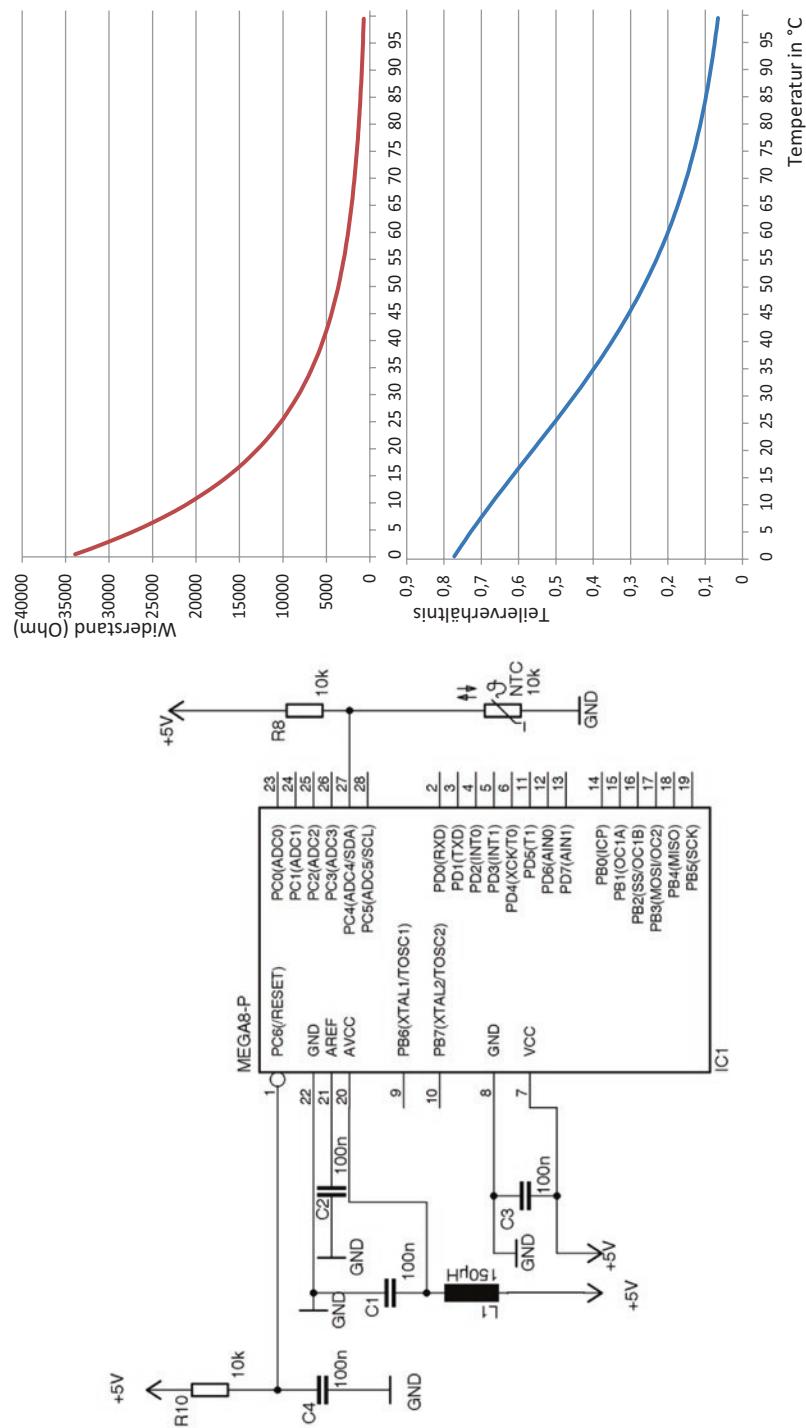


Abb. 3.21 Schaltung einer einfachen Temperaturmessung mit NTC mit typischen Kennlinien

Um nun aus der angelegten Spannung auf die Temperatur zu schließen, kann man folgendes Verfahren anwenden. Man definiert eine Tabelle (lookup table), 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};
```

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;
    }
}
```

### 3.8.5 Beispiel: Effektivwertmessung an einer Sinusspannung

Um eine Wechselspannung mit dem ADC zu messen, beispielsweise für eine Leistungsmessung nach Abschn. 3.7.2.6), könnte man diese mit hoher Frequenz abtasten und bei bekanntem Verlauf (beispielsweise Sinus) auf die Parameter Spitzenwert und Effektivwert mathematisch rückschließen. Nach dem Nyquist-Kriterium müsste man das Signal mit mindestens der doppelten Grundfrequenz abtasten um diese Werte zu erhalten, das funktioniert aber in der Praxis nicht, da der Abtastzeitpunkt genau in den Phasenwinkeln 90° und 270° erfolgen müsste (siehe auch Abschn. 14.1). Praktisch müsste man das Signal also erheblich überabtasten um daraus mathematisch den Verlauf und damit den Effektivwert korrekt zu erhalten.

Der Effektivwert ist deshalb so wichtig, weil er eine konstante Spannung bezeichnet, die denselben Leistungsumsatz an einem ohmschen Verbraucher erzeugt, wie die gemessene Wechselspannung. Die Leistung an einem ohmschen Verbraucher:

$$P = UI = U \frac{U}{R} = \frac{U^2}{R} \sim U^2 \quad (3.18)$$

ist dem Quadrat der Spannung proportional, die Leistung einer Wechselspannung ist der Mittelwert der Leistung aus dem zeitlich veränderlichen Verlauf:

$$P = \overline{u(t) \cdot i(t)} = \frac{1}{T} \int_{t_0}^{t_0+T} \frac{u(t)^2}{R} dt \quad (3.19)$$

Daraus folgt, dass der Effektivwert einer Spannung dem der Wurzel aus dem quadratischen Mittelwert der Spannung ermittelt werden kann:

$$U_{eff} = \sqrt{\overline{u(t) \cdot i(t)}} = \sqrt{\frac{1}{T} \int_{t_0}^{t_0+T} \frac{u(t)^2}{R} dt} = \sqrt{\overline{u(t)^2}} \quad (3.20)$$

Mit etwas Nachrechnen ist bei exakt sinusförmigem Verlauf der Effektivwert

$$u(t) = \hat{U} \sin \omega t \quad U_{eff} = \frac{\hat{U}}{\sqrt{2}} \quad (3.21)$$

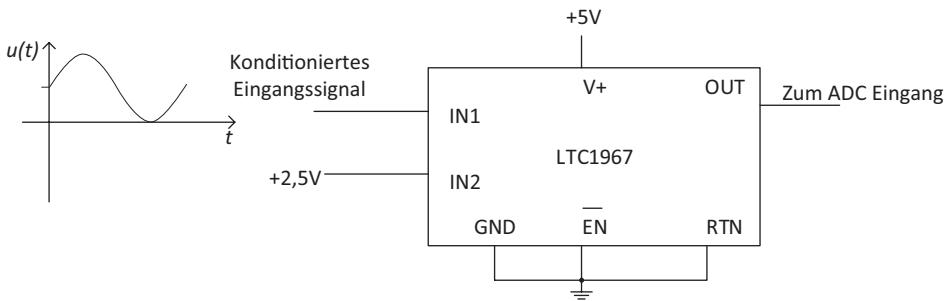
Technisch handhabbar wird eine Effektivwertmessung am besten mit einer externen Beschaltung beispielsweise dem IC LTC1967 von Linear Technology [6]. Der Chip berechnet den echten Effektivwert (RMS = root mean square) einer Spannung von beliebigem Verlauf. Abb. 3.22 zeigt die prinzipielle Beschaltung des Bausteins. In der Regel ist das Signal linear zu konditionieren, z. B. um aus dem bipolaren Signal ( $\pm \hat{U}$ ) ein unipolares zu machen ( $0 \dots 2\hat{U}$ ). Über den Eingang IN2 lässt sich dieser Offset wieder kompensieren. Softwaretechnisch ist keine weitere Vorkehrung zu treffen außer der Kalibrierung des gemessenen Wertes.

### 3.9 Power Management

Eingebettete Systeme müssen oftmals mit sehr geringen Energiereserven auskommen<sup>14</sup>. Wie die meisten Mikrocontroller besitzt auch die AVR-Familie verschiedene Mechanismen um den Energieverbrauch an die Rechenerfordernisse anzupassen. Insbesondere bei batteriebetriebenen Schaltungen, die nicht explizit ausgeschaltet werden können oder sollen, ist es sinnvoll, 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, sodass 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

---

<sup>14</sup>Siehe auch Abschn. 6.2.



**Abb. 3.22** Beschaltung des LTC1967 nach [6]

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, sodass 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 aufgrund 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 der Power-down-Modus 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 (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. 3.28).

Durch Setzen einer 1 in eines der Felder werden die Two-Wire-Interface (TWI) Schnittstelle (Bit7), 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. Gleicher gilt für power\_twi\_enable() und power\_twi\_disable() und andere. Hier empfiehlt sich die Lektüre von [3].

**Tab. 3.28** 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

## 3.10 Internes EEPROM

Das interne EEPROM<sup>15</sup> dient dem Zweck, Daten persistent, also dauerhaft und unabhängig von der Energieversorgung des Prozessors abzulegen. Diese überstehen dann auch einen 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 Fehlerspeichereinträge und Parametrierungen beziehungsweise Anpassungen von Funktionen, die nicht im Quellcode stehen sollen, wie beispielsweise Reglerparameter, Seriennummern, Anlernparameter, Kennfelder. Da 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 wegen seiner geringen Größe nicht besonders gut, hier empfiehlt sich der Einsatz eines externen Flash-Speichers, dazu finden Sie Hinweise im Kap. 22.

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 zur Laufzeit einzulesen. Dazu wird das EEPROM über die Programmierschnittstelle beschrieben und im Code werden die Daten gelesen und verwendet.

---

<sup>15</sup>Electrically Erasable Programmable Read-Only Memory: Nichtflüchtiger, programmierbarer Speicher.

Beispielsweise können sich damit Funktionen zur Laufzeit freischalten lassen, Zeitkonstanten könnten angepasst oder das Verhalten von Reglern beeinflusst 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/lib.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. 3.23 dargestellt.

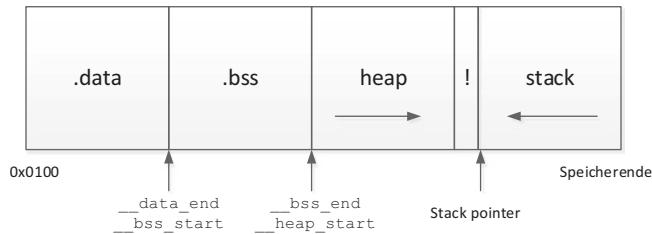
Der AVR-C-Compiler kennt vier verschiedene Arbeitsspeicher [3, 4]:

- 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";
```

- 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-Serviceroutinen 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



**Abb. 3.23** Aufbau des Speichers [3, 4]

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 allozieren<sup>16</sup> (anzulegen), wird die Funktion `malloc(size)` verwendet.

```
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.

<sup>16</sup> Allokieren meint wörtlich: „einen Ort zuweisen“.

```
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. Wenn diese Tabellen konstant sind (z. B. für die Umrechnung einer Sensorkennlinie 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 [4] 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 erheblich gerechtfertigt und schont den Speicherplatz im Arbeitsspeicher.

## Literatur

1. Microchip: Reference Manual ATmega48/168. <https://www.microchip.com/wwwproducts/en/ATmega88A>. Zugegriffen: 6. Jan. 2021.
2. Mikrocontroller.net. <http://www.mikrocontroller.net>. Zugegriffen: 20. Dez. 2020.
3. NONGNU: AVR Libc. <http://www.nongnu.org/avr-libc/user-manual/index.html>. Zugegriffen: Aug. 2020.
4. AVR Libc: <http://savannah.nongnu.org/projects/avr-libc/> – zuletzt abgerufen August 2020, auch bei <https://onlinedocs.microchip.com/> erhältlich.
5. Ansgar Meroth, Boris Tolg (2008) Infotainmentsysteme im Kraftfahrzeug. Grundlagen, Komponenten, Systeme und Anwendungen. Vieweg, Wiesbaden
6. Linear Technology: LTC1967 Precision Extended Bandwidth, RMS-to-DC Converter, Datenblatt (z. B. bei <https://www.mouser.de/datasheet/2/609/1967f-1271505.pdf>). Zugegriffen: 20. Jan. 2021).

## Weiterführende Literatur

7. Herbert Bernstein: Mikrocontroller: Grundlagen der Hard- und Software der Mikrocontroller ATtiny2313, ATtiny26 und ATmega32–2. Auflage Springer 2020.
8. Günter Schmitt: Mikrocomputertechnik mit Controllern der Atmel AVR-RISC-Familie – 5. Auflage De Gruyter 2011
9. Gaicher, H., & Gaicher, P. (2016). *AVR Mikrocontroller – Programmierung in C: Eigene Projekte selbst entwickeln und verstehen* (1. Aufl.). Tredition.
10. Salzburger, L., & Meister, I. (2013). *AVR-Mikrocontroller-Kochbuch* (1. Aufl.). Franzis.
11. Spanner, G. (2010). *AVR-Mikrocontroller in C programmieren: Über 30 Selbstbauprojekte mit ATtiny13, ATmega8, ATmega32 (PC & Elektronik)* (1. Aufl.). Franzis.
12. Elliot Williams: Make: AVR Programming: Learning to Write Software for Hardware – 1. Auflage, O'Reilly and Associates, Februar 2014
13. Florian Schäffer: AVR: Hardware und Programmierung in C – Überarbeitete und erweiterte Neuauflage, Elektor, Dezember 2014



# Software Framework

4

## Zusammenfassung

In diesem Kapitel wird die grundsätzliche Architektur eines Messsystems vorgestellt, insbesondere die Hardware-Abstraktion.

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. Daneben enthält das Kapitel einige Betrachtungen zur Zeitsteuerung der Software, mithin ein einfaches Multitasking-Schema.

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/ATMEL hat hierzu eine eigene Applikationsschrift [3] herausgegeben.

---

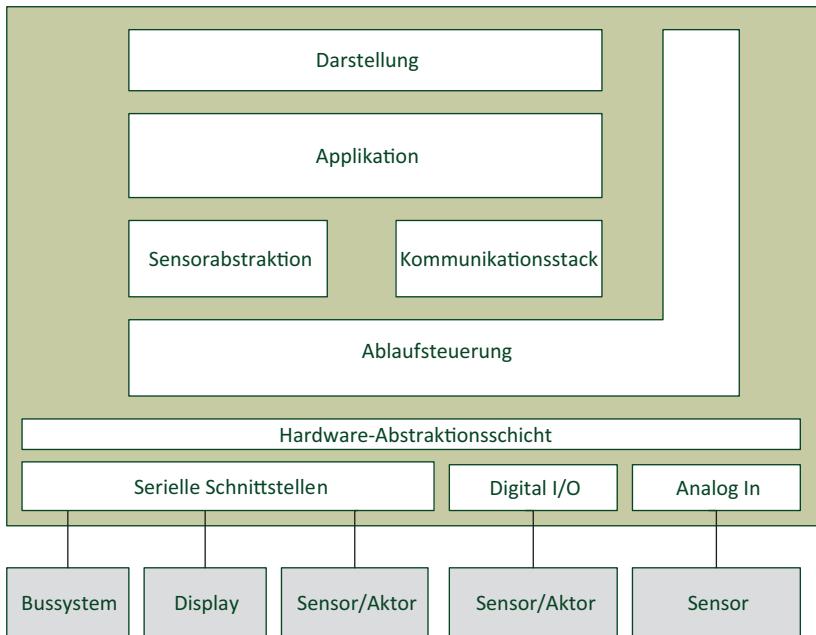
## 4.1 Sichten

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, Helligkeitssensoren, Näherungssensoren, Beschleunigungssensoren mit Analogausgang), an einer digitalen Schnittstelle (Präsenz- und Annäherungssensoren, End- oder Schwellwertschalter) oder an einer der seriellen Schnittstellen angeschlossen, die in den folgenden Kap. 7 bis 11 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 Filterung) in eine eigene Sensorabstraktionsschicht ausgelagert werden kann. Das Ergebnis der Verarbeitung steht dann entweder an einem Bussystem als Nachricht zur Verfügung (Kap. 7–11), wird auf einem Display angezeigt (Kap. 25) 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, deren 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 LED1:

```
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. 5.1.1 ff.) au7s 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. 5.2 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 10 ms, 50 ms und 100 ms benötigt, außerdem soll eine „Stoppuhrfunktion“ mitteilen, wie viel Zeit seit dem letzten Aufruf der Funktion in 100 ms verstrichen ist. In einem Modul Timer.c werden folgende globalen Variablen deklariert:

```
unsigned char ucTimer1_Flag_10ms = 0; //wird 1, wenn 10 ms ver-
strichen sind

unsigned char ucTimer1_Flag_50ms = 0; //wird 1, wenn 50 ms ver-
strichen sind
unsigned char ucTimer1_Cnt_50ms = 0; //zählt die 1ms bis 50 ms ver-
strichen ist

unsigned char ucTimer1_Flag_100ms = 0; //wird 1, wenn 100 ms ver-
strichen sind
unsigned char ucTimer1_Cnt_100ms = 0; //zählt die 1ms bis 100 ms ver-
strichen 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 10 ms einen Interrupt auslöst, was bei einem 16-Bit-Timer bei 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-Task<sup>1</sup>. Gleicher 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 er meistens weggelassen.

---

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

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 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 zwischenzeitlich 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.

---

## Literatur

1. ISO/IEC/IEEE 42010 – Recommended practice for architectural description of software intensive systems. (2020). <http://www.iso-architecture.org/ieee-1471>. Zugriffen: 3. Jan. 2020
2. Starke, G., & Hruschka, P. (2011). *Software-Architektur kompakt* (2. Aufl.). Spektrum Akademischer Verlag
3. Microchip/ATMEL. (2016). AVR035: Efficient C coding for AVR. <https://www.microchip.com/wwwAppNotes/AppNotes.aspx?appnote=en590906>. Zugriffen: 2. Apr. 2021



# Speicherkonzepte und Algorithmen

5

## Zusammenfassung

In diesem Kapitel werden grundsätzliche Algorithmen wie Warteschlange, FIFO und Zustandsautomat eingeführt, die in vielen Sensorprojekten notwendig sind.

In diesem Kapitel werden einige wichtige Speicherkonzepte und Algorithmen dargestellt. Alle hier verwendeten Beispiele sind prozessorunabhängig und können auch in anderen Systemen angewendet werden. Für Anfänger sind nur wenige der im Folgenden beschriebenen Algorithmen geeignet, am ehesten die im Abschn. 5.1.1 genannten einfachen FIFOs und der erste der beiden Zustandsautomaten.

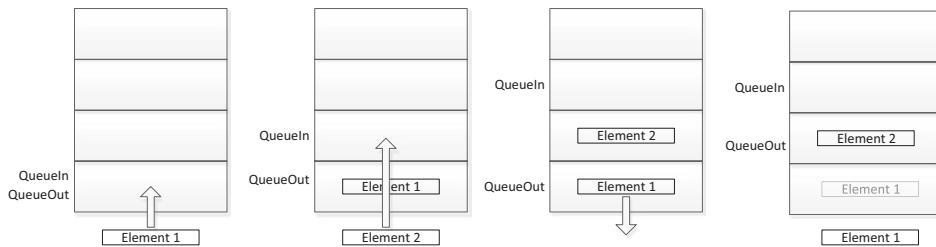
## 5.1 Wichtige Speicherkonzepte

Obwohl die Möglichkeiten der Prozessoren der AVR-Familie begrenzt sind, lohnt es sich dennoch auf einige wenige Speicherkonzepte zu schauen, diese sind ohnehin unabhängig vom verwendeten Prozessortyp. Speziell die Warteschlange ist dann von Bedeutung,

---

Die Originalversion dieses Kapitels wurde revidiert. Ein Erratum ist verfügbar unter  
[https://doi.org/10.1007/978-3-658-31709-6\\_27](https://doi.org/10.1007/978-3-658-31709-6_27)

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann  
[https://doi.org/10.1007/978-3-658-31709-6\\_5](https://doi.org/10.1007/978-3-658-31709-6_5).



**Abb. 5.1** FIFO-Prinzip

wenn es um Datenkommunikation geht, insbesondere, Datenpakete unregelmäßig (nicht isochron) anfallen. In diesem Kapitel werden schlanke Implementierungen vorgestellt, die leicht portier- und einsetzbar sind.

### 5.1.1 Warteschlangen und Ringpuffer (FIFO)

Warteschlangen<sup>1</sup> 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 Eingangsgrößen aus einem Netzwerk in einem isochronen Prozess<sup>2</sup> verarbeitet werden sollen (Puffer). Aufgrund der Zugriffsreihenfolge First in – First Out werden Warteschlangen auch FIFO genannt. Anwendungsbeispiele finden sich beispielsweise in Abschn. 9.3.5

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. 5.1 links zu sehen ist. Wird nun ein Element in den FIFO geschrieben, so 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

<sup>1</sup> Engl. *queue*.

<sup>2</sup>Von griech *ισος* = „gleich“: Abtastprozesse mit immer gleichen Abtastabständen.

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 „überrundet“, 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, in der Regel ein Byte.

```
typedef unsigned char 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. Damit ist eine wichtige Funktion des FIFO, das Einschreiben, realisiert:

```

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 aufwendige Abfrage nach der Arraygrenze kann man sich sparen, wenn man den Index über eine Modulo-Operation ermittelt [1]

```
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 [2]:

```
#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 [1] und [2] sehen kann.

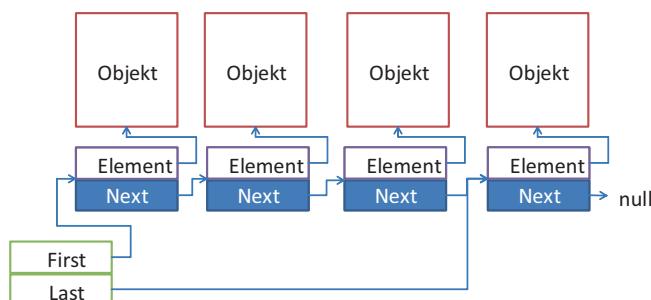
- Wenn gewünscht ist, dass die Warteschlange aus einem Interrupt gelesen oder beschrieben werden soll, ist es wichtig, dass alle Interrupts vor dem Zugriff auf die Queue deaktiviert werden!

### 5.1.2 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. 5.2). 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 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.



**Abb. 5.2** Aufbau einer einfach verketteten Liste

Würde der Zeiger `last` auf das erste Element zeigen, hätten wir wieder einen Ringpuffer oder Zyklus. In Abb. 5.2 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 zeigt 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);
```

### 5.1.3 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 erste 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);
...
```

### 5.1.4 Mehrere Warteschlangen mit unterschiedlichen Typen

Möchte man Warteschlangen unabhängig vom Typ der Inhalte verwalten, muss noch ein weiteres Konzept berücksichtigt werden. In diesem Fall kann man nämlich nicht direkt in die Warteschlange schreiben oder aus der Warteschlange lesen, weil die Zugriffsfunktionen den Typ des Elements berücksichtigen müssen (lesend als Rückgabeparameter, schreibend als Eingangsparameter). Hier hilft man sich durch eine Erweiterung des Handles mit Pointern auf speziell dafür geschriebenen Funktionen:

```
typedef void FIFO_t;
typedef struct sFIFOH {
    int QueueIn;
    int QueueOut;
    int QueueSize;
    void (*setfnct) (FIFO_t *element, int index);
    FIFO_t* (*getfnct) (int index);
    void *Fifo;
} FIFOH_t;
```

Der Typ `FIFO_t` ist ein Pointer auf `void`, der dann jeweils in den Zugriffsfunktionen explizit umgewandelt werden muss.

Bei der Initialisierung der Warteschlange werden die Zugriffsfunktionen also mit übergeben:

```
void QueueInit(FIFOH_t *Handle, int size, FIFO_t *Fifo,
               void (* setfnct) (FIFO_t *element,
                                 int index), FIFO_t* (* getfnct) (int index))
{
    Handle->QueueIn = 0;
    Handle->QueueOut = 0;
    Handle->QueueSize = size;
    Handle->Fifo = Fifo;
    Handle->getfnct = getfnct;
    Handle->setfnct = setfnct;
}
```

Die Getter- und Setterfunktionen (Zugriffsfunktionen auf die Arrays) ersetzen dann einfach die Indexklammern in der Queue, sodass die Queuefunktionen `put` und `get` wie folgt aussehen:

```

int QueuePut(FIFOH_t *Handle, FIFO_t *new_element)
{
    if ((Handle->QueueIn + 1 == Handle->QueueOut) ||
        (Handle->QueueOut == 0 &&
         Handle->QueueIn == Handle->QueueSize-1))
        return QUEUE_FULL ;
    Handle->setfnct(new_element, Handle->QueueIn);
    Handle->QueueIn++;
    if (Handle->QueueIn >= Handle->QueueSize) Handle->QueueIn= 0;
    return QUEUE_NOERROR; // No errors
}

FIFO_t* QueueGet(FIFOH_t *Handle, int *errcode)
{
    FIFO_t *result;
    if(Handle->QueueIn == Handle->QueueOut)
    {
        *errcode = QUEUE_EMPTY;
        return NULL;
    }
    result = Handle->getfnct(Handle->QueueOut);
    Handle->QueueOut++;
    if (Handle->QueueOut >= Handle->QueueSize) Handle->QueueOut= 0;
    *errcode = QUEUE_NOERROR;
    return result;
}

```

Der Aufruf könnte dann so aussehen:

```

typedef struct a {
    int a;
    float b;
} a_t;
FIFO_t* get_a(int index)
{
    return &Queue_a[index];
}
void set_a(FIFO_t *element, int index)
{
    Queue_a[index]=*(a_t*)element;
}
int main()
{
    ...
    a_t Queue_a[5];

```

```

FIFOH_t ha;
QueueInit(&ha, 5, Queue_a, set_a, get_a);
...
}

```

## 5.2 Zustandsautomaten (Statemachine)

Zustandsautomaten kommen im Programmiereralltag insbesondere im embedded-Bereich häufig vor. Oft wünscht man sich dabei ein Schema, das einfach abänderbar ist und wenige Ressourcen benötigt. Zwei dieser Schemen sind hier ausgeführt, ein sehr simples, das allerdings händisch für jeden Fall neu geschrieben werden muss und ein etwas komplexeres, das speziell bei größeren Zustandsautomaten angewendet werden kann und das dafür sehr gut lesbar ist.

### 5.2.1 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  $e_j$ :

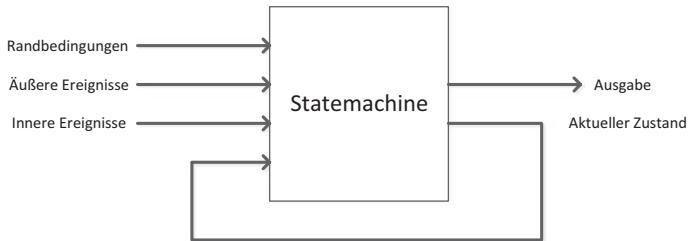
$$z_i^{n+1} = f(z_i^n, e_j) \quad (5.1)$$

und zusätzlich wird die Menge der aktuellen Zustände und Eingangsgrößen auf eine Menge von Reaktionen oder Ausgangsgrößen  $o_k$  abgebildet:

$$o_k = f(z_i^n, e_j) \quad (5.2)$$

Abb. 5.3 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.



**Abb. 5.3** Allgemeine Ausführung eines Zustandsautomaten

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

Besonders häufige Anwendungsfälle von Zustandsautomaten sind:

- Die Steuerung sequenzieller Abläufe, die auf externen Ereignissen beruhen
- Mensch-Maschine-Schnittstellen: Reaktionen auf Benutzereingaben und Maschinenereignisse
- 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)

### 5.2.2 Beschreibung von Zustandsautomaten

Für die Beschreibung von Zustandsautomaten wird entweder das in der UML (Unified Modeling Language) beziehungsweise SysML (Systems Modeling Language) definierte Zustandsdiagramm [3–5] (engl. state chart) genutzt oder die Zustandsübergangstabelle, auch Zustandsfolgetabelle genannt. Die beiden Techniken werden hier anhand des folgenden Beispiels vorgestellt.

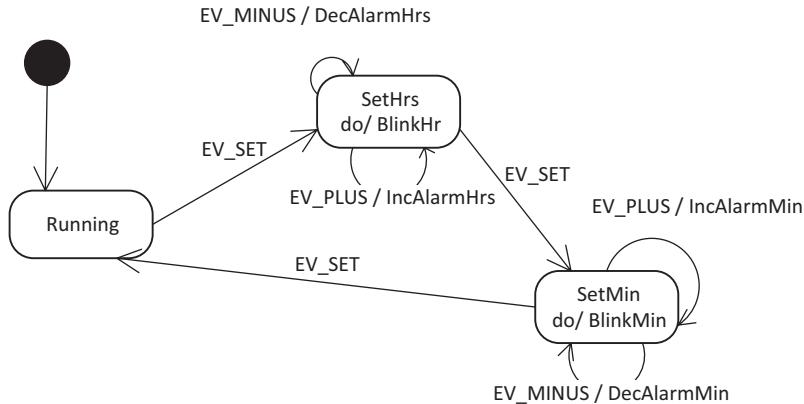
#### Beispiel

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. 5.4 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 äußere Transition auslöst, während EV\_PLUS und EV\_MINUS innere Transitionen auslösen.

In einer Zustandsfolgetabelle sieht dieselbe Beschreibung wie in Tab. 5.1 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. 5.4).



**Abb. 5.4** Zustandsautomat des Weckers für die Zeiteinstellung

**Tab. 5.1** Zustandsfolgetabelle für den Wecker

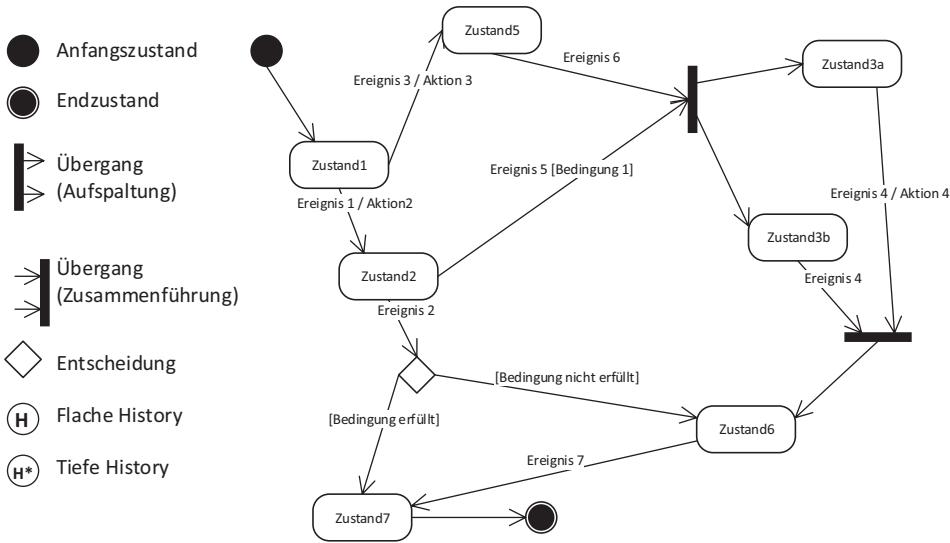
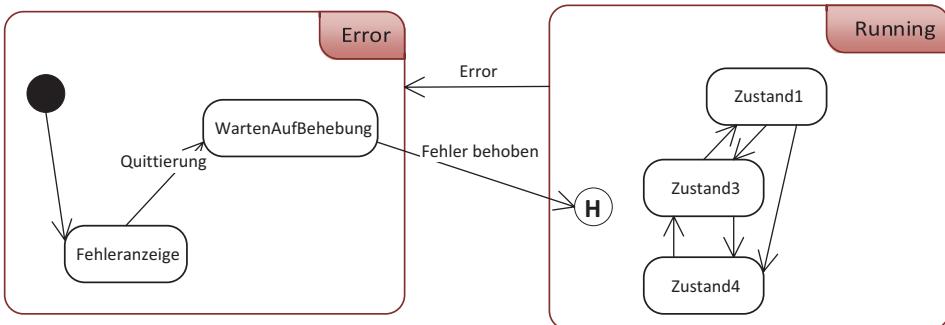
Zustand	Ereignis	Folgezustand	Aktion
Running	Set	SetHrs	
SetHrs	Plus	SetHrs	IncAlarmHrs
SetHrs	Minus	SetHrs	DecAlarmHrs
SetHrs	Set	SetMin	
SetMin	Plus	SetMin	IncAlarmMin
SetMin	Minus	SetMin	DecAlarmMin
SetMin	Set	Running	

In Abb. 5.5 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 State machine eingenommen wird
- Der Endzustand bezeichnet die Terminierung der State machine. Er besitzt keine ausgehenden Transitionen
- 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. 5.6 ist ein System im Überzustand „running“ und kann dort die Zustände 1 bis 3

**Abb. 5.5** UML-Sprachelemente des Zustandsdiagramms**Abb. 5.6** Hierarchische Zustandsautomaten

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 behoben 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.

### 5.2.3 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 angeibt, 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 abarbeiten 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. <https://stackoverflow.com/questions/215557/how-do-i-implement-a-circular-list-ring-buffer-in-c>. Zugegriffen: 27. Dez. 2020.
2. <https://www.mikrocontroller.net/articles/FIFO>. Zugegriffen: 27. Dez. 2020.
3. Booch, G., Rumbaugh, J., & Jacobson, I. (2006). *Das UML-Benutzerhandbuch, Aktuell zur Version 2.0*. Addison-Wesley. ISBN 978-3827325709.
4. Balzert, H. *UML2 in 5 Tagen*, 2008, W3L. ISBN 978.3937137612
5. Alt, O. (2012). *Modellbasierte Systementwicklung mit SysML*. München: Carl Hanser Verlag GmbH & Co. KG.



# Theoretische Überlegungen zu IoT Netzwerken

6

## Zusammenfassung

Das Kapitel enthält Informationen zum ISO/OSI-Schichtenmodell und weiterhin zu Codierung, Physical Layer, Leitungen, Transceiver, Sicherheit, Topologien, Synchronisation, MAC, Medienzugriff, sowie zu Protokollen der höheren Schichten.

Nach den ersten Vorüberlegungen zum Programmieren und zum Software-framework im ersten Teil des Buches, befasst sich der zweite Teil mit der Vernetzung von Sensoren. Dazu wird im sechsten Kapitel zunächst kurz das ISO/OSI-Schichtenmodell vorgeführt. Da sich das Buch auf kleine eingebettete (8-Bit) Systeme beschränkt, wird der Fokus auf die Umsetzung der Bitübertragungsschicht und der Sicherungsschicht gelegt. Dennoch soll den Leser\*innen ein kurzer Überblick über Protokolle im Internet of Things gegeben werden. Am Ende des Kapitels werden die Einschränkungen betrachtet, denen Sensorknoten im IoT unterworfen sind.

## 6.1 Das ISO/OSI Schichtenmodell<sup>1</sup>

Kommunikation findet statt, wenn Daten<sup>2</sup> 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

- 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 OSI<sup>3</sup> 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 Vermittlungsstellen an den Empfängerrechner weiter gibt, der seinerseits die Nachricht der betroffenen Anwendung zur Verfügung stellt (vertikale Kommunikation vs. Horizontale Kommunikation), wie die folgende Abb. 6.1 zeigt.

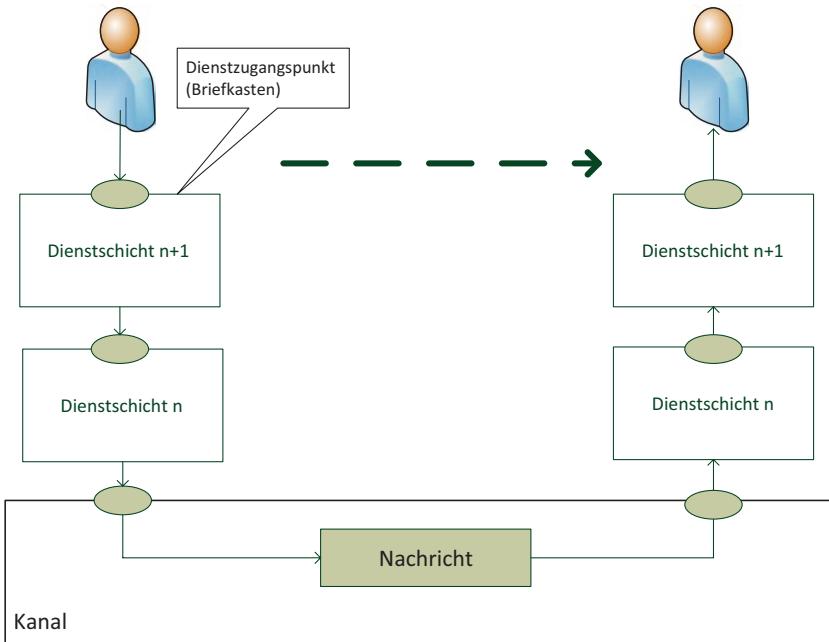
Für die Kommunikationssteuerung hat dies beträchtliche Konsequenzen. Die Schichten, die als eigenständige Instanzen, beispielsweise als Teil eines Betriebssystems oder einer Laufzeitumgebung existieren können, benötigen für die (virtuelle) horizontale Kommunikation untereinander Protokolle aber auch die Nutzung der nächsttieferliegenden Schicht für die vertikale Kommunikation erfolgt nach einem Protokoll. Die dafür benötigten Protokolldaten (für die horizontale) bzw. Interface-daten (für die vertikale Kommunikation) werden vor (Header) oder hinter (Trailer) die eigentlichen Nutzdaten (engl. Payload) angehängt. Damit erhöht sich die Größe des

---

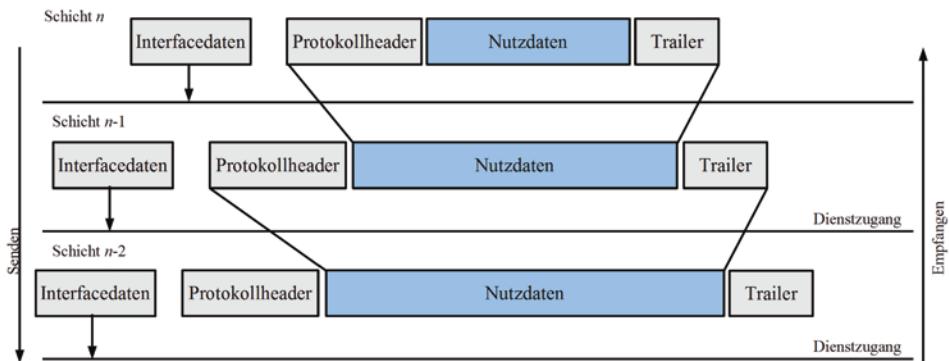
<sup>1</sup> Dieser Abschnitt wurde dem Buch Meroth, Tolg: „Bussysteme im Kfz“ [1] entnommen und stark überarbeitet.

<sup>2</sup> Erst durch Interpretation durch den Empfänger werden daraus Informationen.

<sup>3</sup> Open Systems Interconnection Reference Model.



**Abb. 6.1** Vertikale und horizontale Kommunikation im Schichtenmodell



**Abb. 6.2** Header und Trailer als Bestandteil des Protokollstapels

Datenpakete von Schicht zu Schicht, wobei Protokolldaten und Nutzdaten der nächsthöheren Schicht als Nutzdaten der nächsttieferen Schicht interpretiert werden (s. Abb. 6.2). 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, dessen Wurzeln bereits in die 70er des letzten Jahrhunderts zurückreichen, fungiert als Ordnungsrahmen, der beim Entwurf neuer Protokolle eine generelle Vorgehensweise vorgibt<sup>4</sup>. Daneben existieren die für das Internet bedeutsamen RFCs, *Request for Comments* (für *Bitte um Kommentare*), eine Reihe technischer und organisatorischer Dokumente zum Internet, die seit April 1969 herausgegeben werden [4]. Ursprünglich handelte es sich um buchstäblich zur Diskussion gestellte Vorschläge, heute findet die Diskussion bereits während der Erstellung der Entwürfe statt, sodass ein veröffentlichtes RFC in der Regel als eine begutachtete technische Spezifikation betrachtet werden kann.

In der folgenden Tab. 6.1 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.

Für die in diesem Buch beschriebenen Netzwerke und Softwarelösungen genügt das Verständnis der Layer 1 und 2 des ISO/OSI-Schichtenmodells und ein kurzer Ausflug in die TCP/IP-Kommunikation, darüber hinaus sei auf die umfangreiche Literatur und viele Publikationen im Internet verwiesen.

## 6.1.1 Schicht 1: Physikalische Bitübertragung

Im folgenden Abschnitt werden einige grundlegende Überlegungen zur physikalischen Datenübertragung auf Leitungen angestellt, soweit dies zum Verständnis der beschriebenen 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.

### 6.1.1.1 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 Ausbreitungsgeschwindigkeit gleich der

---

<sup>4</sup> Spezifiziert im Standard ISO 7498–1994 [2].

**Tab. 6.1** 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; Segmentierung 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

Lichtgeschwindigkeit  $c_0$  also  $c=c_0 \approx 3 \cdot 10^8$  m/s, bei realen Leitungen wird sie hauptsächlich vom verwendeten Isolierstoff bestimmt und liegt etwa bei  $c=0,6c_0$ . 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, sodass eine reale Leitung in erster Näherung ein Tiefpassfilter aus einem Serienwiderstand und einer Parallelkapazität darstellt<sup>5</sup>. Dadurch wird das Signal verschliffen, sodass bei großen Leitungslängen eine Signalregenerierung notwendig ist. Dieser ist bei den hier beschriebenen Anwendungen in der Regel noch nicht relevant, muss bei größeren Entfernung zu den Sensoren aber berücksichtigt werden.

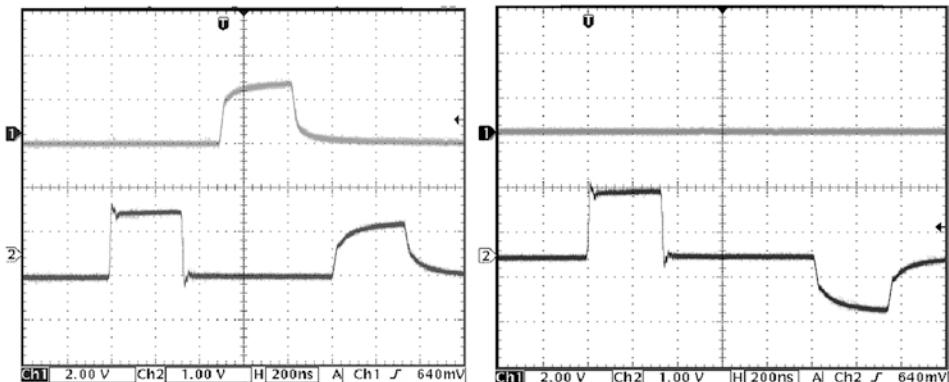
Neben der Speicherung und Dämpfung hat eine Leitung jedoch auch die Eigenschaft, Energie abzustrahlen und Energie aus der Umgebung der Leitung zu absorbieren. Dies führt zum Nebensprechen, engl. *crosstalk*, wenn mehrere Leitungen parallel nebeneinander geführt werden, oder zur Einstreuung von Störimpulsen, 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 [11]. Neben dem Verdrillen und Schirmen von Leitungen und der Schirmung von Gehäusen gehen EMV-Anforderungen direkt in die Spezifikation der Bussysteme ein. Auf der Platine ist unter anderem auf ausreichende Masseflächen zu achten, sowie darauf, Signal- und (geschaltete) Stromleitungen nicht parallel zu führen, bzw. über 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.

---

<sup>5</sup>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.



**Abb. 6.3** Impuls von 320 ns Dauer an einer 100 m langen Koaxialleitung ( $50 \Omega$ ); links: offenes Ende (jeweils oberes Signal), rechts: Kurzschluss am Ende. Kanal 1 (oben) ist jeweils am Leitungsende gemessen, Kanal 2 (unten) am Leitungsanfang

Das Verhältnis von hinlaufender zu rücklaufender Welle wird über den Reflexionsfaktor  $r$  mit

$$U_{2r} = r \cdot U_{2h} \quad (6.1)$$

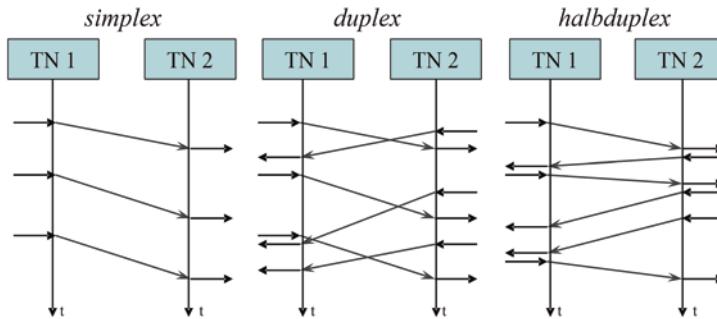
$$I_{2r} = -r \cdot I_{2h} \quad (6.2)$$

angegeben, der sich wie folgt berechnet:

$$r = \frac{Z_V - Z_L}{Z_V + Z_L} \quad (6.3)$$

Abb. 6.3 zeigt die Messung eines Impulses von 320 ns Dauer bei abgeschlossenem Leitungsanfang auf einem 100 m langen Koaxialkabel mit einem Wellenwiderstand von  $50 \Omega$ . 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 (linke Abbildung) 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



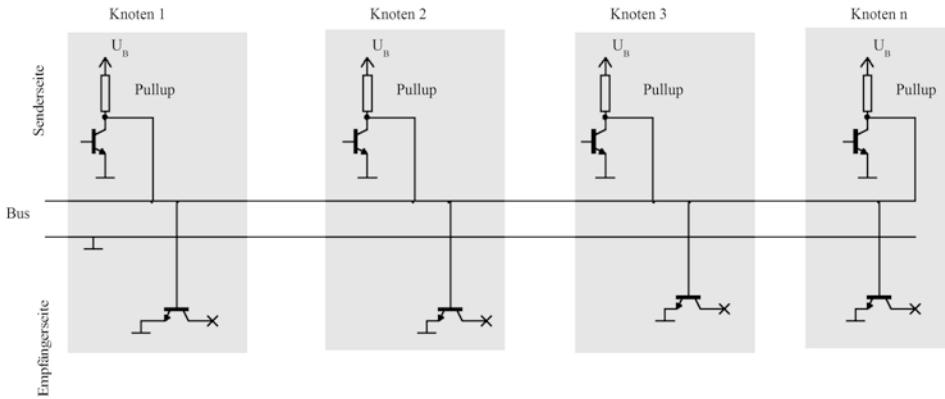
**Abb. 6.4** Übertragungsarten Simplex, Vollduplex und Halbduplex

wichtig. In der Regel sind die Abschlusswiderstände rein ohmsch, bestehen bisweilen aber auch aus einem R-C-Glied.

### 6.1.1.2 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 beide Richtungen verlaufen, spricht man von Duplex-Betrieb, wobei zwischen Vollduplex und Halbduplex unterschieden wird. Abb. 6.4 verdeutlicht den Zusammenhang anhand eines vereinfachten Sequenzdiagramms (siehe auch Abb. 7.10). Beim Vollduplexbetrieb 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, I<sup>2</sup>C/TWI) 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 angeschlossen (Open-Collector). Sendet irgendein

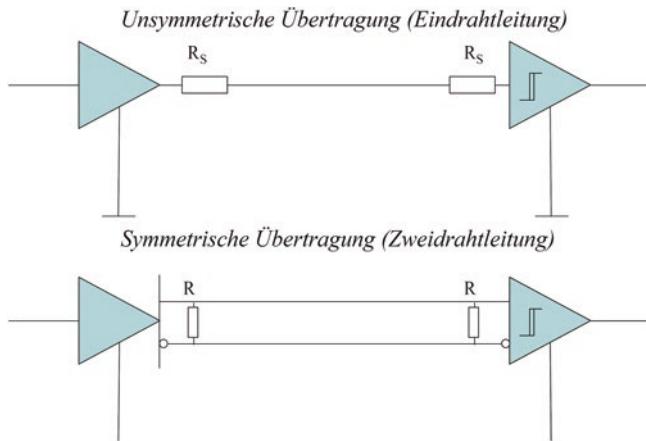


**Abb. 6.5** Wired-AND-Schaltung (Open Collector)

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 *rezzessiv*. Daher die Bezeichnung wired-AND (Abb. 6.5), 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 einen Zugriffsversuch unternimmt. Eine Konsequenz dieser Kollisionsmöglichkeit ist die Forderung nach einer Zugriffssteuerung, die später beschrieben wird.
- Leitungsart Eindrahtleitung oder Zweidrahtleitung: Je nach Zuverlässigkeitseinforderung werden die Signalleitungen als Ein- oder Zweidrahtleitungen ausgeführt. Die Eindrahtleitung (unsymmetrische Schnittstelle) (Abb. 6.6) ist die kostengünstigere Variante. Der Signalpegel wird gegen die allgemeine Masse angelegt. Dies macht die Eindrahtleitung anfällig gegenüber Störungen durch Einstreuung<sup>6</sup> 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 Spannungspegeln 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.

<sup>6</sup>Hin- und Rückleiter bilden eine Schleife mit sehr großer Schleifenfläche.



**Abb. 6.6** Unsymmetrische und symmetrische Übertragung

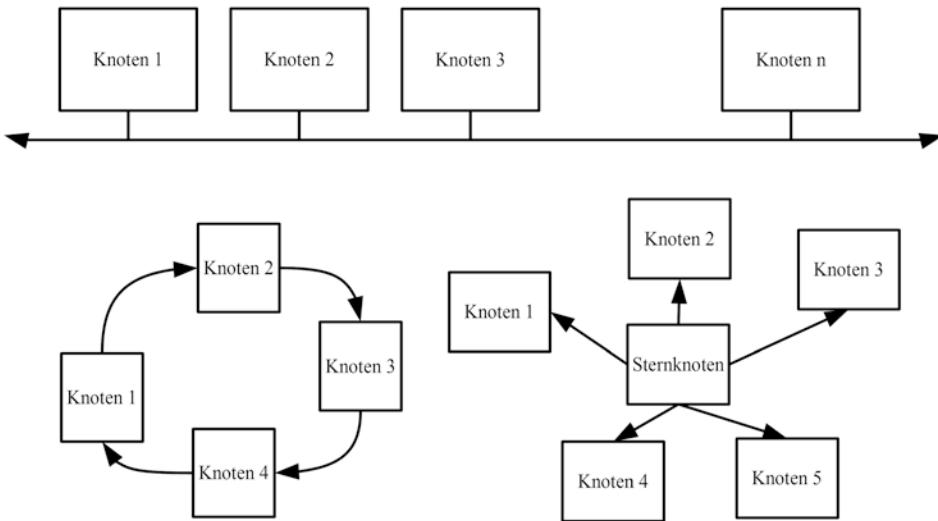
### 6.1.1.3 Übertragungssicherheit

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 [5], das heißt zur gegenseitigen Beeinflussung 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 sogenannte 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. 6.1.2) 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.

### 6.1.1.4 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. 6.7) durchgesetzt.



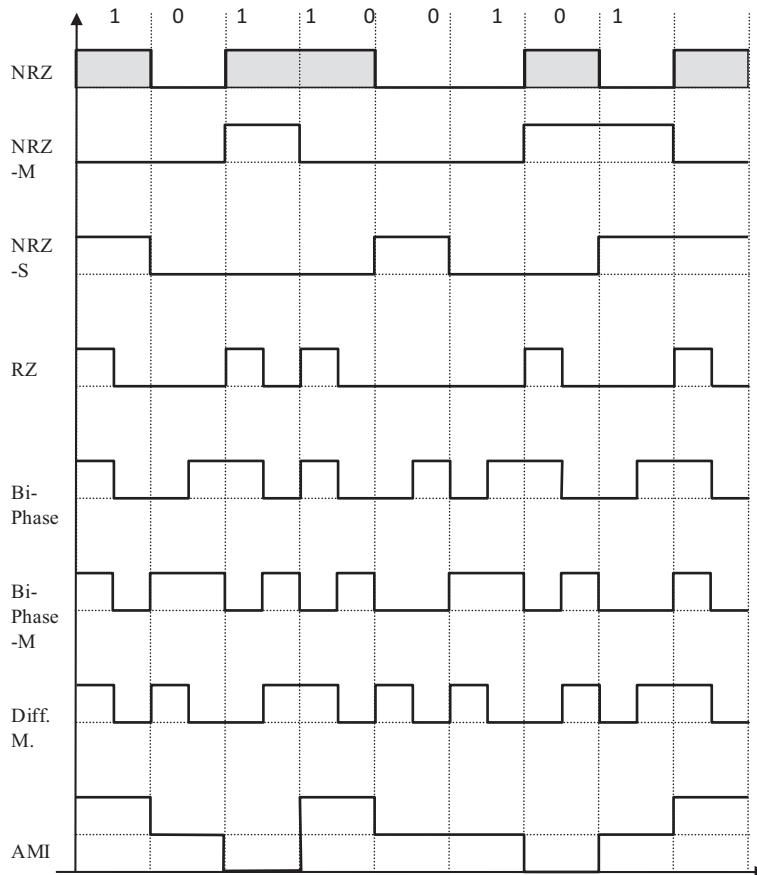
**Abb. 6.7** Bus, Ring und Stern-Topologie

- **Bus:** Der Bus, (auch Linie oder Line-Bus) ist die häufigste Topologie, vertreten z. B. durch CAN und I<sup>2</sup>C/TWI, die auf dem Point-to-Multipoint-Zugriff beruht. Alle Knoten senden und empfangen über dasselbe Medium, sodass 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 Sterntopologie, 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 Netzttopologie sind die einzelnen Knoten mit mehreren, im Extremfall mit allen Knoten im Netz direkt verbunden. Ein Beispiel findet sich in Kap. 13 bei ZigBee

- *Hierarchische Topologien:* Hierarchische Topologien sind solche, in denen einzelne Netzknoten den Ausgangspunkt für Subnetze darstellen (z. B. als Gateway). Man kann 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 Automatisierungs-technik findet sich beispielsweise in Fahrzeugbussystemen im Bereich lokaler Subnetze (LIN) und in den Diagnoseschnittstellen.

### 6.1.1.5 Synchronisation und Leitungscodierung

Die Leitungscodierung hat die Aufgabe, die physikalische Signaldarstellung auf dem Übertragungsmedium festzulegen (siehe Abb. 6.8). Den binären Bitsequenzen werden Pegelfolgen, z. B. Spannungen, Ströme oder Lichtintensitäten, zugeordnet. Dafür sind verschiedene Anforderungen relevant:



**Abb. 6.8** Leitungscodierung

- 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 I<sup>2</sup>S- oder I<sup>2</sup>C-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 getroffen werden, damit die Taktabweichungen nicht über Bitgrenzen hinauslaufen und damit eine Fehlmessung eintritt (siehe Kap. 7).
- 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 Spannungspegel  $V_1$  (High) dargestellt, eine logische 0 durch einen Spannungspegel  $V_2$ , 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

$$c_c = 2 \cdot B \quad (6.4)$$

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

$$c_c = 2 \cdot B \cdot \log_2 N \quad (6.5)$$

Leider werden mit einer wachsenden Zahl von Pegelstufen auch die Einflüsse eines immer gleichbleibenden Rauschpegels höher, sodass nach Claude Shannon die maximale Kanalkapazität des Übertragungsmediums  $c_c$  auf

$$c_c = 2 \cdot B \log_2 (1 + \text{SNR}) \quad (6.6)$$

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 Spannungspiegel auf dem Medium selbst dargestellt wird (Modulation). Die direkte Abbildung des Signalpegels auf den Spannungspiegel 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.

### 6.1.2 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. Dazu gehören WLAN (802.11), Ethernet (802.3), Mechanismen zum Verbindungsau- und -abbau (LLC 802.2), aber auch Bluetooth ® (802.15.1) oder die anderen Funknetzen zugrundeliegenden Sicherungsschichten (z. B. 802.15.4 für ZigBee).

Im Automobilbereich ist der Basis-Standard von CAN (ISO 11898 bzw. Bosch Spezifikation ebenfalls auf diese beiden Schichten beschränkt [6, 7]. 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:

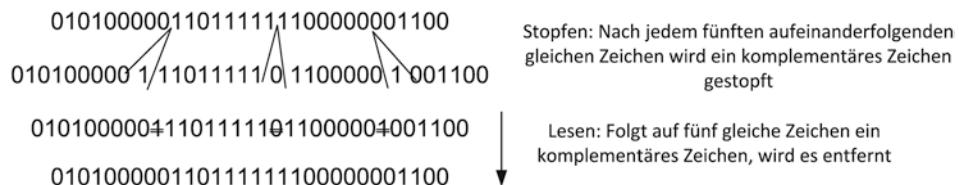
- Framing Sublayer für die Rahmenbildung
- Media Access Control (MAC) Sublayer für die Zugriffssteuerung auf den Physical Layer
- Logical Link Control (LLC) für die Kommunikationssteuerung und gegebenenfalls die Fehlerbehandlung

### 6.1.2.1 Rahmenbildung

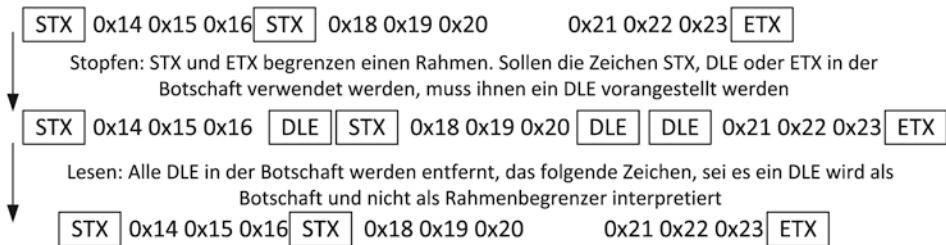
Aufgabe der Rahmenbildung ist es, zum einen dem Empfänger mitzuteilen, wo eine Nachrichteneinheit beginnt und wo sie endet, zum anderen, um Protokollinformation an ein Nutzdatenpaket zu hängen. Rahmen werden in der Regel an den Anfang eines Datenpakets gehängt (Header), in diesem Fall ist die Paketlänge 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: 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 Bittaktoszillatator 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-Oszillatoren. (siehe Kap. 7)
- 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. 6.9). Sobald der Sender mehr als fünf aufeinander folgende Bit gleichen Wertes senden soll, schiebt er ein Bit des komplementären Wertes in den Bitstrom ein [6]. 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, siehe Kap. 10) bis 1500 Byte (Ethernet) ohne weiteren Overhead außer dem Protokolloverhead übertragen.



**Abb. 6.9** Bitstuffing

**Abb. 6.10** Bytestuffing

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. 6.10). 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 vorkommt. 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.

Anderer Protokolle nutzen zur Erkennung eines neuen Rahmens das Bustiming: Eine Sendepause signalisiert beispielsweise im RTU-Protokoll des Modbus (Kap. 11) den Beginn eines neuen Rahmens. Wird diese Pause während der Übertragung des Rahmens festgestellt, muss der bis dahin übertragene Rahmen verworfen werden.

### 6.1.2.2 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, sodass 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 Messwerte<sup>7</sup> 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 bestimmten 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), sodass theoretisch nach einer beliebig langen Zeit jede Botschaft übertragen wird.

---

<sup>7</sup>Isochron heißt, dass die Signale periodisch mit exakt gleichbleibender Periodendauer übertragen werden.

- 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 Sendevorschlag. Das Verfahren vermeidet also Kollisionen und heißt daher CSMA/CA für Collision avoidance (siehe Kap. 10).

### 6.1.2.3 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 ist<sup>8</sup> (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 vonstatten geht 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:

- 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,

---

<sup>8</sup>Damit ist die Verbindung kontextbehaftet und das Protokoll muss für die Bereitstellung des Kontextes sorgen, d. h. für den Verbindungsaufl- und -abbau.

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.

#### 6.1.2.4 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.

#### 6.1.2.5 Fehlerbehandlung

Als letzte Aufgabe der Sicherungsschicht sei hier die Fehlerbehandlung beschrieben. Werden Daten über einen Nachrichtenkanal versendet oder auch nur von einem Speichermedium gelesen<sup>9</sup>, 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.

---

<sup>9</sup>Auch ein Speichermedium ist im weitesten Sinne ein Nachrichtenkanal mit hoher Latenzzeit.

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 Kommunikationstheorie<sup>10</sup> 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-Code<sup>11</sup> (Polynomial Code, zyklischer Code). CRC Codes basieren auf der Annahme, dass eine Bitfolge (ab jetzt Rahmen genannt) der festen Länge  $m$  als ein Polynom von Zweierpotenzen vom Grad  $r$  aufgefasst werden kann, beispielsweise die Bitfolge 110001 als Polynom vom Grad 5

$$\begin{aligned} & 1 \cdot x^5 + 1 \cdot x^4 + 0 \cdot x^3 + 0 \cdot x^2 + 0 \cdot x^1 + 1 \cdot x^0 \\ & = x^5 + x^4 + x^0 \end{aligned}$$

---

<sup>10</sup> Dieses mathematische Großbauwerk geht auf die Arbeit von Claude Shannon zurück, die dieser 1948 veröffentlicht hat.

<sup>11</sup> Cyclic redundancy check.

In der sogenannten Modulo-2-Arithmetik gelten stark vereinfachte Regeln, beispielsweise  $a+b=a-b$  (also  $1+1=0=1-1$ ,  $1+0=1=1-0$ ,  $0+1=1=0-1$ ), ohne Rest, damit entspricht eine Addition einer Subtraktion und einem logischen Exklusiv-Oder. Eine Modulo-Division ist entsprechend einfacher: Wenn ein Polynom in ein anderes „passt“, ist das Ergebnis 1, sonst 0. Es lässt sich mathematisch beweisen, dass hier ein algebraischer Körper vorliegt, in dem alle algebraischen Operationen abbildungbar sind. Eine Multiplikation eines Polynom mit einer Zweierpotenz  $x^n$  entspricht ein Verschieben der Bitfolge mit Anhängen von Nullen um  $n$ . Eine Multiplikation kann nur mit 1 oder 0 erfolgen, also die Bitfolge erhalten oder nicht. Ein Vorteil der CRC ist, dass man die Berechnung sehr einfach in Hardware umsetzen kann, da die wesentliche Operation – die Division – mit einer Schiebeoperation vorgenommen wird, die beim Senden im Sendeschieberregister automatisch stattfindet.

Das Vorgehen zur Berechnung der Prüfsumme lautet:

- Wähle ein Generatorpolynom  $G(x)$  vom Grad  $r$  mit  $r+1$  Bits
- Das höchstwertige und das niedrigstwertige Bit (MSB und LSB) müssen 1 sein
- Hänge  $r$  0-Bits an den Rahmen an. Es entsteht ein Polynom  $x^r \cdot M(x)$  mit  $m+r$  Bit-Länge
- Teile die Bitkette  $x^r \cdot M(x)$  durch  $G(x)$  anhand der Modulo-2-Division
- Ziehe den Rest (immer  $r$  Bits oder weniger) von  $x^r \cdot M(x)$  ab. Das Ergebnis ist das Polynom  $T_S(x)$ , dessen entsprechende Bitfolge übertragen wird.

Mathematisch geschrieben ist also die folgende Sendefolge zu bilden:

$$T_S(x) = x^r \cdot M(x) - \left[ \frac{x^r \cdot M(x)}{G(x)} \right] \quad (6.7)$$

Ein Empfänger empfängt die Bitfolge  $T_E(x)$  die sich aus  $T_S(x)$  und einem Fehler  $E(x)$  zusammensetzt, der sich nach den geltenden physikalischen Gesetzen additiv überlagert.

$$T_E(x) = T_S(x) + E(x) = x^r \cdot M(x) - \left[ \frac{x^r \cdot M(x)}{G(x)} \right] + E(x) \quad (6.8)$$

Teilt nun der Empfänger die empfangene Bitfolge erneut nach denselben Regeln durch das Generatorpolynom, so muss das Ergebnis, sofern der Fehler 0 ist, logischerweise auch 0 sein, da durch das Abziehen des Rests bei erneuter Division kein Rest mehr bleiben kann (gilt auch für die gewohnte Arithmetik). Sobald aber ein additiver Fehler vorliegt, wird das Ergebnis zu

$$\frac{T_E(x)}{G(x)} = \frac{x^r \cdot M(x) - \left[ \frac{x^r \cdot M(x)}{G(x)} \right] + E(x)}{G(x)} = \frac{E(x)}{G(x)} \quad (6.9)$$

Das heißt, alle Fehler  $E(x)$  werden erkannt, die nicht Vielfache des Generatorpolynoms  $G(x)$  sind.

Ein Beispiel:

- Das Generatorpolynom wird mit  $G(x)=x^4+x^3+x^1+x^0$  (Bitfolge 10011,  $r=4$ ) gewählt. Es existieren verschiedene standardisierte Generatorpolynome, die sich bewährt haben. Für ein Protokoll wird eines festgelegt und ist Bestandteil des Protokolls, das heißt es ändert sich auch nicht und muss für Empfänger und Sender natürlich identisch sein.
- Der Rahmen in diesem Beispiel sei  $M(x)=x^3+x^2+x^1+x^0$  (Bitfolge 1111)
- Die Verschiebung  $x^r * M(x)=x^7+x^6+x^5+x^4$  liefert die Bitfolge 11110000
- Nun wird  $x^r * M(x) \% G(x)$  modulo dividiert:

$$\begin{array}{r} \underline{11110000} \\ -\underline{10011} \\ \underline{011010} \\ -\underline{10011} \\ \underline{010010} \\ -\underline{10011} \\ \hline 000010 \end{array}$$

- Anschließend wird das Sendepolynom gesendet:  $T_S(x)=x^r \cdot M(x) - [x^r \cdot M(x) \% G(x)] = 000011110010$

Der Empfänger teilt das empfangene Polynom nochmals durch das Generatorpolynom, hier:

$$\begin{array}{r} \underline{11110010} \\ -\underline{10011} \\ \underline{011010} \\ -\underline{10011} \\ \underline{010011} \\ -\underline{10011} \\ \hline 000000 \end{array}$$

Standardisierte Generatorpolynome sind beispielsweise:

• IEEE802:	$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x^1+1$
• CRC-12:	$x^{12}+x^{11}+x^3+x^2+x+1$
• CRC-16:	$x^{16}+x^{15}+x^2+1$
• CRC-CCITT:	$x^{16}+x^{12}+x^5+1$
• CAN-CRC:	$x^{15}+x^{14}+x^{10}+x^8+x^7+x^4+x^3+1$

Sie werden natürlich auf größere Rahmen angewendet, in der Regel einige hundert Bit. In Abschn. 12.1.4.2 ist ein praktisches Beispiel zur Programmierung und für die hardwaretechnische Realisierung einer CRC-Prüfsumme dargestellt.

Für Übertragungsnetze, die nicht mit festen Codewortlängen arbeiten, existieren Faltungscodierer, die aus einem theoretisch unendlich langen Bitstrom einen neuen,

längerem 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 ist dies nicht der Fall, bei anderen Anwendungen ist die Latenzzeit zu hoch (beispielsweise bei einer Marssonde) für eine Rückkommunikation.

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, sodass der Sender nicht weiß, welche Nachricht er bei Bedarf zu wiederholen hat. Zwei Mechanismen helfen hier insbesondere weiter [9]:

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 Zwischen-Speicher 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.

### 6.1.3 Schicht 3: Vermittlungsschicht

Die Vermittlungsschicht hat die Aufgabe, Datenpakete über Netzwerkgrenzen hinweg zu routen. Ein Datenpaket der Sicherungsschicht adressiert immer einen Knoten im Netzwerk des sendenden Knotens. Möchte man Daten über Netzwerke hinaus senden (beispielsweise aus dem eigenen WLAN ins Internet), wird in die Nutzlast der Sicherungsschicht ein Datenpaket der Vermittlungsschicht gepackt und an ein im Netz notwendiges Gateway geschickt. Ein Gateway ist immer ein Netzknoten, der Verbindung zu mehreren Netzen hat, unabhängig von deren Sicherungsschichtprotokoll. Gateways, die mit Schicht-3-Protokollen arbeiten, werden Router genannt, es gibt jedoch auch andere (beispielsweise arbeiten CAN-Gateways nur mit dem CAN-Protokoll, das auf Schicht 2 angesiedelt ist). Genaugenommen ist die Aufgabe des Routers aber nicht, eine Route zu berechnen (als Gesamtheit aller Wege), sondern nur das Datenpaket an den nächsten Router weiterzuleiten (forwarding). Dies geschieht zwar mit einer beabsichtigten Route, letztlich obliegt es aber jedem Knoten, an dem das Paket vorbei-kommt, wie die weitere Route auszusehen hat, sodass theoretisch auch endlose Schleifen möglich sind. Ein Schicht-3-Protokoll muss somit auch Mechanismen zur Schleifenbildung enthalten.

Das IP-Protokoll ist das bekannteste Schicht-3-Protokoll und hat sich insgesamt durchgesetzt. Im IPV4 besteht die netzunabhängige Adresse aus vier Byte, die durch vier mit einem Punkt getrennte Dezimalzahlen 0...255 dargestellt werden. Ein Teil des Adressraums ist als lokal reserviert und wird nicht nach außen geroutet. Dieser Adressraum ist 10.x.x.x, 172.16.0.0 bis 172.131.255.255 und 192.168.0.0 bis 192.168.255.255. Alle anderen Adressen dürfen nur einmal weltweit vorkommen und werden zentral von der „Internet Assigned Numbers Authority“ (IANA) vergeben, die über den gesamten IP-Adressraum herrscht. Sie vergibt IP-Adressblöcke an so genannte „Regional Internet Registries“ (RIR), die für die regionale IP-Adressvergabe zuständig sind. Diese ihrerseits vergeben IP-Adressbereiche an beispielsweise Internet-Service-Provider (ISP). Das Adressierungsschema von IP teilt sich in die eigentliche Netzadresse und eine

**Tab. 6.2** Adressklassen in IPv4

Name	Kennzeichen	Adressraum	Umfang
Class A	Erstes Bit des ersten Bytes ist 0	1.0.0.0 bis 127.255.255.255	126 Netze mit je 16 Mio. Knoten
Class B	Erste zwei Bit des ersten Bytes sind 10	128.0.0.0 bis 191.255.255.255	16.384 Netze mit je 65.534 Knoten
Class C	Erste drei Bit des ersten Bytes sind 110	192.0.0.0 bis 223.255.255.255	2 Mio Netze mit je 254 Knoten
Class D	Erste vier Bit des ersten Bytes sind 1110	Multicast Adresse 224.0.0.0 bis 239.255.255.255	Multicast Adresse spricht keine einzelnen Knoten an: $2^{28}$ Netze
Class E	Erste fünf Bit des ersten Bytes sind 11110	240.0.0.0 bis 247.255.255.255	Reserviert

Geräteadresse auf. Die Grenze zwischen Netzadresse und Geräteadresse ist nicht fest, sondern wird durch zwei Mechanismen bestimmt: Die Adressklassen und die Subnetzmasken.

In den Anfängen des Internet hatte man nicht mit der schiere Masse der angeschlossenen Rechner gerechnet. Zunächst wurden die in Tab. 6.2 genannten Adressklassen definiert.

Die meisten Netze (Firmen, Internetprovider, Hochschulen) haben mehr Bedarf als Class C aber weniger als Class B, sodass als weiterer Mechanismus die Subnetzmaske eingeführt wurde.

Subnetzmasken kennzeichnen den Bereich der IP-Adresse, der das Netzwerk und das Subnetzwerk beschreibt. Dieser Bereich wird dabei durch Einsen („1“) in der binären Form der Subnetzmaske festgestellt. Damit ist es möglich, dass sich mehrere Netze/Dienste eine Class B Adresse teilen. Damit die Adressen besser ausgenutzt werden, werden sie im lokalen Netz gerne dynamisch vergeben. Dazu wird unter anderem das DHCP-Protokoll verwendet, bei dem ein Knoten, der sich im Netz anmeldet, eine IP-Adresse zugewiesen bekommt. Private Anwender bekommen in der Regel keine öffentlichen Adressen zugewiesen sondern nutzen private Adressräume. Der an den Provider angeschlossene Router bekommt eine dynamische IP zugewiesen, die sich regelmäßig ändern kann. Das Weiterleiten der Daten an ein Endgerät erfolgt durch Schicht-4-Protokoll, dem NAT (Network Address Translation), siehe den nächsten Abschnitt.

Um eine Systematik in die Flut von Internetadressen zu bekommen, wurde bereits 1983 im RFC882 das Domain Name System eingeführt. Dieses gliedert das Internet hierarchisch auf. Die so genannten Top-level-domains werden von der IANA vergeben und haben sich von einem rein amerikanischen System (beispielsweise .com für Firmen, .gov für die Regierung, .org für Vereine, .edu für Hochschulen) über länderspezifische Domains zu einem breiten Spektrum weiter entwickelt. Jede Top-level-domain wird von

einem Network Information Center (NIC) oder einer Domain Name Registry verwaltet, die ihrerseits darunterliegende Domains verwaltet. In Deutschland ist dies das DENIC, das zunächst in Karlsruhe angesiedelt war und nun in Frankfurt residiert (in der Schweiz: SWITCH und in Österreich nic.at).

Die Domainnamen werden auf so genannten Domain-Name Servern (DNS) in einem Namensverzeichnis mit IP Adressen in Verbindung gebracht. Da diese wechseln können, tauschen die Domain-Name-Server sich regelmäßig über Änderungen aus. Befindet sich ein Name nicht in einem Namensverzeichnis, kann der DNS einen übergeordneten DNS rekursiv anfragen. Wird ein Knoten im Internet als Domainname adressiert, muss der sendende Knoten also zunächst eine Namensanfrage stellen, nach der er eine IP Adresse zurückbekommt. Da IP Adressen ständig wechseln können (beispielsweise zum Lastausgleich bei Serverfarmen oder im privaten Bereich), ist der Domainname die stabilere Adressierungsvariante. Für Privatpersonen steht ein sogenannter DynDNS zur Verfügung, hier bekommt man eine Domain zugewiesen, der heimische Router meldet dann bei jeder IP-Adressänderung die neue Adresse an den DynDNS Server, der seinerseits die anderen DNS in seiner Umgebung informiert.

Da der Adressraum von  $2^{32}$  Adressen bereits vor Jahren knapp wurde, wird inzwischen auf das IPv6 Protokoll umgestellt, das 128-Bit-Adressen zur Verfügung stellt und damit  $2^{128} = 3,4 \cdot 10^{38}$  Adressen. Dies ist so viel, dass eine statische Adressvergabe nicht mehr notwendig ist. Die Adressen werden hexadezimal mit Doppelpunkt getrennt geschrieben, beispielsweise: 2002:6dc1:ac2a:0:1cce:7c9d:62b8:d6e0. Auch in IPv6 wird die Adresse in einen Netzwerkteil und einen Knotenteil unterteilt.

Wie läuft nun die Kommunikation von Schicht-3-Botschaften über ein Netzwerk der Schicht 2 ab?

Will ein Knoten einen anderen, innerhalb oder außerhalb des eigenen Netzes über eine IP-Adresse erreichen, sendet er eine ARP-(Address Resolution Protocol) Nachricht ins Netz und fragt damit an, wer diese IP-Adresse besitzt. Es kann entweder ein anderer Knoten antworten oder aber der Router, der diese Adresse kennt bzw. auf eine Routentabelle Zugriff hat, die diese Adresse beinhaltet. Mit der Antwort kommt die lokale (MAC) Adresse, an die dann der ursprünglich sendende Knoten seine IP-Botschaft in einen Schicht2-Rahmen verpackt sendet. Der Router packt die Botschaft aus und sendet sie ihrerseits in das Netzwerk, das hinsichtlich definierter Qualitätsfaktoren die bestmögliche Route zum adressierten Ziel darstellt. Damit diese Routen immer aktuell und optimal sind, tauschen sich alle Router regelmäßig mit Routingprotokollen (z. B. BGP – Border Gateway Protocol oder RIP – Routing Information Protocol) aus.

Optimal ist eine Route beispielsweise, wenn sie möglichst wenige zwischen geschaltete Router (Hops) beinhaltet, oder möglichst wenig Paketverluste aufweist oder möglichst kurze Laufzeiten. Auch die Verbindungskosten können eine Rolle spielen. Welche Route letztlich genommen wird, entscheidet jeder Router anhand eines Routing Algorithmus, der auf den genannten Informationen seiner Nachbarrouter basiert.

Das IP-Protokoll nutzt also das Store-And-Forward-Verfahren und ist verbindungslos, da jedes IP-Paket für sich mit voller Adressinformation versendet wird. Jeder Hop

speichert das Paket kurzfristig um es dann weiterzuleiten. Sollen vertrauliche Daten verschickt werden, müssen sie daher Ende-zu-Ende verschlüsselt werden. Auf Schicht 3 existiert daher eine verschlüsselte Erweiterung IPsec des IP-Protokolls.

### 6.1.4 Schicht 4: Transportschicht

Die Transportschicht hat zur Aufgabe, das Netzwerk von der Anwendung zu trennen. Die Anwendungen liefern also der Transportschicht Daten und diese bereitet sie so auf, dass sie richtig versendet und der richtigen Anwendung auf der Gegenseite zugestellt werden. Hierzu gibt es mehrere Mechanismen, die in den bekanntesten Internet-Transportschichten realisiert werden, unter anderem:

- Segmentierung: Zu große Botschaften können aufgeteilt und beim Empfänger wieder zusammengesetzt werden (ein Mechanismus, der auch in der CAN-Transportschicht umgesetzt ist)
- Flusskontrolle: Die Transportschicht steuert die Kommunikation zwischen asynchronen Partnern (insbesondere, wenn der Empfänger langsamer ist als der Sender) so, dass eine möglichst kontinuierliche Datenübermittlung ohne Verluste erfolgen kann.
- Zuverlässigkeit (Absicherung der Kommunikation): Basiert die Kommunikation auf unzuverlässige untere Schichten (wie IP), wird hier mit den in Abschn. 6.1.2.5 beschriebenen Verfahren dafür gesorgt, dass die Daten fehlerfrei, verlustfrei und in der richtigen Reihenfolge übertragen werden
- Ende-zu-Ende Adressierung: Schicht-3-Protokolle verbinden in der Regel Rechner. Da aber die Kommunikation zwischen Anwendungen (Beispiel: Mailclient an Mailserver, Browser an Webserver, Sensor an Broker), stattfindet, wird ein weiterer Mechanismus (im Internet: Port) eingeführt, der einer Addresserweiterung gleichkommt. Eine Anwendung wird einem Port zugewiesen und alle an diesen Port gerichteten Nachrichten gehen an diese Anwendung. Gleichzeitig legt ein Sender den Port für eine Rückantwort fest. Bekannte und durch gemeinsame Konventionen festgelegte Ports im Internet sind 20 (FTP), 22 (Secure Shell), 25 (Mail), 80 (http) und so weiter.
- Verbindungssteuerung: Im Internet gibt es zwei sehr bekannte Transportschicht TCP und UDP, siehe unten. Das TCP-Protokoll ist verbindungsorientiert, das UDP-Protokoll ist verbindungslos (siehe Abschn. 6.1.2.3).

TCP (Transport Control Protocol) vereint die oben genannten Mechanismen. Es ist in der Lage zu segmentieren, baut einen Verbindungskontext auf, besitzt Handshakemechanismen zu Absicherung des Datenverkehrs, eine Prüfsumme, Flusskontrolle um die wichtigsten zu nennen. TCP wird beispielsweise vom http Protokoll genutzt und ist immer dann angebracht, wenn größere Datenpakete sicher versendet werden müssen.

UDP (User Datagram Protocol) ist sehr simpel und eigentlich nur eine Erweiterung von IP um eine Portadresse und eine Prüfsumme. Es sind keine Handshakemechanismen vorgesehen und keine Fehlerbehandlung. Dafür ist das Protokoll schnell und eignet sich für kleinere Datenpakete bis 65 kByte, bei denen ein eventueller Paketverlust eher in Kauf zu nehmen ist als eine aufwendige Wiederversendung.

Auf komplexeren Betriebssystemen (wie UNIX/Linux oder Windows) wird die Transportschicht in Form von Sockets bereitgestellt, einer Programmierschnittstelle, die von der Anwendung genutzt werden kann. Der Ablauf gestaltet sich wie folgt:

TCP-Sockets (Stream Socket) Auf der Client-Seite:

- Socket erstellen (mit `socket()`)
- Den erstellten Socket mit der Server-Adresse verbinden, von welcher Daten angefordert werden sollen, zunächst mit Abfrage der IP-Adresse vom DNS oder Erzeugen einer IP-Adressstruktur aus den bekannten IPv4 oder IPv6 Bytes: `inet_pton(AF_INET, "1.2.3.4", &srv.sin_addr);` anschließend einen Port zuweisen und verbinden: `srv.sin_port=htons(1234); connect(sockfd, (struct sockaddr*)&srv, sizeof(struct sockaddr_in));`
- Senden und Empfangen von Daten `read()`, `write()`
- Gegebenenfalls am Ende Socket herunterfahren (`shutdown()`, `close()`)
- Verbindung trennen, Socket schließen

Server-seitig:

- Server-Socket erstellen (s. o.)
- Binden des Sockets an einen Port, über den Anfragen akzeptiert werden (`bind()`)
- auf Anfragen warten (`listen()`)
- Anfrage akzeptieren und damit ein neues Socket-Paar für diesen Client erstellen (`accept()`)
- Bearbeiten der Client-Anfrage auf dem neuen Client-Socket
- Client-Socket wieder schließen.

UDP (Datagram) Sockets.

Client-seitig:

- Socket erstellen
- Senden (`send()`)

Server-seitig:

- Socket erstellen
- Socket binden
- Warten auf Pakete

Für Server gibt es noch die Spezialität, dass das Warten auf eine Anfrage die Software blockiert, ebenso das Warten auf eine Antwort bei Client und Server. Dies kann in einem höheren Betriebssystem dadurch umgangen werden, dass das Warten in einen Thread verlagert werden kann.

### 6.1.5 Anwendungsprotokolle für IoT Netzwerke

Auf der Seite der Anwendungsschichten (Layer 5–7 des ISO-OSI Schichtenmodells) sind die semantischen Aspekte der übertragenen Daten gebündelt.

Im Internet wird eine ganze Reihe von Protokollen standardisiert, die die Anwendungen miteinander verbinden, die bekanntesten sind:

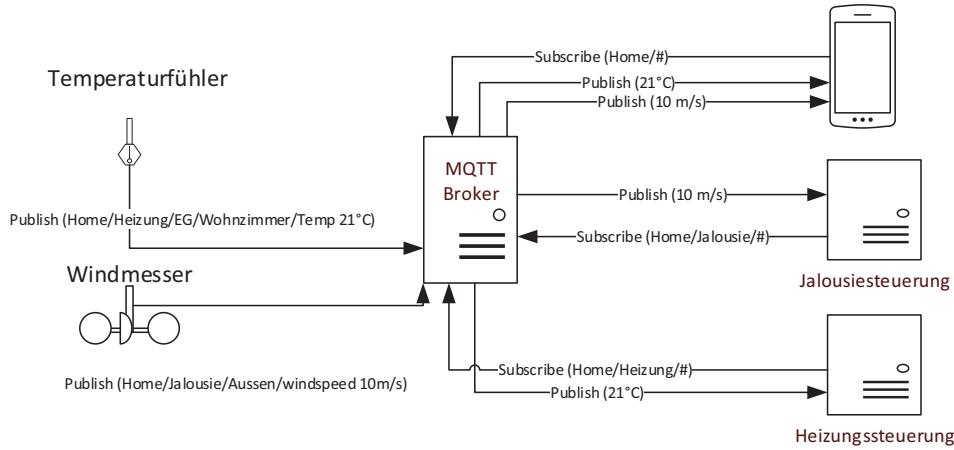
- DoIP (Diagnostics over IP) – Transportprotokoll für Fahrzeugdiagnose
- FTP (File Transfer Protocol) – Dateitransfer
- HTTP (Hypertext Transfer Protocol, WWW)
- HTTPS (Hypertext Transfer Protocol Secure)
- IMAP (Internet Message Access Protocol) – Zugriff auf E-Mails
- NTP (Network Time Protocol)
- POP3 (Post Office Protocol, Version 3) – E-Mail-Abruf
- PTP (Precision Time Protocol) – Zeitsynchronisation von Uhren in einem Netzwerk
- RDP (Remote Desktop Protocol) – Darstellen und Steuern von Desktops auf fernem Computern (Microsoft)
- RTP (Real-Time Transport Protocol)
- SNMP (Simple Network Management Protocol) – Verwaltung von Geräten im Netzwerk
- SMTP (Simple Mail Transfer Protocol) – E-Mail-Versand
- SSH (Secure Shell) – verschlüsseltes remote terminal<sup>12</sup>
- Telnet – Unverschlüsseltes Login auf entfernten Rechnern (remote terminal)

Ebenfalls im Anwendungssystem befinden sich verschiedene Formate für Sensordaten, die an dieser Stelle kurz Erwähnung finden sollen:

- MQTT: Message Queuing Telemetry Transport (MQTT) ist ein offenes Client-Server-Protokoll für Machine-to-Machine-Kommunikation (M2M), das die Übertragung von Mess- und Telemetrie-Daten trotz hoher Verzögerungen oder beschränkter Netzwerke ermöglicht [8]. Ein Client sendet dem Server (dem MQTT-Broker) ein so genanntes Topic, das hierarchisch benannt ist (beispielsweise Home/Heizung/EG/Wohnzimmer/TemperaturIST). Andere Clients, z. B. Steuerungen oder

---

<sup>12</sup>Eine recht gute Liste findet sich in Wikipedia beim Schlagwort „Internetprotokollfamilie“.

**Abb. 6.11** MQTT Prinzip

Endgeräte können diese Topics abonnieren (subscribe) und bekommen sie daraufhin bei jeder Änderung zugestellt. Wildcards sind möglich, z. B. abonniert Home/Heizung/+ /TemperaturIST alle Temperatur-Ist-Werte aus allen Räumen, während Home/Heizung/EG/# alle heizungsrelevanten Daten aus allen Räumen des Erdgeschosses abonniert. Ein vereinfachtes Beispiel ist in Abb. 6.11 zu sehen.

- **JSON:** Die JavaScript Object Notation ist kein Protokoll sondern ein kompaktes Datenformat in einer einfach lesbaren Textform und dient dem Zweck des Datenaustausches zwischen Anwendungen. JSON – definiert in RFC 8259 – ist von der Programmiersprache unabhängig. Parser und Generatoren existieren in allen verbreiteten Sprachen. In JSON werden Datenstrukturen ausgetauscht, die hierarchisch gegliedert sind. Jeder Eintrag besteht aus einem Namen (Schlüssel) und einem Wert, wobei dieser wieder aus einer Struktur bestehen kann. Auch Arrays sind möglich.

```
{
  "city": {
    "name": "Heilbronn",
    "country": "DE",
    "coord": {
      "lon": 9.206321,
      "lat": 49.147134
    },
    "temp": [281.15, 285.31, 288.22, 290.51, 292.98]
  }
}
```

JSON löst derzeit mehr und mehr XML ab, das von der Struktur her ähnlich ist, oft aber ambivalent notiert werden kann.

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 [9, 10] zu erhalten.

---

## 6.2 Anforderungen an IoT Netzwerke

Im RFC 7228 (siehe [11] und) werden die speziellen Anforderungen an IoT Netzwerke mit Constraint Nodes beschrieben. Constraint Nodes sind Netzwerkteilnehmer, bei denen „einige der Eigenschaften, die sonst für Internet-Knoten ziemlich selbstverständlich sind, zum Zeitpunkt des Schreibens nicht erreichbar sind, oft aufgrund von Kostenbeschränkungen und/oder physikalischen Einschränkungen bei Eigenschaften wie Größe, Gewicht und verfügbare Leistung und Energie.“ (RFC7228).

Die in diesem Buch vorgestellten Konzepte erfüllen diese Definition, insoweit sie auf 8-Bit Mikrocontrollern umgesetzt sind. Aufgrund der Portabilität der vorgestellten Codes sei aber darauf hingewiesen, dass bereits eine Umsetzung mit einem Kleinstcomputer mit LINUX Betriebssystem (z. B. Raspberry Pi) kaum noch von einem Constraint Node geredet werden muss, insbesondere, wenn eine Stromversorgung aus dem Netz zur Verfügung steht.

Die wesentlichen Eigenschaften solcher Netzwerke sind:

- Einbindung des Netzwerks in bestehende Infrastrukturen, die mindestens im Gateway oder im Backend eine Anbindung an das Internet haben.
- Für die Entwicklung und den Betrieb wird auf bekannte Technologien und Werkzeuge gesetzt.
- Für die „letzte Meile“ wird auf neue Standards und Protokolle gesetzt, die Protokolle sind oft offengelegt (nicht immer!).
- Im Vergleich zum „normalen“ Internet besteht ein Sensornetzwerk aus einer großen Anzahl einfacher Geräte.
- Diese unterliegen häufig physischen und ökonomischen Zwängen.

Geräte im IoT sind in ihren Ressourcen oftmals eingeschränkt. Dies betrifft die

- Baugröße und die Schnittstellen, insbesondere die Benutzungsschnittstelle (user interface), die oft ganz fehlt, aber auch Programmierschnittstellen zum Update der Software, die oft gänzlich fehlen oder nur mit physischer Nähe oder eingeschränkter Bandbreite zugreifbar sind (im Gegensatz zum vollen Remote- oder OTA (Over-the-air) Zugriff bei komplexeren Knoten)
- Stromversorgung, insbesondere bei Batteriebetrieb oder beim Betrieb mit Energy Harvesting, also dem Betrieb, bei dem die Energieversorgung aus der Umwelt entnommen wird, beispielsweise mit Solarzellen, Windgeneratoren, aber auch Temperaturunterschieden, mechanischer Bewegung (Vibration, Druck) oder

Magnetfeldern. So existieren seit geraumer Zeit Funkschalter, die über Piezokristalle so viel Energie erzeugen, dass ihr Funksignal ausgesendet werden kann. Auch RFID bedient sich dieser Technik.

- Rechenleistung und Speicher: Baugröße, große Anzahl (Kosten) und niedriges Energiedargebot begrenzen auch die Rechenleistung und den Speicher, ein Grund dafür, dass in diesem Buch ausschließlich Konzepte vorgestellt werden, die auf 8-Bit Mikrocontrollern getestet wurden. Entsprechendes gilt für die Codekomplexität, die aufgrund der kleinen Speicher angepasst werden muss.
- Netzwerk-Anbindung: Entsprechend sind Bandbreite, Datenaufkommen und Latenzzeiten begrenzt und es werden eigene Protokolle benötigt, auf eine Auswahl davon wird in den folgenden Kapiteln näher eingegangen.

Speziell die Netzwerke werden als Constraint Network bezeichnet. Zu den Einschränkungen können gehören [11]:

- niedrige erreichbare Bitrate/Durchsatz (einschließlich begrenzter Abtastraten),
- hohe Gefahr von Paketverlust und hohe Variabilität des Paketverlustes aufgrund der niedrigen Übertragungsleistungen und regulatorischer Beschränkung der verfügbaren Übertragungsbänder,
- stark asymmetrische Link-Eigenschaften, beispielsweise kein Rückkanal,
- hohe Strafen für die Verwendung größerer Pakete (speziell in den öffentlichen ISM Bändern), was zur Fragmentierung<sup>13</sup> der Datenpakete führen kann (dadurch ebenso Gefahr von Paketverlust),
- Grenzen der Erreichbarkeit über die Zeit (eine erhebliche Anzahl von Geräten könnte aus energetischen Gründen ausgeschaltet sein, diese werden aber in der Regel periodisch „geweckt“ und können für kurze Zeiträume kommunizieren) und
- das Fehlen von (oder starke Einschränkungen bei) fortgeschrittenen Diensten wie IP

Multicast.

Zusammengefasst kann man sagen, dass die Einschränkungen:

- auf Kostenseite existieren,
- durch die Knoten und ihre Einsatzbedingungen entstehen können,
- physikalischen Ursprung haben können (Umwelt, wie EMV, Temperatur, Temperaturwechsel, Wetterexposition, Vibrationen, Wasser),
- regulatorische Gründe haben können, wie bei der Übertragung im ISM Band, das von den in der Folge vorgestellten Funkstandards genutzt wird.

---

<sup>13</sup> Als Fragmentierung oder Segmentierung bezeichnet man die Zerkleinerung eines Datenpakets in kleine Unterpakete, die voneinander unabhängig übertragen werden. Durch den Einsatz geeigneter Protokolle werden sie beim Empfänger wieder zum Originalpaket zusammengesetzt.

**Tab. 6.3** Energieklassen im IoT [10]

Name	Art der Energiebeschränkung	Beispiel für Energiequellen
E0	Beschränkte Energie pro Ereignis (event), z. B. beim Betätigen eines Schalters	Ereignis basierte Energieversorgung (Harvesting), z. B. aus der Betätigungsenergie des Schalters oder einem Magnetfeld
E1	Zeitlich beschränkte Energie	Batterie oder Akku mit Wechsel- oder Aufladezyklen
E2	Energie über die limitierte Lebenszeit verfügbar	Nicht austauschbare Batterie (z. B. in Rauchmeldern)
E9	Keine Beschränkungen	Mit Netzteil versorgt

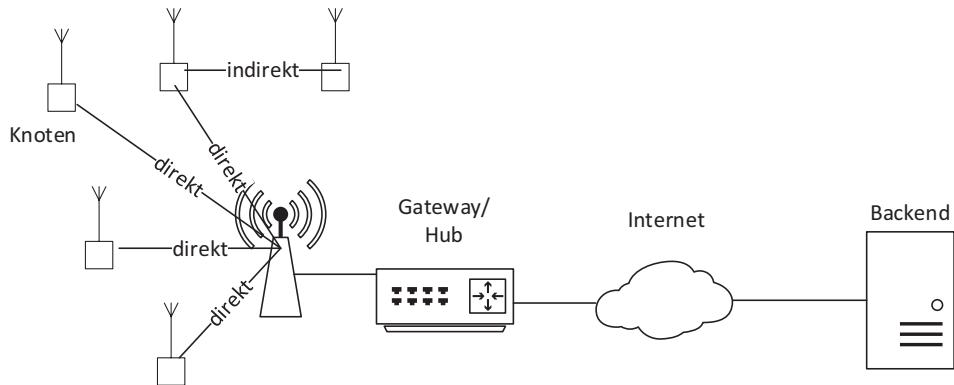
Zudem darf man in einem wachsenden IoT auch davon ausgehen, dass Technologien und Protokolle verschiedener Reifegrade und Entwicklungsstufen nebeneinander existieren können und daher bei der nachgeordneten Verarbeitung streng auf Kompatibilität geachtet werden muss.

Für die Energieversorgung gibt es zwei charakteristische Größen, nämlich die durchschnittliche Leistungsaufnahme während des Betriebs, die beispielsweise die Größe einer Solarzelle oder eines anderen Energiewandlers bestimmt, und die minimale Energie, die zur Verfügung stehen muss um das Gerät am Leben zu erhalten. Letztere bestimmt die Größe eines eventuellen Speichers (Batterie, Powercap).

RFC 7228 schlägt vier Energieklassen vor, die der Tab. 6.3 zu entnehmen sind

Je nach Energieklasse kann man unterschiedliche Strategien fahren, siehe dazu auch Abschn. 3.9, die in der RFC 7228 beschrieben sind:

- Always on: Hier sind keine bestimmten Verhaltensweisen vorgeschrieben wenn es aufgrund der energetischen Randbedingungen nicht notwendig ist. Das Gerät ist schlicht immer an, es mag aber sinnvoll sein, die Hardware leistungsoptimiert auszulegen, den Prozessor langsam zu takten, die Zahl der Übertragungen zu limitieren oder andere Energiesparmaßnahmen zu implementieren. Das Gerät ist im Netzwerk aber immer erreichbar.
- Normally off: Hier ist das Gerät in der Regel ausgeschaltet und wird nur aktiviert, wenn es Daten übertragen muss. Dabei muss es sich jedes Mal neu mit dem Netzwerk verbinden. Das Optimierungsziel ist es also, die Netzwerkeinwahl möglichst schnell und billig (im Sinn des Rechenaufwands und des resultierenden Energieverbrauchs) durchzuführen. Wenn die Ausschaltzeit lang ist und nur sehr wenige Daten übertragen werden, kann aber ein Mehrverbrauch durch die Einwahl in Kauf genommen werden.
- Low-power: Geräte fahren diese Strategie, wenn sie bei möglichst geringem Energieverbrauch dennoch am Netz erreichbar sein müssen, gegebenenfalls mit etwas Verzögerung (Latenz). In Fahrzeugen sind dies in der Regel Steuergeräte, die am Dauerplus hängen (Klemme 30) und durch Busaktivitäten geweckt werden. Optimierungsziel ist dabei ein geringer Energieverbrauch für die Minimalaktivität am Netzwerk.



**Abb. 6.12** Stark vereinfachter Aufbau eines IoT-Netzwerks

IoT Netze nutzen für diese Zwecke optimierte drahtlose oder drahtgebundene Protokolle, von denen in den Kap. 10 bis 13 die Rede sein wird, zuvor, in den Kap. 7 bis 9 werden die den meisten Mikroprozessoren eigenen seriellen Schnittstellen UART, SPI und I<sup>2</sup>C beschrieben, die den Brückenkopf zur Außenwelt darstellen. Die Topologie dieser Netze ist oftmals so aufgebaut, dass die Endgeräte (Netzknoten) nach den oben beschriebenen Strategien und mit möglichst geringen Datenraten einfache Verbindungen realisieren, gegebenenfalls (wie bei BlueTooth® Mesh) werden Netzknoten auch als Repeater genutzt. Die Datenpakete werden von einem Frontend-Gateway empfangen, in IP-Pakete eingepackt und über das Internet an einen Backend-Rechner weitergegeben, von wo die Datenkonzentration und Weiterverarbeitung möglich ist.

Eine stark vereinfachte Darstellung einer solchen Architektur ist in Abb. 6.12 zu sehen.

---

## Literatur

1. Meroth, A., & Tolg, B. (2008). *Infotainmentsysteme im Kraftfahrzeug. Grundlagen, Komponenten, Systeme und Anwendungen*. Vieweg.
2. ISO: ISO/IEC 7498-1: Information technology — Open systems interconnection — Basic reference model 1994, zugl. ITU-T Recommendation X.200 öffentlich. [https://standards.iso.org/ittf/PubliclyAvailableStandards/s020269\\_ISO\\_IEC\\_7498-1\\_1994\(E\).zip](https://standards.iso.org/ittf/PubliclyAvailableStandards/s020269_ISO_IEC_7498-1_1994(E).zip). Zugegriffen: 31. Dez. 2020.
3. Adolf, J. (2007). Schwab. Wolfgang Kürner: Elektromagnetische Verträglichkeit, Springer Verlag Berlin, Heidelberg.
4. RFC Editor: RFC Index. <https://www.rfc-editor.org/>. Zugegriffen: 31. Dez. 2020
5. Roppel, C. (2006). *Grundlagen der digitalen Kommunikationstechnik*. Carl Hanser.
6. NXP: CAN Bosch Controller Area Network (CAN) Version 2.0 PROTOCOL STANDARD <http://www.nxp.com/assets/documents/data/en/reference-manuals/BCANPSV2.pdf>. Zugegriffen: 1. Dez. 2016
7. ISO 11898: Road vehicles -- Controller area network (CAN) (6 Teile)

8. MQTT: Offizielle Webseite von MQTT. <https://mqtt.org/>. Zugegriffen: 1. Jan. 2021
9. Andrew, S. T., & David, J. (2012) Wetherall: Computernetzwerke (5., aktualisierte Aufl.). Pearson
10. Riggert, W., Märtin, C., & Lutz, M. (2015). *Rechnernetze – Grundlagen, Ethernet, Internet* (5., aktualisierte Aufl.). Hanser.
11. Bormann, N. et al. (2014). RFC 7228 terminology for constrained-node networks. <https://www.rfc-editor.org/rfc/rfc7228.html>. Zugegriffen: 31. Dez. 2020.



# Asynchrone serielle Schnittstellen

7

## Zusammenfassung

Dieses Kapitel beschreibt die asynchrone Schnittstelle der AVR Familie sowie das Prinzip der UART im Allgemeinen.

Nachdem im sechsten Kapitel die der Kommunikation zugrundeliegende Theorie erörtert wurde, widmen sich die drei folgenden Kapitel den Schnittstellen, die in typischen Mikrocontrollern und Sensoren, beziehungsweise in Netzwerkkomponenten verbaut sind. Der Klassiker ist die UART-Schnittstelle, die sicher schon etwas in die Jahre gekommen ist aber immer noch in vielen Anwendungen verwendet wird. In weiteren Kapiteln sind dann die SPI und die TWI/I<sup>2</sup>C beschrieben.

## 7.1 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 (TIA/EIA-232), aber auch die in der Industrie nach wie vor verbreiteten Protokolle TIA/EIA-422,

---

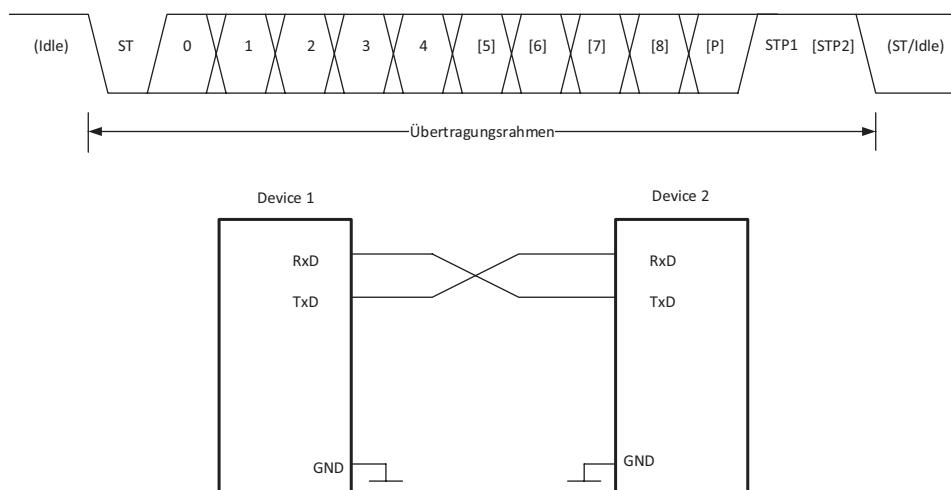
Die Originalversion dieses Kapitels wurde revidiert. Ein Erratum ist verfügbar unter  
[https://doi.org/10.1007/978-3-658-31709-6\\_27](https://doi.org/10.1007/978-3-658-31709-6_27)

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann  
[https://doi.org/10.1007/978-3-658-31709-6\\_7](https://doi.org/10.1007/978-3-658-31709-6_7).

TIA/EIA-485 und zum Teil auch MODBUS und Profibus. Auch andere Eindrahtbus-systeme, wie der im Automobil etablierte LIN-Bus basiert in Grundzügen auf der UART-Schnittstelle und kann einfach mit einem UART aufgebaut werden. Beachten Sie dazu die Kap. 11 und 12. 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 Spannungspiegel 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 darüber 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. 7.1 zu sehen, beginnt



**Abb. 7.1** Prinzipieller Aufbau eines UART Datenrahmens und der Beschaltung

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 verdoppelt<sup>1</sup>, sodass nur noch jeder zweite theoretisch mögliche Datenrahmen eine gültige Botschaft trägt. Damit kann pro Datenrahmen eine ungerade Zahl von fehlerhaften Bits erkannt werden. Die Ermittlungsvorschrift<sup>2</sup> für die Parität ist

$$P_{even} = d_{n-1} \oplus \dots \oplus d_3 \oplus d_2 \oplus d_1 \oplus d_0 \oplus 0 \quad (7.1)$$

$$P_{odd} = d_{n-1} \oplus \dots \oplus d_3 \oplus d_2 \oplus d_1 \oplus d_0 \oplus 1 \quad (7.2)$$

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. 7.1 zeigt zusätzlich die minimale Schaltung 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.

### 7.1.1 Hardwareanbindung in der AVR-Familie

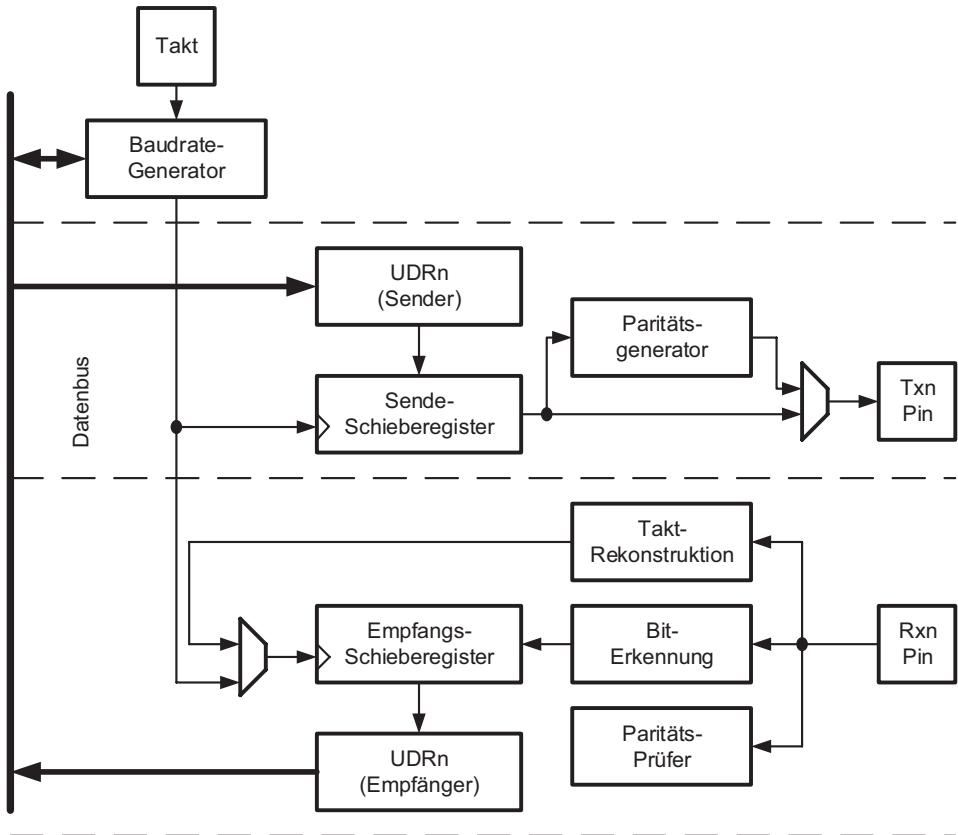
In der AVR-Familie ist mindestens ein USART auf jedem Prozessor vorhanden. An dieser Stelle soll nur der asynchrone Modus beschrieben werden. Die grundsätzliche interne Schaltung zeigt Abb. 7.2. Die USARTs sind bei AVR durchnummert, wir benutzen hier der Einfachheit halber immer den USART 0.

Die Datenrate wird beim ATMega88 im Baudrate-Generator durch

$$BAUD = \frac{f_{osc}}{16 \cdot (UBBR0 + 1)}$$

<sup>1</sup> Man spricht von einer „Hamming-Distanz“ von 2.

<sup>2</sup> Das Zeichen &#xF0C5; steht für eine Exklusiv-ODER-Funktion.



**Abb. 7.2** Prinzip des UART im asynchronen Modus bei der AVR-Familie

gebildet, wobei UBBR0 der Wert des gleichnamigen Baudratenregisters darstellt. Mit einer vorgegebenen Baudrate muss dieses also auf

$$UBBR0 = \frac{f_{osc}}{16 \cdot BAUD} - 1$$

gesetzt werden. Bei 9600 Baud und einer Quarzfrequenz von 18,432 MHz ist der Wert von UBBR0=119.

### 7.1.2 UART-Register beim ATmega 88

Neben dem Baudratenregister sind noch weitere Register für die Konfiguration und den Betrieb der USART-Schnittstelle notwendig. Das Sende- und das Empfangsregister der Schnittstelle sind auf derselben Adresse ansprechbar. Senden erfolgt durch Schreiben in das Register UDR0, nach einem erfolgreichen Empfangsvorgang sind die empfangenen

UCSROA	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Richtung	RXC0	TXC0	UDRE0	FE0	DOR0	UPE0	U2X0	MPCM0
Anfangswert	R	R/W	R	R	R	R	R/W	R/W
	0	0	1	0	0	0	0	0

UCSR0B	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Richtung	RXCIE0	TXCIE0	UDRIE0	RXENO	TXENO	UCSZ02	RXB80	TXB80
Anfangswert	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W
	0	0	0	0	0	0	0	0

UCSR0C	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Richtung	UMSEL01	UMSEL00	UPM01	UPM00	USBS0	UCSZ01	UCSZ00	UCPOLO
Anfangswert	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
	0	0	0	0	0	1	1	0

**Abb. 7.3** 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. 7.3).

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 Multiprozessor-Modus ein. In diesem Modus werden mehrere Empfänger parallelgeschaltet. Zunächst wird eine Adresse übertragen und nur derjenige Prozessor reagiert, der dieser Adresse zugeordnet ist.

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. UMP01 und UPM00 selektieren die Parität gemäß folgendem Schema (Tab. 7.1):

USBS0 gesetzt bedeutet „zwei Stoppbits“, nicht gesetzt „ein Stopppbit“. Schließlich geben die Bits UCSZ02...UCSZ00 (USCR0B) die Zahl der Datenbits an. 0 1 1 bedeutet

**Tab. 7.1** 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

„8 Bit“, das übliche Maß, s. Codebeispiel. Schließlich gibt UCPOL0 die Polarität des Taktes im synchronen Modus an. Der synchrone Modus ist in Kapitel 8.6 beschrieben.

### 7.1.3 Initialisieren der UART Schnittstelle beim ATmega88

Mit einem üblichen Wert von 8 Datenbits, keiner Paritätsprüfung und 2 Stopppbits 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 einer Quarzfrequenz 18,432 MHz und einer angestrebten 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>

```

Allerdings sollte man mit dem Macro-basierten Berechnen der Baudratenparameter aufpassen. Die Berechnung im Integer findet immer mit Abrunden ab, bei einer Quarzfrequenz von 18,432 MHz gehen die gängigen Baudraten ganzzahlig auf, bei anderen Frequenzen (z. B. 16 MHz) können die Rundungsfehler erhebliche Abweichung in der realen Baudrate erzeugen und es kann ein besseres Ergebnis erzielt werden, wenn man den Wert von Hand ausrechnet, auf- statt abrundet und direkt als Konstante einsetzt. Im Handbuch des ATmega 88 finden sich ausführliche Tabellen mit den optimalen Werten und deren Abweichungen von der idealen Frequenz.

### 7.1.4 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 10 ms 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 unbedachte 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 (hier eine globale Variable `data`) 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-Serviceroutine 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 (siehe Kap. 6).

Eine andere wichtige Bedeutung der Funktion `UART_CheckReceived()` liegt in der Kapselung der globalen Variablen. Die gesamte Funktionalität der UART-Schnittstelle kann damit in ein einzelnes, vorcompiliertes Modul ausgelagert werden, ohne dass die Anwendungssoftware, die diese nutzt, auf globale Variablen dieses Moduls zugreifen muss.

### 7.1.5 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 Stoppbits, 8 Datenbits = 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 anzeigt. 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.

### 7.1.6 Implementierung von uartWriteBuffer()

Eine praktische Möglichkeit, einen ganzen Datenpuffer zu versenden, bietet die Nutzung der ISR (USART\_TX\_vect). Liegen die Daten in einem Bytearray 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;
char *data;
#define TXCOMPLETE 1
#define TXNOTCOMPLETE 0
```

```

void uartWriteBuffer(char* databuf, unsigned char length)
{
    ucSendIndex=0;
    UDR0 = databuf[ucSendIndex++];
    ucBufLength = length;
    data=databuf;
    ucUART_Tx_Complete= TXNOTCOMPLETE;
}

ISR (USART_TX_vect)
{
    if (data[ucSendIndex] && ucSendIndex<ucBufLength)
        UDR0 = data[ucSendIndex++];
    else ucUART_Tx_Complete= TXCOMPLETE;
}

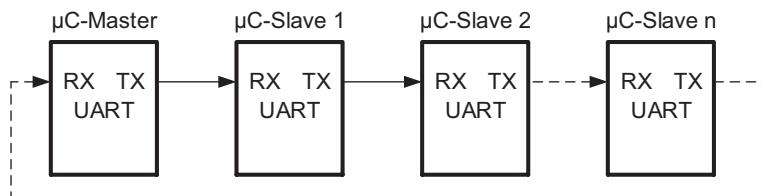
```

Eine einfache Kontrolle ob der Datenpuffer vollständig übertragen wurde, besteht im regelmäßigen Test von ucUART\_Tx\_Complete auf TXCOMPLETE. Achtung! Um die ISR USART\_TX\_vect zu starten muss natürlich das entsprechende Byte TXCIE0 im Register UCSR0B gesetzt sein! Das Array data[] muss global definiert sein, damit es in der ISR sichtbar ist. Der Speicherplatz muss natürlich in der Funktion reserviert werden, die uartWriteBuffer() aufruft.

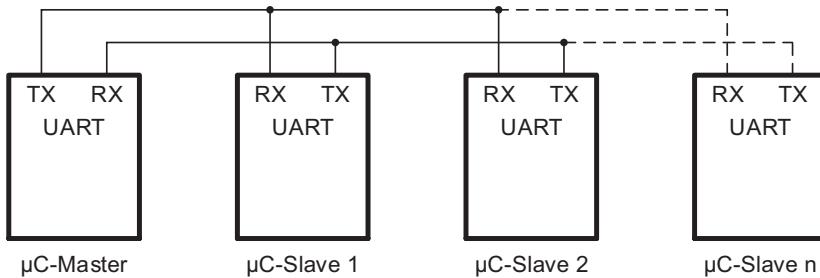
### 7.1.7 UART-Multiprozessor-Modus

Mehrere Mikrocontroller können über UART wie in Abb. 7.4 vernetzt werden um einen Daisy-Chain-Anschluss zu bilden. Dieser kann durch die Verbindung des letzten mit dem ersten Mikrocontroller in eine Ring-Topologie umgewandelt werden, die Kommunikation findet aber nur unidirektional statt.

Eine bidirektionale Kommunikation ermöglicht der UART-Bus in Abb. 7.5, der nach dem Master-Slave-Prinzip mit einem durch die Hardware festgelegten Master funktioniert. Abhängig von ihrer Rolle werden die Busteilnehmer an die zwei Datenleitungen mit vorgegebener Kommunikationsrichtung angeschlossen. Jede



**Abb. 7.4** UART- Daisy-Chain-Anschluss



**Abb. 7.5** UART – Vernetzung von mehreren Mikrocontrollern

Identifier	Länge des Datenfeldes	Datenfeld	Prüfsumme
------------	-----------------------	-----------	-----------

**Abb. 7.6** UART- Multiprozessor-Bus-Frame

Kommunikationssitzung, bestehend aus einem Master-Frame und eventuell der Frame-Antwort des adressierten Slaves wird vom Master eingeleitet. Der UART-TX-Anschluss des Masters ist mit den RX-Anschlüssen aller Slaves verbunden. An seinem RX-Anschluss kann der Master die von den Slaves gesendeten Nachrichten empfangen. Das UART-Protokoll in der hier vorliegenden Form sieht keine Arbitrierung vor, deshalb können die Slaves nur nach Aufforderung durch den Master ihre Nachricht senden. Ein Datenaustausch zwischen den Slaves dieses Busses ist ausgeschlossen. Die Kommunikation mit den einzelnen Slaves kann mittels expliziter oder inhaltsbasierender Adressierung stattfinden. Im ersten Fall wird jedem Slave eine systemweit (also für alle angeschlossenen Knoten) einmalige Adresse zugewiesen, im zweiten Fall werden die Nachrichten mit einem Identifier versehen. Über den Identifier kann der Master einen einzigen Slave oder mehrere gleichzeitig ansprechen. Wenn mehrere Slaves gleichzeitig adressiert werden, darf der Identifier keine Antwort auslösen.

Die Zusammensetzung eines Frames könnte wie in Abb. 7.6 aussehen. Im folgenden Beispiel ist der Identifier ein Byte groß und nimmt Werte im Bereich 0...0x3F an. Das folgende Byte gibt die Anzahl der Datenbytes an und somit kann die Kommunikation variabel gestaltet werden. Die 1-Byte-Summe aller Bytes einschließlich der Datenbytes wird am Ende des Frames als Prüfsumme angefügt. Die Frames können in einer Datenstruktur `uart_mp_frame` wie im folgenden Programmcode gespeichert werden:

```
typedef struct
{
    uint8_t ucID; //Identifier
    uint8_t ucLength; //Datenfeldlänge
    uint8_t ucData[8]; /*Datenvektor für die maximale Nachrichten-
    läng*/
```

`}uart_mp_frame;`

Eine derartige Vernetzung der Mikrocontroller über die standardmäßige UART hat den Nachteil, dass alle Nachrichten, die der Master sendet, auch von allen Slaves empfangen werden müssen, was zu deren Überlastung führen kann.

### 7.1.7.1 Initialisierung der Busteilnehmer im UART-Multiprozessor-Modus

Um das zu vermeiden, können die Mikrocontroller der Familie ATmega in ein UART-Multiprozessor-Netz integriert werden. Aus dem gesamten Master-Frame löst im Multiprozessor-Modus allein der Identifier bei allen Slaves einen UART-RX-Interrupt aus. Die weiteren Bytes eines solchen Frames lösen einen Interrupt nur bei den adressierten Slaves aus. Die Initialisierungen des Masters und der Slaves sind unterschiedlich, können aber in einer einzigen Funktion zusammengefasst wie es im folgenden Programmcode für einen ATmega88 veranschaulicht ist. Ein globales Array ucBaudrate wird mit den Werten für UBBR0 gefüllt, die Indexe könnten dann so aussehen (für 18,432 MHz):

```
#define BAUDRATE_2400 0
#define BAUDRATE_4800 1
#define BAUDRATE_9600 2
#define BAUDRATE_14400 3
#define BAUDRATE_19200 4

unsigned int ucBaudrate[] =
{479,239,119,79,59};
```

Beim Aufruf dieser Funktion muss als Parameter ucbaudrate – also der Index der Baudrate im genannten Feld – übergeben werden, sowie die Rolle des Mikrocontrollers im Netz: Master oder Slave. Über den gleichen Index wird auch die Timeout-Zeit eingestellt. Ein Timeout wird benötigt, um eine Zeitüberschreitung bei der Übertragung der einzelnen Bytes zu signalisieren und als Folge den Empfang einer Nachricht abzubrechen.

```
void UART_MP_Init(uint16_t ucbaudrate, uint8_t ucposition)
{
    UBRR0 = ucBaudrate[ucbaudrate]; //die Baudrate wird eingestellt
    UCSR0B = 1 << UCSZ02; //neun Datenbits
    UCSR0C = (1<<UCSZ00) | (1 << UCSZ01);
    UCSR0B |= 1 << RXEN0; //der Empfänger wird eingeschaltet und
    UCSR0B |= 1 << RXCIE0; //der Empfangsinterrupt aktiviert
    //der Sender des Masters wird aktiviert
    if(ucposition == MASTER) //UCSR0B |= 1 << TXENO;
    /*das MPCM0 Bit wird für den Slave aktiviert um IDs empfangen zu können/
    else if(ucposition == SLAVE) // UCSR0A |= (1 << MPCM0);
    ucPosition = ucposition;
```

```

    UART_MP_Init_IdFilter(); /*das Empfangsfilter wird
    initialisiert*/
    //der Timer 0 für den Timeout wird initialisiert
    TimeOut_Init(ucbaudrate);
}

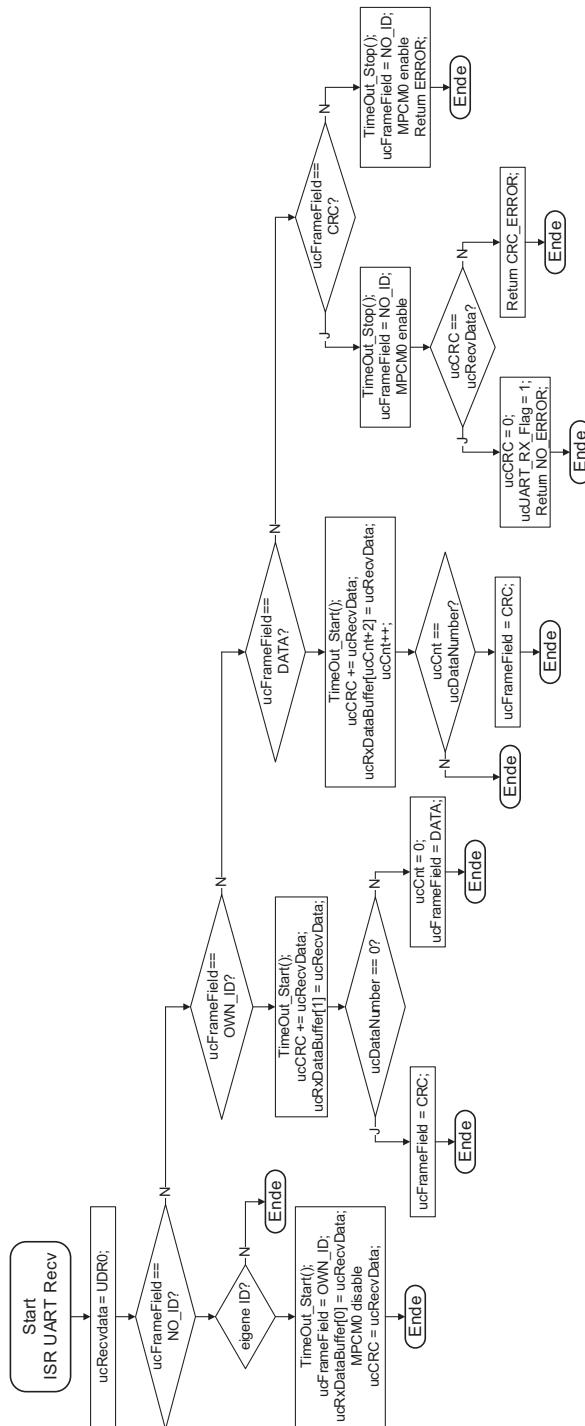
```

Die Kommunikation wird für neun Datenbits pro UART-Frame initialisiert und der Empfang eines Datenbytes muss bei allen Busteilnehmern einen RX-Interrupt auslösen. Für den Master wird zusätzlich der Sender und für die Slaves der Multiprozessor-Modus aktiviert. Die Funktion initialisiert eine Liste mit Identifiern, auf die der Busteilnehmer reagiert und einen Timer, um Timeout-Fehler während der Übertragung zu detektieren. Wenn das MPCMO Bit gesetzt und die 9-Bit-Übertragung aktiviert ist, wird bei den Empfängern ein RX-Interrupt nur dann ausgelöst, wenn das empfangene neunte Bit „1“ ist. Um einen Identifier zu senden, setzt der Master zuerst das Bit TX8B0 in das Register UCSR0B und dann, mit dem Speichern des 8-Bit-Identifiers in das Datenregister UDR0, beginnt die Übertragung des UART-Frames. Die Slaves vergleichen den empfangenen Identifier mit der eigenen Liste und bei Übereinstimmung setzen sie das Bit MPCMO zurück. Nach der Übertragung des Identifiers setzt der Master das Bit TX8B0 zurück und sendet die weiteren Bytes, die nur bei den adressierten Slaves einen UART-RX Interrupt auslösen. Wenn gefordert, aktiviert der adressierte Slave seinen UART-TX und sendet sein Frame, das aufgrund der Verdrahtung wie oben beschrieben nur vom Master empfangen werden kann. Anschließend deaktiviert er den UART-TX und aktiviert wieder den Multiprozessor-Modus.

### 7.1.7.2 Datenempfang im UART-Multiprozessor-Modus

Der gesamte Empfang einer Nachricht bei höheren Übertragungsraten muss im UART-RX-Interrupt stattfinden. In Abb. 7.7 wird beispielhaft das Flussdiagramm einer Interrupt-Serviceroutine dargestellt, die als Zustandsautomat konzipiert ist. Mit dieser Routine wird eine Master-Nachricht empfangen, in einen Puffer gespeichert und einen Code generiert der zeigt ob der Empfang fehlerfrei war. Man unterscheidet vier Zustände:

- **NO\_ID** – in diesem Zustand befinden sich die Slaves die auf einen Identifier warten. Beim Empfang eines Identifiers aus der eigenen Liste wird in den Zustand OWN\_ID geschaltet.
- **OWN\_ID** – das Byte das in diesem Zustand empfangen wird, gibt die Datenfeldlänge an. Wenn die Länge null ist, wird in den Zustand CRC geschaltet, ansonsten in den Zustand DATA.
- **DATA** – wenn die Zahl der, in diesem Zustand empfangenen Bytes die Datenfeldlänge erreicht, springt das System in den Zustand CRC. Beim Empfang eines Bytes in den ersten drei Zuständen wird ein Timeout-Timer gestartet und erst beim Empfang der Prüfsumme gestoppt. Wenn ein Byte nicht innerhalb der eingestellten Zeit



**Abb. 7.7** Zustandsübergangsdiagramm für den UART-RX-Interrupt im UART-Multiprozessor-Modus

empfangen wurde, wird ein TIMEOUT\_ERROR generiert und die Schnittstelle wird zurückgesetzt und für den nächsten Empfang vorbereitet.

- **CRC** – das empfangene Byte wird als Prüfsumme betrachtet und mit der berechneten Summe aller Bytes dieser Nachricht verglichen. Bei Übereinstimmung wird MESSAGE\_OK, ansonsten CRC\_ERROR generiert und die Schnittstelle wird für den Empfang der nächsten Nachricht vorbereitet.

In der main-Funktion wird im Polling der Empfang einer neuen Nachricht, bzw. der Auftritt eines Fehlers geprüft. Im Fall eines erfolgreichen Empfangs, gibt die Funktion UART\_MP\_Check\_Message (&sRecFrame) beim Aufruf MESSAGE\_RECEIVED zurück und speichert die Nachricht in die Variable sRecFrame die von Datentyp uart\_mp\_frame sein muss. Beim Auftritt eines Fehlers gibt die Funktion UART\_MP\_get\_Error beim Aufruf den Identifier der letzten, fehlerhaften Nachricht und die Zahl der Fehler zurück.

```
uint8_t UART_MP_Check_Message(uart_mp_frame *sframe)
{
    if(!ucUART_Rx_Flag) return NO_MESSAGE; //fehlerhafte Nachricht
    ucUART_Rx_Flag = 0; //der Empfangsflag wird zurückgesetzt
    sframe->ucID = ucRxDataBuffer[0];
    sframe->ucLength = ucRxDataBuffer[1];
    for(uint8_t ucI = 0; ucI < ucRxDataBuffer[1]; ucI++)
    {
        sframe->ucData[ucI] = ucRxDataBuffer[ucI + 2];
    }
    return MESSAGE_RECEIVED; /*eine fehlerfreie Nachricht wurde
    empfangen*/
}

uint8_t UART_MP_Get_Error(uart_mp_errorframe *sframe)
{
    if(!ucFaultReception) return NO_ERROR; //keine Empfangsfehler
    sframe->ucLastError = ucFaultReception;
    sframe->ucErrCnt = ucErrorCnt;
    ucFaultReception = NO_ERROR;
    return ERROR;
}
```

### 7.1.7.3 Senden von Frames im UART-Multiprozessor-Modus

Das uart\_mp\_frame wurde beispielhaft für maximal acht Datenbytes deklariert. Vor dem Senden, müssen im Sendeframe (sSendFrame) die aktuellen Werte gespeichert werden wie im folgenden Beispiel:

```
sSendFrame.ucID = 0x28; //speichern der aktuellen ID
sSendFrame.ucLength = 2; //es werden zwei Datenbytes gesendet
sSendFrame.ucData[0] = 'S'; //das erste Byte
sSendFrame.ucData[1] = '2';
UART_MP_Send_Message(&sSendFrame, SLAVE); //ein Slave sendet das Frame
```

Der Aufbau der Sendefunktion eines Frames im UART-Multiprozessor-Modus wird im folgenden Programmcode veranschaulicht. Als Parameter werden die Adresse der Datenstruktur, die die Elemente des Frames speichert und die Rolle des Busteilnehmers im Bus: Master oder Slave übergeben. In der Funktion wird die Prüfsumme berechnet und diese wird als letztes Byte des Frames gesendet.

```
void UART_MP_Send_Message(uart_mp_frame *sframe, uint8_t ucposition)
{
    //die Variable wird die Prüfsumme des Frames speichern
    uint8_t ucCRC = 0;
    if(ucposition == SLAVE)
    {
        //der UART-Sender des Slaves wird eingeschaltet
        UART_MP_Set_On(SEND_ON);
        MPCM_SET_OFF; //das MPCM Bit wird zurückgesetzt
        TimeOut_Start(Delay_YES);
    /*es wird gewartet bis die Leitung einen stabilen Zustand erreicht.
    Nur so wird gewährleistet, dass der Master das Startbit erkennt*/
        while(TimeOut_Get_State() == TIMEOUT_RUNNING);
    }
    else if(ucposition == MASTER)
    {
        //das Bit TXB80 wird gesetzt um die ID zu senden
        UART_MP_Set_On(TXB8_ON);
    }
    UART_MP_Send_Char(sframe->ucID); /*Master oder Slave senden
    die ID*/
    ucCRC += sframe->ucID;
    //das Bit wird zurückgesetzt, damit der Master Daten sendet
    //für den Slave ist der Wert dieses Bits irrelevant
    UART_MP_Set_Off(TXB8_OFF);

    UART_MP_Send_Char(sframe->ucLength); /*Die Datenfeldlänge wird
    gesendet*/
    ucCRC += sframe->ucLength;
    for(uint8_t i = 0; i < sframe->ucLength; i++)
    {
```

```
        UART_MP_Send_Char(sframe->ucData[i]); /*die Daten  
        warden gesendet*/  
        ucCRC += sframe->ucData[i];  
    }  
    UART_MP_Send_Char(ucCRC); //die Prüfsumme wird gesendet  
    if(ucposition == SLAVE)  
    {  
        /*das UART Sendeteil des Slaves wird abgeschaltet um  
        Buskollisionen zu vermeiden*/  
        UART_MP_Set_Off(SEND_OFF);  
        /*das MPCM Bit wird gesetzt, damit der Slave die  
        nächste ID empfangen kann*/  
        MPCM_SET_ON;  
    }  
}
```

---

## 7.2 Anbindung der seriellen Schnittstelle an USB

Für die Kommunikation des Prozessors mit einem PC kann die serielle Schnittstelle unter Verwendung des USB genutzt werden. Hierzu ist die Umsetzung des UART auf USB notwendig. Die gängigste Variante dafür ist die Verwendung eines Bausteins von FTDI. 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 [1] abgerufen werden.

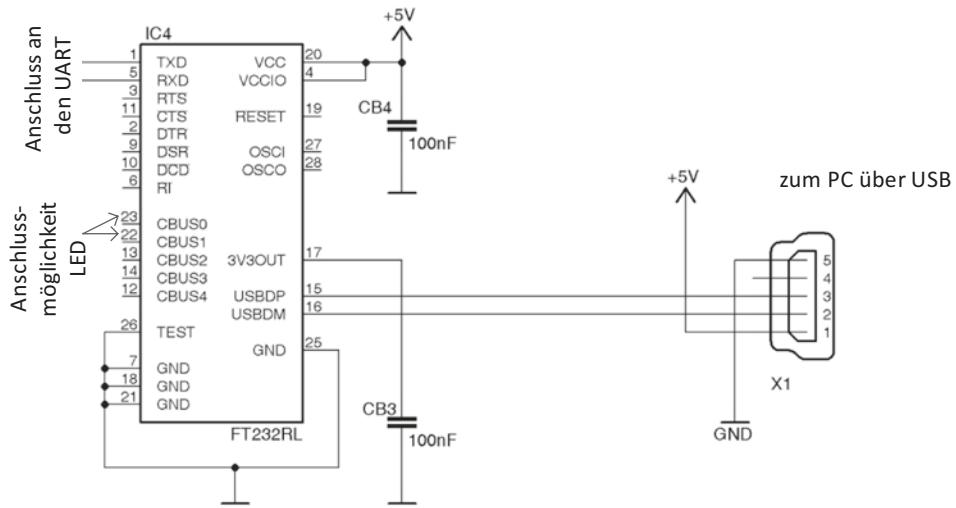
Abb. 7.8 zeigt die einfachste mögliche Beschaltung des FT232R. Darüber hinaus lassen sich noch LEDs anschließen, die die Kommunikationszustände anzeigen, sowie weitere digitale Ein/Ausgänge schalten. Außerdem verfügt der Chip über die Möglichkeit, einen Hardwarehandshake durchzuführen.

---

## 7.3 Ein einfaches serielles Protokoll

In diesem Abschnitt wird ein einfaches Protokoll beschrieben, das die folgenden, weitgehend in Kap. 6 beschriebenen Merkmale vereint:

- Framing, d. h. ein Anfangs- und Endebyte
- Eine Befehlsadressierung
- Einen Paketzähler zur Detektion ausgefallener Datenpakete
- Eine Prüfsumme



**Abb. 7.8** Einfachste Beschaltung des FT232R

Um den Code überschaubar zu halten, werden nur zwei Datenbyte übertragen, beispielsweise vom 16-Bit-AD-Wandler. Protokolle dieser Art finden sich in vielen Sensoren, die im Folgenden beschrieben sind. Das Protokoll stützt sich auf die byteweise Übertragung von Steuer- und Inhaltsbytes. Ein Frame beginnt immer mit einem ASCII SOH (0x01), dem sich eine Adresse anschließt. Im Minimalfall muss der Empfänger also nur auf ein SOH warten und die Adresse lesen. Ist die Adresse gültig, schließt sich ein Kommando an. In diesem Fall haben wir nur ein einziges Kommando ausgesucht um die Statemachine nicht aufzublähen: 0xC0. Dieser Wert ist willkürlich gewählt und bedeutet in unserem Beispiel „Temperatur folgt“. Es könnte auch Kommandos geben wie: „Einschlafen“, „Kalibrieren“ oder was auch immer notwendig ist.

Anschließend nach genau diesem Kommando folgen zwei Datenbytes, für die natürlich klar gestellt sein muss, ob das höherwertige Byte zuerst (Motorola-Format, big-endian, MSB first) oder das niedriger wertige Byte zuerst (Intel-Format, little-endian, LSB first) übertragen wird. Dies ist Thema in allen Protokollen! Wir entscheiden uns für big-endian.

Im Anschluss an die Datenbytes wird ein Botschaftenzählerbyte gesendet, das nach 256 Botschaften natürlich von 0 startet. Hier soll offen bleiben, was bei einem Datenverlust unternommen wird, dies hängt an der Applikation. Datenbytes und Botschaftenzähler bilden gemeinsam die Prüfsumme, in diesem Fall ist das eine einfache arithmetische Summe (statt einer CRC durchaus üblich), die nach 8 Bit abgeschnitten wird. Den Abschluss des Frames bildet ein ASCII EOT (0x04).

Eine Botschaft (Daten 0x12 0x34, Botschaftszähler 0x56) könnte also lauten 0x01 0xFF 0xC0 0x12 0x34 0x56 0x9C 0x04

Der Sender muss ein Bytefeld der Länge acht bereitstellen, das mit der Sendefunktion aus Abschn. 7.1.6 versendet wird. Aufgefüllt werden letztlich nur die Daten (Byte 4 und 5), der Zähler (Byte 6) und die Prüfsumme (Byte 7).

Der Empfänger bildet sich als Zustandsautomat ab, der auf zwei Ereignisse reagiert: Ein Byte wurde empfangen oder im letzten Schritt wurde ein Fehler detektiert. Aufgerufen wird der Zustandsautomat über ein Polling in der Hauptschleife (Abb. 7.9).

Den Empfängercode kann man über den Code realisieren, wie er in Abschn. 5.2 beschrieben ist.

Für die Nachrichtenbytes definieren wir im Empfänger

```
#define SOH 0x01
#define MY_ADDRESS 0xFF
#define READ_INTEGER 0xC0
#define EOT 0x04
```

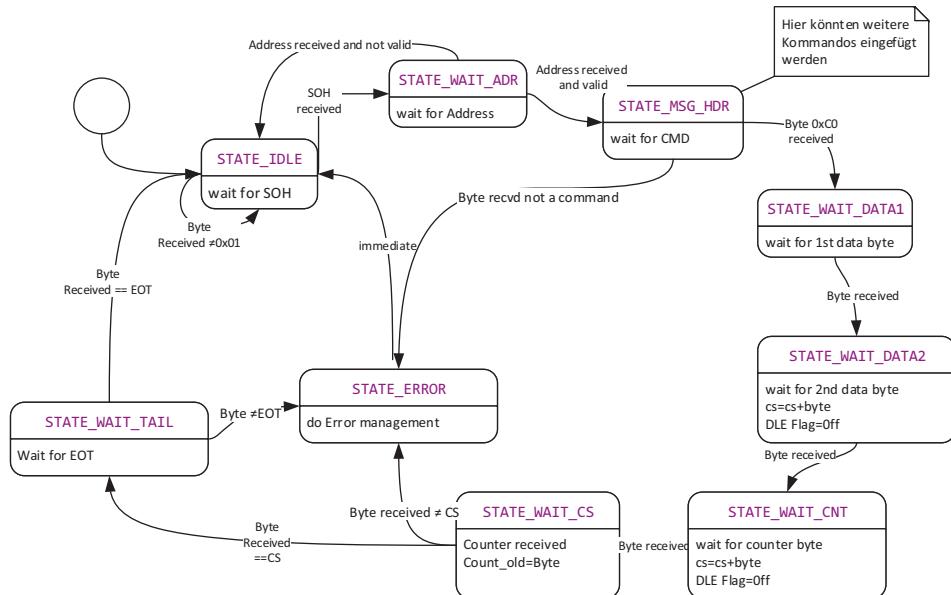


Abb. 7.9 Zustandsautomat für ein fiktives serielles Protokoll

Für die Zustände

```
#define STATE_IDLE 1
#define STATE_WAIT_ADR 2
#define STATE_MSG_HDR 3
#define STATE_WAIT_DATA1 4
#define STATE_WAIT_DATA2 5
#define STATE_WAIT_CNT 6
#define STATE_WAIT_CS 7
#define STATE_WAIT_TAIL 8
#define STATE_ERROR 9
```

Und für die Fehlercodes

```
#define ERROR_WRONG_CHECKSUM 3
#define ERROR_WRONG_COMMAND 2
#define ERROR_FRAME_ERROR 1
#define ERROR_NO_ERROR 0
```

Wir benötigen einige globale Variablen

```
static unsigned char ucState; //aktueller Zustand
static unsigned char ucChecksum; //Prüfsumme
static unsigned char ucError, ucSuccess; //Fehler oder Erfolg
short uiWordToReceive; // zu empfangende Daten
```

In der Hauptschleife (nachdem selbstverständlich alle Variablen und die serielle Schnittstelle initialisiert wurden, wie es in Abschn. 7.1.3 und folgende erklärt wurde), wird in der Hauptschleife die State machine ausgeführt. Diese wird entweder durch ein eingehendes Byte getriggert oder automatisch, wenn sie mit einem Fehler aus dem letzten Aufruf zurückgekommen ist:

```
while (1)
{
    if (uartCheckReceived() || error)
    {
        if (!error) ev=uartGetReceived();
        error = StateMachine(ev);
    }
    if (ucSuccess)
    {
        // Nachricht wurde komplett und fehlerfrei empfangen
        // Code hier ergänzen
    }
}
```

Der Zustandsautomat sieht wie folgt aus:

```
unsigned char StateMachine(unsigned char event)
{
    switch(ucState)
    {
        case STATE_IDLE:
            if (event!=SOH) ucState=STATE_IDLE;      // Zustand 1 -
            Idle bleibt
            else ucState=STATE_WAIT_ADR;           // SOH empfangen:
            Warte auf die Adresse
            ucSuccess=0;                          // Das ist
            der Beginn einer Botschaft
            break;
        case STATE_WAIT_ADR :
            if (event!= MY_ADDRESS) ucState=STATE_IDLE; //Adresse
            war nicht für uns
            else ucState= STATE_MSG_HDR; // Adresse ok, warte auf
            ein Kommando
            break;
        case STATE_MSG_HDR :
            if (event!= READ_INTEGER) // Kommando unbekannt
            {ucError= ERROR_WRONG_COMMAND; ucState=STATE_ERROR; }
            else ucState= STATE_WAIT_DATA1;           //
            Kommando bekannt, warte auf die Daten
            break;
        case STATE_WAIT_DATA1 :
            ucState=STATE_WAIT_DATA2;
            // Daten werden gelesen
            ucChecksum +=event;                     //
            Prüfumme wird aktualisiert
            uiWordToReceive=((short)event)<<8; // Erstes Byte ins
            Ergebnis schre
            break;                                // Big endian
            Format
        case STATE_WAIT_DATA2 :
            ucState=STATE_WAIT_CNT;                 // Daten werden
            gelesen
            ucChecksum +=event;                   // Prüfsumme wird
            aktualisiert
            uiWordToReceive|=((short)event); // Zweites Byte ins
            Ergebnis schreiben
            break;                                // Warte auf Zähler
        case STATE_WAIT_CNT : // Hier steht dann bei Bedarf die
            Behandlung des Zählers
```

```

        ucState=STATE_WAIT_CS;                      // Zähler
        gelesen
        ucChecksum +=event;                         // Prüfsumme wird
        aktualisiert
                                            // Warte auf Prüfsumme
        break;
    case STATE_WAIT_CS :
        if (event!=ucChecksum)                     // Prüfsumme
        falsch
        {ucError = ERROR_WRONG_CHECKSUM; ucState=STATE_ERROR
        else ucState=STATE_WAIT_TAIL; // Prüfsumme korrekt
        ucChecksum=0;
        break;
    case STATE_WAIT_TAIL :
        if (event!=EOT)
                                            // EOT Zeichen falsch
        { ucError=ERROR_FRAME_ERROR; ucState=STATE_ERROR; }
        else {ucState=STATE_IDLE; ucSuccess=1; ucError=ERROR_
        NO_ERROR; }
                                            // EOT korrekt, Nachricht komplett
        break;
    case STATE_ERROR : //do error handling
        ErrorHandling();                         // Mach
        was mit dem Fehler
        ucState=STATE_IDLE;
        return 1;                                // Liefere 1
        zurück, damit die SM wieder
                                            // getriggert wird, ohne,
                                            // dass ein Zeichen
                                            // empfangen wurde
        default: break;
    }
    return 0;                                  // Letztes Zeichen
    fehlerfrei abgearbeitet
}

```

### 7.3.1 Herstellung von Codetransparenz durch Bytestuffing

Etwas komplizierter wird der Code, wenn nach dem Kommando eine beliebige Anzahl von Bytes folgen kann. Dann sollte man noch ein Längenfeld einführen und bei Bedarf auch ein Bytestuffing, da die Daten natürlich auch den Wert 0x01 beinhalten könnten, der ja für den Header reserviert ist. In diesem Fall müsste man ein DLE (ASCII 0x10)

einschieben, sobald das Zeichen 0x01 im Datenfeld gesendet werden soll und natürlich auch, wenn das DLE-Zeichen selbst gesendet wird (siehe Kap. 6).

Ein Beispiel mit Bytestuffing ist (Daten 0x01 0x23, Botschaftszähler 0x21):

0x01 0xFF 0xC0 0x10 0x01 0x23 0x21 0x45 0x04

Das Bytestuffing wird natürlich nicht für die Berechnung der Prüfsumme herangezogen. Im Sender werden die entsprechenden Bytes ohne Bytestuffing in ein Feld der Länge 8 geschrieben. Das Bytestuffing findet in der bereits oben vorgestellten Sendeinterrupt-Serviceroutine statt, die dafür mit der globalen Variable `stuffed` entsprechend angepasst wird:

```
ISR(USART_TX_vect)
{
    if ((datBuf[ucSendIndex]) && (ucDataLen>ucSendIndex))
    {
        if (datBuf[ucSendIndex]==0x01 && !stuffed)
        {
            UDR0=0x10;
            stuffed=1;
        }
        else
        {
            UDR0 = datBuf[ucSendIndex++];
            stuffed=0;
        }
    }
}
```

Das Headerbyte bleibt vor dieser Maßnahme verschont, da es ja nicht mit DLE aufgefüllt werden muss! Mit der oben vorgestellten Funktion `uartWriteBuffer()` kann man die zuvor komplett erstellte Botschaft senden und das Stuffing wird automatisch durchgeführt. Im Empfänger müssen die DLEs natürlich im Zustandsautomaten wieder entfernt werden. Wenn keine DLEs vor einer 0x01 stehen, könnte der Zustandsautomat dies dann so interpretieren, dass ein unvollständiges Frame empfangen wurde und daher wieder an den Protokollanfang springen.

Der Zustandsautomat des Empfängers wird dahingehend erweitert, dass in allen Zuständen außer in `STATE_WAIT_ADR`, wo ja das Headerbyte empfangen wurde, nach Eingang eines Bytes geprüft wird, ob es sich beim eingehenden Byte um ein DLE gehandelt hat. In diesem Fall wird das Byte verworfen, der Zustand wird nicht verlassen und ein `StuffDetected`-Flag wird gesetzt. Beim nächsten eingehenden Byte und gesetztem Flag wird das Byte wieder übernommen und das Flag zurückgesetzt.

Eine Implementierung wird hier aus Platzgründen den Lesern und Leserinnen überlassen.

Ein weiteres serielles Protokoll finden Sie in Kap. 11 im Modbus.

---

## Literatur

1. Microchip: Reference Manual ATmega48/168. <https://www.microchip.com/wwwproducts/en/ATmega88A>. Zugegriffen: 6. Jan. 2021.
2. FTDI: FT232R – USB UART IC, online. <http://www.ftdichip.com/Products/ICs/FT232R.htm>. Zugegriffen: 5. Jan. 2021.



# Serial Peripheral Interface (SPI)

8

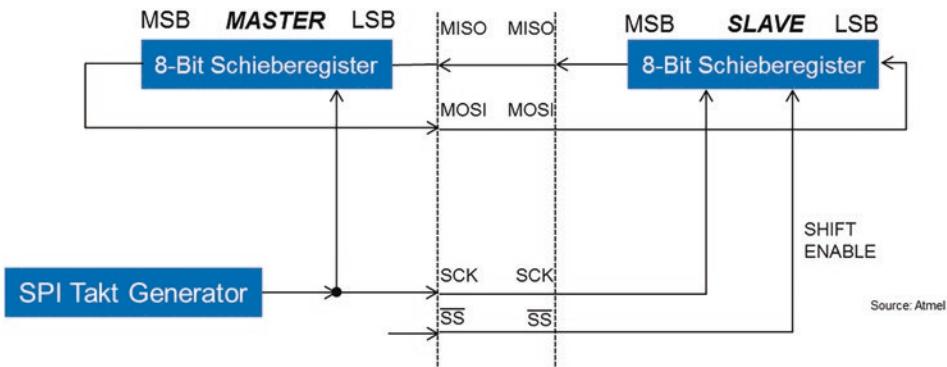
## Zusammenfassung

In diesem Kapitel wird die SPI Schnittstelle der AVR-Familie beschrieben.

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 Takteleitung (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 Originalversion dieses Kapitels wurde revidiert. Ein Erratum ist verfügbar unter  
[https://doi.org/10.1007/978-3-658-31709-6\\_27](https://doi.org/10.1007/978-3-658-31709-6_27)

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann  
[https://doi.org/10.1007/978-3-658-31709-6\\_8](https://doi.org/10.1007/978-3-658-31709-6_8).



**Abb. 8.1** Schemazeichnung der SPI-Schnittstelle. (Nach Microchip/ATMEL)

## 8.1 Aufbau und Funktionsweise

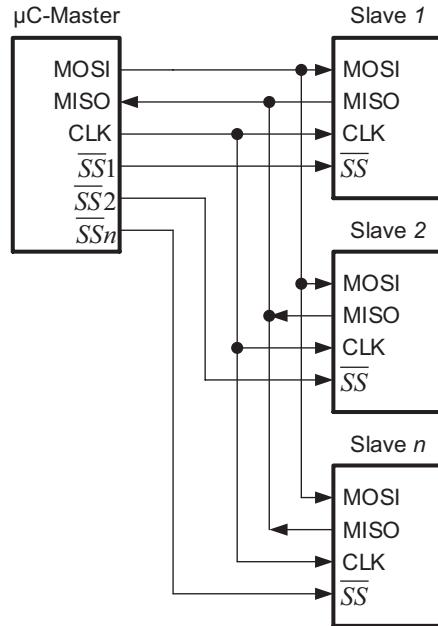
Die SPI Schnittstelle ist immer dann eine schnelle und sehr bequeme Kommunikations-schnittstelle, wenn es um den Betrieb einer Master-CPU mit mehreren untergeordneten Knoten (Slaves) geht. Der Takt geht dabei immer vom Master aus, insofern muss er beim Slave nicht konfiguriert werden. Der jeweilige Slave wird über die Slave-Select- ( $\overline{SS}$ ) 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. 8.1). 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, sodass 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. 8.2 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.

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

**Abb. 8.2** Schaltung mehrerer Sensoren über die SPI-Schnittstelle



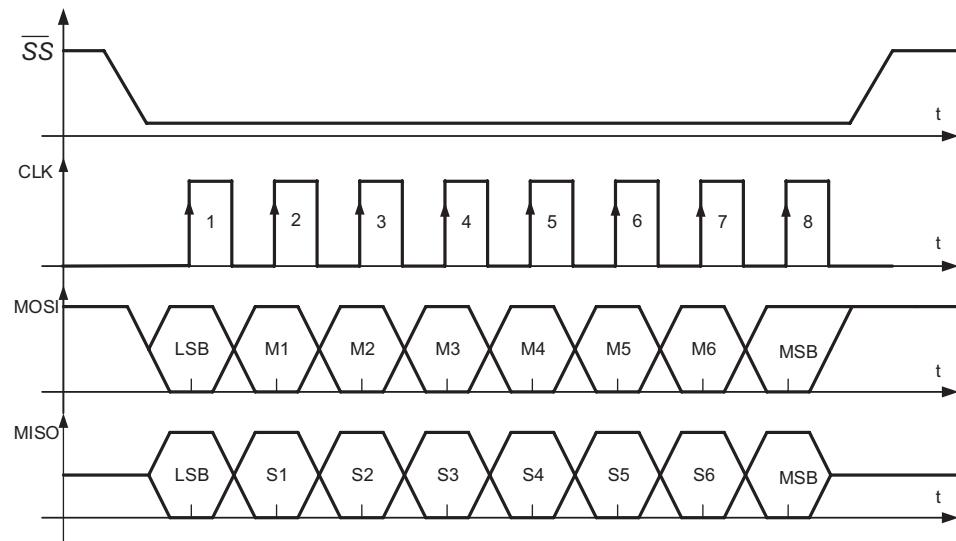
Sensorprotokolle zwischen dem Mikrocontroller und dem Sensor oft aus mehreren Datenworten, häufig wird zunächst ein Befehl an den Slave getaktet, anschließend wird eine 0x00 oder 0xFF nachgesendet, die dann dafür sorgt, dass die inzwischen bereitgestellten Daten des Sensors an den Mikrocontroller übertragen werden. In den folgenden Abschnitten finden sich einige Beispiele zu diesem Vorgehen.

## 8.2 Konfiguration der SPI-Schnittstelle

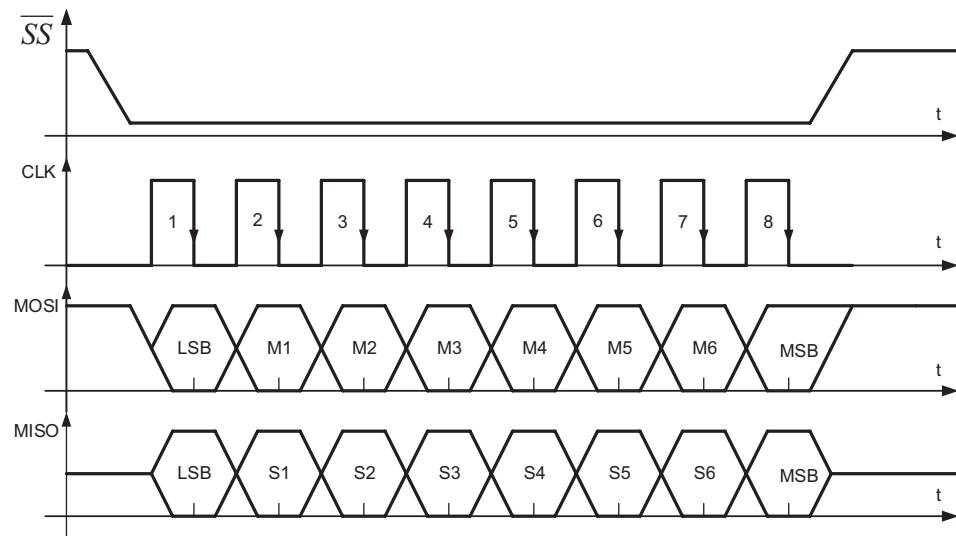
Um die Konfiguration der SPI-Schnittstelle zu verstehen, muss man sich zunächst über die Darstellung eines Bytes und die Art der Abtastung klarwerden. Bei Sensoren ist diese in der Regel festgelegt, manchmal lässt sie sich auch konfigurieren, in jedem Fall müssen Master und Slave sich auf gleiches Verhalten 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 Taktphase (CPHA)

unterscheiden (vergl. Abb. 8.3, 8.4, 8.5 und 8.6).

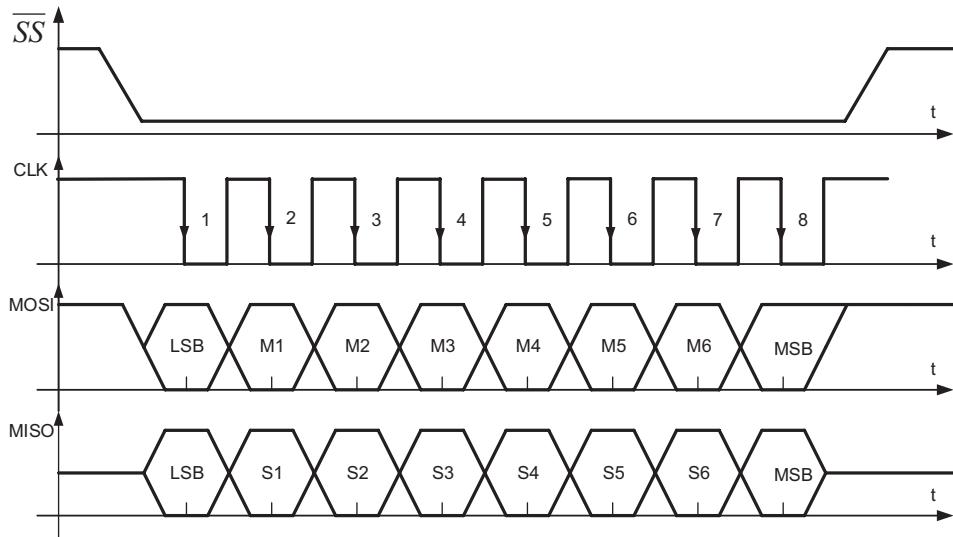
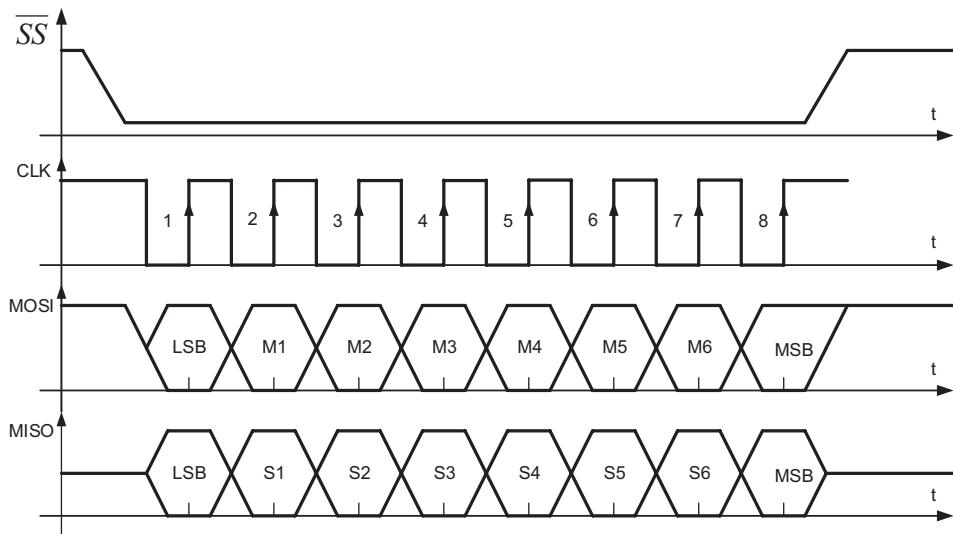


**Abb. 8.3** SPI-Modus 0: CPOL=0, CPHA=0 LSB first



**Abb. 8.4** SPI-Modus-1: CPOL=0, CPHA=1 LSB first

CPOL=0 bedeutet, dass der Takt den Ruhepegel 0 (Low) hat, bei CPOL=1 ist der Ruhepegel 1 (High). Die Taktphase 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.

**Abb. 8.5** SPI-Modus 2: CPOL = 1, CPHA = 0 LSB first**Abb. 8.6** SPI-Modus 3: CPOL = 1, CPHA = 1 LSB first

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 Taktsignals CLK das anliegende Signal abgetastet (Master tastet MISO ab, Slave tastet MOSI ab). Die darauffolgende umgekehrte Flanke veranlasst Master und Slave jeweils dazu, das nächste Bit auf die Datenleitungen zu geben, bis das gesamte Datenwort übertragen ist.

**Tab. 8.1** Taktraten im SPCR Register

SPR1	SPR0	Takt
0	0	$f_{osc}/4$
0	1	$f_{osc}/16$
1	0	$f_{osc}/64$
1	1	$f_{osc}/128$

(nach ATMEL)

Bei  $CPHA=1$  wird erst auf die zweite Taktflanke 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 angeschlossenen Slaves bleibt. In der AVR-Familie werden die Taktfrequenzen über die Bits SPR0 und SPR1 in SPCR-Register gesteuert wie in Tab. 8.1 dargestellt.

Über das Bit SPI2X (double SPI speed bit) im SPSR (SPI status register) können die Taktraten auf  $f_{osc}/2$ ,  $f_{osc}/8$ ,  $f_{osc}/32$  und weiterhin auf  $f_{osc}/64$  verdoppelt werden.

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 Flag SPIF 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)));

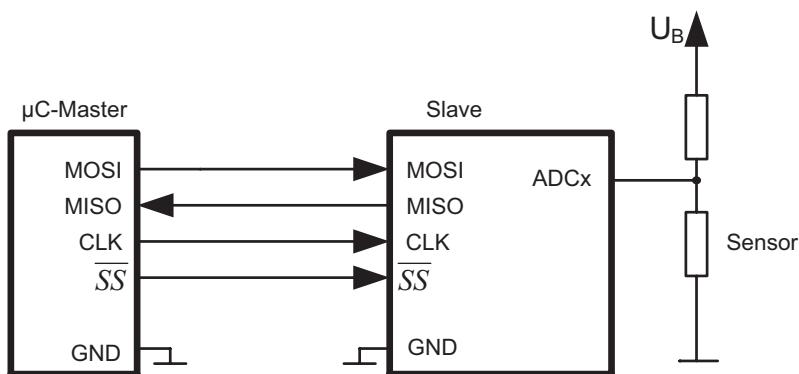
```

Alternativ sendet man aus einem zeitgesteuerten Task heraus, bzw. interruptgesteuert, bis der Sendepuffer leer ist. Nach dem Sendevorgang befindet sich gemäß Abb. 8.1 der Inhalt des Slavesenderegisters im Register SPDR. Insofern gibt es keine eigene Empfangsfunktion für die SPI-Schnittstelle.

---

### 8.3 SPI-Schnittstelle im Slavebetrieb

Im Slavebetrieb empfiehlt sich dringend die Nutzung der Interrupt-Serviceroutine der SPI Schnittstelle. An folgendem Szenario sei dies verdeutlicht (Abb. 8.7).



**Abb. 8.7** Nutzung eines Mikrocontrollers mit analogem Sensor im Slavebetrieb

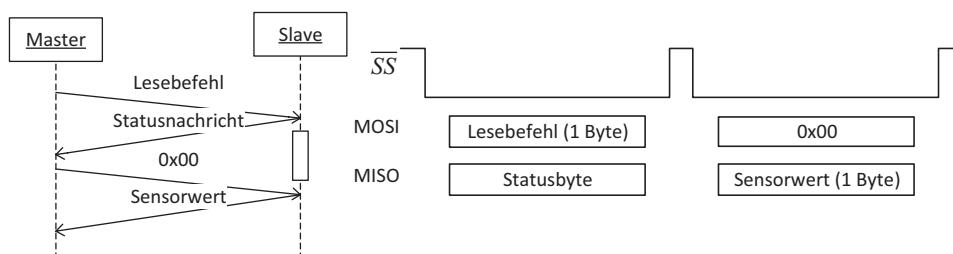
Der rechte Mikrocontroller ist im Slavebetrieb konfiguriert:

```
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. 8.7 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 Fehlerspeichereintrag existiert. Nach dem Befehlswort sendet der Master acht Nullen (0x00), danach liegt im Register SPDR der abzufragende Wert des Sensors bereit, dazu muss der Sensor ihn zwischen dem Befehlswort und der 0-Folge in sein eigenes Senderegister eingestellt haben.

Der prinzipielle Ablauf ist in Abb. 8.8 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



**Abb. 8.8** Ablauf einer SPI-Sequenz mit Befehlsbyte

dieses Beispiel angenommen, dass die CS-Leitung des Sensors auf Port B2 verdrahtet ist. Wir erweitern die Senderoutine im Master folgendermaßen:

```
#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. 8.7) 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 das immer der aktuelle Status eingeschrieben wird.

```
ISR(SPI_STC_vect)
{
    SPI_recByte = SPDR ;
    if (SPI_READSENSOR == SPI_recByte)
    {
        /* Beim nächsten Takt 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.

---

## 8.4 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 folgende Abb. 8.9 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 anzusteuernden 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 Anschluss 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_Master_SlaveSelectInit` auf High gesetzt. Als Aufrufparameter dieser Funktion wird das entsprechende Definitionsarray des Sensors benutzt. Der Code dieser Funktion ist hier aufgelistet:

```

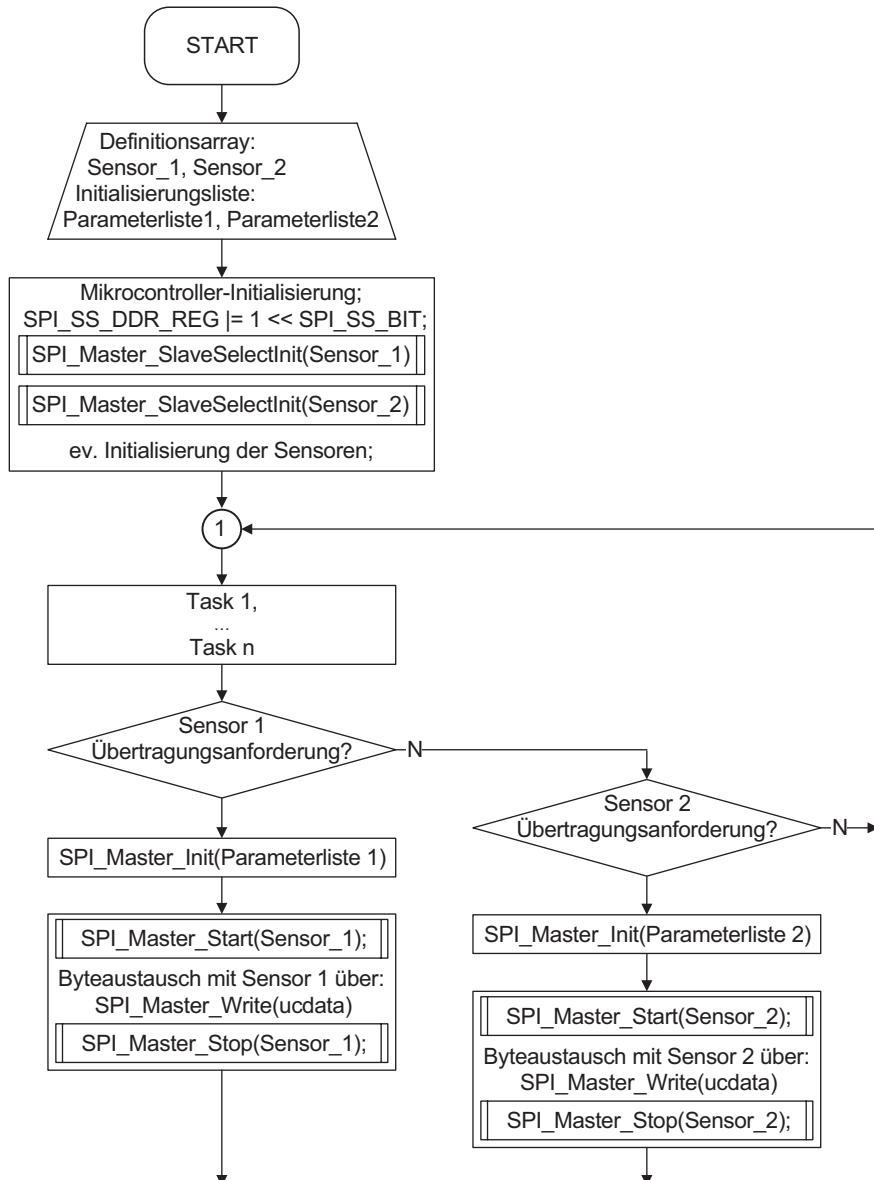
void SPI_Master_SlaveSelectInit(tspiHandle tspi_pins)
{
    //entsprechendes Bit im Data-Direction-Register wird auf High
    gesetzt (Output)
}

```

```

*tspi_pins.CS_DDR |= 1 << sdevice_pins.CS_pin;
// entsprechendes Bit im PORT-Register wird auf High gesetzt
*tspi_pins.CS_PORT |= 1 << sdevice_pins.CS_pin;
}

```



**Abb. 8.9** Ansteuerung zweier Sensoren mit unterschiedlichen SPI-Modi

Beispiele für konkrete Strukturen sind in den folgenden Kapiteln 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.

```
void SPI_Master_Init(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
    //μC wird als Master deklariert, die SPI-Schnittstelle wird
    freigegeben
    //die Bitreihenfolge wird bestimmt und der Übertragungsmodus
    wird bestimmt
    SPI_CONTROL_REGISTER = SPI_MASTER | SPI_ENABLE |
    ucspi_data_order |
                    (ucspi_mode << 2) | (ucspi_sck_freq % 4);
    //die gewünschte SPI-Taktfrequenz wird gewählt
    SPI_STATUS_REGISTER = ucspi_sck_freq / 4;
}
```

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
//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 uedata)
{
    #define SPI_DATA_REGISTER      SPDR //Definition in der
    SPI.h Datei
    SPI_DATA_REGISTER = uedata;
    while(SPI_RUNNING);
    return SPI_DATA_REGISTER;
}
```

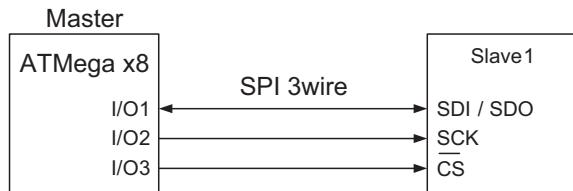
## 8.5 3-wire-SPI-Kommunikation

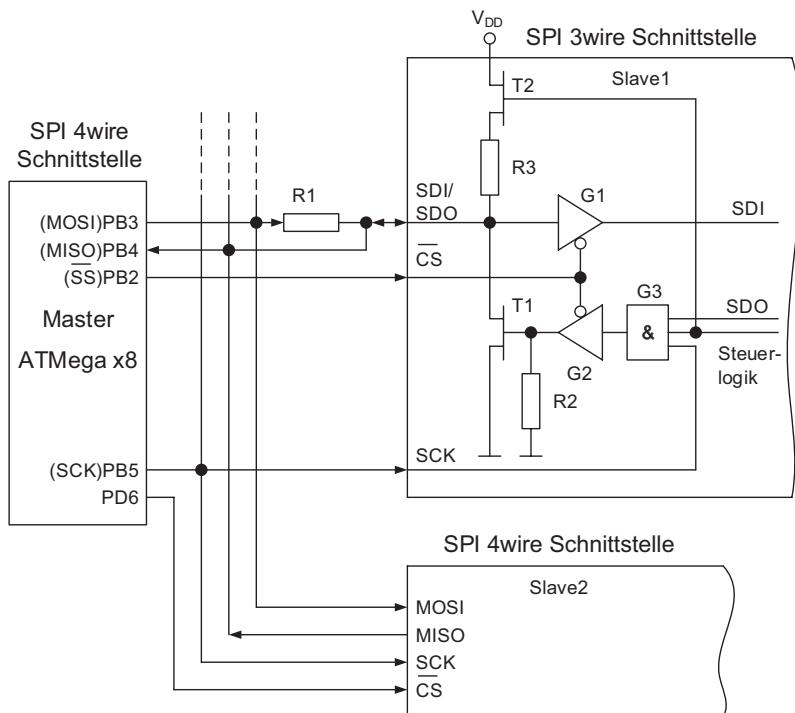
SPI wurde grundsätzlich als 4-Leitung-Bus entwickelt, der den synchronen Datenaustausch zwischen dem Master und einem Slave im Vollduplex-Modus ermöglicht. Aus Kostengründen werden einige Slaves mit einer 3-Leitung-SPI-Schnittstelle ausgestattet. Die Datenübertragung findet über einen einzigen Pin statt. Dadurch wird ein externer Anschluss eingespart, die Kommunikation kann dann jedoch nur noch im Halbduplex-Modus stattfinden. Die Ansteuerung eines solchen Slaves kann von einem Mikrocontroller über allgemeine I/O-Pins realisiert werden (siehe Abb. 8.10).

Bei der Ansteuerung eines 4-wire-SPI-Busses mit einer dedizierten Schnittstelle muss die Software auf der Seite des Masters nur die Ansteuerung der CS-Leitung, das Speichern des zu sendenden Bytes in das Register SPDR und das Lesen des empfangenen Bytes gewährleisten. Die Taktzeugung sowie die Bitübertragung werden von der Hardware realisiert. Mit einer minimalen Hardware-Änderung können die genannten Vorteile auch für die Ansteuerung von Slaves, die eine 3-wire-SPI-Schnittstelle besitzen, genutzt werden. Abb. 8.11 zeigt einen SPI-Bus, an dem Slave1 über nur drei Leitungen statt vier am Bus angeschlossen ist. Der MISO-Pin des Masters ist mit dem SDI/SDO-Pin des Slaves direkt verbunden, während zwischen dem MOSI und dem SDI/SDO Anschluss ein Widerstand geschaltet ist. Der Widerstand R1 verhindert, dass es zu einem Kurzschluss zwischen den Pins MOSI des Masters und SDI/SDO des Slaves kommen kann, während der Master ein Dummy-Byte sendet, um für den Empfang den Takt zu erzeugen. Gleichzeitig bleibt es aber möglich, dass das Signal, das am Anschluss SDI/SDO anliegt, über MISO gelesen werden kann.

Intern besitzen solche Slaves beide Signalleitungen SDI und SDO, die wie in Abb. 8.11 mit dem Anschluss SDI/SDO verbunden sind. Die internen Sende- und Empfangspuffer der 3-wire-Schnittstelle werden wie üblich erst aktiviert, wenn das CS-Signal auf low geht, ansonsten ist der Anschluss SDI/SDO hochohmig. Die 3-wire-SPI-Kommunikation kann nur im Halbduplex-Modus stattfinden, deshalb setzt der adressierte Slave1 am Anfang einer Kommunikationssequenz das Signal SDO auf „0“. Damit schaltet er in den Empfangsmodus und wartet auf einen gültigen Befehlscode, der die Kommunikationsrichtung bestimmt. Bei einem Schreibbefehl bleibt der SDI/SDO-Anschluss des Slaves auf Eingang und empfängt die folgenden Datenbytes. Bei einem Lesebefehl wird das Signal entsprechend der zu sendenden Bits synchron mit dem Takt signal geschaltet.

**Abb. 8.10** 3-wire-SPI over I/O's





**Abb. 8.11** 3-wire-SPI-Kommunikation

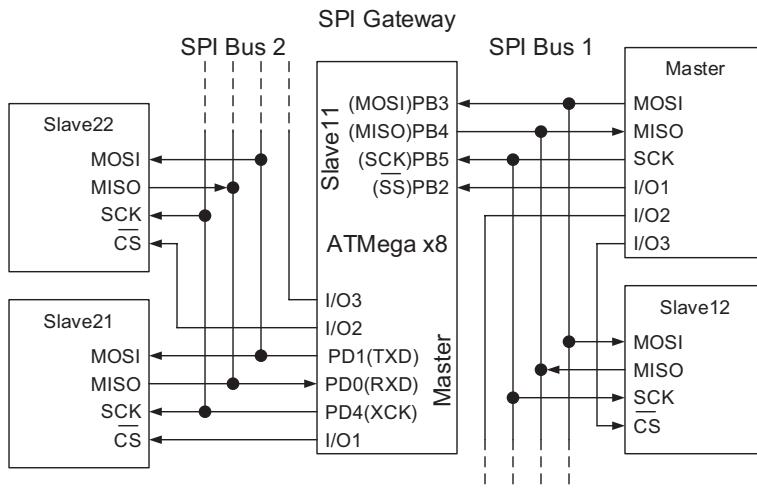
Sensoren wie ADXL312 (Kap. 16) und L3GD20 (Kap. 17) sind für die serielle Kommunikation sowohl mit einer I<sup>2</sup>C- als auch mit einer SPI-Schnittstelle ausgestattet. Die Wahl der aktiven Schnittstelle wird durch eine externe Beschaltung realisiert. Über ein internes Registerflag kann zusätzlich zwischen der voreingestellten 4-wire- und einer 3-wire-SPI-Kommunikation umgeschaltet werden. Der Master beginnt eine SPI-Kommunikation mit solchen Sensoren mit einem Startbyte. Das erste Bit dieses Bytes gibt die Übertragungsrichtung vor, während die letzten sechs Bits eine Registeradresse übertragen. Wenn das Read/Write-Bit „0“ ist, überschreibt der Master den Inhalt im Zielregister des Slaves, ansonsten fordert er von dem Sensor die Übertragung des adressierten Registers an.

In Kap. 24 wird die Kommunikation über eine 3-wire-SPI zwischen einem Mikrocontroller und einem digitalen Potentiometer vom Typ MCP4151 genauer beschrieben. Der Master sendet auch in diesem Fall ein Startbyte am Anfang der Kommunikation. Die ersten sechs Bits dieses Bytes bilden einen Befehlscode, der von dem Sender auf Gültigkeit geprüft wird. Bei Unstimmigkeit setzt der Slave das siebte Bit des Startbytes auf low, um dem Master den Fehler zu signalisieren.

## 8.6 SPI-Master über USART

In diesem Kapitel wurde bis jetzt die dedizierte SPI-Schnittstelle eines ATmega Mikrocontrollers beschrieben. Über diese Schnittstelle kann der Mikrocontroller in einem SPI-Netzwerk sowohl als Master als auch als Slave konfiguriert werden. Eine zweite SPI-Schnittstelle wäre nützlich, wenn man vorhat, verschiedene Slaves mit unterschiedlichen SPI-Modi anzusteuern. Dies ist insbesondere dann kritisch, wenn die jeweilige Neuinitialisierung der Schnittstelle zu zeitlichen Einschränkungen führen kann. Mit einer zweiten Schnittstelle könnte ein Mikrocontroller auch als SPI-Gateway zwischen einem übergeordneten SPI-Bus und einem untergeordneten SPI-Sensornetzwerk arbeiten, wie in Abb. 8.12 dargestellt.

Durch die Konfiguration der USART-Schnittstelle im Master-SPI-Modus (MSPIM) verfügen die Mikrocontroller ATmegaX8 sowie ATmega640/1280/2560 über eine zweite SPI-Schnittstelle zur Ansteuerung von SPI-Slaves. In Tab. 8.2 sind die Pins dieser Schnittstelle mit ihren Funktionen beschrieben. Es fehlt ein dedizierter Slave-Select-Eingang für diese Schnittstelle, deshalb kann sie nur für den Masterbetrieb konfiguriert werden.



**Abb. 8.12** ATmegaX8 als SPI-Gateway

**Tab. 8.2** Portpins der USART im MSPI-Modus konfiguriert

Pinname	SPI-Funktion
TXD	Master Out Slave In
RXD	Master In Slave Out
XCK	Clock

Die Register des USART zur Konfiguration des Mikrocontrollers im MSPI-Modus sind in Tab. 8.3 erläutert. Die Funktionen der aktiven Bits der Register UCSR0A und UCSR0B für diesen Betriebsmodus sind identisch mit denen im Kap. 7 beschrieben. Mit dem Setzen der Bits UMSEL00 und UMSEL01 aus dem Register UCSR0C wird die Schnittstelle als Master-SPI konfiguriert. Mit dem Setzen des Bits UDORD0 wird das LSB-Bit zuerst übertragen, die Kombination der Bits UCPOL0 und UCPHA0 bestimmt den SPI-Modus.

Für diese Schnittstelle wird hier ein allgemeines Softwaremodul erstellt, das wie im Abschn. 8.4 beschrieben, aufgebaut ist. Für die Ansteuerung der Chip-Select-Eingänge der Slaves wird analog zum obigen Vorgehen in der Headerdatei des Softwaremoduls eine Datenstruktur definiert:

```
typedef struct{

    volatile uint8_t* CS_DDR;
    volatile uint8_t* CS_PORT;
    uint8_t CS_pin;
    uint8_t CS_state;
} tmspimHandle;
```

Die Schnittstelle muss gemäß [1] vor ihrer Benutzung zuerst initialisiert werden wie beispielsweise in der folgenden Funktion:

```
void MSPIM_Master_Init(uint8_t ucspi_data_order, uint8_t ucspi_mode,
                      uint8_t ucspi_sck_freq)
{
    MSPIM_BAUDRATE_REGISTER = 0;
    //CLK-Pin wird auf Ausgang deklariert; Master Modus wird
    freigegeben
    MSPIM_CLK_DDR_REG |= 1 << MSPIM_CLK_BIT;
    //der Mikrocontroller wird als Master konfiguriert
    // die Bitreihenfolge und der SPI-Übertragungsmodus werden
    bestimmt
    MSPIM_CONTROL_REGISTER_C = MSPIM_MASTER | ucspi_data_order |
    ucspi_mode;
    //die USART als SPI-Schnittstelle wird freigegeben
    MSPIM_CONTROL_REGISTER_B = MSPIM_ENABLE;
    //die gewünschte SPI-Taktfrequenz wird eingestellt
    MSPIM_BAUDRATE_REGISTER = ucspi_sck_freq;
}
```

Durch die Freigabe des Senders und des Empfängers durch das Setzen der Bits TXEN0 und RXEN0 im Register UCSR0B werden die entsprechenden Pins TxD bzw. RxD

**Tab. 8.3** Die Register zur Konfiguration des USART im MSPI-Modus (nach Microchip)

		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
UCSR0A	RXC0	TXCO	UDRE0	—	—	—	—	—	—
Richtung	R	R/W	R	R	R	R	R	R	R
Anfangswert	0	0	0	0	0	1	1	1	0
<b>UCSR0B</b>	<b>Bit 7</b>	<b>Bit 6</b>	<b>Bit 5</b>	<b>Bit 4</b>	<b>Bit 3</b>	<b>Bit 2</b>	<b>Bit 1</b>	<b>Bit 0</b>	
RXCIE0	TXCIE0	UDRIE0	RXEN0	TXEN0	—	—	—	—	
Richtung	R/W	R							
Anfangswert	0	0	0	0	0	1	1	1	0
<b>UCSR0C</b>	<b>Bit 7</b>	<b>Bit 6</b>	<b>Bit 5</b>	<b>Bit 4</b>	<b>Bit 3</b>	<b>Bit 2</b>	<b>Bit 1</b>	<b>Bit 0</b>	
UMSEL01	UMSEL00	—	—	—	UDORD0	UCPHAO	UCPOL0		
Richtung	R/W	R	R	R	R/W	R/W	R/W	R/W	
Anfangswert	0	0	0	0	1	1	1	0	

automatisch auf Ausgang, bzw. Eingang konfiguriert. Eine explizite Konfiguration der zwei Pins ist nicht nötig. Die Aufrufparameter der Funktion `MSPIM_Master_Init()` für einen Mikrocontroller ATmegaX8 werden in der Headerdatei des Moduls folgendermaßen definiert:

```
//MSPIM Register
#define MSPIM_DATA_REGISTER UDR0
#define MSPIM_CONTROL_REGISTER_A UCSR0A
#define MSPIM_CONTROL_REGISTER_B UCSR0B
#define MSPIM_CONTROL_REGISTER_C UCSR0C
#define MSPIM_BAUDRATE_REGISTER UBRRO
//  

#define MSPIM_ENABLE 0x18 //Bits RXENO und TXENO im Register UCSR0B
#define MSPIM_MASTER 0xC0 //Bits UMSL00 und UMSL01 im Register UCSR0C
//UDORD0-Bit in das UCSR0C-Register (data order LSB or MSB first)
#define MSPIM_LSB_FIRST 0x04
#define MSPIM_MSB_FIRST 0x00
//SPI-Modus wird über 2 Bits bestimmt UCPOLO und UCPHA0 im UCSR0C-Register
#define MSPIM_MODE_0 0x00 //UCPHA0 = 0, UCPOLO = 0
#define MSPIM_MODE_1 0x02 //UCPHA0 = 1, UCPOLO = 0
#define MSPIM_MODE_2 0x01 //UCPHA0 = 0, UCPOLO = 1
#define MSPIM_MODE_3 0x03 //UCPHA0 = 1, UCPOLO = 1
//Taktfrequenz der SPI-Schnittstelle (FOSC = Taktfrequenz des Mikrocontrollers)
#define MSPIM_FOSC_DIV_2 0x00
#define MSPIM_FOSC_DIV_4 0x01
#define MSPIM_FOSC_DIV_8 0x03
#define MSPIM_FOSC_DIV_16 0x07
#define MSPIM_FOSC_DIV_32 0x0F
#define MSPIM_FOSC_DIV_64 0x1F
#define MSPIM_FOSC_DIV_128 0x3F
#define MSPIM_SENDING (! (UCSR0A & (1 << UDRE0))) // das Senden eines Bytes läuft
#define MSPIM RECEIVING (! (UCSR0A & (1 << RXC0))) //der Empfang eines Bytes läuft
//Clock-Pin
#define MSPIM_CLK_DDR_REG DDRD
#define MSPIM_CLK_PORT_REG PORTD
#define MSPIM_CLK_BIT PD4
```

Nach der Initialisierung funktioniert das USART wie die SPI-Schnittstelle. Mit dem Speichern eines Bytes in das Datenregister UDR0 werden die einzelnen Bits in der eingestellten Reihenfolge auf die MOSI-Leitung synchron mit dem, am Pin XCK erzeugten Takt geschoben. Gleichzeitig werden die von dem aktivierten Slave gesendeten Bits in den Eingangspuffer geladen. Der Datenaustausch zwischen dem Master und einem Slave findet mit dem Aufruf folgender Funktion statt.

```
uint8_t MSPIM_Master_Write(uint8_t uedata)
{
    while(MSPIM_SENDING);
    MSPIM_DATA_REGISTER = uedata;
    while(MSPIM_RECEIVING);
    return MSPIM_DATA_REGISTER;
}
```

Anders als bei der asynchronen Übertragung, wird die Bitrate bei der synchronen Übertragung mit Gl. 8.1 berechnet.

$$Baud = \frac{f_{osc}}{(UBRRn + 1) * 2} \quad (8.1)$$

Mit  $UBRRn=0$  wird die maximale Übertragungsrate von  $f_{osc}/2$  erreicht, wobei  $f_{osc}$  die Taktfrequenz des Mikrocontrollers ist. Die eingestellte Übertragungsrate soll kleiner als die maximale Übertragungsrate aller am Bus angeschlossenen Slaves sein.

---

## Literatur

1. Microchip: Reference Manual ATmega48/168. <https://www.microchip.com/wwwproducts/en/ATmega88A>. Zugegriffen: 6. Jan. 2021.



# Die I<sup>2</sup>C/TWI-Schnittstelle

9

## Zusammenfassung

In diesem Kapitel wird die I<sup>2</sup>C/TWI-Schnittstelle beschrieben

Nachdem im achten Kapitel die SPI-Schnittstelle beschrieben wurde, fehlt als dritte fundamentale serielle Schnittstelle noch das im Microchip/ATMEL-Jargon „Two wire“ (TWI) genannte Interface. In vielen Anwendungen ist es als I<sup>2</sup>C-Bus bekannt und ist eigentlich mehr als eine serielle Schnittstelle. Im Gegensatz zu den vorgenannten ist die I<sup>2</sup>C-Schnittstelle nämlich erstens gut skalierbar und zweitens wegen ihrer schlanken Verkabelung (nur eine Daten- und eine Taktleitung, keine Select-Leitungen oder ähnliches) auch über ein wenig längere Strecken, beispielsweise in einem Schaltschrank, einsetzbar. Die Hardware gibt jedoch keine Übertragung über weite Strecken her, hierzu muss dann ein „echtes“ Bussystem eingesetzt werden.

Der I<sup>2</sup>C-Bus<sup>1</sup> (oder I2C) wurde von der Firma Philips (heute NXP)([2–4]) entwickelt und ermöglicht die Vernetzung von Mikrocontrollern, Sensoren und anderen Bausteinen

<sup>1</sup> I<sup>2</sup>C – Inter-Integrated Circuit.

Die Originalversion dieses Kapitels wurde revidiert. Ein Erratum ist verfügbar unter  
[https://doi.org/10.1007/978-3-658-31709-6\\_27](https://doi.org/10.1007/978-3-658-31709-6_27)

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann  
[https://doi.org/10.1007/978-3-658-31709-6\\_9](https://doi.org/10.1007/978-3-658-31709-6_9).

**Tab. 9.1** I<sup>2</sup>C-Modi

Modus	Übertragungsrate	Übertragungsrichtung
Standard	<100kBit/s	Bidirektional
Fast-mode	<400kBit/s	Bidirektional
Fast-mode Plus	<1Mbit/s	Bidirektional
High-speed-mode	<3,4Mbit/s	Bidirektional
Ultra-Fast-mode	<5Mbit/s	Unidirektional

wie: RAM- und EEPROM-Speicher, A/D-, D/A-Wandler, Port-Erweiterungen, LCD- und LED-Display Ansteuerungen, Echtzeituhren, Radio- und TV-Empfänger, I<sup>2</sup>C-Bus-Erweiterungen und vielen anderen. Basierend auf dieser Schnittstelle wurden weitere Protokolle 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 I<sup>2</sup>C-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. 9.1).

Der I<sup>2</sup>C-Bus funktioniert nach dem Master-Slave-Prinzip. Master ist derjenige Busknoten, der die I<sup>2</sup>C-Kommunikation initiiert und beendet, den Takt für die Synchronisation der Datenübertragung bereitstellt und entsprechend der programmierten Software Kommunikationsprobleme erkennt und löst. Jeder Slave besitzt eine Adresse, die ihn in einem Bus eindeutig identifiziert. Die Adresse kann sieben oder zehn 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 I<sup>2</sup>C-Bauteile haben diese Funktion implementiert!

I<sup>2</sup>C-Bauteile wie Speicher oder manche Sensoren bieten neben der fest programmierten Device Typ 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

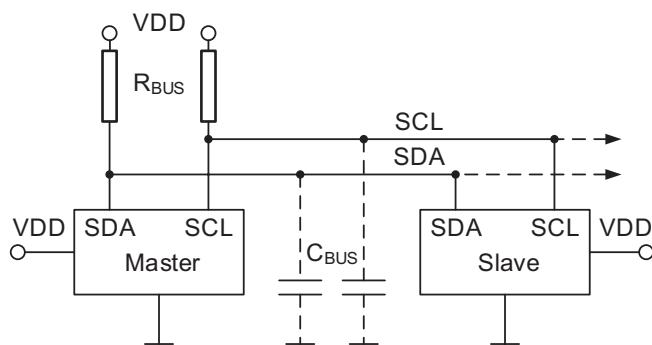
**Tab. 9.2** Adressierung eines M24C64-Bausteins

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 <sub>CC</sub>	1	0	1	0	0	0	1	1/0
GND	V <sub>CC</sub>	GND	1	0	1	0	0	1	0	1/0
GND	V <sub>CC</sub>	V <sub>CC</sub>	1	0	1	0	0	1	1	1/0
V <sub>CC</sub>	GND	GND	1	0	1	0	1	0	0	1/0
V <sub>CC</sub>	GND	V <sub>CC</sub>	1	0	1	0	1	0	1	1/0
V <sub>CC</sub>	V <sub>CC</sub>	GND	1	0	1	0	1	1	0	1/0
V <sub>CC</sub>	V <sub>CC</sub>	V <sub>CC</sub>	1	0	1	0	1	1	1	1/0

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 acht mögliche Adresskombinationen die in der Tab. 9.2 aufgelistet sind. Ein mit V<sub>CC</sub> verbundener Adresseingang setzt das entsprechende Adressbit auf „1“. Selten, wie im Fall des Temperatursensors TMP175, kann einem nicht angeschlossenen Pin ein dritter Zustand zugewiesen werden. Aus dem Datenblatt des Bausteins ist die Adresse entsprechend jeder Eingangskombination zu entnehmen.

## 9.1 I<sup>2</sup>C-Bus-Konfiguration

In der minimalen Konfiguration aus Abb. 9.1 wird der Master mit dem Slave über die bidirektionalen Busleitungen: SDA (Datenleitung) und SCL (Takteleitung) 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

**Abb. 9.1** I<sup>2</sup>C – minimale Konfiguration

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.

I<sup>2</sup>C 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. 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 selber adressiert sein soll.

Über I<sup>2</sup>C 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:

$$\begin{aligned} V_{IL} &= 0,3 \times V_{DD} \\ V_{OH} &= 0,7 \times V_{DD}, \end{aligned} \quad (9.1)$$

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 I<sup>2</sup>C-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 I<sup>2</sup>C-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:

$$V_{Cbus} = V_{DD} \left( 1 - e^{-\frac{t}{\tau_{Bus}}} \right) \quad (9.2)$$

mit der Zeitkonstante:

$$\tau_{Bus} = R_{Bus} \cdot C_{Bus} \quad (9.3)$$

Die Anstiegszeit der Spannung  $V_{Cbus}$  zwischen den, in der Gl. 9.1, angegebenen Grenzen muss kleiner sein als die spezifizierte Anstiegszeit  $t_r$  für jeden I<sup>2</sup>C-Modus. Mit diesen Überlegungen ergibt sich für den Buswiderstand der maximale Wert [1]:

$$R_{Bus(max)} = \frac{t_r}{0,8473 \cdot C_{Bus}} \quad (9.4)$$

Um den maximalen Strom durch die Ausgangsstufen auf den spezifizierten Wert zu begrenzen, wird der minimale Buswiderstand entsprechend [1] berechnet:

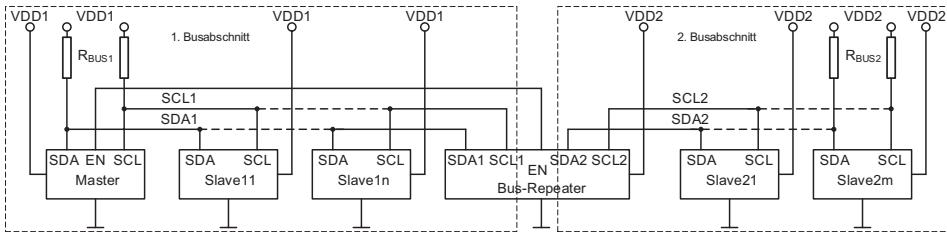
$$R_{Bus(min)} = \frac{V_{DD} - V_{OL(max)}}{I_{OL}} \quad (9.5)$$

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.

---

## 9.2 Bus-Erweiterung

Um komplexe I<sup>2</sup>C-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 I<sup>2</sup>C angesteuert wird. Die genauen Bedingungen, unter denen man diese Bus-Erweiterungen einsetzen kann, sind in [3] und [4] beschrieben. Alle Buserweiterungen erzeugen eine Laufzeitverzögerung der Signale, die man berücksichtigen muss.



**Abb. 9.2** I<sup>2</sup>C-Bus-Erweiterung mit einem Repeater

### 9.2.1 I<sup>2</sup>C-Repeater

Ein I<sup>2</sup>C-Repeater besitzt bidirektionale Treiber für die Busleitungen und spaltet den Bus in zwei Busabschnitte wie in Abb. 9.2. 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 langsamen. Bevor der Master die Kommunikation mit 400 kBit/s initiiert, muss er über den Repeater den rechten Busabschnitt isolieren. Der EN-Eingang des Multiplexers darf während einer Kommunikation nicht umgeschaltet werden.

### 9.2.2 I<sup>2</sup>C-Hub

Ein I<sup>2</sup>C-Hub ist eine Erweiterung eines I<sup>2</sup>C-Repeater. 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.

### 9.2.3 I<sup>2</sup>C-Multiplexer

Um Devices mit der gleichen Adresse anzusteuern, ohne dass Konflikte auftreten, verwendet man einen I<sup>2</sup>C-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 I<sup>2</sup>C-Multiplexer ermöglicht die Versorgung jedes Busabschnittes mit einer anderen Spannung ohne zusätzliche Pegelwandler.

### 9.2.4 I<sup>2</sup>C-Switch

Ein I<sup>2</sup>C-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.

---

## 9.3 TWI in der AVR-Familie

TWI<sup>2</sup> ist eine Schnittstelle, die standardmäßig in den Mikrocontroller der Familie ATmega implementiert und mit I<sup>2</sup>C kompatibel ist. 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. In diesem Abschnitt stellen wir eine Beispielimplementierung für AVR-Mikrocontroller vor, die als Teilnehmer eines Einzelmastert-TWI-Busses auftreten. Diese Implementierung lässt sich natürlich abändern und legt auch einige grundsätzliche Mechanismen der TWI-Kommunikation offen. Die Kommunikation in unserem Beispiel ist byte-orientiert und die Schnittstelle interruptbasiert. Es werden immer acht Bit (Byte) übertragen, mit dem höherwertigen Bit (MSB) zuerst. Das Ende eines TWI-Kommunikationsvorgangs 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 nach der Übertragung eines Bytes auf Low gehalten, bis der Interrupt bedient wurde, um die Busübernahme durch einen anderen Master zu verhindern. Der Master erzeugt acht Bittakte für die Bit-Synchronisation und einen neunten Takt für die Empfangsbestätigung. Wenn die SDA-Leitung vom Datenempfänger 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.

---

<sup>2</sup>TWI – two wire interface.

**Tab. 9.3** Einstellung des TWI-Prescalers

TWPS1	TWPS0	TWI-Prescaler
0	0	1
0	1	4
1	0	16
1	1	64

### 9.3.1 TWI-Register beim ATmega 88

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

$$f_{SCL} = \frac{f_{osc}}{16 + 2 \cdot TWBR \cdot TWI_{Prescaler}} \quad (9.6)$$

bestimmt, wobei  $f_{osc}$  die Taktfrequenz des Mikrocontrollers ist. Der Wert von TWBR (TWI bit rate register) wird gemäß Gl. 9.7 eingestellt und der Wert des  $TWI_{Prescaler}$  wird entsprechend Tab. 9.3 selektiert. TWPS1 und TWPS0 gehören zum Statusregister.

Wenn man Gl. 9.6 nach TWBR auflöst, erhält man:

$$TWBR = \left( \frac{f_{osc}}{f_{SCL}} - 16 \right) / (2 \cdot TWI_{Prescaler}) \quad (9.7)$$

Mit der Einstellung TWI-Prescaler gleich 1 lassen sich Werte des TWBR-Registers für Taktfrequenzen  $f_{SCL}$  ab ca. 40 kHz bei  $f_{osc}=20\text{ MHz}$  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-Register(TWI control 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, 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-Serviceroutine 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 Bits1..0, TWPS1..0 dienen zur Prescalerwahl (siehe Tab. 9.3). Um den Statuscode auszuwerten, muss das Statusregister maskiert werden, so dass die oben erwähnten Prescalerbits ausgeblendet sind:

```
#define TWI_STATUS_REGISTER (TWSR &0xF8)
```

### 9.3.2 Initialisieren der TWI-Schnittstelle

In einem Einzelmaster-TWI-Bus wird ein Mikrocontroller entweder als Master oder als Slave konfiguriert. Die Einstellparameter sollten idealerweise in der Headerdatei des

TWI-Moduls in einer Datenstruktur zusammengestellt werden, wie das folgende Beispiel veranschaulicht:

```
typedef struct{
    uint8_t ucDevice; //TWI_MASTER oder TWI_SLAVE
    uint8_t ucTwiClock; //Übertragungsrate für den Master
    uint8_t ucSlaveAddress; //Slave-Adresse
    uint8_t ucGenAddress; //enable/disable general call
    uint8_t ucAddressMask; //ev. Adressmaske für den Slave
} TWI_InitParam;
```

Die einzelnen Parameter werden in der Hauptdatei (.c Datei) definiert. Über den ersten Parameter wird die Rolle des Busteilnehmers bestimmt: Master oder Slave. Wenn der erste Parameter TWI\_MASTER ist, muss der zweite Parameter der codierte Wert der Taktfrequenz sein. Die weiteren Parameter sind für den Master nicht relevant.

Für einen Slave ist der Parameter Clock-Frequenz nicht relevant, stattdessen muss die 7-Bit-Adresse und eventuell die Adress-Maske als 7-Bit-Zahl eingegeben werden. Zusätzlich sollte als vierter Parameter CGA\_ENABLE (general call enable) übergeben werden, um zu bestimmen, ob der Slave über die allgemeine Adresse 0x00 ansprechbar ist.

Die Initialisierungsfunktion gibt die Schnittstelle frei. Falls der Prozessor als Slave konfiguriert ist, wird der TWI-Interrupt zusätzlich wie folgt freigegeben:

```
void TWI_Init(TWI_InitParam sinit_param)
{
    if(sinit_param.ucDevice == TWI_MASTER)
    {
        TWCR = (1 << TWEA) | (1 << TWEN);
        TWBR = sinit_param.ucTwiClock;
        //die Taktfrequenz wird in der Haupt-Datei ausgerechnet
    }
    else if(sinit_param.ucDevice == TWI_SLAVE)
    {
        TWAR = (sinit_param.ucSlaveAddress << 1) | sinit_param.
        ucGenAddress;
        //die Adresse des Slaves wird bestimmt und die Freigabe für die
        allgemeine Adresse 0x00
        TWAMR = sinit_param.ucAddressMask <<1; //Speicherung der
        Adressen-Maske
        TWCR |= (1 << TWEN) | (1 << TWEA) | (1 << TWIE);
    }
}
//Die Funktion wird folgendermaßen aufgerufen:
TWI_Init(sTWI_InitParam);
```

### 9.3.3 TWI-Kommunikation

Hardwaremäßig findet die TWI-Kommunikation an AVR-Controllern genau wie bei I<sup>2</sup>C oben beschrieben statt. In unserem Beispiel werden alle nötigen Funktionen in einem TWI-Softwaremodul zusammengefasst und sind so aufgebaut, dass der Mikrocontroller durch Initialisierung im Master- oder im Slave-Modus betrieben werden kann. Für den Einzelmaster-Modus (ein TWI-Netzwerk mit nur einem Master) stellen wir in den folgenden Abschnitten die grundlegenden Funktionen vor, die dazu dienen, eine TWI-Kommunikation zu initiieren, zu beenden, und ein Byte in beide Richtungen (Master-Slave und Slave-Master) zu übertragen. Weil der Master in diesem Modus die Kontrolle über den Zeitpunkt der Kommunikation hat, wird auf das Ende einer TWI-Sequenz blockierend gewartet. Diese einfachen Funktionen bieten eine gute Grundlage für die Programmierung von übergeordneten Funktionen für die Ansteuerung diverser I<sup>2</sup>C-Slaves. Zahlreiche Beispiele in folgenden Kapiteln bauen auf genau diesen Funktionen auf.

Dem Mikrocontroller im Slave-Modus ist der Zeitpunkt einer Kommunikation nicht bekannt. Damit der Mikrocontroller in diesem Modus nicht unnötig mit Polling belastet wird, wurde auf Funktionen mit blockierendem Warten verzichtet. Die gesamte Slave-Kommunikation ist von der TWI-ISR gesteuert.

Als TWI-Nachrichtenrahmen wird folgende Datenstruktur vorgeschlagen:

```
typedef struct
{
    uint8_t ucAddress; //für den Slave: private oder allgemeine
    Adresse
    uint8_t ucTWIData[4];
    uint8_t ucTWIDataLength;
}twi_frame;
```

Die Struktur-Komponente ucAddress ist nur für den Slave als Empfänger relevant. Sie codiert die Adressierungsart der Nachricht als privat (nur für diesen Slave bestimmt) oder allgemein (Broadcast-Nachricht). Die maximale Anzahl der Datenbytes eines Frames ucTWIDataLength kann für die konkrete Anwendung beliebig angepasst werden.

### 9.3.4 Der Mikrocontroller ATmega als TWI-Master

Der Aufruf der Funktion `TWI_Init()` mit folgendem Parameter initialisiert einen Mikrocontroller als TWI-Master:

```

TWI_InitParam sTWI_InitParam = { /*ucDevice*/
    /*ucTwiClock*/
    /*ucSlaveAddress*/
    /*ucGenAddress*/
    /*ucAddressMask*/
} TWI_MASTER,
TWI_MASTER_CLOCK,
0x80,
CGA_DISABLE,
0x00};

```

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()`

```

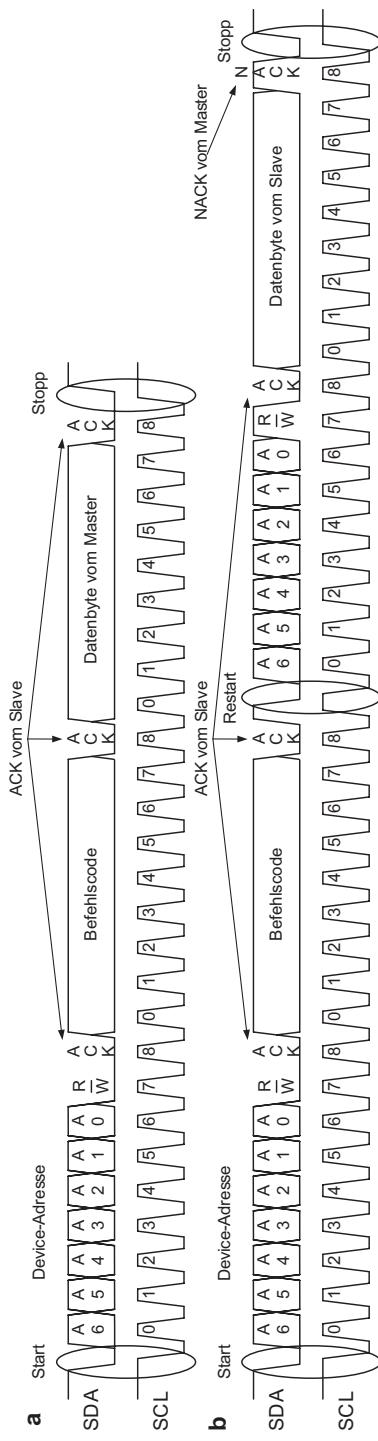
void TWI_Master_Start(void)
{
    TWCR = (1 << TWINT) | (1 << TWSTA) | (1 << TWEN);
    //Start: SDA-Leitung vor der SCL-Leitung auf Low setzen
    while(!(TWCR & (1 << TWINT))); //warten bis SDA gesetzt wird
}

```

startet der Mikrocontroller, nunmehr im Master-Modus, eine START-Sequenz sobald der Bus frei ist. Nach dem START gilt der Bus als besetzt. Nach einem erfolgreichen START enthält das Statusregister des Masters 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; über das Bit 0 (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. Es muss während des gesamten SCL-Pulses einen stabilen Zustand einnehmen, so wie in Abb. 9.3 skizziert. Diese Anforderung ist in der hardwareseitigen Sicherungsschicht des Protokolls implementiert und muss durch die Anwendersoftware nicht mehr implementiert werden.

### 9.3.4.1 Der TWI-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. 9.3a 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.



**Abb. 9.3** TWI-Kommunikation

```

void TWI_Master_Transmit(unsigned char uodata)
{
    TWDR = uodata;
    TWCR = (1 << TWINT) | (1 << TWEN);
    while (!(TWCR & (1 << TWINT)));
/*die Hardware setzt das Bit TWINT wenn das Byte uodata 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 der SCL- vor der SDA-Leitung auf High, was softwaremäßig mit dem Aufruf der Funktion `TWI_Master_Stop()` zu realisieren ist. Nach dem STOP gilt der Bus nach einer im Standard spezifizierten Verzögerungszeit als frei und kann von einem anderen Master in Anspruch genommen werden.

Eine einfache Funktion, mit der der Master unter den genannten Bedingungen ein Frame an den Mikrocontroller-Slave sendet, sieht folgendermaßen aus. Das zu sendende Frame wird in der Hauptdatei ausgefüllt und die Adressierung kann privat, per Multicast oder per Broadcast erfolgen. Der Abschluss jeder TWI-Sequenz wird geprüft. Falls Fehler auftreten, wird die Funktion unterbrochen und ein Fehlercode an die aufrufende Stelle zurückgeschickt.

```

uint8_t TWI_Send_Frame(uint8_t ucdevice_address, twi_frame *sframe)
{
    uint8_t ucDeviceAddress;
    ucDeviceAddress = (ucdevice_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)
    {
        TWI_Master_Stop();
        return TWI_ERROR;
    }
    for(uint8_t i = 0; i < sframe->ucTWIDataLength; i++)
    {
        TWI_Master_Transmit(sframe->ucTWIData[i]);
        if(TWI_STATUS_REGISTER == TWI_MT_DATA_NACK)
        {
            TWI_Master_Stop();
            return TWI_NACK;
        }
    }
}

```

```

        }
    }
    TWI_Master_Stop(); // Stop
    return TWI_OK;
}

```

### 9.3.4.2 Der TWI-Master als Empfänger

Weil die meisten I<sup>2</sup>C-Bausteine eine komplexe Struktur aufweisen und daher vielfältige Daten ausgeben beziehungsweise breit konfigurierbar sind, kann ein Master in der Regel 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 oder ein Funktionscode sein. Um die Kontrolle des Busses nicht zu verlieren, generiert der Master eine so genannte RESTART-Sequenz durch einen erneuten START und reserviert dadurch weiterhin den Bus, so dass ein eventuell weiterer Master nicht zugreifen kann. Es folgt eine neue Adressierung des Slaves, diesmal mit dem Read/Write-Bit gesetzt. Wenn der Slave den Empfang mit ACK bestätigt, enthält das Statusregister im Master 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. 9.3b 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 Funktionen für den Empfang eines Bytes vorgestellt, je nachdem ob der Master mit ACK oder NACK den Byteempfang bestätigt:

```

//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-, bzw. Lese-Funktionen sind in den folgenden Kapiteln über Sensorbeispiele vorgestellt.

Die Kommunikation des Masters als Empfänger mit einem Mikrocontroller-Slave ist einfacher und wird im Folgenden anhand der Funktion `TWI_Read_Frame()` veranschaulicht. Der Master adressiert nur einmal privat den Slave mit dem gesetzten R/W-Bit und stellt weiterhin den Takt für den Datenempfang her. Die Datenbytes werden in einer Datenstruktur vom Typ `twi_frame` gespeichert. Der Master bestätigt jedes Byte, bis auf das letzte mit Acknowledge. Unterschiedliche Funktionen können auch in diesem Fall über unterschiedliche Adressen erreicht werden.

```
uint8_t TWI_Read_Frame(uint8_t ucdevice_address, twi_frame *sframe)
{
    uint8_t ucDeviceAddress, i;
    ucDeviceAddress = (ucdevice_address << 1);
    ucDeviceAddress |= TWI_READ; //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_MR_SLA_ACK)
    {
        TWI_Master_Stop();
        return TWI_ERROR;
    }
    for(i = 0; i < (sframe->ucTWIDataLength - 1); i++)
    {
        sframe->ucTWIData[i] = TWI_Master_Read_Ack();
        if(TWI_STATUS_REGISTER == TWI_MR_DATA_NACK)
            {//sollte der Slave mit NACK antworten, beendet der
             Master die Kommunikation
            TWI_Master_Stop();
            return TWI_NACK;
        }
    }
    /*auf das letzte Byte antwortet der Master mit NACK, und
     beendet die Kommunikation*/
    sframe->ucTWIData[i] = TWI_Master_Read_NAck();
    TWI_Master_Stop(); // Stop
    return TWI_OK;
}
```

### 9.3.5 Der Mikrocontroller ATmega als TWI-Slave

Um den Mikrocontroller als Slave zu konfigurieren, wird die Funktion `TWI_Init()` beispielhaft mit folgendem Parameter `sTWI_Init_Param` aufgerufen:

```
TWI_InitParam sTWI_InitParam = { /*ucDevice*/           TWI_SLAVE,
                                /*ucTwiClock*/        TWI_MASTER_CLOCK,
                                /*ucSlaveAddress*/    0x10,
                                /*ucGenAddress*/      CGA_ENABLE,
                                /*ucAddressMask*/     0x0B};
```

Die Struktur wird in der Hauptdatei definiert. Die Komponente `ucTwiClock` ist für den Slave nicht relevant und kann einen beliebigen Wert annehmen.

In der Initialisierungsfunktion werden die TWI-Schnittstelle und der TWI-Interrupt aktiviert. Damit übernimmt die Hardware die Kontrolle der Anschlüsse SCL und SDA und der interne Datenfluss wird durch Interrupt gesteuert.

Jedem Mikrocontroller, der im Slave-Modus arbeiten soll, wird softwaremäßig eine 7-Bit-Adresse zugewiesen, die einmalig im TWI-Bus auftreten darf. Diese Adresse wird um ein Bit nach links versetzt und in das Adressregister `TWAR` gespeichert. Mit dem Setzen des niederwertigen Bits des Registers `TWAR` kann der Slave Broadcast-Nachrichten empfangen. Diese Nachrichten werden von dem Master an die allgemeine Adresse 0x00 gesendet. Die 7-Bit-Adressen der Form 1111 xxx sind für spätere Anwendungen reserviert.

Der Slave vergleicht die über TWI empfangene Adresse mit der eigenen und bei Übereinstimmung bestätigt er die Adressierungssequenz mit Acknowledge. Bei diesem Vergleich werden die mit einer „1“ im Masken-Register `TWAMR` markierten Bits vernachlässigt. Mit jedem gesetzten Bit im Masken-Register erhöht sich der Adressen-Bereich eines Slaves um den Faktor zwei wie in Tab. 9.4. gezeigt.

Wenn der Slave als Empfänger adressiert ist, wird das mit „x“ bezeichnete Bit in Tab. 9.4 zu 0 gesetzt; wenn der Slave als Transmitter adressiert wird, zu 1. Die Broadcast- und Multicast-Adressierung sind nur für die Slaves im Empfänger-Modus sinnvoll.

**Tab. 9.4** Bestimmung der Slave-Adressen

TWAR	TWAMR	Bestätigte Adresse
0001 0000	0000 0000	0001 000x
0001 0000	0000 0010	0001 000x 0001 001x
0001 0001	0000 0010	0000 0000 0001 000x 0001 001x

**Tab. 9.5** Einstellungen zweier Slaves für eine Multicast-Kommunikation

	Slave 1	Slave 2
TWAR	0010 0000	0001 0000
TWAMR	0001 0110	0010 0110
Bestätigte Adressen	0010 0000 0010 0010 0010 0100 0010 0110 0011 0000 0011 0010 0011 0100 0011 0110	0001 0000 0001 0010 0001 0100 0001 0110 0011 0000 0011 0010 0011 0100 0011 0110

Die Adressen-Maskierung öffnet neue Möglichkeiten für die Kommunikation mit Mikrocontroller-Slaves. Dadurch kann jede Schreib- oder Lese-Funktion eines Slaves direkt über eine der Adressen aufgerufen werden. Das führt zu einer Verkürzung der Nachrichten und zur Vereinfachung der Master-Funktionen.

Der Mikrocontroller als Slave initialisiert befindet sich im nicht adressierten Modus. Das Read/Write-Bit in der Adressierungssequenz schaltet ihn in den Empfänger- oder Transmitter-Modus.

### 9.3.5.1 Der TWI-Slave als Empfänger

Ein TWI-Slave schaltet in den Empfänger-Modus wenn er mit dem R/W-Bit gleich „0“ adressiert wird. In diesem Modus ist eine Point-to-Point, bzw. Broadcast-Kommunikation möglich. Durch eine geschickte Nutzung der Adressen-Maskierung ist auch eine Multicast-Kommunikation möglich. In diesem Modus kann der Master eine Slave-Gruppe gleichzeitig adressieren wie in Tab. 9.5.

Mit dem Setzen von drei Bits im Register TWAMR erhöht sich die Zahl der privaten Adressen der zwei Slaves auf jeweils acht. Die letzten vier aufgelisteten Adressen sind für die zwei Slaves identisch, was dem Master die gemeinsame Adressierung (Multicast-Modus) nur dieser zwei Slaves im Empfänger-Modus ermöglicht.

Folgende TWI-Ereignisse führen zu einem Interrupt für den Slave im Empfänger-Modus:

- Erkennung einer Adresse (private oder allgemeine); in das Status-Register wird der Code 0x60 beim Empfang einer privaten Adresse, bzw. 0x70 beim Empfang der allgemeinen Adresse geschrieben. Entsprechend dieser Adresse wird in unserer Beispiel-Implementierung ein Code in `twi_frame->ucAddress` gespeichert. Dieser dient der späteren Zuordnung der Nachricht.
- Empfang eines Datenbytes führt zum Speichern des Codes 0x80 in das Register TWSR; die TWI-Schnittstelle bietet keine Möglichkeit, die empfangenen Datenbytes

zu puffern. Deshalb werden diese aus dem Datenregister TWDR in einem Empfangspuffer vom Typ `twi_frame` gespeichert und eine Laufvariable wird inkrementiert.

- Erkennung einer Stopp- oder einer neuen Start-Sequenz. Nach dem Eintreten eines dieser Ereignisse wird der Code 0xA0 in das Status-Register gespeichert. Das bedeutet das Ende eines TWI-Rahmens und die Zahl der empfangenen Bytes (der Inhalt der Laufvariable) wird in `twi_frame->ucDataLength` gespeichert.

Der Slave bestätigt im Empfänger-Modus alle empfangenen Bytes mit Acknowledge.

Wenn die Verarbeitung der Nachrichten unterschiedlich lange Zeit braucht, ist es sinnvoll die Frames in ein Array zu speichern:

```
#define MAX_REC_BUFFER      5
twi_frame stWI_RecBuffer[MAX_REC_BUFFER];
```

Dieses Array wird als FIFO-Ringpuffer behandelt und in der Interrupt-Serviceroutine verwaltet. Die maximale Zahl der Komponenten wird entsprechend der konkreten Anwendung global definiert. Eine Speicher-Laufvariable gibt den Array-Index an, an dem der nächste Rahmen gespeichert wird. Eine Lese-Laufvariable zeigt auf den ältesten ungelesenen Rahmen. Weitere Ausführungen und Alternativen zu FIFO-Ringpuffern finden Sie in Abschn [5.1.1](#).

```
case 0xA0: //ein Stopp oder ein neuer Start wurde festgestellt; ein
neuer Rahmen wurde empfangen
    stWI_RecBuffer[ucTWI_Buff2Save].ucTWIaDataLength = ucIndex;
    ucTWI_Buff2Save++; //Laufvariable für das zu speichernde Frame
    if(ucTWI_Buff2Save == MAX_REC_BUFFER) ucTWI_Buff2Save = 0;
    if(ucTWI_Buff2Save == ucTWI_Buff2Read)
    { //der Fall tritt ein wenn der erste Schreib-Puffer
    überschrieben, aber nicht gelesen wurde
        ucTWI_Buff2Read++; //Laufvariable für das zu lesende Frame
        if(ucTWI_Buff2Read == MAX_REC_BUFFER) ucTWI_Buff2Read = 0;
    }
    TWCR |= (1 << TWINT) | (1 << TWEA); //umschalten in den nicht
    adressierten Slave-Modus
break;
```

Wenn beim Speichern einer neuen Nachricht die Speicher- mit der Lese-Laufvariable des FIFO-Puffers gleich sind, findet ein Überlauf vom FIFO statt und die älteste, nicht gelesene Nachricht wird überschrieben.

Nach der Initialisierung der TWI-Schnittstelle sind beide Laufvariablen `ucTWI_Buff2Save` und `ucTWI_Buff2Read` null. Später zeigt die Ungleichheit der Laufvariablen `ucTWI_Buff2Save` und `ucTWI_Buff2Read` an, dass ungelesene Nachrichten vorhanden sind. Diese Prüfung kann im Polling-Betrieb mit der Funktion

TWI\_Check\_Message() realisiert werden. Diese Funktion gibt zurück MESSAGE\_RECEIVED falls es neue Nachrichten gibt und in diesem Fall kopiert die älteste, nicht gelesene Nachricht in die als Parameter übergebene Datenstruktur und aktualisiert die Lese-Laufvariable.

```
uint8_t TWI_Check_Message(twi_frame *sframe)
{
    if(ucTWI_Buff2Save == ucTWI_Buff2Read)
    {
        return NO_MESSAGE; //keine neuen Nachrichten
    }
    else
    {
        sframe->ucAddress = sTWI_RecBuffer[ucTWI_Buff2Read] .
        ucAddress;
        sframe->ucTWIDataLength = sTWI_RecBuffer
        [ucTWI_Buff2Read].ucTWIDataLength;
        for(uint8_t i = 0; i < sTWI_RecBuffer[ucTWI_
        Buff2Read].ucTWIDataLength; i++)
        {
            sframe->ucTWIData[i] = sTWI_RecBuffer
            [uctWI_Buff2Read].uctWIData[i];
        }
        ucTWI_Buff2Read++; //die Lese-Laufvariable wird
        aktualisiert
        if(ucTWI_Buff2Read == MAX_REC_BUFFER) ucTWI_Buff2Read = 0;
    }
    return MESSAGE_RECEIVED; //es gab eine ungelesenen Nachricht
}
```

### 9.3.5.2 Der TWI-Slave als Transmitter

Der Mikrocontroller wird als Slave in den Transmitter-Modus versetzt, wenn das Read/Write-Bit in der Adressierungssequenz auf 1 gesetzt ist. In diesem Modus sind die Broadcast- und Multicast-Kommunikationen sinnlos. Durch die Adressen-Maskierung können auch in diesem Modus Slave-Aktionen direkt ausgelöst werden.

Für den Slave im Transmitter-Modus wird ein Sendepuffer vom Typ `twi_frame` deklariert. Die Struktur-Komponente `ucAddress` ist in diesem Modus nicht relevant. Die übergeordnete Anwender-Software sorgt für die Aktualisierung dieses Puffers aus dem die Datenbytes, gesteuert von der ISR gesendet werden. Das kann wie im folgenden Beispiel realisiert werden:

```
void TWI_Fill_TransmitBuffer(twi_frame *sframe)
{
    sTWI_TransmitBuffer.ucTWIDataLength = sframe->ucTWIDataLength;
    for(uint8_t i = 0; i < sTWI_TransmitBuffer.ucTWIDataLength; i++)
    {
        sTWI_TransmitBuffer.ucTWIData[i] = sframe->ucTWIData[i];
    }
}
```

Der Zeitpunkt der Datenübertragung wird vom Master bestimmt, der die Übertragung initiiert. Eine Aktualisierung des Sendepuffers während der Kommunikation führt zur Verfälschung der Nachricht. Um das zu verhindern, wird vorher geprüft ob der Sendepuffer leer ist.

Die Ereignisse, die zu einem Interrupt für den Slave im Transmitter-Modus führen, sind:

- Bei der Erkennung der eigenen Adresse wird der Code 0xA8 in das Status-Register gespeichert. In der ISR wird das erste Datenbyte aus dem Sendepuffer gesendet.
- Das Senden eines Datenbytes, das von dem Master mit einem Acknowledge bestätigt wird, ändert den Code in das Status-Register zu 0xB8. Das nächste Datenbyte wird gesendet und die Laufvariable wird inkrementiert.
- Das Senden eines Datenbytes, das von dem Master mit einem NACK quittiert wurde, führt zum Code 0xC0 im Status-Register. Der Slave wertet den Code als Kommunikationsende, schaltet in den nicht adressierten Slave-Modus und speichert in die Variable `ucTWI_TransmitBuffer` den Wert `TRANSMIT_BUFFER_EMPTY`.
- Beim Senden des letzten Datenbytes antwortet der Master mit ACK und in das Status-Register wird der Code 0xC8 gespeichert. Der Slave schaltet in den nicht adressierten Slave-Modus und meldet den Sendepuffer als leer.

---

## Literatur

1. Meroth, A., & Tolg B. (2008). *Infotainmentsysteme im Kraftfahrzeug. Grundlagen, Komponenten, Systeme und Anwendungen*. Wiesbaden.
2. NXP Semiconductors. UM10204 – I<sup>2</sup>C-bus specification and user manual- Rev.6–4 April 2014. [www.nxp.com](http://www.nxp.com).
3. NXP Semiconductors. Application Note AN255-02 – I<sup>2</sup>C/SMBus Repeaters, Hubs and Expanders, 2015. [www.nxp.com](http://www.nxp.com).
4. NXP Semiconductors. Application Note AN262\_2 – PCA954x Family of I<sup>2</sup>C/SMBus Multiplexers and Switches, 2015. [www.nxp.com](http://www.nxp.com).
5. Microchip: Reference Manual ATmega48/168. <https://www.microchip.com/wwwproducts/en/ATmega88A>. Zugegriffen: 6. Jan. 2021.



### Zusammenfassung

In diesem Kapitel wird der CAN Bus erklärt und eine Bibliotheksimplementierung vorgestellt.

Nachdem in den letzten Kapiteln die fundamentalen seriellen Schnittstellen zur Vernetzung von Sensoren und anderen Peripheriekomponenten vorgestellt wurden, widmen sich die folgenden Kapitel einigen ausgewählten drahtgebundenen Netzwerken zwischen Steuergeräten. Den Anfang macht der CAN-Bus, der in der Automobilindustrie entstand aber inzwischen aus der Automatisierungstechnik nicht mehr wegzudenken ist.

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 [1] ist er nun als ISO 11898 (Teile 1–6) international genormt [2]. 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 Originalversion dieses Kapitels wurde revidiert. Ein Erratum ist verfügbar unter  
[https://doi.org/10.1007/978-3-658-31709-6\\_27](https://doi.org/10.1007/978-3-658-31709-6_27)

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann  
[https://doi.org/10.1007/978-3-658-31709-6\\_10](https://doi.org/10.1007/978-3-658-31709-6_10).

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 [3] und [4].

## 10.1 CAN-Grundlagen kompakt

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 (siehe Abschn. 6.1), 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 (CAN2.0A), ist also botschaftenorientiert. In der Spezifikation CAN2.0 B ist der Identifier auf 29 Bit erweiterbar. Man nennt den Botschaftsrahmen dann „extended“ statt „standard“. 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 rezessives (=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.

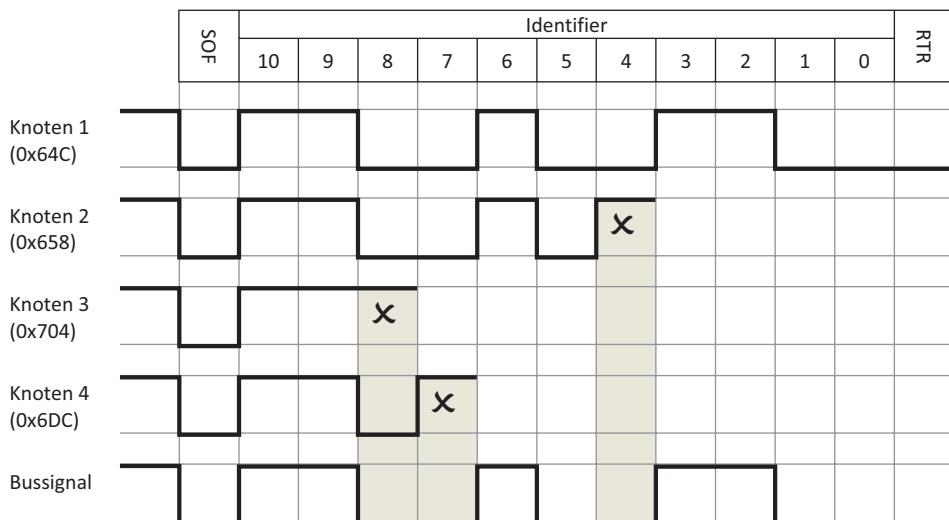
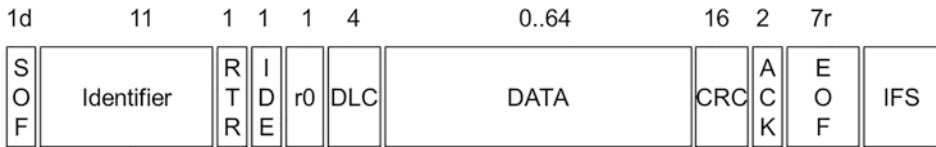


Abb. 10.1 CSMA/CA-Konkurrenzperiode

**Abb. 10.2** CAN-Frame

In Abb. 10.1 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. 10.2) 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 ist Platz für maximal acht 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 in Bytes an.
- **DATA:** Nutzdaten: können null bis acht Byte umfassen.
- **CRC:** CRC-Prüfsumme (siehe Seite 170).
- **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 sieben 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 größer 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.

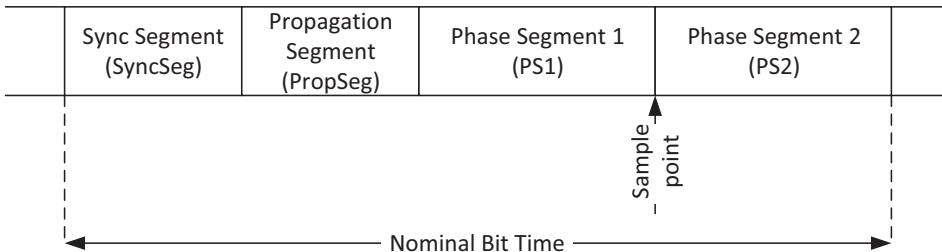
Das CAN-Protokoll entspricht der Sicherungsschicht (Schicht 2) im OSI-Schichtenmodell. Darüber definiert die ISO 15765-2 ein Transportprotokoll auf Schicht 4 im OSI-Schichtenmodell, das beispielsweise in der Diagnose eingesetzt wird. Eine ausführliche Beschreibung findet sich beispielsweise bei [3] und [4].

---

## 10.2 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 PS2 auch für die Verarbeitung zuständig (Information Processing Time).



**Abb. 10.3** Bit-Timing im CAN

Abb. 10.3 zeigt den Aufbau eines Bit graphisch an. Die gesamt Bitzeit ist also

$$t_{Bit} = t_{SyncSeg} + t_{PropSeg} + t_{PS1} + t_{PS2} \quad (10.1)$$

Hieraus ergibt sich dann die nominelle Bitrate.

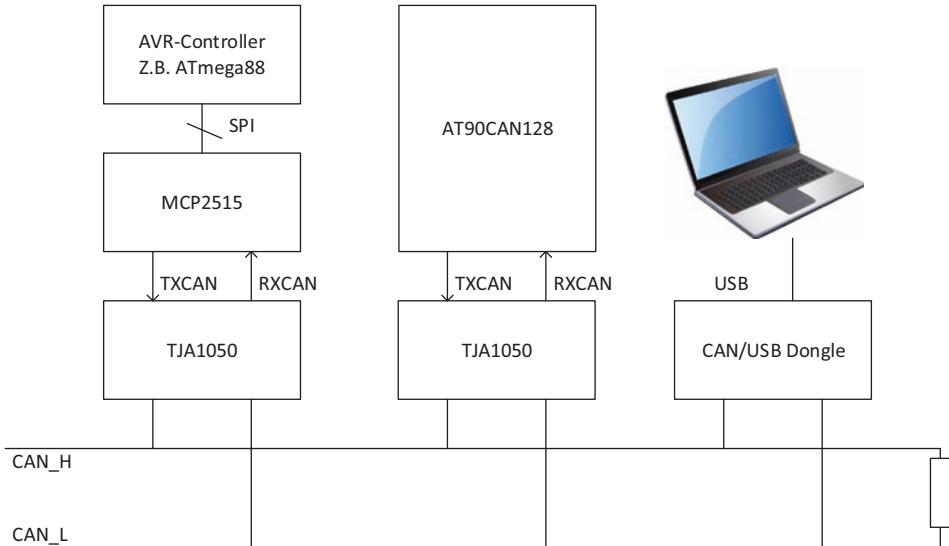
$$NBR = \frac{1}{t_{Bit}} \quad (10.2)$$

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.

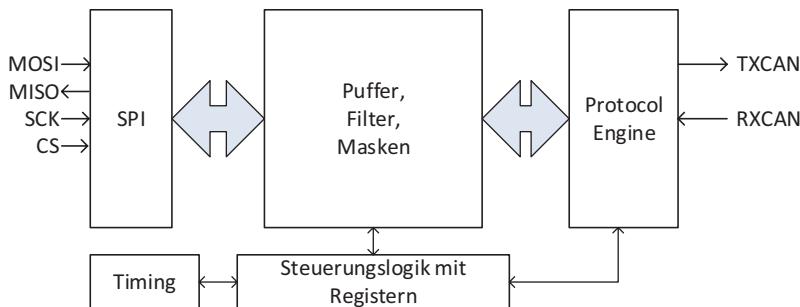
## 10.3 Nutzung von CAN mit Prozessoren der AVR-Familie

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 [5]. Wir stellen hier eine eigene Bibliothek vor, die auf der Webseite des Buches als Quellcode frei heruntergeladen werden kann.

In Abb. 10.4 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. 10.4** CAN Netzwerk mit drei Teilnehmern



**Abb. 10.5** Blockschaltbild\_des\_MCP\_2515. (Nach [6])

### 10.3.1 CAN-Controller MCP2515

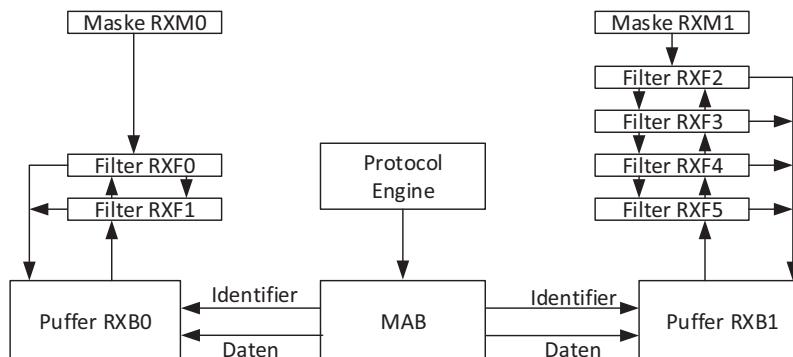
Der CAN-Controller MCP2515 von Microchip [6] ist einer der erfolgreichsten CAN-Controller auf dem Markt. Er ist über SPI anschließbar und liefert am Ausgang Signale für CAN Receive (CAN Rx) und Transmit (CAN Tx), die über einen Transceiver elektrisch an das Bussystem angeschlossen werden müssen (hier: TJA1050). Eine Übersicht über die Architektur ist in Abb. 10.5 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 [6]

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 CSMA/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 erledigt 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 auf Low gezogen werden, sodass der MCP2515 beim Prozessor einen Interrupt (Pin Change 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. 10.6 gibt einen Überblick über die Struktur des Empfangssystems.



**Abb. 10.6** Strukturen der Empfangspuffer im MCP2515. (Nach [6])

### 10.3.1.1 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.

$$TQ = \frac{2 \cdot BRP}{f_{osc}} \quad (10.3)$$

Unsere Bibliothek erledigt dies mit einem einfachen Aufruf

```
//CAN Baudrate in MCP2515_HHN.h
#define BAUDRATE_20_KBPS      0
#define BAUDRATE_50_KBPS      1
#define BAUDRATE_100_KBPS     2
#define BAUDRATE_125_KBPS     3
#define BAUDRATE_250_KBPS     4
#define BAUDRATE_500_KBPS     5
#define BAUDRATE_1_MBPS       6

//Aufruf
MCP2515_Init(MCP2515_1, BAUDRATE_250_KBPS);
```

Zuvor wird einmalig wie in Abschn. 8.4 beschrieben die Verwaltungsstruktur für die SPI Schnittstelle zusammengestellt. Diese Maßnahme ermöglicht, dass neben dem CAN Bus auch noch andere Sensoren oder ein oder mehrere weitere CAN-Busse mit unterschiedlichen SS-Ausgängen angesteuert werden können.

```
MCP2515_pins MCP2515_1 ={{
/*CS_DDR*/      &DDRB,
/*CS_PORT*/     &PORTB,
/*CS_pin*/       PB2,
/*CS_state*/    ON} };
```

Wobei diese Struktur MCP2515\_pins die aus Abschn. 8.4 bekannte Struktur vom Typ tspiHandle enthält. Daher müssen auch die zwei geschweiften Klammern geöffnet und geschlossen werden.

### 10.3.1.2 Botschaften senden

Die zentrale Datenstruktur für den Einsatz von CAN ist die CAN-Botschaft. In der CAN-Bibliothek wird sie für Standard- oder Extended-CAN-Nachrichten wie folgt definiert:

```
typedef struct
{
    uint8_t EIDE_Bit; //STANDARD_ID oder EXTENDED_ID
    uint8_t RTR_Bit; //DATA_FRAME oder REMOTE_FRAME
    uint8_t RB0_Bit; //Reserved bit: 0x00 oder 0x01
    uint8_t RB1_Bit; //Reserved bit: 0x00 oder 0x02
    uint32_t ulID; //11 oder 29 Bit ID
    uint8_t ucLength; //Anzahl der Datenbytes
    uint8_t ucData[8]; //Datenvektor für die maximale
                      Nachrichtenlänge
}can_frame;
```

Das ID-Feld mit 32 Bit sowohl für Extended-Identifier-Botschaften (29 Bit) als auch für Standard-Identifier-Botschaften geeignet. Manchmal wird auch ein Zeitstempel (16 Bit genügen meist) mitgespeichert, der allerdings von der Hardware des MCP2515 nicht unterstützt wird, sondern dann als Botschaftsbestandteil mit versendet werden muss.

Das Senden einer Botschaft durch die Funktion `MCP2515_Send_Message()` läuft wie folgt ab:

- Zunächst prüft die Funktion, 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 (TXREQ0..2) des Buffer-Status-Registers 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 Kommandos Load TX Buffer (0x40) zusammen mit der Adresse des TX-Puffers mit anschließender Übertragung des Identifiers, des Längenbytes, eventuell zusammen mit einem RTR (siehe [6]) 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).

Der Aufruf in unserer Bibliothek ist simpel. Das Beispiel zeigt das Versenden einer 16 Bit Nachricht mit einem Temperaturwert in little-endian-Format. Zunächst wird die Botschaft zusammengestellt und danach mit einem Aufruf gesendet.

```

can_frame sSendFrame;
sSendFrame.EIDE_Bit = STANDARD_ID;
sSendFrame.RTR_Bit = DATA_FRAME;
sSendFrame.ulID = 0x19; // bitte entsprechend eintragen!!
sSendFrame.ucLength=2;
/*****************/
sSendFrame.ucData[0]=(unsigned char) MeineTemperatur;
sSendFrame.ucData[1]=(unsigned char) (MeineTemperatur>>8);
//Klammer ist wichtig, da typecast höhere Prio hat als shift
/*****************/
MCP2515_Send_Message(MCP2515_1, &sSendFrame);

```

Eine einfache Sensorschaltung, die alle 100 ms einen Sensorwert verschickt, muss lediglich in einem 100-ms-Task einen Sendebefehl mit den entsprechenden Daten absetzen.

### 10.3.1.3 Botschaften empfangen

Das Empfangen von Botschaften kann über mehrere Wege erfolgen:

- Sind die Interrupt-Leitungen RX0BF und RX1BF verdrahtet, löst eine neue Botschaft in einem der beiden Empfangspuffer einen Interrupt aus, der den Inhalt der Register abholen kann. In unserer Bibliothek verzichten wir auf den Interrupt-Betrieb, mit dem wir keine guten Erfahrungen gemacht haben. Bei hohen Buslasten können nämlich bei der in [5] verwendeten Bibliothek Stack-Überläufe auftreten, wenn mehr Nachrichten kommen, als Interrupts abgearbeitet werden können. Dies führt zu einem Totalabsturz des Systems, was die Autoren erfahren durften.
- Verzichtet man also auf den Interrupt, kann man über das Kommando Read Status (0xB0) den Empfangsstatus der beiden Puffer abfragen (siehe [6]) und erhält den Empfangsstatus und die Filternummern, über die der Empfang freigegeben wurde (Polling-Betrieb). In beiden Fällen muss ohnehin anschließend die Nachricht abgeholt werden, indem das Kommando Read Rx-Buffer gesendet wird und anschließend die Pufferinhalte ausgelesen werden.

Generell ist bei hohen Buslasten zu empfehlen, die Filterfunktion zu nutzen, in jedem Fall ist die Abfrage ziemlich einfach (in diesem Fall werden zwei Frames mit den IDs 0x10 und 0x15 gelesen):

```

can_frame sRecFrame;
if(MCP2515_Check_Message(MCP2515_1, &sRecFrame))
{ // Hier kann man jetzt das Frame auswerten:
    switch(sRecFrame.ulID)
    {
        case 0x10: Call_function_1(&sRecFrame); break;

```



Die Empfangsfilter, die nicht benutzt werden, werden auf 0x7FF für die Standard-Identifier, bzw. 0xFFFF für die Extended-Identifier gesetzt. Diese Identifier dürfen im Bus nicht mehr verwendet werden.

Wenn man Nachrichten auch mit dem Identifier 0x121 empfangen möchte, kann dieser in der Filterliste eingetragen werden. Da 0x123 und 0x121 sich nur durch das Bit 1 unterscheiden, muss die Filterliste nicht geändert werden, stattdessen wird die Empfangsmaske des Empfangspuffers 0 auf 0x7FD gesetzt. Das Setzen von n-Bits einer Empfangsmaske auf 0 erhöht die Zahl der gefilterten Identifier auf  $m \cdot 2^n$ , wobei m=2 für den Empfangspuffer 0 und m=4 für den Empfangspuffer 1.

### 10.3.2 AT90CANxx

Der AT90CAN128 besitzt einen eigenen CAN-Controller, der gemäß der CAN-Spezifikation folgende Aufgaben erfüllt:

- er realisiert die Logical Link Control (LLC)-Subschicht der Sicherungsschicht,
- die Medium Access Control (MAC)-Unterschicht
- vom physical layer wird die Physical Signalling (PLS) Subschicht implementiert

Nicht unterstützt werden das Physical Medium Attach (PMA) und das Medium Dependent Interface (MDI), das wie oben erwähnt durch einen Transceiver realisiert werden muss.

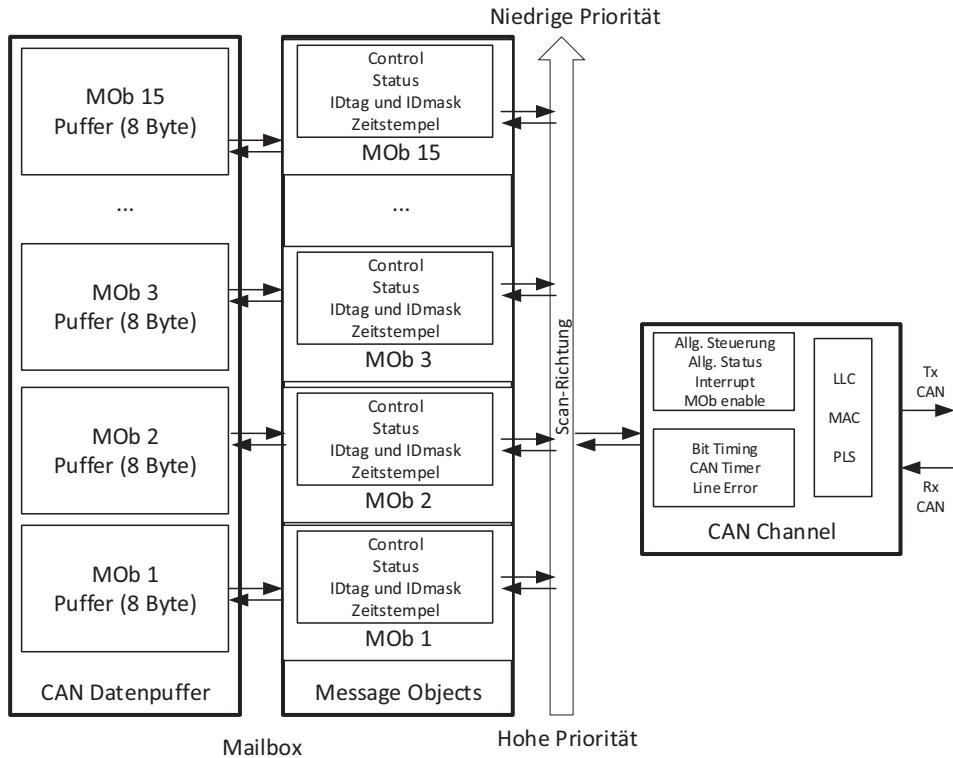
Der CAN-Controller ist in der Lage, alle Arten von Frames (Data, Remote, Error und Overload) zu verarbeiten und erreicht eine Bitrate von 1 Mbit/s. Soweit entspricht er dem oben beschriebenen Konzept, funktioniert aber völlig anders:

Der Controller enthält nämlich einen „Briefkasten“ (Mailbox), der bis zu 15 Bot-schaften enthalten kann. Er teilt sich in einen 8x15-Byte (120 Byte) Puffer, der die Nutz-daten enthält und einen Registersatz, in dem für 15 so genannte Message Objects (MOB) alle für das Senden/Empfangen notwendigen Daten abgelegt werden. In Abb. 10.7 ist die in [7] beschriebene Architektur abgebildet.

Die Steuerung des CAN-Controllers scannt die Mailbox nach zu sendenden Bot-schaften bzw. nach freien Message Objekten, in denen empfangene Nachrichten abgelegt werden.

Um Daten senden oder empfangen zu können, muss man die MOB initialisieren. Es gibt dabei fünf Modi:

- Im Disabled-Modus ist das MOB nicht im Gebrauch (frei)
- Im Tx-Data/Remote-Frame-Modus muss die Nachricht in einem freien MOB mit allen Feldern (also DLC, RTR, IDE, reserved bits, Identifier und die Daten), Abschn. 10.1 abgelegt werden und eine Anfrage zum Senden. Anschließend werden die beiden



**Abb. 10.7** Architektur des internen CAN-Controllers auf dem AT90CANx. (Nach [7])

CONMOB-Bits mit dem Sendebefehl 01b gesetzt. Daraufhin startet der Controller die Aussendung der Botschaft und quittiert mit einem TXOK-Flag bzw. Interrupt.

- Im Rx Data/Remote-Frame-Modus müssen ebenfalls die komplette Nachricht (bis auf die Daten, die ja empfangen werden sollen) sowie eine Akzeptanzfiltermaske in das MOB geschrieben werden. Wie beim MCP2515 bedeutet auch hier eine 1 am entsprechenden Bit der Filtermaske, dass dieses Bit in der ID mit dem entsprechenden ID-Bit in der zu empfangenden Botschaft übereinstimmen muss. Außerdem müssen die CONMOB-Bits auf 10b gesetzt werden. Wurde ein Frame auf dem Bus empfangen, scannt der Controller alle verfügbaren MOBs und schreibt deren Daten in das MOB mit der höchsten Priorität, das mit ID und Maske zu der Botschaft passt. Der Erfolg wird mit einem RXOK-Flag und einem Interrupt quittiert. Die Behandlung von RTR Anfragen muss von Hand erfolgen.
- Im Automatic-Reply-Modus kann man die Daten von RTR Empfangsbotschaften bereits vorbelegen. Sobald eine RTR-Botschaft auf dem Bus empfangen wird, vervollständigt der Controller die Botschaft automatisch mit den Daten. Voraussetzung ist, dass das Reply valid (RPLV) Bit gesetzt war. Nach der erfolgreichen Antwort wird

dieses Bit automatisch gelöscht. Man kann also hier sehr einfach asynchron Daten bereitstellen, die von einem entfernten Client per RTR-Botschaft angefragt werden.

- Will man mehrere Botschaften auf dem Bus empfangen, so bietet sich der Frame-Buffer-Receive-Modus an. Hier werden alle Botschaften gesammelt und erst dann ein RXOK bzw. Interrupt ausgelöst, wenn alle empfangen wurden.

Um eine eigene CAN-Bibliothek für den AT90CANxx zu schreiben kommt man um ein sorgfältiges Studium des Manuals [7] nicht herum, dieser Abschnitt soll nur zum ersten Verständnis beitragen. Allerdings sind auch hier die Bibliotheken aus [5] oder unsere eigene Bibliothek (auf der Webseite des Buchs beim Verlag zum Herunterladen) hilfreich. Nach guten Erfahrungen mit [5] haben wir uns für den Schritt der Eigenentwicklung entschlossen, weil unsere Bibliothek etwas mehr an den spezifischen Anforderungen der Prozessoren orientiert ist, für mehrere SPI-Teilnehmer besser angepasst ist und auch den Prozessor nach unserer Meinung etwas effizienter nutzt, während die aus [5] verschiedene Controller vereinheitlicht und damit nicht notwendigerweise vollständig ausreicht. Dafür ist sie insbesondere von Anfängern schneller zu nutzen und gut dokumentiert.

### 10.3.2.1 Implementierung auf dem AT90CANx

Der CAN-Controller des Mikrocontrollers AT90CANxx wird nach der POR-Phase<sup>1</sup> in den Standby-Modus geschaltet und die Messageobjekte (MOB) werden zufällig initialisiert. Bevor der CAN-Controller freigeschaltet wird, müssen also die MOB explizit initialisiert werden!

#### 10.3.2.1.1 Initialisierung

Mit dem Aufruf der Funktion `AT90CAN_Init` werden alle MOB in den Disabled-Modus geschaltet und der CAN-Kanal wird nach einer intern bedingten Zeitverzögerung in den „Normal-Operation“-Modus versetzt. Anschließend wird die Baudrate eingestellt.

```
uint8_t AT90CAN_Init(uint8_t ucbaud)
{
    AT90CAN_Init_AllBuffer();
    AT90CAN_Set_OpMode(ENABLE_MODE);
    //der Normal-Operation-Modus wird gesetzt
    //es wird gewartet bis dieser Modus auch eingestellt ist
    while(!(AT90CAN_Read_Status() & (1 << ENFG)));
    AT90CAN_Set_Baudrate(ucbaud); //die Baudrate wird eingestellt
    return CAN_OK;
}
```

---

<sup>1</sup> Power-On-Reset (siehe Kap. 3).

Jedes einzelne MOb kann in einem der oben beschriebenen Modi als Sende- oder Empfangspuffer eingestellt werden.

```
#define TX_BUFFER_1          0 //MOb 0 wird als TX_BUFFER_1 genannt
#define RX_BUFFER_1           5 //MOb 5 wird als RX_BUFFER_1 genannt
#define AUTOREPLAY_BUFFER_1   10 //MOb 10 wird als AUTOREPLAY_BUFFER_1
                             genannt
```

Die Botschaften-Datenstruktur ist wie folgt definiert:

```
typedef struct
{
    uint8_t EIDE_Bit; //STANDARD_ID oder EXTENDED_ID
    uint8_t RTR_Bit; //DATA_FRAME oder REMOTE_FRAME
    uint8_t RB0_Bit; //Reserved bit: 0x00 oder 0x01
    uint8_t RB1_Bit; //Reserved bit: 0x00 oder 0x02
    uint32_t uID; //11 oder 29 Bit ID
    uint8_t ucLength; //Anzahl der Datenbytes
    uint8_t ucData[8]; //Datenvektor für die maximale Nachrichtenlänge
}can_frame;
```

### 10.3.2.1.2 Botschaften senden

Die Initialisierung des MOb 0 als erster Sendepuffer könnte folgendermaßen aussehen:

```
sSendFrame.EIDE_Bit = STANDARD_ID;
sSendFrame.RB0_Bit = 0x00;
sSendFrame.RTR_Bit = DATA_FRAME;
sSendFrame.ucLength = 0x04;
sSendFrame.uID = 0x77;
AT90CAN_Init_Buffer(TX_BUFFER_1, &sSendFrame);
```

wobei sSendFrame vom Datentyp can\_frame ist. Diese Elemente der Datenstruktur werden in den Speicher der MOb-Seite gespeichert. Vor dem Senden der Botschaft werden die Sendedaten aktualisiert und mit dem Aufruf:

```
AT90CAN_Send_Message(TX_BUFFER_1, &sSendFrame);
```

steht die Botschaft zum Senden bereit.

### 10.3.2.1.3 Botschaften empfangen

Für eine bessere Übersicht wird für den Datenempfang eine Datenstruktur sRecframe vom gleichen Datentyp can\_frame deklariert. Ähnlich wie bei der Initialisierung des Sendepuffers wird auch MOb 5 für den Empfang von Botschaften mit dem Standard-Identifier 0x101 initialisiert.

---

```
sRecFrame.EIDE_Bit = STANDARD_ID;
sRecFrame.RB0_Bit = 0x00;
sRecFrame.RTR_Bit = DATA_FRAME;
sRecFrame.ucLength = 0x04;
sRecFrame.ulID = 0x101;
AT90CAN_Init_Buffer(RX_BUFFER_1, &sRecFrame);
```

Die gespeicherten Parameter können mit einer erneuten Initialisierung geändert werden. Über eine Filtermaske wird der Empfang von Botschaften mit mehreren Identifier ermöglicht. Die Maske als Datenstruktur muss zuerst definiert werden und dann mit dem Aufruf der Funktion AT90CAN\_Set\_ReceiveMask gespeichert. Mit den folgenden Einstellungen empfängt der Puffer RX\_BUFFER\_1 die CAN-Botschaften mit den Standard-Identifier 101 und 103.

```
sMask.EIDE_Bit = STANDARD_ID;
sMask.IDE_MaskBit = 1; //es wird eine Maske verwendet
sMask.RTR_MaskBit = 0; //keine RTR-Botschaft
sMask.ulMask = 0x7FD; //mit dieser Maske kann der Puffer 2 ID's
empfangen
AT90CAN_Set_ReceiveMask(RX_BUFFER_1, &sMask); //Setzen der Maske
//Empfang der ersten Nachricht wird freigegeben
AT90CAN_Enable_Reception(RX_BUFFER_1);
```

Der Empfang einer neuen Botschaft in einer Mailbox wird durch das Setzen des Bit RXOK in das Register CANSTMOB signalisiert. Ein CAN-Interrupt kann ausgelöst werden wenn er entsprechend eingestellt wurde. Die Funktion AT90CAN\_Receive\_Message prüft das Bit RXOK der gewünschten Mailbox und wenn dieses gesetzt ist, gibt sie CAN\_RECEIVED zurück und speichert die neue Botschaft in einer Datenstruktur vom Typ can\_frame.

```
if(AT90CAN_Receive_Message(RX_BUFFER_1, &sRecFrame) == CAN_RECEIVED)
{
    //Auswertung der Botschaft
}
```

### 10.3.3 Implementierung mit der CAN-Bibliothek des Roboterclub Aachen

Bei dieser Bibliothek [5] muss zunächst über #defines in config.h ausgesucht werden, welcher CAN-Controller benutzt werden soll. Wir zeigen hier die Vorgehensweise anhand des MCP2515. Die Initialisierung erfolgt mit dem 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);

```

Die Botschaften-Datenstruktur ist 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 MCP2515 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 wie folgt ab:

```
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 verschickt, muss lediglich in einem 100 ms Task einen Sendebefehl mit den entsprechenden Daten absetzen.

Das Empfangen von Botschaften im Polling-Betrieb ist hier 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 vorzubelegen. 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.

---

## 10.4 CAN-Transportprotokoll

Das CAN-Transportprotokoll nach ISO 15765-2 bildet eine Transportschicht (siehe Abschn. 6.1.4) ab, deren Aufgaben lauten

- Übertragung von großen Datenblöcken (Segmentierung),
- Flusskontrolle,
- Abstraktion der Transportschichten gegenüber den Anwendungsschichten.

Diese Aufgaben sind im CAN-ISO-TP, wie es auch genannt wird, relativ einfach gelöst. Die Idee besteht darin, dass die Anwendung große Datenblöcke (also größer acht Bytes)

wie eine einzige Botschaft versendet, die Segmentierung wird von der ISO-TP-Schicht übernommen.

Dazu stellt das Protokoll innerhalb des Datenrahmens von acht Byte vier verschiedene Botschaftstypen zur Verfügung: Single Frame, First Frame, Consecutive Frame, Flow Control Frame.

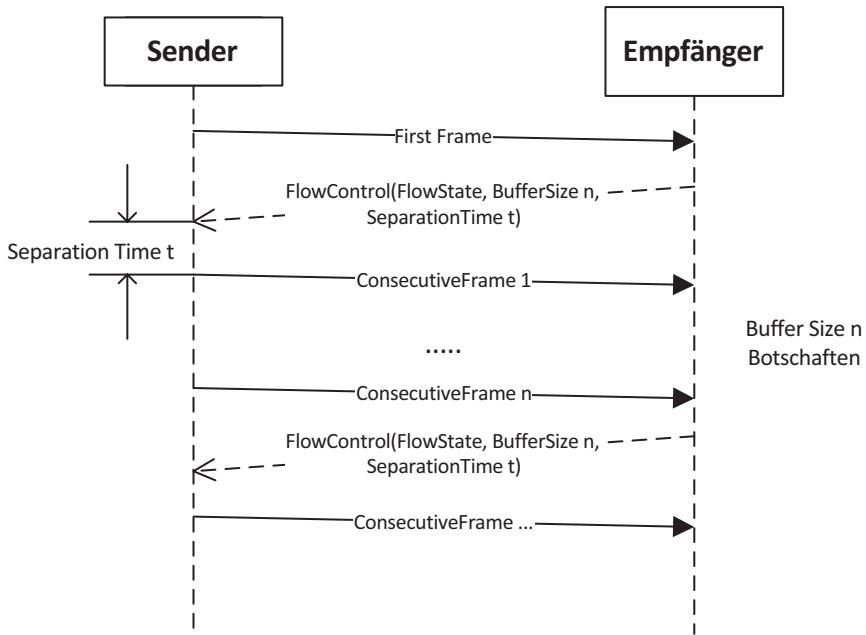
Grundsätzlich wird über den Message-Identifier codiert, ob die Nachricht eine „normale“ Layer-2-CAN-Botschaft ist oder eine ISO-TP-Botschaft, sprich, das müssen Sender und Empfänger vorher vereinbaren. In einer ISO-TP-Botschaft werden die ersten vier Bit zur Codierung des Rahmentyps verwendet. Abb. 10.8 zeigt die derzeit spezifizierten Rahmen. Sind diese vier Bit gleich 0, so wird die Botschaft als Single Frame interpretiert, in dem noch sieben Nutzdatenbytes zur Verfügung stehen. In den unteren vier Bit der Botschaft ist die Zahl (eins bis sieben) der Nutzdatenbytes codiert, während bei allen ISO-TP-Botschaften das DLE-Byte auf acht gesetzt ist.

Soll nun eine segmentierte Botschaft gesendet werden, initiiert der Sender die Übertragung mit einem „First Frame“, in dessen ersten Byte die obersten vier Bit die Zahl 0001 (binär) führen und die unteren vier Bit zusammen mit dem zweiten Byte die Gesamtlänge der Botschaft bestimmen. Da das Längenfeld 12 Bit besitzt, können damit Botschaften bis zur Länge 4095 segmentiert werden. Der Slave antwortet mit einem Flow Control Frame, das mit 0011 (binär) im oberen Nibble (=vier Bit) des ersten Byte versehen ist. Das zweite Nibble codiert den Flow State, dieser kann Continue to Send (0x00) oder Wait (0x01) beinhalten. Das zweite Byte enthält die Block Size (BS=1...255) also die Anzahl von weiteren CAN Botschaften, die unmittelbar empfangen werden können (Puffergröße des Empfängers). BS gleich 0 bedeutet beliebig viele Blöcke. Die Separation Time gibt die minimale Zeit in Millisekunden an, die der Sender zwischen zwei Botschaften warten muss.

Nach der Empfängerantwort schickt der Sender nun die Consecutive Frames gemäß der nunmehr getroffenen Vereinbarungen, hier wird im oberen Nibble eine 0010 codiert



**Abb. 10.8** Frames des ISO-Transportprotokolls. (Nach [4] und [8])



**Abb. 10.9** Ablauf des Sendens einer segmentierten Botschaft im ISO-TP. (Nach [4] und [8])

und im unteren eine Sequenznummer, die genutzt werden kann, um die Reihenfolgerichtigkeit der Botschaften nach dem Empfang sicherzustellen (auch im Falle einer Fehlerbehandlung wie in Kap. 6 beschrieben). Nach der vereinbarten Zahl der Botschaften (Block Size) wartet der Sender, bis der Empfänger mit einem erneuten Flow Control Frame quittiert und setzt dann die Übertragung fort. In Abb. 10.9 ist der Ablauf der Kommunikation als UML Sequenzdiagramm zusammengefasst.

Gemäß der Norm müssen Datenblöcke, die die CAN-Botschaft nicht füllen, aufgefüllt werden (padding) um lange Nullfolgen zu vermeiden.

Ein Beispiel:

Der Sender möchte die 24 Byte lange Botschaft: 0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08 0x09 0x0A 0x0B 0x0C 0x0D 0x0E 0x0F 0x10 0x11 0x12 0x13 0x14 0x15 0x16 0x17 0x18 verschicken.

Das First Frame ist: 0x10 0x18 0x01 0x02 0x03 0x04 0x05 0x06, wobei die 0x10 0x18 am Anfang bedeutet: First Frame der Länge 0x018 (also 24).

Der Empfänger antwortet mit 0x30 0x00 0x00 (also Flow Control Frame Continue, beliebig viele Botschaften, keine minimale Wartezeit).

Danach sendet der Sender: 0x21 0x07 0x08 0x09 0x0A 0x0B 0x0C 0x0D (also erstes consecutive Frame) mit den entsprechenden nächsten Daten. Die weiteren Botschaften sind:

0x22 0x0E 0x0F 0x10 0x11 0x12 0x13 0x14 und 0x23 0x15 0x16 0x17 0x18 0xAA 0xAA 0xAA (also die folgenden, weitergezählten Frames). Im letzten Frame wird das padding angewendet.

Das ISO-TP ist Bestandteil jedes Diagnosesystems im Fahrzeug. Es wird durch alternative Transportprotokolle (beispielsweise dem VW eigenen) erweitert. In der weit verbreiteten AUTOSAR-middleware ist es ein fester Bestandteil [9], schlanke Implementierungen sind auf github zu finden [10].

---

## 10.5 CANopen in der Industriesteuerungstechnik

Für die industrielle Steuerungstechnik existiert ein Kommunikationsprotokoll mit dem Namen CANopen auf Basis der CAN Sicherungsschicht [11]. CANopen wurde als standardisiertes Embedded-Netzwerk mit hochflexiblen Konfigurationsmöglichkeiten als offener Standard insbesondere für den Mittelstand entwickelt. Es war ursprünglich für bewegungsorientierte Maschinensteuerungen, wie beispielsweise Handhabungssysteme, konzipiert. Heute wird es in verschiedenen Anwendungsbereichen eingesetzt, unter anderem in medizinischen Geräten, Geländewagen, Schiffselektronik, Bahn-anwendungen oder in der Gebäudeautomatisierung beziehungsweise Aufzugstechnik.

Die Vereinigung CiA (CAN in Automation) mit Sitz in Nürnberg kümmert sich um die Festlegung und Fortschreibung der Standards, die jeweils den Namen CiA, gefolgt von einer Nummer, tragen. CANopen stellt Kommunikationsobjekte zur Verfügung, die sich wie folgt aufteilen:

- Servicedatenobjekte (SDO) zur Parametrierung von Objektverzeichniseinträgen,
- Prozessdatenobjekte (PDO) zum Transport von Echtzeitdaten,
- Netzwerkmanagement-Objekte (NMT) zur Steuerung des Zustandsautomaten des CANopen-Geräts und zur Überwachung der Knoten,
- weitere Objekte wie Synchronisationsobjekt (SYNC), Zeitstempel und Fehler-Nachrichten (EMCY).

Das hier erwähnte Objektverzeichnis (OV) (engl. Object Dictionary (OD)) ist das Herz eines CANopen-Stacks. Es enthält alle Kommunikationsobjekte und alle Anwendungsobjekte. Das OV ist im CANopen-Gerätemodell das Bindeglied zwischen der Anwendung und der CANopen-Kommunikationseinheit. Das heißt, die Anwendung arbeitet mit Prozessparametern, schickt diese an den Protocol-Stack, der mithilfe des OV eine CAN-Botschaft erzeugt. Umgekehrt wird der Protocol-Stack eine empfangene Botschaft mithilfe des Objektverzeichnisses dekodieren und an die Anwendung als Prozessparameter weitergeben. Darüber hinaus enthält das OV auch die gesamte Parametrierung des Kommunikationsablaufs.

Der NMT-Master sorgt dafür, dass alle Geräte im Netzwerk angemeldet und funktionsbereit sind. Er versendet in der Regel auch einen Heartbeat (eine wiederkehrende Botschaft) zur Synchronisation der Teilnehmer. Inzwischen existieren eine Reihe von offenen CANopen-Stacks, die man bei github herunterladen kann (beispielsweise [12] und [13]). Eine Eigenentwicklung lohnt sich in der Regel nicht. Da der Stack recht datenintensiv ist, sollte er auf einem größeren System (idealerweise einem embedded Linux-System) eingesetzt werden.

In [14] und [15] ist eine Anwendung für Aufzüge (CiA417) beschrieben.

---

## Literatur

1. NXP: CAN Bosch Controller Area Network (CAN) Version 2.0 PROTOCOL STANDARD. <http://www.nxp.com/assets/documents/data/en/reference-manuals/BCANPSV2.pdf>. Zugegriffen: 1. Apr. 2021.
2. ISO 11898: Road vehicles – Controller area network (CAN) (6 Teile).
3. Etschberger, K. (Hrsg.). (1994). *CAN-Controller Area Network – Grundlagen, Protokolle, Bausteine, Anwendungen*. Hanser.
4. Zimmermann, W., & Schmidgall, R. (2014). *Bussysteme in der Fahrzeugtechnik – Protokolle, Standards und Softwarearchitektur* (5. Aufl.). Springer.
5. Fabian Greif, Roboterclub Aachen: Universelle CAN Bibliothek. (2008). <http://www.kreatives-chaos.com/artikel/universelle-can-bibliothek>. Zugegriffen: Jan. 2021.
6. Microchip Technology Inc. MCP2515 Datasheet. <https://www.microchip.com/wwwproducts/en/en010406>. Zugegriffen: 10. Jan. 2021.
7. Microchip Technology Inc. 8-bit Microcontroller with 32K/64K/128K Bytes of ISP Flash and CAN-Controller. (2015). <https://www.microchip.com/wwwproducts/en/AT90CAN128>. Zugegriffen: 10. Jan. 2021.
8. ISO: ISO 15765-2:2016. 15765-2:2011 Road vehicles – Diagnostic communication over Controller Area Network (DoCAN) – Part 2: Transport protocol and network layer services 2016. Auflage. <https://www.iso.org/standard/66574.html>. Zugegriffen: 9. Jan. 2021.
9. AUTOSAR: Classic Platform. <https://www.autosar.org/standards/classic-platform/>. Zugegriffen: 10. Jan. 2021.
10. Peplin, C., & Boll, D. (Ford): ISO-TP (ISO 15765-2) Support Library in C. <https://github.com/openxc/isotp-c/tree/master/src/isotp>. Zugegriffen: 23. Jan. 2021.
11. CiA: CANopen Protocol. <https://www.can-cia.org/canopen/>. Zugegriffen: 1. Febr. 2021.
12. CAN Festival: CAN Festival free CANopen framework. <https://canfestival.org/>. Zugegriffen: 1. Febr. 2021.
13. <https://github.com/CANopenNode/CANopenNode>. Zugegriffen: 1. Febr. 2021.
14. Sußmann, N., & Meroth, A. (2017). „Model based development and verification of CANopen components“ 22nd IEEE International Conference on Emerging Technologies And Factory Automation ETFA 2017, September 12–15, Limassol, Cyprus.
15. CANopen Lift. <http://de.canopen-lift.org>. Zugegriffen: 1. Febr. 2021.



## Zusammenfassung

In diesem Kapitel werden einige Aspekte zum Modbus beschrieben, der als industrieller Feldbus weite Verbreitung gefunden hat.

## Übersicht

Der MODBUS-Standard spezifiziert die Anwendungsschicht (die 7. Schicht des OSI-Modells) und teilweise die Sicherungsschicht des Kommunikationsprotokolls zwischen vernetzten Geräten [1]. Aufgrund seiner Vielseitigkeit hat er in der Industrie breite Verwendung gefunden. Die Modbus Organisation berät und gestaltet die Standards rund um Modbus.

Die Kommunikation kann entweder über TCP/IP (IEC 61.158), oder über eine serielle, asynchrone Schnittstelle wie TIA<sup>1</sup>/EIA<sup>2</sup>-232 oder TIA/EIA-485 stattfinden [2]. In jüngster Zeit wurde auch eine sichere Variante des Modbus/TCP Protokoll auf der Basis von Transport Layer Security (TLS) veröffentlicht, die digitale X.509v3 Zertifikate nutzt, um Server und Client zu authentifizieren, außerdem wird eine rollenbasierte Zugriffssteuerung vorgeschlagen.

Über Modbus sind in der Regel Steuergeräte mit höherer Leistungsfähigkeit verbunden, sodass die Variante TCP/IP in der Regel vorzuziehen ist. Dies fällt in diesem Buch aufgrund der niedrigen Leistung der Prozessoren der AVR-Familie, die hier beschrieben

<sup>1</sup>TIA – Telecommunications Industry Association.

<sup>2</sup>EIA – Electronic Industry Alliance.

wird, weg. Dennoch möchten wir hier die Variante mit ASCII-Übertragung vorstellen, da das Protokoll dazu genutzt werden kann, Sensoren mit geringerer Rechnerleistung in einem Modbus-System anzuschließen und den Rahmen der Anwendungsschicht transparent über ein zwischengeschaltetes Gateway in das System einzuspeisen.

Für das serielle Protokoll definiert MODBUS die Sicherungsschicht (Data Link Layer) des OSI-Modells. Es wird ein Master-Slave-Bus mit bis zu 247 Slaves spezifiziert. 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 unter der Bedingung, dass zu jedem Zeitpunkt nur ein Master aktiv ist, auch als Multi-Master betrieben werden. Der Datentransfer zwischen einem Master und einem einzelnen räumlich nicht zu weit entfernten Slave (point-to-point Verbindung) bei einer niedrigen Übertragungsrate kann über TIA/EIA-232 stattfinden, ansonsten über TIA/EIA-485. In dem hier kurz beschriebenen Szenario arbeitet ein Mikrocontroller mit einem oder mehreren lokalen Sensoren, die beispielsweise über I<sup>2</sup>C angeschlossen sind, verarbeitet die Daten vor und überträgt sie in einem MODBUS-Protokollrahmen über TIA/EIA-232 an den Master, der wiederum über das TCP/Modbus-Protokoll an die Steuerungs- oder Prozessleitebene angeschlossen ist.

---

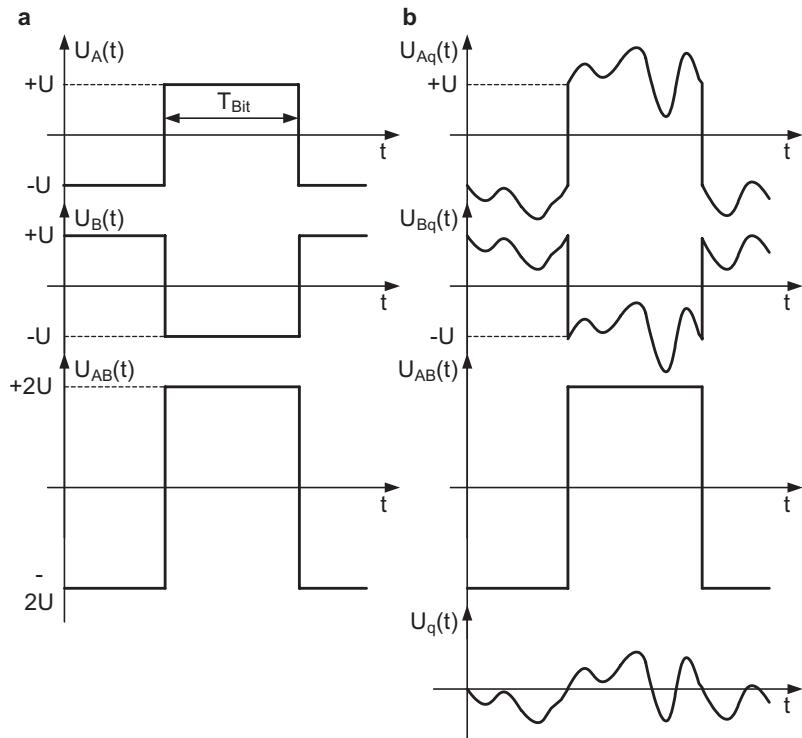
## 11.1 TIA/EIA-485 als Bitübertragungsschicht für MODBUS

Der industrielle Standard TIA/EIA-485, auch als RS-485 bekannt, ist als Nachfolger von RS<sup>3</sup>-232 (offiziell TIA/EIA-232) entstanden, um höhere Datenraten (<50 Mbit/s) zu erreichen, längere Distanzen (<1200 m) zu überbrücken, unempfindlicher als eine auf Spannungspegeln aufbauende Schnittstelle gegenüber Störungen zu sein und die Vernetzung von mehreren Geräten an einem physikalischen Kanal zu ermöglichen. RS-485 ist eine serielle Schnittstelle, die die erste Schicht des OSI-Modells (siehe Abschn. 6.1.1) implementiert. Die Bitübertragung findet im Gegenbetrieb (Vollduplex) oder Wechselbetrieb (Halbduplex) statt und ist symmetrisch und asynchron. Die symmetrische Übertragung benutzt zwei Signalleitungen, die verdrillt sein sollten und 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 Gleichaktstörungen (Abb. 11.1), die auf den Signalleitungen auftreten, und verbessert die Bitfehlerrate.

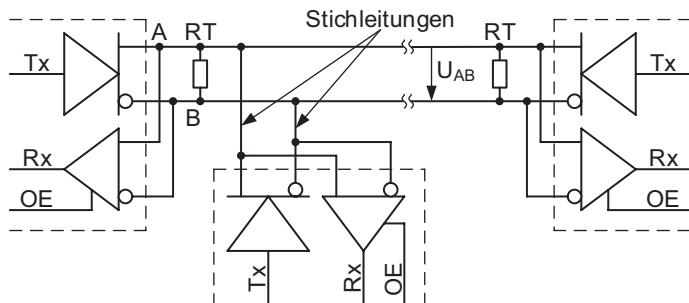
RS-485 ermöglicht theoretisch die Verbindung mehrerer Geräte oder Sensoren, die kein gemeinsames Bezugspotenzial (Masse) haben. In der Praxis wird das Verbinden

---

<sup>3</sup>RS – Recommended Standard.



**Abb. 11.1** a Störungsfreie symmetrische Übertragung; b symmetrische Übertragung mit Gleichaktstörungen



**Abb. 11.2** MODBUS-Netzwerktopologie

der Massen über eine elektrische Leitung empfohlen. Es wird ebenso empfohlen, dass alle Geräte direkt an den Signalbus angeschlossen werden (Daisy-Chain-Topologie), oder mittels kurzen Stichleitungen wie in der Abb. 11.2. Der Datenbus soll mit Abschlusswiderständen (RT) terminiert werden. Diese Widerstände entsprechen dem

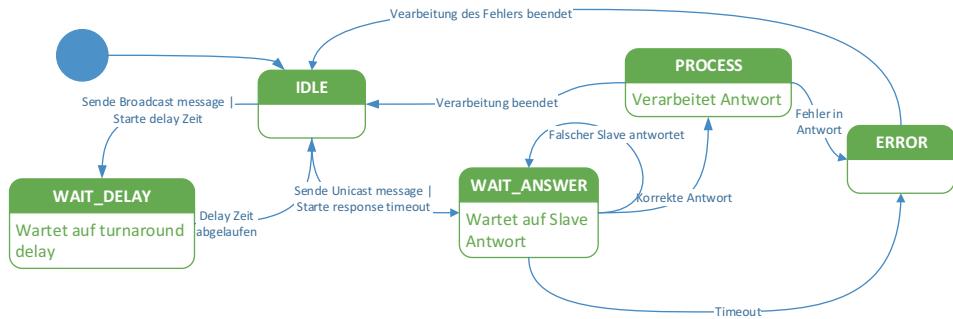


Abb. 11.3 Zustandsautomat des MODBUS-Masters

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 [3] zu entnehmen.

## 11.2 MODBUS-Kommunikation

Die MODBUS-Übertragung auf seriellen Leitungen ist datenwort-orientiert und kann im Remote-Terminal-Unit- (RTU) oder ASCII-Modus stattfinden. In diesem Abschnitt beschäftigen wir uns ausschließlich mit der Übertragung auf seriellen Leitungen und nicht mit der Kommunikation über TCP/IP gemäß IEC 61158.

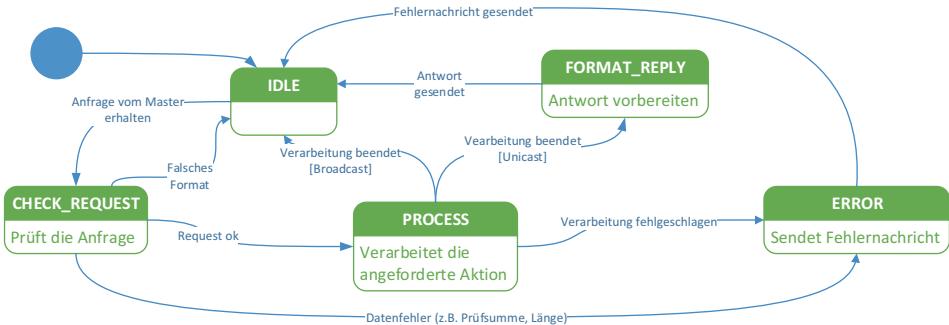
Grundsätzlich geht jede Kommunikation vom Master aus, es ist also keine direkte Slave-zu-Slave Kommunikation möglich. Wenn der Master eine Nachricht sendet, antwortet der Slave mit seiner eigenen Adresse. Bei einer Broadcast-Nachricht wird keine Antwort gesendet.

Master und Slave arbeiten in einer übergeordneten Statemachine wie in Abb. 11.3 und 11.4 gezeigt [2].

### 11.2.1 Remote-Terminal-Unit-Übertragung

Im RTU-Modus besteht eine MODBUS-Nachricht aus der Adresse des angesprochenen Slaves, einem Befehlscode, 0 bis 252 Datenbytes und einer CRC<sup>4</sup>- Prüfsumme (siehe Abschn. 6.1.2.5). Diese Adressierungsart (Gerät plus Funktionscode) unterscheidet sich also grundsätzlich von der des CAN (Botschaftenadressierung), die in Kap. 10 beschrieben wurde. Abb. 11.5 oben zeigt den RTU-Datenrahmen.

<sup>4</sup>CRC –cyclic redundancy check (zyklische Redundanzprüfung).

**Abb. 11.4** Zustandsautomat des MODBUS-Slaves

Slave-Adresse	Function code	Daten	CRC low	CRC high	
1 Byte	1 Byte	1 Byte ... 252 Bytes	1 Byte	1 Byte	RTU Frame

Start :	Slave-Adresse	Function code	Daten	LRC	Ende CR - LF	
1 char	2 char	2 char	0... 2 x 252 char	2 char	2 char	ASCII Frame

**Abb. 11.5** Modbus Frames im seriellen Modus (oben RTU, unten ASCII)

Jedem Slave aus dem Netzwerk wird eine 1-Byte-Adresse aus dem Bereich 1...247 zugewiesen. Die Adresse 0 ist für einen Rundruf (broadcast) 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.

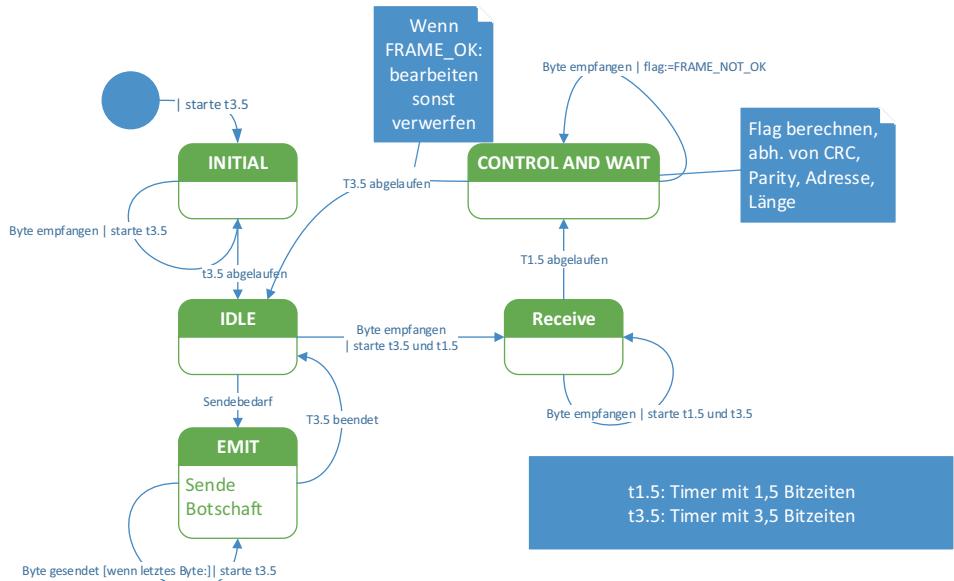
Ein Datenwort ist im RTU-Modus immer elf Bit groß und ist ähnlich wie der der UART-Schnittstelle (Kap. 7) aufgebaut. Der Pulsrahmen beginnt mit einem Startbit, gefolgt von acht Datenbits, einem Paritätsbit und einem Stoppbitt. Die Übertragung des Datenbytes beginnt immer mit dem niederwertigsten Bit zuerst. In der Standard-Einstellung wird mit gerader Parität gerechnet. Die MODBUS-Spezifikation fordert, dass auch ungerade Parität einstellbar sein soll. Soll das Paritätsbit ganz wegfallen, muss der Datenrahmen mit zwei Stoppbits beendet werden. Das Paritätsbit wird auf gleiche Weise wie in Abschn. 7 berechnet und kann damit per UART Hardware ausgeführt werden.

### Alternativer Aufbau des Datenworts im RTU-Modus

Start	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7	Bit 8	Parity	Stop
Start	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7	Bit 8	Stop	Stop

Die Baudrate und die Struktur des Datenworts 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 mindestens die Dauer von 3,5 Bytes keine Busaktivitäten stattgefunden haben. Wenn diese Wartezeit zwischen zwei Nachrichten nicht eingehalten wird, betrachten die Slaves die neue Nachricht als fehlerhafte Fortsetzung der vorigen. Ein Rahmenstartbyte ist daher nicht vorgesehen, da sich der Rahmenstart aus diesem Bus-Timing (3,5 Bytelängen Wartezeit vor dem Rahmenstart) ergibt, somit entfällt die Notwendigkeit zur Herstellung von Codetransparenz (siehe Abschn. 6.1.2.1).

Bei der Übertragung einer Nachricht ist zwischen den einzelnen Bytes eine Wartezeit erlaubt, die nicht größer als die Übertragungszeit von 1,5 Bytes sein darf, ansonsten erfolgt ein Abbruch und das Frame wird verworfen (FRAME\_NOT\_OK in Abb. 11.6). 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,



**Abb. 11.6** Zustandsautomat mit Timings für den Ablauf der Master- und Slavekommunikation

gefolgt von dem empfangenen Befehlscode oder einem Fehlercode. Der Master prüft, ob die Antwort des von ihm adressierten Slaves zurückkommt und die Integrität dieser Nachricht (Parität plus Prüfsumme). Im Fehlerfall kann er die Nachricht noch einmal senden Abb. 11.6.

### 11.2.2 ASCII-Übertragung

Im ASCII-Modus wird die zu übertragende Nachricht wie im RTU-Modus gebildet. Anstatt der CRC-Prüfsumme wird aber eine 1-Byte große Längsparität<sup>5</sup>-Prüfsumme berechnet und angehängt. Ausführliche Beispiele für die Berechnung der zwei Prüfsummen befinden sich in [2]. In diesem Modus werden jedem Byte der Grundnachricht zwei ASCII-Zeichen ('0', '1'...'9', 'A'...'F') aus der 7-Bit-ASCII-Tabelle zugewiesen. Diese Zeichen codieren die hexadezimalen Werte (0...F) des höherwertigen und niedrigwertigen Nibble eines Bytes. Beispiel: Das Byte 0x5B ist als zwei Zeichen kodiert: 0x35 und 0x42 (0x35 = "5", und 0x42 = "B" in ASCII). 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. Ein Datenwort besteht immer aus zehn Bits und beinhaltet nur sieben Datenbits (ASCII).

Alternativer Aufbau des Datenworts im ASCII-Modus

Start	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7	Parity	Stop
Start	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7	Stop	Stop

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;
```

---

<sup>5</sup>en. LRC – longitudinal redundancy check.

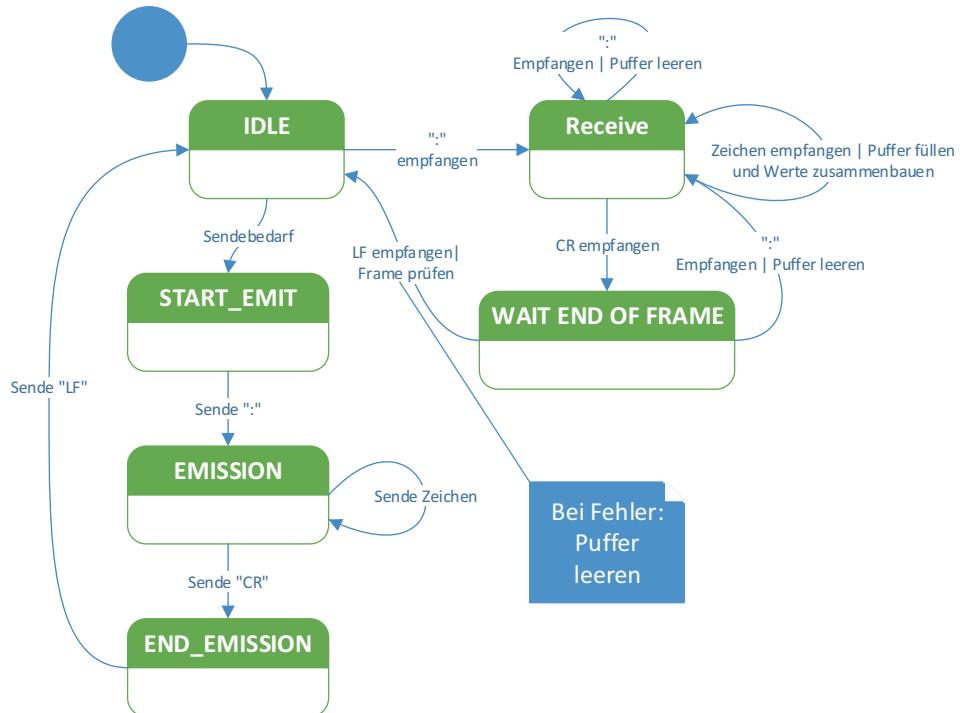


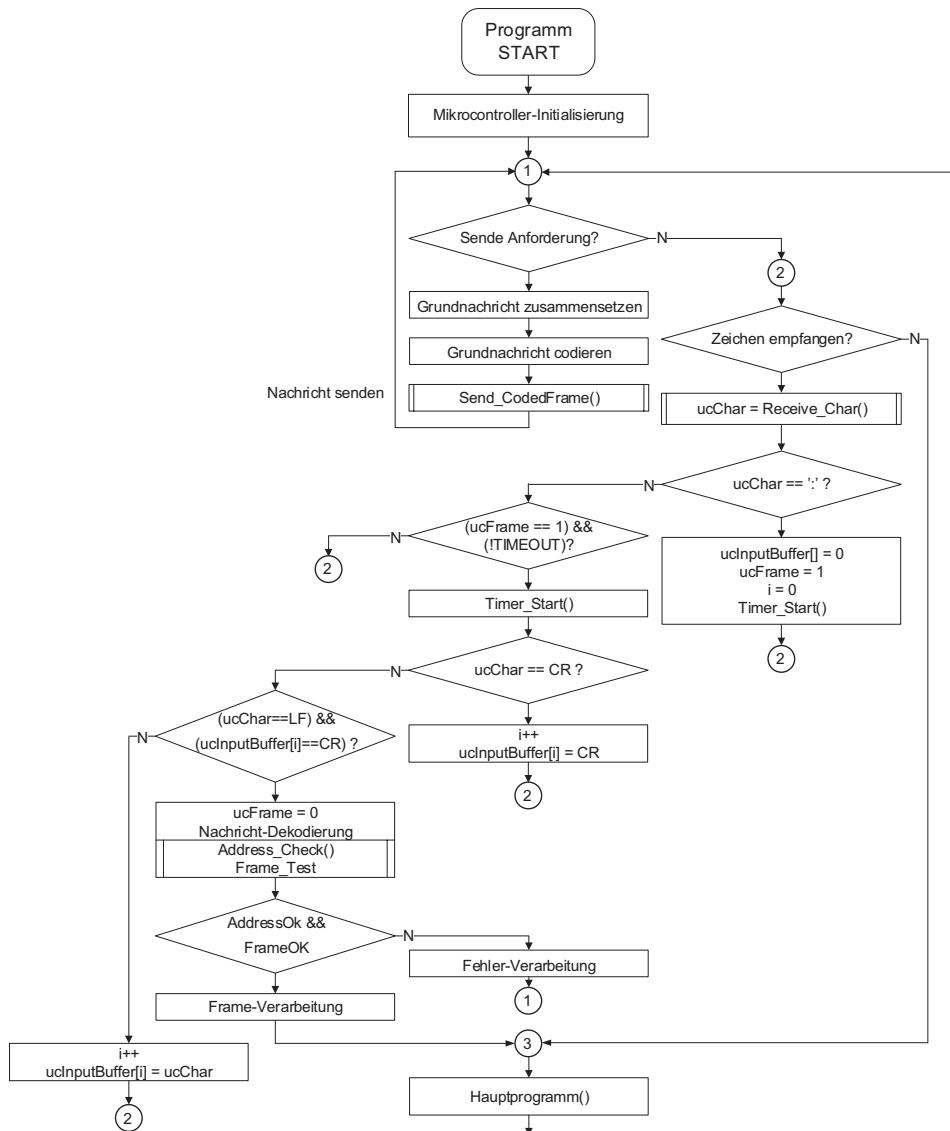
Abb. 11.7 Zustandsautomat Master und Slave auf Byteebene (ASCII-Modus)

```
//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;
}
```

Als Startzeichen einer neuen Nachricht wird ein Doppelpunkt-Zeichen ‘:’ gesendet. Die codierte Grundnachricht wird zeichenweise übertragen und mit der Zeichenfolge ‘CR<sup>6</sup>‘ und ‘LF<sup>7</sup>‘ beendet. Die Wartezeit zwischen der Übertragung zweier Zeichen soll einstellbar sein (siehe auch Abb. 11.5 unten). Sowohl der Master als auch der Slave implementieren im ASCII-Modus ein ähnliches Zustandsdiagramm wie im RTU-Modus. Durch Verwendung der Rahmenstart (:) und Rahmenendekennung (CR LF) werden aber keine Timings vorgegeben (siehe Abb. 11.7). Die Forderung nach Codetransparenz

<sup>6</sup>CR – carriage return=0x0D (Wagenrücklauf)

<sup>7</sup>LF – line feed=0x0A (Zeilenumbruch).



**Abb. 11.8** MODBUS-Ablaufdiagramm des ASCII-Modus

entfällt, da ja nur die hexadezimalen ASCII-Zeichen für die Datenübertragung herangezogen werden.

In Abb. 11.8 ist zudem ein Ablaufdiagramm dargestellt. 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.

Schließlich sei noch die Bildung der LRC- oder Längenparitäts-Prüfsumme angesprochen. Diese wird berechnet, indem aufeinanderfolgende Bytes (außer dem “:“ und dem CR LF) in der Nachricht modulo-addiert werden, d. h. alle Überträge werden verworfen. Dies entspricht einer kontinuierlichen XOR-Verknüpfung aller übertragenen Bytes, wobei ein Byte aus 7 Bit Wert und 1 Bit Parität besteht. Im Prinzip ist die LRC eine einfache Paritätsprüfung aller jeweils ersten, zweiten, dritten usw. Bits einer Nachricht. Wenn diese Summe zur Nachricht hinzugaddiert wird, muss folglich bei erneuter LRC-Berechnung eine Null herauskommen. Die Kombination von Paritätsbit eines Bytes und Längsparität kann also zur Lokalisierung und Korrektur eines Bitfehlers herangezogen werden. In MODBUS wird die LRC im Zweierkomplement übertragen. Die Spezifikation schlägt den folgenden Code zur Bildung der LRC vor, sie liefert die LRC zurück. auchMsg enthält die Botschaft ohne “:“ und CR LF

```
static unsigned char LRC(auchMsg, usDataLen)
unsigned char *auchMsg ;           /* Botschaft (message buffer)*/
unsigned short usDataLen ;        /* Länge der Botschaft */
{
    unsigned char uchLRC=0;         /* LRC wird initialisiert */
    while (usDataLen--)           /* Durchlaufe message buffer */
        uchLRC+=*auchMsg++;       /* Addiere die Zeichen */
    return ((unsigned char)(-((char)uchLRC))); /* Zweierkomplement*/
}
```

---

## Literatur

1. Modbus.org. MODBUS Application Protocol Specification V1.1b3, 2016. [www.modbus.org](http://www.modbus.org). Zugriffen: 11. Jan. 2021.
2. Modbus.org. MODBUS over serial line. Specification and implementation guide V1.02, 2016. [www.modbus.org](http://www.modbus.org)
3. Corrigan, S. Interface circuits for TIA/EIA-485 (RS-485). Application Report SLLA036D Revised August 2008. [www.ti.com](http://www.ti.com). Zugriffen: 1. Apr. 2021.



# Eindrahtbussysteme

12

## Zusammenfassung

In diesem Kapitel werden weitere Eindrahtbussysteme beschrieben, die für Sensornetzwerke genutzt werden können.

Eindrahtbussysteme erfreuen sich wegen ihrer sehr einfachen und kostengünstigen Verdrahtung insbesondere für Teilsysteme (Subbusse) in der Industrietechnik, der Haustechnik und im Fahrzeug einer zunehmenden Beliebtheit. Sie basieren teilweise auf dem UART. Wir haben hier die Systeme 1-Wire®-Bus und UNI/O® für eine genauere Beschreibung ausgesucht und den LIN-Bus zumindest so erwähnt, dass ein grundlegendes Verständnis erworben werden kann, in der weiterführenden Literatur finden sich detailliertere Angaben.

Die hier beschriebenen Eindrahtbussysteme sind asynchrone Master-Slave-Busse, die für die Kommunikation nur eine Datenleitung benötigen, da kein Takt übertragen wird und die Kommunikation im bidirektional im Halbduplexbetrieb erfolgt. Ein einziger Busmaster steuert den Datenfluss in einem Netzwerk dieser Art. Dieser initiiert jede Kommunikation, die meist zwischen dem Master und dem adressierten Slave stattfindet.

Die Originalversion dieses Kapitels wurde revidiert. Ein Erratum ist verfügbar unter  
[https://doi.org/10.1007/978-3-658-31709-6\\_27](https://doi.org/10.1007/978-3-658-31709-6_27)

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann  
[https://doi.org/10.1007/978-3-658-31709-6\\_12](https://doi.org/10.1007/978-3-658-31709-6_12).

Manche Protokolle erlauben auch die Kommunikation zwischen dem Master und allen Slaves, der direkte Datenaustausch zwischen den Slaves wird in der Regel nicht unterstützt. Die Bauteile, die solche Protokolle implementieren, sind dank des einfachen Aufbaus schaltungstechnisch leicht zu integrieren. Viele Slaves benötigen dafür nur drei Anschlüsse: Masse (Ground), Versorgungsspannung und Datenleitung. Bei manchen Bussen kann der Busmaster die Slaves über die Datenleitung sogar mit Energie versorgen. Diese einfache Hardware-Konfiguration erschwert oder verhindert die Unterscheidung identischer Slaves, die am selben Bus angeschlossen sind – in diesem Fall müssten noch weitere Pins für die hardwareseitige Codierung vorgesehen werden. Die Adresse muss bei diesen Slaves während des Herstellungsprozesses in die Bauteile eingebrannt werden, oder man verzichtet auf eine Mehrfachverwendung. Manche Sensoren werden ab Katalog mit unterschiedlichen Adressen ausgeliefert (wie auch bei I<sup>2</sup>C/TWI). Die asynchrone Übertragung fordert engere Zeitvorgaben vom Master beim Aufbau der einzelnen Bits.

---

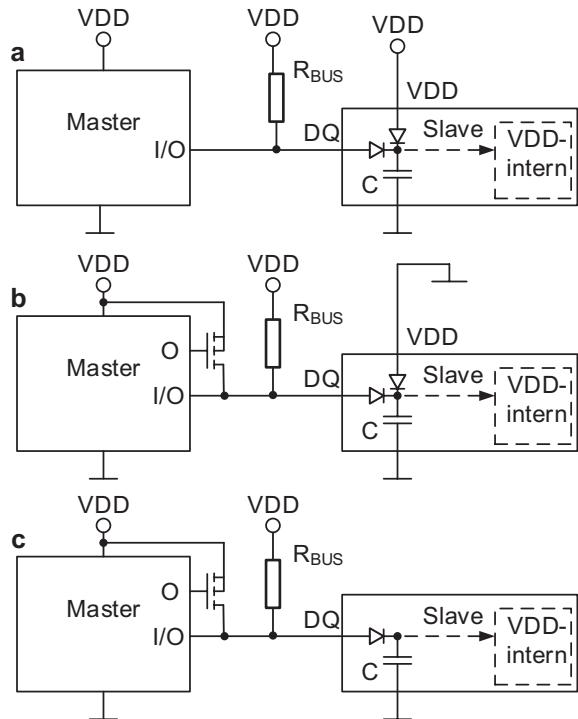
## 12.1 1-Wire® -BUS

1-Wire® ist ein proprietäres Bus-Protokoll, das von der Firma Dallas Semiconductor, jetzt Maxim Integrated [1], für die Übertragung kleinerer Datenmengen mit längeren Übertragungspausen für kurze Distanzen konzipiert wurde, beispielsweise für den Betrieb in Schaltschränken. Das Protokoll ermöglicht eine Halbduplex-Kommunikation zwischen einem Master mit bis zu  $2^{64}$  Slaves. Um den Bus so einfach wie möglich zu gestalten, benötigen die Slaves keine externen Bauteile und werden über einen internen, 64-Bit langen, unveränderbaren Identifikationscode (ID) identifiziert. Dieser besteht aus einem 8-Bit-Family-Code, einer 48-Bit-Seriennummer (Unique-Device-ID) sowie einer 8-Bit-CRC-Prüfsumme, sodass alle angeschlossenen Knoten eindeutig identifizierbar sind (siehe Abschn. 12.2.4.2). Die übertragenen Pulse sind unipolar und NRZ-codiert, die Kommunikation findet auf einer einzigen Datenleitung statt. Die Busleitung wird über einen Pull-up-Widerstand an die Versorgungsspannung angeschlossen. Für diesen Bus wurden Sensoren, A/D-Wandler, EEPROMs u. a. entwickelt.

### 12.1.1 Netzwerktopologie

Eine minimale Konfiguration einer 1-wire-Beschaltung ist in Abb. 12.1a dargestellt. Der Slave wird über den Anschluss VDD versorgt, der Datenaustausch erfolgt über den Pin DQ. Der Master soll für diesen Bus am besten einen Open-drain- oder einen bidirektionalen Anschluss besitzen. Eine interne Beschaltung der Slaves ermöglicht eine so genannte „parasitäre“ Spannungsversorgung. In diesem Fall wird auf eine zusätzliche Versorgungsleitung verzichtet wie in Abb. 12.1b oder sogar auf den Versorgungspin wie in Abb. 12.1c. Während der rezessiven Phase des Busses (High-Pegel) kann der interne Pufferkondensator über den Buswiderstand aufgeladen werden. Die Größe des

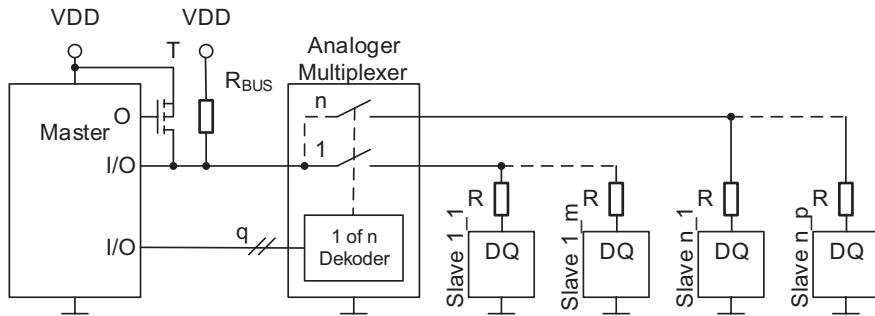
**Abb. 12.1** a minimale 1-wire-Konfiguration, b „parasitäre“ Versorgung eines Slaves mit Versorgungspin, c „parasitäre“ Versorgung eines Slaves ohne Versorgungspin



Buswiderstandes muss an die Energiebedürfnisse der Slaves angepasst werden. Bei der parasitären Versorgung wird für Vorgänge, die mehr Energie fordern, wie das Speichern in einen EEPROM, oder das Konstanthalten der Spannung über längere Zeit (Driftverhalten von Sensoren!) die Versorgungsspannung über einen digitalen Schalter wie in Abb. 12.1b und c kurzfristig auf die Datenleitung geschaltet um den Kondensator frisch aufzuladen. In dieser Zeit kann über den Bus keine Kommunikation stattfinden.

Die Beschaltung der einzelnen Slaves ist sehr einfach, da sie keine zusätzlichen Bauelemente benötigen, nicht einmal einen Blockkondensator am Versorgungspin. Die 1-wire-Knoten können als Stern oder Bus mit bis zu 3 m langen Stichleitungen angeordnet werden [2]. Die Kommunikation mit reduzierter Übertragungsrate mit Slaves, die mit 5 V versorgt werden, funktioniert auch über längere Leitungen (siehe [2]). Das ermöglicht die preiswerte Vernetzung von 1-wire-Sensoren in großen Räumen oder sogar in einem ganzen Haus. Damit die Netzwerkkommunikation störungsfrei funktioniert, muss auf die genaue Gestaltung des Netzwerks geachtet werden. Unterschiedliche Länge der Datenleitungen zu den Slaves führt zu unterschiedlichen Reaktionszeiten der Slaves wie z. B. während der Businitialisierung (siehe Abschn. 12.1.2) oder beim Knotensuchen. Bei der Sterntopologie können zusätzlich Reflexionen am Leitungsende die Signale verfälschen.

Komplexe, gemischte Netzwerke führen oft zu Übertragungsfehlern. In Abb. 12.2 ist eine geschaltete Netzwerktopologie [2] dargestellt, die einen guten Kompromiss zwischen Netzwerkkomplexität und Übertragungssicherheit realisiert. Der Master schaltet durch

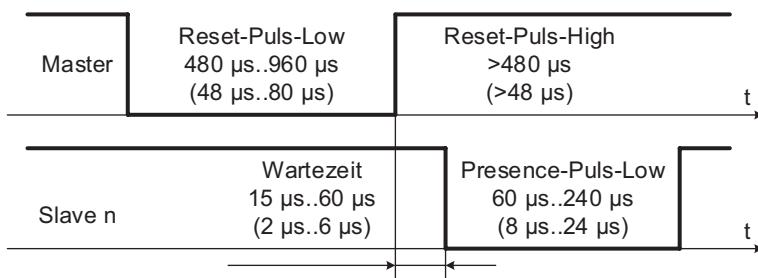


**Abb. 12.2** 1-wire-geschaltete-Netzwerktopologie

den Decoder jeweils einen Netzwerkzweig, der in diesem Fall eine einfache Bustopologie aufweist. Der Master benötigt einen I/O-Anschluss für die Datenkommunikation und  $q = \log_2 n$  Ausgänge für das Selektieren der  $n$ -Zweige. Die Slaves können entweder über VDD, oder „parasitär“ versorgt werden. Die mit den Slaves in Reihe geschaltete Widerstände R werden empfohlen, um den Einfluss der Reflexionen am Leitungsende zu minimieren.

### 12.1.2 Initialisierung des Busses

Vor jeder Kommunikationssitzung muss der Master die Slaves initialisieren, die am Bus angeschlossen sind. Dieser Vorgang ist in Abb. 12.3 dargestellt. Der Master erzeugt einen Reset-Puls-Low für mindestens 480 µs und gibt anschließend den Bus frei. In den nächsten 480 µs wartet der Master auf die Antwort der Slaves. Für einen Mikrocontroller der Familie ATmega kann der Bus freigegeben werden, indem der steuernde Pin von Ausgang auf Eingang geschaltet wird und noch der interne Pull-up-Widerstand wirkt, der durch Setzen des PORT-Registers angeschaltet wird, wenn das entsprechende DDR-Register auf Eingang steht. Dadurch wird das Aufladen der Buskapazität über den Pull-up-Widerstand ermöglicht. Nach der steigenden Flanke des Reset-Puls-High erzeugen die betriebsbereiten Slaves nach einer Wartezeit einen so genannten Presence-Puls-Low und



**Abb. 12.3** 1-wire-Bus-Initialisierung

geben danach ebenfalls die Leitung frei. Unter Berücksichtigung der Geschwindigkeit, mit der die Slaves antworten, und dem Abstand zum Master (Ausbreitungszeit), wird für die Wartezeit eine Zeitspanne von 15 µs...60 µs und für den Presence-Puls-Low eine Zeitspanne von 60 µs...240 µs vorgesehen. Für die Slaves, die eine schnellere Übertragungsrate ermöglichen, gelten die Zeiten aus den Klammern.

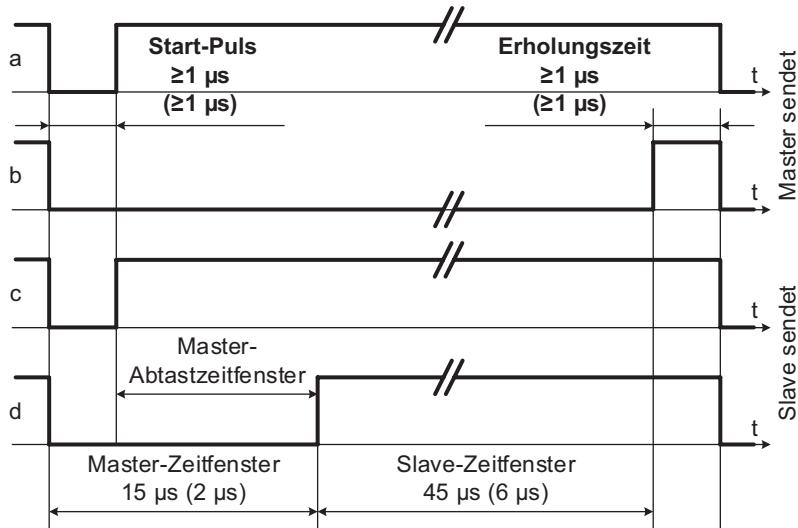
Eine mögliche Initialisierungsroutine kann für die Zeitbestimmung entweder einen Timer oder Wartefunktionen, die auf die Quarzfrequenz des Mikrocontrollers abgestimmt sind, benutzen. Diese Routine soll für die Slaves mit langsamer Datenrate die Initialisierung als erfolglos zurückmelden wenn:

- während des Reset-Puls-High kein Presence-Puls-Low detektiert wurde;
- die Wartezeit vor dem Presence-Puls-Low nicht im Bereich 15 µs...60 µs liegt;
- die Dauer des Presence-Pulses-Low nicht im Bereich 60 µs...240 µs liegt.

### 12.1.3 1-wire-Bitübertragung

Die Übertragung eines Bytes beginnt immer mit dem niederwertigsten Bit zuerst. Das 1-wire-Bus-Protokoll sieht für die Bitübertragung ein synchrones Zeitmultiplexverfahren vor. Die Dauer eines Bits ist für die normale Übertragungsgeschwindigkeit auf 60 µs festgelegt. Im ersten Zeitfenster mit einer Dauer von 15 µs können die Slaves ihr Bit auf den Bus legen, das zweite Zeitfenster mit einer Dauer von 45 µs ist für das vom Master übertragene Bit reserviert. Mit einer Erholungszeit – Wartezeit zwischen zwei Bits – von mindestens 1 µs kann die Übertragungsrate maximal 16,3 kBit/s erreichen. Für die Synchronisation der Übertragung sorgt der Master mit einem Startpuls (Low-Pegel) von mindestens 1 µs. Dieser leitet die Übertragung jedes Bits ein. Nach dem Startpuls gibt der Master normalerweise den Bus frei. Die Signalabläufe der einzelnen Bits sind von deren logischen Werten und der Übertragungsrichtung abhängig, so wie es in Abb. 12.4 zu sehen ist. Die in den Klammern angegebenen Zeiten betreffen die Übertragung mit erhöhter Geschwindigkeit, die im idealen Fall 142 kBit/s erreichen kann. Diese hohe Übertragungsgeschwindigkeit wird nicht von allen Bauteilen unterstützt. Zu beachten sind auch die abweichenden Zeiten, die in den Datenblättern der jeweiligen Bauteile vermerkt sind (z. B. [4]).

Der Master beginnt die Übertragung einer “1“ mit einem Start-Puls, nach dem er die Busleitung wieder frei (rezessiv high) lässt. Über den Pull-up-Widerstand lädt sich die Buskapazität innerhalb des ersten Zeitfensters auf und im zweiten tasten die Slaves das Signal ab. (Abb. 12.4a). Um eine “0“ zu senden, hält der Master nach dem Start-Puls die Busleitung für die gesamte Bitdauer weiterhin auf Low (Abb. 12.4b). Der Master fordert die Slaves Daten zu senden durch so genannte ROM-Befehle (siehe Abschn. 12.1.4.1). Jedes aufgeforderte Bit muss der Master mit einem Start-Puls einleiten. Für die Übertragung einer logischen “1“ greift der Slave nach dem Start-Puls nicht auf die Datenleitung zu, so dass diese gleich im High Pegel verbleibt (Abb. 12.4c). Wenn ein Slave eine “0“ senden will, hält er nach dem Start-Puls das Signal bis zum Ende des ersten



**Abb. 12.4** 1-wire-Bit-Timing. **a** Master sendet eine “1”, **b** Master sendet eine “0”, **c** Slave antwortet mit einer “1”, **d** Slave antwortet mit einer “0”

Zeitfensters auf Low und gibt anschließend die Busleitung frei (Abb. 12.4d). So wie in Abb. 12.4 zu sehen ist, ist die Busleitung überwiegend auf High, was zur Aufladung der internen Kapazität der Slaves führt und die parasitäre Versorgung begünstigt. Man spricht davon, dass das Protokoll nicht gleichspannungsfrei ist, was normalerweise vermieden werden soll, hier aber nützlich ist. Problematisch im Fall einer parasitären Versorgung ist die Übertragung langer Nullfolgen in beiden Richtungen, da diese die internen Kapazitäten entlädt und damit den Slave von der Versorgung abschneidet.

Der Master muss dafür sorgen, dass im gesamten Slave-Zeitfenster der Signalpegel konstant bleibt. Es wird empfohlen, dass der Master das Bussignal im letzten Drittel des Master-Zeitfensters abtastet, um Fehler wegen einer zu hohen Zeitkonstante (Einschwingen) des Busses zu vermeiden.

Jeder I/O-Anschluss eines Mikrocontrollers der Familie ATmega kann für die Ansteuerung eines 1-wire-Busses verwendet werden. Eine mögliche Hardware-Abstraktion für den Pin 2 vom Port B der einen 1-wire-Bus steuern soll, lautet:

#define _1_WIRE_DDR_REG	DDRB
#define _1_WIRE_PORT_REG	PORTB
#define _1_WIRE_PIN_REG	PINB
#define _1_WIRE_BIT	PB2

Mit der vorgestellten Hardware-Abstraktion kann der Master die Ansteuerung des Busses, bzw. Abtastung des Bussignals folgendermaßen realisieren:

```
//Initialisierung des Registers PORT
_1_WIRE_PORT_REG &= ~ (1<<_1_WIRE_BIT);

/*der Anschluss wird auf Ausgang deklariert; mit dem entsprechenden
Bit im Register PORT auf Low, wird der Buspegel auf Low gesetzt, was
einer logischen "0" entspricht*/

_1_WIRE_DDR_REG |= 1<<_1_WIRE_BIT;

/*der Anschluss wird auf Eingang deklariert was ihn in den
Hohenimpedanz Zustand versetzt. Dadurch wird der Bus freigegeben und
die Buskapazität auf ein High-Pegel aufgeladen*/

_1_WIRE_DDR_REG &= ~ (1<<_1_WIRE_BIT);

//mit der folgenden Anweisung wird das Bussignal abgetastet
ucBit=_1_WIRE_PIN_REG & (1<<_1_WIRE_BIT);
```

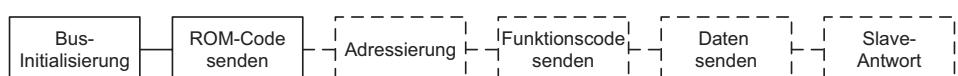
### 12.1.4 Kommunikationssitzung

Die Kommunikation des Masters mit den Slaves erfolgt generell entweder mit sogenannten ROM-Befehlen oder mit Funktionsbefehlen. Die Funktionsbefehle veranlassen den Slave zu einer Aktion oder zum Senden einer Antwortbotschaft. Die ROM-Befehle haben die Aufgabe, die Slaves zu adressieren (es folgt ein Funktionsbefehl) oder die ID der Slaves zu lesen.

Wie in Abb. 12.5 dargestellt, sendet der Master nach der erfolgreichen Businitialisierung (siehe Abschn. 12.1.2) sofort einen ROM-Befehl.

In einem Bus mit mehreren Teilnehmern muss der Master deren IDs kennen, um sie einzeln ansprechen zu können. Diese IDs sind speziell bei der Inbetriebnahme möglicherweise nicht bekannt und müssen dann zunächst in einem Rundruf ermittelt werden.

Ein Beispiel mit in einem Schaltschrank verteilten Temperatursensoren soll die Problematik bei der Inbetriebnahme des Busses verdeutlichen. Die Sensoren werden an verschiedenen Orten im Schaltschrank verbaut und besitzen eindeutige IDs, allerdings ist nicht einfach ersichtlich, welche ID zu welchem Sensor gehört. Da die Temperatur mit einem Ort in Verbindung gebracht werden muss, ist es sinnvoll, die Sensoren nach und nach einzubauen und dem Master anzulernen.



**Abb. 12.5** 1-wire-Kommunikationssitzung

**Tab. 12.1** Bitermittlung beim Lesen des Identifikationscodes

1. Abtastung	2. Abtastung	
0	0	Weil einige Teilnehmer an dieser Stelle eine "0" und die anderen eine "1" besitzen, werden beide Abtastwerte als "0" ausgewertet und somit eine Buskollision erkannt. Bei der ersten Kollision für diese Bitstelle speichert der Master für den aktuellen Teilnehmer an die aktuelle Bitposition eine "0" und sendet an die Teilnehmer zur Bestätigung eine "0". Die Teilnehmer, deren Bit bestätigt wurde, werden weiter senden, die anderen hören bis zur nächsten Businitialisierung auf. Der Master merkt sich die Bitstelle, bei der die Kollision auftrat. Wenn alle IDs, die an dieser Stelle eine "0" haben, erkannt wurden, speichert der Master für den aktuellen Teilnehmer eine "1" und sendet zur Bestätigung eine "1", um die anderen erkennen zu können
0	1	Alle Teilnehmer haben das erste Mal eine "0" gesendet und das zweite Mal eine "1"; für den aktuellen Teilnehmer wird an die aktuelle Bitposition eine "0" gespeichert und der Master sendet zur Bestätigung eine "0"
1	0	Alle Teilnehmer haben das erste Mal eine "1" gesendet und das zweite Mal eine "0"; für den aktuellen Teilnehmer wird an die aktuellen Bitposition eine 1 gespeichert und der Master sendet zur Bestätigung eine "1"
1	1	Fehler; der gesamte Vorgang wird abgebrochen und wiederholt

#### 12.1.4.1 ROM-Befehle

Man unterscheidet zwischen allgemeinen und spezifischen ROM-Befehlen. Die allgemeinen Befehle, die im Folgenden kurz beschrieben werden, sind in allen 1-wire-Slaves implementiert, während die spezifischen nur in gewissen Familien von Bauteilen.

**Search ROM (Code 0xF0)** Mit diesem ROM-Befehl werden die Teilnehmer aufgefordert ihre Identifikationscodes zu senden. Nach der erfolgreichen Businitialisierung und dem Senden des Search-ROM-Codes leitet der Master das Senden jedes Bits ein und alle Teilnehmer senden simultan, angefangen mit dem niederwertigsten Bit. Beim gleichzeitigen Senden von rezessiven ("1") und dominanten ("0") Bits, wegen der Wired-AND-Beschaltung der Slaves entstehen Buskollisionen. Anders als CAN, setzt das 1-wire-Protokoll nicht auf Kollisionsvermeidung, sondern auf Kollisionserkennung. Das Auflösen dieser Kollisionen findet in einem komplexen, wiederholenden Verfahren statt, das in [1] genau beschrieben ist. Um die Übertragungskollisionen zu erkennen, fordert der Master für jeden Teilnehmer zweimal das Senden jedes der 64 Bits an. Das erste Mal senden die Teilnehmer das reale und das zweite Mal das invertierte Bit. Das Ergebnis dieses doppelten Ergebnisses ist in der Tab. 12.1 aufgelistet.

Der Vorgang wird fortgesetzt bis alle 64 Bits gelesen sind und wird wiederholt (inklusive Businitialisierung und Senden des Search-ROM-Codes) bis alle Identifikationscodes ermittelt sind.

**Read ROM (Code 0x33)** Dieser ROM-Befehl fordert den einzigen am Bus angeschlossenen Slave auf, seinen Identifikationscode zu senden. Die Bits werden in einem normalen Leseverfahren ermittelt.

**Match ROM (Code 0x55)** Diesem ROM-Befehl folgen das Senden des Identifikationscodes eines Teilnehmers und ein Funktionsbefehl. Auf diesen Funktionsbefehl reagiert nur der adressierte Teilnehmer.

**Skip ROM (Code 0xCC)** In einem Netzwerk mit einem einzigen Slave kann mit diesem ROM-Befehl die Adressierungsphase übersprungen werden. In einem Netzwerk mit gleichen Slaves – zum Beispiel Temperatursensoren – teilt mit diesem ROM-Befehl der Master den Slaves mit, dass ein Broadcast-Funktionsbefehl folgt. Dieser an alle Slaves adressierte Befehl darf kein Lesebefehl sein, um Buskollisionen zu vermeiden.

Zu den ROM-Befehlen zählt der Hersteller weitere Befehle, die aber nicht zwingend in allen Bauteilen implementiert sind:

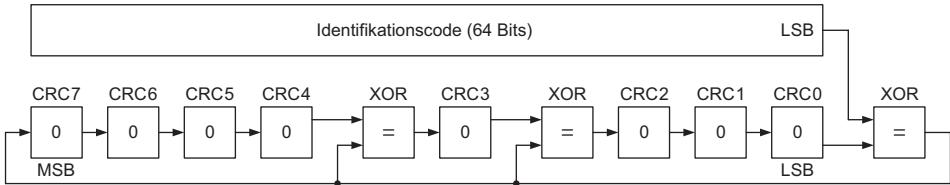
- **Alarm Search** bewirkt die Suche in einem 1-wire-Bus nach den Temperatursensoren vom Typ DS18S20, die einen Temperaturalarm ausgelöst haben;
- **Overdrive Skip ROM** und **Overdrive Match ROM** versetzen alle Busteilnehmer in diesen Modus, die in der Lage sind, mit erhöhter Bitrate zu kommunizieren. Nach diesem Befehl wird die gesamte Buskommunikation mit erhöhter Übertragungsrate stattfinden und die anderen Busteilnehmer schalten in den inaktiven Modus um. Dieser Zustand wird mit einem Reset-Puls beendet.

#### 12.1.4.2 Adressierung

Soll ein Slave eine Funktion ausführen, muss der Master diesen über dessen ID adressieren. Die Adressierung ist nur nach dem Match ROM-Befehl notwendig. Der Identifikationscode aller 1-wire-Slaves ist 64 Bits (acht Bytes) groß und hat, die in der Tab. 12.2 dargestellte Konfiguration. Der Familiencode hat die niederwertigste Stellung und identifiziert die Bauteile mit der gleichen Funktion. Die Temperatursensoren der Serie DS18B20 z. B. haben den Familiencode 0x28. Die Seriennummer identifiziert innerhalb einer Bauteilfamilie einmalig die 1-wire-Slaves. Eine CRC-Prüfsumme über diese sieben Bytes wird berechnet und alle zusammen werden während des Herstellungsprozesses in das Bauteil eingebrannt. Der Master muss beim Einschalten die Identifikationscodes aller Busteilnehmer ermitteln, um sie adressieren zu können. Die Prüfsumme wird nach dem Empfang neu berechnet und mit der empfangenen verglichen, um Übertragungsfehler zu erkennen.

**Tab. 12.2** Zusammensetzung des Identifikationscodes

CRC-Prüfsumme	Seriennummer	Familiencode
1 Byte (8 Bits)	6 Bytes (48 Bits)	1 Byte (8 Bits)



**Abb. 12.6** CRC-Berechnung nach [3]

Die Prüfsumme wird berechnet als Rest der Polynomdivision  $v(x)$ :  $g(x)$  wobei  $v(x)$  der Gl. 12.1 entspricht.

$$v(x) = a_n x^{n-1} + a_{n-1} x^{n-2} + \dots + a_2 x + a_1 \quad (12.1)$$

mit  $n=64$ ,  $a_0$  das niederwertigste Bit des FamilienCodes und  $a_n$  das hochwertigste Bit der empfangenen Prüfsumme. In Abschn. 6.1.2.5 ist die Bildung von CRC-Prüfsummen genauer beschrieben. Die Gleichung bildet die Polynomdarstellung des Identifikationscodes ab. Das Polynom  $g(x)$  wird als Generatorpolynom genannt und ist für dieses Verfahren der Form  $x^8+x^5+x^4+1$ . Im Fall einer fehlerfreien Übertragung ist der Rest der Polynomdivision gleich Null. Die in Abb. 12.6 abgebildete Schaltung implementiert den euklidischen Divisionsalgorithmus. Das CRC-Byte wird vor der Berechnung mit 0x00 initialisiert.

Die folgende Funktion berechnet die Prüfsumme eines Byte-Arrays nach dem Verfahren der Firma Maxim und gibt diesen Wert an die aufrufende Stelle zurück.

```
uint8_t _1_wire_CRC_Maxim(uint8_t *ucbytefield, uint8_t ucnumber)
{
    uint8_t ucCRC=0, ucMask, ucPolynom=0x18, ucTest;

    for(uint8_t j=0; j<ucnumber; j++) //Bytezahl des Arrays
    {
        ucMask=0x01; //Bitzeiger, beginnt mit dem LSB
        for(uint8_t i=0; i<8; i++)
        {
            //es wird geprüft ob "0" oder "1" an MSB
            //geschoben wird
            if(ucbytefield[j] & ucMask) ucTest=(ucCRC &
            0x01) ^ 0x01;
            else ucTest=(ucCRC & 0x01) ^ 0x00;
            if(ucTest)
            {
                ucCRC=ucCRC ^ ucPolynom;
                ucCRC=(ucCRC>>1) | 0x80;
            }
            else ucCRC=ucCRC>>1;
        }
    }
}
```

```

        ucMask=ucMask << 1;
    }
}

return ucCRC;
}

```

### 12.1.4.3 Funktionsbefehle

Einen Funktionsbefehl sendet der Master nach dem Skip ROM-Befehl oder nach Match ROM gefolgt von der Adresse des gewünschten Slaves. Die erste Gruppe von Befehlen löst in den Slaves eine interne Aktion aus, wie beispielsweise den Start einer neuen Messung oder den Datentransfer aus einem flüchtigen in einen nicht flüchtigen Speicherbereich. Nach der Übermittlung solcher Befehle beendet der Master die Kommunikationssitzung. Wenn diese Befehle alle Busteilnehmer gleicher Art betreffen, werden sie als Broadcast-Botschaft nach dem Skip ROM-Befehl gesendet. Mit einer zweiten Gruppe von Befehlen kann der Master entweder Register oder Speicherbereiche der Slaves neu beschreiben. Diesen Datensatz sendet der Master direkt nach dem Funktionsbefehl. Auch in diesem Fall sind Broadcast-Botschaften erlaubt, wenn die Slaves auf die Botschaft nicht antworten müssen. Die dritte Gruppe von Befehlen fordert eine Antwort eines einzelnen Busteilnehmers und in diesem Fall muss der Slave mit seinem Identifikationscode adressiert werden. Broadcast-Nachrichten sind wegen Kollisionsgefahr in diesem Fall nicht erlaubt.

### 12.1.5 Softwareaufbau der 1-wire-Buskommunikation

Betrachtet wird der allgemeine Fall eines ATmega-Mikrocontrollers der als Busmaster mehrere 1-wire-Busse ansteuert. An jeden Bus können unterschiedliche Slaves angeschlossen werden. Für die Buskommunikation wird ein allgemeines Softwaremodul erstellt, das die grundlegenden Funktionen für die Bitgenerierung und die generellen ROM-Funktionen zur Verfügung stellt. In der Header-Datei des Softwaremoduls `_1_wire.h` wird eine Softwarestruktur definiert, die den Busanschluss abstrahiert.

```

typedef struct{
    volatile uint8_t* DQ_DDR_REG;
    volatile uint8_t* DQ_PORT_REG;
    volatile uint8_t* DQ_PIN_REG;
    uint8_t DQ_pin;
} t1wireHandle;

```

Um das Modul unabhängig von der Hardware zu gestalten, werden in der Hauptdatei die Mikrocontrolleranschlüsse für jeden Bus definiert. Ein Beispiel eines Busanschlusses an den Pin 2 des Ports B ist im folgenden Programmabschnitt dargestellt.

```
t1wireHandle _1wire_Bus_1={{/*DQ_DDR_REG*/          &DDR2,
                           /*DQ_PORT_REG*/        &PORTB,
                           /*DQ_PIN_REG*/         &PINB,
                           /*DQ_pin*/             PB2}};
```

Die Funktionen des Moduls werden hierarchisch aufgebaut und können sowohl aus den Software-Modulen der einzelnen Slaves, als auch aus der Hauptdatei aufgerufen werden. Durch diesen Aufbau ist es leicht, die spezifischen Funktionen der einzelnen Slaves zu programmieren. Der Master steuert folgendermaßen die Busleitung auf High, bzw. auf Low:

```
void _1wire_Set_High(t1wireHandle sdevice_pins)
{
    *sdevice_pins.DQ_DDR_REG &= ~(1<<sdevice_pins.DQ_pin);
}
void _1wire_Set_Low(t1wireHandle sdevice_pins)
{
    *sdevice_pins.DQ_DDR_REG |= 1<<sdevice_pins.DQ_pin;
}
```

Die Funktionen, die dem Master die Generierung der einzelnen Bits dienen: `_1wire_Write_Zero` oder `_1wire_Write_One`, sind aufgrund des, in der Abb. 12.4 dargestellten Timing für die Kommunikation mit langsamem Slaves aufgebaut. Mit diesen primitiven Funktionen werden komplexere Funktionen – wie z. B. Übertragung eines Bytes – zusammengestellt oder ganze Abläufe programmiert.

```
void _1wire_Write_Zero(t1wireHandle sdevice_pins)
{
    /*1-wire-Datenleitung wird auf Low gesetzt, die Übertragung
    eines Bits wird initiiert*/
    _1wire_Set_Low(sdevice_pins);
    _1wire_Delay_Start(ucDelayTime[TIME_60_MICROSEC]); //der Timer
    wird gestartet
    while(_1wire_Delay_Get_State() == _1WIRE_DELAY_RUNNING);
    _1wire_Set_High(sdevice_pins); //der µC gibt die Datenleitung frei
}

void _1wire_Write_One(t1wireHandle sdevice_pins)
{
    /*1-wire-Datenleitung wird auf Low gesetzt, die Übertragung eines
    Bits wird initiiert*/
    _1wire_Set_Low(sdevice_pins);
    _1wire_Set_High(sdevice_pins); //der µC gibt die Datenleitung frei
    _1wire_Delay_Start(ucDelayTime[TIME_60_MICROSEC]); //der Timer
    wird gestartet
    while(_1wire_Delay_Get_State() == _1WIRE_DELAY_RUNNING);
}
```

## 12.1.6 Ansteuerung eines 1-wire-Temperatursensors vom Typ DS18B20

Der Baustein DS18S20 [6] gehört zu einer Familie von Temperatursensoren mit einstellbarer Auflösung. Der große Adressbereich der 1-wire-Bausteine ermöglicht ihren Einsatz für die Temperaturüberwachung bzw. Temperaturregelung in Räumen mit vielen Messpunkten. Die Messgenauigkeit über den gesamten Messbereich  $-55^{\circ}\text{C} \dots +125^{\circ}\text{C}$  beträgt  $\pm 2^{\circ}\text{C}$ , über den Bereich  $-10^{\circ}\text{C} \dots +85^{\circ}\text{C}$  sogar  $\pm 0,5^{\circ}\text{C}$ . Die einstellbare Bitauflösung von 9 bis 12 Bit ermöglicht eine Temperaturauflösung von  $0,5^{\circ}\text{C}$  bis  $0,0625^{\circ}\text{C}$ . Die Erhöhung der Auflösung um ein Bit verdoppelt die Messdauer, die bei 12 Bit 750 ms erreicht. Der Baustein besitzt eine Alarmfunktion, die beim Überschreiten, bzw. Unterschreiten von eingestellten Temperaturgrenzen aktiviert wird.

### 12.1.6.1 Speicherorganisation

Der Baustein besitzt drei Bytes nichtflüchtiger Speicher, in dem die Temperaturgrenzen und Bitauflösung dauerhaft gespeichert werden können. Zusätzlich kann der Busmaster auf einen neun Byte großen RAM (scratchpad) mit der in Tab. 12.3 dargestellte Zusammensetzung zugreifen.

Die Bytes 2, 3 und 4 werden nach dem Hochfahren des Bausteins mit den im EEPROM gespeicherten Werten initialisiert. Am Ende einer Temperaturmessung wird der gemessene Wert in die Bytes 0 und 1 gespeichert, die Prüfsumme neu berechnet und in das Byte 8 gespeichert.

### 12.1.6.2 Temperaturcodierung

Der Sensor speichert die Temperaturwerte im Zweierkomplement (siehe Abb. 12.7) in die Bytes 0 und 1, die von dem Master nur gelesen werden können. In Abb. 12.7 sind die Bits 11 bis 15 Null für positive und Eins für negative Temperaturwerte. Die mit x bezeichneten Bits sind für die entsprechenden Auflösungen undefiniert. Die Bytes 2 und 3 speichern im Zweierkomplement als Ganzzahl die Temperaturgrenzen. Wenn der gemessene Temperaturwert diese Grenzen erreicht oder überschreitet, wird intern

**Tab. 12.3** Zusammensetzung des RAMs bei DS18B20

Byte 0	Temperaturwert LSB
Byte 1	Temperaturwert MSB
Byte 2	Obere Temperaturgrenze
Byte 3	Untere Temperaturgrenze
Byte 4	Konfiguration-Register
Byte 5	Reserviert (0xFF)
Byte 6	Reserviert
Byte 7	Reserviert (0x10)
Byte 8	CRC

Kofiguration-Register		Temperatur-Register																
Bit 6 R1	Bit 5 R0	Auflösung [Bit]	MSB (Byte 1)								LSB (Byte 0)							
		Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	
1	1	12	±	±	±	±	±	$2^6$	$2^5$	$2^4$	,							°C
1	0	11	±	±	±	±	±	$2^6$	$2^5$	$2^4$	,							°C
0	1	10	±	±	±	±	±	$2^6$	$2^5$	$2^4$	,							°C
0	0	9	±	±	±	±	±	$2^6$	$2^5$	$2^4$	,							°C

**Abb. 12.7** DS18B20-Codierung der Temperaturwerte

ein Flag gesetzt, das zurückgesetzt wird, sobald die Temperatur wieder innerhalb des zulässigen Bereiches verläuft. Die Bitauflösung wird über die Bits 5 und 6 des Konfiguration-Registers (Byte 4) entsprechend der Abb. 12.7 bestimmt. Die weiteren Bits dieses Registers haben feste Werte, die von dem Master nicht geändert werden können.

### 12.1.6.3 Funktionsbefehle DS18B20

**Start Temperaturmessung (0x44)** Der Temperatursensor, der diesen Befehl empfängt, startet eine neue Messung.

**RAM schreiben (0x4E)** Mit diesem Befehl, gefolgt von drei Datenbytes, kann der Busmaster die Temperaturgrenzen (Byte 2 und 3) und die Bitauflösung (Byte 4) eines Slaves ändern.

**RAM lesen (0xBE)** Der gesamte RAM-Bereich (Byte 0 bis 8) eines Temperatursensors kann in einer Kommunikationssitzung, beginnend mit Byte 0, gelesen werden. Nach dem Lesen kann die Prüfsumme über die ersten acht Byte – z. B. mit der vorgestellten Funktion `_1_wire_CRC_Maxim` – gebildet werden und danach mit dem Byte 8 verglichen, um eventuelle Übertragungsfehler zu detektieren. Der Master kann die Übertragung mit dem Senden eines Reset-Pulses beenden, wenn nur ein Teil der Daten gewünscht ist. Mit der Funktion `DS18B20_Read_Scratchpad` kann der gesamte RAM-Bereich gelesen werden.

```

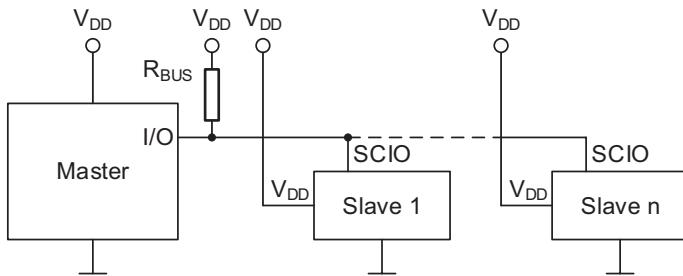
uint8_t DS18B20_Read_Scratchpad(t1wireHandle sdevice_pins,
                                uint8_t ucrom_command,
                                uint8_t *ucbytefield,
                                uint8_t *ucromfield)
{
    uint8_t ucMask;
    //Initialisierung
    if(_1wire_Init_Comm(sdevice_pins) == INIT_NOK) return INIT_NOK;
    _1wire_Write_Byte(sdevice_pins,ucrom_command); //der ROM-
    Befehl wird übertragen
    if(ucrom_command==MATCH_ROM)
    {
        for(uint8_t i=0; i<8; i++)
        {
            _1wire_Write_Byte(sdevice_pins,ucromfield[i]);
        }
    }
    _1wire_Write_Byte(sdevice_pins,READ_SCRATCHPAD); //der
    Befehlscode wird übertragen
    for(uint8_t i=0; i<9; i++) //es werden alle neun Bytes
    gelesen
    {
        ucMask=0;
        ucbytefield[i]=0;
        for(uint8_t j=0; j<8; j++) //die acht Bits eines
        Bytes werden gelesen
        {
            ucbytefield[i] += _1wire_Read_Bit(sdevice_
            pins) << ucMask;
            ucMask++;
        }
    }
    return INIT_OK;
}

```

Wenn der Identifikationscode eines Temperatursensors im Array ucROM gespeichert ist, kann sein RAM-Bereich in den Vektor ucScratchpad mit dem Aufruf gelesen werden: DS18B20\_Read\_Scratchpad(\_1wire\_Bus\_1,MATCH\_ROM, ucScrachpad, ucROMCode).

**Daten im EEPROM sichern (0x48)** – Dieser Befehl bewirkt das dauerhafte Speichern der Bytes 2, 3 und 4 vom RAM-Bereich in den EEPROM.

**Daten aus EEPROM abrufen (0xB8)** – die Bytes 2, 3 und 4 aus dem RAM-Bereich werden mit den im EEPROM dauerhaft gespeicherte Werte überschrieben.



**Abb. 12.8** UNI/O-Bus-Konfiguration

**Abfrage Versorgungsart (0xB4)** – Nach dem Empfang dieses Befehls schalten die Slaves DS18B20, die parasitär versorgt werden, als Antwort die Datenleitung auf Low.

## 12.2 UNI/O® -Bus

UNI/O® ist ein proprietäres Bus-Protokoll [7], das von der Firma Microchip Technology entwickelt wurde, um ähnlich wie bei 1-wire die bidirektionale Kommunikation zwischen einem Master und mehreren Slaves über eine einzige Datenleitung (SCIO) zu realisieren. Die Bus-Spezifikation erwähnt als mögliche Slaves: EEPROMs, Temperatursensoren, A/D-Wandler. Der Master initiiert die Kommunikation und baut sie mit den einzelnen Slaves über eine fest gespeicherte oder programmierbare, acht Bit oder zwölf Bit große Adresse auf. Slaves mit 8-Bit- und 12-Bit-Adresse können in einem Bus koexistieren ohne Kollisionen zu verursachen. Die Kommunikation kann laut der Spezifikation mit einer Übertragungsrate zwischen 10 kbps<sup>1</sup> und 100 kbps stattfinden. Die Versorgungsspannung und die Herstellungsprozesse der Slaves werden nicht spezifiziert, die gesamte Buskapazität ist aber auf 100 pF begrenzt.

### 12.2.1 Netzwerktopologie

Um die Vorteile der 1-Leitung-Übertragung zu gewährleisten, werden die UNI/O-Slaves über drei Anschlüsse wie im Abb. 12.8 dargestellt, angeschlossen. Während die Slaves 1 bis n die Leitung  $V_{DD}$  für die eigene Versorgung benutzen, wird der Slave in Abb. 12.11 „parasitär“ über die Datenleitung versorgt. Dafür werden zusätzlich externe Bauteile: eine schnelle Gleichrichterdiode mit niedrigem Spannungsabfall und einen Pufferkondensator benötigt.

<sup>1</sup> kbps – kilobits per second (Kilobit pro Sekunde).

### 12.2.2 Bitcodierung

Ähnlich wie in Kap. 6 beschrieben, werden die einzelnen Bits mit einem unipolaren Manchester-Code codiert, wie es auch in Abb. 6.8 zu sehen ist. In der ersten Bithälfte ist die logische “0“ auf High-Pegel, in der zweiten auf Low-Pegel (das erste Bit der Präambel). Die logische “1“ dagegen steht in der ersten Bithälfte auf Low-Pegel und in der zweiten auf High-Pegel (das zweite Bit der Präambel). Letztlich findet also zur Codierung einer “0“ in der Bitmitte ein 1-0-Übergang statt und umgekehrt bei der “1“. Der resultierende Datenstrom begünstigt wegen der Pegelumschaltung in der Mitte eines jeden Bits die Takt synchronisation.

### 12.2.3 UNI/O-Pulsrahmen

Ein Pulsrahmen besteht aus einem Byte (acht Bit), der Bestätigung des Masters und der Antwort des Slaves. Das Byte wird vom Master oder dem adressierten Slave gesendet, beginnend mit dem höherwertigen Bit (MSB). Anschließend sendet der Master ein Bestätigungsbit und wartet auf die Slave-Antwort. Mit einer logischen “1“ (MAK=Master Acknowledge) signalisiert der Master die Fortsetzung der Kommunikation, mit einer logischen “0“ (NoMAK=Not Master Acknowledge) deren Beendung. Mit der Erkennung der eigenen Adresse antwortet der Slave nach jeder Bestätigung (MAK oder NoMAK) des Masters mit einer logischen “1“ (SAK=Slave Acknowledge). Die Slaves, die am Bus angeschlossen sind, antworten vor der erfolgreichen Adressierung mit NoSAK (Not Slave Acknowledge). Dieses Signal mit der Dauer eines Bits unterscheidet sich in seiner Zusammensetzung von SAK. Mit NoSAK antwortet auch ein bereits adressierter Slave im Fall eines Übertragungsfehlers. Die doppelte Bestätigung vom Master und vom Slave verringert die Nettodatenrate, erhöht aber die Sicherheit der Kommunikation.

### 12.2.4 Kommunikationssitzung

Nach der POR<sup>2</sup>-Phase oder nach einem Fehler schalten die Slaves in den sogenannten idle-Modus in dem sie keine digitalen Signale berücksichtigen und warten darauf, wieder in den Standby-Modus versetzt zu werden. Die Slaves schalten in den Standby-Modus, wenn die Datenleitung länger als 600 µs auf High gehalten wird. In diesem Modus ist der Energieverbrauch gering und die Slaves sind bereit, digitale Signale zu verarbeiten. Es wird empfohlen, einen Pull-up-Widerstand für die Busleitung zu benutzen, um den

---

<sup>2</sup>POR – Power On Reset; Teilschaltung eines komplexen Bausteins die dafür sorgt, dass nach dem Einschalten oder einem Reset ein definierter Zustand erreicht wird.

Standby-Modus auch in einer Zeit zu gewährleisten, in der der Master keine Kontrolle über den Bus hat.

### 12.2.4.1 Initialisierung der Kommunikation

Eine UNI/O-Kommunikation wird vom Master mit einem Standby-Puls (Abb. 12.9) initiiert, oder auch vorzeitig beendet. Die nächste Kommunikationsphase wird mit einem Low-Puls mit einer Dauer von mindestens 10 µs eingeleitet. Das folgende Präambel-Byte (0x01010101) dient zur Synchronisation der Bittakte aller Busteilnehmer. Der Master bestätigt die Präambel mit MAK und alle Slaves antworten mit NoSAK. Die Bitrate muss im genannten Bereich liegen und während einer Kommunikationssitzung sich nicht um mehr als  $\pm 5\%$  ändern. Bei der Übertragung eines gesamten Bytes sind Abweichungen von  $\pm 7,5\%$  von der nominalen Bitrate zulässig und die Flanke eines Bits kann von der Bitmitte um  $\pm 8\%$  von der Bitdauer abweichen. Diese Frequenz- und Zeiteinschränkungen können in der Software mit Hilfe eines Timers oder einer Warteschleife, deren Durchlauf nahezu konstant bleibt, erfüllt werden.

### 12.2.4.2 Adressierung

In Abb. 12.9 a und c sendet der Master die 8-Bit-Adresse 0xA0 und bestätigt mit MAK. Ist am Bus ein EEPROM aus der Familie 11XXYY0 (siehe [8]) angeschlossen, so wird er mit SAK antworten. Allgemein stellt das höherwertige Nibble der 8-Bit-Adresse den Familiencode des Bausteins dar. In der Busspezifikation [7] sind die Kombinationen “0000“ und “1111“ reserviert, die letzte wird für das Bilden der zwei Bytes für die

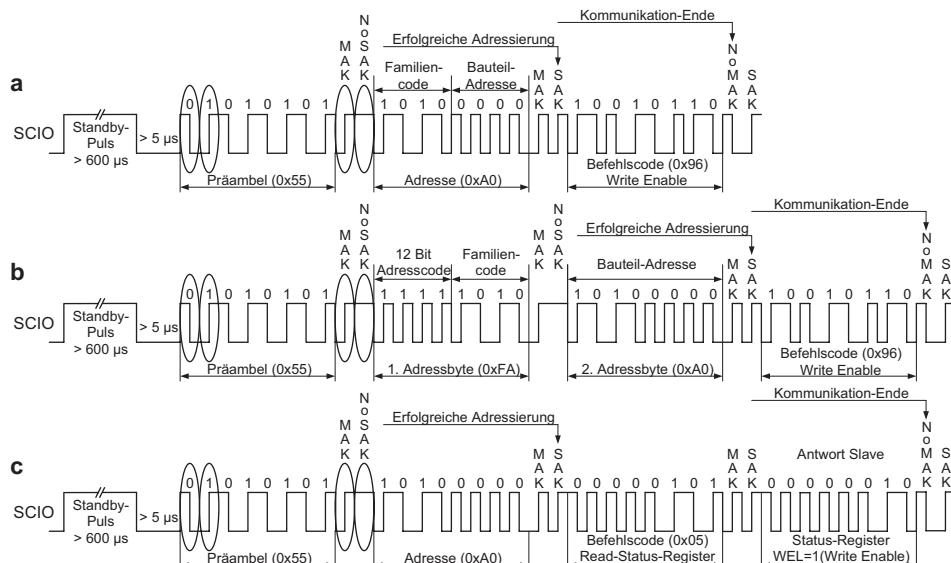


Abb. 12.9 UNI/O-Kommunikationsstruktur

12-Bit-Adressen verwendet. Der 12-Bit-Adresse wird die Kombination “1111“ vorangestellt. In der Abb. 12.9 b ist der zeitliche Ablauf einer Kommunikationssitzung mit einem hypothetischen Baustein, dessen 12-Bit-Adresse 0xAA0 lautet, dargestellt. Die 12-Bit-Addressierung verringert die Nettodatenrate, bietet aber einen größeren Adressraum für Bauteile einer Familie an, die am gleichen Bus angeschlossen sind. Auf das höherwertige Byte einer solchen Adresse antworten die Slaves, die mit einer 8-Bit-Adresse ausgestattet sind, mit NoSAK, da keine gültige Adresse erkannt wird. Somit können Busteilnehmer mit 8- und 12-Bit-Adressen kollisionsfrei am gleichen Bus betrieben werden.

Die Eigenschaft des Protokolls, dass die Slaves mit NoSAK auf fremde Adressen antworten, kann genutzt werden, um die noch unbekannten Adressen im Bus zu detektieren. Der Busmaster muss für jede getestete Adresse eine neue Kommunikation wie oben beschrieben starten und sie nach der Antwort der Slaves auf die gesendete Adresse mit einem Standby-Puls beenden. Wenn die Adresse abgelehnt wurde, muss der Vorgang wiederholt werden. Bei der Bestätigung der Adresse wird der Vorgang nur in dem Fall wiederholt, dass weitere Slaves mit unbekannten Adressen am Bus angeschlossen sind.

#### 12.2.4.3 Funktionsbefehle

Der Master steuert die Slaves über codierte Funktionsbefehle, die gleich nach der Adresse gesendet werden. Man unterscheidet zwischen:

- Befehle, die eine interne Aktion eines Slaves mit zeitlichem Umfang auslösen (Abb. 12.9a und b). Der Master bestätigt einen solchen Befehl mit NoMAK, um dem Slave das Ende der Kommunikation zu signalisieren und wartet, dass die Aktion (Messung oder Speichervorgang) beendet wird.
- Schreib/Lese-Befehle, die auf eine implizite Adresse (Registeradresse oder Adresszähler) verweisen. Der Befehl wird von weiteren Bytes gefolgt, die vom Master oder vom Slave gesendet werden. In Abb. 12.9 c sendet der Master an den EEPROM den Befehlscode 0x05 (Lesebefehl des Statusregisters) und empfängt nach der SAK-Antwort des Slaves den Inhalt des Statusregisters.
- Schreib/Lese-Befehle, die auf eine explizite Adresse verweisen. Der Busmaster sendet nach dem Befehlscode die vollständige Adresse einer Speicherzelle und abhängig vom Befehl werden weitere Bytes vom Master oder vom Slave gesendet.

Das UNI/O-Protokoll sieht kein Broadcasting vor.

#### 12.2.5 Softwareaufbau einer UNI/O-Buskommunikation

Um einen wie in Abb. 12.8 dargestellten Bus zu betreiben, verfügt der Master idealerweise über einen I/O-Pin. Die Mikrocontroller der Familie ATmega besitzen solche Pins,

die den Anforderungen entsprechen und können benutzt werden, um mehrere solche Busse zu betreiben. Im Allgemeinen kann der Busanschluss in einem Softwaremodul UNI\_O folgendermaßen abstrahiert werden:

```
typedef struct{
    volatile uint8_t* SCIO_DDR_REG;
    volatile uint8_t* SCIO_PORT_REG;
    volatile uint8_t* SCIO_PIN_REG;
    uint8_t SCIO_pin;
} tuni_oHandle;
```

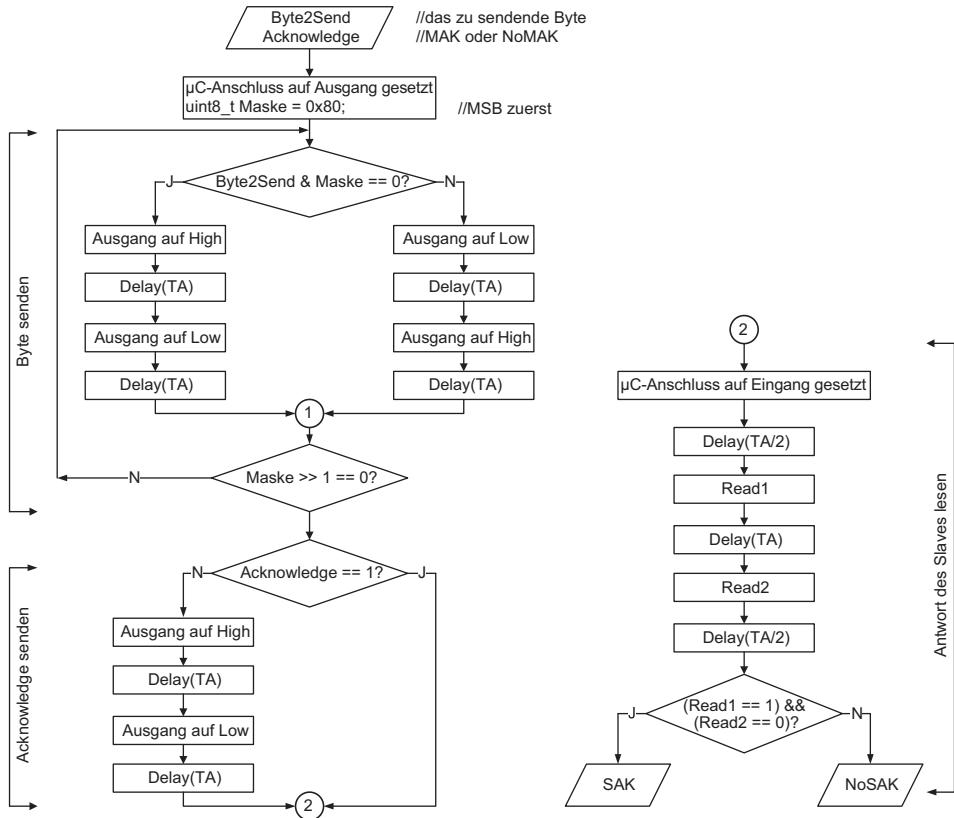
Man merkt die Ähnlichkeit mit der Softwarestruktur eines 1-wire-Busses. Auf ähnliche Weise wird auch in diesem Fall der Mikrocontrolleranschluss in der main-Datei definiert.

Als Bausteine des Softwaremoduls könnten folgende Funktionen dienen:

- Standby-Puls senden;
- Senden eines Bytes
- Empfang eines Bytes.

Auf eine weitere Gliederung der Funktionen auf Bit-Ebene (wie bei 1-wire) wurde verzichtet, um die Zeiteinschränkungen des UNI/O-Busses einzuhalten. Um die nötigen Zeiten für die Kommunikation zu gestalten, wird eine Delay-Funktion benutzt, die in Assembler programmiert wurde. Diese Funktion ermöglicht Übertragungsraten bis ca. 16 kBit/s. Um höhere Übertragungsraten zu erreichen, müssten die Funktionen als Zustandsautomaten gestaltet werden, die über einen Timerinterrupt angesteuert werden.

Die Funktion UNI\_O\_Write\_Byte wird benutzt, um das Präambel-Byte, die Befehlscodes sowie die Datenbytes zu senden. Der Funktion, deren Flussdiagramm in Abb. 12.10 dargestellt ist, werden beim Aufruf das zu übertragene Byte, ucByte2Send sowie die Masterbestätigung (MAK oder NoMAK) als Parameter übergeben. Der Mikrocontroller setzt am Anfang der Funktion den Busanschluss auf Ausgang und beginnt das Senden des Bytes mit dem höherwertigen Bit zuerst. Die Generierung eines Bits wird in Abhängigkeit der Bitwichtigkeit mit dem Setzen des Ausgangs auf High oder auf Low eingeleitet. Der Ausgang bleibt für die Dauer eines Halbbits (TA) in diesem Zustand. Danach wird der Ausgang umgeschaltet und für TA Mikrosekunden gewartet. Nach dem Senden der acht Bits und der Bestätigung wird der Busanschluss auf Eingang geschaltet, um die Antwort des adressierten Slaves zu lesen. Diese Antwort wird in zwei Schritten durch das Lesen in der Mitte der Halbbits ermittelt, daher die Verzögerung um  $TA/2$  Mikrosekunden am Anfang und Ende des Lesens. Diese Antwort wird von der Funktion zurückgegeben, um die weiteren Übertragungsschritte bestimmen zu können. Der erfolgreiche Byteempfang wird vom Slave, angefangen mit dem Erkennen der eigenen Adresse mit SAK beantwortet.



**Abb. 12.10** Byte-Sendefunktion des UNI/O-Busses

Eine Beispielfunktion für das Lesen eines Bytes stellt die Funktion `UNI_O_Read_Byte` dar. Die einzelnen Bits werden wie die Slave-Antwort im vorigen Beispiel schrittweise ermittelt und zu einem Byte zusammengestellt. Die Funktion speichert das gelesene Byte an die gewünschte Adresse (`*ucbyte2read`), sendet die Masterbestätigung und empfängt die Slave-Antwort. Das empfangene Byte wird von einer übergeordneten Funktion verarbeitet nur im Fall einer SAK-Antwort, ansonsten wird das Byte verworfen und die Kommunikation mit einem Standby-Puls beendet.

```

uint8_t UNI_O_Read_Byte(tuni_oHandle sdevice_pins,
                        uint8_t *ucbyte2read,
                        uint8_t uc_ack)
{
    uint8_t ucByte2Read=0, ucBit2Read=0;
    uint8_t uc_Ack=0;
    //die Datenleitung wird auf Eingang gesetzt
  
```

```
*sdevice_pins.SCIO_DDR_REG &=~ (1<<sdevice_pins.SCIO_pin);
Delay(TA_HALF); //Wartezeit um die Mitte der Bithälfte zu
ermitteln

for(uint8_t i=0; i<7; i++) .
{
    //Lesen eines Bits (1...7)
    if(*sdevice_pins.SCIO_PIN_REG & (1<<sdevice_pins.
    SCIO_pin)) ucBit2Read=2;
    Delay(TA);
    if(*sdevice_pins.SCIO_PIN_REG & (1<<sdevice_pins.
    SCIO_pin)) ucBit2Read |= 1;
    Delay(TA);
    switch(ucBit2Read)
    {
        case 1:
            ucByte2Read |= ucBit2Read;
            ucByte2Read=ucByte2Read<<1;
            break;
        case 2:
            ucByte2Read=ucByte2Read<<1;
            break;
        default:
            return RECEIVE_NOK;
            break;
    }
    ucBit2Read=0;
}
//Lesen des 8. Bits.
if(*sdevice_pins.SCIO_PIN_REG & (1<<sdevice_pins.SCIO_pin))
ucBit2Read=2;
Delay(TA);
if(*sdevice_pins.SCIO_PIN_REG & (1<<sdevice_pins.SCIO_pin))
ucBit2Read |= 1;
Delay(TA_HALF);
switch(ucBit2Read)
{
    case 1:
        ucByte2Read |= ucBit2Read;
        break;
    case 2:
        break;
    default:
        return RECEIVE_NOK;
        break;
}
```

```

*ucbyte2read=ucByte2Read;
*sdevice_pins.SCIO_DDR_REG |= 1<<sdevice_pins.SCIO_pin;
if(uc_ack==MAK) .
{
    //der Master sendet MAK
    *sdevice_pins.SCIO_PORT_REG &=~ (1<<sdevice_pins.
    SCIO_pin); //Set_Low
    Delay(TA);
    *sdevice_pins.SCIO_PORT_REG |= 1<<sdevice_pins.SCIO_
    pin; //Set_High
    Delay(TA);
}
else
{
    //der Master sendet NoMAK
    *sdevice_pins.SCIO_PORT_REG |= 1<<sdevice_pins.SCIO_
    pin; //Set_High
    Delay(TA);
    *sdevice_pins.SCIO_PORT_REG &=~ (1<<sdevice_pins.
    SCIO_pin); //Set_Low
    Delay(TA);
}
/*die Datenleitung wird auf Eingang gesetzt*/
*sdevice_pins.SCIO_DDR_REG &=~ (1<<sdevice_pins.SCIO_pin);
//die Antwort des Slaves wird gelesen SAK/NoSAK
Delay(TA_HALF);
if(*sdevice_pins.SCIO_PIN_REG & (1<<sdevice_pins.SCIO_pin))
uc_Ack=2;
Delay(TA);
if(*sdevice_pins.SCIO_PIN_REG & (1<<sdevice_pins.SCIO_pin))
uc_Ack |= 1;
Delay(TA_HALF);
if(uc_Ack==SAK) return SAK;
else return NOSAK;
}

```

### 12.2.6 Ansteuerung eines 11XXYYZ-EEPROM

Die Bausteine mit der Bezeichnung 11XXYYZ sind serielle EEPROMs, die über den Bus UNI/O angesteuert werden [8]. Wegen der kleinen Speicherkapazitäten (128 Byte bis 2 kByte) sind sie lediglich für das Speichern von kleinen Datenmengen geeignet, benötigen aber wenig Platz und nur eine Leitung für ihre Ansteuerung. Die innere Architektur unterscheidet sich von denen in Abschn. 21 beschriebenen Bausteinen nur durch die Besonderheiten der Kommunikationsschnittstelle. Die Speicher sind in 16-Byte große

Speicherseiten organisiert. Die Buchstaben XX codieren den zulässigen Temperaturbereich und die Versorgungsspannung:

- mit AA sind die Bausteine gekennzeichnet die für den industriellen Temperaturbereich ( $-40^{\circ}\text{C} \dots +85^{\circ}\text{C}$ ) und eine Versorgungsspannung von 1,8 V bis 5,5 V spezifiziert sind;
- die mit LC gekennzeichneten Bausteine decken den Automotive Temperaturbereich ( $-40^{\circ}\text{C} \dots +125^{\circ}\text{C}$ ) ab, müssen aber mit enger tolerierten Spannung (+2,5 V ... +5,5 V) versorgt werden.

Die zweistellige Zahl YY codiert die Speichergröße in Kilobit.

### 12.2.6.1 Speicherschrebschutz der 11XXYYZ-Bausteine

Wegen der reduzierten Anzahl von Pins kann der Speicher nur softwaremäßig schreibgeschützt werden. Das Statusregister, das auch den Speicherschutz regelt, ist folgendermaßen aufgebaut:

- **Bit 7:4** – sind reserviert und liefern beim Lesen “0“;
- **Bit 3:2** – BP1, BP0 bestimmen, wie in der Tab. 12.4 erläutert, den schreibgeschützten Speicherbereich;
- **Bit 1** – WEL (Write Enable Latch); wenn dieses Bit “0“ ist, ist der Schreibzugriff sowohl auf das Statusregister, als auch auf den gesamten Speicher gesperrt; das Bit wird nach der POR-Phase und nach allen Schreibvorgängen, die erfolgreich abgeschlossen werden auf “0“ zurückgesetzt;
- **Bit 0** – WIP ist ein Read-only-Bit, das intern nur während eines Speichervorgangs auf “1“ gesetzt wird.

### 12.2.6.2 Adressierung der 11XXYYZ-EEPROMs

Die acht Bit große Adresse wird beim Herstellen in die Bausteine gespeichert und kann nicht mehr geändert werden. Die Ziffer “Z“ im Bausteinname codiert die Bausteinadresse, die “0000“ für Z=0 ist, bzw. “0001“ für Z=1. Zusammen mit dem Familiencode “1010“ ergeben sich die Adressen 0xA0, bzw. 0xA1. Eine Besonderheit der Bausteine 11AA02E48, bzw. 11AA02E64 besteht darin, dass im Laufe des Herstellungsprozesses an den letzten sechs, bzw. acht Bytes des Speicherbereiches ein 48-, bzw.

**Tab. 12.4** Schreibgeschützte-Speicherbereiche der 11XXYYZ-Bausteine

BP1	BP0	Schreibgeschützter Speicherbereich
0	0	Keiner
0	1	Das oberste Viertel
1	0	Die oberste Hälfte
1	1	Gesamter Speicherbereich

64-Bit großer einmaliger Identifier gespeichert wird. Damit dieser Identifier versehentlich nicht überschrieben wird, wird der entsprechende Speicherbereich durch das Setzen des Bits BP0 im Statusregister schreibgeschützt. Anders als bei den 1-wire-Bausteine, können diese EEPROMs über diesen Identifier nicht adressiert werden, sondern nur identifiziert.

### 12.2.6.3 Funktionsbefehle der 11 XXYYZ EEPROMs

Die Ansteuerfunktionen dieser EEPROMs sind Teil eines Softwaremoduls mit dem Namen \_11XXYYZ das auf die beschriebenen UNI/O-Funktionen aufgebaut ist.

#### 12.2.6.3.1 Befehle, die eine interne Aktion auslösen

**Schreibfreigabe (0x96)** – setzt das Bit WEL im Statusregister auf “1“ und somit wird das Ändern des Statusregisters und des nicht schreibgeschützten Speichers freigegeben. Eine entsprechende Funktion wird weiter aufgelistet:

```
uint8_t _11XXYYZ_Write_Enable(tuni_oHandle sdevice_pins,
                               uint8_t ucstdby_pulse,
                               uint8_t ucaddress)
{
    if(UNI_O_Start_Header(sdevice_pins, ucstdby_pulse) == SAK)
        return START_HEADER_NOK;
    if(UNI_O_Write_Byte(sdevice_pins, ucaddress, MAK) != SAK)
        return SLAVE_ADDR_NOK;
    if(UNI_O_Write_Byte(sdevice_pins, WREN, NOMAK) != SAK) return
COMMAND_NOK;
    return OK;
}
```

Mit dem Aufruf der Funktion UNI\_O\_Start\_Header initiiert der Master die Kommunikation, danach adressiert er den gewünschten Baustein und überträgt den Befehlscode. Nur wenn eine Kommunikationsphase erfolgreich abgeschlossen ist, wird die nächste gestartet, ansonsten wird der Vorgang abgebrochen und die Funktion gibt einen Fehlercode zurück der bei der Fehleridentifikation hilft. Diese Funktion muss aufgerufen werden vor jedem Schreibvorgang.

**Schreibsperrre (0x91)** – setzt das Bit WEL auf “0“ zurück und sperrt das Schreiben auf das Statusregister und auf den gesamten Speicher.

**Speicher löschen (0x6D)** – setzt alle Bytes des Speichers auf 0x00.

**Speicher setzen (0x67)** – setzt alle Bytes des Speichers auf 0xFF.

### 12.2.6.3.2 Befehle, die auf eine implizite Adresse hinweisen

**Statusregister lesen (0x05)** – liest den Inhalt des Statusregisters und speichert ihn in eine Variable. Mit dem Lesen des Statusregisters und Testen des WIP-Bits kann festgestellt werden, wann ein Schreibvorgang beendet wird.

**Statusregister schreiben (0x6E)** – nach der erfolgreichen Schreibfreigabe wird mit diesem Befehl der Inhalt des Statusregisters geändert.

**Speicherlesen ab der aktuellen Adresse (0x06)** – mit diesem Befehl kann eine beliebige Zahl von Speicherzellen ab der aktuellen Adresse gelesen werden. Der Adresszähler, der die aktuelle Adresse speichert, wird nach der POR-Phase mit einem zufälligen Wert initialisiert. Aus diesem Grund soll die Funktion erst aufgerufen werden, wenn die Adresse eindeutig festgelegt wurde. Im Allgemeinen wird der Adresszähler nach dem Lesen einer Speicherzelle innerhalb des gesamten Speicherbereiches inkrementiert. Nach dem Lesen von der höchsten Speicheradresse springt der Adresszähler auf die Adresse 0.

### 12.2.6.3.3 Befehle, die auf eine explizite Adresse hinweisen

**Speicher lesen (0x03)** – mit dem Befehl kann eine beliebige Zahl von Speicherzellen gelesen werden. Die Anfangsadresse muss als 2-Byte-Zahl (mit dem höherwertigen Byte zuerst) nach dem Senden des Befehlscodes übermittelt werden.

**Speicher schreiben (0x6C)** – mit diesem Befehl, so wie in der folgenden Beispielfunktion veranschaulicht, kann der Inhalt von bis zu 16 Speicherzellen (eine Speicherseite) geändert werden. Die Anfangsadresse wird wie beim Speicherlesen übermittelt. Die zu speichernden Bytes werden zuerst in einem flüchtigen 16-Byte großen Speicherpuffer zwischengespeichert. Aus dem Zwischenspeicher werden die übertragenen Bytes im EEPROM dauerhaft abgelegt. Dieser Vorgang wird mit dem fehlerfreien Abschluss der Kommunikation nach dem Empfang der NOMAK Bestätigung des Masters gestartet und dauert ca. 5 ms. Um sicher zu sein, dass die Funktion auch bei der parasitären Versorgung des Slaves fehlerfrei funktioniert, wird die Busleitung am Ende der Funktion auf High gesetzt. Nach dem Schreiben eines Bytes wird der Adresszähler nur innerhalb der Speicherseite inkrementiert. Nach dem Erreichen der höchsten Adresse einer Seite springt der Adresszähler auf die Anfangsadresse dieser Seite.

```
uint8_t _11XXYYZ_Write_Memory(tuni_oHandle sdevice_pins, uint8_t
ucstdby_pulse,
                                uint8_t ucdev_address, uint16_t uimemory_
address,
                                uint16_t uin_bytes, uint8_t
*uctargetaddress)
```

```

{
    uint16_t i;
    uint8_t ucMemoryLowAddress=uimemory_address,
    ucMemoryHighAddress= (uimemory_address>>8);
    if(_11XXYYZ_Write_Enable(sdevice_pins, ucstdby_pulse, ucdev_
address) !=OK) return NOK;
    if(UNI_O_Start_Header(sdevice_pins, ucstdby_pulse)==SAK)
    return START_HEADER_NOK;
    if(UNI_O_Write_Byte(sdevice_pins, ucdev_address, MAK) !=SAK)
    return
    SLAVE_ADDRESS_NOK;
    if(UNI_O_Write_Byte(sdevice_pins, WRITE, MAK) !=SAK) return
    COMMAND_NOK;
    if(UNI_O_Write_Byte(sdevice_pins, ucMemoryHighAddress, MAK)
!=SAK) return           MEMORY_ADDR_NOK;
    if(UNI_O_Write_Byte(sdevice_pins, ucMemoryLowAddress, MAK)
!=SAK) return
    MEMORY_ADDRESS_NOK;

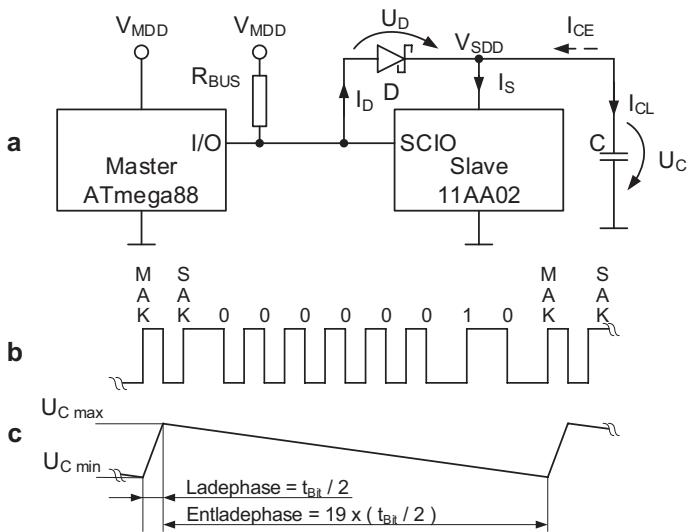
    for(i=0; i<(uin_bytes - 1); i++)
    {
        if(UNI_O_Write_Byte(sdevice_pins, uctargetaddress[i],
MAK) !=SAK) return
        RECEIVE_NOK;
    }
    if(UNI_O_Write_Byte(sdevice_pins,uctargetaddress[i], NOMAK)
!=SAK) return RECEIVE_NOK;
    UNI_O_Set_SCIO_High(sdevice_pins);
    return OK;
}

```

#### 12.2.6.4 Parasitäre Versorgung eines 11XXYYZ-EEPROM

Die parasitäre Versorgung eines solchen EEPROM ist sinnvoll, wenn der Verzicht auf die Versorgungsleitung wichtiger ist, als die komplexere Beschaltung, wie im Bild Abb. 12.11 dargestellt. Die Wahl der Diode und die Dimensionierung des Kondensators sind in [10] beschrieben. Im Folgenden wird die Vernetzung eines Mikrocontrollers ATmega88 mit einem UNI/O-Slave vom Typ 11AA02, der parasitär versorgt wird, betrachtet. Zuerst wird geprüft, ob die Voraussetzungen einer parasitären Versorgung erfüllt sind. Für eine Versorgungsspannung des Busmasters  $V_{MDD}=5\text{ V}$  findet man folgende Angaben im Datenblatt [8]:

- maximaler High-Ausgangsstrom  $I_{MOH\ max}=40\text{ mA}$ ;
- minimale High-Ausgangsspannung  $V_{MOH\ min}=4,2\text{ V}$  @  $I_{MOH}=20\text{ mA}$ ;
- minimale High-Eingangsspannung  $V_{MIH\ min}=0,6 \times V_{MDD}=3\text{ V}$ .



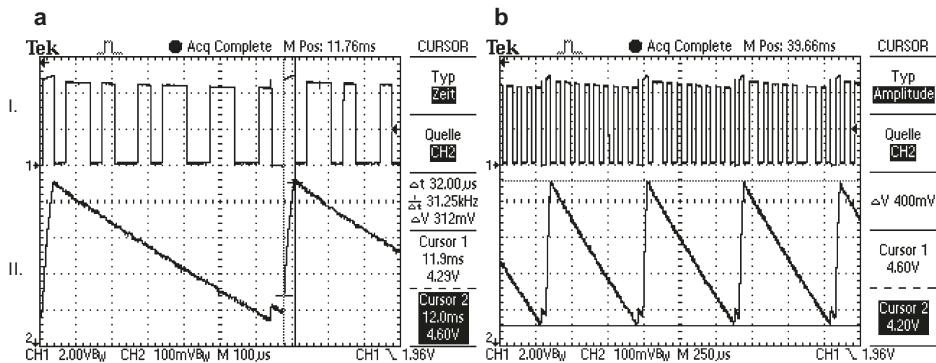
**Abb. 12.11** Uni/O-Bus mit parasitären Beschaltung des Slaves. a) Beschaltung; b) Zeitlicher Verlauf bei der Übertragung eines Bytes; c) Spannungsverlauf am Kondensator beim Lesen eines Bytes

Für den Baustein 11AA02 sind folgende Angaben dem Datenblatt [8] zu entnehmen:

- minimale High-Ausgangsspannung  $V_{SOH \ min} = V_{SDD} - 0,5 \text{ V}$ , wobei  $V_{SDD}$  die Versorgungsspannung des Slaves ist;
- Versorgungsstrom beim Lesen  $I_{S \ Read} = 3 \text{ mA}$  @  $V_{SDD} = 5,5 \text{ V}$ ;
- Versorgungsstrom beim Speichern  $I_{S \ Write} = 5 \text{ mA}$ ,  $t_{Write} = 5 \text{ ms}$ .

Eine Schottky-Diode ist wegen der vernachlässigbaren Umschaltzeiten und niedriger Durchlassspannung an dieser Stelle die erste Wahl. Die Schottky-Diode BAT48 weist eine Durchlassspannung  $U_D$  von 0,35 V [11] bei einem Durchlassstrom von 40 mA auf. Der Kondensator soll die elektrische Energie puffern und sie für die Versorgung des EEPROM bereitstellen. In Abb. 12.11b ist beispielhaft die Übertragung eines Bytes dargestellt. Wegen der Manchester Codierung spielt die Zusammensetzung des Bytes keine Rolle weil die Hälfte jedes Bits auf High ist.

Man unterscheidet zwei Betriebsphasen des Kondensators. In der ersten ist der Mikrocontrolleranschluss auf Ausgang und High geschaltet, die Diode leitet und der Kondensator lädt sich auf. In der zweiten Betriebsphase ist der Anschluss auf Ausgang und Low, oder auf Eingang geschaltet. In dieser zweiten Phase sperrt die Diode und der Kondensator entlädt sich über den Versorgungspin des Slaves. Das Auslesen eines größeren Speicherbereiches ist der ungünstigste Fall, der weiterhin betrachtet werden soll. Beim Auslesen eines Bytes (zehn Bit) sendet der Master nur das Bestätigungsbit MAK, sodass der Kondensator nur über 5 % der Bytedauer aufgeladen werden kann.



**Abb. 12.12** Parasitäre Versorgung eines UNI/O-EEPROM

Während der Ladephase kann der Ausgang des Mikrocontrollers als eine Konstantstromquelle betrachtet werden die maximal 40 mA liefern kann und den Kondensator mit dem Strom:

$$I_{CL} = I_{MOH \max} - I_{S \text{ Read}} = 40 \text{ mA} - 3 \text{ mA} = 37 \text{ mA} \quad (12.2)$$

auflädt. Am Ende dieser Phase erreicht die Kondensatorspannung den maximalen Wert:

$$U_C \max = U_{SDD} = V_{MOH \min} - U_D = 4,2 \text{ V} - 0,35 \text{ V} = 3,85 \text{ V} \quad (12.3)$$

und die zugeführte elektrische Ladung kann folgendermaßen berechnet werden:

$$Q_L = I_{CL} \cdot t_{Bit}/2 \quad (12.4)$$

In der Entladungsphase kann der Rückwärtsstrom der Diode und der Eingangsstrom des Mikrocontrollers (beide ca. 1 μA) vernachlässigt und der Entladestrom  $I_{S \text{ Read}} = 3 \text{ mA}$  als konstant angenommen werden. Die Kondensatorspannung erreicht am Ende dieser Phase den Wert  $U_C \min$  und die entnommene elektrische Ladung beträgt:

$$Q_E = I_{CE} \cdot 19 \cdot (t_{Bit}/2) \quad (12.5)$$

Ein stationärer Zustand wird erzielt, indem die Kondensatorspannung nach jedem gelesenen Byte denselben Wert erreicht, dafür müssen die zugeführte und die entnommene elektrische Ladung gleich sein. Aus den Gleichungen Gl. 12.4 und 12.5 kann der maximale Entladestrom berechnet werden:

$$I_{CE \ max} = \frac{I_{CL}}{19} = \frac{37}{19} \approx 1,95 \text{ mA} \quad (12.6)$$

Der berechnete Strom ist kleiner als der im Datenblatt angegebene  $I_{S \text{ Read}}$ . Der Versorgungsstrom sinkt aber bei niedrigeren Versorgungsspannungen des EEPROM (siehe [8]) und muss in der Praxis getestet werden. Ist die Bedingung:

$$I_{S \text{ Read real}} \leq \frac{I_{MOH \ max} - I_{S \text{ Read real}}}{19} \quad (12.7)$$

nicht gültig, so ist die Voraussetzung für die parasitäre Versorgung nicht erfüllt. Der stationäre Zustand wird nicht erreicht und nach einigen Bytes treten Kommunikationsfehler auf.

Im nächsten Schritt wird die Kapazität des Kondensators berechnet bei einer Bitrate von 10 kBit/s in der Annahme, dass die Bedingung aus Gl. 12.7 erfüllt ist. Im Grenzfall ist die fehlerfreie Kommunikation gewährleistet, wenn:

$$V_{SOH\ min} = V_{SDD\ min} - 0,5\text{ V} \geq V_{MIH\ min}. \quad (12.8)$$

Aus  $V_{SDD\ min} = U_{C\ min}$ , folgt

$$U_{C\ min} \geq V_{MIH\ min} + 0,5\text{ V} = 3\text{ V} + 0,5\text{ V} = 3,5\text{ V} \quad (12.9)$$

In der Entladungsphase gibt der Kondensator die maximale elektrische Ladung  $Q_E$  ab

$$Q_E = I_{S\ Read} \cdot 19 \cdot t_{Bit}/2 = \frac{3 \cdot 10^{-3} \cdot 19\text{ A} \cdot \text{s}}{2 \cdot 10000} = 2,85\text{ }\mu\text{As}$$

und der Spannungsabfall am Kondensator beträgt:

$$\Delta U_C = U_{C\ max} - U_{C\ min} = 3,85\text{ V} - 3,5\text{ V} = 0,35\text{ V}.$$

Die abgegebene Ladung ist proportional zu der Kapazität und dem Spannungsabfall des Kondensators:

$$Q_E = C \cdot \Delta U_C \quad (12.10)$$

Daraus resultiert:  $C = \frac{Q_E}{\Delta U_C} = \frac{2,85\text{ }\mu\text{As}}{0,35\text{ V}} = 8,1\text{ }\mu\text{F}$ .

Experimentell wurde das Lesen eines Datenpakets von 128 Bytes unter der Verwendung für die parasitäre Versorgung des EEPROM eines keramischen Kondensators mit einer nominalen Kapazität von 1  $\mu\text{F}$  untersucht. Die Abb. 12.12 zeigt zeitliche Abschnitte dieser Übertragung im stationären Zustand. Die Abb. 12.12 I. zeigt den Datenverlauf am Bus und Abb. 12.12 II. den Spannungsverlauf am Pufferkondensator. Während des Lesens eines Bytes variiert die Spannung am Kondensator zwischen 4,2 V und 4,6 V. Diese Spannung reicht für die sichere Versorgung des EEPROM.

Der Versuch zeigt, dass auch bei kleineren Kapazitäten als die mit Gl. 12.10 berechnet und bei höheren Frequenzen die Datenübertragung bei parasitärer Versorgung fehlerfrei funktioniert, sie muss aber experimentell getestet werden.

## 12.3 LIN-Bus

Das Local Interconnect Network (LIN) wird als Subbus bezeichnet, weil es für minimale Kosten, minimalen Ressourcenverbrauch und folglich auf nur geringe Datenraten spezifiziert ist. Das Spezifikationsdokument [13] und das Buch [15] beschreiben das System ausführlich. Bis 2019 war die letzte Spezifikation 2.2 A im Netz frei verfügbar, manche Anbieter halten sie noch zum Download bereit.

In diesem Buch beschreiben wir den LIN-Bus der Vollständigkeit halber, jedoch nicht in seiner Implementierung.

LIN basiert auf dem asynchronen UART, der in jedem gängigen Mikrocontroller integriert ist. Im Gegensatz zu einer vollduplexfähigen UART Schnittstelle ist er jedoch an die Spannungspiegel im Fahrzeug angepasst und als Halbduplexsystem in Wired-AND-Technik (open collector) mit einer unsymmetrischen Eindrahtschnittstelle ausgelegt. Da LIN auch von Kleinstrechnern (intelligente Lampen, Schalter oder Stellmotoren) gegebenenfalls ohne eigenen Quarzoszillator laufen soll, liegt die spezifizierte Grenze der Datenrate bei 20 kbit/s. Der LIN-Transceiver enthält je nach Ausführung die Pegelanpassung, Powermanagement-Komponenten (Wake-up), Überwachungsschaltungen (Watchdog) und die Energieversorgung des Mikrocontrollers. Auf diese Weise lassen sich sehr kompakte Schaltungen realisieren.

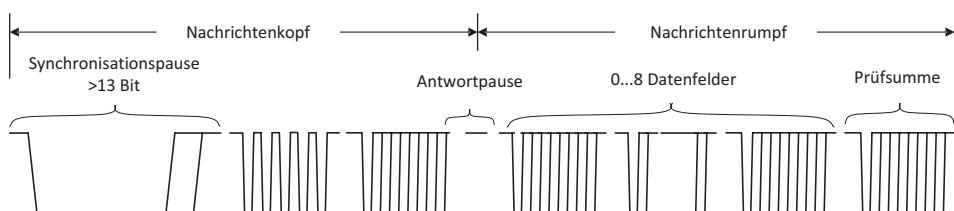
LIN ist wie CAN botschaftenadressiert, d. h. Nachrichten werden nicht zwischen Instanzen verschickt sondern die Adresse bezieht sich auf die Bedeutung der Nachricht, so dass diese bei Bedarf von verschiedenen Busteilnehmern gelesen werden kann. In der Kommunikationstechnik wird dies als Producer–Consumer-Modell bezeichnet.

Zur Bussteuerung, verwendet auch LIN ein Master-Slave-Protokoll: Jede Kommunikation auf dem Bus wird von einem Masterknoten angestoßen. Dieser sendet, nach einer mehr oder weniger festen Ablauftabelle (Schedule) einen Nachrichtenkopf (Header), der von einem Synchronisationsfeld der Länge 13 Bit angeführt wird (Übergang von einem rezessiven auf ein dominantes Bit), während dessen die angeschlossenen Slaves den Empfang vorbereiten können. Ab diesem Zeitpunkt wird die Kommunikation bytewise durch UART übernommen. Zunächst wird eine hexadezimale 0x55, also eine binäre 01010101, gesendet, damit sich die Slaves auf die Taktrate synchronisieren können. Die Bytesynchronisation wird dabei von den UART-eigenen Start- und Stoppbits durchgeführt. Danach wird ebenfalls per UART ein 6 Bit Identifier (ID0...ID5) mit 2 Bit-Prüfsumme verschickt, sodass insgesamt 64 Nachrichten adressiert werden können (siehe Abb. 12.13). Dieser Identifier wird auch PID genannt.

Die 2-Bit-Prüfsumme besteht aus zwei Paritätsbits:

$$P0 = ID0 \oplus ID1 \oplus ID2 \oplus ID4 \quad (12.11)$$

$$P1 = ! (ID1 \oplus ID3 \oplus ID4 \oplus ID5) \quad (12.12)$$



**Abb. 12.13** LIN-Frame

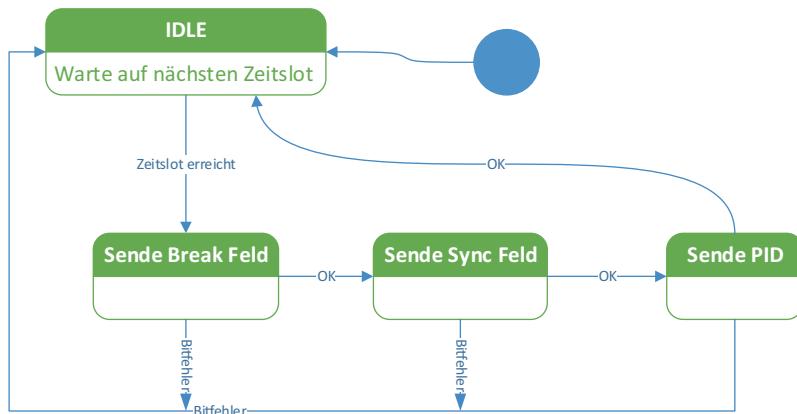
Nun werden folgende Fälle unterschieden:

- Die Botschaft soll vom Master an einen oder mehrere Slaves geschickt werden: In diesem Fall setzt der Master die UART-Übertragung mit dem Datenfeld fort, das aus acht Bytes bestehen darf. Die Slaves, die sich für die Botschaft interessieren, lesen mit.
- Die Botschaft soll von einem Slave versendet werden. In diesem Fall stellt der Master die Übertragung nach dem Versenden des Identifiers samt Prüfsumme ein und der Slave setzt sie mit dem Datenfeld fort. Aus Kapazitätsgründen sind auch so genannte eventgetriggerte Frames erlaubt, d. h. mehrere Slaves könnten auf einen Identifier hin eine Antwort senden, wenn die entsprechenden Sendegründe vorliegen. In diesem Fall gibt es eine Kollision, die vom Master erkannt und durch Einzelabfrage mit getrennten Identifiern behoben wird, wofür jeweils ein weiteres Standard-Frame zur Verfügung gestellt werden muss.
- Am Ende der Übertragung wird noch eine 8-Bit-Prüfsumme gesendet. Ihre Berechnung erfolgt wie beim Modbus als LRC nach folgender Formel:

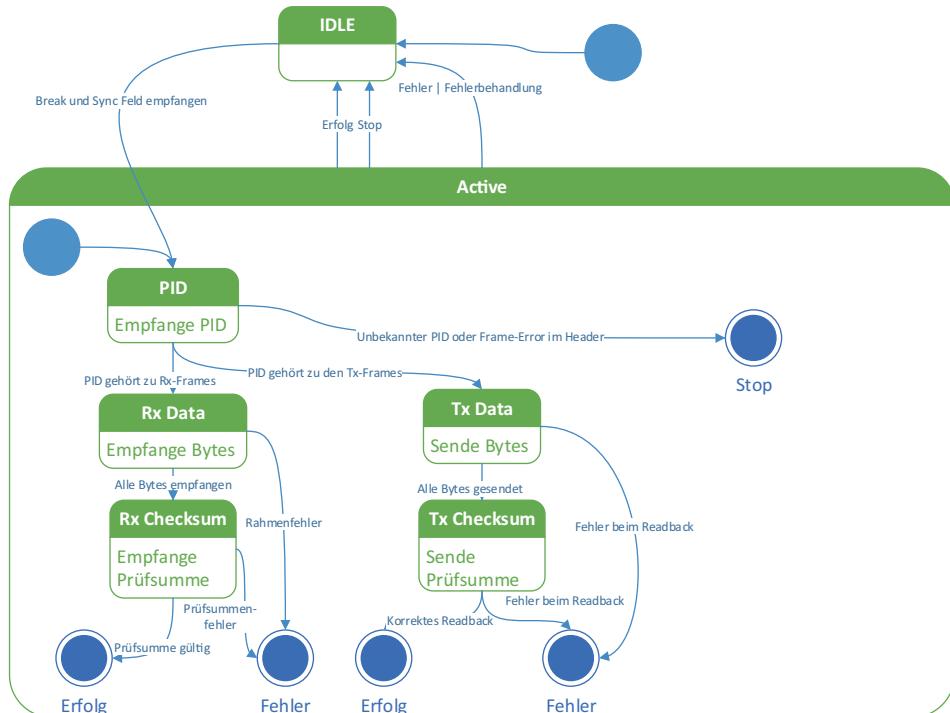
$$\text{Checksum} = \text{INV}(\text{Datenbyte 1} \oplus \text{Datenbyte 2} \oplus \dots \oplus \text{Datenbyte 8}) \quad (12.13)$$

Zur Bildung der Prüfsumme werden die einzelnen Datenbytes per Modulo-256-Arithmetik addiert (XOR) und anschließend bitweise invertiert.

Letztlich laufen auf einem LIN-Master zwei Zustandsautomaten ab, der des Masters, für die Versendung des Nachrichtenkopfs und der eines Slaves zum Senden oder Empfangen einer Botschaft. Jeder Slave besitzt eine Tabelle mit den Identifiern, die für ihn bestimmt sind. Diese verweist auf die vorgesehene Richtung (Rx-Frame = der Slave liest mit, Tx-Frame = der Slave vervollständigt die Botschaft mit Daten) und auf die Anzahl der zu sendenden beziehungsweise zu empfangenden Bytes in der Botschaft. Der Ablauf ist in den Abb. 12.14 und 12.15 skizziert.



**Abb. 12.14** Zustandsautomat des LIN-Masters nach [13]



**Abb. 12.15** Zustandsautomat des LIN-Slaves nach [13]

Da jeder LIN-Knoten aufgrund der open-collector-Anbindung an den Bus die Botschaften mitlesen kann, die er sendet, kann auf diese Weise ein Übertragungsfehler identifiziert werden.

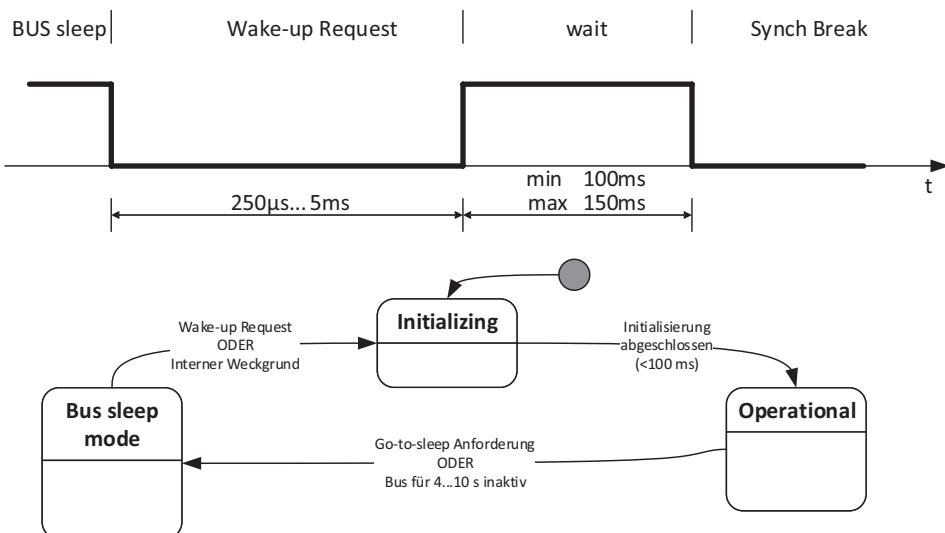
Die Botschaften mit den Identifiern 0x60 und 0x61 sind für Konfigurations- und Diagnosebotschaften reserviert. Hier sorgt ein überlagertes Transportprotokoll für die Segmentierung der Nachrichten, die in der Regel größer als 8 Byte sind. Beispielsweise kann man Message Identifier gegenüber den Voreinstellungen modifizieren, den Knotenstatus auslesen, den Knoten schlafenlegen usw. Die Vorgehensweise entspricht vereinfacht dem UDS-Protokoll (Unified Diagnostic Service) und würde den Rahmens des Buchs sprengen.

Die LIN-Spezifikation legt nicht nur die Protokolle und die Bitübertragungsschicht fest sondern auch die Softwareschnittstellen des Protokollstacks (API) und vor allem ein Entwicklungskonzept. Dieses erleichtert die Integration von LIN- Komponenten erheblich. So muss mit jeder LIN-Komponente eine standardisierte Beschreibung ihrer Signale und Botschaften mitgeliefert werden (Node capability file NCF). Die Beschreibungen aller Komponenten im System ergibt das sogenannte LIN-Description-File (LDF), aus dem die Botschaftenidentifier ermittelt werden können. Das LIN-Diagnoseprotokoll fordert, dass die Identifier dynamisch durch Konfiguration in den Knoten verändert

werden können. Auf diese Weise lassen sich die für das Preissegment notwendigen hohen Stückzahlen realisieren bei erleichterter Integration in das Gesamtsystem. Dennoch ist LIN natürlich kein Plug&Play-Netzwerk, denn die funktionale Vielfalt ist auf der Transportebene bewusst nicht standardisiert.

Schließlich wird in LIN auch ein Powermanagement realisiert. Man geht davon aus, dass alle Sensoren oder Aktoren, die an LIN angeschlossen sind, dauerversorgt werden (im Fahrzeug ist das die so genannte Klemme 30). Sollte das System in einen Stromsparmodus versetzt werden (Sleep Mode), wird das Netzteil (im Fahrzeug ist das der DC/DC-Wandler von der Bordnetzspannung zur Versorgungsspannung 5 V oder 3,3 V der Elektronik) in einen Zustand der minimalen Energieaufnahme versetzt. Ein Betrieb des Prozessors ist damit nicht oder nur eingeschränkt möglich (siehe Kap. 3), aber der LIN-Baustein reagiert weiterhin auf den Bus. Dieser Sleep-Mode wird erreicht, wenn das Mastersteuergerät 4...10 s keine Anforderungen schickt (Bus inaktiv) oder explizit einen Goto-Sleep Request mit der Botschaft 0x60 versendet. Zum Aufwecken muss der Master einen dominanten Impuls von mindestens 250 µs bis 5 ms auf den Bus geben und danach mindestens 100 ms warten, bis der Slave seinen Bootvorgang abgeschlossen hat. Der Weckimpuls veranlasst den LIN-Controller, das Netzteil wieder einzuschalten und einen Power-on-Reset des Prozessors durchzuführen. Anschließend beginnt die ganz normale Kommunikation nach Schedule-Tabelle. Abb. 12.16 veranschaulicht diesen Vorgang.

LIN ist standardmäßig in AUTOSAR integriert, einer offenen und standardisierten Softwarearchitektur für elektronische Steuergeräte im Fahrzeug. Daneben existieren einige quelloffene Stacks, die mehr oder weniger komplett ausgeführt sind.



**Abb. 12.16** LIN Wake-up Sequenz und Zustandsautomat des Powermanagements eines LIN-Knotens nach [13]

## Literatur

1. Maxim Integrated Products. AN937: Book of iButton® standards. [www.maximintegrated.com](http://www.maximintegrated.com). Zugegriffen: 20. März 2021.
2. Maxim Integrated Products. AN148: Guidelines for reliable long line 1-wire networks. [www.maximintegrated.com](http://www.maximintegrated.com). Zugegriffen: 27. März 2021.
3. Maxim Integrated Products. AN27: Understanding and using cyclic redundancy checks with maxim 1-wire and iButton products. [www.maximintegrated.com](http://www.maximintegrated.com). Zugegriffen: 21. März 2021.
4. Maxim Integrated Products. DS1972 data sheet. 1024-Bit EEPROM iButton. [www.maximintegrated.com](http://www.maximintegrated.com). Zugegriffen: 19. März 2021.
5. Werner, M. (2008). *Information und Codierung*. Vieweg + Teubner.
6. Maxim Integrated Products. DS18B20 data sheet. Programmable resolution 1-wire digital thermometer. [www.maximintegrated.com](http://www.maximintegrated.com). Zugegriffen: 9. Febr. 2021.
7. Microchip Technology Inc. UNI/O® bus specification. [www.microchip.com](http://www.microchip.com). Zugegriffen: 31. Mai 2021.
8. Microchip Technology Inc. 1K-16K UNI/O® Serial EEPROM family data sheet. [www.microchip.com](http://www.microchip.com). Zugegriffen: 26. Okt. 2021.
9. Microchip Technology Inc. 2K UNI/O® Serial EEPROMs with EUI-48™ or EUI-64™ node identity. [www.microchip.com](http://www.microchip.com). Zugegriffen: 13. Mai 2021.
10. Microchip Technology Inc. AN1213 Powering a UNI/O® bus device through SCIO. [www.microchip.com](http://www.microchip.com). Zugegriffen: 13. Mai 2021.
11. Microchip. Reference Manual ATmega48/168, <https://www.microchip.com/wwwproducts/en/ATmega88A>. Zugegriffen: 6. Jan. 2021.
12. STMicroelectronics. BAT48 – Small signal Schottky diode. [www.st.com](http://www.st.com). Zugegriffen: 7. Aug. 2021.
13. ISO: ISO 17987 Part 1–7:2016 Road vehicles — Local Interconnect Network (LIN).
14. Grzembra, A., & von der Wense, A. (2005). *LIN-Bus: Systeme, Protokolle, Tests von LIN-Systemen, Tools, Hardware, Applikationen* (1. Aufl.). Franzis.
15. Microchip Technology Inc. Local Interconnect Network (LIN) Bus. <https://microchip-developer.com/lin:start>. Zugegriffen: 2. Febr. 2021.



# Drahtlose Netzwerke

13

## Zusammenfassung

In diesem Kapitel werden drahtlose Netzwerke, insbesondere im Bereich der ISM-Bänder um 433 MHz und 868 MHz, sowie im ISM-Band 2,4 GHz beschrieben. Hierfür existieren zahllose Bausteine, von einigen ausgewählten wird der Einsatz hier erklärt

Der Einsatz von Mikrocontrollern und Sensoren in Bereichen, bei denen 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 Dinge<sup>1</sup>“. Die leitungsgebundene Übertragung ist sicherer und zuverlässiger als die Funkübertragung, die Funknetzwerke können aber leichter erweitert werden.

<sup>1</sup> englisch IoT = Internet of Things.

Die Originalversion dieses Kapitels wurde revidiert. Ein Erratum ist verfügbar unter  
[https://doi.org/10.1007/978-3-658-31709-6\\_27](https://doi.org/10.1007/978-3-658-31709-6_27)

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann  
[https://doi.org/10.1007/978-3-658-31709-6\\_13](https://doi.org/10.1007/978-3-658-31709-6_13).

## 13.1 Grundlagen der Funkschnittstellen

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 (SRD<sup>2</sup>) die maximale Sendeleistung für jeden Frequenzbereich [7]. Diese Begrenzung trägt zur Störsicherheit der Funkkommunikation bei. Für industrielle, wirtschaftliche, medizinische, häusliche oder ähnliche Anwendungen (ISM<sup>3</sup>) werden unlizenzierte Frequenzbereiche zugewiesen, die mit wenigen Einschränkungen frei benutzt werden können [9]. Das 2,4-GHz-ISM-Band, dem in Deutschland der Frequenzbereich 2,4 GHz bis 2,5 GHz zugewiesen ist, ist weltweit definiert und bildet den Auswahlraum für den Übertragungskanal zahlreicher Funkprotokolle.

### 13.1.1 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.

#### Frequenzmultiplex (FDMA<sup>4</sup>)

Beim Frequenzmultiplex-Verfahren wird jedem Sender ein anderer Funkkanal zugewiesen. Die Frequenzbereiche der verwendeten Funkkanäle sollen sich nicht überlappen.

#### Zeitmultiplex (TDMA<sup>5</sup>)

Das Zeitmultiplex-Verfahren ermöglicht die mehrfache Nutzung des gleichen Funkkanals. Die Sendezeit wird in Zeitrahmen fester Dauer und der Zeitrahmen in Zeitschlitz unterteilt. Beim synchronen Verfahren ist die Anzahl der Zeitschlitz gleich der Senderzahl und jedes Gerät darf nur in dem ihm zugeordneten Zeitschlitz senden. Das asynchrone Verfahren sieht eine dynamische Vergabe der Zeitschlitz vor, abhängig vom Sendebedarf der Geräte [10].

#### Codemultiplex (CDMA<sup>6</sup>)

Das Codemultiplex-Verfahren gehört zu der Spreizbandtechnik. Jedes Bit einer Nachricht wird vor der Modulation mit einem speziellen, binären Spreizcode multipliziert,

---

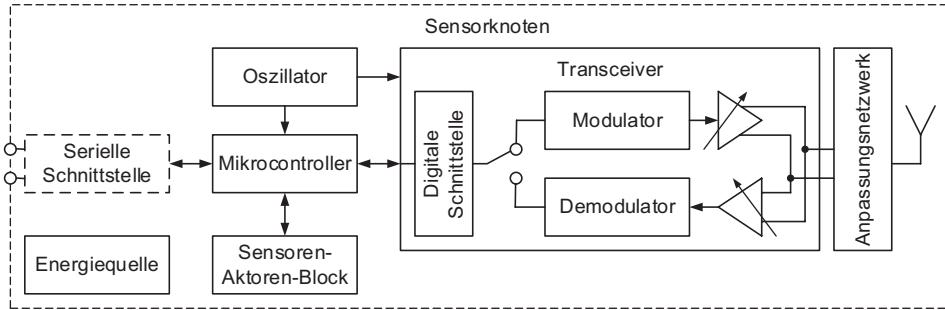
<sup>2</sup>SRD – Short Range Device.

<sup>3</sup>ISM – Industrial, Scientific and Medical.

<sup>4</sup>FDMA – Frequency Division Multiplex Access.

<sup>5</sup>TDMA – Time Division Multiplex Access.

<sup>6</sup>CDMA – Code Division Multiplex Access.



**Abb. 13.1** Funkknoten – Blockschaltbild

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.

### 13.1.2 Sensorknoten

Ein einfacher Funkknoten, der, wie in Abb. 13.1 dargestellt, Sensoren und Aktoren beinhaltet, wird als Sensorknoten bezeichnet [1]. 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.

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 Eingangs-empfindlichkeit und einem RF-Schalter. Zusätzlich können einige Transceiver Aufgaben der Sicherungsschicht übernehmen (z. B. [2–4]) wie zum Beispiel:

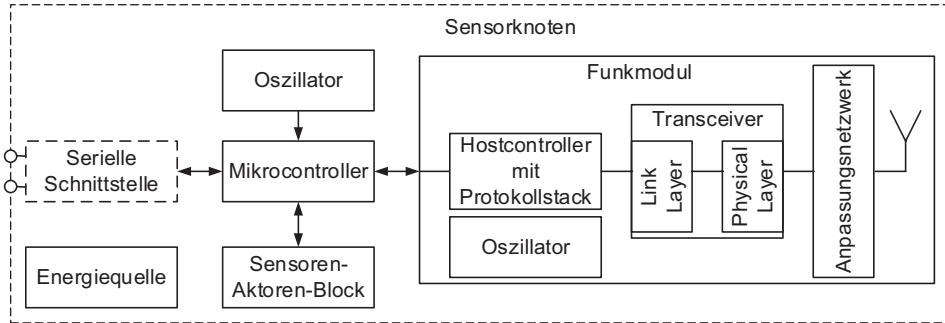


Abb. 13.2 Funkknoten mit Funkmodul

- 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 RSSI<sup>7</sup> als analoges, oder digitales Signal ist proportional mit der Leistung des empfangenen Signals;
- das LQI<sup>8</sup> 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).

Der in Abb. 13.1 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. 13.2 dargestellt.

<sup>7</sup>RSSI – Received Signal Strength Indication.

<sup>8</sup>LQI – Link Quality Indication.

Auf dem Mikrocontroller sind weiterhin die oberen Schichten des ISO/OSI-Modells implementiert. Der Transceiver, als Teil des Funkmoduls, integriert meist die Bitübertragungs- und die Sicherung-Schicht, während auf dem Hostcontroller ein Protokoll-Stack mit den weiteren Schichten gespeichert ist. Manche Funkmodule bieten zusätzlich digitale I/Os eventuell auch analoge Eingänge, um Sensoren direkt einzulesen, bzw. Aktoren direkt anzusteuern.

---

## 13.2 Funkübertragung im 433 MHz und 868 MHz ISM-Band

RFM12B ist ein Funk-Transceiver der Firma HOPERF Microelectronics [14], der die digitale Frequenzmodulation FSK<sup>9</sup> benutzt und die in Europa zugelassene 433 und 868 MHz ISM-Frequenzbänder abdeckt. Unter dem gleichen Namen werden fertige Module hergestellt, die neben dem Transceiver einen Quarzkristall, Entstörkondensatoren, eventuell einen analogen Filter und ein Antennen-Anpassung-Netzwerk beinhaltet. Die Einstellung des Transceivers erfolgt über eine Vierdraht-SPI-Schnittstelle. Die zu sendenden Daten werden über SPI oder als PCM-Datenstrom über den Pin FSK/DATA bereitgestellt. Die empfangenen Daten können über SPI oder als PCM-Datenstrom zusammen mit dem rekonstruierten Takt über die Pins FSK/DATA, bzw. DCLK gelesen werden. Wenn die Daten über SPI übermittelt werden, kann die Luft-Übertragungsrate 115,2 kBit/s erreichen, während sie bei der PCM-Übermittlung 256 kBit/s erreichen kann.

### 13.2.1 Aufbau des RFM12B

Ein Blockschaltbild des Transceivers mit den für das Funkmodul relevanten Anschlüsse ist in Abb. 13.3 zu sehen. Das Modul verfügt über zwei Kommunikationskanäle: eine leitungsgebundene Schnittstelle (SPI) für die Kommunikation mit dem übergeordneten Mikrocontroller und ein Funkkanal für die Kommunikation mit anderen Funkmodulen. Der Transceiver besteht aus einem HF<sup>10</sup> – und einem NF<sup>11</sup> -Teil. Im HF-Teil ist ein Sender, der mit dem FSK-Modulationsverfahren arbeitet und ein I/Q<sup>12</sup> – Demodulator untergebracht. Wegen des geringen Energieverbrauchs kann der Transceiver in batteriebetriebene Geräte integriert werden. Um den Energieverbrauch zu optimieren und zu vermeiden, dass beim Umschalten zwischen Sender- und Empfänger-Betrieb ein Spannungsabfall stattfindet, der zu einem Interrupt führen könnte, können die

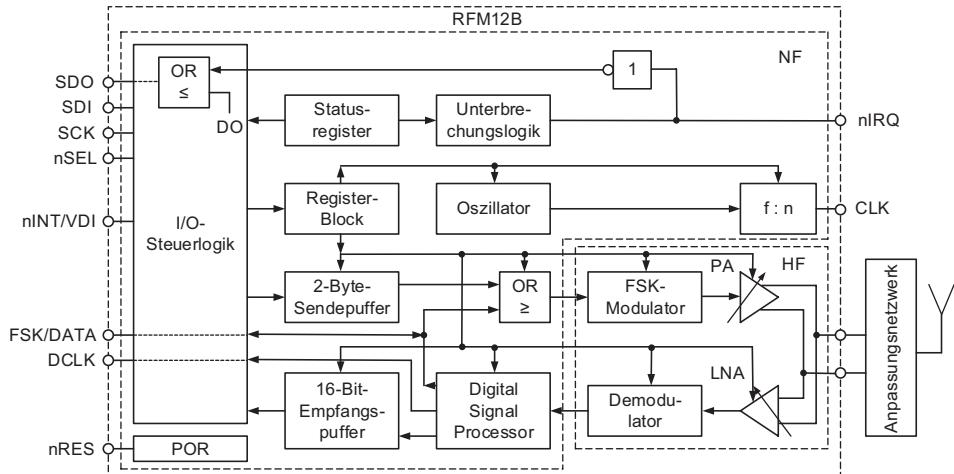
---

<sup>9</sup>FSK – Frequency Shift Keying.

<sup>10</sup>HF – Hochfrequenz.

<sup>11</sup>NF – Niederfrequenz.

<sup>12</sup>I/Q – In Phase & Quadrature (Demodulation Verfahren).



**Abb. 13.3** RFM12B-Blockschaltbild

entsprechenden Schaltungsteile auch nur teilweise ein- und ausgeschaltet werden. Im NF-Teil übernimmt die I/O-Steuerlogik die Pinansteuerung und die SPI-Kommunikation.

Ein umschaltbarer Oszillator taktet den gesamten HF-Teil. Über einen einstellbaren Frequenzteiler kann am Pin CLK die gewünschte Frequenz für die Ansteuerung des übergeordneten Mikrocontrollers erzeugt werden.

Beim Einschalten des Bausteins beginnt eine Phase, in der die POR-Schaltung den Chip in einen sicheren Zustand versetzt. Diese Phase soll laut [14] maximal 100 ms dauern. Die POR-Schaltung wird auch nach einem Hardware- oder Software-Reset aktiv. Während dieser Phase darf keine Kommunikation über SPI stattfinden. Das Ende der Phase wird durch das Setzen des Bit 14 im Statusregister signalisiert.

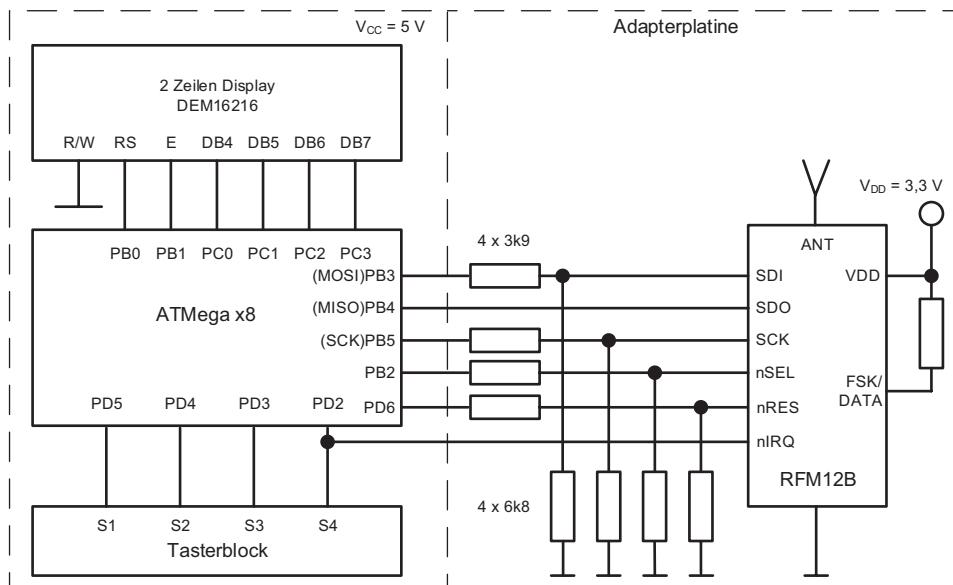
Die Änderungen der während der POR-Phase gespeicherten Einstellungen erfolgen mit Hilfe von Befehlen, die über SPI übertragen werden. Die Einstellungen werden in den Register-Block abgelegt.

Ein zuschaltbarer und zwei Bytes großer Sendepuffer kann die zu sendenden Bytes zwischenspeichern bevor sie moduliert und per Funk übertragen werden.

Ein DSP (Digital Signal Processor) verarbeitet die demodulierten Daten und leitet sie entweder in einen 16 Bit großen FIFO-Puffer oder gibt sie an Pin FSK/DATA als PCM-Datenstrom zusammen mit dem rekonstruierten Taktsignal am Pin DCLK aus.

### 13.2.2 Beschaltung des RFM12-Funkmoduls

Eine mögliche Ansteuerung des Funkmoduls mit einem Mikrocontroller ATmega88 ist in Abb. 13.4 zu sehen. Die Nutzdaten werden bei dieser Beschaltung über SPI übermittelt. Zusätzlich zu den SPI-Anschlüssen: SDI, SDO, SCK und nSEL steuert der



**Abb. 13.4** Beschaltung eines Funkmoduls RFM12B

Mikrocontroller den Reset-Pin nRES und wertet den Interrupt-Ausgang nIRQ des Transceivers aus. Der Transceiver liefert auf SPI-Anfrage am Pin SDO den Inhalt des Statusregisters, ansonsten den invertierten Zustand des Interrupt-Pins nIRQ, siehe Abb. 13.3. Ein Pull-up-Widerstand wurde am Pin FSK/DATA vorgesehen, um bei den getesteten Modulen RFM12S Rev. 3.0 die volle Funktionalität zu gewährleisten. Ohne diesen Widerstand wird mit dem Senden des ersten Bytes ein Interrupt ausgelöst und der Pin nIRQ wird dauernd auf Low geschaltet.

### 13.2.3 Die SPI-Kommunikation

Der Transceiver RFM12B ist als SPI-Slave voreingestellt. Die leitungsgebundene Kommunikation mit dem steuernden Mikrocontroller findet über eine Vierdraht-SPI-Schnittstelle im Modus 0 statt. Das höchstwertige Bit eines Bytes wird zuerst übertragen. Die Taktfrequenz der Kommunikation muss kleiner als ein Viertel der Quarzfrequenz sein. Mit einem Quarz, der mit 10 MHz schwingt, kann die Übertragungsrate also höchstens 2,5 Mbit/s erreichen.

Ein Softwaremodul für die Ansteuerung des Transceivers stellt Funktionen für die Verwaltung folgender Pins zur Verfügung: nSEL, nIRQ, nRES und nINT/VDI. Mit einem Low-Puls an dem, als Eingang konfigurierter Pin nINT kann der Transceiver aus dem Standby- in den aktiven Zustand versetzt werden. Als Ausgang wird der Pin VDI (Valid Data Indicator) von dem Transceiver auf High gesetzt wenn die Signalstärke des

empfangenen Signals und/oder die Empfangsqualität die eingestellten Schwellenwerte übersteigen.

Diese Pins werden als Teil einer Datenstruktur codiert, die in der Datei main.h entsprechend der Schaltung aus Abb. 13.4 initialisiert werden:

```
RFM12_pins RFM12_1={ { /*NSEL_DDR*/          &DDRB,
                      /*NSEL_PORT*/        &PORTB,
                      /*NSEL_pin*/         PB2,
                      /*NSEL_state*/       ON} ,

                      /*VDI_DDR*/          &DDRD,
                      /*VDI_PORT*/         &PORTD,
                      /*VDI_PIN*/          &PIND,
                      /*VDI_pin*/          PD6,
                      /*VDI_Pin_Dir*/      VDI_OUT,
                      /*VDI_state*/        OFF,

                      /*NIRQ_DDR*/          &DDRD,
                      /*NIRQ_PORT*/         &PORTD,
                      /*NIRQ_PIN*/          &PIND,
                      /*NIRQ_pin*/          PD2,
                      /*NIRQ_state*/        ON,

                      /*NRES_DDR*/          &DDRD,
                      /*NRES_PORT*/         &PORTD,
                      /*NRES_pin*/          PD6,
                      /*NRES_state*/        ON} ;
```

Mit dem Aufruf der Funktion *RFM12\_Init\_Hard(RFM12\_1)* wird der Mikrocontroller für die Ansteuerung der Pins des Funkmoduls vorbereitet und mit dem nächsten Aufruf:

```
SPI_Master_Init(SPI_INTERRUPT_DISABLE, SPI_MSB_FIRST, SPI_MODE_0, SPI_FOSC_DIV_16)
```

wird der Mikrocontroller als SPI-Master im Modus 0 initialisiert.

### 13.2.4 Der Befehlssatz

Um die gewünschten Einstellungen neu zu laden, legt der Mikrocontroller die Select-Leitung auf Low, überträgt ein Befehlsframe bestehend aus zwei Bytes und schließlich legt er die Leitung wieder auf High. Ein Befehlsframe besteht aus einem Befehlscode, der die höherwertige Bits des Frames belegt, gefolgt von Einstellungsparametern. Mit der Beispiel-Funktion *RFM12\_Write\_CMD* kann der Mikrocontroller einzelne Befehle, die als Parameter übergeben werden an den Transceiver übertragen. Die Funktion benutzt das SPI-Modul das in Kap. 8 vorgestellt ist.

```

void RFM12_Write_CMD(RFM12_pins sdevice_pins, uint16_t uicmd)
{
    uint8_t ucCmdLow, ucCmdHigh;
    ucCmdLow=uicmd;
    ucCmdHigh=uicmd>>8;

    SPI_Master_Start(sdevice_pins.RFM12spi); //CS auf Low
    SPI_Master_Write(ucCmdHigh); //Operation Code wird übertragen
    SPI_Master_Write(ucCmdLow); //die Parameter-Bits werden
    übertragen
    SPI_Master_Stop(sdevice_pins.RFM12spi); //CS auf High
}.

```

Die Zusammensetzung der Befehle und eine kurze Beschreibung werden im Folgenden erläutert.

**Configuration Setting Command** ist der Befehl, mit dem der Transceiver eingestellt wird.

- **Bit 15:8** – bilden den Befehlscode ab: 1000 0000 (0x80).
- **Bit 7** – EL=1 validiert den Sendepuffer
- **Bit 6** – EF=1 validiert den FIFO-Empfangspuffer und schaltet den PCM-Modus ab.
- **Bit 5:4** – mit diesen zwei Bits wird das Frequenzband gewählt: 01–433 MHz, 10–868 MHz, 11–915 MHz.
- **Bit 3:0** – über diese vier Bits wird die Kapazität für den Quarzoszillator eingestellt. Die 16 Kombinationen codieren die Werte zwischen 8,5 pF und 16 pF in Schritten von 0,5 pF.

#### **Power Management Command**

- **Bit 15:8** – Befehlscode: 1000 0010 (0x82).
- **Bit 7** – ER=1 schaltet den Oszillatator und den gesamten Empfänger ein.
- **Bit 6** – EBB=1 schaltet nur den Demodulator ein.
- **Bit 5** – ET=1 schaltet den Oszillatator und den gesamten Sender ein und initiiert das Senden aus dem Sendepuffer. Bei der Initialisierung des Transceivers soll dieses Bit nicht gesetzt sein.
- **Bit 4** – ES=1 schaltet nur den FSK-Modulator ein.
- **Bit 3** – EX=1 schaltet nur den Oszillatator ein.
- **Bit 2** – EB=1 validiert den Unterspannungsdetektor.
- **Bit 1** – EW=1 validiert den internen Timeout-Timer.
- **Bit 0** – DC=0 gibt den Taktausgang frei.

**Frequency Setting Command** Der Befehl dient zur Einstellung der Trägerfrequenz für den Sender, bzw. den Empfänger.

- **Bit 15:12** – Befehlscode: 1010 (0xA).
- **Bit 11:0** – F11:0 codieren die Trägerfrequenz. Die 12-Bit-Zahl F muss zwischen 96 und 3903 liegen, ansonsten wird die Änderung nicht umgesetzt. Abhängig vom gewählten Frequenzband können dadurch Frequenzen zwischen 430,24 MHz und 439,7575 MHz, zwischen 860,48 MHz und 879,515 MHz und zwischen 900,72 MHz oder 929,2725 MHz eingestellt werden. Die Trägerfrequenz kann mit einer Auflösung von 2,5 kHz im 430 MHz-Band, bzw. 5 kHz im 860 MHz-Band eingestellt werden. Für die in Europa zugelassenen Frequenzbänder kann mit Gl. 13.1 der Frequenzcode F einer gültigen Trägerfrequenz  $f_0$  im MHz berechnet werden oder die Trägerfrequenz, wenn der Code bekannt ist.

$$\begin{aligned} \text{430 MHz Band: } & F = 400 \cdot (f_0 - 430) \\ & f_0 = F/400 + 430 \\ \text{860 MHz Band: } & F = 200 \cdot (f_0 - 860) \\ & f_0 = F/200 + 860 \end{aligned} \quad (13.1)$$

**Data Rate Command** Mit diesem Befehl wird die Übertragungsrate in der Luft eingestellt.

- **Bit 15:8** – Befehlscode: 1100 0110 (0xC6).
- **Bit 7** – CS ist ein Parameter für die Berechnung der Bitrate.
- **Bit 6:0** – R6:0 codieren die Bitrate. Für eine vorgegebene Bitrate **BR** in kBit/s rechnet man den Bitrate-Code **R** als Ganzzahl mit der Gl. 13.2

$$R = \frac{10000}{29 \cdot BR \cdot (1 + CS \cdot 7)} \quad (13.2)$$

Um die Abweichung zwischen der gewünschten und der realen Bitrate klein zu halten, setzt man für Bitraten zwischen 4,8 kBit/s und 115,2 kBit/s den Parameter **CS** auf 1.

### Receiver Control Command

- **Bit 15:11** – Befehlscode: 1001 0.
- **Bit 10** – P16 bestimmt die Richtung des Pins nINT/VDI. Wenn das Bit 0 ist, ist der Pin ein Interrupt-Eingang.
- **Bit 9:8** – D1:0 bestimmen den Ausgang des VDI-Pins.
- **Bit 7:5** – I2:0 mit gültigen binären Werten zwischen 001 und 110 wählt man die Bandbreite des Empfängers zwischen 400 kHz und 67 kHz.
- **Bit 4:3** – G1:0 stellen den Verstärkungsfaktor des Eingangsverstärker LNA ein. Mit Werten zwischen 00 und 11 werden Verstärkungsfaktoren von 0 dB, -6 dB, -14 dB und -20 dB eingestellt.

- **Bit 2:0** – R2:0 stellen die RSSI-Schwelle ein. Der Empfänger demoduliert und dekodiert das empfangene Signal wenn die Signalstärke oberhalb der eingestellten RSSI-Schwelle liegt. Mit Werten zwischen 000 und 101 kann die RSSI-Schwelle zwischen  $-103$  dB und  $-73$  dB in Schritten von 6 dB eingestellt werden.

### ***Data Filter Command***

- **Bit 15:8** – Befehlscode: 1100 0010 (0xC2).
- **Bit 7,6** – AL,ML; die zwei Bits beeinflussen die Taktrückgewinnung. Es wird empfohlen beide Bits auf null zu setzen, um eine 2-Byte-Präambel einzustellen.
- **Bit 5, 3** – sind undokumentiert und sollen beide auf 1 gesetzt werden.
- **Bit 4** – S=0 aktiviert den internen digitalen Filter für den SPI-Modus.
- **Bit 2:0** – F2:0 bestimmt die Schwelle für den Datenqualitätsdetektor die größer als vier sein soll.

### ***FIFO and Reset Mode Command***

- **Bit 15:8** – Befehlscode: 1100 1010 (0xCA).
- **Bit 7:4** – F3:0 mit dem Setzen dieser 4-Bit-Zahl auf  $1000_2$  wird nach dem Empfang eines Bytes ein Interrupt ausgelöst. Der Mikrocontroller muss ein Byte aus dem FIFO lesen, um den Überlauf des Empfangspuffers zu verhindern.
- **Bit 3** – SP bestimmt die Länge des Rahmenkennwortes vor den Datenbytes. Wenn SP=1, dann besteht das Rahmenkennwort aus dem Byte 0x2D, ansonsten aus zwei Bytes 0x2DD4.
- **Bit 2** – AL=0 ermöglicht das Füllen des Empfangspuffers erst nach der Erkennung des Rahmenkennwortes, ansonsten werden alle dekodierten Bytes in den Puffer gespeichert.
- **Bit 1** – FF=1 validiert das Füllen des Empfangspuffers nach der Erkennung des Rahmenkennwortes. Wenn das Bit FF zurückgesetzt ist, wird das Speichern in dem Eingangspuffer unterbrochen.
- **Bit 0** – wenn dieses Bit zurückgesetzt ist, wird ein Hardware-Reset ausgelöst wenn die Versorgungsspannung unter 1,6 V fällt und zusätzlich ermöglicht einen Software-Reset. Wenn das Bit gesetzt ist, wird die Schwelle der Versorgungsspannung für das Auslösen des Hardware-Reset auf 0,25 V gesenkt.

### ***Synchron Pattern Command***

- **Bit 15:8** – Befehlscode: 1100 1110 (0xCE).
- **Bit 7:0** – mit diesen acht Bits kann der Befehl das niederwertige Byte des Rahmenkennwortes ersetzen.

### **AFC<sup>13</sup> Command**

- **Bit 15:8** – Befehlscode: 1100 0100 (0xC4).
- **Bit 7:6** – die Bits bestimmen den AFC-Modus. Mit der Kombination 00 wird die automatische Frequenzkorrektur abgeschaltet, was zu gute Ergebnisse geführt hat.
- **Bit 5:4** – wenn man das Frequenzoffsetregister nicht begrenzen will, werden diese zwei Bits zurückgesetzt.
- **Bit 3** – mit dem Setzen dieses Bits wird die letzte gemessene Frequenzabweichung gespeichert.
- **Bit 2** – FI=1 ermöglicht eine bessere Genauigkeit bei der Ermittlung der Frequenzabweichung.
- **Bit 1** – OE=1 validiert das Frequenzoffsetregister.
- **Bit 0** – EN=1 ermöglicht die Berechnung der Frequenzabweichung.

### **TX Configuration Control Command**

- **Bit 15:9** – Befehlscode: 1001 100.
- **Bit 8** – mit MP=0 wird die höhere Sendefrequenz mit "0" und die niedrigere mit "1" codiert
- **Bit 7:4** – (M3:0+1) × 15 kHz bestimmt den Frequenzhub als Betrag der Differenz zwischen der Trägerfrequenz und der Frequenz des modulierten Signals.
- **Bit 3** = 0.
- **Bit 2:0** – über die drei Bits wird die relative Ausgangsleistung des Senders codiert. Als Referenz dient die Ausgangsleistung an einer angepassten Antenne. Mit den Kombinationen von 000 bis 111 werden die relativen Ausgangsleistungen von 0 dB bis −17,5 dB in Schritten von −2,5 dB codiert.

### **PLL Setting Command**

- **Bit 15:8** – Befehlscode: 1100 1100 (0xCC).
- **Bit 7:0** – dieses Byte erhält nach der POR-Phase den Wert 0x77 und wird von dem Hersteller empfohlen.

### **Wake-Up Timer Command**

- **Bit 15:13** – Befehlscode: 111.
- **Bit 12:8** – R4:0 ein Parameter der Werte zwischen 0 und 29 annehmen kann.
- **Bit 7:0** – M7:0.

---

<sup>13</sup>AFC Automatic Frequency Control (en) – automatische Frequenzkontrolle.

Mit dem Setzen des Bits *EW* mit dem Befehl *Power Management* wird ein Interrupt ausgelöst nachdem die Zeit  $T_Z$  verstrichen ist:

$$T_Z/\text{ms} = 1,03 \cdot M \cdot 2^R + 0,5 \quad (13.3)$$

Damit der Interrupt nach der Zeit  $T_Z$  wiederholt ausgelöst wird, muss das Bit *EW* mit dem Befehl *Power Management* zurückgesetzt und wieder gesetzt werden.

**Low Duty-Cycle Command** Um Energie zu sparen, kann der Transceiver als Empfänger aus dem Stand-by-Modus (das Bit *ER* muss mit dem Befehl *Power Management* zurückgesetzt werden) in den aktiven Modus periodisch versetzt werden. Die Periodendauer dieses Zyklus  $T_Z$  wird mithilfe des Befehls *Wake-up Timer* und die Dauer des aktiven Betriebs  $T_{\text{aktiv}}$  mit dem Befehl *Low Duty-Cycle* bestimmt. Auch wenn der Wake-up-Timer aktiv ist, wird durch die Validierung dieses Betriebs kein Interrupt ausgelöst.

- **Bit 15:8** – Befehlscode: 1100 1000 (0xC8).
- **Bit 7:1** – D6:0 bilden ein Parameter für die Berechnung des Tastverhältnisses gemäß der Gl. 13.4 ab.
- **Bit 0** – EN=1 validiert diesen zyklischen Betrieb.

$$\frac{T_{\text{aktiv}}}{T_Z} = \frac{2 \cdot D + 1}{M} \cdot 100 \text{ in \%} \quad (13.4)$$

wobei M der Parameter des Wake-up-Timer ist.

### **Low Battery Detector and Microcontroller Clock Divider Command**

- **Bit 15:8** – Befehlscode: 1100 0000 (0xC0).
- **Bit 7:5** – wenn das Bit DC=0 ist, (siehe den Befehl *Power Management*) bestimmen diese Bits die Frequenz des Taktausgangs zwischen 1 MHz und 10 MHz.
- **Bit 4** – ist null.
- **Bit 3:0** – V3:0 diese Bits legen die Schwelle  $U_S$  für den Unterspannungsdetektor entsprechend der Gl. 13.5 fest.

$$U_S = 2,25 + 0,1 \cdot V \text{ in Volt} \quad (13.5)$$

#### **13.2.5 Das Statusregister**

Das Statusregister ist ein 16 Bit großes Nur-Lese-Register, das Informationen über den Zustand des Transceivers speichert. Beim Auslösen eines Interrupts wird der Interrupt-Ausgang nIRQ auf Low, der SPI-Datenausgang SDO auf High und ein Bit aus dem Bereich 15:10 im Statusregister gesetzt, durch das die Quelle des Interrupts identifiziert wird. Das Löschen der meisten Interrupts erfolgt durch das Lesen des Statusregisters.

Damit wird der Ausgang nIRQ auf High gesetzt und das entsprechende Interrupt-Bit zurückgesetzt.

Wenn der Transceiver eine logische null als erstes Bit eines Befehls erkennt, schiebt er bitweise auf die SDO-Leitung die 16 Bit des Statusregisters, die folgende Bedeutung haben:

- **Bit 15** – zeigt, wenn es gesetzt ist, dass der Sender für den Empfang über SPI eines weiteren Sendebytes bereit ist; der Interrupt wird durch das Übertragen eines Bytes oder nach dem letzten Byte eines Frames durch das Zurücksetzen des Bits *ET* mit dem Befehl *Power Management* gelöscht. Wenn dieses Bit beim Empfänger gesetzt ist, zeigt an, dass die empfangene Anzahl von Bits im FIFO-Puffer die eingestellte Schwelle (meist 8 Bit) erreicht hat. In diesem Fall wird der Interrupt durch das Lesen eines Bytes aus dem Puffer gelöscht.
- **Bit 14** – wird nach erfolgreichem Beenden der POR-Phase gesetzt.
- **Bit 13** – wird beim Überlauf des Sende-, bzw. Empfangspuffers gesetzt.
- **Bit 12** – wird “1“ beim Überlauf des Wake-up-Timers, vorausgesetzt der Interrupt wurde durch das Setzen des Bits *EW* mit dem Befehl *Power Management* freigegeben.
- **Bit 11** – das Detektieren eines Low-Pulses an dem Eingang nINT löst einen Interrupt aus und setzt dieses Bit.
- **Bit 10** – wird “1“, wenn der Unterspannungsdetektor aktiviert ist und die Versorgungsspannung unter dem eingestellten Wert fällt.
- **Bit 9** – wird gesetzt, wenn der Empfangspuffer leer ist.
- **Bit 8** – wird gesetzt, wenn an der Sender-Antenne ein genug starkes Signal detektiert wurde, oder die empfangene Signalstärke auf der Receiver-Seite oberhalb der eingestellten RSSI-Schwelle ist.
- **Bit 7** – Ausgang des Signalqualität-Detektors.
- **Bit 6** –= “1“, wenn das rekonstruierte Clock-Signal verriegelt ist.
- **Bit 5** – wird mit jedem AFC-Messzyklus der empfangenen Trägerfrequenz umgeschaltet.
- **Bit 4** – bildet das Vorzeichen der Differenz der Trägerfrequenzen des Senders und Empfängers ab.
- **Bit 3:0** – entsprechen den niederwertigen Bits der gemessenen Differenz zwischen den Trägerfrequenzen. Diese vorzeichenbehaftete Differenz kann auf der Empfängerseite zu dem Trägerfrequenz-Code addiert werden.

### 13.2.6 Initialisierung des Transceivers

Nach dem Initialisierungsvorgang soll gewährleistet sein, dass der Transceiver bereit ist, seine Aufgabe zu erfüllen. Mit dem Aufruf der Funktion *RFM12\_Init* wird die Leitung nRES für 10 ms auf Low gelegt und damit ein Hardware-Reset initiiert. Es wird mindestens 100 ms gewartet und anschließend wird der Status der POR-Phase durch das

Testen des Bit 14 vom Statusregister abgefragt. Wenn diese Phase erfolgreich beendet wurde, werden die aktuellen Konfigurationsparameter, die in einem Array gespeichert sind, übertragen und in den Register-Block abgelegt. Als Übergabeparameter fordert die Funktion die Variable *ucsens*, um zu bestimmen, ob der Transceiver als Sender oder Empfänger arbeitet sowie die codierten Werte für die Funkrate und die Trägerfrequenz festzulegen. Die Pinbeschaltung des Transceivers wird mit der Datenstruktur *sdevice\_pins* als Parameter übermittelt. Die Funktion gibt den Wert INIT\_NOK zurück, wenn sie erfolglos war und muss in diesem Fall erneut aufgerufen werden.

```

uint8_t RFM12_Init(RFM12_pins sdevice_pins, uint8_t ucsens, //  

ucsens=TX oder RX  

                    uint8_t ucbaudrate, //ucbaudrate entspricht  

                    der codierten Funkrate  

                    uint16_t uicenter_freq) //Trägerfrequenz  

{  

    uint16_t uiInitStatus;  

    uint16_t uiCmd_RX[14]={0x80D8, 0x8289, 0x9420,  

    0xC22C, 0xCA81, 0xCED4, 0xC407,  

    0x9850, 0xE000, 0xCC77, 0xC800, 0xC000, (0xC600 | ucbaudrate),  

    (0xA000 | uicenter_freq)};  

    uint16_t uiCmd_TX[14]={0x80D8, 0x8209, 0x9420, 0xC22C,  

    0xCA81, 0xCED4, 0xC4C7,  

    0x9850, 0xE000, 0xCC77, 0xC800, 0xC000, (0xC600 | ucbaudrate),  

    (0xA000 | uicenter_freq)};  

    //der Reset-Eingang des Transceivers wird auf Low gesetzt  

    RFM12_Set_NRESLow(sdevice_pins);  

    TimeOut_Start(DELAY_YES); //die Wartezeit wird gestartet  

    while(!TimeOut_Get_State()); //es werden 10 ms gewartet  

    RFM12_Set_NRESHigh(sdevice_pins); //der Reset-Pin des  

    Transceivers wird auf High gesetzt  

    /*der Transceiver braucht nach dem Reset mindestens 100 ms, um  

    einen stabilen Zustand zu erreichen*/  

    for(uint8_t i=0; i<20; i++)  

    {  

        TimeOut_Start(DELAY_YES);  

        while(!TimeOut_Get_State());  

    }  

    uiInitStatus=RFM12_Read_StatusReg(sdevice_pins); //das  

    Statusregister wird gelesen  

    if(uiInitStatus & 0x4000) //wenn wahr, dann erfolgreiche  

    POR-Phase  

    {  

        for(uint8_t i=0; i<14; i++)  

        {

```

```

        //die neuen Einstellungen werden übertragen
        if(ucsens==TX) RFM12_Write_CMD(sdevice_pins,
            uiCmd_TX[i]);
        else RFM12_Write_CMD(sdevice_pins,
            uiCmd_RX[i]);
    }
    return INIT_OK;
}
return INIT_NOK;
}

```

Wenn flexiblere Einstellmöglichkeiten gewünscht sind, dann sollten wie im folgenden Beispiel die Übergabeparameter in einer Struktur definiert und die Initialisierungsfunktion entsprechend angepasst werden.

```

typedef struct{
    uint8_t ucFreqBand; //Frequenzband: 315 MHz, 430 MHz, 866 MHz,
    915 MHz
    uint8_t ucSense; //Richtung Sender oder Empfänger
    uint16_t uiFreq; //Trägerfrequenz: Code für die
    Trägerfrequenz; darf Werte zwischen
    //96 und 3902 annehmen. Dieser Wert wird folgendermaßen
    berechnet:
    //433 MHz Band: uicenter_freq=(F / MHz - 430): 0,0025
    //868 MHz Band: uicenter_freq=(F / MHz - 860): 0,005
    uint8_t ucBaudrate; //Funkrate
    uint8_t ucRecv_BW; //Bandbreite
    uint8_t ucLNA_gain; //LNA-Verstärkungsfaktor
    uint8_t ucRSSI_Tresh; //RSSI-Schwelle
    uint8_t ucModparam; //Modulationsparameter
    uint8_t ucOutPow; //Ausgangsleistung
} RFM12Param;

```

### 13.2.7 Daten senden

Der Transceiver wird als Transmitter durch das Setzen des Bits *ET* und das Rücksetzen des Bits *ER* mit dem Befehl *Power Management* eingestellt. Im SPI-Sendemodus werden über Funk die Bytes, die im Sendepuffer gespeichert sind, gesendet. Dieser Puffer wird mit dem Setzen des Bits *EL* (siehe *Configuration Setting Command*) freigegeben. Der Mikrocontroller überträgt zuerst den Befehlscode 0xB8 und danach die zu sendenden Bytes, die in den Puffer gespeichert werden. Das Senden des ersten Bytes einer Nachricht beginnt mit dem Setzen des Bits *ET* mit dem Befehl *Power Management*. Der Transceiver signalisiert dem Mikrocontroller das vollständige Senden eines Bytes durch

**Tab. 13.1** Sendeframe

Sendebefehlscode 0xB8	Header-Segment		Payload-Segment n-Datenbytes + eventuelle CRC- Prüfsumme	Trailer- Segment z. B. 0xAAAA
	Präambel 0xAAAA	Rahmenkennwort z. B. 0x2DD4		

das Setzen des Ausgangs nIRQ auf Low, bzw. des Ausgangs SDO auf High. Der Mikrocontroller überträgt ein weiteres Byte und setzt damit den ausgelösten Interrupt zurück. Nach der Übertragung des letzten Bytes wird mit dem Rücksetzen des Bit *ET* das Senden beendet und der Interrupt gelöscht. Um eine Folge von Datenbytes zu senden, muss der Mikrocontroller ein Sendeframe wie in Tab. 13.1 über SPI an den Transceiver übertragen.

Die Präambel besteht aus zwei oder mehrere Bytes mit abwechselnden Bits (0xAA) und dient zur Takt synchronisation auf der Empfängerseite. Wenn der Empfänger die Erkennung des Rahmenkennwortes aktiviert hat, werden nur die Bytes ab dem Payload-Segment in den FIFO-Puffer gespeichert. Das Trailer-Segment besteht aus Dummy-Bytes und wird benötigt, um zu gewährleisten, dass alle Payload-Bytes vollständig gesendet wurden wenn das Bit *ET* nach der SPI-Übertragung des letzten Trailer-Bytes zurückgesetzt wird.

Mit der Funktion *RFM12\_Send\_Array* wird ein gesamtes Payload-Segment gesendet. Der Funktion wird die Adresse des Arrays, das die Payload-Bytes speichert, zusammen mit der Länge des Arrays als Aufrufparameter übergeben.

```
void RFM12_Send_Array(RFM12_pins sdevice_pins, uint8_t *ucarray2send,
                      uint8_t ucarray_length)
{
    uint8_t ucSendHeader[4] = {0xAA, 0xAA, 0x2D, 0xD4}; /*dient
    zur Synchronisation des Empfängers und zur Übertragung des
    Synchronisations Bitmuster für das Füllen von FIFO*/
    uint8_t ucSendTrailer[2] = {0xAA, 0xAA}; /*werden benutzt, um
    die letzten Payload-Bytes aus dem Sende puffer zu senden*/
    //mit dem Setzen des Bits ET beginnt das Senden
    RFM12_Write_CMD(sdevice_pins, (POWER_MANAGMT_CMD | EX | DC
    | ET));
    //es wird auf den Interrupt gewartet
    while((RFM12_Get_NIRQState(sdevice_pins)));
    SPI_Master_Start(sdevice_pins.RFM12spi);
    //CS wird auf Low gesetzt
    SPI_Master_Write(0xB8); //Übertragung des Sende-Befehlscode
    //der Header wird übertragen
    for(uint8_t i=0; i<4; i++)
    {
        SPI_Master_Write(ucSendHeader[i]);
    }
}
```

```

        while((RFM12_Get_NIRQState(sdevice_pins)));
    }
    //die Payload-Bytes werden übertragen
    for(uint8_t i=0; i<ucarray_length; i++)
    {
        SPI_Master_Write(ucarray2send[i]);
        while((RFM12_Get_NIRQState(sdevice_pins)));
    }
    //der Trailer wird übertragen
    for(uint8_t i=0; i<2; i++)
    {
        SPI_Master_Write(ucSendTrailer[i]);
        while((RFM12_Get_NIRQState(sdevice_pins)));
    }
    SPI_Master_Stop(sdevice_pins.RFM12spi); //CS wird auf High
    gesetzt
    //der Sender wird deaktiviert
    RFM12_Write_CMD(sdevice_pins, ((POWER_MANAGMT_CMD | EX | DC) &
    ET_NOT));
}

```

### 13.2.8 Lesen der empfangenen Daten

Um Nachrichten zu empfangen, muss ein Transceiver RFM12B als Receiver durch das Setzen des Bits *ER* mit dem Befehl *Power Management* eingestellt werden. Zusätzlich muss der FIFO-Empfangspuffer aktiviert sein und das gleiche Rahmenkennwort wie auf der Senderseite eingestellt werden. Das Speichern der empfangenen Bytes wird erst nach der Erkennung des Rahmenkennwortes validiert und der Parameter F3:0 wird mit dem Befehl *FIFO and Reset Mode* auf acht eingestellt. Mit dem Setzen des Bits *FF* wird jedes empfangene Payload-Byte gespeichert und dabei ein Interrupt ausgelöst.

Der Mikrocontroller überwacht die nIRQ-Leitung und wenn diese auf Low gesetzt ist, liest der Mikrocontroller das Statusregister und prüft ob der Empfang eines Bytes den Interrupt ausgelöst hat. Der Mikrocontroller überträgt über SPI den Befehlscode *FIFO Read* (0xB0) gefolgt von einem Dummy-Byte, um ein Byte aus dem FIFO-Puffer zu lesen und einen Überlauf zu verhindern. Damit wird der Interrupt gelöscht und die Leitung nIRQ wieder auf High gesetzt. Dieser Vorgang wird wiederholt bis alle Bytes des Payload-Segments gelesen werden. Am Ende wird das Bit *FF* zurückgesetzt, um das Speichern in den Puffer zu sperren.

Die beschriebene Vorgehensweise ist in der Funktion *RFM12\_Read\_Array* implementiert.

```
void RFM12_Read_Array(RFM12_pins sdevice_pins, uint8_t *ucarray2rcvd,
                      uint8_t ucarray_length)
{
    uint16_t uiStatusReg; //Inhalt des Statusregisters
    uint8_t ucInd=0; //Laufvariable
    //das Speichern in den FIFO-Puffer wird freigegeben
    RFM12_Write_CMD(sdevice_pins, (FIFO_RESET_MODE_CMD | FIFO_INT_
    LEVEL | FF));
    while(ucInd<ucarray_length)
    {
        while(RFM12_Get_NIRQState(sdevice_pins)); //Warten auf
        einen Interrupt
        uiStatusReg=RFM12_Read_StatusReg(sdevice_pins); //
        Lesen des Statusregisters
        if(uiStatusReg & 0x8000) //wahr, wenn ein Payload-Byte
        empfangen wurde
        {
            /*das Payload-Byte wird gespeichert und der
            Interrupt gelöscht*/
            ucarray2rcvd[ucInd]=RFM12_Read_
            RecvByte(sdevice_pins);
            ucInd++;
        }
    }
    //das Speichern in den FIFO-Puffer wird gesperrt
    RFM12_Write_CMD(sdevice_pins, ((FIFO_RESET_MODE_CMD | FIFO_
    INT_LEVEL) & FF_NOT));
}
```

---

## 13.3 Funkprotokolle im 2,4-GHz-ISM-Band

### 13.3.1 Bluetooth

Bluetooth ist ein Funkprotokoll basierend auf dem IEEE 802.15.1-Standard [8], 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.

Das Protokoll hat sich im Lauf der Jahre entwickelt und Bluetooth hat sich als Standard etabliert, der den Weg in Massenprodukte wie Computer aller Art und Handys gefunden hat.

Ab der Version 4 des Protokolls wurde neben dem BR/EDR–Modus<sup>14</sup> der LE–Modus<sup>15</sup> definiert. Im LE–Modus 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 [11]. Es können keine Verbindungen zwischen Geräten, die mit unterschiedlichen Modi arbeiten, realisiert werden.

Bluetooth definiert *Profile* als 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. Außerdem werden *Services* (Dienste) als Datenstrukturen definiert, die zur Charakterisierung einer Funktion oder Eigenschaft eines Gerätes dienen.

### 13.3.1.1 Bitübertragungsschicht [12, 15]

Die Übertragung findet im 2,4-GHz-ISM-Band zwischen 2402 MHz und 2480 MHz statt. Im BR/EDR–Modus werden 79 Funkkanäle mit einer Bandbreite und einem Abstand von 1 MHz definiert. Im BR–Modus werden die Daten mit einer gaußschen Frequenzumtastung<sup>16</sup> moduliert und eine Bruttoübertragungsrate von 1 MBit/s wird spezifiziert. Um höhere Bitraten von 2 MBit/s oder sogar 3 MBit/s bei gleichem Frequenzkanalabstand zu erreichen, wird die Phasenumtastung<sup>17</sup> benutzt. 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.

Im LE–Modus werden 40 Funkkanäle mit einer Bandbreite von 2 MHz definiert und die gaußsche Frequenzumtastung für die Signalmodulation verwendet. Drei von diesen Funkkanälen sind für Advertising reserviert, eine Prozedur die von einem Gerät verwendet wird, um regelmäßig unidirektionale Botschaften zu senden. Die weiteren 37 Funkkanäle sind Datenkanäle. In der Version 4 des Protokolls wird für den LE–Modus eine Übertragungsrate von 1 MBit/s (LE 1M) spezifiziert.

---

<sup>14</sup>BR/EDR engl. Basic Rate /Enhanced Data Rate.

<sup>15</sup>LE engl. Low Energy.

<sup>16</sup>engl. Frequency Shift Keying (FSK).

<sup>17</sup>engl. Phase Shift Keying (PSK).

Ab der Version 5 werden für den LE-Modus weitere Übertragungsraten und eine weitere Leistungsklasse 1,5 mit 10 dBm (10 mW) spezifiziert. Im LE 2M-Modus können die Daten mit 2 MBit/s, bzw. im LE Coded-Modus mit 500 kBit/s oder 250 kBit/s übertragen werden. Die Übertragungsrate kann jetzt abhängig von den Anforderungen umgeschaltet werden. Eine Reduzierung der Übertragungsrate ermöglicht eine Leistungsreduktion bei gleich bleibender Reichweite. Um eventuelle Übertragungsfehler zu detektieren, werden die Bluetooth-Nachrichten in allen Versionen mit einer 24-Bit-CRC-Prüfsumme ergänzt. Durch die Einführung einer fehlerkorrigierenden Datencodierung kann die Reichweite im LE-Coded-Modus bei gleich bleibender Sendeleistung erhöht werden. Ab der Version 5.2 ist auch im LE-Modus die Anpassung der Sendeleistung möglich.

Um Interferenzen mit anderen Protokollen aus dem 2,4-GHz-ISM-Band zu vermeiden, benutzt Bluetooth ab der Version 1.2 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 mithilfe 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 fünf Taktperioden der Funkkanal gewechselt. Es finden 1600 Frequenzsprünge in der Sekunde statt.

### 13.3.1.2 Kommunikationstopologien

Bluetooth wurde entwickelt, um eine Point-to-Point Kommunikation zu ermöglichen. Das gilt auch für die sternförmige Bluetoothnetze die Piconet genannt werden und aus einem Master und bis zu sieben Slaves bestehen. 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. Die Verbindung zwischen Geräten in Reichweite kann ad-hoc stattfinden, also ohne Verabredung. Folgende Regeln gelten beim Aufbau der Piconetze:

- in jedem Piconetz darf nur ein Master sein;
- ein Gerät darf Teil zweier Piconetze sein;
- ein Gerät darf nur in einem Piconetz Master sein.

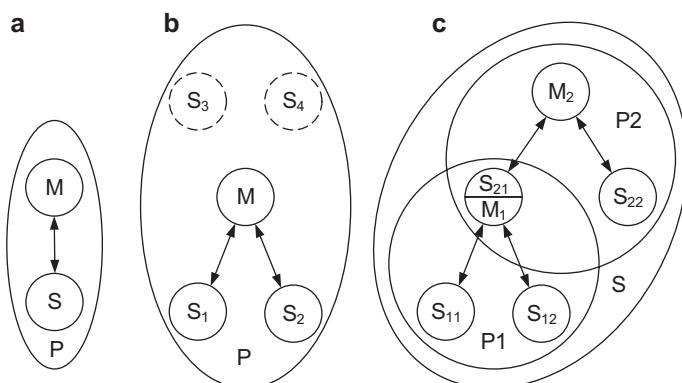
### 13.3.1.2.1 BR/EDR-Topologien

In einem BR/EDR-Piconet kommunizieren alle Teilnehmer über den gleichen physikalischen Übertragungskanal. Sie verwenden einen gemeinsamen Takt zur Synchronisation der Kommunikation und eine gemeinsame Frequenzsprungfolge. Die Kommunikation kann nur zwischen dem Master und dem adressierten Slave stattfinden. Die Kommunikation zwischen den Slaves sowie der Rundruf sind nicht vorgesehen.

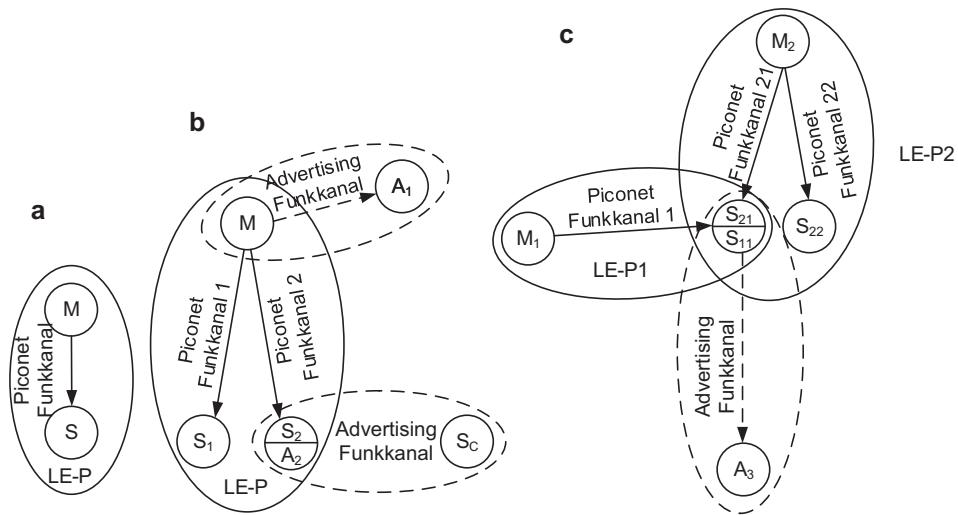
Durch die Teilnahme eines Gerätes an zwei oder mehreren Piconetzen entsteht ein Scatternet wie in Abb. 13.5 aber kein Meshnetz weil die Spezifikation des Protokolls das Weiterleiten des Datenflusses von einem Piconetz zum anderen nicht erlaubt. Falls gewünscht, kann der Datenaustausch zwischen Piconetzen auf einer anderen Ebene geregelt werden. Unterschiedliche Piconetze können in unmittelbarer Nähe koexistieren, müssen aber unterschiedliche Übertragungskanäle benutzen. Bis zur Version 5 von Bluetooth konnten neben den sieben aktiven Slaves bis zu 255 inaktive (geparkte) Slaves, die aber mit dem Master synchronisiert waren. Der Master kann den Zustand (aktiv oder inaktiv) der mit ihm synchronisierten Slaves ändern. Ab der Version 5 wurde der „Park“ Zustand aus der Spezifikation entfernt.

### 13.3.1.2.2 LE-Topologien [15]

In der Abb. 13.6 sind die LE-Topologien entsprechend der Version 5.2 von Bluetooth dargestellt. In einem LE-Piconetz kommuniziert jeder Slave mit dem Master über einen anderen Übertragungskanal. Der Master stellt eine Point-to-point-Verbindung mit einem Slave her, oder erreicht alle Slaves des Piconetzes mit einem Rundruf. Die Übertragung von Daten zwischen Piconetzen ist ab der Version 5.1 erlaubt, so dass Bluetooth-Meshnetze mit großer Reichweite möglich sind.



**Abb. 13.5** Bluetooth-BR/EDR-Netztopologien. a Piconetz mit einem Slave, b Piconetz mit zwei aktiven ( $S_1$  und  $S_2$ ) und 2 inaktiven ( $S_3$  und  $S_4$ ) Slaves, c Scatternet (S) bestehend aus den Piconetzen P1 und P2



**Abb. 13.6** Bluetooth-LE-Netztopologien. a Piconetz mit einem Slave, b Piconetz mit zwei Slaves, ein Advertiser und ein Scanner, c Scatternetz, bestehend aus den Piconetzen LE-P1 und LE-P2 und einem Advertiser

In Abb. 13.6b ist ein LE-Piconetz dargestellt, das aus einem Master und zwei Slaves besteht. Die Kommunikation findet über unterschiedliche Funkkanäle statt. Der Master möchte das Piconetz erweitern. In der Rolle des Initiators horcht (scannt) er auf ein Advertising-Funkkanal nach Advertiser (in diesem Fall A<sub>1</sub>) in der Reichweite, die ihre Verbindungsbereitschaft senden. Der Slave S<sub>2</sub> sendet als Advertiser und seine Botschaften werden von dem Scanner S<sub>C</sub> empfangen. Eine Verbindung (connection) zwischen den zwei Geräten besteht nicht und ist auch nicht angestrebt.

In Abb. 13.6c ist ein LE-Scatternetz, bestehend aus den Piconetzen LE-P1 und LE-P2 dargestellt. Der Slave S<sub>21</sub>/S<sub>11</sub> ist Teil beider Piconetzen und strebt an ein Piconetz als Master mit dem Advertiser A<sub>3</sub>, der seine Verbindungsbereitschaft signalisiert hat, aufzubauen.

### 13.3.1.3 Aufbau eines Piconetzes

Ein Gerät, das dafür konfiguriert wurde, ein Piconetz aufzubauen, sucht nach anderen Geräten in Reichweite. Die Suche kann nach beliebigen oder vorab durch ihre Adresse definierten Geräten stattfinden. 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 Verbindungsauflauf akzeptiert.

### 13.3.1.3.1 Aufbau eines BR/EDR-Piconetzes

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;
- 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.

### 13.3.1.3.2 Aufbau eines LE-Piconetzes

- **Advertising** ist eine Prozedur die von einem BLE<sup>18</sup>-Gerät verwendet wird, um unidirektionale Datenpakete auf einem Advertiser-Funkkanal zu definierten Zeitschlitzten, mit einer vordefinierten Datenstruktur zu senden. Die Prozedur kann auch für den Aufbau einer Verbindung mit einem Gerät als Initiator verwendet werden. Der Initiator ist ein Gerät, das eine Verbindung mit einem verbindungsfähigen Advertiser versucht. Ein Initiator kann ein Master, ein Slave, oder ein Gerät das noch keine Verbindung aufweist, sein.
- **Scanning** ist eine Prozedur, die von einem BLE-Gerät verwendet wird, um Advertising-Datenpakete zu empfangen. Ein Scanner kann zusätzliche Daten anfordern, die von dem Advertiser übermittelt werden.
- **Discovering** ist eine Prozedur die von einem BLE-Gerät verwendet wird, um andere Geräte in Reichweite zu suchen, bzw. seine Anwesenheit anderen Geräten zu signalisieren. Die Geräte verwenden dafür die Advertising- oder die Scanning-Prozedur.
- **Connecting** ist die Prozedur durch die, die Verbindung zwischen zwei BLE-Geräte aufgebaut wird. Während dieser Prozedur ist ein Gerät Scanner, während das andere ein Advertiser ist. Der Advertiser muss eine Verbindungsbereitschaft senden.

### 13.3.1.4 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 Verbindungsaufforderung akzeptiert wurde.

---

<sup>18</sup>BLE – Bluetooth Low Energy.

- Modus 3 (link level enforced security). In diesem Modus wird die Authentifizierung während des Verbindungsaufbaus 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 [12] 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.

### 13.3.2 ZigBee

ZigBee wurde von der ZigBee Alliance als Funkprotokoll für die Vernetzung von Sensor-knoten 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.

#### 13.3.2.1 ZigBee-Geräte

Abhängig von der Komplexität der Geräte unterscheidet ZigBee zwischen FFD<sup>19</sup>- und RFD<sup>20</sup>-Geräten (siehe Abb. 13.7). 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.

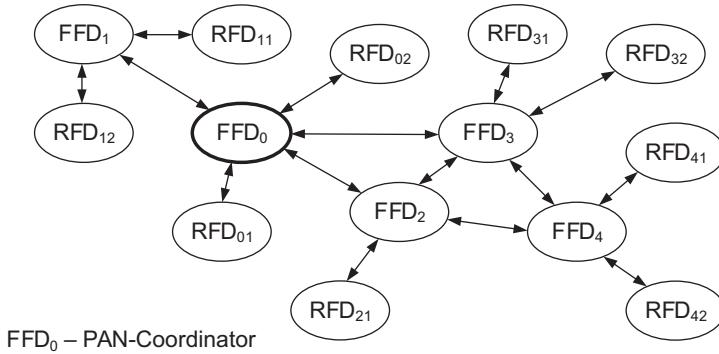
Ein ZigBee-Gerät kann in einem Netzwerk folgende Rolle haben:

- 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.

---

<sup>19</sup>FFD – Full Function Device.

<sup>20</sup>RFD – Reduced Function Device.



**Abb. 13.7** ZigBee-Netzwerk

- 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:

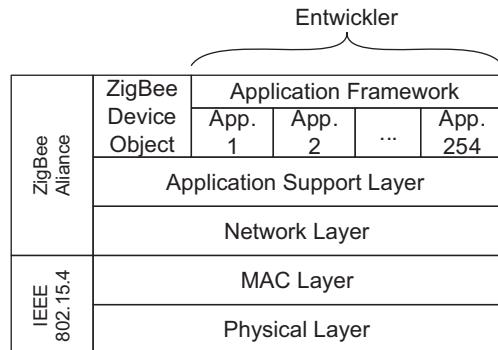
- Green Power Devices: Das sind Geräte mit einem extrem niedrigen Energieverbrauch, der über Energy Harvesting oder lebenslang von einer Batterie abgedeckt werden kann.
- End-Devices befinden sich überwiegend im Sleep-Modus und werden über eine austauschbare Batterie versorgt.
- Router / Coordinator müssen die ganze Zeit aktiv sein, um die Nachrichten weiterzuleiten und werden über das Netz versorgt.

### 13.3.2.2 Protokoll-Aufbau

Abb. 13.8 stellt das Schichten-Modell der ZigBee Version 3 dar [13]. Die Bitübertragungsschicht und die Medienzugriffssteuerung des Protokolls werden vom IEEE 802.15.4 Standard spezifiziert, während die oberen Schichten von der ZigBee Alliance spezifiziert sind. Im Vordergrund stehen die Standardisierung des Protokolls auf allen Ebenen, sowie der Energieverbrauch.

### 13.3.2.3 Bitübertragungsschicht

Das ZigBee-Protokoll definiert 27 Kommunikationskanäle (siehe Tab. 13.2). 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

**Abb. 13.8** ZigBee 3 – Schichten-Modell**Tab. 13.2** 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				

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 zugeteilt um den Energieverbrauch zu reduzieren.

### 13.3.2.4 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 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.

### 13.3.2.5 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.

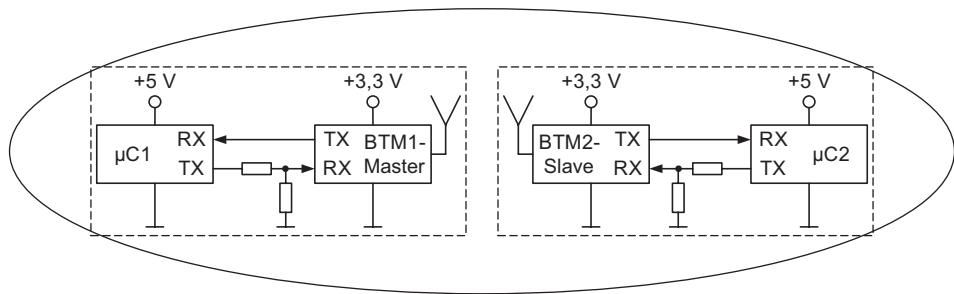
### 13.3.2.6 Application Framework

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.



**Abb. 13.9** Piconet mit zwei Mikrocontrollern

## 13.4 Bluetooth® Kommunikation mit dem serial profile

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. 13.9) mehrere seriellen Schnittstellen wie USB, UART, SPI, PCM zur Verfügung.

Für die bilaterale Kommunikation bietet sich das SPP<sup>21</sup>-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 Kommunikationsparameter sind auf acht Datenbit, keine Parität, ein Stopppbit 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 Kap. 7 vorgestellten Funktionen benutzt werden.

<sup>21</sup>SPP – Serial Port Profile.

### 13.4.1 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.

### 13.4.2 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 [5]. 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) [6] 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 „CR<sup>22</sup>“ akzeptiert.

#### Beispiel

„ATL1\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“. ◀

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.

<sup>22</sup>Im ASCII Code mit 0x0D codiert.

### 13.4.3 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 Stopbits) 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 Kap. 7 vorgestellte Funktion UART\_WriteBuffer verwendet.

#### 13.4.3.1 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.

### 13.4.3.2 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, wird
    gespeichert
    UART_WriteBuffer((uint8_t*) "ATD=00126F250A6A\r\0",18);
    delay(100); //100 ms Warten um den vorigen Befehl auszuführen
}
```

---

## Literatur

1. Yang, S.-H. (2014). *Wireless sensor networks. Principles, design and applications*. Springer.
2. Microchip: AT86RF231 Low Power 2.4 GHz Transceiver for ZigBee, IEEE 802.15.4, 6LoWPAN, RF4CE, SP100, Wireless HART and ISM Applications. <https://www.microchip.com/wwwproducts/en/AT86RF231> bzw. <http://ww1.microchip.com/downloads/en/devicedoc/doc8111.pdf>. Zugegriffen: 4. Apr. 2021.
3. Nordic Semiconductor. nRF24L01+ Single Chip 2.4 GHz Transceiver. Product Specification v.1.0. <https://www.nordicsemi.com/Products/Low-power-short-range-wireless/nRF24-series>. Zugegriffen: 4. Apr. 2021.
4. Microchip Technology Inc. MRF24J40 Data Sheet. IEEE 802.15.4<sup>TM</sup> 2.4 GHz RF Transceiver. [www.microchip.com](http://www.microchip.com) bzw. <https://ww1.microchip.com/downloads/en/DeviceDoc/39776C.pdf>. Zugegriffen: 4. Apr. 2021.
5. Rayson Technology co. Ltd. BTM-230 Data sheet. BC04-EXT Class1 Module BTM-230. [http://www.rayson.com/rayson/en/upload/prod\\_file/NP0019\\_1.pdf](http://www.rayson.com/rayson/en/upload/prod_file/NP0019_1.pdf). Zugegriffen: 4. Apr. 2021.
6. International Telecommunication Union. ITU-T V.250. Serial asynchronous automatic dialing and control. <https://www.itu.int/rec/T-REC-V.250/de>. Zugegriffen: 4. Apr. 2021.
7. VFG 30/2014, geändert mit Vfg 36/2014, geändert mit Vfg 69/2014. [https://www.bundesnetzagentur.de/DE/Sachgebiete/Telekommunikation/Unternehmen\\_Institutionen/Frequenzen/Allgemeinzuteilungen/allgemeinzuteilungen-node.html](https://www.bundesnetzagentur.de/DE/Sachgebiete/Telekommunikation/Unternehmen_Institutionen/Frequenzen/Allgemeinzuteilungen/allgemeinzuteilungen-node.html). Zugegriffen: 4. Apr. 2021.
8. IEEE Std 802.15.1<sup>TM</sup>-2002. Wireless Medium Access Control (MAC) and Physical Layer (PHY) specifications for Wireless personal Area Networks (WPANs). <http://ieeexplore.ieee.org>. Zugegriffen: 4. Apr. 2021.

9. 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. [www.bundesnetzagentur.de](http://www.bundesnetzagentur.de). Zugegriffen: 4. Apr. 2021.
10. Beuth, K., Breide, S., Lüders, C., Kurz, G., Hanebuth R. (2009). Nachrichtentechnik. Vogel Industrie Medien GmbH & Co. KG, Würzburg.
11. Heydon, R. (2013). *Bluetooth low energy – The developer's handbook*. Prentice Hall.
12. Bluetooth. (2014). Specification of the bluetooth system core 4.2. [www.bluetooth.com](http://www.bluetooth.com). . Zugegriffen: 4. Apr. 2021.
13. ZigBee 3: Präsentation. (2014). <https://zigbeealliance.org/solution/zigbee/>. Zugegriffen: 4. Apr. 2021.
14. HOPE MICROELECTRONICS CO., LTD. RFM12B – Universal ISM Band FSK Transceiver. [www.hoperf.com](http://www.hoperf.com). . Zugegriffen: 10. Juli 2021.
15. Bluetooth SIG. Bluetooth core specification, Revision v5.2. [www.bluetooth.com](http://www.bluetooth.com). Zugegriffen: 15. Febr. 2021.



# Sensorik Systemtechnische Überlegungen

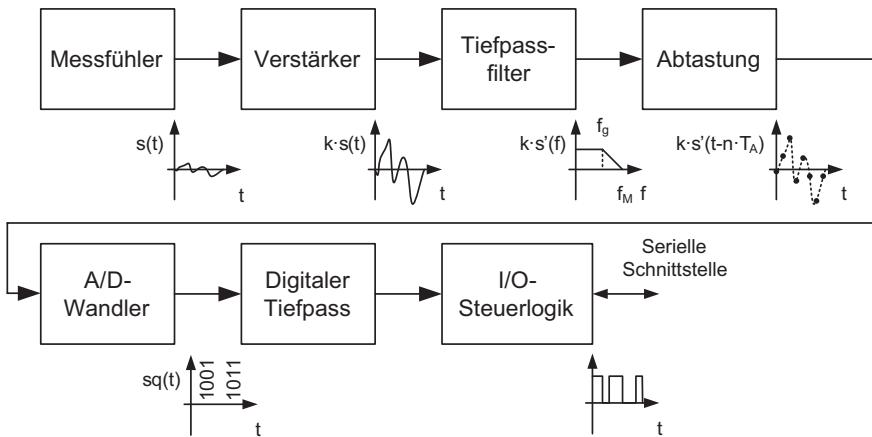
14

## Zusammenfassung

In diesem Kapitel werden einige systemtechnische Begriffe geklärt, wie Abtastung, Quantisierung, digitale Filter. Außerdem gibt es einige softwaretechnische Tricks zur Abstraktion der Hardware-Pins und zur Verwendung von Integer-Arithmetik.

Ein Sensor besteht in der Regel aus einem oder mehreren analogen 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. Bilden der Sensor und die Digitalisierung eine Einheit, spricht man von digitalen Sensoren. Es ist wichtig, den Verarbeitungspfad der Digitalisierung zumindest in Ansätzen zu verstehen, um die Qualität und Aussagekraft des aufbereiteten Signals einschätzen zu können.

Moderne Technologien, insbesondere die Mikrosystemtechnik (MEMS) und die Hochintegration unterschiedlicher Halbleiterstrukturen ermöglichen, eine immer größere Anzahl von Sensoren in immer mehr Geräten einzubauen beziehungsweise in ein einziges Gehäuse 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



**Abb. 14.1** Digitaler Sensor – Blockschaltbild

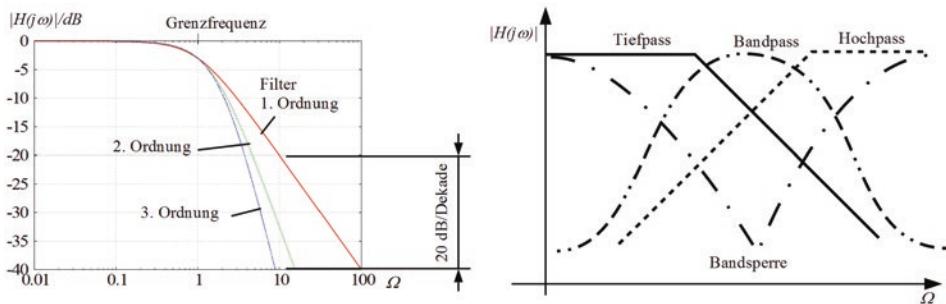
entstanden. Ein beispielhaftes Blockschaltbild eines solchen Sensors ist in Abb. 14.1 zu sehen. Eventuelle Störungen auf der analogen Seite des Sensors werden wegen der extrem kurzen Leitungen auf ein Minimum reduziert. Die Messwerte werden in digitaler Form mit auswählbarer Auflösung auf einem Datenbus, der der Vernetzung mehrerer Schaltungskomponenten dient, zur Verfügung gestellt.

## 14.1 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:

$$2 \cdot f_M \leq f_a = \frac{1}{T_a}, \quad (14.1)$$

wobei  $n$  eine natürliche Zahl und  $f_a$  die Abtastfrequenz ist. Die mit der Gl. 14.1 beschriebene Forderung ist als Abtasttheorem oder Nyquist-Kriterium bekannt und die Frequenz  $f_a/2$  wird in der Literatur als Nyquist-Frequenz bezeichnet. In der Praxis ist dieser Wert kaum haltbar, d. h. in der Regel wird man erheblich über dieser Frequenz abtasten, in der Regel etwa sieben bis zehnmal höher als die Grenzfrequenz des zu messenden Signals. Um diese Grenzfrequenz auf den relevanten Spektralbereich des analogen Signals zu limitieren, wird das Signal vor der Abtastung mit einem Tiefpassfilter begrenzt, wodurch die irrelevanten Anteile des analogen Signals entfernt werden. Dieser Vorgang ist irreversibel, die entfernten Anteile können später nicht



**Abb. 14.2** Butterworth-Filter – Bodediagramme (aus [1])

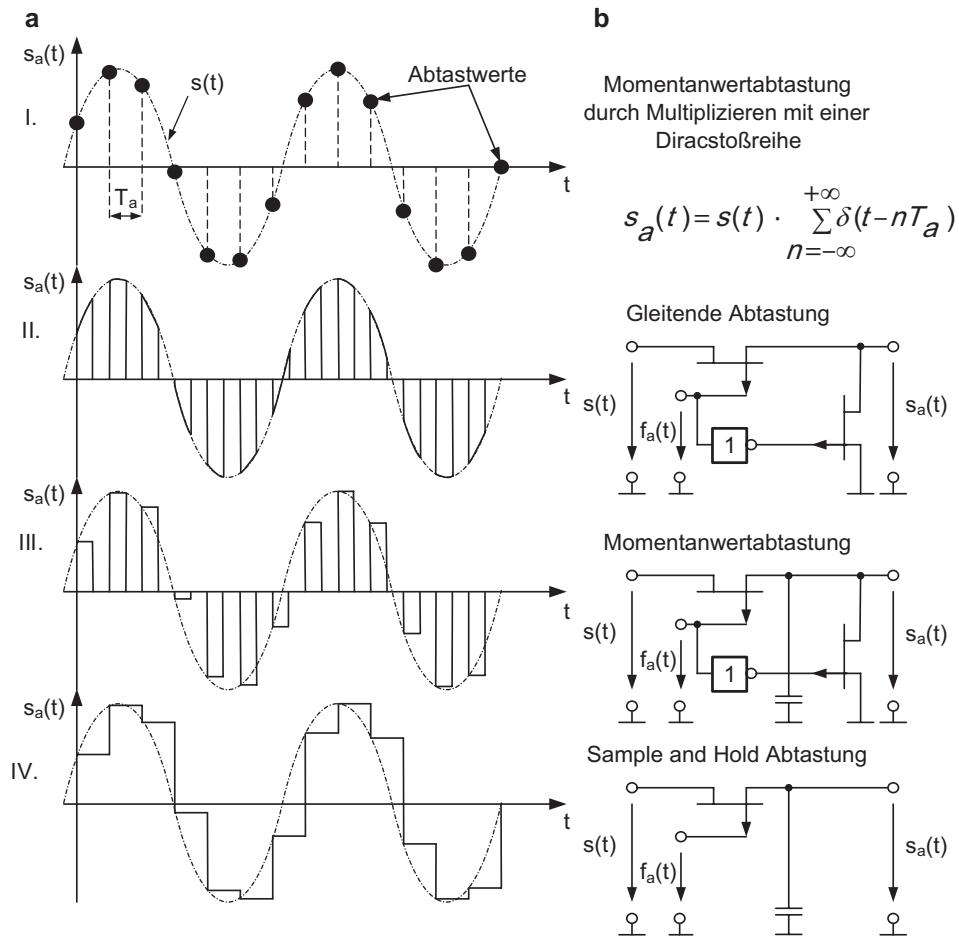
mehr zurückgewonnen werden. Die Bandbegrenzung wird meist mit aktiven Filtern realisiert. Mit steigender Komplexität (Ordnung) des Filters steigt die Steilheit, mit der der Amplitudengang des Filters ab der Grenzfrequenz  $f_g$  abfällt. Ein Filter höherer Ordnung ermöglicht die Wahl der Grenzfrequenz nah an der Nyquist-Frequenz wie es in Abb. 14.2 verdeutlicht ist. Das Bild stellt den Betrag der Übertragungsfunktion  $|H(j\omega)|$  in dB für Butterworth-Filter 1., 2. und 3. Ordnung mit unitärem Verstärkungsfaktor in Abhängigkeit der normierten Kreisfrequenz  $\Omega = \omega/\omega_g$  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  $n \cdot 20 \text{ dB/Dekade}$  auf, wobei  $n$  die Ordnung des Filters ist. Durch das Zurückrechnen auf lineare Werte erhält man einen Dämpfungsfaktor von  $10^n/\text{Dekade}$ .

Durch die Abtastung eines analogen Signals entsteht ein PAM<sup>1</sup>-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. 14.3a 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 FET<sup>2</sup>-Transistoren als elektronische Schalter realisiert wie in Abb. 14.3b prinzipiell dargestellt ist. Die gleitende und die Momentanwertabtastung Abb. 14.3 II, III werden benutzt um die Abtastwerte mehrerer Kanäle auf eine Datenleitung im Zeitmultiplexverfahren zu übertragen. Das Sample-and-Hold<sup>3</sup>-Verfahren Abb. 14.3 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.

<sup>1</sup>PAM -Pulsamplitudenmodulation.

<sup>2</sup>FET – Field Effect Transistor (Feldeffekttransistor).

<sup>3</sup>Sample and Hold – Abtasthalteglied.



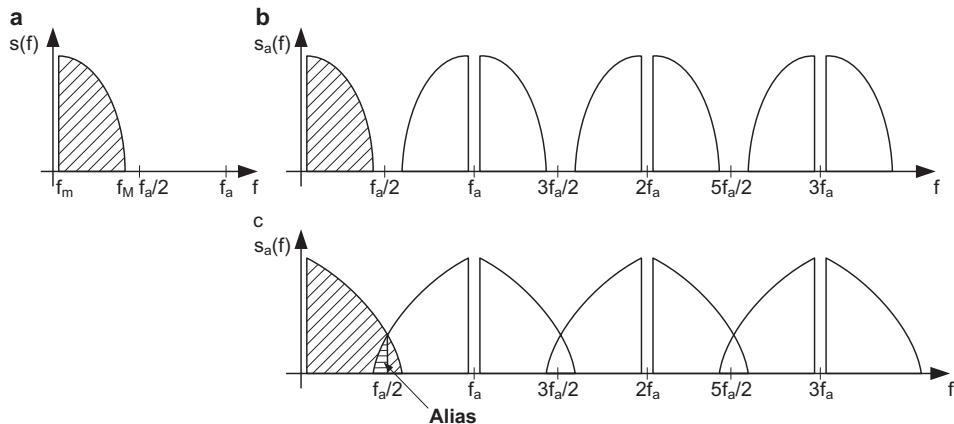
**Abb. 14.3** Abtastverfahren

Das Spektrum  $s(f)$  eines analogen bandbegrenzten Signals  $s(t)$  ist in Abb. 14.4 dargestellt. Der mathematische Ausdruck des Fourier-Spektrums  $s_a(f)$  des abgetasteten Signals lautet [3]:

$$s_a(f) = \frac{1}{T_a} \cdot \sum_{n=-\infty}^{+\infty} s(f - nf_a) \quad (14.2)$$

Gl. 14.2 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.



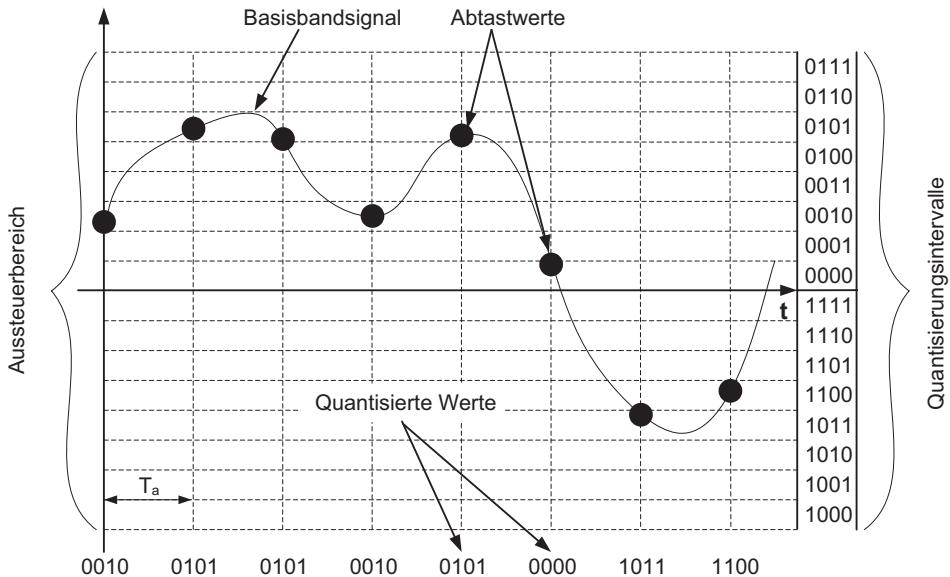
**Abb. 14.4** Spektrum des bandbegrenzten Signals  $s(f)$  und des abgetasteten Signals  $s_a(f)$  für  $2f_M < f_a$  und  $2f_M > f_a$

Abb. 14.4b 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, eine zu hohe Grenzfrequenz hat oder nicht steil genug ist, entsteht eine Überlappung der Teilspektren wie in Abb. 14.4c, 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. Das Vorhandensein der Alias-Komponenten ist auch die Basis für das oben erwähnte Nyquist-Kriterium. Da das Spektrum wie in Abb. 14.4 gezeigt gespiegelt wird, liegt es nahe, dass es auf den Bereich  $\pm f_a/2$  begrenzt sein muss.

## 14.2 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. 14.5 beispielhaft für  $n=4$  dargestellt ist. Jedem Quantisierungs-



**Abb. 14.5** Lineare Quantisierung

intervall 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 Quantisierungsintervalle 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 [5], dann nimmt das Quantisierungsrauschen Werte im Bereich  $[-u_q/2; +u_q/2]$  an. Ein Maß für dieses Rauschen bildet der Störabstand ab [4]:

$$\left. \frac{S}{N_q} \right|_{dB} = 10 \log \frac{S}{N_q} \approx n \cdot 6 \text{ in dB} \quad (14.3)$$

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.

## 14.3 Digitale Filterung

Ein digitales Filter wird so eingesetzt, wie das Blockschaltbild eines digitalen Sensors Abb. 14.1 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.

### 14.3.1 Finite-Impulse-Response-Filter (FIR)

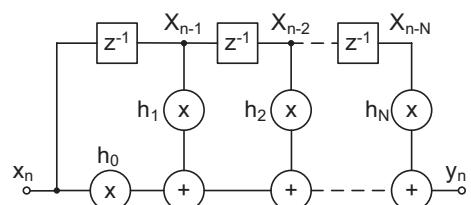
FIR-Filter sind Filter mit endlicher Impulsantwort mit der Übertragungsfunktion [3]:

$$H(z) = \sum_{k=0}^N h[k] \cdot z^{-k} \quad (14.4)$$

$N$  ist die Filterordnung,  $h[k]$  sind die Filterkoeffizienten, die durch die Abtastung der Impulsantwort des Filters berechnet werden können [3] und  $z^{-k}$  bedeutet eine Zeitverzögerung.

$$h[k] = \frac{1}{2} \cdot \sin\left(\frac{\pi}{2} \cdot \left(k - \frac{N}{2}\right)\right) \text{ mit } k = 0 \dots N. \quad (14.5)$$

Abb. 14.6 FIR-Filter



Die si-Funktion, oder Spaltfunktion wird definiert als  $si(x) = \sin(x)/x$ . Die Struktur eines FIR-Filters ist in Abb. 14.6 dargestellt. Für die Berechnung des gefilterten Wertes  $y_n$ , werden außer dem aktuellen Abtastwert nur noch die  $N$  vorangegangenen Werte betrachtet:

$$y_n = \sum_{k=0}^N x[n-k] \cdot h[k] \quad (14.6)$$

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:

$$w[k] = a - b \cdot \cos\left(\frac{2\pi \cdot k}{N}\right) \quad (14.7)$$

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, 3] wird als weitere Entwurfsmethode die Optimalmethode beschrieben, die zu einer niedrigeren Filterordnung führt.

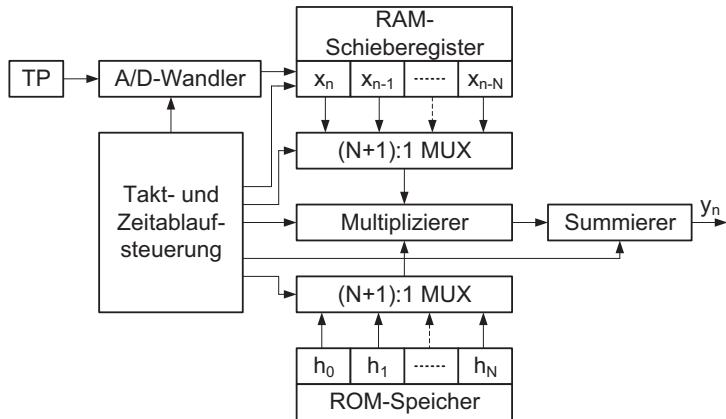
Ein digitales Filter, das in Echtzeit arbeitet, kann hardwaremäßig – wie in Abb. 14.7 dargestellt – realisiert werden. Die hohe Taktfrequenz und die parallelen Abläufe erlauben eine komplexere Filterstruktur. Eine anwendungsspezifische integrierte Schaltung (ASIC<sup>4</sup>) kann ein digitales Filter zusammen mit dem Antialiasing-Tiefpassfilter (TP) und dem A/D-Wandler integrieren. Ein FPGA<sup>5</sup> braucht zusätzlich ein externes Filter und die weniger komplexen CPLD<sup>6</sup>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. 14.6 auch mit einem Digital-Signal-Processor (DSP) oder einem Mikrocontroller gelöst werden. Dies ist in Abb. 14.8 prinzipiell dargestellt. 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. 14.6 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.

<sup>4</sup>ASIC – application specific integrated circuit.

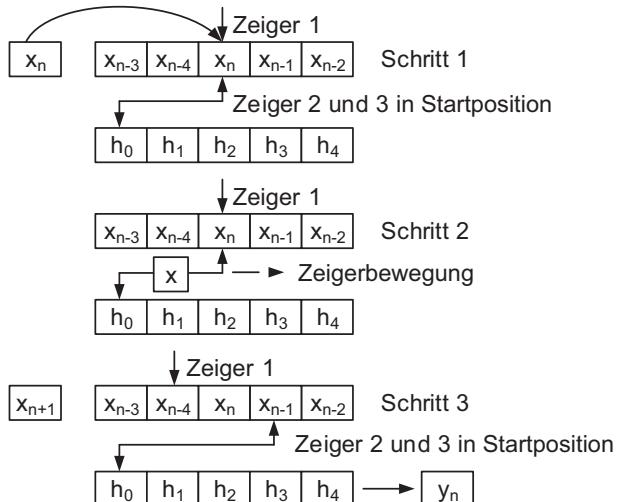
<sup>5</sup>FPGA – field programmable gate array.

<sup>6</sup>CPLD – complex programmable logic device.



**Abb. 14.7** Digitales Filter – prinzipieller Aufbau

**Abb. 14.8** FIR-Filter – Softwarelösung



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.

### 14.3.2 Infinite-Impulse-Response-Filter (IIR)

Die Übertragungsfunktion eines IIR-Filters bildet diejenige eines analogen Filters nach:

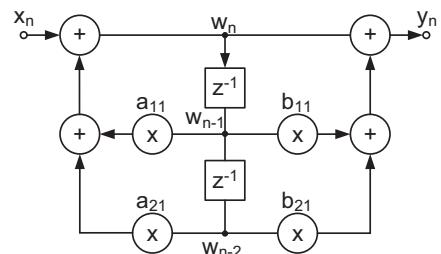
$$H(z) = \frac{1 + b_1 \cdot z^{-1} + \dots + b_N \cdot z^{-N}}{1 - a_1 \cdot z^{-1} - \dots - a_M \cdot z^{-M}} \quad (14.8)$$

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:

$$H(z) = H_1 + \prod_{k=1}^n H_{2k}(z) \quad (14.9)$$

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. 14.9 dargestellt.

**Abb. 14.9** IIR-Filter 2. Ordnung



**Tab. 14.1** 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

### 14.3.3 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 Obergewelligkeit 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. 14.1 nur die ersten neun aufgelistet. Die mit Matlab berechneten Koeffizienten werden mit 128 multipliziert und anschließend gerundet. 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 Ein-
gangspuffer (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.

---

## 14.4 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 I<sup>2</sup>C 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.

---

## 14.5 Abstraktion der I/O-Pins

Im Allgemeinen besitzen Sensoren außer den seriellen Datenleitungen weitere Steuer-eingänge. 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. 8) 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.

---

## 14.6 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.

### 14.6.1 Mikrocontrollerinterne Zahlenformate

Die Arbeitsregister eines 8-Bit-Mikrocontroller der Familie ATmega und die Speicherzellen, die einzeln adressierbar sind, sind alle acht Bit groß. Um einen 64 kByte großen Speicherbereich adressieren zu können, werden Registerpaare verwendet. Man unterscheidet zwischen vorzeichenlosen (*unsigned*) und vorzeichenbehafteten (*signed*)

**Tab. 14.2** 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

Ganzzahltypen. Ein vorzeichenbehafteter Ganzzahltyp speichert Ganzzahlen, ein vorzeichenloser Typ nur natürliche Zahlen und die „0“. Tab. 14.2 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 acht Bit sind, durch das Statusregister SREG:

- **Bit 5** – 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.
- **Bit 4** – Sign bit (S) – wird folgendermaßen berechnet:  $S = N \oplus V$ , und wird beim Testen von *signed*-Datentypen verwendet.
- **Bit 3** – 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.
- **Bit 2** – Negative flag (N) – bildet das höchstwertige Bit des Ergebnisses einer arithmetischen oder logischen Operation ab.
- **Bit 1** – Zero flag (Z) – wird gesetzt, wenn das Ergebnis einer arithmetischen oder logischen Operation Null ist.
- **Bit 0** – 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.

### 14.6.2 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:

$$\begin{aligned} (2^n - 1) + 1 &= 0 \text{ bzw. f\"ur } m < (2^n - 1) & (2^n - 1) + m &= m - 1 \\ 0 - 1 &= 2^n - 1 & 0 - m &= 2^n - m \end{aligned} \quad (14.10)$$

Operationen (siehe Gl. 14.10), die als Ergebnis eine Zahl außerhalb des Wertebereiches liefern, f\"uhren zu einem \"Uberlauf-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 *unsigned char* ein negativer Wert zugewiesen wird, speichert die Variable nach dem Ausf\"uhren der Anweisung (z. B. *ucZahl1 = -1*) das Zweierkomplement des Zahlbetrags (in diesem Beispiel 255).
- **Addition und Multiplikation:** Grunds\"atzlich kann die Addition und die Multiplikation zweier Variablen *ucZahl1* und *ucZahl2* zu einer Bereichs\"uberschreitung f\"uhren. Wird das nicht ber\"ucksichtigt, wird das Ergebnis folgendermaßen entstehen:
  - *ucErgebnis = (ucZahl1 + ucZahl2) % 256* bzw.
  - *ucErgebnis = (ucZahl1 \* ucZahl2) % 256*.
- **Division:** Bei der Ganzzahl Division kann ein Rundungsfehler auftreten.
- **Vergleich:** Die Anweisung1 aus dem folgenden Beispiel wird nie ausgef\"uhrt, weil die Bedingung als falsch ausgewertet wird f\"ur alle Werte von *Zahl1* und die Anweisung2 wird immer ausgef\"uhrt, weil f\"ur alle Werte der Variable *ucZahl2*, die Bedingung wahr ist:

```
if(ucZahl1<0) Anweisung1;
if(ucZahl2>-7) Anweisung2;
```

- **Schieben:** Solange der Wertebereich nicht \"uberschritten 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  $2^n$ .

### 14.6.3 Vorzeichenbehaftete Ganzahltypen

Die vorzeichenbehafteten Ganzahldatentypen 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 acht Bit betrüge 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 es keine Symmetrie der positiven und negativen Zahlen gegenüber der Null. Der Wertebereich eines  $n$ -Bit vorzeichenbehafteten Ganzahltyps 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:

$$x + (-x) = 0 \quad (14.11)$$

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. 14.12 stellt den Zweierkomplement-Überlauf in mathematischer Form dar. Ein Zahlenbeispiel für solch einen Zweierkomplement-Überlauf ist in der Tab. 14.3 veranschaulicht. Die Variable scZahl ist vom Typ signed char.

$$\begin{aligned} (2^{n-1} - 1) + 1 &= -2^{n-1} \\ (-2^{n-1}) - 1 &= (2^{n-1} - 1) \end{aligned} \quad (14.12)$$

**Tab. 14.3** Zahlenbeispiel für den Zweierkomplement-Überlauf

Operation	Wert vor der Operation	Wert nach der Operation
scZahl++;	+127	-128
scZahl--;	-128	+127

**Tab. 14.4** 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

In Tab. 14.4 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. 14.4 ist der Unterschied zwischen der Ganzzahldivision und der Rechtsschiebung der negativen Werte zu entnehmen.

#### 14.6.4 Erkennung und Verhinderung eines Überlaufs

Die vom Überlauf verursachten Fehler haben besonders in rekursiven Algorithmen gravierendere 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.

Mithilfe 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 Wertebereichsüberprüfung durchzuführen. In der Tat haben Wertebereichsüberschreitungen bereits zu dramatischen Unfällen geführt, zum Beispiel dem Absturz der Ariane-Rakete beim Jungfernflug 501 im Jahr 1996<sup>7</sup>.

---

## Literatur

1. Meroth, A., & Tolg, B. (2008). *Infotainmentsysteme im Kraftfahrzeug*. Friedr. Vieweg & Sohn Verlag & GWV Fachverlage GmbH.
2. von Grünigen, D. C. (2008). *Digitale Signalverarbeitung*. Hanser.
3. Roppel, C. (2006). *Grundlagen der digitalen Kommunikation*. Hanser.
4. Werner, M. (2006). *Nachrichten-Übertragungstechnik*. Friedr. Vieweg & Sohn Verlag & GWV Fachverlage GmbH.
5. Borucki, L. (1985). *Grundlagen der Digitaltechnik*. Teubner Verlag.
6. Microchip. (2021). 8-bit Atmel Microcontroller with 4/8/16k bytes in-system programmable flash. [www.microchip.com](http://www.microchip.com).
7. 8-bit AVR® Instruction set. <http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf>. Zugegriffen: 2. Apr. 2021.

---

<sup>7</sup>Vgl. <http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf> bzw. <http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html>.



# Umweltsensoren

15

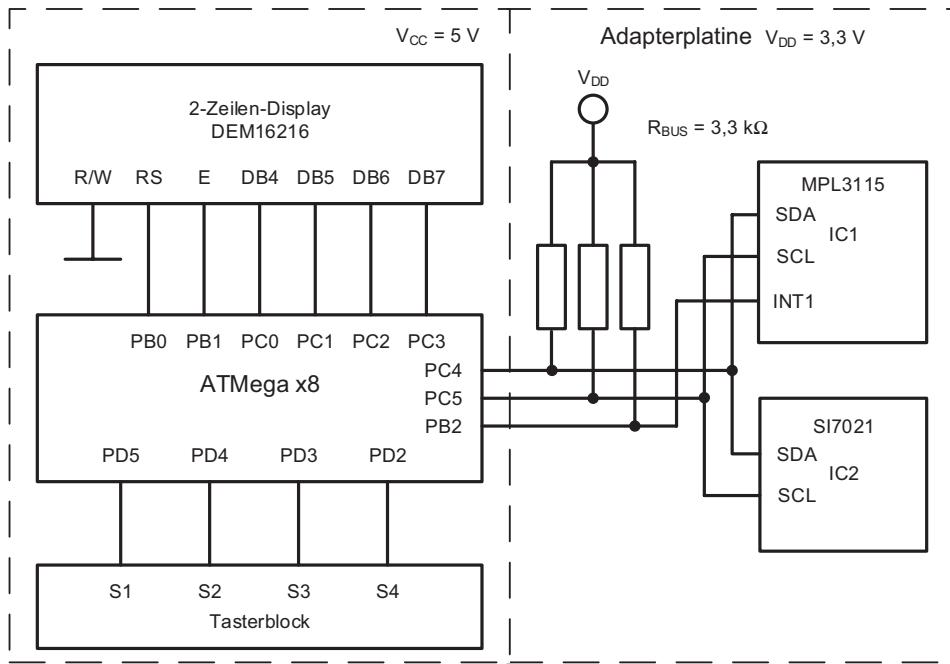
## Zusammenfassung

In diesem Kapitel werden vier digitale Umweltsensoren vorgestellt, ein digitaler Luftdruckmesser mit Temperaturmessung, ein Luftfeuchtigkeitsmesser mit Temperaturmessung, ein Temperatursensor und ein Feinstaubsensor.

Umweltsensoren erlauben die Erfassung von Daten aus Luft oder Wasser. Sie werden in verschiedenen Anwendungen benötigt, zum einen in der Erfassung von Atmosphärendaten wie Druck, Temperatur und Luftfeuchtigkeit mit dem Ziel, das Wetter zu bestimmen oder die Umgebungsbedingungen beispielsweise bei der Lagerung von Produkten. Zum anderen liefern sie Aussagen über die Qualität der Umgebung, beispielsweise die Feinstaubkonzentration oder die Konzentration bestimmter Gase in der Luft. Drittens benötigt man ihre Daten für die indirekte Ermittlung von Größen, beispielsweise des Taupunkts, der Höhe über dem Meeresspiegel oder einem Luftpengendurchsatz. In diesem Kapitel werden beispielhaft vier Sensoren vorgestellt, der digitale Drucksensor MPL3115, der Luftfeuchte-sensor SI7021, der Temperatursensor TMP75 und der Feinstaubsensor SDS011.

Die Originalversion dieses Kapitels wurde revidiert. Ein Erratum ist verfügbar unter  
[https://doi.org/10.1007/978-3-658-31709-6\\_27](https://doi.org/10.1007/978-3-658-31709-6_27)

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann  
[https://doi.org/10.1007/978-3-658-31709-6\\_15](https://doi.org/10.1007/978-3-658-31709-6_15).



**Abb. 15.1** Elektronische Wetterstation mit MPL3115 und SI7021

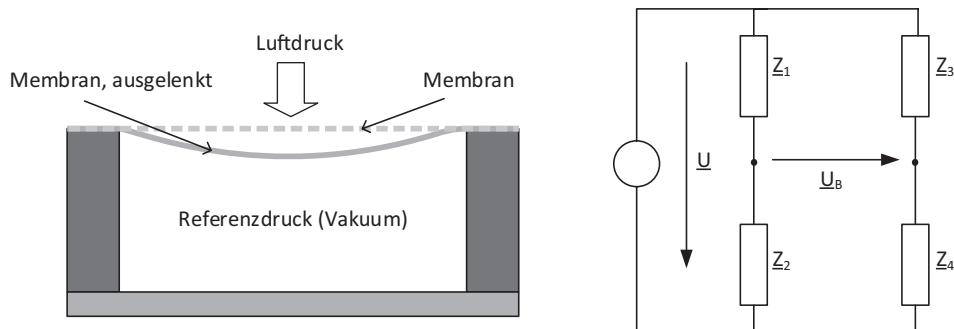
## 15.1 MPL3115 digitaler Luftdrucksensor

MPL3115 [9, 10] und [11] 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 I<sup>2</sup>C-Slave vorkonfiguriert und kann mit einem Master im Fast-Modus kommunizieren. Über I<sup>2</sup>C überträgt der Master die gewünschten Einstellungen und liest die gemessenen Werte aus. Über einstellbare Interrupts und ansteuerbare Ausgänge können dem Master eingetretene Ereignisse signalisiert werden.

In Abb. 15.1 wird die Schaltung einer elektronischen Wetterstation dargestellt, die aus einem Luftdrucksensor MPL3115 und einem Luftfeuchtigkeitssensor SI7021 besteht. Der Feuchtigkeitssensor wird im Abschn. 15.2 beschrieben.

### 15.1.1 Funktionsweise

Grundsätzlich arbeiten Sensoren zur Erfassung des Absolutdrucks mit einer Membran, die im Vakuum oder einem sehr niedrigen Referenzdruck auf einen Kammer mit



**Abb. 15.2** Prinzip eines Drucksensors (links) und Brückenschaltung (rechts)

stabilen Wänden aufgebracht ist und diese hermetisch versiegelt. Sobald der Sensor dem Atmosphärendruck ausgesetzt ist, erfährt die Membran in Richtung des Vakuums eine Kraft und wird wegen ihrer Elastizität ausgelenkt (siehe Abb. 15.2). Das Maß der Auslenkung ist mit dem Druck über die Elastizität, eine feste Form- und Materialgröße, verknüpft. Um den Druck zu bestimmen, muss man also die Auslenkung bestimmen, die naturgemäß sehr gering ist. Bei einem mikromechanischen Sensor (MEMS<sup>1</sup>-Sensor) liegen die Werte für die Membrandicke bei einigen  $\mu\text{m}$ , den Abstand der Membran zur Grundplatte bei einigen zehn bis hundert  $\mu\text{m}$  und der Durchmesser der Druckkammer bei einigen hundert  $\mu\text{m}$ .

Um die Auslenkung zu messen, hat man prinzipiell zwei Möglichkeiten:

- Bei der resistiven Messung nutzt man den Effekt aus, dass sich der Widerstand eines Materials aufgrund seiner Dehnung ändert. Dies kann ein metallisches Material sein, ein Metalloxid oder ein Halbleiter, im letzteren Fall wird meist der piezoresistive Effekt genutzt. Da die Widerstandsänderungen extrem klein sind, wird man die Messung mit einer Brückenschaltung durchführen (Abb. 15.2 rechts). Die Impedanzen (bei resistiven Sensoren die Widerstände) der Brücke bilden zwei Spannungsteiler und die Brückenspannung  $U_B$  ist im ausgeglichenen Fall 0 V, sobald die Brücke verstimmt wird, ändert sich die Spannung. Je nach Messprinzip kann man nun einen Widerstand (meist  $Z_2$ ), zwei diagonal gegenüberliegende Widerstände ( $Z_2$  und  $Z_3$ ) oder alle vier verändern, wobei man dann in der Regel dafür sorgt, dass sich je zwei Widerstände gegenläufig mit der Messgröße verändern, wodurch die Empfindlichkeit erheblich gesteigert wird.

Eine sehr anschauliche Herleitung und Berechnungsgrundlage findet sich in Kap. 8 von [1]. Resistive Sensoren sind oft empfindlich gegenüber Temperaturschwankungen und werden dementsprechend kompensiert.

<sup>1</sup> MEMS steht für Micro-Electro-Mechanical Systems.

- Bei der kapazitiven Messung nutzt man den Effekt aus, dass sich die Kapazität der Membran gegenüber der Grundplatte ändert, wenn sich die Membran verbiegt. Die Kapazität wird mit Wechselstrom entweder ebenfalls in einer Brückenschaltung oder durch Verstimmen der Resonanzfrequenz eines Schwingkreises ermittelt (Kap. 9 in [1])
- Bei der piezoelektrischen Messung verändert sich die elektrostatische Feldstärke aufgrund der Verformung eines Kristalls, indem Ladungen verschoben werden.
- Weitere Messelemente können magnetische Hallsensoren oder Tauchspulen sein. Diese werden in der MEMS-Technik nicht eingesetzt.

Neben Absolutdruck-Sensoren gibt es auch Relativdrucksensoren und Differenzdrucksensoren. Relativdrucksensoren messen den Druck in einem Raum gegenüber dem Außendruck (beispielsweise Reifendrucksensoren, da hier nicht der absolute Druck sondern der Druck im Reifen gegenüber dem Außendruck interessant ist). Eine Öffnung in der Referenzkammer sorgt bei diesen Sensoren für einen Druckausgleich gegenüber der Umgebung. Differenzdrucksensoren messen den Druck in zwei voneinander getrennten Räumen, beispielsweise vor oder hinter einem Filter oder einer Verengung. Damit kann man die Leistungsfähigkeit eines Filters bestimmen oder – im Fall der Verengung – indirekt auf die Strömungsgeschwindigkeit und damit den Durchsatz des Mediums schließen. Selbstverständlich funktionieren beide Messprinzipien auch unter Verwendung zweier Absolutdruck-Sensoren und anschließender Berechnung. Eine sehr gut lesbare Übersicht über die Druckmesstechnik liefert [2] (kostenlos beim Verlag erhältlich).

### 15.1.2 Aufbau des MPL3115

Ein schematisches Blockschaltbild des Bausteins zeigt Abb. 15.3.

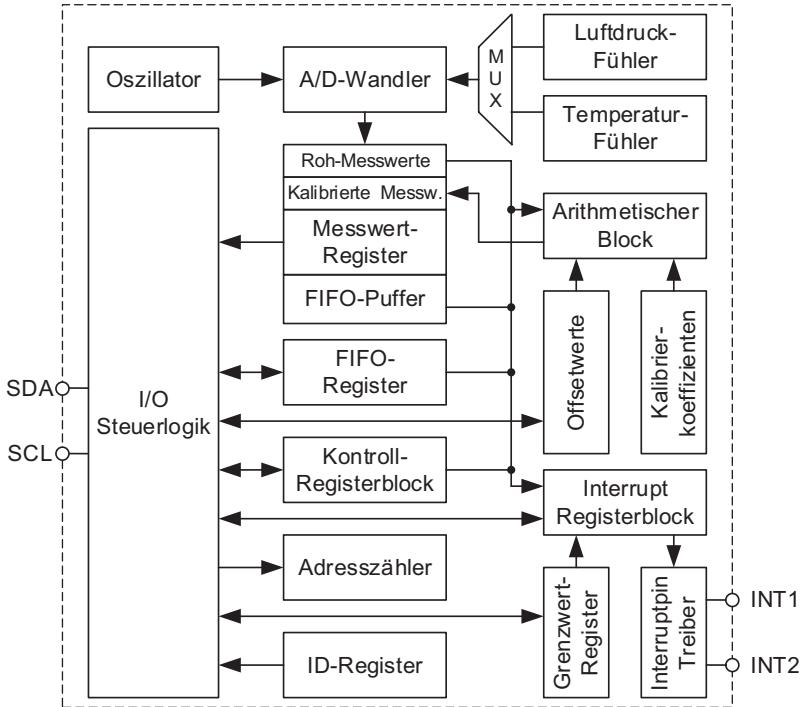
#### 15.1.2.1 Messfühler

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 [4] berechnet werden:

$$p = p_0 \cdot e^{-\frac{H}{8000}}, \quad (15.1)$$

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 Meereshöhe  $p_0$  mit der Gl. 15.2 aus [7] berechnet werden:

$$p_0 = \frac{p}{\left(1 - \frac{\text{Höhe}}{44330,77}\right)^{\frac{1}{0,1902632}}} \quad (15.2)$$



**Abb. 15.3** MPL3115 – Blockschaltbild

Der Koeffizient 44330,77 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 [4]):

$$44330 \approx \left| \frac{T_m}{\gamma} \right| \quad (15.3)$$

Der Sensor berechnet mit dem normalen Luftdruck auf Meereshöhe  $p_0 = 1013,25\text{ hPa}$  die Höhe mit der Gl. 15.4 [9]. 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.

$$H = 44330,77 \cdot \left( 1 - \left( \frac{p}{p_0} \right)^{0,1902632} \right) \quad (15.4)$$

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.

### 15.1.2.2 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 I<sup>2</sup>C 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 [9] zu entnehmen. Die Werte werden in die Register im Big-Endian-Format gespeichert.

#### 15.1.2.2.1 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 unkom pensierten 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:

$$t_{OS} = 2^{OS} \cdot 4 + 2 \text{ in ms} \quad (15.5)$$

- **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 Einzelmessung, 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<sup>15</sup>] 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.

### 15.1.2.2.2 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.

### 15.1.2.2.3 Messwertregister

#### 15.1.2.2.3.1 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, zwei 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, vier 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, vier 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;
```

#### 15.1.2.2.3.2 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\_CSB* (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.

#### 15.1.2.2.3.3 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. 15.1 dargestellt. Sie können jederzeit auf „0“ gesetzt werden, um nach einem neuen Minimum bzw. Maximum zu suchen.

#### 15.1.2.2.4 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.

**Tab. 15.1** 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

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. 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(); //Stopp
return TWI_OK;
}

```

### 15.1.2.2.5 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  $\Delta p$  ( $\neq 0$ ) 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  $\Delta t$  ( $\neq 0$ ) in die Register *T\_WIND\_REG* (0x1B) gespeichert;
- **Bit 3** – der gemessene Luftdruck/die berechnete Höhe überquert einen der Grenzwerte:  $\Delta p$ ,  $p_g - \Delta p$  oder  $p_g + \Delta p$ ;
- **Bit 2** – die gemessene Temperatur überschreitet einen der Grenzwerte:  $\Delta t$ ,  $t_g - \Delta t$  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(); //Stopp  
    return TWI_OK;  
}
```

#### 15.1.2.2.6 Device-Identifikation-Register

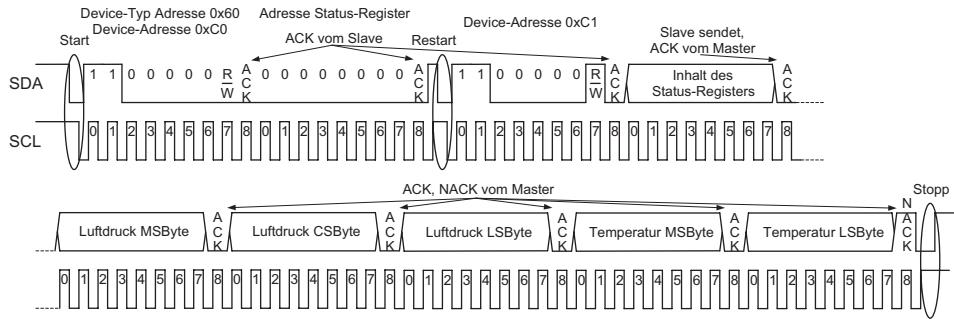
Das Device-Identifikation-Register *WHO\_I\_AM\_REG* (0x0C) speichert den Wert 0xC4 und kann in einem Bussystem zur Identifikation des Bausteins und zum Testen des Busses dienen.

### 15.1.3 Serielle Kommunikation

Der Sensor MPL3115 implementiert das I<sup>2</sup>C-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. 15.4 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.

### 15.1.4 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



**Abb. 15.4** MPL3115 – Serielle Kommunikation

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.

### 15.1.5 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. 15.5) 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.

### 15.1.6 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.

---

## 15.2 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  $\text{g/m}^3$  oder  $\text{kg/m}^3$  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  $\text{RH}^2$  ist das Verhältnis zwischen der aktuellen Wasserdampfmasse und derjenigen im Sättigungszustand (bei gleicher Lufttemperatur und gleichem Luftdruck) und wird in Prozent gemessen. Die Taupunkttemperatur oder der Taupunkt ist die Temperatur bei der

---

<sup>2</sup>RH – relative humidity.

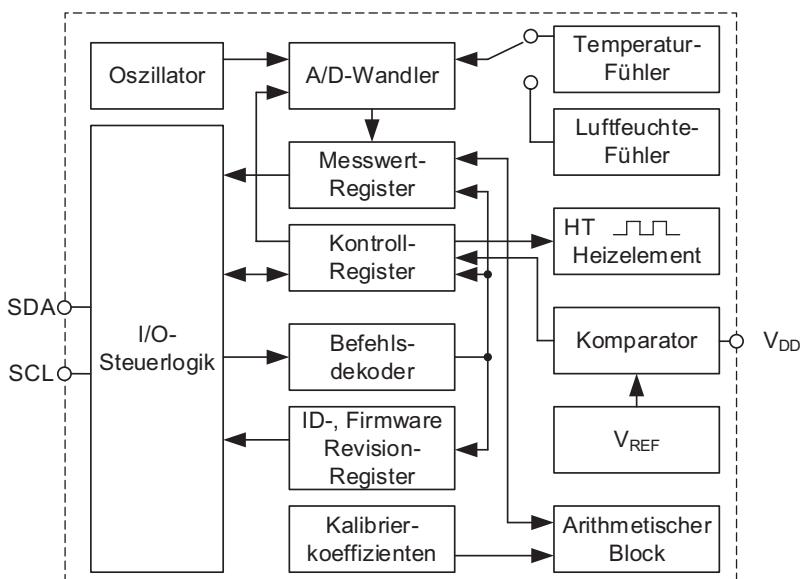
die relative Luftfeuchtigkeit 100 % erreicht. Der Wasserdampf kondensiert, wenn die Temperatur unter den Taupunkt sinkt.

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 Luftfeuchtigkeit auf der Messung des elektrischen Widerstandes oder der elektrischen Kapazität beruhen.

### 15.2.1 Aufbau des SI7021

SI7021 [5, 6] 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. 15.5 dargestellt.



**Abb. 15.5** SI7021- Blockschaltbild

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.

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. 15.2 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 I<sup>2</sup>C empfangen wurden, und steuert entsprechend die internen Baugruppen des Sensors. Diese Befehle

**Tab. 15.2** SI7021 Bitauflösung und Messdauer der Temperatur und Luftfeuchtigkeit

RES1	RES0	Temperatur		Luftfeuchtigkeit	
		Bitauflösung (Bit)	Messdauer (ms)	Bitauflösung (Bit)	Messdauer (ms)
0	0	14	10,8	12	12
0	1	12	3,8	8	3,1
1	0	13	6,2	10	4,5
1	1	11	2,4	11	7

werden in einer Botschaft gleich nach der Adressierung im Schreibmodus des Slaves platziert, so wie in Abb. 15.6 dargestellt.

Der Sensor speichert in acht 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 I<sup>2</sup>C ablesen.

### 15.2.2 Serielle Kommunikation

Der Sensor ist als I<sup>2</sup>C-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 V und 3,6 V versorgt werden. Die SDA- und 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.

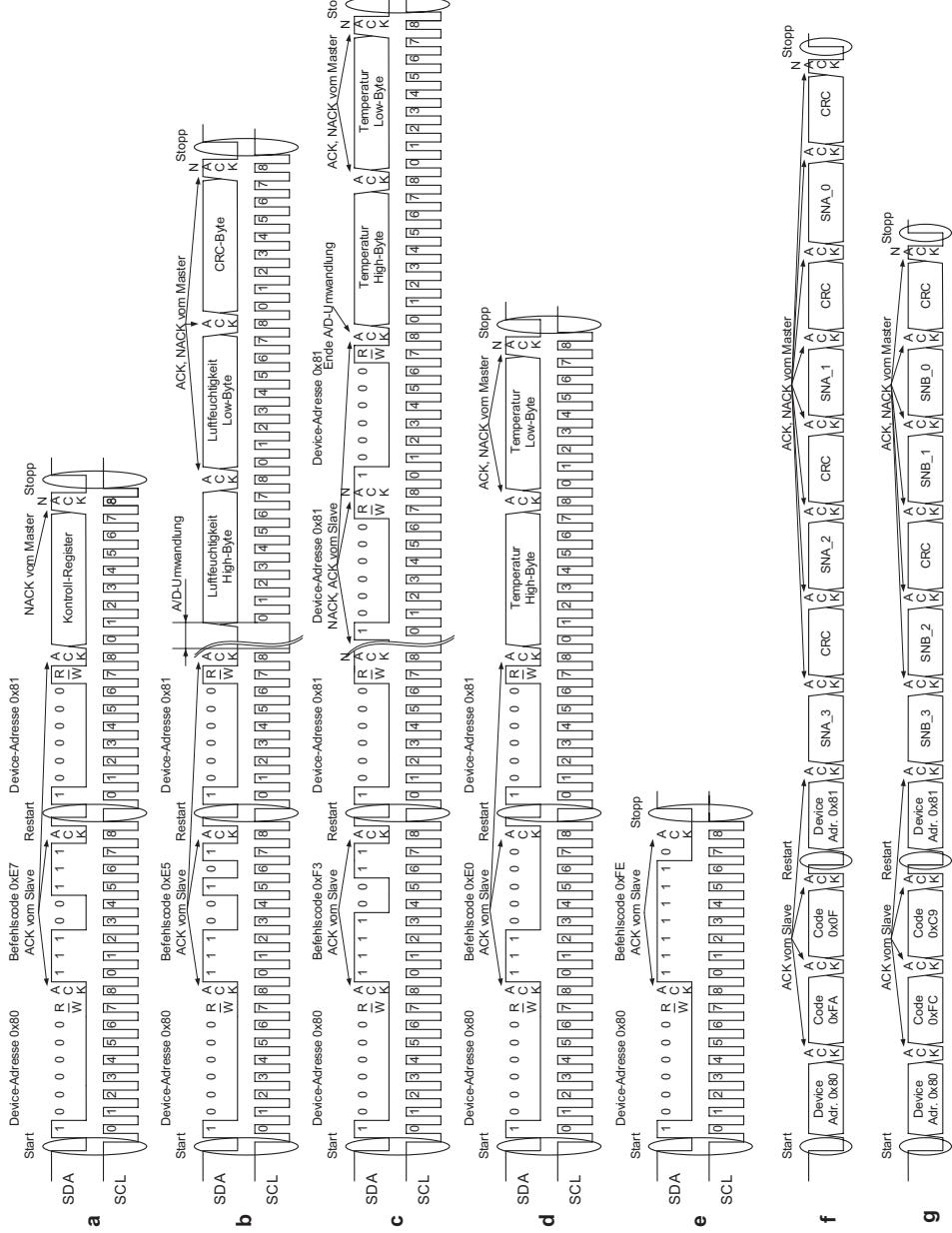
Um Energie sparen zu können, funktioniert der Baustein nur im Einzelmess-Modus. Jede weitere Messung einer Größe muss extra gestartet werden.

#### 15.2.2.1 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. 15.2). 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. 15.6a 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.

Abb. 15.6 SI7021 I<sup>2</sup>C Kommunikation

### 15.2.2.2 Messung der relativen Luftfeuchtigkeit

Eine neue Messung der relativen Luftfeuchtigkeit wird mit dem Code `0xE5` im Hold-Modus (siehe Abb. 15.6b 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. 15.2 zu finden ist. Nach RESTART und erneuter Adressierung streckt der Slave im Hold-Modus das SCL-Signal während der Umwandlung und anschließend, 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 CRC<sup>3</sup>-Byte das vom Master mit NACK quittiert wird. Unabhängig von der eingestellten Bitauflösung für die Messung ist das Bit 15 des Messwertes das höchstwertige Bit und die Bit 1:0 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. 15.2.3 vorgestellten Formel in % umgerechnet werden.

Die Clock-Streckung ist ein Verfahren, das vom I<sup>2</sup>C-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. 15.6b. 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
```

<sup>3</sup>CRC –cyclic redundancy check (zyklische Redundanzprüfung).

```

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(); //Stopp
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.

### 15.2.2.3 Messung der Temperatur

Eine neue Temperaturmessung im Hold-Modus wird mit dem Code *0xE3* und im No-Hold-Modus mit *0xF3* (Abb. 15.6c) gestartet. Die Kommunikation verläuft ähnlich wie in Abschn. 15.2.2.2 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. 15.6d dargestellt ist, gelesen werden. Für diese Übertragung gibt es keine Prüfsumme. Unabhängig von der gewählten Bitauflösung, ist das Bit 15 für die Messung des Messwertes das höchstwertige Bit und die Bit 1:0 haben den binären Wert „00“. Mit der Formel aus Abschn. 15.2.3 kann der gemessene Wert in °C umgerechnet werden.

### 15.2.2.4 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. 15.6f 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 I<sup>2</sup>C-Lese-Botschaft nach der Adressierungsphase den Befehl 0x84 und 0xB8. In der Slave-Antwort folgt dem Byte mit der Firmwareversion kein CRC-Byte.

### 15.2.3 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 [5] berechnet:

$$\varphi = \frac{125 \cdot \varphi_{\text{gelesen}}}{2^{16}} - 6 \text{ in \%} \quad (15.6)$$

Um die Luftfeuchtigkeit mit einer Auflösung von 0,1 % berechnen zu können wird die Gl. 15.6 mit 10 multipliziert:

$$\varphi = \frac{1250 \cdot \varphi_{\text{gelesen}}}{2^{16}} - 60 \text{ in } 0,1 \% \quad (15.7)$$

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 [5] die Lufttemperatur berechnet werden:

$$t = \frac{175,72 \cdot t_{\text{gelesen}}}{2^{16}} - 46,85 \text{ in } {}^{\circ}\text{C} \quad (15.8)$$

Im Folgenden wird die Umsetzung der Gl. 15.8 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 zwei ausgelesenen Bytes werden zusammengefasst
    uitemp = (ucTempOrHumidValue[0] << 8) +
    ucTempOrHumidValue[1];
    ftemp = (((float) uitemp * 1757.2) / ((float) 65356)) - 468.5;
    return ftemp;
}
```

Für die zweite Beispiefunktion 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 zwei 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 mithilfe der in [6] vorgestellten Magnus-Formel die Taupunkttemperatur berechnet werden:

$$\tau = \frac{B1 \cdot \left( \ln\left(\frac{\varphi}{100}\right) + \frac{A1 \cdot t}{B1 \cdot t} \right)}{A1 - \ln\left(\frac{\varphi}{100}\right) - \frac{A1 \cdot t}{B1 + t}} \quad (15.9)$$

mit  $\tau$ =Taupunkttemperatur,

$\varphi$ =relative Luftfeuchtigkeit,

$t$ =gemessene Lufttemperatur,

$A1 = 17,625; B1 = 243,04; C1 = 610,94$ , const.

#### 15.2.4 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. 15.2.2.4 beschrieben ist. Im Allgemeinen beruht die Berechnung der CRC-Codes auf einer Polynomdivision (siehe [7]). Es gibt unterschiedliche Verfahren um einen CRC-Code zu berechnen, für den SI7021 wurde das CRC-8-Verfahren von der Firma Maxim Integrated [8] 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 kBByte groß sein

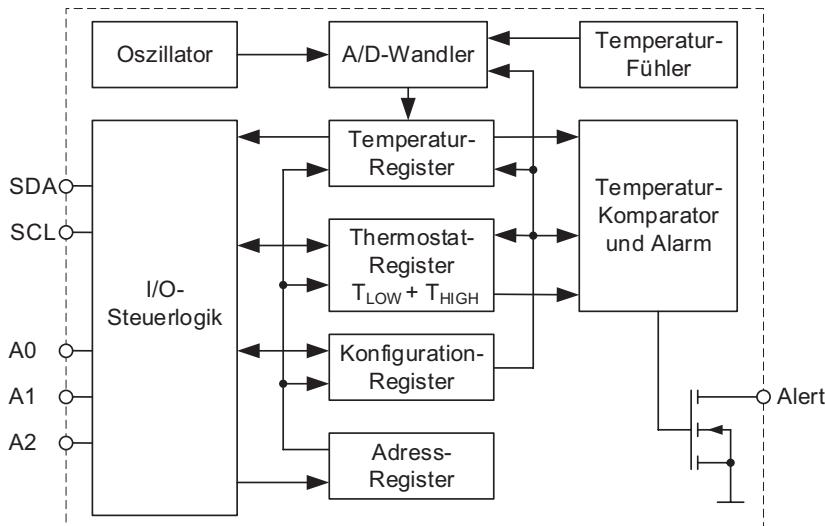
muss. In einem solchen Fall muss der CRC-Code berechnet werden, beispielsweise mit folgender Funktion `uint8_t ComputationCRC(uint16_t ucvalue)`, die das oben genannte Verfahren implementiert. Die Funktion berechnet den CRC-Code eines 1- oder 2-Byte-Wertes und gibt diesen als Rückgabewert zurück.

```
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;
}
```

---

## 15.3 Temperaturmessung mit dem TMP75

TMP75 ist ein digitaler Temperatursensor, der die Temperatur im spezifizierten Bereich von  $-40\text{ }^{\circ}\text{C}$  bis  $+125\text{ }^{\circ}\text{C}$  mit einer typischen Genauigkeit von  $\pm 0,5\text{ }^{\circ}\text{C}$  misst [3]. Er kann als Slave über I<sup>2</sup>C oder SMBus im Fast- oder High-speed-Modus konfiguriert und ausgelesen werden. Bis zu acht gleiche Sensoren können dank der drei Adress-eingänge am gleichen Bus betrieben werden. Ein grobes Blockschaltbild des Sensors ist in Abb. 15.7 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



**Abb. 15.7** TMP75 – Blockschaubild des Sensors

Energiesparmodus geschaltet werden. In diesem Modus bleibt nur noch die serielle Schnittstelle aktiv. Über die eingebaute Thermostatkfunktion 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.

### 15.3.1 Sensorkonfigurierung

Das 8-Bit-Konfigurationsregister mit der Adresse 0x01 kann gelesen oder geschrieben werden. Die Bedeutung der einzelnen Bits ist im Folgenden erläutert:

- **Bit 7 – 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.
- **Bit 6:5 – R1:R0** – Mit diesen zwei Bits wird die Temperaturauflösung eingestellt (Tab. 15.3).
- **Bit 4:3 – F1:F0** – Über diese Bits wird die Anzahl der aufeinanderfolgenden Über- und Unterschreitungen der Temperaturgrenzen eingestellt um einen Temperaturalarm auszulösen (Tab. 15.4).

**Tab. 15.3** TMP75 Auflösung und Messdauer

R1	R0	Bitauflösung	Temperaturschritt (°C)	max. Messdauer (ms)
0	0	9	0,5	37,5
0	1	10	0,25	75
1	0	11	0,125	150
1	1	12	0,0625	300

**Tab. 15.4** TMP75 Codierung der Fehlerzahl

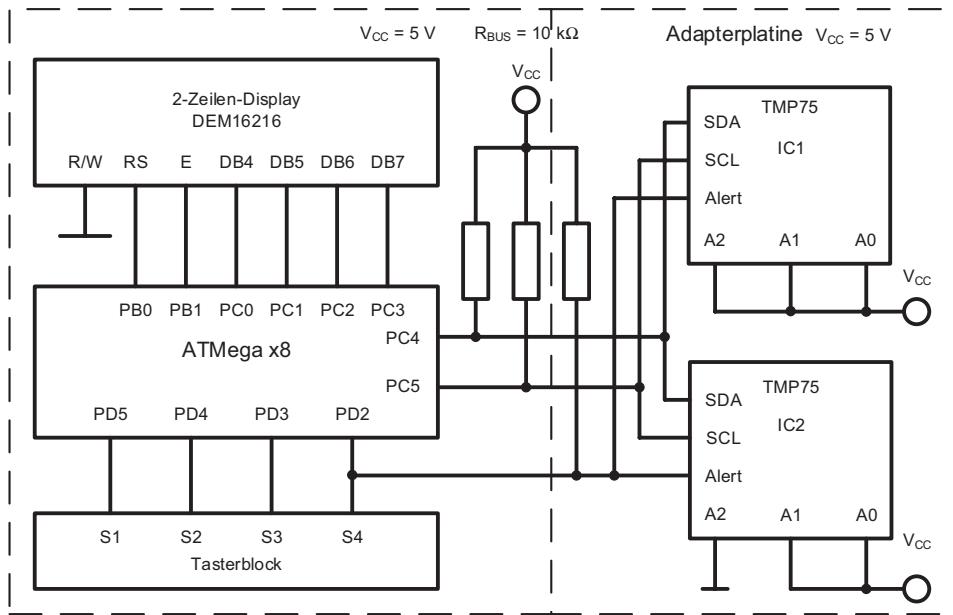
F1	F0	Aufeinanderfolgende Fehler
0	0	1
0	1	2
1	0	4
1	1	6

- **Bit 2 – POL** – Dieses Bit bestimmt die Polarität des Alarmsignals. Wenn das Bit „0“ ist, wird das Alarmsignal im Alarmfall auf Low gesetzt.
- **Bit 1 – TM** – Wenn dieses Bit „1“ ist, arbeitet der Thermostat im Komparator-Modus, ansonsten im Interrupt-Modus.
- **Bit 0 – SD** – Mit dem Setzen dieses Bits durch die Ansteueroftware wird der Sensor in den Energiesparmodus versetzt.

### 15.3.2 Serielle Schnittstelle

Der TMP75 implementiert für die Kommunikation mit einem Master die I<sup>2</sup>C-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 I<sup>2</sup>C-Spezifikation (0 Hz) leicht ab. Über die drei Adresseingänge können die niedrigwertigen 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 oder nicht angeschlossen werden. Die genauen, einstellbaren Adressen sind dem Datenblatt [3] zu entnehmen.

Die Grundadresse des TMP75, wenn alle Adresseingängen 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



**Abb. 15.8** TMP75 – Beschaltung zweier Temperatursensoren

Beispiel in Abb. 15.8 lautet die Adresse von IC1 0x9E (wie die Adresse des Echtzeituhr-Bausteins MAX31629) und die von IC2 0x96.

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 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 Adresseingänge.

Falls sich einer der Teilnehmer in einem I<sup>2</sup>C-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 STOPP-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.

### 15.3.3 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 Bit 0 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 Festkommazahlen als Ganzzahlen. Der Inhalt des Temperaturregisters wird auf zwei Bytes aufgeteilt, wie in Tab. 15.5 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:

- 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) | 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(); //Stopp
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 Aufrufschnittstelle zu:

```

int TMP75_Get_Temperature(void)
{
    return iTemperature;
}

```

### 15.3.4 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

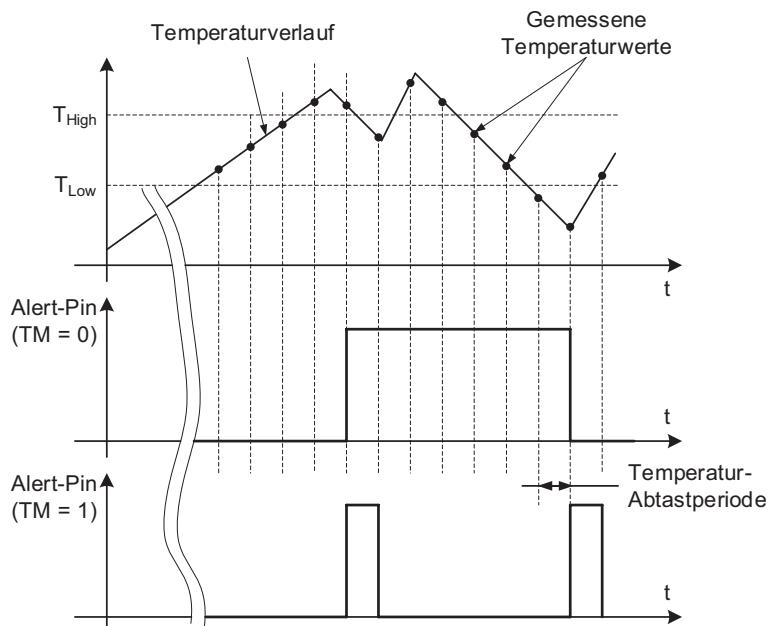
**Tab. 15.5** TMP75 Temperaturwert-Codierung

Temperatur [°C]	Höchstwertiges Byte								Niederwertiges Byte							
	T11	T10	T9	T8	T7	T6	T5	T4	T3	T2	T1	T0	T0	T0	T0	T0
+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

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).

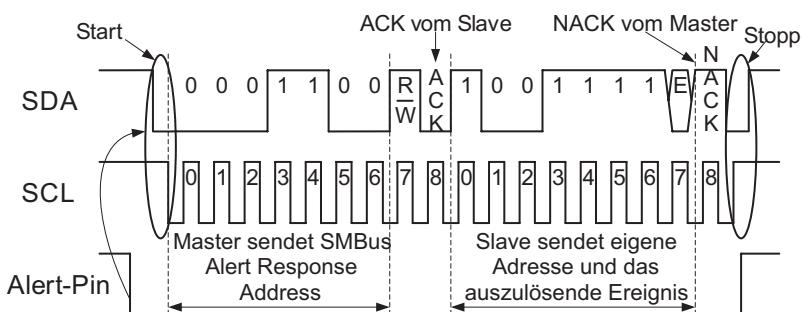
Das Thermostatregister mit der Adresse 0x02 speichert die untere Grenztemperatur, das Register mit der Adresse 0x03 die obere. Das Überschreiten der oberen Grenze beziehungsweise das Unterschreiten der unteren führt zu einem Alarmzustand, der am Alert Ausgang (siehe Abb. 15.9) signalisiert wird. Wenn 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



**Abb. 15.9** Die Thermostat Funktion des Sensors TMP75

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. 15.4 eingestellt werden. Dadurch kann das Auslösen eines Alarms durch Temperaturschwanken vermieden werden. In Abb. 15.9 ist die Thermostatfunktion für die Einstellungen  $POL = „1“$ ,  $F1 = „0“$  und  $F0 = „1“$  graphisch dargestellt.

In einer Schaltung mit mehreren Sensoren, wie in Abb. 15.8 dargestellt, werden die Alert-Ausgänge zu einer UND-Verdrahtung zusammengeschaltet. Weil im Komparator-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. 15.10 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 I<sup>2</sup>C-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 Arbitrierung, 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. 15.10** TMP75 – SMBus-Alert-Response-Funktion

Folgender Code zeigt die Implementierung des zeitlichen Verlaufs aus der Abb. 15.10. 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(); //Stopp
    return TWI_OK;
}

uint8_t TMP75_Get_AlarmAddress(void)
{
    return ucAlarmAddress;
}

```

---

## 15.4 Feinstaubsensor SDS011

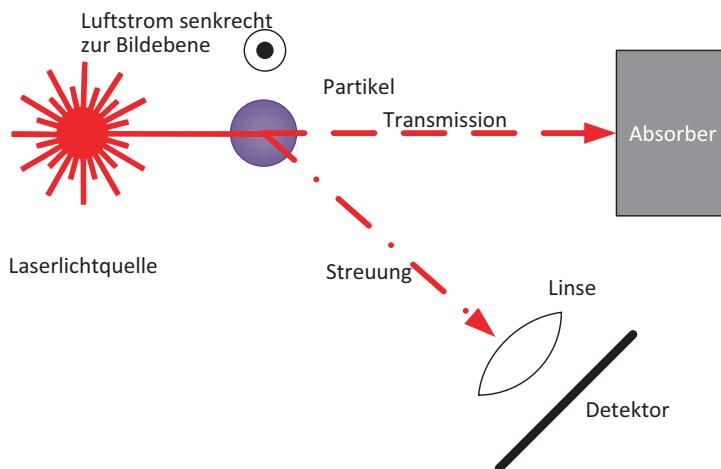
Feinstaub ist eine Mischung von festen Partikeln und feinen Tröpfchen, die in der Luft schweben (Aerosol). Feinstaub kann aufgrund seiner Lungengängigkeit und teilweise sogar Resorption (Aufnahme in den Blutkreislauf) eine erhebliche gesundheitliche Gefahr darstellen. Daher werden zunehmend zur Messung der Luftqualität Feinstaubsensoren eingesetzt. Feinstaub wird in der Regel nicht nach seiner stofflichen Zusammensetzung klassifiziert, sondern nach der Größe der Teilchen. Hierzu ist die Nomenklatur PM<sub>x</sub> vorgesehen, wobei x den maximalen Partikeldurchmesser im Aerosol bezeichnet. Größere Partikel bis 10µm reizen vor allem die Schleimhäute, während Partikel kleiner als 2,5 µm bis tief in die Lunge eindringen und Atemwegserkrankungen verursachen können [12].

Eine gewisse Berühmtheit hat der Feinstaubsensor SDS011 von Nova Fitness, einem Spinoff der Uni Jinan in China, bekommen, da er für ca. 25 € zu kaufen ist und in zahlreichen Umweltüberwachungsprojekten eingesetzt wird. Er misst Feinstäube von  $0,3\mu\text{m}$  bis  $10\mu\text{m}$  in einer Konzentration bis  $999,9 \mu\text{g}/\text{m}^3$  und einer nominellen Auflösung von  $0,3 \mu\text{g}/\text{m}^3$  in den Fraktionen PM2.5 (Teilchen bis maximal  $2,5 \mu\text{m}$  Durchmesser) und PM10 (Teilchen bis maximal  $10\mu\text{m}$  Durchmesser) erfassen. Diese Fraktionen sind gesetzlich für Luftgütemessungen vorgeschrieben. Zahlreiche Veröffentlichungen finden sich über diesen Sensor, seine Genauigkeit wird kritisch gesehen, er verfügt aber über eine gute Dynamik von wenigen Sekunden und ist im Bereich  $3\mu\text{m}$  wohl hinreichend genau, während größere Partikel insbesondere bei hoher Luftfeuchte eher ungenau gemessen werden [13]. Wenn die Abhängigkeiten der Abweichungen bekannt sind, lassen sie sich jedoch korrigieren. Mindestens für die qualitative Messung der Entwicklung der Feinstaubkonzentration lässt er sich jedoch sehr gut verwenden.

### 15.4.1 Messprinzip

Der Sensor ist seitens des Herstellers nur sehr spärlich dokumentiert. Er bedient sich des Prinzips der Laserstreuung. Das Prinzip basiert auf der Erkenntnis, dass ein einzelnes Teilchen in einem ebenen Laserstrahl wie eine Blende verhält (siehe Abb. 15.11). Sobald ein Lichtstrahl auf ein Teilchen trifft, wird er teilweise absorbiert, teilweise reflektiert (bei kleinen Teilchen vernachlässigbar) und teilweise gestreut. Diese Streuung ergibt wie bei einer Blende ein Interferenzmuster. Setzt man den Partikeldurchmesser  $\pi d$  ins Verhältnis zur Wellenlänge  $\lambda$ , so greift bei

$$0,1 \leq \pi d / \lambda \leq 10 \quad (15.10)$$



**Abb. 15.11** Prinzip der Laserstreuung

das Modell der Mie-Streuung<sup>4</sup>. Dieses stellt im optischen Fernfeld des Partikels (durch Linsensysteme in der Messkammer herstellbar) den Verlauf der Streuintensität an einem Detektor in ein komplexes Verhältnis zum Partikeldurchmesser. Bei teuren Sensoren werden Detektorzeilen mit vielen Messkanälen eingesetzt um die Interferenzmuster zu charakterisieren. Bei einem mit bekannter Geschwindigkeit vorbeifliegenden Partikel lassen aber auch zeitliche Verläufe an einem punktförmigen Detektor mit hinreichender Genauigkeit Rückschlüsse auf den Durchmesser zu. Daher wird mithilfe eines eingebauten Ventilators ein gleichmäßiger Partikelstrom erzeugt [14, 15].

Aus den zeitlich veränderlichen Mustern rechnet der in den Sensor eingebaute Mikrocontroller die Partikelgrößenverteilung nach einem vom Hersteller nicht weiter beschriebenen Algorithmus aus.

Laut Hersteller hat die Laserdiode eine Lebensdauer von 8000 h.

### 15.4.2 Ansteuerung und serielle Kommunikation

Der SDS011 sendet jede Sekunde (genau alle 1004 ms) ein Datenpaket mit den Messdaten über eine serielle Schnittstelle (9600 baud, 8 Datenbit, no parity, 1 Stopbit) aus, dieses sieht wie folgt aus:

Byte Nummer	Name	Inhalt
0	Message Header	0xAA
1	Command	0xC0
2	DATA 1	PM2.5 low Byte
3	DATA 1	PM2.5 high Byte
4	DATA 1	PM10 low Byte
5	DATA 1	PM 10 high Byte
6	DATA 1	ID Byte 1
7	DATA 1	ID Byte 2
8	Checksum	Prüfsumme (arithm. Summer der Datenbytes 1..6)
9	Message Tail	0xAB

Die 16 Bit Werte geben die Werte in  $0,1 \mu\text{g}/\text{m}^3$  an, also

$$\text{PM}_x = ((\text{PM}_x \text{ high Byte} \ll 8) + \text{PM}_x \text{ low byte}) / 10$$

Zum Auslesen dieser Information wird im Modul `uart.h/uart.c` eine UART Receive Interrupt Service Routine definiert, die beiden globalen 8-Bit Variablen `RecFlag` und `ByteRec` beschreibt:

---

<sup>4</sup>Nach Gustav Mie, deutscher Physiker 1868–1957.

```
ISR(USART_RX_vect)
{
    RecFlag=1;
    ByteRec=UDR0;
}
```

Mit den beiden Macros

```
#define UART_BYTE_RECEIVED 1
#define UART_IDLE 0
```

liefert die folgende Funktion `UARTcheckRec()` zurück, ob ein neues Byte eingetroffen ist und `UARTgetRec()` liefert das Byte selbst :

```
unsigned char UARTcheckRec()
{
    if (RecFlag)
    {
        RecFlag=0;
        return UART_BYTE_RECEIVED;
    }
    else return UART_IDLE;
}
unsigned char UARTgetRec()
{
    return ByteRec;
}
```

Das ist eleganter als globale Variablen und entkoppelt das Modul `uart.c/h` vollständig vom Hauptprogramm.

Der eigentliche Empfangsmechanismus ist ein Zustandsautomat, der dem Schema in Abb. 15.12 folgt.

In der Hauptschleife wird nun ständig der sehr schlanke Zustandsautomat des Sensors aufgerufen (polling):

```
int main(void)
{
    unsigned char erg;
    Init();
    sei();
    while (1)
    {
        erg=SDS011StateMachine();
        ...
    }
}
```

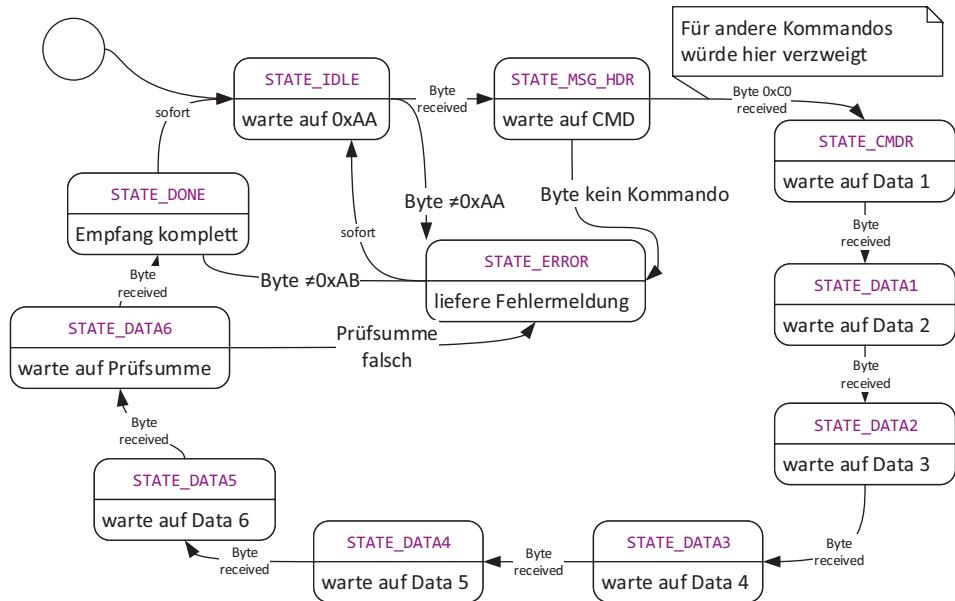


Abb. 15.12 Zustandsautomat beim Empfang der Daten des SDS011

Dieser prüft, ob ein Byte empfangen wurde und nur wenn das der Fall ist, werden die Zustände abgearbeitet, diese müssen natürlich als Makro als beliebige unterschiedliche Zahlen definiert werden, was wir uns hier ersparen:

```
unsigned char SDS011StateMachine()
{
    unsigned char Recv;
    if(UARTcheckRec())
    {
        Recv=UARTgetRec();
        switch (state)
        {
            case STATE_IDLE :      if (Recv==0xAA) state=STATE_MSG_HDR;
                                   else state=STATE_ERROR; CSCalc=0; break;
            case STATE_MSG_HDR : if (Recv==0xC0) state = STATE_CMDR;
                                   else state=STATE_ERROR; break;
            case STATE_CMDR :     PM2_5=Recv;
                                   state=STATE_DATA1; CSCalc+=Recv; break;
            case STATE_DATA1 :   PM2_5|=((unsigned int)Recv)<<8;
                                   state=STATE_DATA2; CSCalc+=Recv; break;
            case STATE_DATA2 :   PM10=Recv;
                                   state=STATE_DATA3; CSCalc+=Recv; break;
            case STATE_DATA3 :   PM10|=((unsigned int)Recv)<<8;
                                   state=STATE_DATA4; CSCalc+=Recv; break;
            case STATE_DATA4 :   PM10|=((unsigned int)Recv)<<16;
                                   state=STATE_DATA5; CSCalc+=Recv; break;
            case STATE_DATA5 :   PM10|=((unsigned int)Recv)<<24;
                                   state=STATE_DATA6; CSCalc+=Recv; break;
            case STATE_DATA6 :   PM10|=((unsigned int)Recv)<<32;
                                   state=STATE_IDLE; CSCalc+=Recv; break;
        }
    }
}
```

```

        case STATE_DATA3 : PM10 |= ((unsigned int)Recv)<<8;
                            state=STATE_DATA4; CSCalc+=Recv; break;
        case STATE_DATA4 : ID=Recv; state=STATE_DATA5;
                            CSCalc+=Recv; break;
        case STATE_DATA5 : ID|=((unsigned int)Recv)<<8;
                            state=STATE_DATA6; CSCalc+=Recv; break;
        case STATE_DATA6 : CSRec=Recv;
                            if (CSRec==CSCalc) state=STATE_CHECKSUM;
                            else state=STATE_ERROR; break;
        case STATE_CHECKSUM : if (Recv==0xAB) state = STATE_DONE;
                               else state=STATE_ERROR; break;
        case STATE_DONE : state=STATE_IDLE; return REC_COMPLETE;
        case STATE_ERROR : state=STATE_IDLE; return NotOK;
                            default: break;
        }
    }
    if (state != STATE_ERROR) return OK; else return NotOK;
}

```

Die Prüfsumme ist bei jeglicher Kommunikation mit dem Sensor das untere Byte der Summe der Datenbytes (ohne Header, Kommandobyte und Tail)

Die globalen Variablen für die Daten (PM10, PM2\_5 und ID) kann man noch mit geeigneten Get-Funktionen ausstatten, so dass der komplette Zustandsautomat in einem Modul SDS011.c/h gekapselt werden kann:

```

unsigned int SDS011GetID()
{
    return ID;
}
unsigned int SDS011GetPM2_5()
{
    return PM2_5;
}
unsigned int SDS011GetPM10()
{
    return PM10;
}

```

Da der Sensor eine begrenzte Lebensdauer hat und eine Messung pro Sekunde in der Regel nicht benötigt wird, kann man ihn per Kommando schlafen legen und aufwecken (beispielsweise jede Minute). Alternativ kann man den Sensor so konfigurieren, dass er jeweils 30 s arbeitet und dann 1 bis 30 min schläft. Dazu sendet man (beispielsweise mit den in Abschn. 7.1.6 gezeigten Funktionen) eine Bytefolge an den Sensor:

```

static const char SLEEPCMD[19] = {
    0xAA,// head
    0xB4,// command id
    0x06,// data byte 1
    0x01,// data byte 2 (set mode)
    0x00,// data byte 3 (sleep)
    0x00,// data byte 4
    0x00,// data byte 5
    0x00,// data byte 6
    0x00,// data byte 7
    0x00,// data byte 8
    0x00,// data byte 9
    0x00,// data byte 10
    0x00,// data byte 11
    0x00,// data byte 12
    0x00,// data byte 13
    0xFF,// data byte 14 (device id byte 1)
    0xFF,// data byte 15 (device id byte 2)
    0x05,// checksum
    0xAB// tail
};

```

Wird Byte 3 auf 0 gesetzt, wird der Sensor schlafen gelegt, wird es auf 1 gesetzt, wird er wieder geweckt. Die Prüfsumme muss natürlich zu 0x06 angepasst werden. Da der Sensor auf das Kommando antwortet, muss der Zustandsautomat angepasst werden. In diesem Fall lautet das zweite Byte (nach dem Header) 0xC5 statt 0xC0. In diesem Fall beinhalten die sechs folgenden Datenbytes statt der Staubkonzentrationen eine Antwort in Form eines Systemstatus. Mit dem hier aufgebauten Verständnis sollte es kein Problem sein, mithilfe des Protokollhandbuchs [16] den entsprechenden Code aufzubauen.

Mithilfe des Handbuchs lassen sich weitere Befehle an den Sensor senden: Man kann mehrere SDS011 parallel schalten, ihnen eine eigene ID geben, so dass man einzelne Sensoren ansprechen kann. Auch ist es möglich, sie nach dem Master-Slave-Prinzip reihum über ihre ID abzufragen. Dies ist jeweils ausführlich im Handbuch erläutert und folgt immer demselben hier beschriebenen Schema.

---

## Literatur

1. Mühl, T. (2020). *Elektrische Messtechnik, Grundlagen, Messverfahren, Anwendungen* (6. Aufl.). Springer.
2. Bräutigam, M. (2017). Elektronische Druckmesstechnik – Technisches Knowhow für den Anwender, JUMO Fulda. <https://campus.jumo.de>. Zugegriffen: 15. Febr. 2021.
3. Texas Instruments. (2015). TMP175 – Digital temperature sensor with two-wire interface. [www.ti.com](http://www.ti.com).

4. Roedel, W., & Wagner, T. (2011). *Physik unserer Umwelt: Die Atmosphäre*. Springer.
5. Silicon Labs. (2015). SI7021 – I2C Humidity and temperature sensor. [www.silabs.com](http://www.silabs.com).
6. Silicon Labs. (2015). AN607 – SI70xx – Humindity sensor designer's guide. [www.silabs.com](http://www.silabs.com).
7. Werner, M. (2008). *Information und Codierung*. Vieweg + Teubner.
8. Maxim Integrated. (2015). AN27 – Understanding and using redundancy checks with Maxim 1-wire and iButton Products. [www.maximintegrated.com](http://www.maximintegrated.com).
9. Freescale Semiconductor. MPL3115A2, I<sup>2</sup>C Precision Altimeter. Data Sheet Rev 4.0, 09/2015.
10. 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.
11. Freescale Semiconductor. Miguel Salhuana Application Note AN4481 – Sensor I<sup>2</sup>C Setup and FAQ. Rev 0.1, 07/2012
12. Lattanzio, L. Feinstauberfassung zur Messung der Luftqualität. <https://www.sensirion.com/de/ueber-uns/newsroom/sensirion-fachartikel/feinstauberfassung-zur-messung-der-luftqualitaet/>. Zugegriffen: 27.Dez. 2020.
13. Budde, M., & Schwarz, A. et al. Potenzial und Grenzen des kostengünstigen SDS011Partikelsensors bei der Überwachung urbaner Luftqualität, 48. Jahrestagung der Gesellschaft für Umweltsimulation. [http://www.teco.edu/~budde/publications/GUS2019\\_budde.pdf](http://www.teco.edu/~budde/publications/GUS2019_budde.pdf). Zugegriffen: 6. Juni 2020.
14. Vogel, M. (2012). Wächter der Luft. Physik im Alltag, Ausgabe 1/2012, S. 44 f. <https://www.pro-physik.de/restricted-files/94561>. Zugegriffen: 7. Juni 2020.
15. Wang, S. (2009). *Partikelgrößenbestimmung mittels eines Laser-Optischen Partikelzählers mit zwei Empfangswinkelbereichen*. Dissertation Uni Cottbus.
16. Nova Fitness Co., Ltd. (2015). Laser Dust Sensor Control Protocol V1.3. <http://www.inovafitness.com/en/a/champinzhongxin/95.html>. Zugegriffen: 27. Dez. 2020.



# Beschleunigungssensoren

16

## Zusammenfassung

In diesem Kapitel werden zwei lineare mikromechanische Beschleunigungssensoren für unterschiedliche Wertebereiche vorgestellt.

Mikromechanische Beschleunigungssensoren haben in den letzten Jahren einen unglaublichen Boom erlebt. In der Vergangenheit existierten Messverfahren für die Lage eines Objekts im Raum (beispielsweise bei Luftfahrzeugen, Raumfahrzeugen oder Schiffen), als Teil einer Lageregelung, sogenannte Inertialsensoren. Die Beschleunigungen hier entsprechen also der Größenordnung von Bruchteilen der Erdbeschleunigung. Andere Anwendungen befassen sich mit linearen Beschleunigungen oder Verzögerungen, die beispielsweise beim Unfallgeschehen eines Fahrzeugs (Crashsensor für den Airbag) auftreten oder bei sich schnell beschleunigenden Fahr- und Flugzeugen (siehe auch Kap. 26 bei der Beschleunigung der Rakete). Ihre Wertebereiche liegen bei Mehrfachem der Erdbeschleunigung (10...100 g). Aufintegration der Beschleunigung ist in verschiedenen Fällen ein Mittel, um eine Geschwindigkeit und einen zurückgelegten Weg zu bestimmen oder zu verifizieren. Schließlich gibt es Anwendungen für rotatorische Bewegungen (Drehratensensoren, siehe Kap. 17).

Die Originalversion dieses Kapitels wurde revidiert. Ein Erratum ist verfügbar unter  
[https://doi.org/10.1007/978-3-658-31709-6\\_27](https://doi.org/10.1007/978-3-658-31709-6_27)

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann  
[https://doi.org/10.1007/978-3-658-31709-6\\_16](https://doi.org/10.1007/978-3-658-31709-6_16).

In der Vergangenheit wurden diese Sensoren mit mechanischen Kreiseln realisiert. Mikromechanische Sensoren sind billig und werden in riesigen Massen hergestellt, einige Anwendungen sind:

- Lagesensorik
- Messung von Verzögerungen für passive Sicherheitssysteme (Airbag, Gurtstraffer etc.)
- Vibrationsmessung
- Aktive Federungen in Fahrzeugen
- Drehratensorik zur aktiven Stabilisierung von Kurvenfahrten und Bremsen (s. Kap. 17)
- Alarmanlagen (Neigung, Sturz)
- Sensoren für Crashtests
- Feststellen von Schadensereignissen (z.B. beim Transport)
- Videospiele
- Ermittlung von Schlafphasen
- Sturzmessung bei Festplatten
- Unfalldatenschreiber

um nur ein paar wenige zu nennen. Das Arbeitsprinzip der MEMS-Sensoren ist im nächsten Abschnitt erklärt, hier sei nur noch angemerkt, dass auf dem Markt auch analoge Beschleunigungssensoren existieren, die eine zur Beschleunigung proportionale Spannung ausgeben. Wir haben uns durchgängig für digitale Sensoren entschieden.

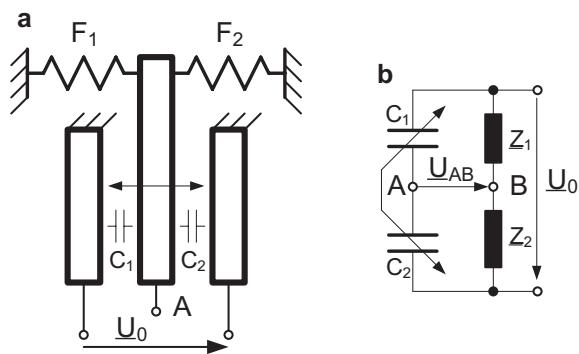
---

## 16.1 Beschleunigungssensor ADXL312

ADXL312 ist ein mikroelektromechanischer Sensor (MEMS) [2] 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 [1]. Mit einer Messrate von bis 3200 Messungen/s wird eine Bandbreite von 0 Hz 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. 16.1a. Um die Empfindlichkeit des Messfühlers zu erhöhen, wird eine Kammstruktur verwendet, wie in [1] beschrieben. Der Messfühler wird elektrisch in eine Brückenschaltung (siehe Abb. 16.1b) 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

**Abb. 16.1** ADXL312 – Prinzipieller Aufbau eines MEMS kapazitiver Beschleunigungssensors



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 Geräten einsetzen zu können. Mit einer flexiblen Interruptverwaltung und zwei Interruptausgängen kann der Baustein in komplexen Netzwerkstrukturen eingesetzt werden.

### 16.1.1 Vernetzung des ADXL312

Für die Vernetzung des Sensors, der als Slave konfiguriert ist, sind zwei serielle Schnittstellen vorgesehen: SPI und I<sup>2</sup>C. Diese Schnittstellen benutzen gemeinsam die Anschlüsse SDI (Data In), CLK (Taktsignal), SDO (Data Out) und CS (Chip Select) (siehe Tab. 16.1). 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

**Tab. 16.1** ADXL312 – Kommunikationsanschlüsse

Anschluss		I <sup>2</sup> 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 <sub>DD</sub> Oder über 10 kΩ an GND
	1	0x31		

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:

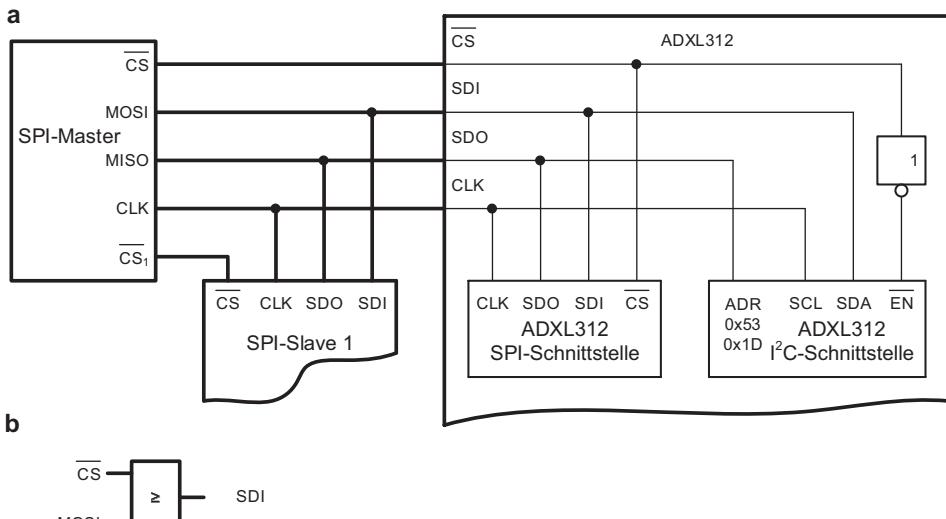
- 1. Element – die Adresse des DDR-Registers an dem die Slave-Select-Leitung angeschlossen ist;
- 2. Element – die Adresse des PORT-Registers an dem die Slave-Select-Leitung angeschlossen ist;
- 3. Element – das entsprechende Bit der Slave-Select-Leitung;
- 4. Element – „1“ wenn die Leitung angesteuert 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. 16.2a) wird die SPI-Schnittstelle dadurch angewählt, dass der Master die CS-Leitung auf Low zieht, und der Sensor wertet die ankommenden Bytes korrekt als SPI-Nachricht aus. Wenn der Master mit dem Slave1 kommuniziert ( $CS=High$ ,  $CS_1=Low$ ), findet am SDI-Eingang des Sensors ein Datenfluss statt, der zusammen mit dem Clock-Signal vom Sensor als I<sup>2</sup>C-Botschaft interpretiert werden kann. Um Busstörungen zu vermeiden, muss verhindert werden, dass am SDI-Eingang das Signal Low wird während CS High ist [2]. Ein ODER-Gatter (siehe Abb. 16.2b) 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.



**Abb. 16.2** ADXL312 – SPI-Vernetzung

Wenn die Messfrequenz niedrig ist oder lediglich die Erdbeschleunigung gemessen wird, kann der Sensor über I<sup>2</sup>C vernetzt werden, wie in Abb. 16.3 dargestellt. In diesem Fall wird der ADXL312 dafür benutzt, die horizontale Lage des elektronischen Kompasses HMC5883 festzustellen. Der CS-Anschluss ist fest mit V<sub>DD</sub> verbunden und die 7-Bit I<sup>2</sup>C-Adresse ist auf 0x53 fixiert. Weiterhin wird die Kommunikation mit dem Sensor über I<sup>2</sup>C 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 (in 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.

### 16.1.2 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.

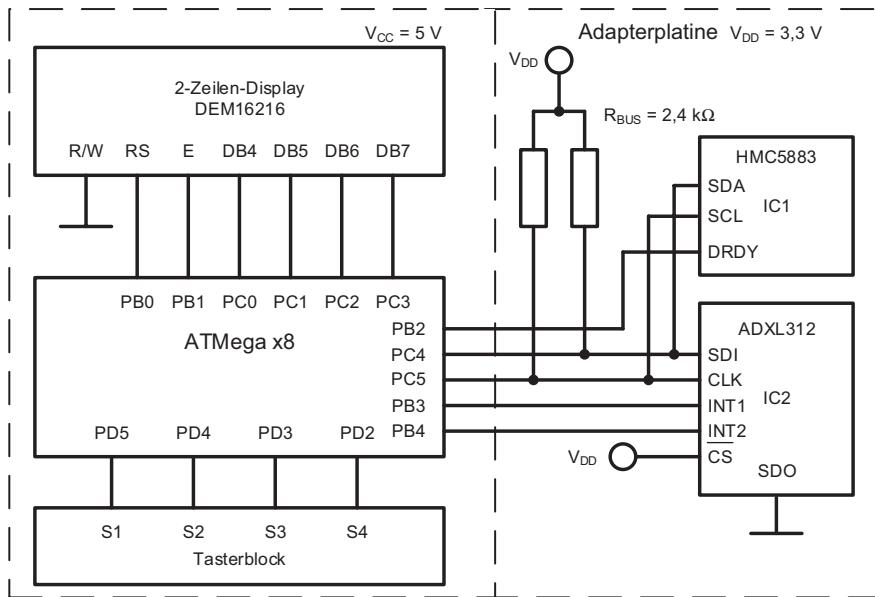


Abb. 16.3 ADXL312 – I<sup>2</sup>C-Vernetzung

#### 16.1.2.1 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*.

```
#define ADXL312_DEVICE_TYPE_ADDRESS 0x53
uint8_t ADXL312_I2C_Write_ByteReg(uint8_t ucreg_address, uint8_t ucdata_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 Ziel
    registers 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(); //Stopp
return TWI_OK;
}

```

### 16.1.2.2 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.

### 16.1.2.3 Messfrequenzeinstellung

Die Bits 3:0 vom Register *BW\_RATE* (0x2 C) 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 eingestellten 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.

### 16.1.2.4 Messdatenformat

Der gesamte Messbereich von  $\pm 12$  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. 16.2 aufgelistet. Wenn dieses Bit gesetzt ist, werden die Messwerte mit der niedrigsten

**Tab. 16.2** 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

Schrittweite umgewandelt, die Bitauflösung stellt sich automatisch mit der Wahl des Messbereiches um;

- **Bit 2 – JUSTIFY-** mit „0“ sind die Messergebnisse als 16-Bit-Zahl, rechtsbündig im Zweierkomplement Format in zwei 1-Byte-Register gespeichert; mit „1“ werden sie linksbündig gespeichert;
- **Bit 1:0 – bestimmen** laut Tab. 16.2 den Messbereich.

### 16.1.2.5 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. 16.3). 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.

**Tab. 16.3** 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 0x1 F
Z	DATA_Z_MSB 0x37	DATA_Z_LSB 0x36	OFS_Z 0x20

- **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:

```
#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.

### 16.1.2.6 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 sechs 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 I<sup>2</sup>C 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-Sensors 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(); //Stopp
    return TWI_OK;
}
```

### 16.1.3 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 (negative Werte)  $iValueMin\_i$  erfasst und gespeichert. Der Offsetwert für die i.-Achse wird mit der Gleichung:

$$iOffset\_i = \frac{iValueMax\_i + iValueMin\_i}{2} \quad (16.1)$$

entsprechend dem gewählten Messbereich berechnet. Die berechneten Offsetwerte müssen abhängig von der gewählten Schrittweite (siehe Tab. 16.3) mit einem Faktor korrigiert werden, bevor sie in die Offset-Register gespeichert werden:

$$iOffsetKorr\_i = -\left( iOffset\_i \cdot \frac{OffsetReg\_i \text{ Auflösung}}{\text{Messbereich Auflösung}} \right) \quad (16.2)$$

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
```

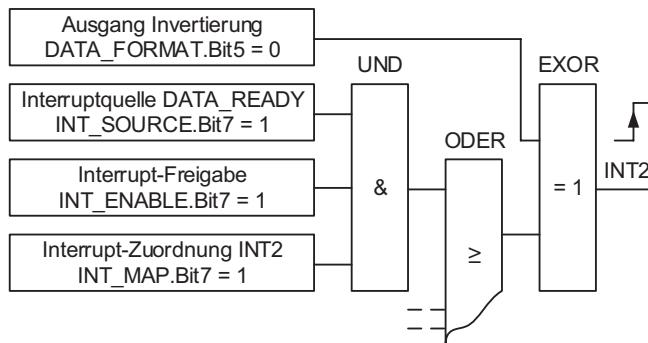
Die gemessenen Werte werden mit den Inhalten der Offsetregister intern addiert bevor sie in die Datenregister gespeichert werden. Werte um die 0 für die X- und die Y-Achse zeigen die horizontale Lage des Sensors.

### 16.1.4 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* ( $0 \times 30$ ) 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 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.
- **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. 16.4. Weitere Details bezüglich Interrupts können dem Datenblatt des Bausteins [2] entnommen werden.



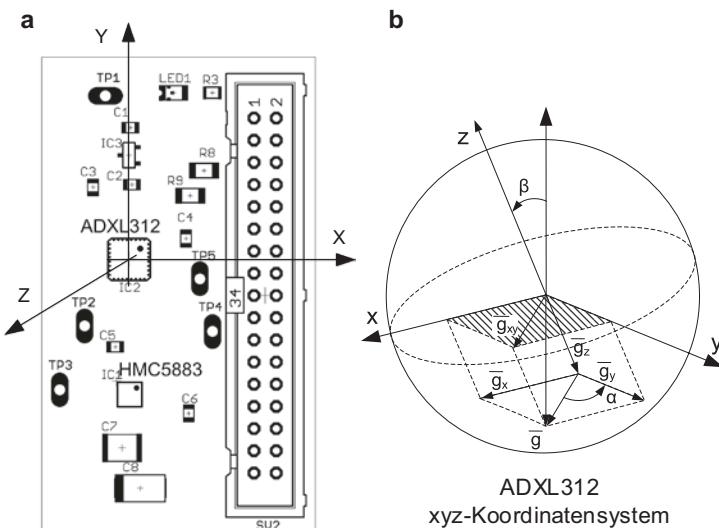
**Abb. 16.4** ADXL312 – Interrupt-Konfiguration und Pin-Zuordnung

Mit dem folgenden Programmausschnitt wird der *DATA\_READY* Interrupt zusätzlich zu den vorhandenen Interrupts eingestellt (siehe Abb. 16.4).

```
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); //Interrupt-
quelle 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
```

### 16.1.5 ADXL312 als Neigungssensor

Im Folgenden wird beispielhaft eine Möglichkeit vorgestellt, den ADXL312 als Neigungssensor zu benutzen. Die Abb. 16.5a zeigt die Achsenanordnung des Sensors, der wie in Abb. 16.3 beschaltet ist. Wenn der Winkel zwischen der positiven Richtung



**Abb. 16.5** ADXL312 als Neigungssensor

einer Achse und dem Vektor der Gravitationsbeschleunigung  $180^\circ$  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

$$g^2 = g_x^2 + g_y^2 + g_z^2 \quad (16.3)$$

### 16.1.5.1 Theoretischer Ansatz

Das Ziel ist die Berechnung des Neigungswinkels  $\beta$  (siehe Abb. 16.5b) zwischen der Vertikalen und der eigenen Z-Achse des Sensors und die Neigungsrichtung über den Winkel  $\alpha$ . Der Neigungswinkel nimmt Werte zwischen  $0^\circ$  und  $180^\circ$  an und kann mit Hilfe des CORDIC-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:

$$g_{xy} = \sqrt{g_x^2 + g_y^2}, \quad (16.4)$$

was aber für einen 8-Bit-Mikrocontroller zeitaufwendig ist. Die Neigungsrichtung wird über den Winkel  $\alpha$  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  $0^\circ$  bis  $360^\circ$  an und kann mit dem in Kap. 18 (Gl. 18.5) vorgestellten CORDIC-Algorithmus berechnet werden. Am Ende des iterativen Verfahrens für die Winkelberechnung liegt der Endvektor  $\bar{g}'_{xy}$  nahezu auf der X-Achse und liefert ein Maß für den Betrag. Laut [4] stehen die Vektorbeträge  $g_{xy}$  und  $g'_{xy}$  im folgenden Verhältnis:

$$\frac{g'_{xy}}{k} = g_{xy}, \quad k > 1 \quad (16.5)$$

wobei  $n$  die Anzahl der Iterationen bedeutet. Für  $n=8$  ist  $k \approx 1.646760258$ . Durch die Teilung der Endvektorlänge durch den Faktor  $k$  erhält man den gewünschten Betrag  $g_{xy}$ . Die im Kap. 18 vorgestellte Funktion für die Winkelberechnung mit der Ergänzung:

```
uint16_t uiLength; //Vektorbetrag
uiLength = uiValue_X * 128; //der Betrag des Endvektors wird durch
1,648 (211/128) geteilt
uiLength = uiLength / 211;
```

kann neben dem Winkel  $\alpha$  auch den Betrag  $uiLength$  des Vektors  $\bar{g}_{xy}$  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.

### 16.1.5.2 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. 16.2). Nach der Initialisierung des Beschleunigungssensors müssen die Offsetwerte wie etwa in Abschn. 16.1.3 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  $\pm 7 \cdot 2,9 \text{ mg}$  wenn:

$$iValueMax\_i + iValueMin\_i = n \cdot 8 - 1 \quad n \in N \quad (16.6)$$

was die Genauigkeit der Berechnungen beeinträchtigt. Bessere Ergebnisse können erzielt werden, wenn die mit der Gl. 16.1 berechneten Offsetwerte direkt zu den ausgelesenen Messwerten addiert werden. Mit zusätzlichen drei Additionen ergibt sich ein maximaler Offset von  $\pm 1$ .

Die Korrektur der Messwerte hat die Berechnung der Korrekturfaktoren zum Ziel, mit denen die Messwerte multipliziert werden damit Gl. 16.7 erfüllt ist.

$$g_{x\_korr}^2 + g_{y\_korr}^2 + g_{z\_korr}^2 = const. \quad (16.7)$$

- 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;
```

### 16.1.5.3 Berechnung des Neigungswinkels und der Neigungsrichtung

Mit den korrigierten Messwerten der X- und Y-Achse werden der Winkel  $\alpha$  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 acht 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;
}
```

## 16.2 MMA6525

MMA6525 ist ein 2-Achsen-Beschleunigungssensor, der Beschleunigungen im Bereich  $\pm 105,5$  g mit einer Auflösung von 0,055 g messen kann [3]. Er wurde für die Ausstattung von Airbag-Steuergeräten im Automotive-Bereich entwickelt, kann aber auch für andere Anwendungen verwendet werden, in denen hohe Beschleunigungen zu erwarten sind.

### 16.2.1 Sensoraufbau

Das Blockschaltbild des Sensors ist in Abb. 16.6 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 Hz...1000 Hz. Die digitalen Beschleunigungswerte können mit oder ohne Vorzeichen ausgelesen werden. Der Messbereich des Sensors ist

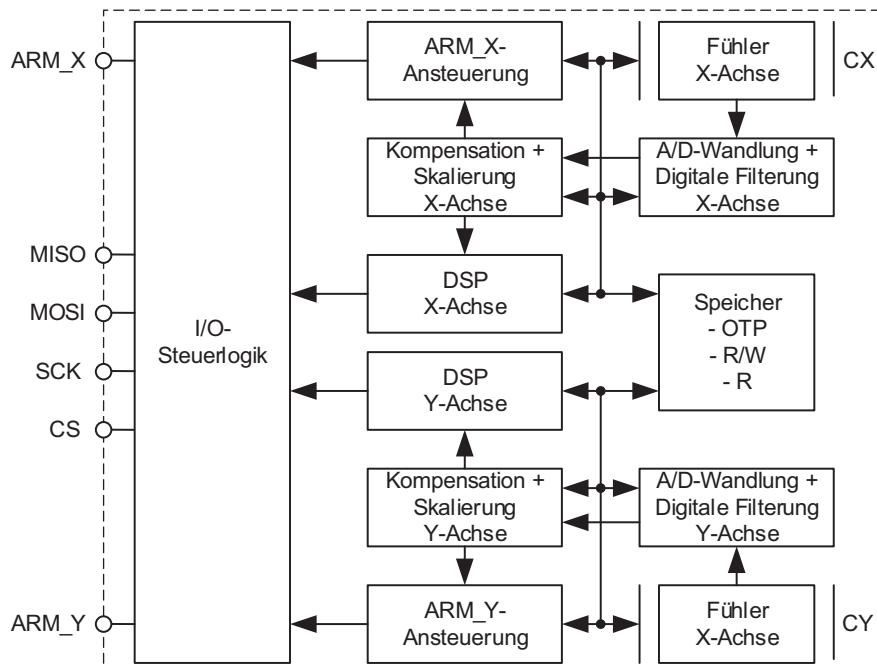


Abb. 16.6 MMA6525 – Blockschaltbild

mit vorzeichenlosen Werten im Bereich +128...+3968 und mit vorzeichenbehafteten Werten im Bereich -1920...+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 interne DSP kann den Messfühler in einen Testmodus versetzen, indem er das elektrostatische Messprinzip ausnutzt. Die Überschreitung eines einstellbaren Beschleunigungsbereiches kann in zeitkritischen Anwendungen über die Anschlüsse ARM\_X und ARM\_Y asynchron signalisiert werden, beispielsweise um einen Airbag zu entriegeln. Diese Anschlüsse können natürlich auch einen Interrupt auslösen.

## 16.2.2 Registerblock

Der Benutzer kann auf die Register im Adressbereich 0x00...0x1D zugreifen.

### 16.2.2.1 OTP<sup>1</sup>-Register

Die Register im Adressbereich 0x00...0x08 sind vom Hersteller einmalig programmiert und können nicht umprogrammiert werden. Die Adresse 0x08 speichert den Wert 0xE1 zur Identifikation des Messbereiches  $\pm 105,5$  g. Jedem Baustein wird eine einmalige 32-Bit große Identifikationsnummer zugewiesen, die an den Adressen 0x00...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.

### 16.2.2.2 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

---

<sup>1</sup>OTP – one time programmable (einmalig programmierbar).

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.

### 16.2.2.3 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.

### 16.2.3 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 (Kap. 14):

---

```
MMA65XX_pins MMA65XX_1 = { /*CS_DDR*/
    /*CS_PORT*/
    /*CS_pin*/
    /*CS_state*/      &DDRB,
                      &PORTB,
                      PB0,
                      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.

### 16.2.3.1 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 der, im Kap. 14 vorgestellte Abstraktion der I/O-Pins, zunächst eine Datenstruktur *MMA65XX\_pins* definiert, die die Pindefinitionen inklusive des SPI-Slave-Select-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.

### 16.2.3.2 Lesen eines Registers

Die Zusammensetzung eines Lesebefehls und die Antwort des Sensors, die mit dem Senden einer weiteren Nachricht folgt, ist in Tab. 16.4 dargestellt. Der Paritätsbit P hängt von der Adresse, bzw. dem Inhalt des Registers ab.

Der Inhalt eines Registers kann mit der folgenden Beispiefunktion gelesen 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 wird die Antwort des Sensors auf die vorige Nachricht gespeichert und in den letzten zwei der Inhalt des Registers entsprechend Tab. 16.4, 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;
    }
}

```

**Tab. 16.4** MMA6525 – Lesebefehl eines Registers und Antwort

```

if(!(ucCnt % 2)) ucaddress |= 0x80;
SPI_Master_Start(sdevice_pins.MMA65XXspi);
//die Chip-Select-Leitung wird auf Low gesetzt
//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);
}

```

### 16.2.3.3 Schreiben eines Registers

Der Schreibbefehl eines Registers und die zu erwartende Antwort ist nach [3] in Tab. 16.5 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.

### 16.2.3.4 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 ausgegeben;
- **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.

**Tab. 16.5** MMA6525 – Schreibbefehl eines Registers und Antwort

	MSB	Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	LSB
Befehl	P	1	0	A4	A3	A2	A1	A0	D7	D6	D5	D4	D3	D2	D1	D0		
Adresse 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	

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. 16.6. 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.

```
uint8_t MMA65XX_Get_Acceleration(MMA65XX_pins sdevice_pins, uint16_t
                                    uiop_code, uint16_t *uiaccel_data)
{
    #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 12-Bit-Beschleunigungswert
    return NO_ERROR;
}
```

**Tab. 16.6** Lesebefehl der Messwerte und Antwort des Sensors

	MSB	Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	LSB
Befehl	0	AX	1	OC	0	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	0	

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 an die Adresse uiaccel\_data gespeichert.

---

## Literatur

1. Glück, M. (2005). *MEMS in der Mikrosystemtechnik*. Springer Fachmedien Wiesbaden
2. Analog Devices Inc. (2017). ADXL312 – Digital Accelerometer. <https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL312.pdf>. Zugegriffen: 3. Apr. 2021.
3. NXP Semiconductors. MMA65xx, dual-axis SPI inertial Sensor. [www.nxp.com](http://www.nxp.com). Zugegriffen: 3. Apr. 2021.
4. Volder, J. (1959). The CORDIC computing technique. *IRE Transactions on Electronic Computers*, 8(3), 330–334.



## Zusammenfassung

In diesem Kapitel wird der Drehratensor L3GD20 beschrieben.

Drehratensensoren (oder Gyroskope) sind Inertialsensoren, die die Rotationsgeschwindigkeit eines Körpers messen. Sie dienen meist zur Lagestabilisierung und erreichten mit der Einführung des Elektronischen Stabilitätsprogramms (ESP) im Fahrzeug zum Ende des letzten Jahrhunderts große Popularität. Manche Sensoren, wie der Bosch BMI160 werden als 6-Achsen Sensoren bezeichnet und enthalten neben linearen Beschleunigungssensoren (Kap. 16) in den drei Raumachsen auch die Drehratensensoren um die drei Raumachsen. In diesem Kapitel wird der Sensor L3GD20 von ST Microelectronics als reines Gyroskop beschrieben.

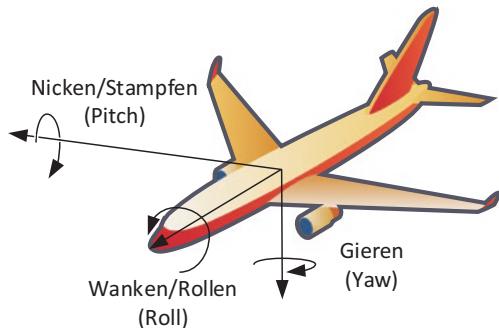
Wenn sich ein Schiff, ein Luftfahrzeug oder ein Raumfahrzeug um seine Längsachse (Longitudinalachse) dreht, spricht man von Rollen (engl. Roll), die Längsachse zeigt dabei nach vorne, Bezugspunkt ist immer der Masseschwerpunkt. Bei einem Fahrzeug spricht man von Wanken. Erfolgt die Drehung um die Normalachse (Achse zeigt von oben nach unten), spricht man von Gieren, engl. Yaw. Und eine Drehung um die Querachse (Transversalachse) heißt in der Nautik Stampfen, bei Flugzeugen und Fahrzeugen Nicken, engl. Pitch.

---

Die Originalversion dieses Kapitels wurde revidiert. Ein Erratum ist verfügbar unter  
[https://doi.org/10.1007/978-3-658-31709-6\\_27](https://doi.org/10.1007/978-3-658-31709-6_27)

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann  
[https://doi.org/10.1007/978-3-658-31709-6\\_17](https://doi.org/10.1007/978-3-658-31709-6_17).

**Abb. 17.1** Drehachsen bei einem Flugzeug



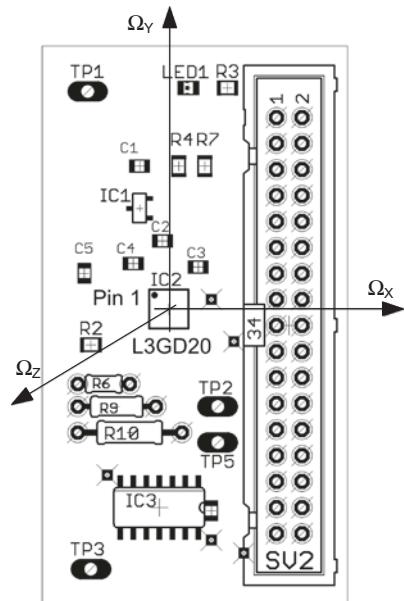
Die zugrundeliegenden Euler-Winkel merkt man sich am einfachsten mit der Rechte-Hand-Regel: Zeigt der Daumen in Richtung der jeweiligen Achse, dann deuten die gekrümmten Finger in die positive Drehrichtung. Abb. 17.1 zeigt die jeweiligen Bewegungen.

## 17.1 Gyroskop

Um Drehraten zu messen, kann man sich entweder der Corioliskraft oder des Sagnac-Effektes bedienen. Die Corioliskraft ist eine Kraft, die auf einen Körper wirkt, wenn er sich relativ zu einem sich drehenden Bezugssystem bewegt. Der Sagnac-Effekt hängt mit der Unveränderlichkeit der Lichtgeschwindigkeit zusammen. Zwischen zwei Strahlen von kohärentem Licht, die im Uhrzeigersinn und im Gegenuhrzeigersinn über Spiegel auf derselben Strecke im Kreis gelenkt werden, tritt eine Phasenverschiebung auf, wenn sich der gesamte Aufbau einschließlich der Lichtquelle dreht. Auf diesem Effekt basieren die hochempfindlichen Laserkreisel, die für die Lagestabilisierung von Raumschiffen oder Flugzeugen verwendet werden. In der Consumertechnik werden MEMS-Sensoren eingesetzt. Diese enthalten mikromechanische Inertialmassen, die federnd aufgehängt sind und durch einen elektrischen Impuls zur Schwingung angeregt werden. Durch die auftretende Corioliskraft werden sie in einem sich drehenden Bezugssystem minimal ausgelenkt, wodurch eine Kapazitätsänderung entsteht, die wiederum über die Verschiebung einer Resonanzfrequenz messbar ist.

Ein digitales Gyroskop ist ein integrierter Schaltkreis, der ein digitales Signal proportional zur Winkelgeschwindigkeit um eine oder mehreren Achsen liefert. Der Sensor L3GD20 ([1, 2]) misst die Winkelgeschwindigkeit um drei Achsen mit der positiven Messrichtung gegen den Uhrzeigersinn, wie in Abb. 17.2 dargestellt. Der Sensor bietet drei Messbereiche an:  $\pm 250$ ,  $\pm 500$  und  $\pm 2000$  grad/s. Ein integrierter Temperatursensor misst jede Sekunde die Temperatur mit einer Auflösung von 1 K 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.

**Abb. 17.2** Achsen-Anordnung des L3GD20



Die Konfiguration des Schaltkreises und das Auslesen der gemessenen Werte können über SPI oder über I<sup>2</sup>C erfolgen. Ein aufwendiges Interrupt-Konzept mit zwei einstellbaren Ausgängen kann zur Entlastung des steuernden Mikrocontrollers führen.

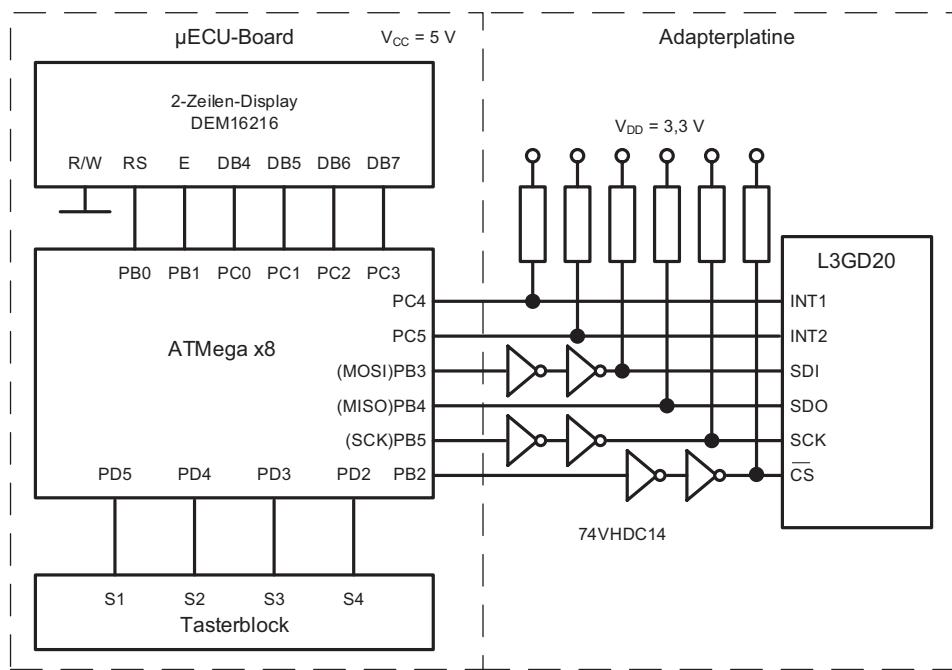
### 17.1.1 Beschaltung des L3GD20

In Abb. 17.3 wird beispielhaft die Beschaltung eines Sensors L3GD20 mit einem Mikrocontroller der Familie ATmegax8 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 I<sup>2</sup>C-Kommunikation aus Versehen initiiert wird.

### 17.1.2 Kommunikationsschnittstellen

Für die Kommunikation mit der Außenwelt stellt der Sensor zwei Interrupt-Ausgänge und vier Schnittstellenanschlüsse zur Verfügung, deren Bedeutung in Tab. 17.1 zu finden ist.

Der Baustein besitzt zwei serielle Schnittstellen: I<sup>2</sup>C und SPI, die die gleichen Anschlüsse benutzen. Eine I<sup>2</sup>C-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 I<sup>2</sup>C-Bus. Diese Schnittstelle wird während einer SPI-Übertragung deaktiviert.



**Abb. 17.3** L3GD20 – Beschaltungsbeispiel

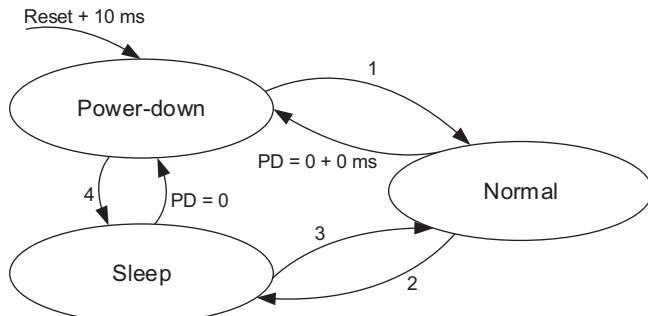
**Tab. 17.1** L3GD20 – Kommunikationsanschlüsse

Anschluss	I <sup>2</sup> C	SPI 4-wire	SPI 3-wire
CS	0	Inaktiv	Aktiv
	1	Aktiv	Inaktiv
SCL/SPC		Clock-Eingang	
SDA/SDI/SDO		Datenanschluss	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 Kap. 10.4 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. 17.5.

Für die Beispielschaltung aus Abb. 17.3 lautet die Datenstruktur zur Ansteuerung der Pins, inklusive des Chip-Select-Eingangs (Kap. 14.5):



- 1)  $(PD = 1) \wedge ((Zen = 1) \vee (Yen = 1) \vee (Xen = 1)) + 250 \text{ ms}$
- 2)  $(PD = 1) \wedge (Zen = 0) \wedge (Yen = 0) \wedge (Xen = 0) + 0 \text{ ms}$
- 3)  $(PD = 1) \wedge ((Zen = 1) \vee (Yen = 1) \vee (Xen = 1))$
- 4)  $(PD = 1) \wedge (Zen = 0) \wedge (Yen = 0) \wedge (Xen = 0)$

**Abb. 17.4** L3GD20 – Arbeitsmodi Zustandsübergangsdiagramm

```
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 acht 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. 17.2.

Das höchstwertige Bit des Startbytes bestimmt die Richtung des Datentransfers. Ist dieses Bit „0“, so wird ein Schreibvorgang gestartet, ansonsten ein Lesevorgang.

**Tab. 17.2** L3GD20 – Konfiguration des Startbytes bei der SPI-Übertragung

Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
R/W	MS	Register-Adresse					

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 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 Adress-
     zählers nach jedem ausgelesenen Register
     ucfirst_address die erste Adresse des auszulesenden Register-
     bereiches*/
    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.

### 17.1.3 Arbeitsmodi

Etwa 10 ms nach dem Einschalten befindet sich der Sensor im Power-down-Modus (Abb. 17.4), 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 (siehe Tab. 17.3). Die Umschaltzeiten die in der Abb. 17.4 nicht eingegeben sind, betragen sechs Abtastperioden oder eine, abhängig davon ob ein interner Tiefpassfilter LPF2 aktiviert ist oder nicht.

#### 17.1.3.1 Winkelgeschwindigkeitsmessung

Die 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

**Tab. 17.3** 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

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 7:6** – DR 1:0 – Auswahl der Abtastfrequenz gemäß Tab. 17.3
- **Bit 5:4** – BW 1:0 – Auswahl der Grenzfrequenz des Tiefpassfilters LPF2 gemäß Tab. 17.3
- **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. 17.3) 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. 17.4
- **Bit 3:0** – HPCF 3:0 – Grenzfrequenz-Konfiguration des Hochpasses gemäß Tab. 17.3

### 17.1.3.2 Zwischenspeichern der Messwerte

Nach der Aufnahme können die Messwerte für jede Achse vor dem Speichern zusätzlich gefiltert werden. Mit dem Bit 4 (*HOPEN*) und den Bits 1:0, (*Out\_Sel 1:0*) aus dem Steuerregister *CTRL\_REG5* (0x24) kann die Art der Filterung vor dem Speichern gewählt werden:

**Tab. 17.4** 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

- OUT1\_SEL1:0 = „00“ – keine Filterung
- OUT1\_SEL1:0 = „01“ – Filterung mit dem Hochpass HPF
- HPEN = „0“  $\wedge$  OUT1\_SEL1 = „1“ – Filterung mit dem Tiefpass LPF2
- HPEN = „1“  $\wedge$  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 FIFO<sup>1</sup>-Datenpuffer gespeichert werden.

#### 17.1.3.2.1 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-Format<sup>2</sup> ist in Tab. 17.5 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.

#### 17.1.3.2.2 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.

**Tab. 17.5** 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

<sup>1</sup>FIFO – first in first out.

<sup>2</sup>Little-Endian-Format – in einem Zwei-Byte-Wort wird das niederwertige Byte an der ersten Adresse gespeichert, im Big-Endian-Format wird zunächst das höchswertige 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

#### 17.1.3.2.2.1 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.

#### 17.1.3.2.2.2 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.

#### 17.1.3.2.2.3 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.

### 17.1.3.3 Auslesen der Messwerte

Die gespeicherten Messwerte werden immer aus den Ausgangsregistern (Tab. 17.5) ausgelesen. Das Auslesen kann seriell entweder über SPI oder I<sup>2</sup>C stattfinden. Bei der Wahl

der Schnittstelle und des Bustaktes muss die gewählte Messfrequenz berücksichtigt werden.

#### 17.1.3.3.1 Auslesen der direkt gespeicherten Messwerte

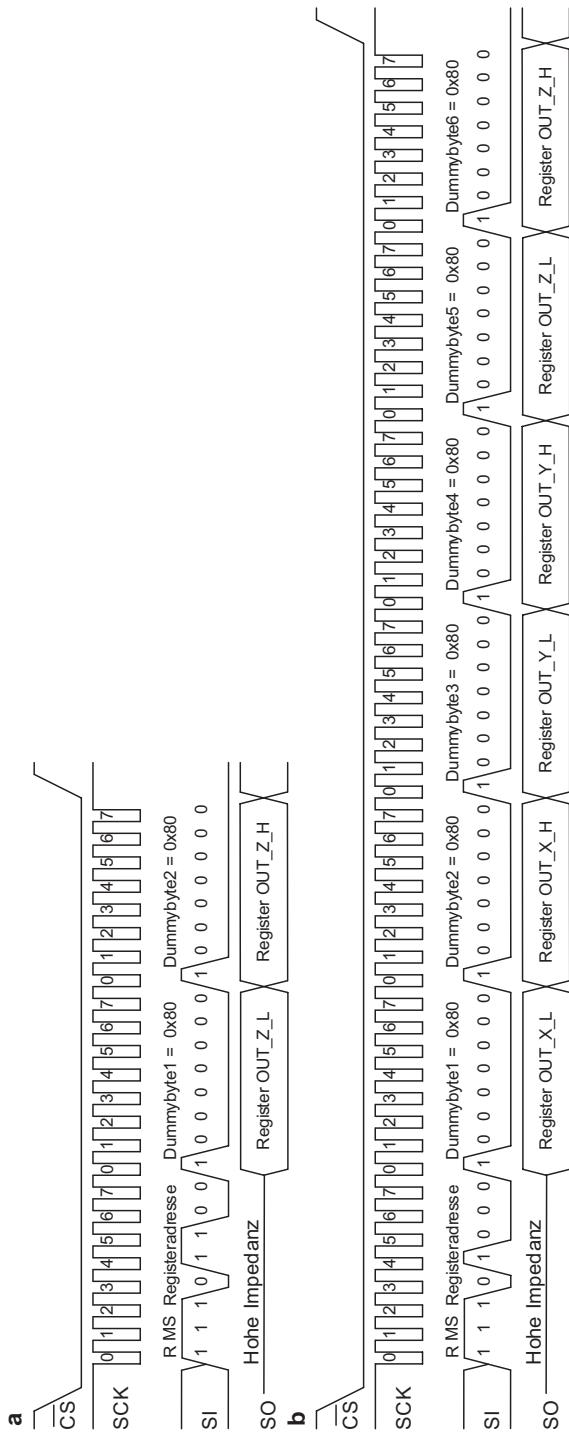
Wenn alle drei Messachsen frei geschaltet sind, müssen innerhalb einer Abtastperiode (1/ODR) drei Messwerte a 2 Bytes, also insgesamt 48 Datenbits übertragen werden, dazu kommen noch die Steuerbits. Um festzustellen ob ein neuer Datensatz vorhanden ist, kann das Bit 3 (ZYXDA) vom Register *STATUS\_REG* (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. 17.5a oder für alle drei Achsen in einem Lesevorgang wie in Abb. 17.5b. In der Abb. 17.5b 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. 17.5a 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);
```

#### 17.1.3.3.2 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. 17.1.3.2.2) wird ein Interrupt



**Abb. 17.5** L3GD20 – Auslesen der Ausgangsregister

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:  $f_{\text{Bit}} = f_q / 4 = 4 \text{ MHz}$
- minimale Bitdauer:  $T_{\text{Bit}} = 1 / f_{\text{Bit}} = 250 \text{ ns}$
- höchste Datenrate des Sensors:  $\text{ODR}_{\max} = 760 \text{ Hz}$
- kürzeste Abtastperiode:  $T_A = 1 / \text{ODR}_{\max} \approx 1,315 \text{ ms}$
- FIFO-Speicherzeit:  $T_{\text{FIFO}} = 32 \times T_A = 42, \text{ ms}$
- gesamte Bitzahl:  $N_{\text{Bit}} = (1 \text{ Befehl-Byte} + 3 \text{ Achsen} \times 2 \times 32 \text{ Byte}) \times 8 \text{ Bit} = 1544 \text{ Bit}$
- 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.  $929 \mu\text{s}$  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  $T_{\text{FIFO}}$  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. 17.5 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.

#### 17.1.3.4 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. 17.6) 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. 17.7), wird in das Register *INT1\_SRC* (0x31) gespeichert. Dieses Register kann nur gelesen werden und hat folgende Konfiguration:

**Tab. 17.6** Interrupt1 – Konfigurationsregister

AND/OR	LIR	ZHIE	ZLIE	YHIE	YLIE	XHIE	XLIE
--------	-----	------	------	------	------	------	------

**Tab. 17.7** INT1 – SRC-Register

0	IA	ZH	ZL	YH	YL	XH	XL
---	----	----	----	----	----	----	----

**Tab. 17.8** 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

- **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.

#### 17.1.3.4.1 Grenzwerteinstellung für den Interrupt 1

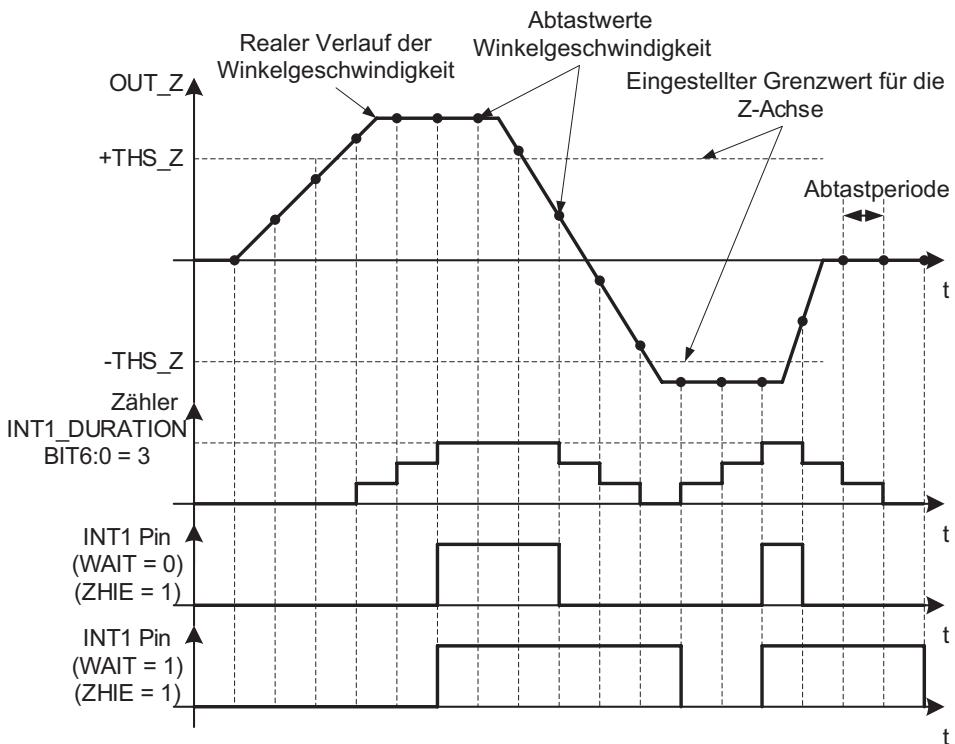
Um eine Überwachung der Messwerte zu gewährleisten, gibt es für jede Achse jeweils ein 16-Bit-Register, in dem der Betrag des Grenzwertes gespeichert wird. Der interne A/D-Wandler hat eine 16-Bit-Auflösung, gemessen wird aber in beiden Drehrichtungen, deshalb kann der Betrag höchstens 15 Bit groß sein. In der Tab. 17.8 sind die Grenzwertregister-Paare und ihre Adressen für alle Achsen aufgelistet.

#### 17.1.3.4.2 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“  $\wedge$  *INT1\_SEL1* = „1“ – Filterung mit dem Tiefpass LPF2;
- *HPEN* = „1“  $\wedge$  *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. 17.6 dargestellt.



**Abb. 17.6** L3GD20 – Interrupt1-Beispiel

### 17.1.3.5 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 genug 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* gespeichert sind, wird folgendermaßen berechnet:

$$cDeltaTemp = (-1) \cdot (cTemp2 - cTemp1)$$

Ein positives Ergebnis bedeutet Temperaturerhöhung.

## Literatur

1. STMicroelectronics. L3GD20 – MEMS motion sensor: Three-axis digital output gyroscope. [www.st.com](http://www.st.com)
2. STMicroelectronics. AN4505 Application Note. L3GD20: 3-axis digital output gyroscope. [www.st.com](http://www.st.com)



# Magnetfeldsensoren

18

## Zusammenfassung

In diesem Kapitel wird der Magnetfeldsensor HMC5883 vorgestellt.

Magnetfeldmessungen finden in den verschiedensten Gebieten Anwendung, beispielsweise in der Messung des Erdmagnetfelds (Kompass), zur rückwirkungslosen Messung von elektrischem Strom oder als Näherungssensor. In diesem Kapitel stellen wir einen Magnetfeldsensor vor, der sich vor allem als Kompass eignet.

Magnetometer dienen zur Messung der magnetischen Flussdichte  $\vec{B}$ . Diese wird in der SI<sup>1</sup>-Einheit Tesla (T) gemessen. Im SI-System ist ein Tesla definiert als

$$1 \text{ T} = 1 \frac{\text{V} \cdot \text{s}}{\text{m}^2} = 1 \frac{\text{kg}}{\text{A} \cdot \text{s}^2} \quad (18.1)$$

und Messbereiche von Magnetometern bewegen sich in einer Größenordnung von circa  $10^{-15} \text{ T}$  bis  $10 \text{ T}$ . Bisweilen werden magnetische Flussdichten auch in Gauss angegeben,

<sup>1</sup> SI – Internationales Einheitssystem.

Die Originalversion dieses Kapitels wurde revidiert. Ein Erratum ist verfügbar unter  
[https://doi.org/10.1007/978-3-658-31709-6\\_27](https://doi.org/10.1007/978-3-658-31709-6_27)

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann  
[https://doi.org/10.1007/978-3-658-31709-6\\_18](https://doi.org/10.1007/978-3-658-31709-6_18).

da der Physiker und Mathematiker Carl Friedrich Gauß im Jahr 1832 das erste Magnetometer entwickelt hatte.

$$1 \text{ Gs} = 10^{-4} \text{ T} \quad (18.2)$$

Die magnetische Flussdichte ist über die Materialgleichungen der Elektrodynamik mit der magnetischen Feldstärke  $\vec{H}$  verknüpft:

$$\vec{B} = \mu \vec{H} \quad (18.3)$$

Die Permeabilitätszahl  $\mu$  ist  $\mu = \mu_0 \mu_r$  und die magnetische Feldkonstante

$$\mu_0 = 1,25666 \dots 10^{-6} \frac{\text{N}}{\text{A}^2} \approx 4\pi \cdot 10^{-7} \frac{\text{N}}{\text{A}^2} \quad (18.4)$$

Die dimensionslose relative Permeabilitätszahl  $\mu_r$  beträgt im Vakuum (und näherungsweise in Luft) 1.

Aus dem Ampèreschen Gesetz

$$\oint \vec{H} d\vec{s} = I \quad (18.5)$$

lässt sich die magnetische Flussdichte für verschiedene geometrische Anordnungen herleiten und berechnet sich beispielsweise um einen langen zylinderförmigen Halbleiter zu:

$$|\vec{B}| = \frac{I}{2\pi r} \mu \quad (18.6)$$

In Zahlen: Im Abstand von 5 cm von der Achse eines geraden Leiters, der einen Strom von 50 A führt, beträgt die magnetische Flussdichte

$$|\vec{B}| = \frac{50 \text{ A}}{2\pi \cdot 0,05 \text{ m}} \cdot 4\pi \cdot 10^{-7} \frac{\text{N}}{\text{A}^2} = 2 \cdot 10^{-4} \text{ T} \quad (18.8)$$

Es gibt sehr viele verschiedene Möglichkeiten, Magnetfelder zu messen. Zum einen können Induktionseffekte genutzt werden, die auftreten, wenn sich ein Leiter (Spule) in einem Magnetfeld bewegt. Zum anderen existieren verschiedene galvanomagnetische Effekte ([1]), wie der anisotrope magnetoresistive Effekt (AMR) oder der Hall-Effekt und viele andere mehr.

Der AMR-Effekt basiert auf der Beobachtung, dass sich der elektrische Widerstand mancher ferromagnetischer (also magnetisierter) Materialien abhängig von der Stärke und Richtung des magnetischen Feldes ändert. Viele Magnetfeld-Sensoren der Konsumerelektronik basieren auf diesem Effekt, so auch der von uns beschriebene.

Der Hall-Effekt beruht auf der Tatsache, dass Ladungsträger in einem Halbleiter durch die Lorenzkraft unter Einwirkung eines Magnetfelds aus ihrer Flussrichtung abgelenkt werden. Dadurch lässt sich an einem Halbleiter, der von einem Strom durchflossen wird, senkrecht zur Stromrichtung eine Spannung messen, wenn wiederum senkrecht dazu ein Magnetfeld einwirkt.

## 18.1 HMC5883-Magnetfeldsensor

Der Baustein HMC5883 ([3]) ermöglicht die Messung der Flussdichte magnetischer Felder in der Stärke des erdmagnetischen Feldes. Das Messverfahren basiert auf dem oben bereits kurz erwähnten anisotropen magnetoresistiven Effekt (AMR). Damit Magnetfeldsensoren eine höhere Empfindlichkeit erreichen, werden die magnetoresistiven 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  $90^\circ$  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.

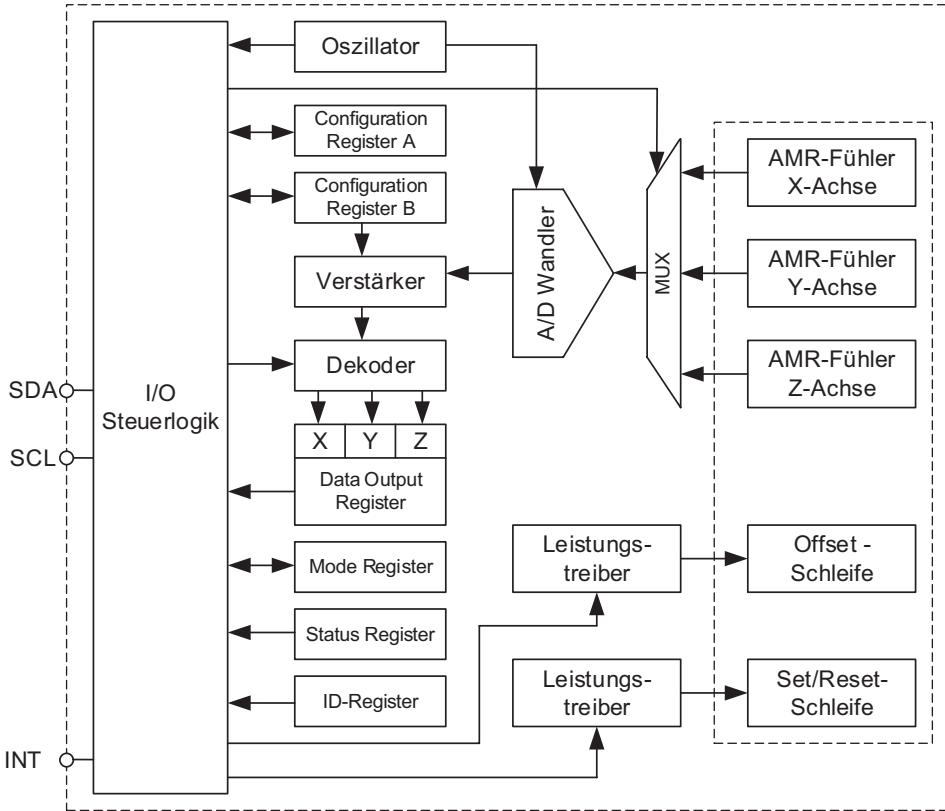
Das natürliche Erdmagnetfeld weist in Europa eine magnetische Flussdichte von ca. 40...50 Mikrotesla ( $\mu\text{T}$ ) auf [2]. Große ferromagnetische Körper sowie Ablagerungen können zu lokalen magnetischen Anomalien von einigen  $\mu\text{T}$  führen. Der HMC5883 kann die magnetische Flussdichte nach drei senkrecht aufeinanderstehenden Achsen messen, und zwar in einem Bereich von 1 Milligauss (mG) bis ca. acht Gauss, was in SI zu  $0,1 \mu\text{T} \dots 800 \mu\text{T}$  entspricht. Somit kann der Baustein zum Vermessen von magnetischen Flussdichten, als Kompass oder zur Feststellung lokaler Anomalien des erdmagnetischen Feldes verwendet werden.

### 18.1.1 Aufbau des HMC5883

Ein Blockschaltbild des Magnetfeldsensors HMC5883 ist in Abb. 18.1 dargestellt.

#### 18.1.1.1 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. 18.1.3 beschrieben wird.



**Abb. 18.1** Blockschaltbild des Sensors HMC5883

### 18.1.1.2 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 prioritär 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.

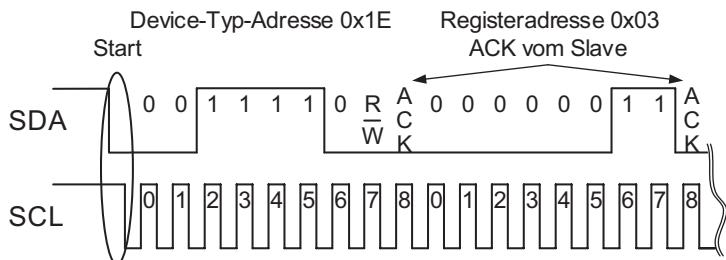
### 18.1.1.3 Serielle Kommunikation

Der Baustein ist mit einer seriellen Schnittstelle ausgestattet, die das I<sup>2</sup>C-Protokoll im Standard- und Fast-Modus implementiert und eine Datenübertragung mit bis zu 400 kBit/s ermöglicht. Die 7-Bit-Adresse des Slaves lautet 0x1E.

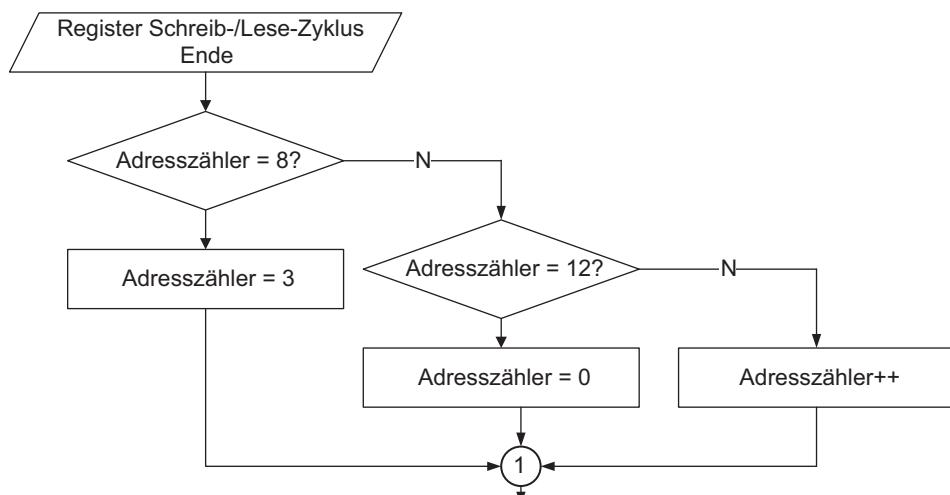
### 18.1.1.4 Registerblock

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. 18.2 wird der Zeiger auf die Adresse 0x03 gesetzt.

Der Master adressiert den Baustein über seine I<sup>2</sup>C-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. 18.3 geändert. Die Steuerlogik kontrolliert die Registeradressierung und das Auslesen der Messwertregister. Dadurch wird die Software-Konfiguration des Bausteins beschleunigt.



**Abb. 18.2** HMC5883 – wahlfreie Registeradressierung



**Abb. 18.3** HMC5883 – Ansteuerung des internen Adresszählers

#### 18.1.1.4.1 Konfiguration Register

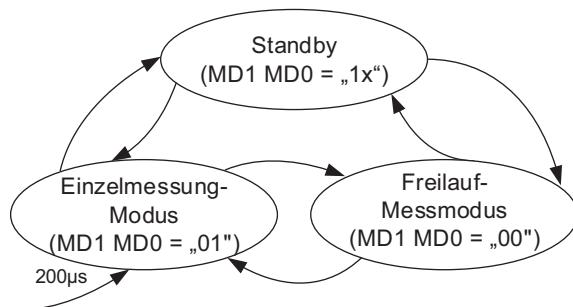
Drei Register dienen der gesamten Konfiguration des Bausteins. Über das Konfigurationsregister A (Adresse 0x00) werden die Messmodi, die Messrate und die Anzahl der gemittelten Messwerte für jeden Messzyklus bestimmt.

- **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  $\pm 88 \mu\text{T}$  für „000“ und  $\pm 810 \mu\text{T}$  für „111“ ausgewählt.

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. 18.4 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.

**Abb. 18.4** HMC5883 –  
Betriebsmodi  
Zustandsübergangsdiagramm



```

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(); //Stopp
    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 acht Messungen
                                //wird gebildet
#define OUT_RATE_1         0x00    //0,75 Hz 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);

```

**Tab. 18.1** HMC5883 – Datenregister

Achse	X		Z		Y	
Register	MSByte	LSByte	MSByte	LSByte	MSByte	LSByte
Adresse	0x03	0x04	0x05	0x06	0x07	0x08

### 18.1.1.4.2 Messwert Register

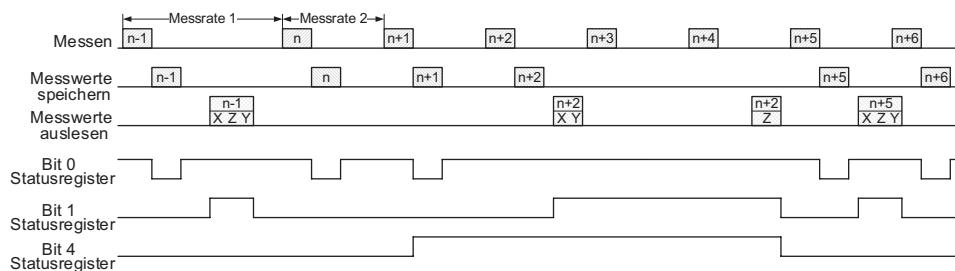
Ein neuer Messwert wird als Mittelwert zweier Aufnahmen berechnet. Vor jeder Aufnahme wird je ein Entmagnetisierungsfeld mit wechselnder Richtung erzeugt. Dadurch vermeidet man eine eventuelle Dauermagnetisierung der magnetoresistiven Widerstände. Die gemessenen Werte der magnetischen Flussdichte werden für jede Achse in je zwei Register im Big-Endian-Format gespeichert wie in Tab. 18.1 dargestellt.

### 18.1.1.4.3 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. 18.5 dargestellt.



**Abb. 18.5** HMC5883 – Freilauf-Messbetrieb – Schematische Darstellung mit asynchronem Datenlesen

#### 18.1.1.4.4 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.

### 18.1.2 HMC5883 Messwerte lesen

Die Dauer einer Messung mit Mittelung von acht Aufnahmen dauert laut [3] 6 ms. Das Auslesen der sechs Datenregister kann beispielsweise mit der Übertragung von neun 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.

#### 18.1.2.1 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  $n$ -ten Messwerte wie im folgenden Codeausschnitt durchgeführt werden:

```
HMC5883_Write_ByteReg(MODE_REG, 0x01); //Start einer neuen Messung  
HMC5883_Read_DataReg(uiData); //Auslesen der sechs 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.

```
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);
```

### 18.1.2.2 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.

#### 18.1.2.2.1 Asynchrones Lesen im Freilauf-Betrieb

So wie in Abschn. 18.1.1.4.2 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.

#### 18.1.2.2.2 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 dieses Anschlusses 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.

## 18.1.3 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 Gs (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 fünften 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 fünften Messbereich sind die Werte aus der Tab. 18.2 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. 18.3 gespeichert.

### Beispiel

$$\text{X\_Offset} = (+\text{X\_Mess} + (-\text{X\_Mess})) / 2 = (569 - 550) / 2 = +9$$

$$\text{V\_X\_Korr} = \text{X\_Test} / (\text{+X\_Mess} - \text{X\_Offset}) = 510 / 560$$

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  $\mu\text{T}$  ausgedrückt. Bei Änderung des Messbereiches oder der Umgebungstemperatur sollte der Kalibriervorgang wiederholt werden.

**Tab. 18.2** Beispiel Kalibrierung: gemessene Werte

+X_Mess	-X_Mess	+Y_Mess	-Y_Mess	+Z_Mess	-Z_Mess
+569	-550	+525	-525	+499	-480

**Tab. 18.3** 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

### 18.1.4 HMC5883 als elektronischer Kompass

Der Sensor kann wegen seiner hohen Empfindlichkeit als elektronischer Kompass verwendet werden. Das erdmagnetische Feld verläuft in unseren Breiten parallel zur Erdoberfläche und als vektorielle Größe ist es immer auf den magnetischen Nordpol gerichtet [4]. 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  $0^\circ$  soll die Richtung der positiven Y-Achse auf den magnetischen Nordpol zeigen. Als Voraussetzung für die Winkelbestimmung müssen sich die Punkte  $P_i(X_{Mess\_i} / Y_{Mess\_i})$  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 [4] 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.

$$\begin{cases} X_{Offset} = (X_{Max} + X_{Min})/2 \\ Y_{Offset} = (Y_{Max} + Y_{Min})/2 \end{cases} \quad (18.8)$$

Unter Berücksichtigung der Offsetwerte wird der Schritt 1. wiederholt und geprüft, dass  $X_{Max} \approx X_{Min}$  und  $Y_{Max} \approx Y_{Min}$ . Die aktuellen Maxima  $X_{Max}$  und  $Y_{Max}$  werden gespeichert. Wenn z. B.  $X_{Max} > Y_{Max}$ , dann werden die Messwerte weiterhin folgendermaßen korrigiert:

$$\begin{cases} X_{Korr} = X_{Mess} - X_{Offset} \\ Y_{Korr} = ((Y_{Mess} - Y_{Offset}) \cdot X_{Max})/Y_{Max} \end{cases} \quad (18.9)$$

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 mithilfe der trigonometrischen Funktion Arkustangens berechnet werden.

$$\beta = atan\left(\frac{X_{Korr}}{Y_{Korr}}\right) \quad (18.10)$$

### 18.1.5 Winkelberechnung mit dem CORDIC-Algorithmus

Um den Winkel aus Gl. 18.3 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 (LUT<sup>2</sup>). 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 [6, 7].

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  $\alpha_0$  zu einem Vektor  $\vec{V}_1$  mit den Komponenten  $X_1$  und  $Y_1$  (siehe Abb. 18.6)

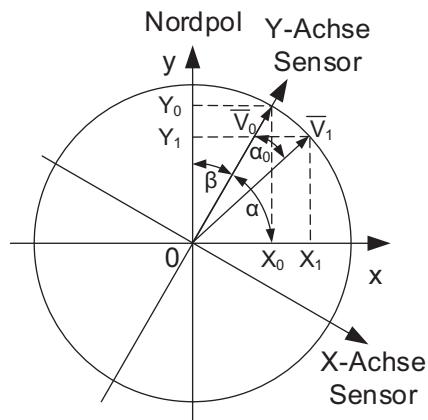
$$\begin{cases} X_1 = X_0 + Y_0 \cdot \tan \alpha_0 \\ Y_1 = Y_0 - X_0 \cdot \tan \alpha_0 \end{cases} \quad (18.11)$$

In einem iterativen Vorgang wird der Vektor  $\vec{V}_1$  weiter in diskreten Schritten rotiert bis die Ordinate des resultierenden Vektors ungefähr Null ist.

$$\begin{cases} X_{i+1} = X_i + Y_i \cdot \sigma_i \cdot \tan \alpha_i & \text{mit} \\ Y_{i+1} = Y_i - X_i \cdot \sigma_i \cdot \tan \alpha_i \end{cases} \quad (18.12)$$

$$\sigma_i = \begin{cases} +1, \text{ wenn } Y_i > 0 \\ -1, \text{ wenn } Y_i < 0 \end{cases} \quad (18.13)$$

**Abb. 18.6** HMC5883 – elektronischer Kompass



<sup>2</sup>Lookup table.

**Tab. 18.4** Stützwerte für den CORDIC-Algorithmus

$\tan \alpha_i$	$2^0$	$2^{-1}$	$2^{-2}$	$2^{-3}$	$2^{-4}$	$2^{-5}$	$2^{-6}$	$2^{-7}$
$\alpha_i$	450	265	140	71	36	18	9	4

Mit  $\tan \alpha_i = 2^{-i}$  lassen sich die Komponente 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. 18.4 sind die dazugehörigen Winkel  $\alpha_i$  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

$$\alpha = \sum_{i=0}^{n-1} \sigma_i \cdot \alpha_i \quad (18.14)$$

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 acht 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;  
}
```

---

## Literatur

1. Tränkler, H.-R., & Reindl, L. M. (Hrsg.) (2014). *Sensortechnik*. Springer Vieweg
2. Knödel, K., Krummel, H., & Lange, G. (Hrsg.). (2005). *Geophysik*. Springer.
3. Honeywell International Inc. (2017). HMC5883L – Three-Axis Digital Compass IC. <https://aerospace.honeywell.com>
4. Honeywell International Inc. (2015). Michael Caruso application of Magnetoresistive sensors in navigation systems. [www.honeywell.com](http://www.honeywell.com)
5. Honeywell International Inc. (2015). AN203 – Compass heading using magnetometers. [www.honeywell.com](http://www.honeywell.com)
6. Volder, J. (1959). The CORDIC computing technique. *IRE Transactions on Electronic Computers*, 8(3), 330–334.
7. Kuhlmann, M., Parhi, K. K., & P-CORDIC. (2002). A precomputation based rotation CORDIC algorithm. *EURASIP Journal on Applied Signal Processing*, 9, 936–943.



## Zusammenfassung

In diesem Kapitel werden ein Ultraschallsensor und ein optischer Näherungssensor vorgestellt.

Näherungssensoren (engl. Proximity Sensor) sind Sensoren, die ein Objekt ohne direkten Kontakt detektieren können. Nach Messprinzip unterscheidet man Ultraschall-, kapazitive, induktive und optische Detektoren. Je nach Messprinzip zeigen Näherungssensoren unterschiedliche Empfindlichkeiten und können daher teilweise nur zur digitalen Detektion einer Annäherung (Näherungsschalter) oder zur Entfernungsmessung herangezogen werden. Die beiden in diesem Kapitel vorgestellten Sensoren eignen sich zur Entfernungsmessung in einem Abstand von wenigen Zentimetern bis wenigen Metern.

## 19.1 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

---

Die Originalversion dieses Kapitels wurde revidiert. Ein Erratum ist verfügbar unter  
[https://doi.org/10.1007/978-3-658-31709-6\\_27](https://doi.org/10.1007/978-3-658-31709-6_27)

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann  
[https://doi.org/10.1007/978-3-658-31709-6\\_19](https://doi.org/10.1007/978-3-658-31709-6_19).

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 Longitudinalwelle<sup>1</sup> 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 [6] und ist von der Temperatur, der relativen Luftfeuchtigkeit und dem Druck abhängig. Die Gleichung Gl. 19.1 [1] beschreibt die Temperaturabhängigkeit der Schallgeschwindigkeit  $c$  von der Lufttemperatur  $T$  in K:

$$c = c_0 \cdot \sqrt{1 + \frac{T}{273,15}} \quad (19.1)$$

wobei  $c_0$  die Ausbreitungsgeschwindigkeit des Schalls bei 0 °C ist. Die Schallgeschwindigkeit erhöht sich mit 5 m/s bei einer Zunahme des Luftdrucks von 30 hPa [1] und gleichbleibender Lufttemperatur. Eine Änderung der relativen Luftfeuchte von 0 auf 100 % führt bei einer Lufttemperatur von 20 °C zu einer Erhöhung der Schallgeschwindigkeit mit ca. 1 m/s [1]. Die Schallintensität bei konstanter Frequenz nimmt mit der Entfernung quadratisch 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.

### 19.1.1 Messprinzip

Ein Ultraschallsensor misst die Zeit  $\Delta t$  zwischen dem Ausstrahlen einer Schallpulsfolge und dem Empfang der reflektierten Schallwelle. Der Abstand  $d$  vom Sensor bis zum Objekt wird folgendermaßen berechnet:

$$d = c \cdot \frac{\Delta t}{2} \quad (19.2)$$

Nach der Anzahl der benutzten Ultraschallwandler unterscheidet man zwischen Einkopf- und Zweikopfsystemen. 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° [2]. In der Sendephase wird der Wandler von der steuernden Elektronik zum Schwingen angeregt und erzeugt eine Schallpulsfolge mit

---

<sup>1</sup>Longitudinalwelle ist eine Welle bei der die Richtung der Druckveränderung dieselbe ist wie die Richtung der Wellenausbreitung [26].

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  $\alpha$ , ist aber wesentlich kleiner als beim Einkopfsystem. Der kleinste Abstand  $d$  zu einem Objekt, der mit diesem System erfasst werden kann, ist (siehe Abb. 19.1):

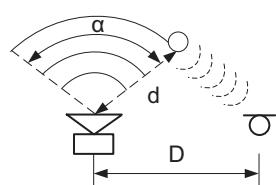
$$d = \frac{D/2}{\sin \frac{\alpha}{2}} \quad (19.3)$$

wobei  $D$  der Abstand zwischen zwei Schallwandlern ist. Dieses System ermöglicht wegen des größeren Öffnungswinkels die Detektion mehrerer Objekte.

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 Ultraschall-sensoren in einem Raum aktiv sind, können sie im Synchron- oder Multiplexbetrieb

**Abb. 19.1** Ultraschall-Zweikopfsystem



arbeiten. Im Synchronbetrieb senden alle Sensoren gleichzeitig ihre Schallpulse, vorausgesetzt sie stören sich nicht gegenseitig. Im Multiplexbetrieb werden die Sensoren nacheinander aktiviert und ausgewertet.

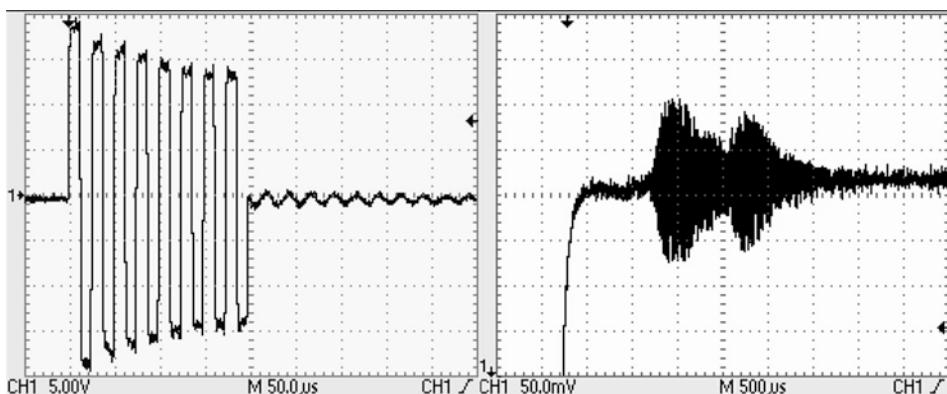
### 19.1.2 SRF08 – Ultraschall-Messmodul

SRF08 ist ein diskret aufgebauter, Mikrocontroller gesteuerter Ultraschallsensor ([7]). Er benutzt Schallwandler mit einem Öffnungswinkel von ca.  $60^\circ$  und erzeugt für die Detektion, bzw. Messung Schallpulse mit einer Frequenz von 40 kHz (Abb. 19.2 rechts). 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 I<sup>2</sup>C-Bus.

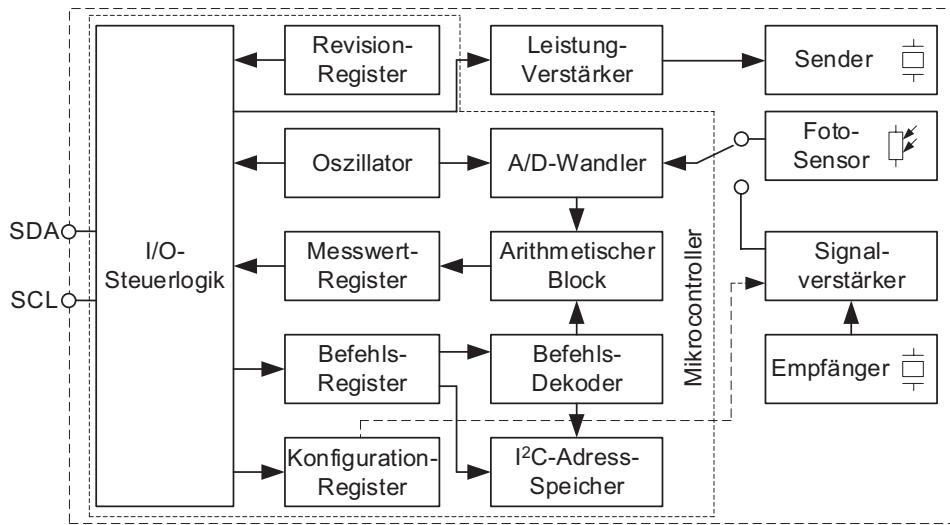
#### 19.1.2.1 Aufbau

Ein Blockschaltbild des Ultraschallsensors SRF08 ist in Abb. 19.3 dargestellt. Der Mikrocontroller regt den Schallsender über einen Pegelwandler und einen Verstärker mit einer elektrischen Pulsfolge zum Schwingen an (siehe Abb. 19.2 links). Die empfangenen Schallpulse werden mit dem Schallempfänger in elektrischen Pulse umgewandelt (Abb. 19.2a), die zur Verbesserung der Reichweite vor der Digitalisierung verstärkt werden können.

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. 19.2 in Längen umgerechnet werden. Der Sensor besitzt auch einen Fotowiderstand, mit dem die Lichthelligkeit geschätzt werden kann. Die Abstandsmessungen werden über das Reichweiten- und Verstärkungsregister konfiguriert. Mit dem Speichern eines Wertes aus dem Bereich [0, 31] in das



**Abb. 19.2** SRF08 gesendete Pulse (links) und empfangene Pulse (rechts)



**Abb. 19.3** SRF08 – Blockschaltbild

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 µs 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 I<sup>2</sup>C-Adresse.

### 19.1.2.2 Serielle Kommunikation

Die Konfiguration des Sensors, der Start neuer Messungen, das Auslesen der Messwerte, und die Änderung der Adresse erfolgen über den I<sup>2</sup>C-Bus. Der Sensor ist als Slave mit der I<sup>2</sup>C-7-Bit-Adresse 0x70 konfiguriert, 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 I<sup>2</sup>C-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 STOPP-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_REG0
    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 I<sup>2</sup>C-Übertragung mehrere Register lesen kann.

### 19.1.2.3 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.

### 19.1.2.3.1 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(); //stopp
    return TWI_OK;
}

```

### 19.1.2.3.2 Artificial Neural network<sup>2</sup>- Messmodus

Die Messungen im Artificial Neural Network – Modus werden mit einem der Befehls-codes 0x53, 0x54 oder 0x55 gestartet. Die Messungen finden unabhängig von den Einstellungen mit dem höchsten Verstärkungsfaktor statt. Das gesamte Mess-Zeit-fenster wird in  $32 \times 2048$   $\mu$ 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.

### 19.1.2.3.3 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)

```

---

<sup>2</sup>Artificial Neural Network – künstliche neuronale Netzwerke.

```
{  
    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-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(); //Stopp  
    return TWI_OK;  
}
```

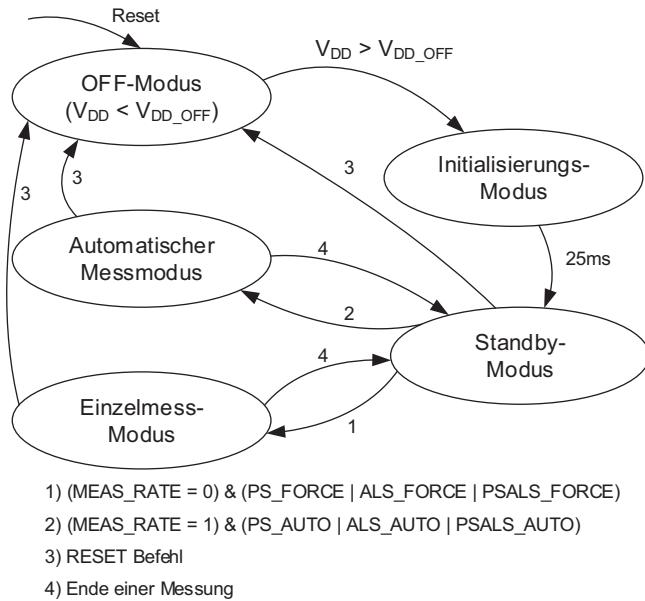
---

## 19.2 SI114x – optischer Näherungssensor

Die Bausteine SI1141, SI1142 und SI1143 besitzen einen Fotosensor für sichtbares und zwei für infrarotes Licht ([3–5]). 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.

### 19.2.1 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



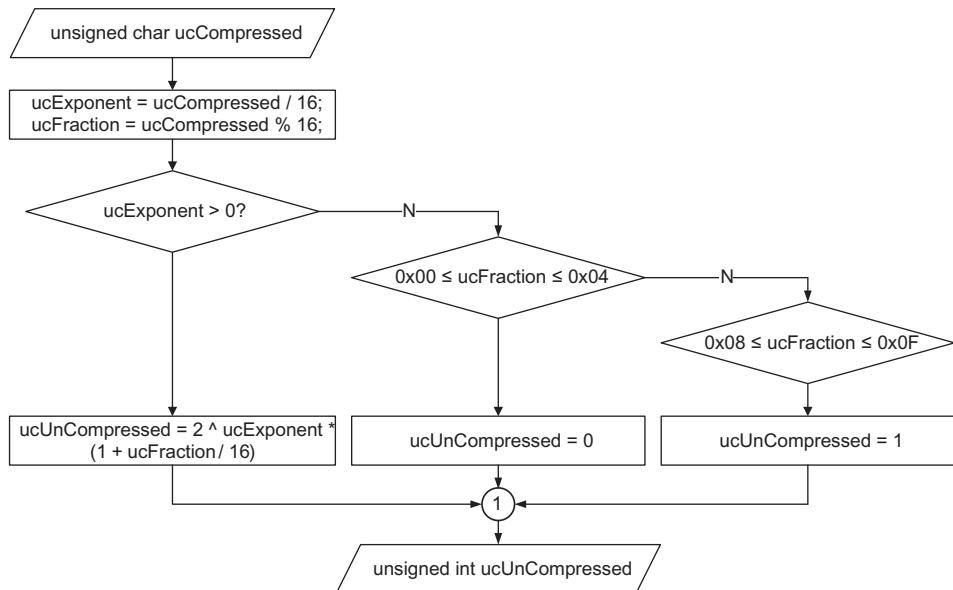
**Abb. 19.4** SI114x – Zustandsübergangsdiagramm

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. 19.4 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 gespeichert 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 komprimierter<sup>3</sup> Wert in das 8-Bit-Register *MEAS\_RATE* (Adresse

<sup>3</sup>Die Kompression wird später erläutert.



**Abb. 19.5** SI114x – Berechnung der unkomprimierten Werte

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 mit einem der Befehle: `PS_PAUSE` (0x09), `ALS_PAUSE` (0x0A) oder `PSALS_PAUSE` (0x0B) gestoppt werden, ohne den automatischen Messmodus zu verlassen.

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  $\mu$ s (1/32 kHz) gibt die Zeit  $T_{mess}$  zwischen zwei möglichen Messungen an. Die Abtastperiode für die Näherungsmessung  $n * T_{mess}$  und für die Messung des Umgebungslichtes  $m * T_{mess}$  sind ein Vielfaches der Grundabtastperiode  $T_{mess}$  wobei  $n$  und  $m$  16-Bit-Ganzzahlen sind. Wenn  $n$  oder  $m$  gleich Null ist, wird die Messung der entsprechenden Größe im automatischen Messmodus nicht durchgeführt. Weil die Register `PS_RATE` (0x0A) 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. 19.5 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.

```

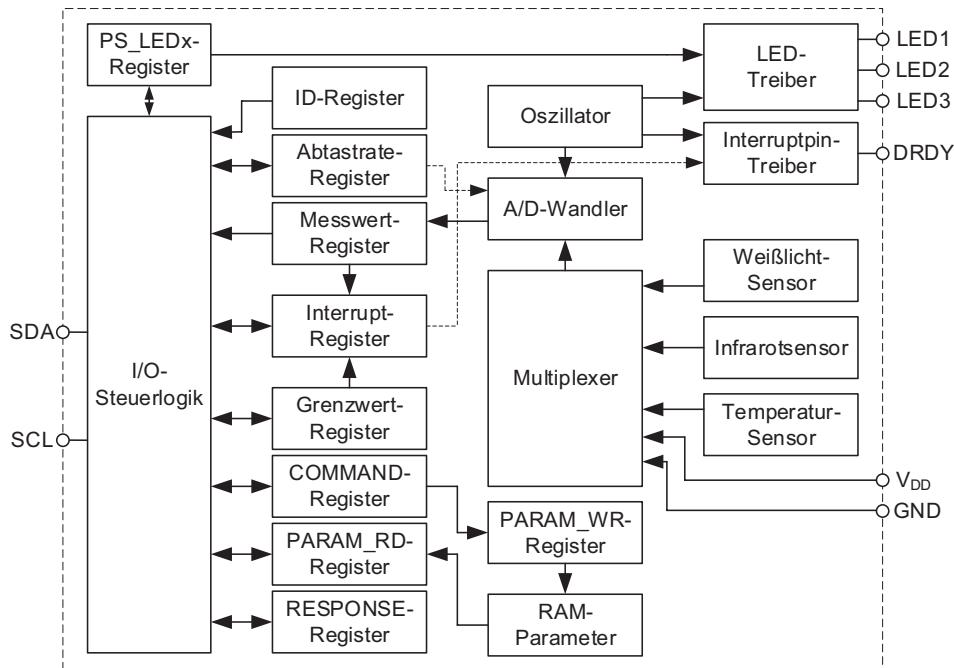
uint8_t SI114x_Set_CompressedValue(unsigned int uiuncompressed)
{
    uint8_t ucExponent = 0xFF, ucFraction, ucCompressed;
    unsigned int uiUnCompressed = uiuncompressed;
    //dem unkomprimierten Wert 0 wird 0x04 und dem 1 wird 0x08
    zugewiesen
    if(!uiuncompressed)      ucCompressed = 0x04;
    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 niederwertige 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.

### 19.2.2 SI114x – Aufbau

Das Blockschaltbild eines Proximity-Sensors vom Typ SI1143 ist in der Abb. 19.6 dargestellt. Der Baustein besitzt einen Sensorblock, der aus einem Weißlichtsensor, zwei



**Abb. 19.6** 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. Das 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 I<sup>2</sup>C-Übertragung (zwischen einer START- und STOPP-Sequenz) wiederholt zu lesen oder zu schreiben ohne die Adressierung des Bausteins und des Registers zu wiederholen.

**Tab. 19.1** 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	

Die 16-Bit-Messergebnisse werden in jeweils zwei Register im Little-Endian-Format gespeichert, wie in der Tab. 19.1 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 [3] 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 intern erfolgreich ausgeführter Befehl wird signalisiert durch das Inkrementieren eines Zählers im Register *RESPONSE* (0x20), bzw. durch das Setzen des Zählers auf 0x01 wenn das *RESPONSE*-Register vorher mit dem Befehl *NOP* (0x00) auf „0“ gesetzt wurde. Die Ausführungszeit der Befehle, die in das *COMMAND*-Register gespeichert wurden, ist unterschiedlich. Die erfolgreiche Ausführung eines Befehls kann durch das

wiederholte Auswerten des *RESPONSE*-Registers festgestellt werden. Dieses Verfahren ist im folgenden Programmcode veranschaulicht. Wenn das Verfahren länger als 25 ms dauert, soll abgebrochen werden und der Befehl wiederholt.

```
#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

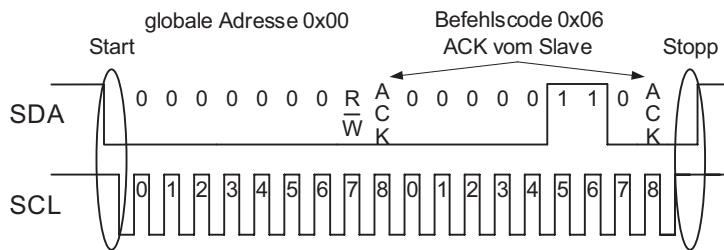
uint8_t SI114x_Write_CommandReg(uint8_t ucdata_byte)
{
    uint8_t ucErrorFlag = 0, ucRepeatCnt = 0;
    ucResponse = 0x01;
    //das Response-Register wird zurückgesetzt
    SI114x_Write_ByteReg(COMMAND_REG, NOP);
    //das Response-Register wird gelesen
    SI114x_Read_ByteReg(RESPONSE_REG, &ucResponse);
    //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;
}
```

### 19.2.3 Serielle Kommunikation

Die Proximity-Sensoren der Reihe SI114x sind als I<sup>2</sup>C-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 I<sup>2</sup>C-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. 19.7 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 I<sup>2</sup>C-Adresse dieser Bausteine kann softwaremäßig geändert werden. Zuerst wird die neue Adresse als *I<sup>2</sup>C-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
#define I2C_ADDR_PARAM         0x00
//Variablen
uint8_t ucSI114xAddress = 0x5A;
//Funktionbeginn
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;
}
```



**Abb. 19.7** I<sup>2</sup>C – globaler Reset-Befehl

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 STOPP-Sequenz.

#### 19.2.4 Messungen 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. 19.1) 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 einem 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 komprimierten Form gespeichert und muss bei der Auswertung der Ergebnisse berücksichtigt werden.

Die Näherungsmessung (Detektion eines Objektes) kann auch für die Gestenerkennung benutzt werden [5]. 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.

#### **19.2.4.1 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.

#### **19.2.4.2 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. Die Strom-

stärke durch die LED1 wird über die Bits 3:0 (*LED1\_I*) vom Register *PS\_LED21* (0x0F) eingestellt und kann mit der Gl. 19.4 berechnet werden. Wenn *LED1\_I* = „0000“ ist, dann wird der Strom durch die Leuchtdiode abgeschaltet.

$$I/mA = \begin{cases} LED1_I \cdot 5,6 & |0 < LED1_I < 3 \\ (LED1_I - 2) \cdot 22,4 & |2 < LED1_I < 13 \\ (LED1_I - 12) \cdot 22,4 & | LED1_I > 12 \end{cases} \quad (19.4)$$

Die Dauer der Lichtpulse  $t_p$  wird über den Parameter *PS\_ADC\_GAIN* (0x0B) eingestellt und ist von der Gl. 19.5 gegeben. Der Einstellbereich des Parameters ist 0...7.

$$t_p = 25,6\mu s \cdot 2^{\text{PS\_ADC\_GAIN}} \quad (19.5)$$

Die Periodendauer des ADC-Taktes wird auch mit dem Faktor  $2^{\text{PS\_ADC\_GAIN}}$  erhöht.

#### 19.2.4.3 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. 19.2 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. 19.6 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.

$$t_R/ns = \begin{cases} 50 \cdot 2^{\text{PS\_ADC\_GAIN}} & |\text{PS\_ADC\_REC} = 0 \\ 50 \cdot 2^{\text{PS\_ADC\_GAIN}} \cdot (2^{2+\text{PS\_ADC\_REC}} - 1) & |\text{PS\_ADC\_REC} > 0 \end{cases} \quad (19.6)$$

**Tab. 19.2** SI1141 – Empfindlichkeitsauswahl und Anpassung an der Umgebungsbeleuchtung

PS_ADC_MISC	
Umgebungsbeleuchtung	
0x04	0x24
Normal	Stark
PS1_ADCMUX	
Empfindlichkeit	
0x00	0x03
Niedrig	Hoch

#### 19.2.4.4 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 (*PS\_ADC\_GAIN*=4) vervielfacht. Daraus erfolgt *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.

### 19.2.4.5 Start der Messung

Nachdem die gewünschten Einstellungen gespeichert wurden, kann eine Näherungsmessung mit dem Befehl *PS\_FORCE* gestartet werden. Die Dauer einer Messung beträgt laut [3] für die Standardeinstellungen 155 µs. Das Ende einer Messung kann über einen Interrupt signalisiert werden.

### 19.2.4.6 Lesen der Messwerte

Das Messergebnis wird in den Registern *PS1\_DATA0* und *PS1\_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(); //Stopp
    return TWI_OK;
}
```

Die Adresse der Variable, in der das Messergebnis gespeichert wird, zusammen mit der Adresse des Registers *PS1\_DATA0* werden als Parameter bei dem Aufruf der Funktion übergeben. Nach dem Auslesen des ersten Registers wird der interne Adresszähler inkrementiert und das zweite Register kann im gleichen I<sup>2</sup>C-Vorgang gelesen werden.

```
unsigned int uiPSValue;
SI114x_Read_WordReg(PS1_DATA0_REG, &uiPSValue);
```

### 19.2.5 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;
```

## 19.2.6 SI114x – Netzwerkidentifikation

Die Bausteine der Serie SI114x 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, sodass 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 komprimierter Form im Register *PS1\_TH0* gespeichert. Weitere Unterschiede sind dem Datenblatt [3] zu entnehmen. Im Register *REV\_ID* ist immer 0x00 gespeichert.

---

## Literatur

1. Hering, E., & Schönfelder, G. (Hrsg.) (2012). *Sensoren in Wissenschaft und Technik*. Vieweg +Teubner
2. Tränkler, H.-R., & Reindl, L. M. (Hrsg.) (2014). *SensorTechnik*. Springer Vieweg
3. Silicon Labs. (2015). SI1141/42/43 Proximity/Ambient light sensor IC with I<sup>2</sup>C interface. [www.silabs.com](http://www.silabs.com)
4. Silicon Labs. (2015). AN498 – SI114x Designer's guide. [www.silabs.com](http://www.silabs.com)
5. Silicon Labs. (2015). AN580 – INFRARED GESTURE SENSING. [www.silabs.com](http://www.silabs.com)
6. Meroth, A., & Tolg, B. (2008). *Infotainmentsysteme im Kraftfahrzeug*. Friedr. Vieweg & Sohn
7. Ultraschall-Modul SRF08. (2021). Datenblatt. <http://www.roboter-teile.de/datasheets/srf08.pdf>  
Zugegriffen: 23. Febr. 2021



# Digital-Analog- und Analog-Digital-Wandler

20

## Zusammenfassung

In diesem Kapitel werden DA- und AD-Wandlerbausteine beschrieben.

Bereits im dritten Kapitel wurden die Möglichkeiten eines Mikrocontrollers beschrieben, mit Bordmitteln analoge Signale einzulesen beziehungsweise über eine integrierende Schaltung mit PWM auch zu erzeugen. Diese integrierten Möglichkeiten sind oft begrenzt, beispielsweise bietet der Atmega8x nur einen einzigen AD-Wandler, der allenfalls gemultiplext werden kann und dessen zuverlässige Auflösung nur bei acht Bit liegt (bei zehn Bit nominelle Auflösung). Auf der anderen Seite stößt die Erzeugung von analogen Signalen über PWM schnell an ihre Grenzen da bei einer Auflösung von beispielsweise acht Bit nur ein 256stel der Taktfrequenz zur Verfügung steht. Hier helfen externe Wandler, die man über SPI oder I<sup>2</sup>C ansteuern kann. Die Funktionsweise der Bausteine wird im jeweiligen Abschnitt erläutert.

## 20.1 MCP48XX SPI-angesteuerte Digital-Analog-Wandler

Die Bausteine der Reihe MCP48XX [3, 4] sind 1- oder 2-Kanal, serielle, unipolare D/A-Wandler mit einer Bitauflösung von 8, 10 oder 12 Bits (siehe Tab. 20.1). Sie zeichnen sich durch eine einfache Außenbeschaltung aus, dank einer einzigen Versorgungsspannung

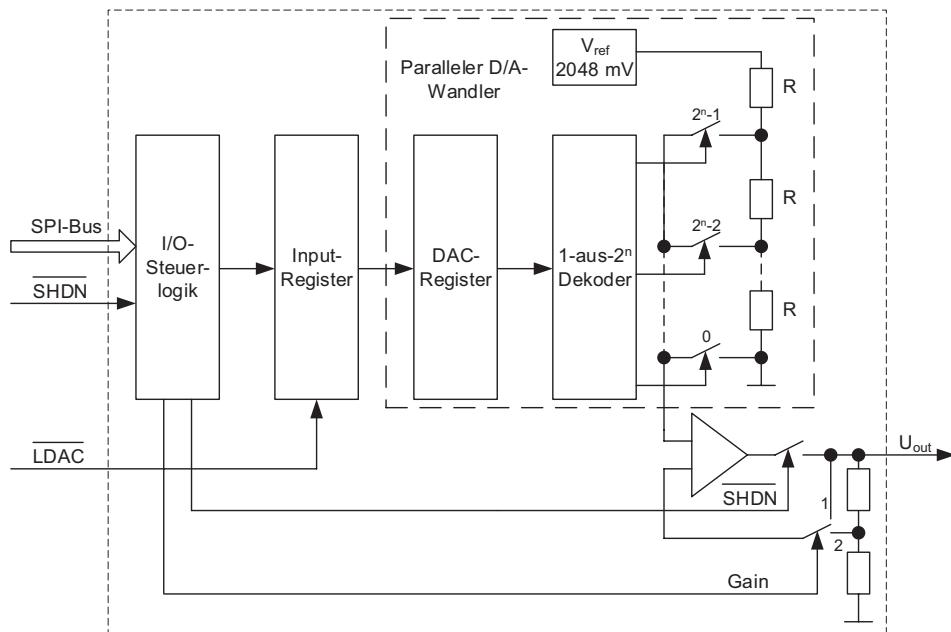
---

Die Originalversion dieses Kapitels wurde revidiert. Ein Erratum ist verfügbar unter  
[https://doi.org/10.1007/978-3-658-31709-6\\_27](https://doi.org/10.1007/978-3-658-31709-6_27)

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann  
[https://doi.org/10.1007/978-3-658-31709-6\\_20](https://doi.org/10.1007/978-3-658-31709-6_20).

**Tab. 20.1** 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. 20.1** MCP48x1 – Blockschaltbild

$V_{DD}$  zwischen 2,7 V 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  $\mu$ s.

Die Bausteine beinhalten wie in Abb. 20.1 dargestellt ein doppeltes Datenregister, ein D/A-Wandler und einen analogen Verstärker.

### 20.1.1 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 zwei 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 Schaltung, 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;
```

### 20.1.2 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 zwei 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);

### 20.1.3 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:

$$U_{LSB} = \frac{U_{REF}}{2^n} \quad (20.1)$$

Mit der Gl. 20.1 lässt sich die Spannung vor dem  $i$ -ten Widerstand berechnen:

$$U_i = (2^i - 1) \cdot U_{LSB}. \quad (20.2)$$

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.

### 20.1.4 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  $0\text{ V} \dots 2*(U_{REF} - U_{LSB})\text{ V}$  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.

### 20.1.5 Synchrone Ansteuerung zweier D/A-Wandler

Wenn man ein PAM-Signal<sup>1</sup> 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. 20.2b ist in Abb. 20.3 dargestellt. Die 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.

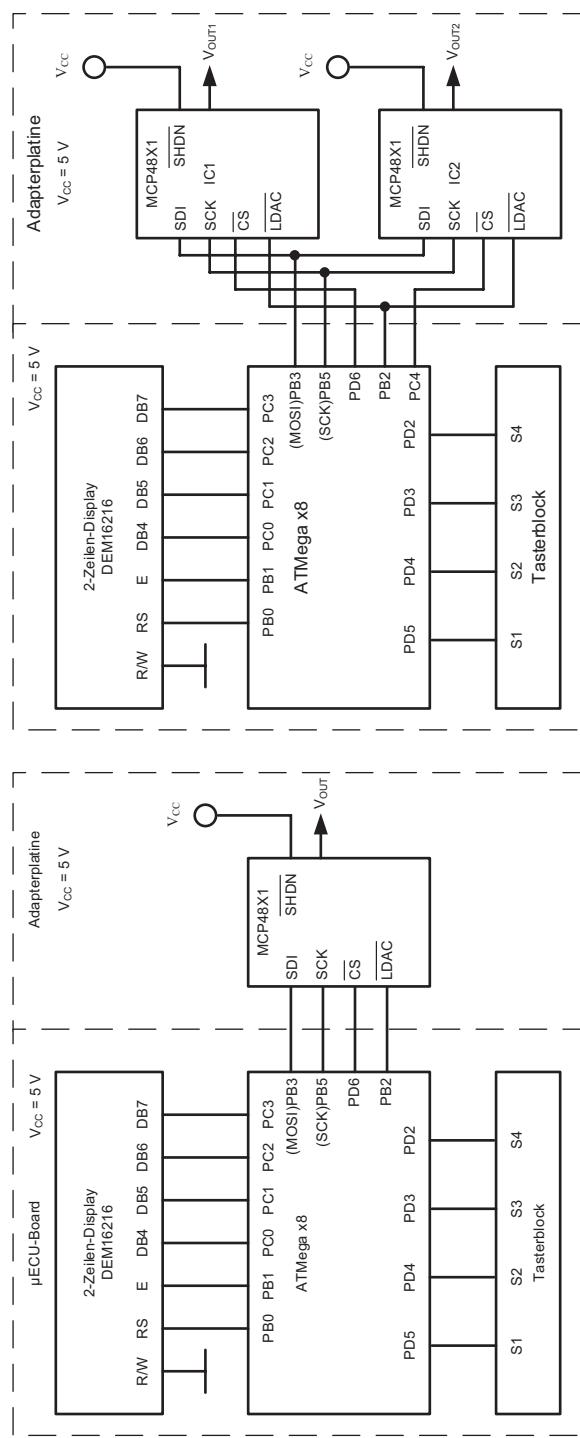
Ein Beispielcode für die in Abb. 20.3 dargestellte Ansteuerung für die Beschaltung aus Abb. 20.2 ist im Folgenden angegeben. Mit:

```
#define ON      1
#define OFF     0
```

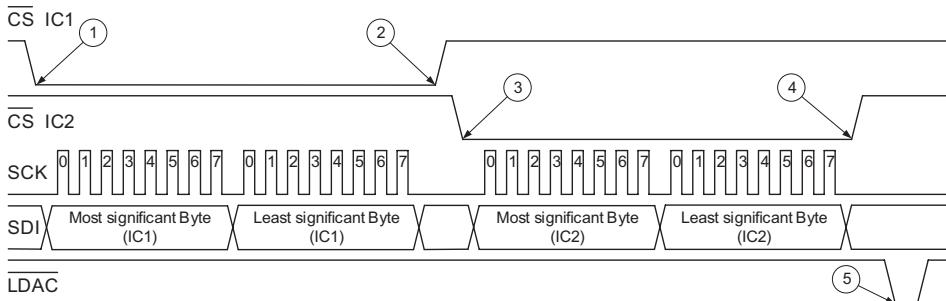
werden die SPI-Datenstrukturen der zwei Bausteine in der Hauptdatei initialisiert:

---

<sup>1</sup>PAM-Signal – Puls Amplitude Moduliertes Signal.



**Abb. 20.2** Anschluss eines SPI-angesteuerten D/A-Wandlers aus der Reihe MCP48x1 an einem Mikrocontroller (a) und Vernetzung zweier D/A-Wandler (b)



- 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.
- Begin der D/A-Wandlung die nach der Einschwingzeit beendet ist.

**Abb. 20.3** Zeitbereichsdarstellung der SPI-Übertragung für die Erzeugung eines analogen Wertepaares mit zwei A/D-Wandlern der Reihe MCP48x1

```
//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 niedlerwertige 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) ;
```

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 aus-
gegeben)*/
MCP48XX_Set_LDACLow(MCP48XX_1) ;
MCP48XX_Set_LDACHigh(MCP48XX_1) ;
```

### 20.1.6 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. 20.1) 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. 20.1). 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. 20.3 bestimmt:

$$U_{OUT} = DAC_n * U_{LSB} * Gain \quad (20.3)$$

wobei  $U_{LSB}$  die Schrittweite angibt und Gain der Verstärkungsfaktor ist. Löst man Gl. 20.3 nach  $DAC_n$  auf, so erhält man:

$$DAC_n = \frac{V_{out}}{U_{LSB} * Gain} \quad (20.4)$$

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 aufgrund 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;
    }

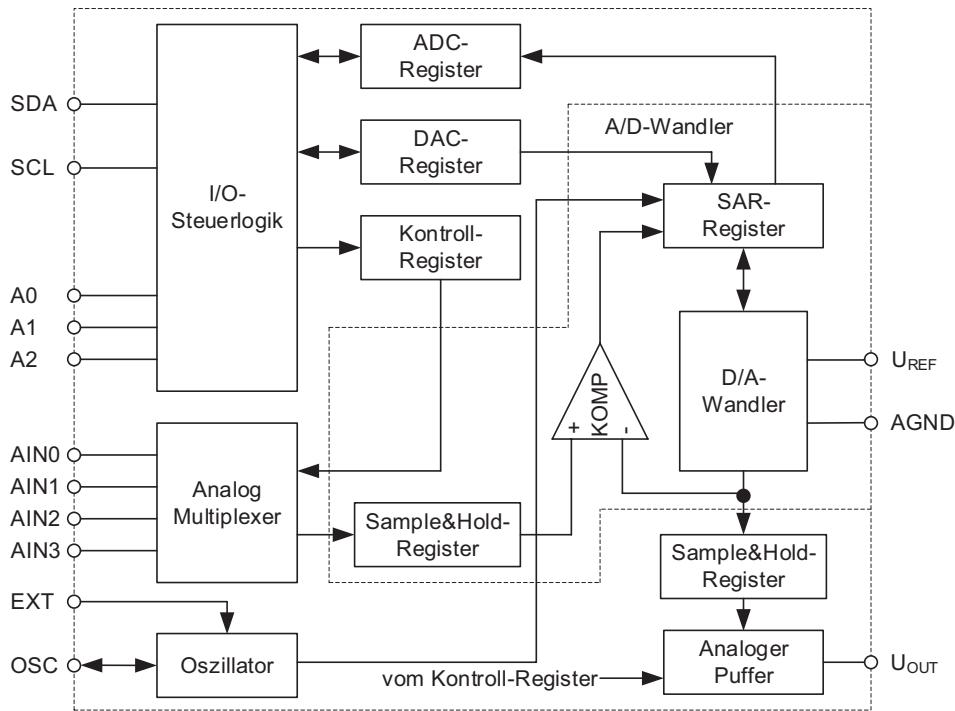
    uiInputRegister |= uiCommandBits; //Berechnung des
    Eingangsregisters
    //das niederwertigste Byte des Eingangsregisters wird
    ermittelt
    ucLSByte = uiInputRegister;
    //das höchstwertige Byte des Eingangsregisters wird ermittelt
    ucMSByte = uiInputRegister >> 8;
    //die Slave-Select-Leitung 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);
}

```

---

## 20.2 PCF8591 I<sup>2</sup>C-angesteuerter D/A- und A/D-Wandler

Der Baustein PCF8591 [5] 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. 20.4 dargestellt. Die Umwandlungsrate ist für beide Richtungen gleich und ist



**Abb. 20.4** PCF8591 – Blockschaltbild

von der Busfrequenz bestimmt. Der Baustein hat einen analogen Ausgang und vier analogen Eingänge und wird über einen I<sup>2</sup>C-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.

### 20.2.1 I<sup>2</sup>C-Kommunikation

Die Kommunikation mit einem Mikrocontroller, der als Master konfiguriert ist, erfolgt bei einer Übertragungsr率e von max. 100 kBit/s. Die Übertragungsr率e spielt eine aktive Rolle f率r die Umwandlungsvorgnge, weil mit der bertragung neue Umwandlungen gestartet werden. Der Master startet die Kommunikation mit der bertragung 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 nchste Byte in das Control-Register gespeichert, um die folgende Umwandlung vorzubereiten. Die nchsten 1 bis n Bytes werden in das DAC-Register gespeichert und am Ende eines Bytes (nach neun Taktn) als analoger Wert ausgegeben.

Dieser Burst-Modus ermöglicht die höchste Umwandlungsrate: 100 kBit/s / 9 Bit/Sample = 11,1 kS<sup>2</sup>/s.

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 acht 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.

### 20.2.2 Der D/A-Wandler

Der 8-Bit-D/A-Wandler des Bausteins funktioniert nach demselben Parallelverfahren, wie es in Abschn. 20.1.1.3 beschrieben wurde. Der interne Spannungsteiler, der für die 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  $U_{LSB}$

$$U_{LSB} = \frac{U_{REF} - U_{AGND}}{256} \quad (20.5)$$

und die Ausgangsspannung

$$U_{OUT} = U_{AGND} + U_{LSB} \cdot OutputByte \quad (20.6)$$

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

---

<sup>2</sup>kS/s – kilo Samples / Second.

100 %, bei einer Belastung mit 10 kΩ nur noch 90 % der Versorgungsspannung erreichen. Mit folgenden Definitionen:

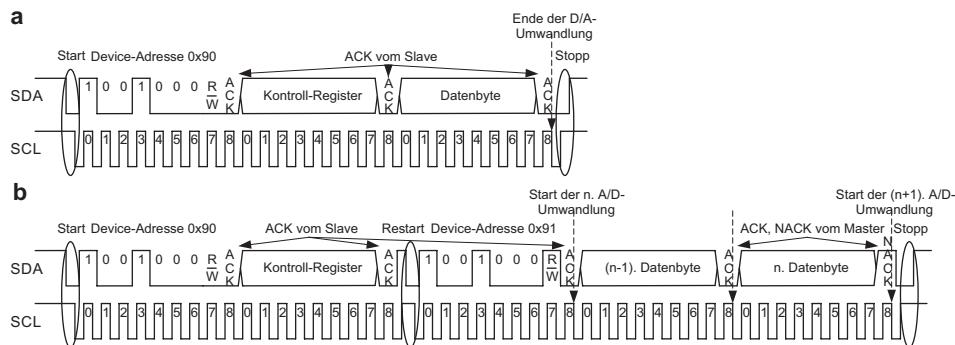
```
#define PCF8591_DEVICE_TYPE_ADDRESS      0x90
#define REFERENCE_VOLTAGE                5000 //mV
#define ANALOG_GROUND_VOLTAGE           0 //mV
#define OUTPUT_STEP_SIZE                 ((REFERENCE_VOLTAGE - ANALOG_
                                         GROUND_VOLTAGE) / 256)
#define D_A_CONVERSION_OPCODE           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 acht 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. 20.5a zu sehen. Wenn man einen analogen Wert erzeugen will und die Referenz- und Offsetspannung der analogen Masse bekannt sind, dann kann Gl. 20.6 nach dem binären Wert aufgelöst werden und man erhält:

$$OutputByte = \frac{U_{OUT} - U_{AGND}}{U_{LSB}}. \quad (20.7)$$



**Abb. 20.5** PCF8591 – I<sup>2</sup>C-Kommunikation für die Ansteuerung des D/A- und A/D-Wandlers

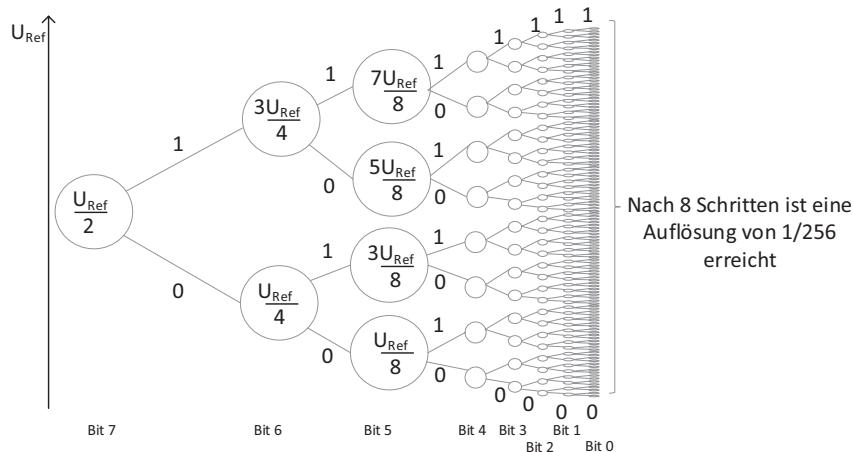
Um eine einzige D/A-Wandlung zu realisieren, startet der Master die Kommunikation und sendet danach die Adresse des Slaves mit dem R/W-Bit auf „0“ (Schreib-Befehl), ein weiteres Byte, das in das Control-Register gespeichert wird und schließlich das umzuwandelnde Byte. Mit einem Stopp-Befehl beendet der 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.

### 20.2.3 Der A/D-Wandler

Die vier analogen Eingänge können wie in Abb. 20.7 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  $-(U_{REF} - U_{AGND})/2$  und die obere bei  $+(U_{REF} - U_{AGND})/2$  und das Ergebnis wird mit Vorzeichen gespeichert.

Der A/D-Wandler funktioniert nach dem Prinzip der sukzessiven Annäherung (successive approximation)<sup>3</sup> und besteht aus dem oben beschriebenen D/A-Wandler, der

<sup>3</sup> Siehe auch Kap. 3 Abschn. 3.8.3



**Abb. 20.6** Prinzip der sukzessiven Approximation (successive approximation)

vorübergehend für die A/D-Wandlung eingesetzt wird, einem analogen Komparator und einem SAR<sup>4</sup>-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  $V_{\text{REF}}/2$  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  $\text{Bit } 7 \cdot U_{\text{REF}}/2 + U_{\text{REF}}/4$ . 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 achten Schritt ist die Annäherung kleiner als  $U_{\text{LSB}}$  und im SAR-Register steht der binäre Wert der im ADC-Register gespeichert wird. Abb. 20.6 verdeutlicht diesen Sachverhalt (vgl. auch Absch. 3.8.3).

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. 20.5 b) dargestellt.

<sup>4</sup>SAR – Successive Approximation Register.

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.

```
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;
```

## 20.2.4 Das Control-Register

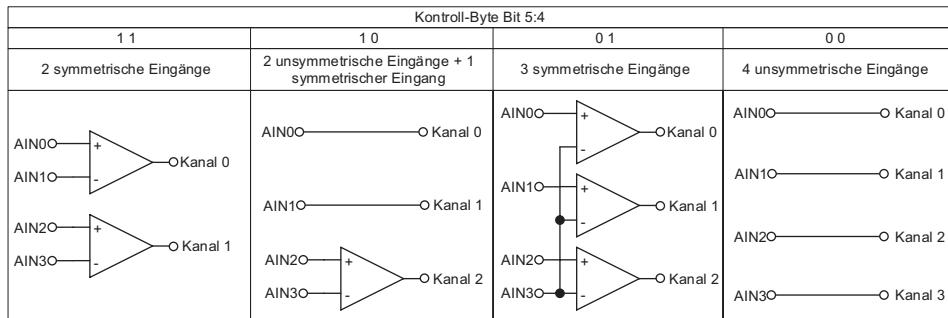
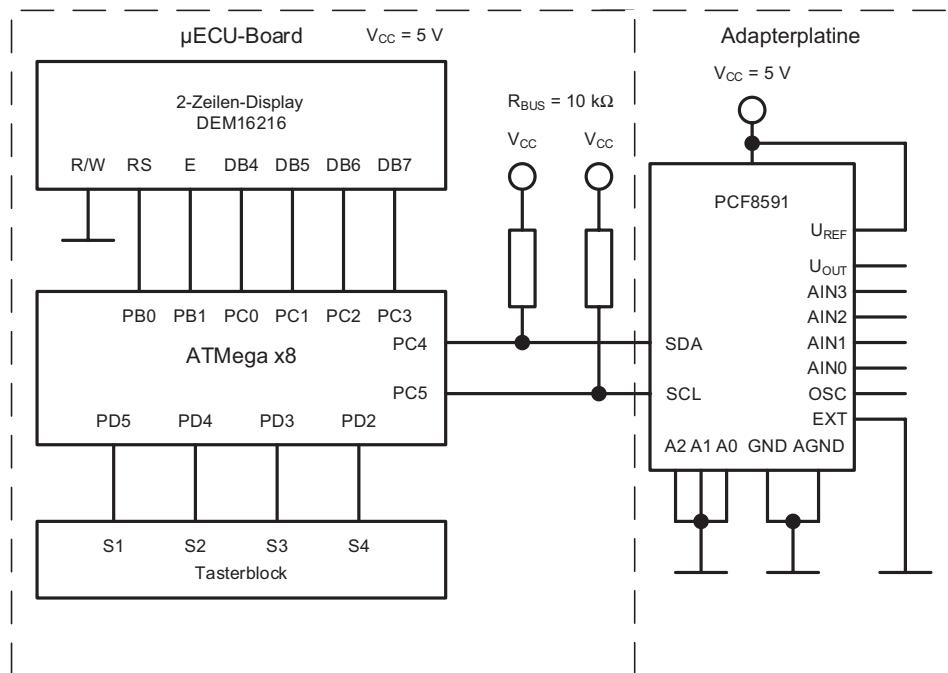
Das Control-Register steuert die Beschaltung der analogen Eingänge, den analogen Multiplexer und den analogen Puffer und hat folgende Konfiguration:

- **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. 20.7)
- **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.)

## 20.2.5 Der Oszillator

Der Baustein benötigt für die A/D-Umwandlung einen Takt. Wenn der Pin EXT mit Masse verbunden ist, wird dieser Takt vom internen Oszillator generiert und ist am Pin OSC von außen zugänglich. Wenn Pin EXT an  $V_{CC}$  angeschlossen ist, dann muss ein externer Oszillator am Pin OSC den benötigten Takt bereitstellen.

In der Beispielschaltung Abb. 20.8 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.

**Abb. 20.7** PCF8591 – Beschaltung analoger Eingänge**Abb. 20.8** Anschluss eines PCF8591 an einem Mikrocontroller

### 20.3 Strommessung mit dem LMP92064

LMP92064 ist ein über SPI vernetzbarer Baustein, der für die gleichzeitige Messung eines Gleichstroms und einer Gleichspannung entwickelt wurde [1]. 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.

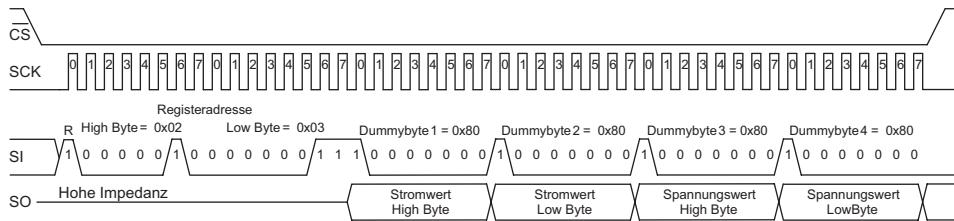
### 20.3.1 LMP92064 Aufbau

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 niedrigerwertige 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. 20.2 aufgelistet sind.

Eine interne Referenzspannung von 2,048 V für beide Messkanäle vereinfacht die Beschaltung des Bausteins. Zwei Operationsverstärker mit einer Eingangsimpedanz von 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

**Tab. 20.2** 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



**Abb. 20.9** LMP92064 – Auslesen der Datenregister

wird mit einem Spannungsteiler bis zu  $U_{max}$  erweitert (siehe Abb. 20.9) was auch zur Erhöhung der Schrittweite um den Faktor  $(R_1+R_2)/R_2$  führt:

$$U_{max} = U_{ref} \cdot \frac{R_1 + R_2}{R_2} \quad (20.8)$$

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. 20.3 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.

### 20.3.2 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

**Tab. 20.3** LMP92064 – Messdatenregister

Datenregister		Adresse
Spannung	Low Byte	0x0200
	High Byte	0x0201
Strom	Low Byte	0x0202
	High Byte	0x0203

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. 20.10 wird in der Hauptdatei folgende SPI-Datenstruktur deklariert (Kap. 14).

```
LMP92064_pins LMP92064_1 = {{ /*CS_DDR*/      &DDRB,
                                /*CS_PORT*/     &PORTB,
                                /*CS_pin*/      PB0,
                                /*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. 20.9 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.

### 20.3.3 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. 20.10 zeigt beispielhaft den Anschluss eines LMP92064 an einem Mikrocontroller, sowie die Mess- und Stromkalibrierschaltung.

### 20.3.3.1 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:

$$I = \frac{U_{max}}{R_1 + R_2} \leq I_D \quad (20.9)$$

- Aus Gl. 20.8 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 besser 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  $\mu$ 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);
    //0,5mV = 1/2 Auflösung des A/D-Wandlers
    ulVoltage = (ulVoltage * (lr1 + lr2)) / 2;
    ulVoltage = ulVoltage / lr2;      //lr1 + lr2 Gesamtwiderstand
    return ulVoltage;
}
```

Wenn das Verhältnis  $R_1 / R_2$  so gewählt wird, dass:

$$\frac{R_1}{R_2} = \begin{cases} (2 \cdot n) - 1 & \text{oder} \\ 2^n - 1 & \text{mit } n \in N \end{cases} \quad (20.10)$$

dann kann der Spannungswert in mV folgendermaßen berechnet werden:

```
ulVoltage = ulVoltage * n; //für  $R_1 / R_2 = 2^{n-1}$ 
ulVoltage = ulVoltage << 2^{n-1}; //für  $R_1 / R_2 = 2^{n-1}$ 
```

was zu einer Rechenzeit unter 2,5  $\mu$ s führt.

### 20.3.3.2 Strommessung

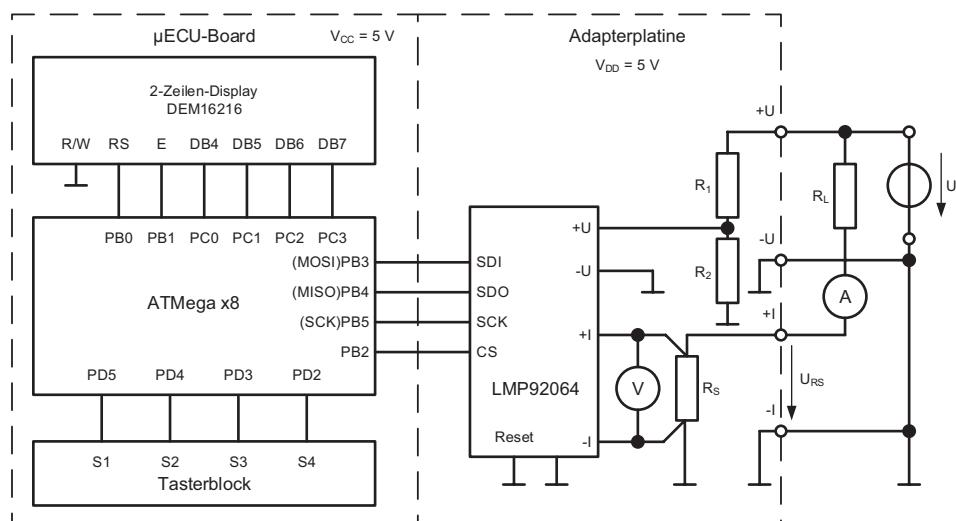
Der zu messende Strom verursacht am Strommesswiderstand einen Spannungsabfall, der verstärkt und digitalisiert als Grundlage für die Berechnung der Stromstärke dient. Um den Einfluss der Kontaktwiderstände  $R_{K1}$  und  $R_{K2}$  auf die Strommessung zu reduzieren wird das Vierleiter-Messverfahren verwendet wie in Abb. 20.11a. 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. 20.11b dargestellt.

Eine präzise Messung wird nur gewährleistet, wenn die gesamte Messkette kalibriert ist. In der Schaltung aus Abb. 20.10 wurde ein 150 mΩ großer SMD-Shunt verbaut. Entsprechend den Stromstärken aus der Tab. 20.4 sind die Spannungsabfälle am Shunt gemessen. Abgesehen von den Messgenauigkeiten ergibt sich aus dieser Messreihe ein Wert von ca. 160 mΩ, mit dem weiterhin gerechnet wird.

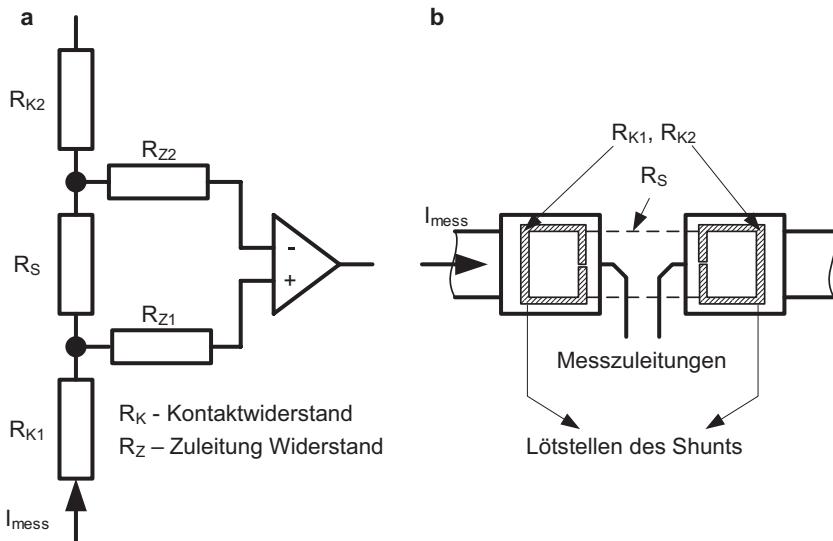
Entsprechend einem eingestellten Strom von 200 mA wird aus dem Datenregister des Bausteins der digitale Wert 1504 gelesen. Mit diesem Wert errechnet sich eine Stromschrittweite von ca. 0,133 mA; der maximale Strom, der mit diesem Shunt gemessen werden kann, ist:

$$I_{\text{mess max}} = 0,133 \text{ mA} \cdot 3840 \approx 510 \text{ mA}.$$

Ein Stromwert wird aus dem digitalen Wert  $ulCurrent$  mit der Anweisung:



**Abb. 20.10** LMP92064 – Beschaltung



**Abb. 20.11** Vierleiter-Messverfahren

**Tab. 20.4** 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

**Tab. 20.5** 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

```
ulCurrent = (ulCurrent * 133) / 1000; //die Stromschrittweite beträgt
0,133 mA
```

in ca. 35  $\mu$ s berechnet. Zum Vergleich stehen in der Tab. 20.5 eine Reihe von eingestellten Stromwerten und die Mittelwerte von über 100 Messungen gegenüber.

## Literatur

1. Texas Instruments. (2014). LMP92064 Precision Low-Side, 125-kSps Simultaneous Sampling, Current Sensor and Voltage Monitor with SPI. [www.ti.com](http://www.ti.com). Zugegriffen: 3. Apr. 2021.
2. Rainer Parthier. (2008). *Messtechnik*. Friedr. Vieweg & Sohn.

3. Microchip Technology Inc. (2015). MCP4801/4811/4821 1 Kanal SPI-D/A-Wandler. [www.microchip.com](http://www.microchip.com). Zugegriffen: 4. Apr. 2021.
4. Microchip Technology Inc. (2015). MCP4802/4812/4822 2 Kanal SPI-D/A-Wandler. [www.microchip.com](http://www.microchip.com). Zugegriffen: 4. Apr. 2021.
5. NXP Semiconductors. (2013). PCF8591 – 8-Bit A/D and D/A converter. [www.nxp.com](http://www.nxp.com). Zugegriffen: 3. Apr. 2021. <https://www.nxp.com/docs/en/data-sheet/PCF8591.pdf>.



# Serielle EEPROMs

21

## Zusammenfassung

Kapitel 21 widmet sich den seriellen EEPROMs, die mit I<sup>2</sup>C oder SPI angesteuert werden.

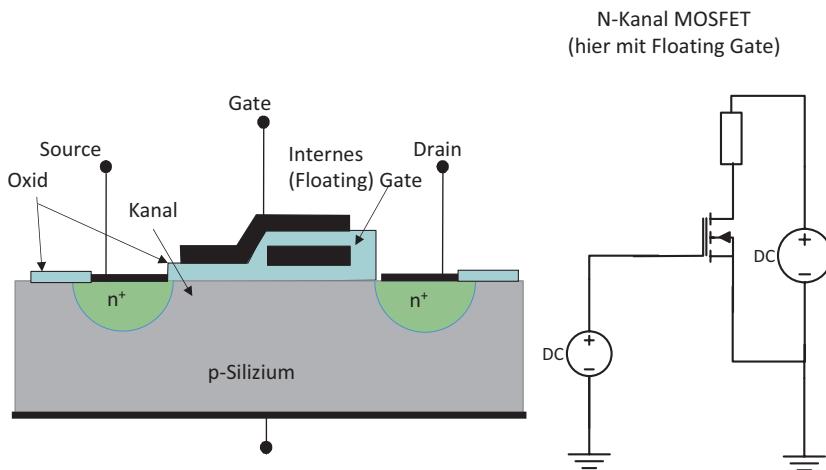
Festwertspeicher sind nichtflüchtige Speicher, in die Daten zur Laufzeit geschrieben werden, die nach Abschalten oder Reset des Systems persistent gespeichert bleiben. Man unterscheidet auch hier (Kap. 3) zwischen EEPROMs und Flashspeichern. Insbesondere, wenn man größere Datenmengen zur Laufzeit speichern will, beispielsweise in offline-Datenloggern, werden Festwertspeicher als Erweiterung des begrenzten Speicherplatzes eines Mikrocontrollers benötigt. In diesem Kapitel werden zunächst typische EEPROM-Speicher beschrieben, die mit I<sup>2</sup>C und SPI angesteuert werden.

EEPROMs und Flashspeicher arbeiten in der Regel mit Floating-Gate-Feldeffekttransistoren (oder Floating-Gate-FET), wie in Abb. 21.1 angedeutet [1]. Ein Feldeffekttransistor ist ein spannungsgesteuertes Schaltelement. Bei einem n-Kanal-MOSFET<sup>1</sup>,

<sup>1</sup> MOS steht für die Schichtfolge Metall-Oxid-Semiconductor (Halbleiter).

Die Originalversion dieses Kapitels wurde revidiert. Ein Erratum ist verfügbar unter  
[https://doi.org/10.1007/978-3-658-31709-6\\_27](https://doi.org/10.1007/978-3-658-31709-6_27)

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann  
[https://doi.org/10.1007/978-3-658-31709-6\\_21](https://doi.org/10.1007/978-3-658-31709-6_21).



**Abb. 21.1** MOSFET mit Floating Gate

wie er in der Abbildung skizziert ist, kontaktieren zwei Elektroden (Drain und Source) zwei stark n-dotierte Zonen, das sind Bereiche, in denen das Siliziumgitter (vier AußenElektronen) mit Fremdatomen verunreinigt sind, die fünf AußenElektronen besitzen. Hier herrscht also ein Ladungsträgerüberschuss. Das Substrat des Halbleiters selbst ist schwach p-dotiert, besitzt also Störstellen mit drei AußenElektronen, somit einen Elektronenmangel, der üblicherweise *Löcherüberschuss* genannt wird. An der Grenze zwischen den Gebieten entsteht ein pn-Übergang, der durch Rekombination der Elektronen mit den Löchern zu einer Sperrsicht führt. Zwischen Drain und Source kann daher kein Strom fließen. Dieser Typ Feldeffekttransistor wird selbstsperrend oder Anreicherungstyp genannt (enhancement type, normal sperrend).

Legt man nun an eine weitere, durch eine SiO<sub>2</sub>-Schicht isolierte Elektrode (Gate) eine positive Spannung gegenüber dem Substrat (Bulk) an, das wiederum mit dem Source-Anschluss verbunden werden kann, werden aus der Spannungsquelle Elektronen in das Substrat gezogen, die zunächst mit den Löchern im Substrat rekombinieren, bei höherer Spannung<sup>2</sup> ( $>U_{th}$ ) aber einen leitfähigen, mit Elektronen angereicherten Kanal bilden, der sich zwischen Drain und Source ausbildet. Ab diesem Moment kann ein Strom fließen. Da die Gate-Bulk-Strecke selbst eine Kapazität darstellt, fließt kein Strom in das Gate, der Transistor ist also spannungsgesteuert.

Ein Floating-Gate-FET besitzt eine weitere Elektrode zwischen Gate und Substrat, die hermetisch isoliert in die Oxidschicht eingebettet ist. Durch den Tunneleffekt (**Fowler-Nordheim-Tunnel**) gelingt es ab einer gewissen Spannung einigen Elektronen, die Barriere des Isolators zu überwinden so dass sich das isolierte Gate negativ auflädt. Durch die Isolierung können sie nicht mehr entweichen und der FET bleibt trotz des

<sup>2</sup> U<sub>th</sub> steht für Threshold-Spannung, zu Deutsch Schwellenspannung, typischerweise 1,5 V bis 3,3 V.

positiven Gates nunmehr dauerhaft gesperrt. Erst durch das Entfernen der Elektronen mit Hilfe von UV-Licht (EPROM) oder einer negativen Spannung (EEPROM) wird das Floating-Gate wieder entladen und der Transistor schaltet wieder normal.

Mit anderen Worten:

- Die Gate-Elektrode dient als Leseelektrode: Liegt eine Spannung  $> U_{th}$  an und ist das Floating-Gate ungeladen, so leitet der Transistor und an Drain liegt Massepotenzial an ( $0 \text{ V} = \text{logisch } „0“$ ). Ist das Floating-Gate aber geladen, so sperrt der FET bei  $U_{th}$  und es liegt weiterhin Batteriespannung ( $=\text{logisch } „1“$ ) an. Da EEPROMs mit einem invertierenden Ausgangstreiber ausgestattet sind, liegen die Signale invertiert vor, d. h. Löschen bedeutet „1“ am Ausgang und Schreiben „0“.
- Die Gate-Elektrode dient als Schreibeletektrode: Erhöht man die Spannung deutlich über die Schwellenspannung hinaus (typischerweise  $10 \text{ V}$ ) wird das Floating-Gate geladen und die Speicherzelle damit auf „1“. Um es wieder zu löschen, wird das Gate auf Masse gelegt und der Drain-Anschluss kurzfristig auf eine hohe Spannung (wiederum  $> 10 \text{ V}$ ) gelegt. Dann können die Elektronen wieder aus dem Floating-Gate über Drain zurücktunnellen und das Floating-Gate ist wieder neutral. Technisch werden EEPROM-Zellen mit zwei Transistoren ausgeführt [1], einer davon schaltet die Schreibspannung an den Speichertransistor.

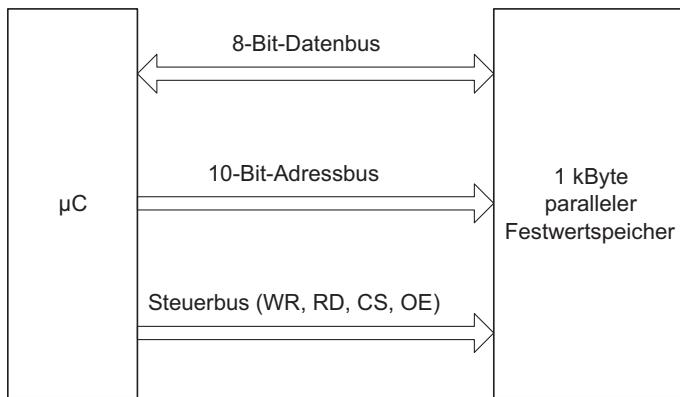
Da das vollständige Tunneln eine gewisse Zeit beansprucht, dauert das Schreiben einer solchen Zelle mit einigen Millisekunden deutlich länger als das Lesen.

Reine EEPROM-Zellen benötigen darüber hinaus relativ viel Platz und Energie. Dafür ist ihre Langzeitstabilität in der Regel hoch. Ihre Beschreibbarkeit wird typischerweise mit  $10^6$  Schreibvorgängen angegeben, die Beständigkeit der Daten mit 10 Jahren. Daher werden EEPROMs in der Regel als Datenspeicher mit geringer Kapazität (beispielsweise für Betriebsstundenzähler, Fehlerspeicher oder zur Parametrierung von Steuergeräten) eingesetzt.

---

## 21.1 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. 21.2). In diesem Beispiel wird innerhalb eines Taktzyklus ein gesamtes Datenbyte (acht 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.



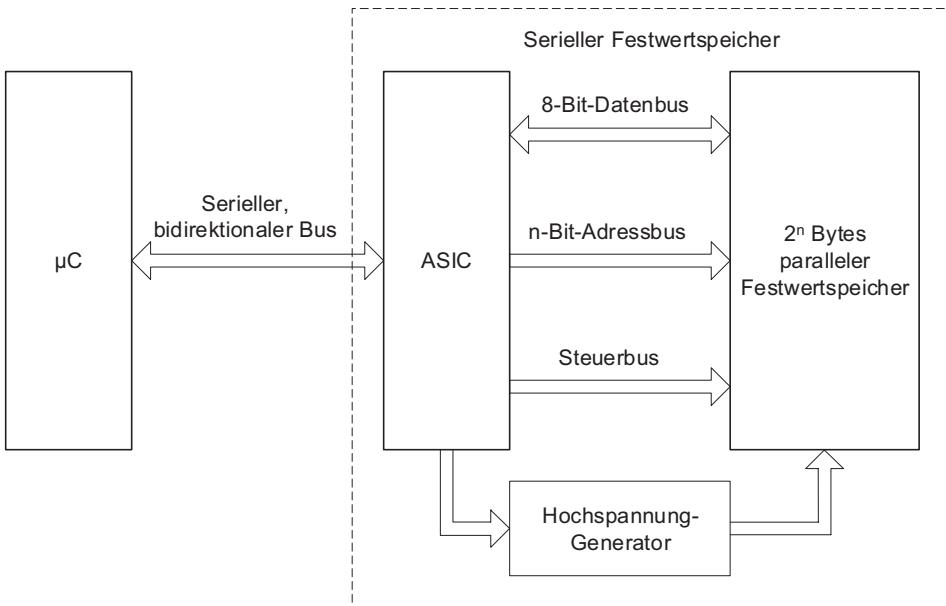
**Abb. 21.2** Externer, paralleler Speicher angeschlossen an einem Mikrocontroller

## 21.2 Serielle EEPROM-Speicher

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. 21.3 zu sehen ist. Die für die Ansteuerung der seriellen Speicher meist verwendeten seriellen Schnittstellen sind: SPI, I<sup>2</sup>C, Microwire und UNI/O (1-Draht-Schnittstelle der Fa. Microchip). Der Daten austausch 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.

### 21.2.1 M24C64 – I<sup>2</sup>C-angesteuerter EEPROM

Als Beispiel für I<sup>2</sup>C-angesteuerte EEPROMs wird im Folgenden die 24xx-Reihe vorgestellt und der Baustein M24C64-R (siehe [2]) näher beschrieben. Diese Speicherreihe wird von den meisten Speicherherstellern mit unterschiedlichen Speicherkapazitäten produziert. Die Bausteine werden über einen I<sup>2</sup>C-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 sogenannten Pages (Speicherseiten) organisiert, die 8, 16 oder 32 Byte groß sein können, abhängig von der

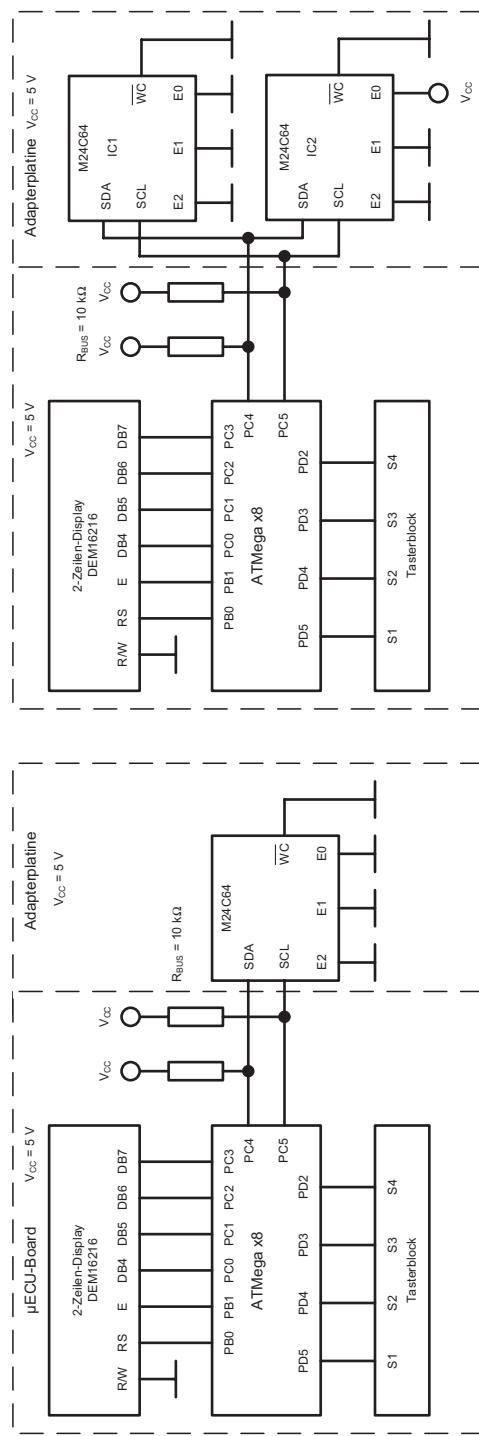


**Abb. 21.3** Externer serieller Speicher angeschlossen an einem Mikrocontroller

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 V 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 I<sup>2</sup>C-Protokoll implementiert. Die Versorgungsspannung kann im Temperaturbereich von -40 °C bis 85 °C zwischen 2,5 V und 5,5 V variieren, bei den -F- und -DF-Ausführungen zwischen 1,7 V und 5,5 V. Die Ausführung M24C64-D besitzt eine zusätzliche Speicherseite (32 Bytes), deren Inhalt dauerhaft schreibgeschützt werden kann.

In Abb. 21.4 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/Programmierzyklen (> 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 I<sup>2</sup>C-Bus-Signale. Der Write-Control-Eingang ist dauerhaft auf Low geschaltet, so dass der Schreibschutz deaktiviert ist.



**Abb. 21.4** Speichererweiterung eines Mikrocontrollers mit seriellen EEPROMs

### 21.2.1.1 I<sup>2</sup>C-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 acht 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. 21.4) 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 I<sup>2</sup>C-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 I<sup>2</sup>C-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 neunten 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 I<sup>2</sup>C-STOPP-Sequenz, gibt den Bus wieder frei und der Slave schaltet in den Stromsparmodus.

### 21.2.1.2 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 Lese-funktionen können auch dann ausgeführt werden, wenn der Write-Control-Eingang auf High geschaltet ist. Folgende Lese-funktionen sind implementiert:

#### 21.2.1.2.1 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 I<sup>2</sup>C-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 acht 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 STOPP-Sequenz.

### 21.2.1.2.2 Sequentialles 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.

### 21.2.1.3 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 STOPP-Sequenz.

Die empfangenen Bytes werden in einen 32 Byte großen, flüchtigen Speicher und von dort, nach der I<sup>2</sup>C-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. 21.5 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(); //Stopp
return TWI_OK;
}

```

Folgender Funktionsaufruf speichert 32 Bytes aus dem Array *ucBuffer[32]* auf der 32. Speicherseite (Byte 1024:1055) von IC1(s. Abb. 21.4):

```
_24C64_Write_Page(0x00, 1024, ucBuffer, 32);
```

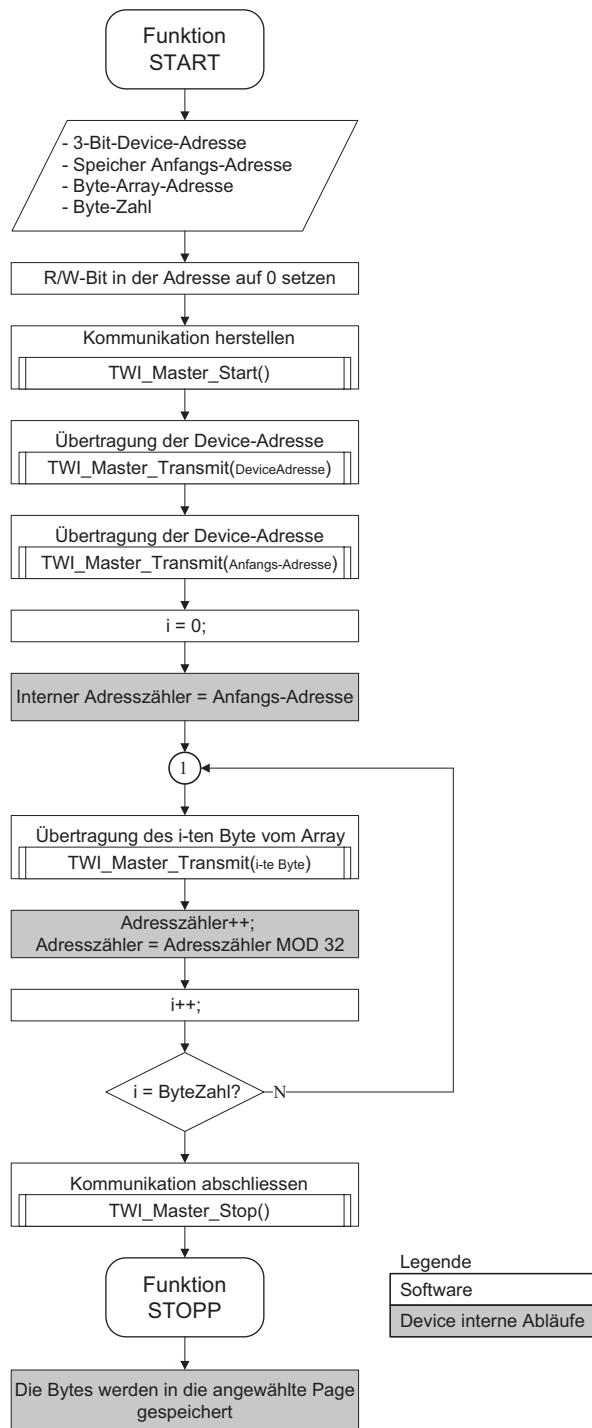
#### 21.2.1.4 Testbarkeit

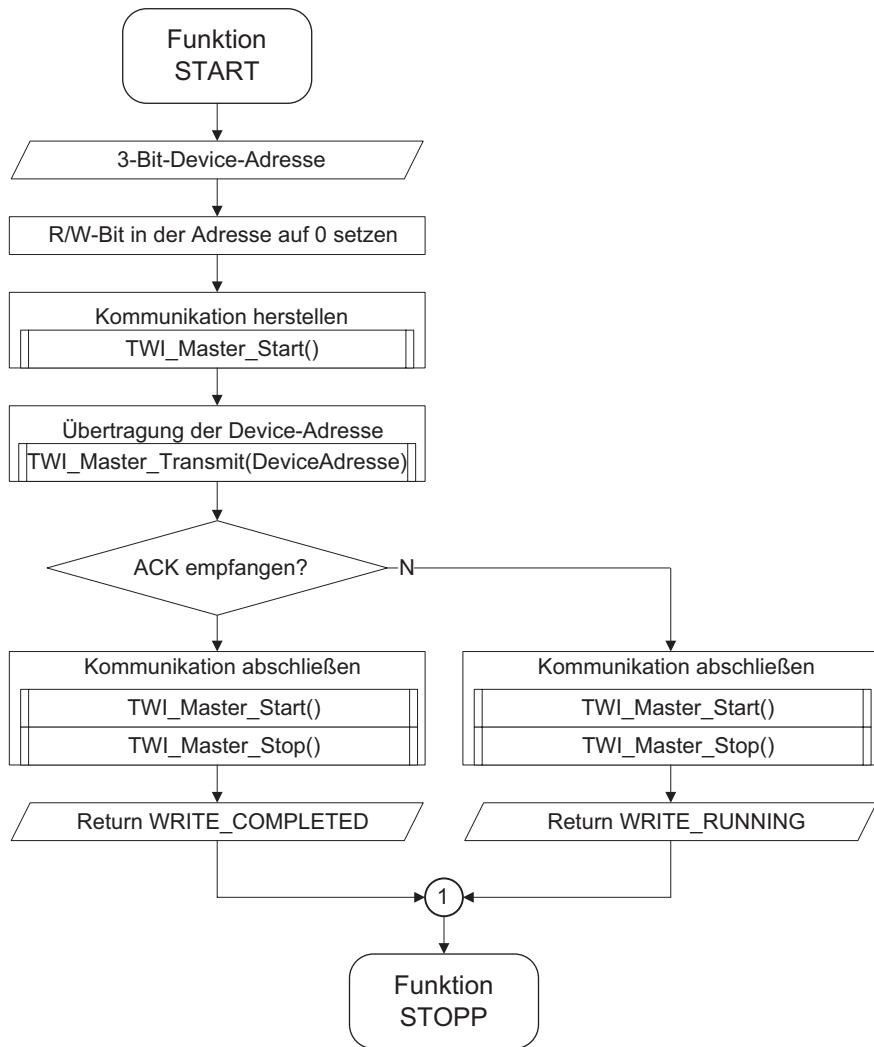
Die elektrische Verbindung zwischen dem Mikrocontroller und dem Speicherbaustein wird schon in der Adressierungsphase getestet. Wenn der Baustein beim Empfang seiner Adresse nicht mit Acknowledge antwortet, dann ist entweder die Adresse falsch, oder die elektrische Verbindung nicht in Ordnung, oder er 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. 21.6 dargestellt und der Code weiter unten aufgelistet.

Um zu prüfen, ob der letzte Schreibvorgang abgeschlossen ist, initiiert der Master die I<sup>2</sup>C-Kommunikation mit einer START-Sequenz und sendet die Adresse des zu testenden Speichers. Wenn die Adresse mit ACK quittiert wird, kann ein weiterer Schreibvorgang

**Abb. 21.5** Flussdiagramm der Funktion PageWrite für den M24C64-Speicherbaustein





**Abb. 21.6** Flussdiagramm – Schreibzustand-Abfrage für den M24C64-Speicherbaustein

gestartet werden, ansonsten gibt der Master mit einer START-STOPP-Sequenz den Bus wieder frei.

```

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)
{
    //der 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. 21.4 rechts 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:

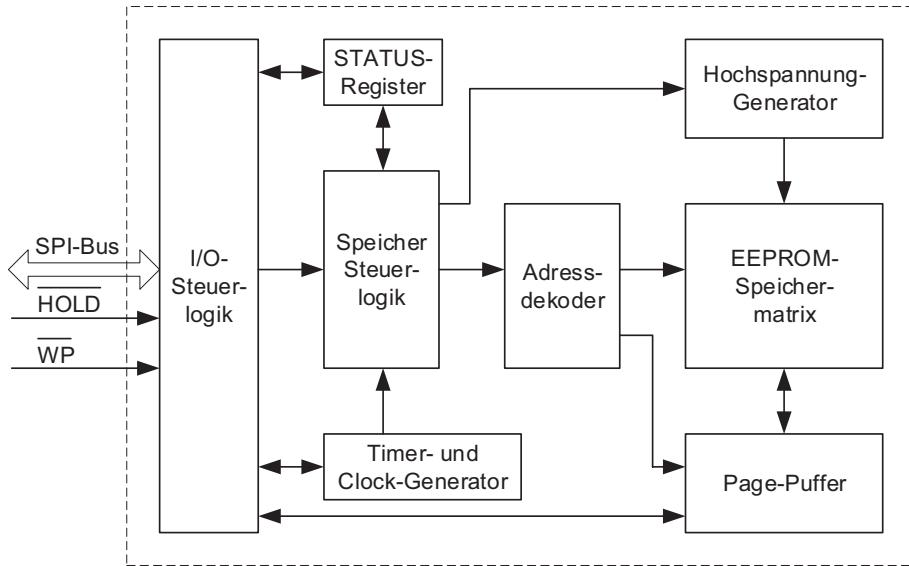
```

if(M24C64_Get_WriteState(0x01) == WRITE_COMPLETED)
{
    _24C64_Write_Byte(0x01, uiAddress+ucIndex, ucBuffer[ucIndex])
    ucIndex++;
}

```

### 21.2.2 25LC256 – SPI-angesteuerte EEPROMs

SPI-angesteuerte EEPROMs werden von den meisten Herstellern unter der Reihe 25xx (oder 95xx [3]) mit Speicherkapazitäten ab 1 kBit (128 Byte) bis 1 Mbit (128 kByte) angeboten. Das Blockschaltbild eines solchen Speichers ist in Abb. 21.7 zu sehen. Als Beispiel wird im Folgenden der Speicherbaustein 25LC256 [4] als Nachfolger des AT25256 [5] vorgestellt. Der Baustein, der eine Speicherkapazität von 265 kBit (32 kByte) besitzt, ist in 64 Byte große Seiten (Pages) organisiert und kann mit



**Abb. 21.7** Blockschaltbild des 25LC256-SPI-Speichers

Spannungen zwischen 2,5 V 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 [3]) 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 Endurance<sup>3</sup> liegt bei 1.000.000 Lösch-/Schreibzyklen (bei [3] bis zu 40.000.000).

### 21.2.2.1 Schrebschutz des Speichers

Über das nichtflüchtige Status-Register des Speichers kann man:

- den Schrebschutz unterschiedlicher Speicherbereiche ändern,
- die Schrebschutzkonfiguration 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 Schrebschutz des Status-Registers (siehe Tab. 21.2);

<sup>3</sup>Frei übersetzt: Lebensdauer.

**Tab. 21.1** 25LC256

Schreibgeschützte  
Speicherbereiche

BP1	BP0	Schreibgeschützter Speicherbereich
0	0	Keiner
0	1	0x6000 – 0x7FFF
1	0	0x4000 – 0x7FFF
1	1	0x0000 – 0x7FFF

- **Bit 3:2 – BP1:BP0** (Block Protection) – diese 2 Bits zeigen an, welcher Speicherbereich gerade schreibgeschützt ist (siehe Tab. 21.1);
- **Bit 1 – WEL** (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:
  - 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 – WIP** (Write-In-Proces) – dieses Bit ist von der internen Logik auf High gesetzt, solange ein interner Speicherzyklus aktiv ist;

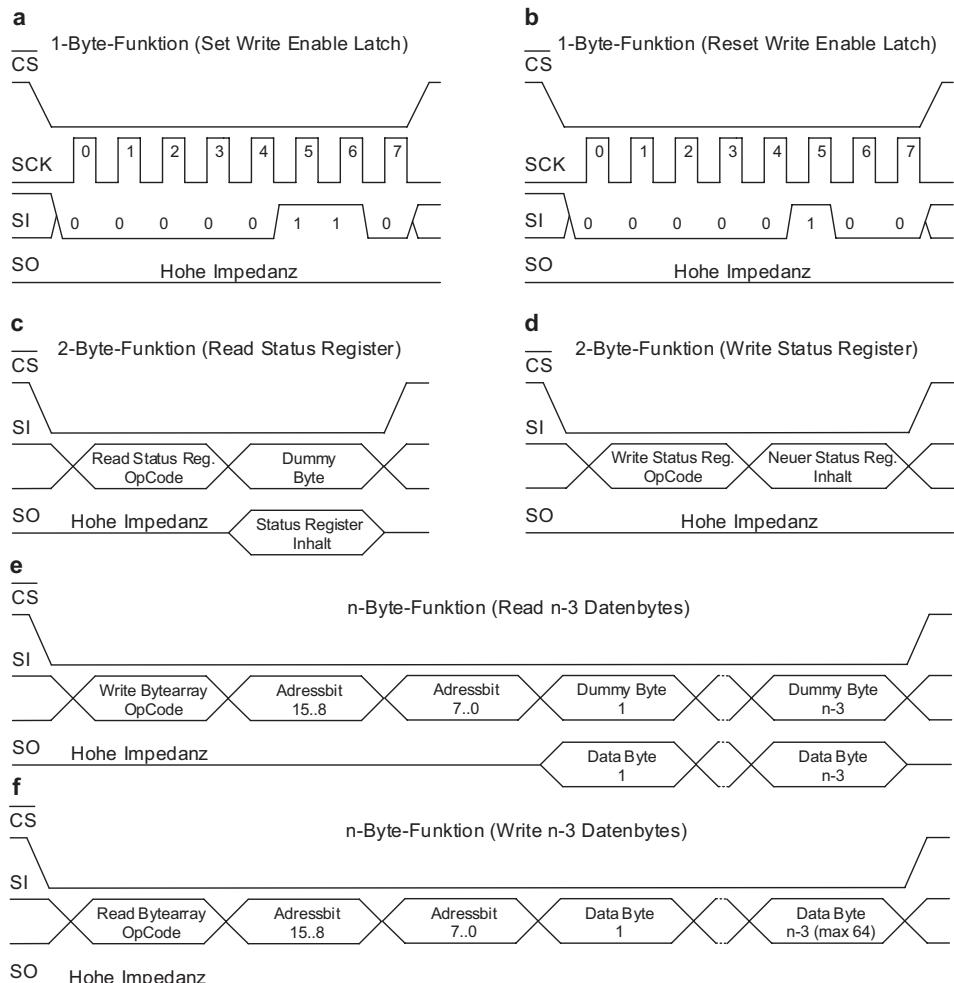
Die Bits 4, 5 und 6 sind nicht belegt.

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. 21.2 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.

**Tab. 21.2** 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



**Abb. 21.8** Überblick über die Funktionen des 25LC256

### 21.2.2.2 Schreib-Lese-Funktionen

Die interne Logik des Bausteins implementiert folgende Funktionen (siehe Abb. 21.8):

- **Read Byte Array:** 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:** 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:** der Write-Enable-Latch wird auf High gesetzt; dadurch wird das Speichern freigegeben (siehe Tab. 21.2).
- **Write Latch Disable:** Rücksetzen des Write-Enable-Latches; das Speichern wird dadurch gesperrt.
- **Read Status Register:** Das Status-Register wird ausgelesen.
- **Write Status Register:** Der Inhalt des Status-Registers wird geändert (nicht die WIP und WEL Bits).

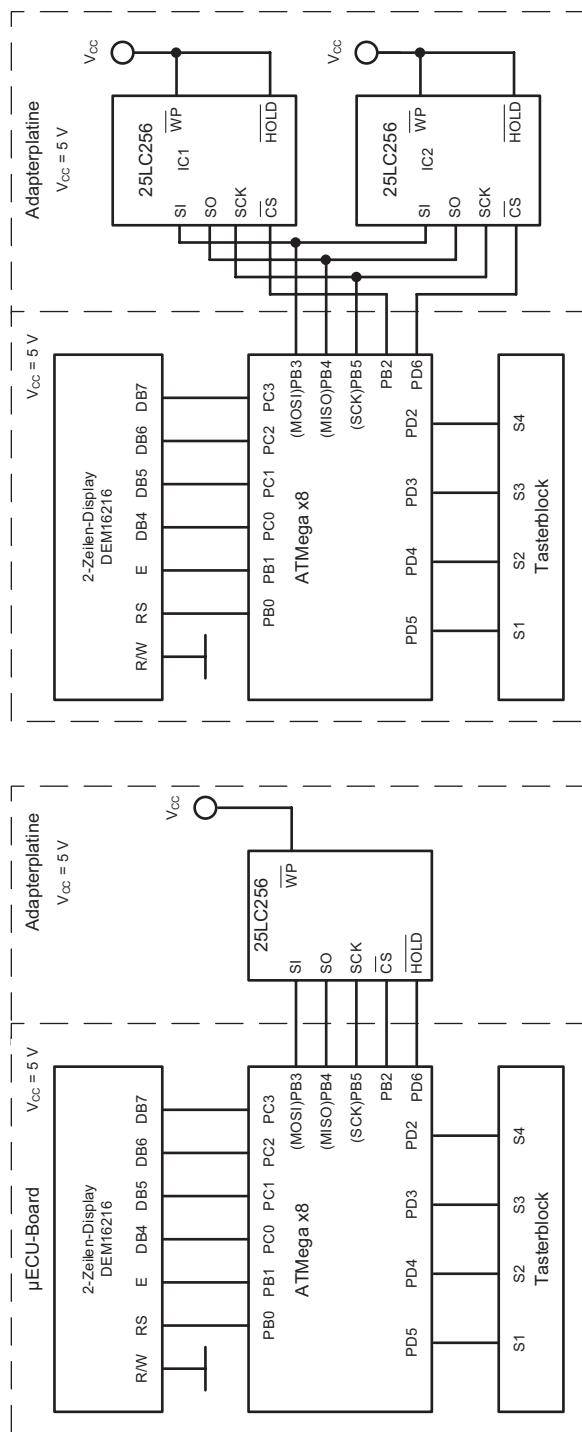
### 21.2.2.3 Speicher-Ansteuerung

Eine mögliche Beschaltung des 25LC256-Speichers ist in Abb. 21.9 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  $V_{CC}$  verbunden. 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. 21.9b) 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.

### 21.2.2.4 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.
- 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



**Abb. 21.9** Beschaltung eines 25LC256-Speichers

- 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. 21.9 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.

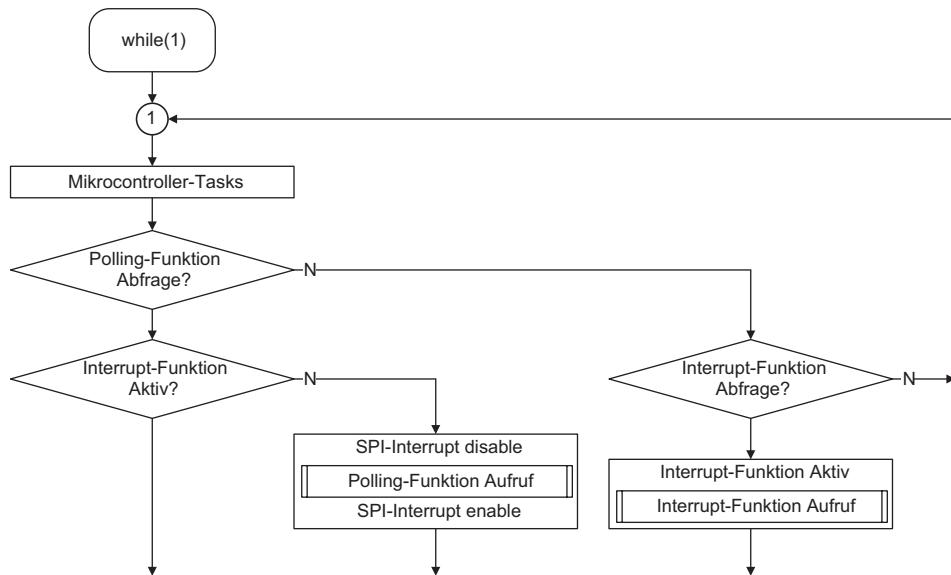
Ein Flussdiagramm für die Gestaltung der Endlosschleife nach beiden Verfahren ist in Abb. 21.10 zu sehen. Die Verwaltung der Interrupt-Funktionen wird mit Hilfe eines Zustandsautomaten in der Interrupt-Service-Routine der SPI-Schnittstelle implementiert.

### 21.2.2.5 Softwarebeispiel

Als Beispiel wird eine Read-Bytearray-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. 8 aufgelistet sind. Die Datenstruktur *\_25LC256\_pins* ist analog zu den Strukturen in Abschn. 14.5 aufgebaut.

```
void _25LC256_Read_ByteArray(_25LC256_pins sdevice_pins,
                           uint16_t uibyte_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 niederwertige Byte der Adresse wird ermittelt
```



**Abb. 21.10** Flussdiagramm: Ansteuerung von 25LC256-Speicherbausteine nach einer Mischung von Polling- und Interrupt-Verfahren

```

ucDummy = uibyte_address;
//das niederwertige 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. 14.5) mit der Pinkonfiguration in der `main`-Datei angepasst werden. Im Folgenden werden die Datenstrukturen der in der Abb. 21.9 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, OFF};
```

---

## Literatur

1. Stiny, L. (2018). *Aktive elektronische Bauelemente – Aufbau, Struktur, Wirkungsweise, Eigenschaften und praktischer Einsatzdiskreter und integrierter Halbleiter-Bauteile* (4. Aufl.). Springer Vieweg Wiesbaden.
2. STMicroelectronics. (2015). M24C24R – 64 kBit I<sup>2</sup>C-Bus serieller EEPROM. [www.st.com](http://www.st.com)
3. STMicroelectronics. (2015). M95256-xx SPI – serielle EEPROMs. [www.st.com](http://www.st.com)
4. Microchip Technology Inc. (2015). 25LC256 – 256 kBit SPI-Bus serieller EEPROM. [www.microchip.com](http://www.microchip.com)
5. Microchip Technology Inc. (2015). AT25256 – 256 kBit serieller SPI-BUS EEPROM. [www.microchip.com](http://www.microchip.com)
6. Microchip Technology Inc. ATmega88 – 8 Bit Microcontroller. [www.microchip.com](http://www.microchip.com)
7. Microchip Technology Inc. (2015). AVR107 – Interfacing AVR serial memories. [www.microchip.com](http://www.microchip.com)



# Serielle Flash-Speicher

22

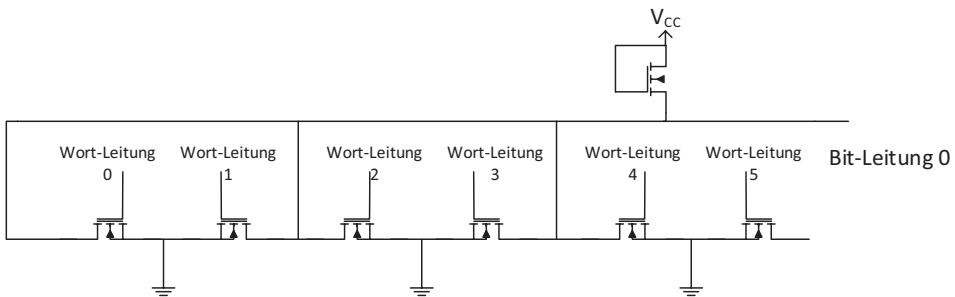
## Zusammenfassung

In diesem Kapitel werden zwei typische Flashspeicher mit serieller Kommunikation beschrieben.

Im Gegensatz zu den im Kap. 21 beschriebenen EEPROM-Bausteinen, zeichnen sich die hier vorgestellten Flash-Bausteine durch hohe Speicherdichte und schnelle Schreibvorgänge aus. Sie werden daher zum Abspeichern größerer Datenmengen verwendet. NOR-Flash-Speicher, die im Mikrocontroller vorwiegend als Programmspeicher dienen, werden üblicherweise parallel über einen Adressbus angesteuert und eignen sich daher für XIP (Execute-In-Place). Sie sind für Programmcode der Speichertyp der Wahl, da sie bauartbedingt blockweise gelöscht werden können, was den Programmiervorgang beschleunigt. NAND-Flashes sind pro Bit billiger und kleiner und eignen sich für hohe Speicherdichten, wie sie bei Speicherkarten vorkommen. Die hier vorgestellten seriellen Flashbausteine sind aufgrund des Datenzugriffs über eine SPI-Schnittstelle nicht als Programmspeicher geeignet. Dennoch fallen gelegentlich größere Datenmengen an, beispielsweise in Datenloggern oder in Speichern für Kennfelder. Die Autoren haben sie auch schon zum Abspeichern von abgetasteten Sprachnachrichten verwendet.

Die Originalversion dieses Kapitels wurde revidiert. Ein Erratum ist verfügbar unter  
[https://doi.org/10.1007/978-3-658-31709-6\\_27](https://doi.org/10.1007/978-3-658-31709-6_27)

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann  
[https://doi.org/10.1007/978-3-658-31709-6\\_22](https://doi.org/10.1007/978-3-658-31709-6_22).



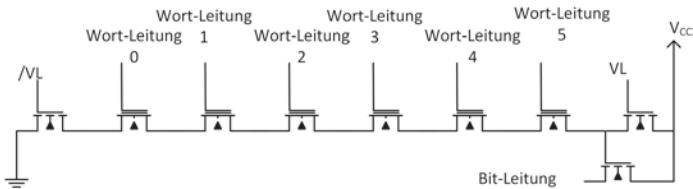
**Abb. 22.1** NOR-Flash

EEPROMs sind in der Regel so aufgebaut, dass jede einzelne Zelle gelöscht oder geschrieben werden kann. Um den Vorgang des Löschens zu beschleunigen und um Platz zu sparen, legt man größere Gruppen von Zellen zusammen, wie in Abb. 22.1 angedeutet. Die Abbildung zeigt ein NOR-Flash in NMOS-Technik mit fünf Speicherzellen von je einem Bit, wobei *Flash* für das schlagartige Löschen aller Blöcke steht. Das Auslesen geschieht mit Anlegen der Lesespannung  $U_{th}$  an eine der Wort-Leitungen, die Daten liegen dann an der Bit-Leitung bereit. Der 0–1-Übergang (Schreiben) erfolgt durch Anlegen der Schreibspannung an die zu schreibende Wortleitung mithilfe von hot-carrier-injection. Hierbei werden die Ladungsträger so stark beschleunigt, dass ihre Energie die Isolationsbarriere überwindet. Das Löschen erfolgt durch Anlegen der Schreibspannung an die Bitleitung, während alle Wortleitungen auf Masse liegen. Die so aufgebauten Ketten können mehrere Kilobyte umfassen. Der Begriff NOR leitet sich aus dem Verhalten an den Wortleitungen ab: Die Parallelschaltung aller open-Drain-Transistoren verhält sich wie ein logisches NOR-Gatter: Ist einer der Transistoren durchgeschaltet, so liegt der Ausgang auf logisch „0“, ist kein Transistor durchgeschaltet, auf logisch „1“. Möchte man die Wortbreite auf beispielsweise acht Bit erhöhen, werden je acht Zellen an den Wortleitungen parallel geschaltet, so dass acht parallele Bitleitungen das gesamte Wort ausgeben.

Die NOR-Flash Speichertechnologie zeichnet sich durch sehr kurze Lesezeiten und langsame Schreibzeiten aus und ist dort geeignet, wo es um schnelles Lesen geht, beispielsweise bei XIP (execution in place), also in Programmspeichern von Mikrocontrollern.

Flash-Speicher mit hoher Speicherdichte werden in NAND-Flash Technologie aufgebaut. Dabei werden nicht wie die in Kap. 21 beschriebenen selbstsperrenden Transistoren als Speicherzellen verwendet, sondern sogenannte Verarmungstypen (depletion type), die zunächst selbstleitend sind. Sie besitzen einen durchgängig dotierten N-Kanal zwischen Drain und Source. Bei einer positiven Spannung am Gate wird dieser Kanal durch das Verdrängen der Ladungsträger aus dem Kanal in die p-Zone unterbrochen, der Transistor sperrt.

Beim NAND Flash werden die Speicherzellen in Reihe geschaltet, wie in Abb. 22.2 zu sehen ist. Gegen Masse und gegen die Versorgungsspannung kann die Reihe durch



**Abb. 22.2** NAND-Flash nach [1]

zwei selbstsperrende FETs geschaltet werden, die im Gegentakt angesteuert werden. Es leitet also entweder der Transistor VL oder der Transistor /VL. Beim Lesen werden zunächst alle Gates der Speicherzellen auf Masse gelegt (mit „0“ angesteuert). Dadurch sind sie alle leitend. Das Gate von Transistor VL liegt auf High, so dass die Spannung an der Bitleitung ebenfalls auf High liegt, da der Transistor /VL gleichzeitig sperrt. Der Kondensator lädt sich auf. Zum Lesen wird nun VL gesperrt, /VL geöffnet und der zu lesende Zellentransistor über die Wortleitung angesteuert. Ist er nicht geladen, so sperrt er und am Ausgang liegt eine „1“. Ist er jedoch geladen, leitet er weiterhin und die Spannung an der Bitleitung geht auf 0, siehe [1]. NAND-Speicherzellen sparen jede Menge Anschlüsse untereinander, so dass sie deutlich kleiner ausfallen als NOR-Speicherzellen. Durch die Serienschaltung benötigt der Ladungsträgerfluss zum Aufbau jedoch wesentlich mehr Zeit, so dass die Lesegeschwindigkeit kleiner ist als die der NOR-Zellen.

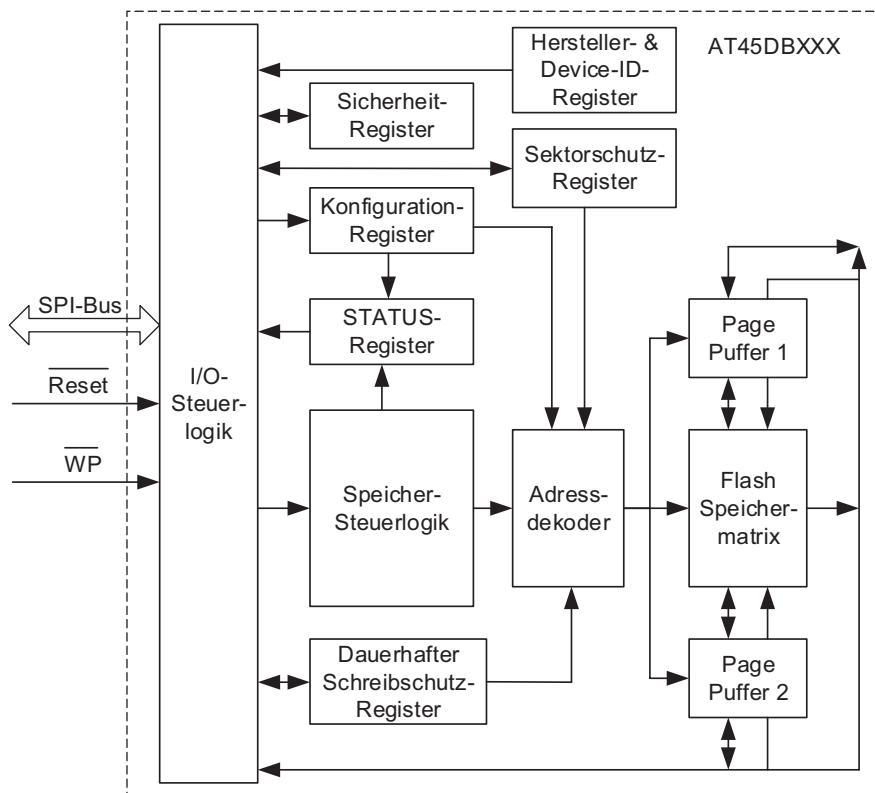
NAND-Flash-Bausteine werden überall da eingesetzt, wo es nicht so sehr auf Lesegeschwindigkeit sondern auf Schreibgeschwindigkeit und auf hohe Speicherdichte ankommt, beispielsweise in SD-Karten oder Memorysticks. Die Speicherdichte kann durch sogenannte Multi-Level-Cells (MLC) noch weiter erhöht werden, hier werden die Transistoren nicht bloß ein- und ausgeschaltet, sondern über die auf dem Floating Gate gespeicherte Ladungsmenge mit einem bestimmten Spannungspiegel beaufschlagt, so dass die Ausgangsspannung einen von  $n$  Pegeln annehmen kann. Somit können  $\log_2 n$  Bit pro Zelle gespeichert werden.

Technologiebedingt haben speziell NAND-Flashspeicher viele bereits in der Herstellung defekte Zellen. Daher wird neben den Daten auch eine CRC-Prüfsumme gespeichert, die für eine Vorwärtsfehlerkorrektur (FEC – Forward Error Correction, siehe Kap. 6) geeignet ist. Eine aufwendige Logik und ein Steuerwerk zum Lesen, Schreiben, Löschen und zur Fehlerkorrektur, sowie in der Regel ein RAM-Puffer, machen diese Bausteine relativ komplex. Zwei wesentliche Kenngrößen dienen der Charakterisierung:

- Die Retention-Time ist die Zeit, nach der die Daten noch sicher gelesen werden können. Sie liegt bei NOR-Flash bei ca. 20 Jahren, bei NAND-Flash bei etwa der Hälfte[2].
- Die Endurance gibt die Anzahl von Schreib- und Löschzyklen an. Diese liegt inzwischen bei circa 100.000 bis eine Million Zyklen und ist bei NAND-Flash generell etwas höher.

## 22.1 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 V 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 ([3]) näher betrachtet, dessen Blockschaltbild in Abb. 22.3 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 Zwischenspeichergröße können über das Konfigurationsregister auf  $512 (= 2^9)$  Bytes umgestellt werden. Es gibt unterschiedliche Lesefunktionen



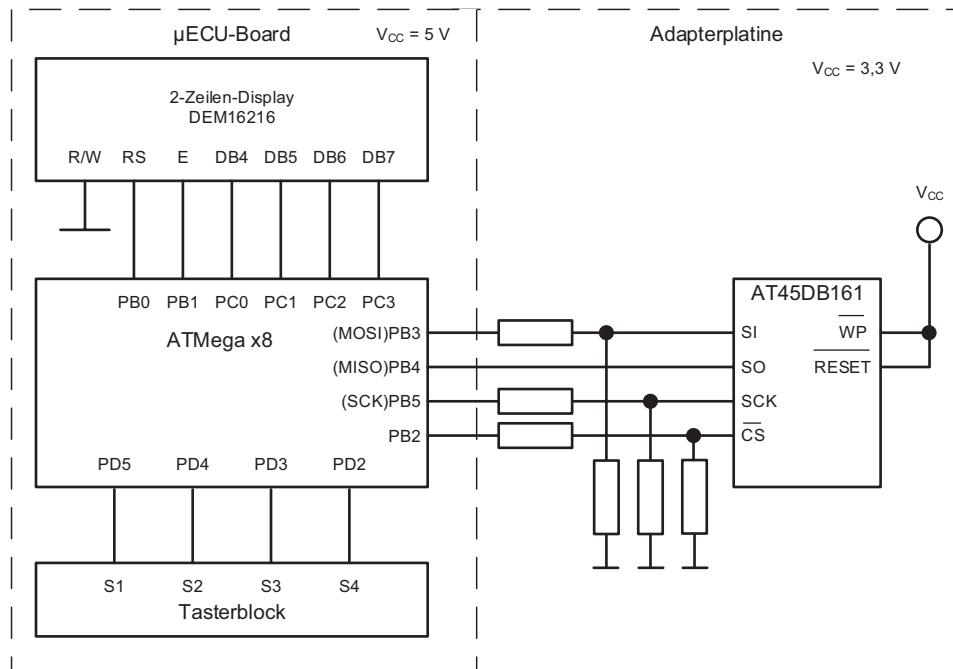
**Abb. 22.3** AT45DBXXX – Blockschaltbild

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.

### 22.1.1 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 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. 22.4 folgendermaßen definiert (Abschn. 14.5):



**Abb. 22.4** AT45DB161 – Beschaltung

**Tab. 22.1** 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

```
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};
```

### 22.1.2 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. 22.1) 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.

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 16-Bit-Adresse wird in einzelnen Bytes zerlegt  
    ucAddressLowByte = uibuffer_address;  
    uibuffer_address = uibuffer_address >> 8;  
    ucAddressMiddleByte = uibuffer_address;  
    //die Kommunikation wird gestartet  
    SPI_Master_Start(sdevice_pins.AT45DB161spi);  
    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. 22.4, 10 Bytes aus der Variable *ucBuffer[]* in den ersten SRAM-Zwischenspeicher ab Adresse 0 speichern.

### 22.1.3 Flash-Hauptspeicher

Der Baustein AT45DB161 ist ein 16-Mbit (2-Mbyte) großer Speicher mit einem Daten-erhalt von 20 Jahren und 100.000 Schreib-/Löschenzyklen für jede Seite. Die komplexe Speicherarchitektur des Bausteins ist in Abb. 22.5 dargestellt. Acht 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.

Sektor 0a = 1 Block = 8 Pages		Block 0	Page 0
Sektor 0a = 31 Blöcke = 248 Pages		Block 1	-----
Sektor 1 = 32 Blöcke = 256 Pages		Block 30	Page 6
		Block 31	Page 7
		Block 32	Page 8
		Block 33	Page 9
		Block 63	-----
		Block 480	Page 15
Sektor 15 = 32 Blöcke = 256 Pages		Block 510	-----
		Block 511	Page 4088
			Page 4094
			Page 4095

**Abb. 22.5** AT45DB161 – Speicherarchitektur

## 22.1.4 Lesen

Der Mikrocontroller als Master kann Daten aus dem Speicherbereich direkt (ohne Zwischenspeichern) oder indirekt (mit Zwischenspeichern) auslesen.

### 22.1.4.1 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 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 acht Takten 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, 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;
    //Start der Kommunikation
    SPI_Master_Start(sdevice_pins.AT45DB161spi);
    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);
    }
    //die Kommunikation wird beendet
    SPI_Master_Stop(sdevice_pins.AT45DB161spi);
}
```

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.

#### 22.1.4.2 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.

#### **22.1.4.3 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 µs 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.

#### **22.1.5 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. 22.2), 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.

**Tab. 22.2** 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 µs/Byte

### 22.1.5.1 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.

### 22.1.5.2 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-Kommunikation wird gestartet
    SPI_Master_Start(sdevice_pins.AT45DB161spi);
    SPI_Master_Write(ucop_code); //der Befehlscode 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).
```

### 22.1.5.3 Vergleich zwischen gespeicherte Seite und Quellpuffer

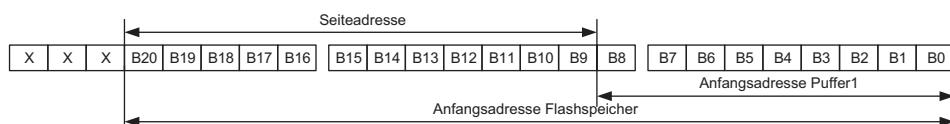
Um zu prüfen ob das Speichern eines gesamten Puffers im Flash fehlerfrei stattgefunden hat, kann man nach dem Vorgangsabschluss die zwei Inhalte vergleichen. Wenn die zwei Inhalte übereinstimmen, dann wird das Bit COMP im Status-Register auf „0“ gesetzt. Dafür muss der Master über den SPI-Bus den Befehlscode *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).

### 22.1.5.4 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 zwölf wird eine Seite adressiert und mit den letzten neun 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 Speichervorgang. Zuerst wird die adressierte Seite gelöscht und anschließend der gesamte Inhalt des ausgewählten Puffers in die Seite gespeichert.

### 22.1.5.5 Flash Speichern mit direkten Zugriff über Puffer 1

Mit dem Senden des Befehlscode *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. 22.6). Der Speicherbaustein wartet auf weitere bis zu 512 Bytes, die in den Puffer ab der Anfangsadresse zwischen gespeichert 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.



**Abb. 22.6** 24-Bit-Adresse-Zusammensetzung für Flash-Speichern mit direktem Zugriff

**Tab. 22.3** AT45DB161 – Löschzeiten

Speicherbereich	Löschzeit	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

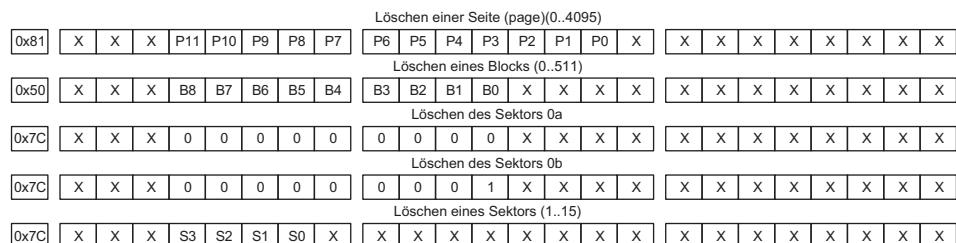
## 22.1.6 Löschen

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 Löschzeiten sind in der Tab. 22.3 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. 22.7 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 Löschvorgang startet, währenddessen das Bit RDY/BUSY aus dem STATUS-Register auf 0 gesetzt ist. Das bedeutet, dass der Speicher besetzt ist. In dieser Zeit sind das Auslesen der STATUS- und Device-ID-Register und das Schreiben in die Zwischenspeicher erlaubt.

## 22.1.7 Speicherschutz

Für den Schreibschutz unterschiedlicher Speicherbereiche sind mehrere Schreibschutzmechanismen implementiert.



**Abb. 22.7** AT45DB161 – Löschsequenzen

### 22.1.7.1 Temporärer Schrebschutz (Sector Protection)

Der Baustein besitzt ein nichtflüchtiges, 16-Byte großes Schrebschutz-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 Schrebschutz 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 Schrebschutz aktiviert und mit `0x3D 2A 7F 9A` deaktiviert.

### 22.1.7.2 Dauerhafter Schrebschutz (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.

## 22.1.8 Testbarkeit

Wird der Baustein in aufwendigen Schaltungen eingebaut, sind wegen seiner Komplexität und Speichergröße umfangreiche Testmöglichkeiten gefordert, die Auskunft über den Zustand und Identität des Bausteins geben.

### 22.1.8.1 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 Sektorenschrebschutz aktiv ist
  - **Bit0 – 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

### 22.1.8.2 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.

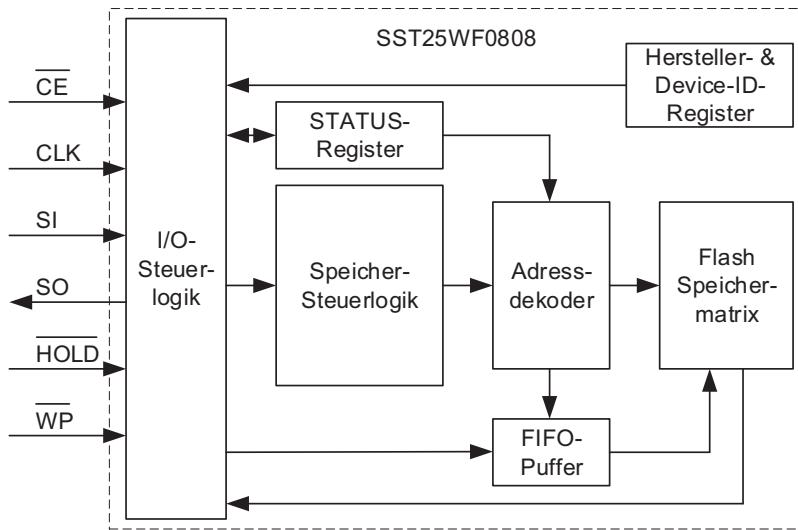
### 22.1.8.3 Hersteller- und Chip-ID-Register

Dieses Read-Only-Register speichert Informationen über den Hersteller und den Baustein im JEDEC-Standard, die im System zur Identifikation des Speichers dienen können.

---

## 22.2 SST25WF0808 serieller Flashspeicher

Der Baustein SST25WF0808 ist ein serieller Flashspeicher der Familie SST25XXYY der Firma Microchip Technology [4]. 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. 22.8 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.



**Abb. 22.8** SST25WF0808 – Blockschaltbild

### 22.2.1 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:

```
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 (Kap. 14 Abschn. 14.5), 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} ;

```

## 22.2.2 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. 22.4);
- **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 [4]); 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. 22.5). 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. 22.4** 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. 22.5** 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
    }
}
```

### 22.2.3 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 ansteuer-
    //baren Pins
    uint8_t ucDataByte;
    SPI_Master_Start(sdevice_pins.SST25PFXXspi); //CS auf Low
    //der Lesebefehl des Status-Registers wird übertragen
    SPI_Master_Write(0x05);
    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 ansteuer
    //baren 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;
    //High-Byte der Adresse wird übertragen
    SPI_Master_Write(ucAddressHigh);
    //Middle-Byte der Adresse wird übertragen
    SPI_Master_Write(ucAddressMiddle);
    //Low-Byte der Adresse wird übertragen
    SPI_Master_Write(ucAddressLow);
    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;
}
```

#### 22.2.4 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 Löschevorgang. 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
    Pins  SST25_WriteEnable(sdevice_pins); //Schreibfreigabe
    SPI_Master_Start(sdevice_pins.SST25PFXXspi); //CS auf Low
    //der gesamte Inhalt des Speichers wird gelöscht
    SPI_Master_Write(0x60);
    SPI_Master_Stop(sdevice_pins.SST25PFXXspi);      //CS auf High
}

```

## 22.2.5 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. 22.5), 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;
//High-Byte der Adresse wird übertragen
SPI_Master_Write(ucAddress_High);
//Middle-Byte der Adresse wird übertragen
SPI_Master_Write(ucAddress_Middle);
//Low-Byte der Adresse wird übertragen
SPI_Master_Write(ucAddress_Low);
//das zu schreibende Byte wird übertragen
SPI_Master_Write(ucbytetowrite);
SPI_Master_Stop(sdevice_pins.SST25PFXXspi); //CS auf High
}
```

## 22.2.6 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 vier 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.

---

## Literatur

1. Stiny, L. (2018). *Aktive elektronische Bauelemente – Aufbau, Struktur, Wirkungsweise, Eigenschaften und praktischer Einsatzdiskreter und integrierter Halbleiter-Bauteile* (4. Aufl.). Springer Vieweg Wiesbaden
2. Aravindan, A. (2021). Flash 101: NAND Flash vs NOR Flash, embedded 2018. <https://www.embedded.com/flash-101-nand-flash-vs-nor-flash/>. Zugegrifen: 1. Mar. 2021
3. Microchip Technology Inc. AT45DB161 – 16 Mbit SPI Serial Flash Memory. <http://ww1.microchip.com/downloads/en/devicedoc/doc2224.pdf>. Zugegriffen: 4. Apr. 2021.
4. Microchip Technology Inc. SST25WF080B – 8 Mbit 1,8 V SPI Serial Flash. [www.microchip.com](http://www.microchip.com). Zugegriffen: 2. Apr. 2021.



# Bausteine für die Audiotechnik

23

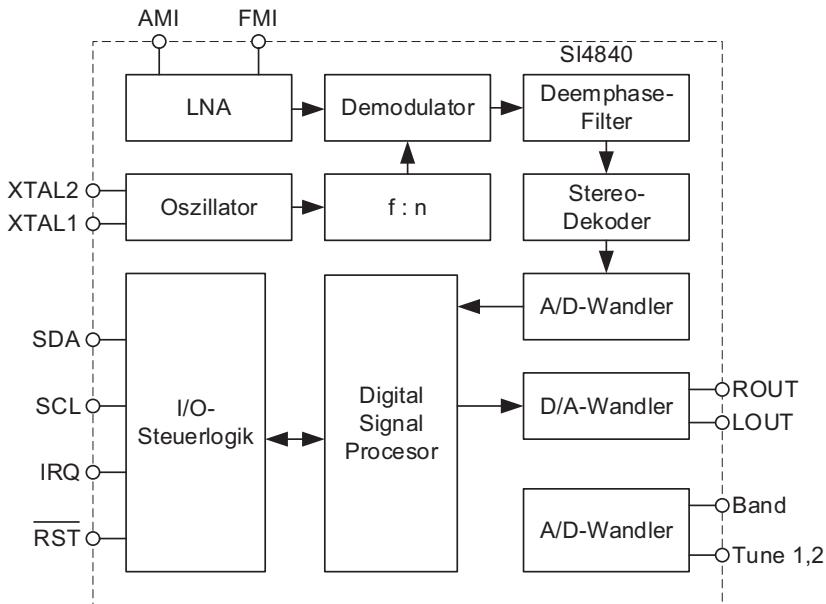
## Zusammenfassung

In diesem Kapitel werden zwei Bausteine beschrieben, die zusammen ein Radio mit Verstärker bilden. Beide werden über I<sup>2</sup>C angesteuert.

Im Bereich der Unterhaltungselektronik werden für die Bedienung häufig Mikrocontroller eingesetzt. Außerdem ist der Einsatz von Aktivboxen beliebt, bei denen der Verstärker in die Box eingebaut ist. In diesen Fällen macht es Sinn, über ein Bus-system steuerbare Bausteine einzusetzen. Man unterscheidet zwischen Bausteinen, bei denen die Mediensignale ebenfalls digital übertragen werden und solchen, in denen lediglich die Steuersignale (command & control) digital übertragen werden, während der Medienkanal weiterhin analog bleibt. Wir haben uns für dieses Buch entschieden, lediglich zwei Bausteine der letzteren Kategorie vorzustellen, da die ersten in der Regel zu komplex für eine Einführung in diesem Rahmen sind.

Die Originalversion dieses Kapitels wurde revidiert. Ein Erratum ist verfügbar unter  
[https://doi.org/10.1007/978-3-658-31709-6\\_27](https://doi.org/10.1007/978-3-658-31709-6_27)

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann  
[https://doi.org/10.1007/978-3-658-31709-6\\_23](https://doi.org/10.1007/978-3-658-31709-6_23).



**Abb. 23.1** SI4840-Blockschaltbild (Nach [2])

## 23.1 SI4840 Radio-IC

Der Baustein SI4840 ([2–4]) von der Fa. Silicon Laboratories ist ein AM<sup>1</sup>/FM<sup>2</sup>-Empfänger mit einem Stereo-Decoder so wie es in Abb. 23.1 dargestellt ist. Er gehört zu einer Reihe von ATDD<sup>3</sup>-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. 23.1) 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 Empfangsqualität automatisch zwischen Monophonie und Stereophonie umgeschaltet. Es werden dafür drei Empfangsqualitätsparameter überwacht:

<sup>1</sup>AM – Amplitudenmodulation.

<sup>2</sup>FM – Frequenzmodulation.

<sup>3</sup>ATDD – Analog Tune Digital Display.

**Tab. 23.1** 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

- die Empfangsfeldstärke – RSSI<sup>4</sup>;
- der Signal-Rausch-Abstand (oder Störabstand) – SNR<sup>5</sup>;
- Intersymbol-Interferenzen wegen dem Mehr-Wege-Empfang.

Sobald einer dieser drei Parameter unter einer digital einstellbaren Grenze fällt, wird auf Mono-Betrieb umgeschaltet.

### 23.1.1 Bausteinbeschreibung

Das Rundfunksignal wird über eine passende Antenne (siehe [4]) empfangen, die an Pin FMI (für UKW<sup>6</sup>) oder AMI (für MW<sup>7</sup>) angeschlossen ist und mit einem rauscharmen Verstärker LNA<sup>8</sup> verstärkt. Der Demodulator benötigt ein Taktsignal mit einer Frequenz von 32.768 Hz. Dieses Taktsignal kann mit Hilfe eines Uhrenquarzes mit einer Genauigkeit von  $\pm 100$  ppm<sup>9</sup> generiert, oder mit einem externen Oszillatator erzeugt. Die Frequenz des externen Oszillators wird mit dem internen einstellbaren Frequenzteiler (:1...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

<sup>4</sup>RSSI – Received Signal Strength Indication.

<sup>5</sup>SNR – Signal to Noise Ratio.

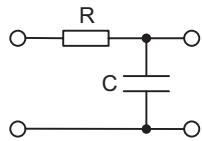
<sup>6</sup>UKW- Ultrakurzwelle.

<sup>7</sup>MW – Mittelwelle.

<sup>8</sup>LNA – Low Noise Amplifier.

<sup>9</sup>ppm – parts per million; 1 ppm = 0,000001.

**Abb. 23.2** SI4840  
Deemphase-Filter



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 Deemphase-Filter (Tiefpassfilter 1. Ordnung Abb. 23.2) wieder hergestellt werden, was zur Absenkung der höheren Frequenzanteile führt. In Europa wird ein Tiefpassfilter mit einer -3dB-Grenzfrequenz von 3,18 kHz verwendet

$$f_{-3\text{dB}} = \frac{1}{2\pi RC} \quad (23.1)$$

und einer Zeitkonstante von

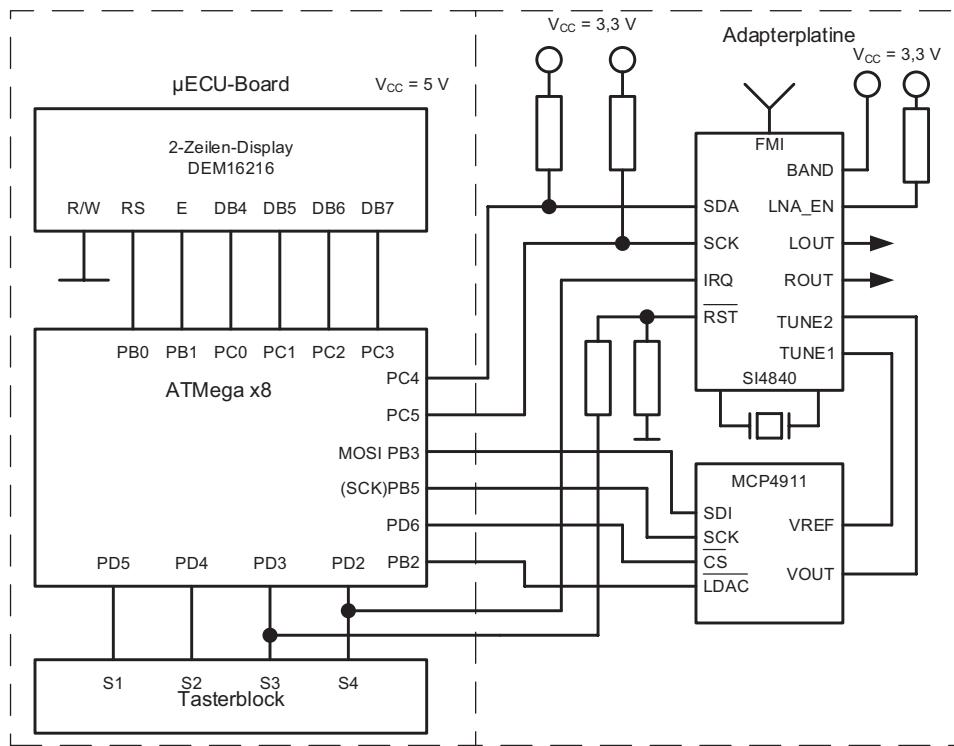
$$\tau = RC = 50 \mu\text{s}. \quad (23.2)$$

Mit diesem Rauschunterdrückungsverfahren ist eine Verbesserung des Rausch-Signal-Abstandes von bis zu 7,8 dB möglich [1]. In den USA wird ein Filter mit einer Zeitkonstante von 75  $\mu\text{s}$  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.

### 23.1.2 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, 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 [4]. 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.



**Abb. 23.3** SI4840 Frequenzband-Auswahl und Frequenzabstimmung über einen Mikrocontroller

In Abb. 23.3 wird eine Schaltung für die Frequenzband-Auswahl und Frequenzabstimmung vorgestellt, die schaltungstechnisch einfach zu realisieren 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. 23.1. 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 Kap. 20) 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 Spannungsspeigel vom Pin TUNE1 geachtet werden.

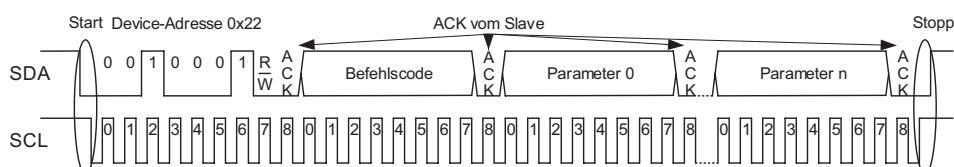
### 23.1.3 Initialisieren des Bausteins

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 dieser 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.

### 23.1.4 Kommunikation mit dem Baustein

Mit dem Baustein kann man mit einer Ansteuerung von einem Mikrocontroller einen kompletten Rundfunkempfänger aufbauen. Der Baustein implementiert ein I<sup>2</sup>C- (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. 23.4 zu sehen. Nach einer I<sup>2</sup>C-Startsequenz sendet der 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 Takten empfängt der Master das erste Byte, das so genannte Status-Byte, das Informationen in kodierter Form über den letzten Befehl



**Abb. 23.4** SI4840 Befehlsaufbau

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, CTS<sup>10</sup>, 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.

### 23.1.4.1 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:

- 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.

---

<sup>10</sup>CTS – Clear to Send (Empfangsbereitschaft).

- 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 oberen 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.

#### 23.1.4.2 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 diesem Befehl.

#### 23.1.4.3 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. [23.1.5](#) vorgestellt.

#### 23.1.4.4 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.

#### **23.1.4.5 Set\_Property-Befehl**

Eigenschaften des Bausteins wie Taktfrequenz, Frequenzteiler, Lautstärke, Hoch- und Tieftonregelung und Ein- und Abschalten eines Audiokanals können mithilfe 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 [3].

#### **23.1.4.6 Get\_Property-Befehl**

Die mit dem Befehl Set\_Property geänderten Eigenschaften, können mithilfe 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.

#### **23.1.4.7 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.

### 23.1.5 Sendersuche mit dem SI4840

Der Baustein 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 als Parameter beim Aufruf einer Funktion überliefert wird, vorgestellt.

Der Baustein liefert am Pin TUNE1 für die potenziometrische Einstellung der analogen Abstimmspannung am Pin TUNE2 eine Spannung von 1,26 V für das Frequenzband von 88...108 MHz. Wie in Abb. 23.3 steuert der Mikrocontroller die Abstimmspannung für den Radiobaustein über einen 10-Bit-D/A-Wandler vom Typ MCP4811. Über einen Spannungsteiler bestehend aus  $R1=1\text{ k}\Omega$  und  $R2=1,6\text{ k}\Omega$  erreicht die Abstimmspannung einen Wert von 1,26 V bei 2,048 V am Ausgang des D/A-Wandlers. Die SPI-Datenstruktur des D/A-Wandlers (siehe Kap. 20) lautet:

```
MCP48XX_pins  MCP4811_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\_Voltage mit dem Prototyp:

```
void MCP48XX_Set_OutputVoltage(MCP48XX_pins sdevice_pins,
                                uint8_t ucdevice,
                                uint8_t ucchannel,
                                uint8_t ucout, float fvalue)
```

stellt am Ausgang des D/A-Wandlers den Spannungswert ein, der beim Aufruf der Funktion als Parameter `fvalue` (in mV) übergeben wurde. Diese Funktion wurde für die Ansteuerung aller Bausteine dieser Familie gedacht. Beim Aufruf der Funktion müssen als Parameter: die SPI-Datenstruktur, der BausteinTyp (z. B. MCP481X für einen 10-Bit-D/A-Wandler), der Kanal (CHANNEL0 für einen Einzelkanal-D/A-Wandler), die Ausgangsfreigabe (OUTPUT\_ENABLE) und der einzustellende Spannungswert in mV übergeben werden.

Nach der Änderung der Abstimmspannung braucht der Radiochip erfahrungsgemäß ca. 200 ms, um die neue Frequenz zu schalten. Im Folgenden wird ein Algorithmus vorgestellt, um eine neue Frequenz zu suchen. Die Suche beginnt bei einer Abstimmspannung von 1024 mV, die Hälfte der Spannung für die obere Grenze des Frequenzbands. Eine Hilfsvariable uiVoltageMax speichert diesen Wert.

Schritt 1: – Die Abstimmspannung wird auf den Wert uiVolt2Search = uiVoltMax eingestellt;

Schritt 2: – es werden 200 ms gewartet;

Schritt 3: – Der Radiochip meldet auf Anfrage die erreichte Frequenz. Ist diese Frequenz die Gesuchte, dann ist die Suche erfolgreich beendet. Wenn nicht, dann weiter mit Schritt 4.

Schritt 4: – der Wert der Variable uiVoltMax wird halbiert. Ist der neue Wert 0, dann ist die Suche erfolglos beendet.

Schritt 5: – Wenn die eingestellte Frequenz größer als die Gesuchte ist, dann: uiVolt2Search -= uiVoltMax, ansonsten uiVolt2Search += uiVoltMax. Weiter mit Schritt 1. Spätestens nach 10 Wiederholungen wird ein Ergebnis erreicht.

Der vorgestellte Algorithmus wird mithilfe der Frequenz SI4840\_Search\_Freq implementiert. Die Funktion liefert das Suchergebnis in eine Datenstruktur:

```
typedef struct
{
    uint8_t ucStatus;
    uint8_t ucFBand;
    uint8_t ucFreq1;
    uint8_t ucFreq2;
} SI4840_response;
```

Durch die Auswertung der Rückgabe dieser Funktion kann zusätzlich ermittelt werden ob an der gesuchten Frequenz der Empfang von guter Qualität ist, oder ob die Sendung Stereo ist.

```
uint8_t SI4840_Search_Freq(MCP48XX_pins sdevice_pins,
                           SI4840_response *sresponse,
                           uint16_t uifrequency)
{
    uint16_t uiFreq2Comp, uiFreq;
    //Begrenzung der Frequenz auf 108 MHz
    if(uifrequency<1081) uiFreq=uifrequency;
    else uiFreq=1080;
```

```
if(ucFreqIndex==0)
{ //bei der Frequenzsuche wird im 1. Durchlauf die Abgleichsspannung
    auf 1024 mV gesetzt
    uiVolt2Search=uiVoltMax;
    MCP48XX_Set_OutputVoltage(sdevice_pins, MCP481X, CHANNEL0,
                               OUTPUT_ENABLE, uiVolt2Search);
    ucFreqIndex++;
}
else
{
    /*bei den nächsten Durchläufe wird der Status des Radio-
     chips gefordert*/
    if(SI4840_Get_ATDDStatus())
    {
        TWI_Master_Stop();
    }
    // und seine Antwort eingelesen
    if(SI4840_Get_ATDDResponse(ucStatusArray, STATUS_RESPONSE_
BYTE))
    {
        TWI_Master_Stop();
    }
    //es wird die erreichte Frequenz berechnet.
    uiFreq2Comp=(uint16_t)((ucStatusArray[2]>>4) * 1000) +
        (uint16_t)((ucStatusArray[2] & 0x0 F) * 100) +
        (ucStatusArray[3]>>4) * 10 + (ucStatusArray[3] &
        0x0 F);
    if(uiFreq2Comp==uiFreq)
    {
        /*wenn die gesuchte Frequenz gleich mir der erreichten
         ist, wird eine neue Suche initialisiert*/
        uiVoltMax=1024;
        ucFreqIndex=0;
        //und die Antwort des Radiochips in das Antwort-Frame
        gespeichert
        sresponse->ucStatus=ucStatusArray[0];
        sresponse->ucFBand=ucStatusArray[1];
        sresponse->ucFreq1=ucStatusArray[2];
        sresponse->ucFreq2=ucStatusArray[3];
        //Frequenz erreicht wird rückgemeldet
        return FREQUENCY_REACHED;
    }
    else
    {
```

```

        /*wenn die zwei Frequenzen unterschiedlich sind, wird die
        Spannung uiVoltMax halbiert*/
        uiVoltMax=uiVoltMax>>1;
        if(uiVoltMax==0)
        {
            /*wenn die Spannung den Wert Null erreicht, ist
            die Suche beendet ohnedie gewünschte Frequenz
            zu erreichen; die Variablen für eine neue Suche
            werden initialisiert*/
            uiVoltMax=1024;
            ucFreqIndex=0;
            //Frequenz nicht erreicht wird zurückgemeldet
            return FREQUENCY_NOT_REACHED;
        }
        else
        {
            //der neue Spannungswert wird berechnet
            if(uiFreq2Comp>uiFreq) uiVolt2Search
            -=uiVoltMax;
            else if(uiFreq2Comp<uiFreq)
            uiVolt2Search+=uiVoltMax;

            MCP48XX_Set_OutputVoltage(sdevice_pins,
                MCP481X, CHANNEL0,
                OUTPUT_ENABLE,
                uiVolt2Search);
            //Frequenz wird gesucht wird zurückgemeldet
            return FREQUENCY_SEARCH;
        }
    }
    /*diese Anweisung wird gebraucht nur um eine Warnung des Compilers
    zu vermeiden*/
    return FREQUENCY_SEARCH;
}

```

Um blockierendes Warten zu vermeiden, wird die Frequenzsuche in der main-Funktion in einem 200-ms-Task durchgeführt. Die Variable ucSearchFlag wird am Anfang der Suche gesetzt und beim Erreichen der gesuchten Funktion zurückgesetzt.

```

if(Timer1_get_200msState() == TIMER_TRIGGERED)
{
    if(ucSearchFlag)
    {
        ucResponseFlag=SI4840_Search_Freq(MCP48XX_1, &sResponse_Frame,
            uiFreq2Search);
    }
}

```

```
    if(ucResponseFlag == FREQUENCY_REACHED)
    {
        //das Ergebnis der Suche wird ausgewertet und
        ucSearchFlag=0;
    }
}
}
```

## 23.2 LM48100Q Verstärker-Baustein

Der LM48100 ist ein analoger symmetrischer Mono-Verstärker (AB-Brückenverstärker) von Texas Instruments, der sich beispielsweise für Audioendstufen in Autos eignet [5]. Er besitzt eine Reihe von Schutzfunktionen und liefert eine Ausgangsleistung von 1,3 W auf einer  $8\ \Omega$  Last bei einer Versorgungsspannung von 3 V bis 5,5 V. Damit kann man keine Säle füllen, er eignet sich aber für kleine Aktivboxen, die über einen Mikrocontroller gesteuert werden. Der Baustein verfügt über zwei Eingänge, die sich mischen lassen, wobei jeder Eingang die Lautstärke in 32 Schritten verändern kann. Mischer, Lautstärkeregler und die Konfiguration des Bausteins werden über einen I<sup>2</sup>C-Anschluss gesteuert. Die Kommunikation findet mit bis zu 400 kBit/s statt.

Endstufenfehler, wie Kurzschlüsse an einer Lautsprecherleitung nach Masse oder Versorgungsspannung, Leitungsbruch, Schlüsse zwischen den Lautsprecherleitungen, Überstrom und Überhitzung können kontinuierlich detektiert werden. Im Fehlerfall wird ein Pin auf Masse gelegt, so dass man über einen Porteingang den Fehler mit dem Mikrocontroller identifizieren kann. Die Diagnose lässt sich so steuern, dass man Fehler eingrenzen kann.

Ebenso lässt sich der Baustein über I<sup>2</sup>C schlafen legen, damit der Stromverbrauch reduziert wird. Eine Blockschaltbild des Bausteins ist in Abb. 23.5 wiedergegeben.

Die I<sup>2</sup>C Geräteadresse des LM48100Q-Q1 ist 111110X, wobei X durch die Leitung ADR bestimmt wird. ADR=1 setzt die Geräteadresse auf 1111101. ADR=0 setzt die Geräteadresse auf 1111100.

Die Programmierung des Bausteins erfolgt, indem an die Geräteadresse (MSB first) ein Byte (MSB first) gesendet wird, mit den in Tab. 23.2 genannten Bedeutungen.

Bei einem der oben erwähnten Fehlerzustände wird die /FAULT-Leitung von open-drain auf Masse gezogen und kann damit über einen Port ausgelesen werden.

Im Einzelnen bedeuten die Bits:

Power On: 1 schaltet den Verstärker ein, 0 schaltet den Verstärker aus

INPUT\_X: 1 schaltet den Eingangspfad X ein, 0 schaltet ihn aus

DG\_EN: Schaltet mit 1 die Diagnose ein (Beim Reset wird einmal immer die Diagnose durchlaufen)

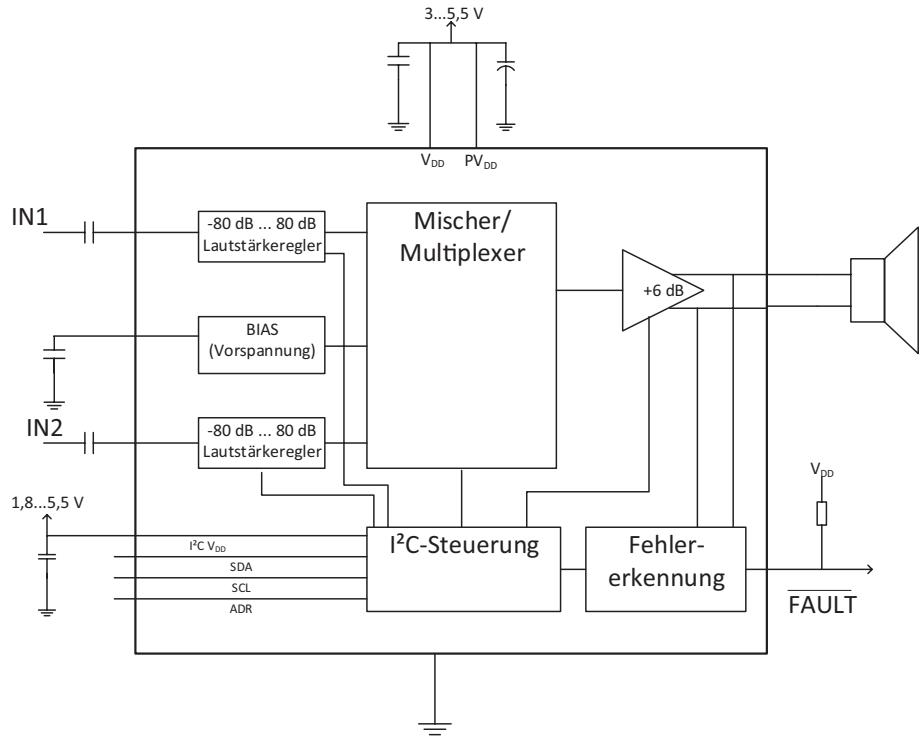


Abb. 23.5 Blockschaltbild des LM48100Q. (Nach [5])

Tab. 23.2 Botschaften für den LM48100Q

Adr	Name	B7	B6	B5	B4	B3	B2	B1	B0
0	MODE CONTROL	0	0	0	Power On	Input 2	Input 1	0	0
1	DIAGNOSTIC CONTROL	0	0	1	DG_EN	DG_Cont	DG_Reset	ILimit	0
2	FAULT DETECTION CONTROL	0	1	0	TSD	OCF	RAIL_SHT	OUTPUT_OPEN	OUTPUT_SHORT
3	VOLUME CONTROL 1	0	1	1	VOL_1_4	VOL_1_3	VOL_1_2	VOL_1_1	VOL_1_0
4	VOLUME CONTROL 2	1	0	0	VOL_2_4	VOL_2_3	VOL_2_2	VOL_2_1	VOL_2_0

DG\_CONT: Schaltet mit 1 einen kontinuierlichen Diagnosemodus ein, d. h. Ausgangskurzschluss auf  $V_{DD}$  und GND, Ausgänge gegeneinander kurzgeschlossen und „keine Last“, also Leitungsbruch werden kontinuierlich gemessen. Bei 0 wird die Diagnose nur einmal nach einem Start mit DG\_EN durchgeführt. Auch ohne Diagnose bleibt die Schutzfunktion erhalten.

DG\_RESET: Nach einem Fehler wird mit einer 1 der Fehlerzustand gelöscht und die/FAULT-Leitung wieder auf open-drain gesetzt.

Die Ausgangstests des LM48100Q werden individuell über das Fault-Detection-Control-Register gesteuert. Wird eines der Bits im Fault-Detection-Control-Register auf 1 gesetzt, ignoriert die FAULT-Schaltung den zugehörigen Test. Wenn beispielsweise B2 (RAIL\_SHT)=1 ist und der Ausgang mit  $V_{DD}$  kurzgeschlossen ist, bleibt der FAULT-Ausgang auf high. Obwohl die FAULT-Schaltung den ausgewählten Test ignoriert, bleibt die Schutzschaltung des LM48100Q dennoch aktiv und schaltet das Gerät ab. Wenn DG\_EN=1 ist und eine Diagnosesequenz eingeleitet wird, werden alle Tests unabhängig vom Zustand des Fehlererkennungssteuerungsregisters durchgeführt. Wenn DG\_EN=0 ist, werden die Tests RAIL\_SHT, OUTPUT\_OPEN und OUTPUT\_SHT nicht durchgeführt, jedoch bleiben die Schaltkreise zur Erkennung von thermischer Überlast und Ausgangsüberstrom aktiv [5].

Die Bits bedeuten:

OUTPUT\_SHT: Kurzschluss zwischen den symmetrischen Lautsprecherleitungen

OUTPUT\_OPEN: Leitungsbruch

RAIL\_SHT: Kurzschluss gegen  $V_{DD}$  oder GND

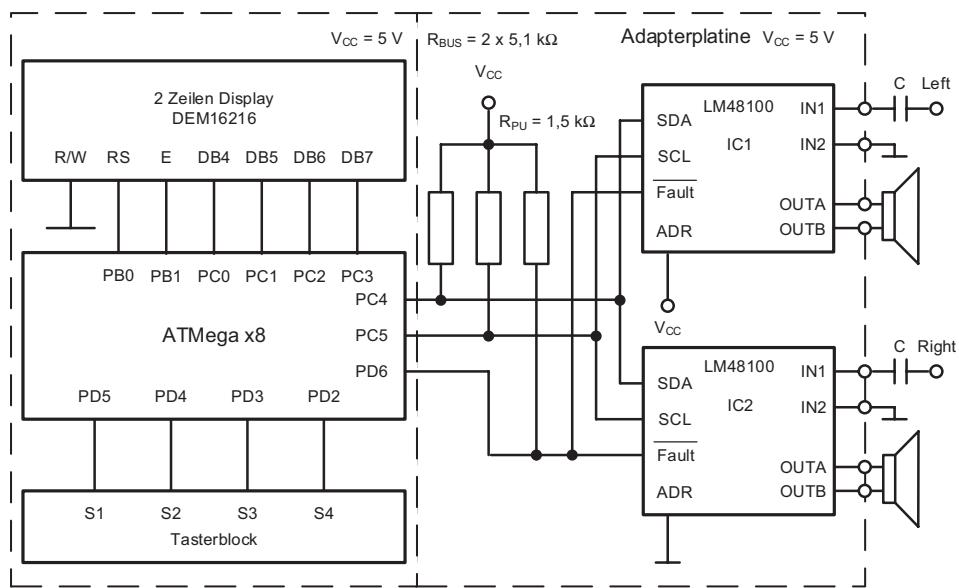
OCF: Überstrom (over current)

TSD: Thermische Überlast

Schließlich werden die VOLUME CONTROL Bits in Stufen von 0... 31 was einer Verstärkung von  $-80 \text{ dB} \dots +18 \text{ dB}$  entspricht, gemäß einer Hökurve eingestellt. Diese ist im Datenblatt [5] tabellarisch aufgeführt.

Die Abb. 23.6 zeigt die Beschaltung eines Stereo-Verstärkers mit dem Baustein LM48100. Die zwei Verstärker werden von einem Mikrocontroller ATmega88 angesteuert.

Nach der Initialisierung der TWI-Schnittstelle werden die zwei Verstärker ein- und ausgeschaltet mit der Funktion LM48100\_Set\_Power.



**Abb. 23.6** Beschaltung eines Stereo-Verstärkers mit LM48100

```

uint8_t LM48100_Set_Power(uint8_t ucamp1_enable, uint8_t ucpowerleft,
                           uint8_t ucpowerright)
{
    switch(ucamp1_enable)
    {
        case AMPLIFIER_LEFT:
            if(LM48100_Write_Mode_Ctrl(AMPLIFIER_LEFT, ucpowerleft))
                return TWI_ERROR;
            break;

        case AMPLIFIER_RIGHT:
            if(LM48100_Write_Mode_Ctrl(AMPLIFIER_RIGHT, ucpowerright))
                return TWI_ERROR;
            break;

        case AMPLIFIER_BOTH:
            if(LM48100_Write_Mode_Ctrl(AMPLIFIER_LEFT, ucpowerleft))
                return TWI_ERROR;
            if(LM48100_Write_Mode_Ctrl(AMPLIFIER_RIGHT, ucpowerright))
                return TWI_ERROR;
            break;
    }
    return TWI_OK;
}

```

Wobei:

```
#define LM48100_DEVICE_TYPE_ADDRESS 0xF8
#define AMPLIFIER_LEFT 0x00
#define AMPLIFIER_RIGHT 0x01
#define AMPLIFIER_BOTH 0x02

//Mode Control Register
#define POWER_OFF 0x00 //Verstärker abgeschaltet
#define POWER_ON_MUTE 0x10 //Verstärker eingeschaltet, Mute on
#define POWER_ON_IN1 0x14 //Verstärker an, Eingang 1 gewählt
#define POWER_ON_IN2 0x18 //Verstärker an, Eingang 2 gewählt
#define POWER_ON_IN1_IN2 0x1C //Verstärker an, Eingang 1 und
                           // 2 gewählt
```

Mit dem Aufruf: LM48100\_Set\_Power(AMPLIFIER\_BOTH, POWER\_ON\_IN1, POWER\_ON\_IN1); werden die zwei Verstärker eingeschaltet und entsprechend der Beschaltung aus der Abb. 23.6 der Eingang IN1 für beide als Signaleingang konfiguriert.

Die Lautstärke kann für jeden Verstärker getrennt eingestellt werden. Beim Aufruf der Funktion LM48100\_Set\_Loudness werden als Parameter die Adresse des Bausteins sowie die Verstärkungsfaktoren für jeden Signaleingang eingegeben. Die Verstärkungsfaktoren müssen im Bereich 0...31 liegen, ansonsten werden sie in der Funktion nach der Prüfung auf 0 gesetzt. Diese Prüfung soll eine falsche Einstellung des Verstärkers verhindern, da die Bits 5, 6 und 7 die interne Adresse des Verstärkers bestimmen.

```
uint8_t LM48100_Set_Loudness(uint8_t ucdevice_address, uint8_t
ucgain1, uint8_t ucgain2)
{
    uint8_t ucDeviceAddress, ucVolumeCtrlReg1, ucVolumeCtrlReg2;
    //Adresse des LM48100-Audioverstärkers bilden
    // (es können bis zu 2 Audioverstärker an einem I2C-Bus angeschlossen
    // sein)
    ucDeviceAddress = (ucdevice_address<<1) | LM48100_DEVICE_TYPE_
ADDRESS;
    ucDeviceAddress |= TWI_WRITE; //Write-Modus
    if(ucgain1>31) ucgain1=0;
    if(ucgain2>31) ucgain2=0;
    ucVolumeCtrlReg1=VOLUME_REG_1_ADDR | ucgain1; // VOLUME_REG_1_
ADDR=0x60
    ucVolumeCtrlReg2=VOLUME_REG_2_ADDR | ucgain2; // VOLUME_REG_2_
ADDR=0x80

    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 Verstärkungsfaktor für den Eingang IN1 wird übertragen
TWI_Master_Transmit(ucVolumeCtrlReg1);
if((TWI_STATUS_REGISTER) !=TWI_MT_DATA_ACK) return TWI_ERROR;

//der Verstärkungsfaktor für den Eingang IN2 wird übertragen
TWI_Master_Transmit(ucVolumeCtrlReg2);
if((TWI_STATUS_REGISTER) !=TWI_MT_DATA_ACK) return TWI_ERROR;

// Stop
TWI_Master_Stop();
return TWI_OK;
}
```

---

## Literatur

1. Werner, M. (2006). *Nachrichten-Übertragungstechnik*. Vieweg.
2. Silicon Labs. SI4840/44 – Broadcast Analog Tuning Digital Display AM/SW/FM Radio Receiver. (2015). [www.silabs.com](http://www.silabs.com). Zugegriffen: 27. Febr. 2021.
3. Silicon Labs. AN610 – SI48XX ATDD Programming Guide V3.0/2013. (2015). [www.silabs.com](http://www.silabs.com). Zugegriffen: 27. Febr. 2021.
4. Silicon Labs. AN602 – SI4822/26/27/40/44 Antenna, Schematic, Layout and Guidelines. (2015). [www.silabs.com](http://www.silabs.com). Zugegriffen: 27. Febr. 2021.
5. Texas Instruments LM48100Q Boomer™ 1.3-W Audio Power Amplifier. <https://www.ti.com/product/LM48100Q-Q1>. Zugegriffen: 27 Febr. 2021.



## Zusammenfassung

In diesem Kapitel werden einige vernetzbare ICs vorgestellt: Ein Port-Expander, ein digitaler Regelwiderstand und eine Echtzeituhr.

In diesem Kapitel wird eine lose Sammlung weiter vernetzbarer Bausteine beschrieben, die in unterschiedlichen Kontexten praktisch sind. Port Expander dienen dazu, die Zahl der digitalen Ein- und Ausgänge eines Mikrocontrollers erheblich zu steigern, im Fall des hier vorgestellten PCF8574 auf bis zu 128. Einstellbare Widerstände koppeln die analoge von der digitalen Welt ab und eine Echtzeituhr findet sich in Netzwerken mit intelligentem Powermanagement wieder, in denen im Ruhezustand nichts als die Uhr laufen muss.

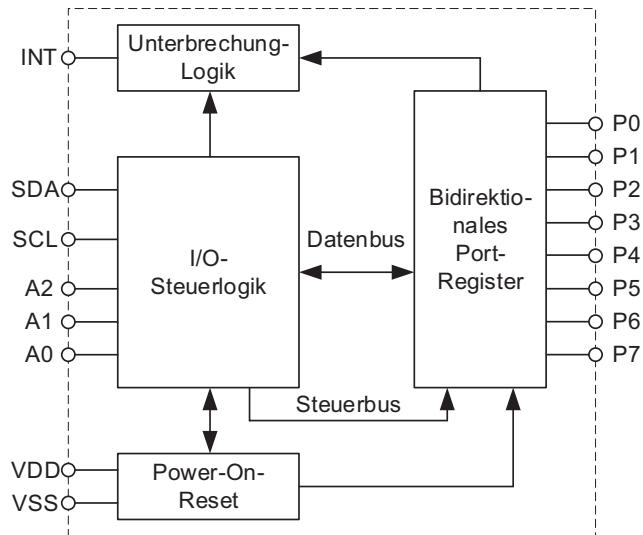
## 24.1 PCF8574 – Port-Expander

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 [3]

---

Die Originalversion dieses Kapitels wurde revidiert. Ein Erratum ist verfügbar unter  
[https://doi.org/10.1007/978-3-658-31709-6\\_27](https://doi.org/10.1007/978-3-658-31709-6_27)

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann  
[https://doi.org/10.1007/978-3-658-31709-6\\_24](https://doi.org/10.1007/978-3-658-31709-6_24).



**Abb. 24.1** PCF8574 – Blockschaltbild

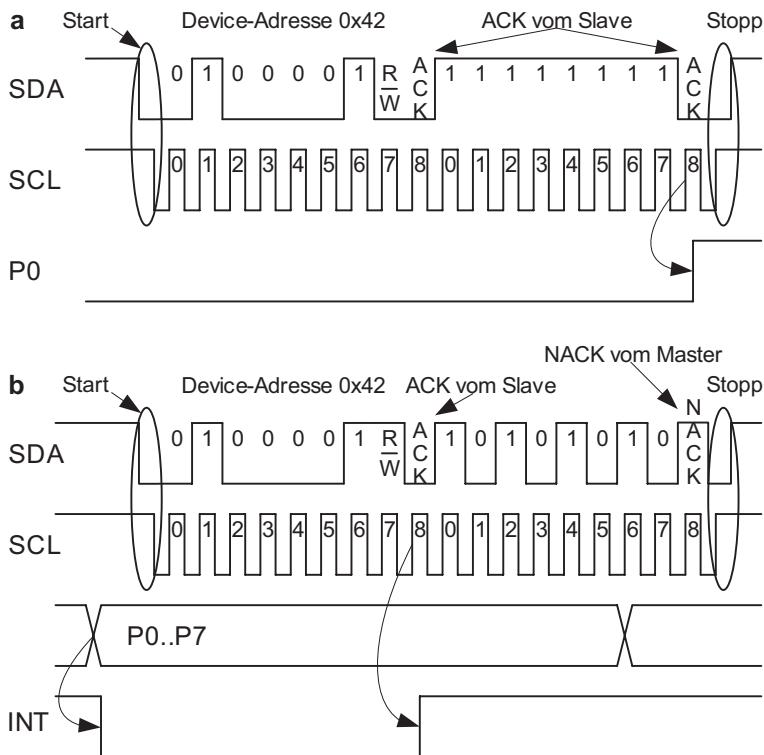
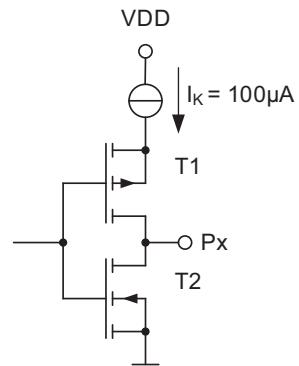
ist ein solcher Port-Expander-Baustein, der die Peripherie eines Mikrocontrollers um einen 8-Bit-Port erweitern kann. Er benötigt für die Kommunikation lediglich die zwei Leitungen eines I<sup>2</sup>C-Busses. Der Baustein wird in zwei Ausführungen hergestellt, die bis auf die Device-Typ-Adresse: 0x40 für PCF8574 und 0x70 für PCF8574A identisch sind. Dank der drei Adressanschlüsse können jeweils bis zu acht 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. 24.1) 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. Die Kommunikation mit dem Baustein erfolgt über I<sup>2</sup>C im Standard-Modus mit bis zu 100 kBit/s.

### 24.1.1 Endstufe eines I/O-Pins

Wenn ein I/O-Pin auf Low geschaltet ist, kann der Transistor T2 in Abb. 24.2 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:

- der minimale Ausgangsstrom im High Zustand muss ausreichend für die Ansteuerung eines Aktors sein;

**Abb. 24.2** PCF8574 –  
Endstufe eines I/O-Pins



**Abb. 24.3** PCF8574 – Ein-/Ausgabe eines Bytes

- 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.

### 24.1.2 Ausgang-Port-Modus

In der Abb. 24.3a 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 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;
    // Stopp
    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);
```

### 24.1.3 Eingang-Port-Modus

In der Abb. 24.3b 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;
    // Stopp
    TWI_Master_Stop();
    return TWI_OK;
}
```

Die Bitdauer bei maximaler Taktfrequenz beträgt:

$$t_{\text{Bit}} = \frac{1}{f_{\max}} = \frac{1}{100.000} = 10 \mu\text{s}$$

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.

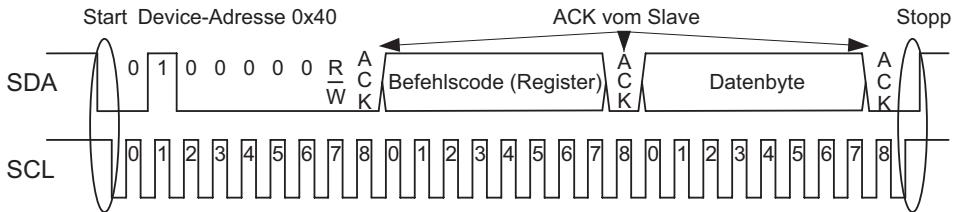
#### 24.1.4 Interrupt-Modus

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. 24.3b 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 Addressbyte+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.

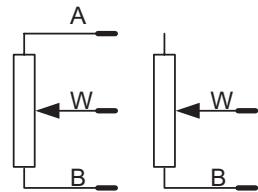
#### 24.1.5 PCA9534

Der PCA9534 [4] 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 I<sup>2</sup>C-Fast-Modus und kann Daten mit bis zu 400 kBit/s



**Abb. 24.4** PCA9534 – Register schreiben

**Abb. 24.5** Regelbare  
Widerstände: links  
Potentiometer, rechts Rheostat



senden und empfangen. In diesem Fall rechnet man mit einer Dauer von ca. 22,5 µs für die Übertragung eines Bytes. 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 Invertierung beim Einlesen des entsprechenden Eingangs. Der Inhalt dieser drei Register kann mit einem I<sup>2</sup>C-Schreibbefehl wie in der Abb. 24.4 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 angewählt wurde, verläuft das Einlesen von Daten genau wie beim PCF8574.

## 24.2 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 Rheostat<sup>1</sup> hergestellt (siehe Abb. 24.5). Der elektrische Widerstand zwischen dem Ende B und dem Schleifer kann nur

<sup>1</sup>Aus dem Griechischen wörtlich übersetzt etwa Flussversteller: Ein einstellbarer Widerstand.

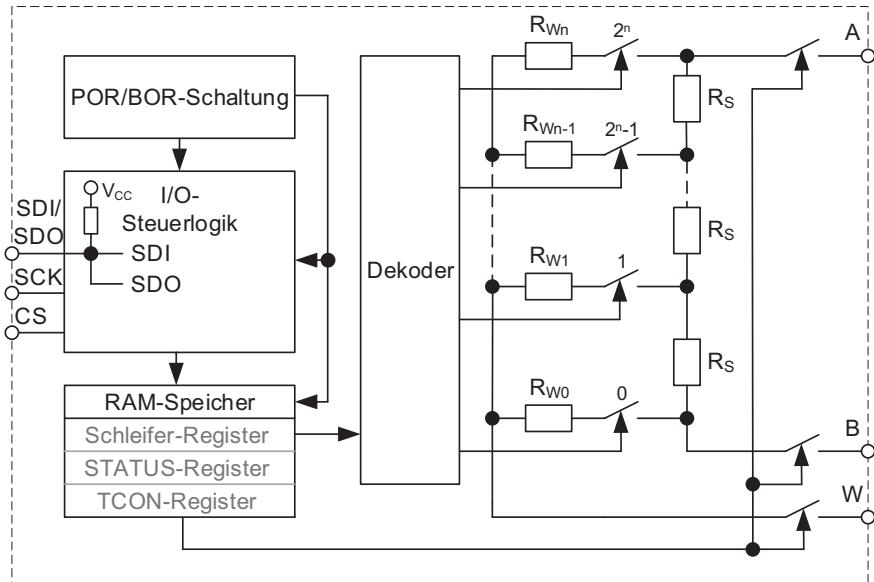


Abb. 24.6 MCP4151 – Blockschaltbild

stufenweise geändert werden. Die Ansteuerung der Bausteine kann über I<sup>2</sup>C 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 [1] ist eine Reihe von einzelnen digitalen Potentiometern, die eine 7/8-Bitauflösung haben und über SPI angesteuert werden. Sie können mit Spannungen zwischen 2,7 V und 5,5 V versorgt werden, während die anderen Anschlüsse Spannungspegel 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 Filtern 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 kΩ- und von 100 kHz bei den 100 kΩ-Potentiometern auf. Ein Blockschaltbild eines MCP4151-Bausteins ist in Abb. 24.6 dargestellt.

### 24.2.1 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.

### 24.2.2 Elektrischer Widerstand

Der elektrische Widerstand des Bausteins MCP4151 besteht aus einer Reihenschaltung von  $2^n$  gleich großen Teilwiderständen  $R_S$ , wobei in diesem Fall  $n=8$ .

$$R_S = \frac{R_{AB}}{256} \quad (24.1)$$

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 FET<sup>2</sup>-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.

### 24.2.3 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 drei 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. 24.1 zu entnehmen.

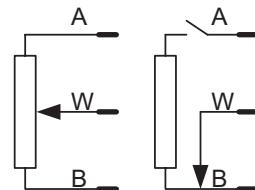
---

<sup>2</sup>FET – Field Effect Transistor.

**Tab. 24.1** MCP4151 – Register

Registername	Registeradresse	Standardwert
Schleifer-Register	0x00	0x80
TCON-Register	0x04	0x1FF
Status-Register	0x05	0xE0

**Abb. 24.7** MCP41X1 im aktiven Modus (a) und im Shutdown-Modus (b)



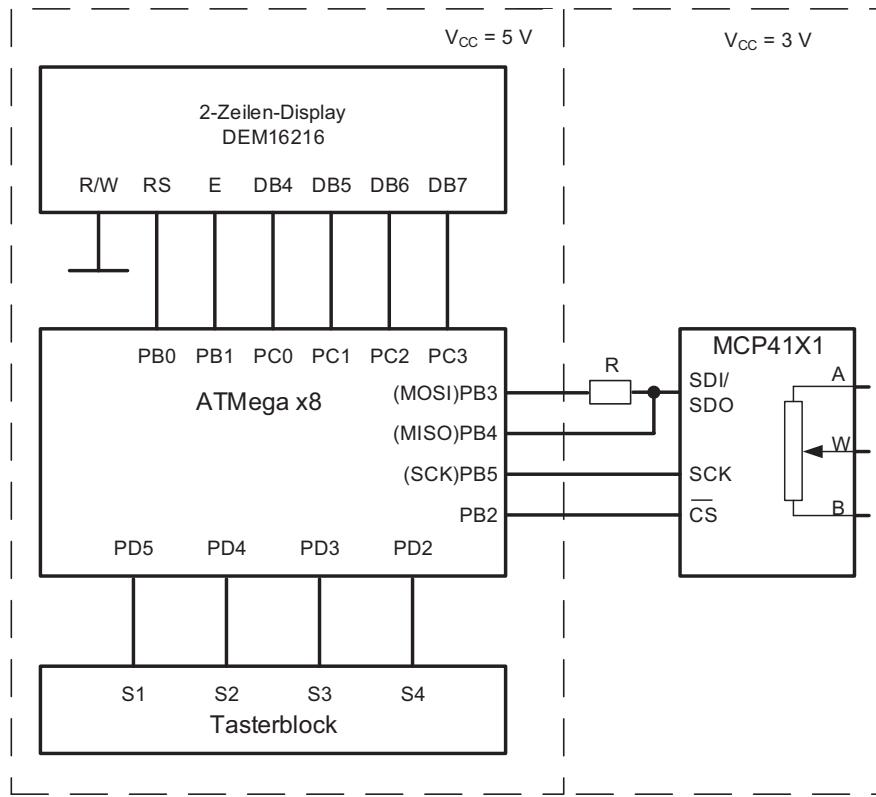
Das **Schleifer**-Register speichert und steuert über einen Dekoder die Schleiferstellung. Nach dem Einschalten wird dieses Register mit dem Wert 0x80 (dezimal 128) bei den 8-Bit-, bzw. mit 0x40 (64) bei den 7-Bit-Potentiometern initialisiert, was der Mitte des Gesamtwiderstandes entspricht. Wenn das Register mit 0x00 geladen wird, so wird der 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. 24.7).
- **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).



**Abb. 24.8** MCP41X1 – Anschluss an einem Mikrocontroller

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 SDI<sup>3</sup> und SDO<sup>4</sup> eine einzige externe Leitung, was einem „wired and“ entspricht. Damit diese Beschaltung problemlos funktioniert, muss der SDI/SDO Anschluss mit dem MOSI- Anschluss des Mikrocontrollers über einen Widerstand verbunden werden (siehe Abb. 24.8), damit beim Datenauslesen nicht zwei Ausgänge zusammen geschaltet werden. Der interne Pull-up Widerstand wird bei der Datenausgabe vom SDO-Signal aktiviert und ermöglicht die

<sup>3</sup> SDI – Slave Data In.

<sup>4</sup> SDO – Slave Data Out.

**Tab. 24.2** 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	

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).

#### 24.2.4 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. 24.2 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. 24.1);
- **Bit 3:2** – die Bits codieren den Befehlscode (Tab. 24.2);
- **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 sechs Bits des Command-Bytes auf die der Baustein reagiert sind in der Tab. 24.3 aufgelistet.

#### 24.2.5 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. 24.8 dargestellte Beschaltung lautet die SPI-Datenstruktur des Bausteins (s. Kap. 14 Abschn. 5):

**Tab. 24.3** Erlaubte Kombinationen der ersten sechs Bits des Command-Bytes

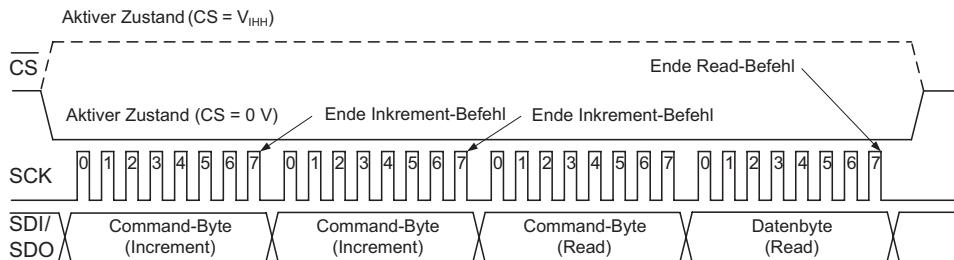
Adresse	Befehlscode/ Funktion	Aktion
0000	00/Write	Das Schleifer-Register wird mit den nachfolgenden Datenbits geladen; ein neues Widerstandsverhältnis an dem Schleifer wird bestimmt
	11/Read	Der Inhalt des Schleifer-Registers wird ausgelesen
	01/Inkrement	Wenn der Inhalt des Schleifers-Registers kleiner 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

```
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. 24.9).

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. 24.3 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:



**Abb. 24.9** MCP41X1 – Verketteten Funktionen

- 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 verketten (Abb. 24.9). In diesem Beispiel werden die Funktion *Increment* zweimal hintereinander und die Funktion *Read* einmal ausgeführt.

## 24.2.6 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;
    //Gestaltung des Command-Bytes
    ucCommandByte = ucreg_address | READ_DATA_OPCODE;
    //die SPI-Übertragung wird gestartet
    SPI_Master_Start(sdevice_pins.MCP41X1spi);
    //das Command-Byte wird übertragen, das neunte 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
    Übertragungsfehlers
    ucDataIn = SPI_Master_Write(0xFF);
    //der Inhalt des eingelesenen Registers wird zusammengefasst
    uiDataIn |= ucDataIn;
    //die SPI-Übertragung wird beendet
    SPI_Master_Stop(sdevice_pins.MCP41X1spi);
    return uiDataIn;
}
uint8_t MCP41X1_Increment_Reg(MCP41X1_pins sdevice_pins,
                               uint8_t ucreg_address)
{
    uint8_t ucCommandByte, ucDataIn;
    //Gestaltung des Command-Bytes
    ucCommandByte = ucreg_address | INCREMENT_OPCODE;
    //die SPI-Übertragung wird gestartet
    SPI_Master_Start(sdevice_pins.MCP41X1spi);
    //die Rückmeldung wird eingelesen
    ucDataIn = SPI_Master_Write(ucCommandByte);
    //die SPI-Übertragung wird beendet
    SPI_Master_Stop(sdevice_pins.MCP41X1spi);
    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.

## 24.3 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 32768 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 [2] betrachtet, der die Funktionen einer Echtzeituhr mit Kalender bis zum Jahr 2100 implementiert. Zusätzlich besitzt er einen Temperatursensor (siehe Abb. 24.10), mit dem die Temperatur von  $-55^{\circ}\text{C}$  bis  $+125^{\circ}\text{C}$  mit einer Genauigkeit von  $\pm 2^{\circ}\text{C}$  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 Mikrocontroller<sup>5</sup> dienen kann. Der Frequenzteiler wird über das Konfiguration-Register eingestellt, wie in Tab. 24.4 dargestellt ist.

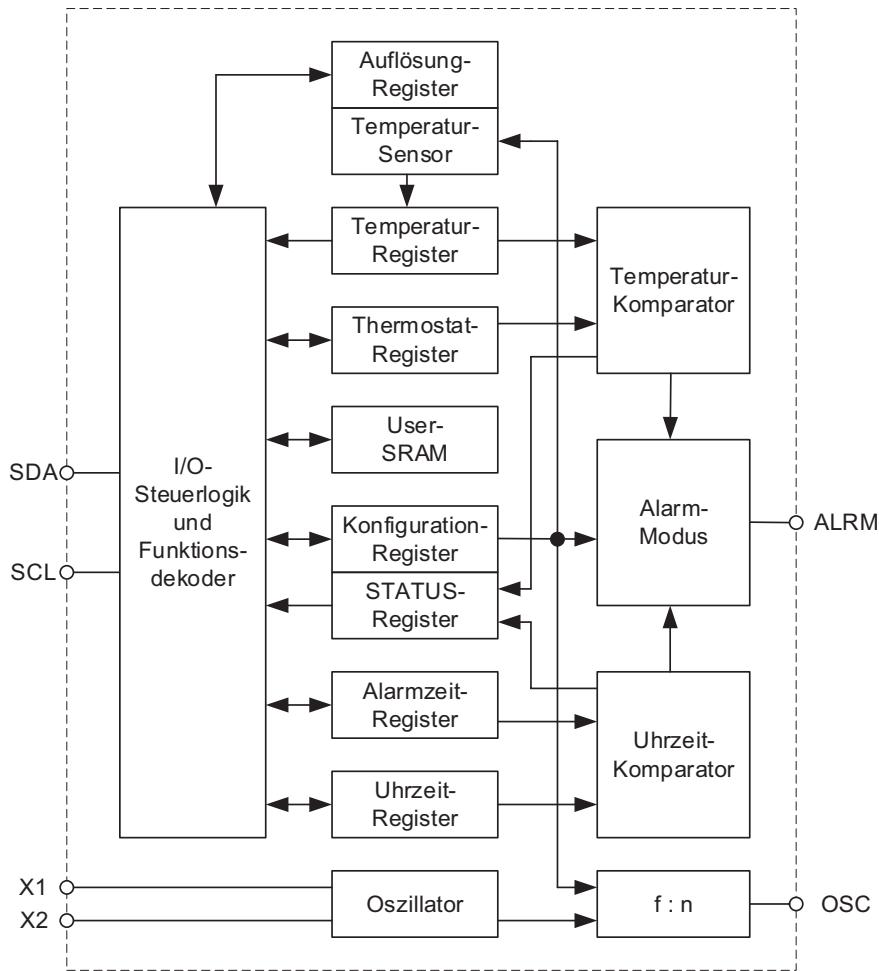
Wenn der Taktausgang nicht gebraucht wird, soll er, um Strom zu sparen, abgeschaltet werden.

### 24.3.1 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. 24.5) 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 gezählt werden (entspricht den Jahren 2000 bis 2099) und werden in BCD<sup>6</sup>-kodierter

<sup>5</sup> Beispielsweise für den erniedrigten Takt im Schlafmodus, siehe dazu Abschn. 3.3.2

<sup>6</sup> BCD – Binary Coded Decimal.

**Abb. 24.10** MAX31629 Blockschaltbild**Tab. 24.4** Frequenzteiler  
Clock

Konfiguration-Register		Clock-Ausgang
Bit 7	Bit6	
0	0	Ausgeschaltet
0	1	:8
1	0	:4
1	1	:1

**Tab. 24.5** Speicherorganisation des MAX31629-Bausteins

Register-Bezeichnung	Register Elemente	Logische Adresse	Befehls-code	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-TH	–	2 Byte	0xA1	EEPROM	Schreiben-Lesen
Thermostat-TL	–	2 Byte	0xA2	EEPROM	Schreiben-Lesen
Konfiguration/STATUS-Register	–	1Byte	0xAC	EEPROM	Schreiben-Lesen
	–	1Byte		SRAM	Nur Lesen
User SRAM	–	32 Bytes	0x17	SRAM	Schreiben-Lesen

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-Decimal- 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;
/*die dezimale, binär kodierte Zahl wird mit dem höherwertigen Nibble
ergänzt*/
ucBCDNumber = ucBCDNumber + (ucDecimalnumber << 4);
```

**Tab. 24.6** MAX31629 –  
Alarm-Modi

Konfiguration-Register		Alarm-Modus
Bit5	Bit4	
0	0	1 – deaktiviert
0	1	2 – nur Temperatur
1	0	3 – nur Zeit
1	1	4 – beide

```
//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/PM<sup>7</sup>) 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.

### 24.3.2 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. 24.6). 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

---

<sup>7</sup>AM – ante meridiem (Vormittag), PM – post meridiem (Nachmittag).

**Tab. 24.7** MAX31629 – Temperaturaufösung 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

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.

### 24.3.3 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. 24.7 aufgelistet ist.

Die Erhöhung der Bitauflösung um eins 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 2-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 Nachkommanteil. 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. 24.8).

**Tab. 24.8** 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

Im Freilauf-Modus steht im Temperaturregister der letzte gemessene Temperaturwert zum Lesen bereit, ein Lese-Zugriff auf dieses Register beeinflusst eine laufende Messung nicht. Mit einer Stopp-Messungs-Funktion (Befehlskode 0x22) wird der Freilauf-Modus vorübergehend gestoppt, nachdem die aktuelle Umwandlung vollständig beendet ist und der Messwert wird gespeichert. Mit einer Start-Messungs-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;
}
```

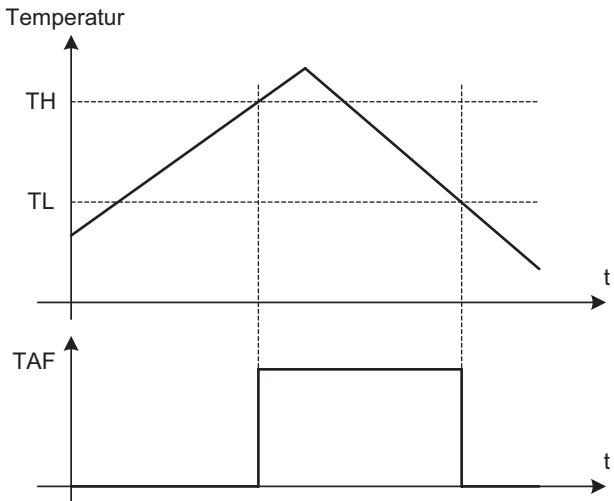
#### 24.3.4 Thermostat mit Alarmfunktion

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. 24.10) um und das Bit 6 (TAF – Thermal Alarm Flag) im Status-Register wird auf „1“ gesetzt. Falls der Alarrrmodus 1

Temperatur-Register																	
MSB								LSB									
$\pm$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	,	$2^{-1}$	$2^{-2}$	$2^{-3}$	$2^{-4}$	$2^{-5}$	$2^{-6}$	$2^{-7}$	$2^{-8}$	°C

**Abb. 24.11** MAX31629 – Temperatur-Kodierung

**Abb. 24.12** MAX 31629 – Temperaturalarm



oder 3 (Tab. 24.6) gewählt ist, wird der Alarmausgang aktiv (Abb. 24.12). 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 Temperaturnachricht erreicht oder überschritten hat. Das TAF-Bit wird zurückgesetzt, sobald die gemessene Temperatur die untere Temperaturnachricht erreicht oder unterschreitet. Die Temperaturauflösung des Thermometers und des Thermostats müssen nicht gleich sein.

Die Temperaturnachrichtswerte müssen vor der Übertragung entsprechend Abb. 24.11 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)
```

```

{
    //die Variable ucSign speichert das Vorzeichen des Temperaturwertes
    ucSign = 1;
    //der Betrag des negativen Wertes wird berechnet
    ftemp = ftemp * (float)(-1.);
}
//ucInteger speichert die Ganzzahl des Temperaturwertes
ucInteger = ftemp;
ftemp = ftemp - ucInteger; //die Nachkommastelle wird berechnet
ucDecimalPlace = ftemp * 16; //Abrundung auf die 12-Bit-Auflösung
//das Low-Byte des Temperaturwerts wird erstellt
ucDecimalPlace = ucDecimalPlace << 4;
//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;
}

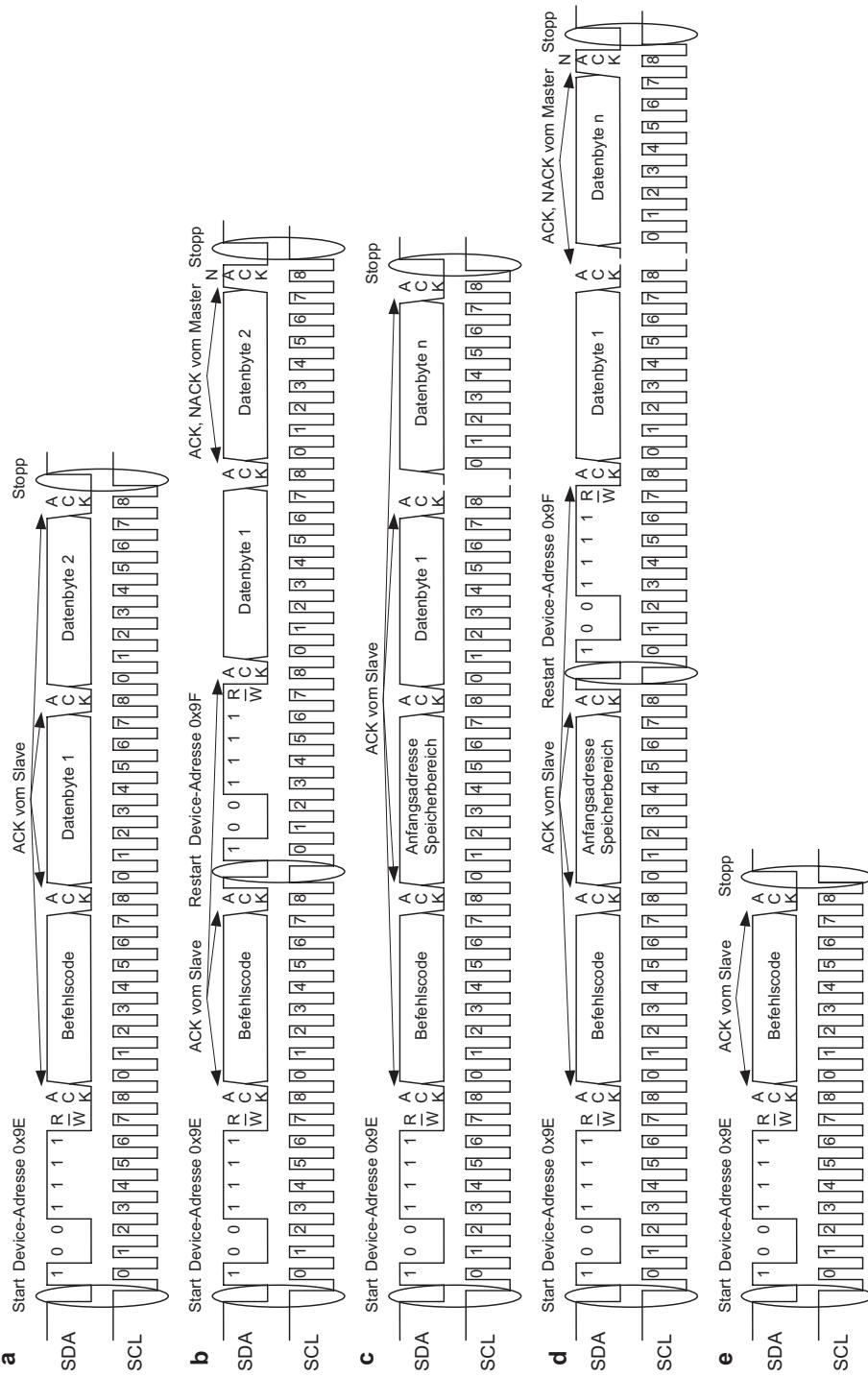
```

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.

### 24.3.5 I<sup>2</sup>C-Kommunikation

Die Kommunikation zwischen einem Mikrocontroller, der als Master konfiguriert ist und dem Baustein MAX31629 als Slave erfolgt über I<sup>2</sup>C bei einer Bitrate von max. 400 kBit/s. Die Device-Typ-Adresse des Bausteins ist *0x9E*. Die einzelnen Speicherbereiche des Bausteins (Tab. 24.5) können über Befehlscodes adressiert und gelesen, bzw. geändert werden. Unterschiedliche Zugriffs-Vorgänge werden in Abb. 24.13 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



**Abb. 24.13** MAX31629 – I<sup>2</sup>C-Kommunikation

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 ersten Datenbytes beendet.

- b) 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“. 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 einen 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).

---

## Literatur

1. Microchip Technology Inc. MCP413X/415X/423X/425X – digitale Potentiometer mit flüchtigen Speicher. (2008). [www.microchip.com](http://www.microchip.com). Zugegriffen: 4. Apr. 2021.
2. Maxim Integrated. MAX31629 – I2C Digital Thermometer and Real-Time-Clock. (2015). [www.maximintegrated.com](http://www.maximintegrated.com). Zugegriffen: 4. Apr. 2021.

3. NXP Semiconductors. PCF8574 – Remote 8-bit I/O expander for I<sup>2</sup>C-bus. (2013). [www.nxp.com](http://www.nxp.com). Zugegriffen: 4. Apr. 2021.
4. NXP Semiconductors. PCA9534 – 8-bit I<sup>2</sup>C-bus and SMBus low power I/O port with interrupt. (2017). [www.nxp.com](http://www.nxp.com). Zugegriffen: 4. Apr. 2021.



# Anzeigen

25

## Zusammenfassung

Das Kapitel gibt 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.

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.

---

Die Originalversion dieses Kapitels wurde revidiert. Ein Erratum ist verfügbar unter  
[https://doi.org/10.1007/978-3-658-31709-6\\_27](https://doi.org/10.1007/978-3-658-31709-6_27)

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann  
[https://doi.org/10.1007/978-3-658-31709-6\\_25](https://doi.org/10.1007/978-3-658-31709-6_25).

## 25.1 Einführung

### 25.1.1 Displaylayout

Alphanumerische Anzeigen sind entwickelt worden, um m Zeichen mithilfe 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 farbfähige Anzeige. Mit der Ansteuerung im Multiplexbetrieb kann die Helligkeit jedes der  $3 \times mxn$  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.

### 25.1.2 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 LC<sup>1</sup>-Anzeigen gehören zu den nicht emissiven Anzeigen und können als passive oder aktive Matrix gebaut werden.

#### 25.1.2.1 Flüssigkristallanzeigen (LCD)

Kristalle besitzen die Eigenschaft der Anisotropie, das heißt, bestimmte Eigenchaften 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

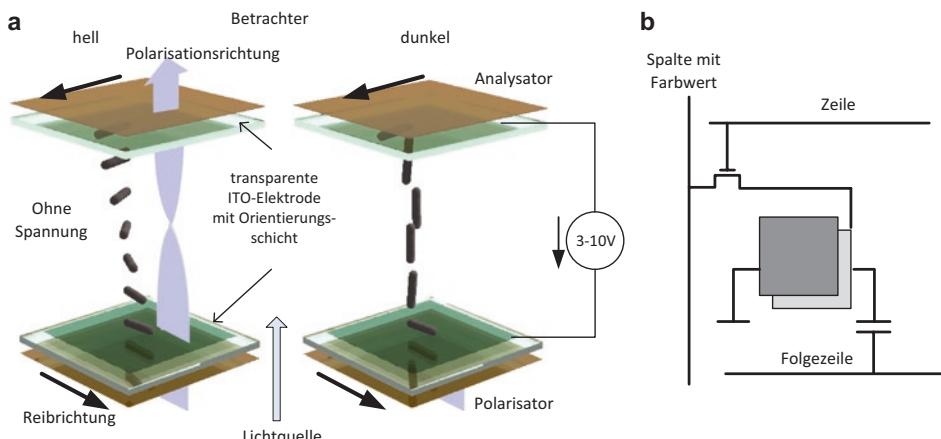
---

<sup>1</sup>LCD – Liquid Cristal Display (deutsch Flüssigkristallanzeige).

zwei Strahlen polarisierten Lichts aus. Die Überlagerung ergibt eine gegenüber der Einfallspolarisation geänderte Polarisation. 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. 25.1a schematisch dargestellt ist. Durch entsprechende Oberflächenbehandlung (Reiben), wird dafür gesorgt, dass die Vorzugsrichtungen (Direktor) der Moleküle direkt auf den beiden Glasplatten um  $90^\circ$  gegeneinander verdreht sind. Aufgrund der Wechselwirkungen zwischen den Molekülen drehen sich die dazwischenliegenden Schichten schraubenförmig auf und drehen dabei die Polarisationsrichtung von durchtretendem Licht um  $90^\circ$ . 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



**Abb. 25.1** Prinzip einer TN-Zelle (**a** hell, **b** dunkel), daneben das Schaltprinzip einer aktiven TFT-Zelle

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. LC-Anzeigen mit IPS<sup>2</sup> – und VA<sup>3</sup> -Zellen verbessern den Kontrast bei erhöhtem Blickwinkel [5]. 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 [7]. 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.

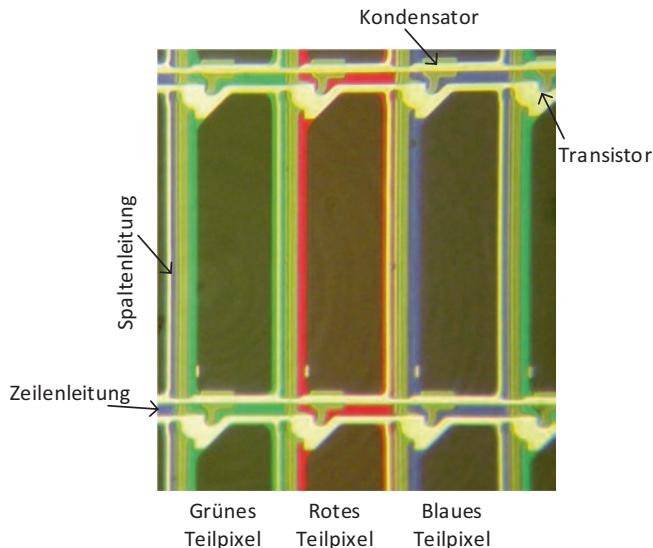
Aktivmatrix LC-Anzeigen beheben diesen Nachteil, indem sie für jedes Bildelement einen Kondensator für die Erhaltung des elektrischen Feldes nutzen. Dieser Kondensator kann über einen Dünnschicht-Transistor (englisch TFT – Thin Film Transistor) nachgeladen werden. Diese TFT-Technologie ermöglicht den Bau vollfarbiger Displays. Man spricht in dem Zusammenhang auch von Aktivmatrix-LC-Anzeigen (AMLCD). In Abb. 25.1b ist das Prinzipschaltbild dargestellt, in Abb. 25.2 ist eine Mikroskop-Aufnahme gezeigt, die der Autor mit einem klassischen Lichtmikroskop im Drauflicht mit einem TFT-Monitor erstellt hat.

Alle LC-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 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

---

<sup>2</sup>IPS – In plane switching.

<sup>3</sup>VA – vertical alignment.



**Abb. 25.2** Mikroskopische Aufnahme eines Pixels aus einem TFT-Monitor

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.

### 25.1.2.2 LED-Anzeigen

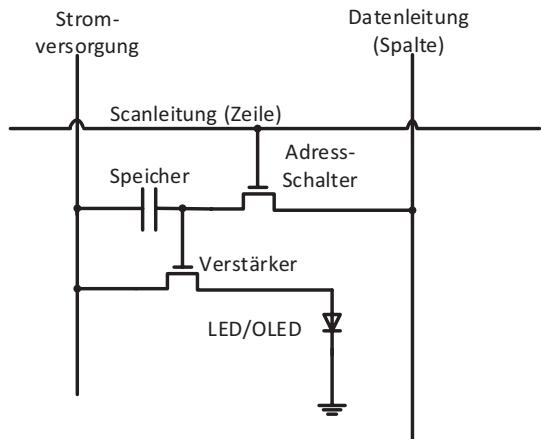
Als emissive Anzeigen werden hier nur die mit LED<sup>4</sup>s gebauten, Anzeigen erwähnt. Heutzutage können großflächige Displays aus anorganischen Leuchtdioden in Halbleiter-Technologie hergestellt werden.

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 werden. Sie können extrem dünn und biegsam sein, bieten einen weiten Blickwinkel, haben aber derzeit noch einen höheren Energieverbrauch als LC-Anzeigen. Aktive OLED-Matrixanzeigen haben einen etwas komplexeren Aufbau als aktive LC-Anzeigen, da neben der Bildinformation auch die Energie für die Erzeugung des Bildpunktes an jeden einzelnen Pixel herangeführt werden muss. In Abb. 25.3 ist eine solche

---

<sup>4</sup>LED – Light Emitting Diode (deutsch Leuchtdiode).

**Abb. 25.3** Ansteuerung eines Pixels in einer aktiv leuchtenden (LED- oder OLED-) Anzeige



Schaltung für einen einzelnen Bildpunkt schematisch dargestellt. Dafür ist jedoch keine Hinterleuchtung notwendig.

### 25.1.3 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 (Framebuffer) 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 ein Bild aufzubauen, werden bei embedded Displays oftmals elementare Rastergrafiken in einem nichtflüchtigen Speicher abgelegt und dann zeilenweise an die entsprechende Position im Framebuffer übertragen. Man nennt diese Technik auch Sprite-Technik und die Grafik-Bausteine Sprites (Kobold, Geistwesen).

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.

In diesem Fall müssen Rasterisierungsalgorithmen direkt in den Grafikcontroller eingebaut sein, die dann nach entsprechender Skalierung die jeweils betroffenen Pixel im Framebuffer ansteuern.

### 25.1.4 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 Vektorgraphik benutzen. Hochauflösende Displays werden nur noch zum Teil analog (RGB, VGA) angesteuert, die meisten über LVDS<sup>5</sup> basierte Bussysteme wie DisplayPort oder TMDS<sup>6</sup> 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 Rechteckmanagement. 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.

---

## 25.2 Punktmatrix-LCD-Display mit paralleler Ansteuerung

In Geräten mit Mikrocontrollern der ATmega-Klasse werden Punktmatrix LC-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 STN<sup>7</sup>-Displays, also monochromatische LC-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.

### 25.2.1 Struktur eines Displays mit einem KS0070B-Controller

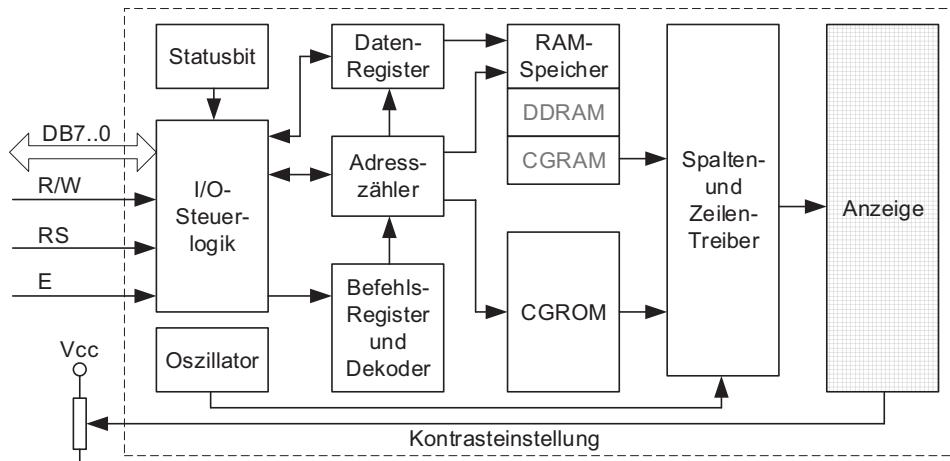
Ein grobes Blockschaltbild eines Displays, das mit einem Controller KS0070B, der mit dem HD44780 kompatibel ist und von der Fa. Samsung [1] hergestellt wird, ist in Abb. 25.4 dargestellt. Der Datenaustausch zwischen dem steuernden Mikrocontroller und dem Display-Controller findet über einem 8-Bit-Datenbus (DB7...DB0) oder

---

<sup>5</sup>Low Voltage Differential Signaling.

<sup>6</sup>Transition Minimized Differential Signaling.

<sup>7</sup>STN – Super-Twisted Nematic.



**Abb. 25.4** LC-Display mit Controller KS0070B – Blockschaltbild

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 Kauf genommen. Der Controller besitzt neben den Datenpins auch drei 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 Datenpuffer 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 5x7-Punkte und 32 5x10-Punkte großen Zeichen besteht, ist im CGROM<sup>8</sup> hinterlegt. Dieser Zeichensatz kann dem Datenblatt entnommen werden. Der Anwender kann selbst 5x8-Punkte große Zeichen definieren und bis zu acht davon gleichzeitig nutzen.

Der RAM-Speicher ist in zwei Bereiche unterteilt:

- **DDRAM**<sup>9</sup> ist ein 80 Byte großer Speicher, der die Display-Positionen abbildet. Die entsprechenden DDRAM-Adressen im hexadezimalen Zahlenformat eines 4zeiligen

<sup>8</sup> CGROM – Character Generator ROM.

<sup>9</sup> DDRAM – Display Data RAM.

**Tab. 25.1** 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

Displays wie zum Beispiel DEM 16.481 [2] von der Fa. Display Elektronik sind in Tab. 25.1 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
```

führt zum Setzen des Cursors auf die fünfte Position der ersten Zeile und zur Anzeige des Zeichens A an dieser Stelle. Beim Schreiben oder Lesen eines Datenbytes wird der Adresszähler entsprechend der Cursoreinstellungen inkrementiert oder dekrementiert.

- CGRAM<sup>10</sup> ist ein 64 Byte großer RAM-Speicher, in dem die Bitmuster von bis zu acht neu definierten Zeichen in codierter Form gespeichert werden können. Im Abschn. 25.2.4 wird die Generierung eines neuen Zeichens beschrieben.

Der Kontrast eines solchen Displays wird über eine analoge Spannung eingestellt.

### 25.2.2 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. 25.2.

- **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;

---

<sup>10</sup>CGRAM – Character Generator RAM.

**Tab. 25.2** 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

- **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. 25.3 zu entnehmen.
- **Display/Cursor an/aus**, wenn **D=1** wird das Display eingeschaltet, bei **C=1** wird der Cursor eingeblendet und **B=1** führt zum Blinken des Cursors;
- **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. 25.4) 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

**Tab. 25.3** Auswirkung der I/D- und SH-Bits

I/D	SH	Beschreibung
0	0	Cursor wird nach links geschoben, der Adresszähler dekrementiert, keine Displayverschiebung
0	1	Cursor wird nach links geschoben, der Adresszähler dekrementiert, Displayverschiebung nach rechts
1	0	Cursor wird nach rechts geschoben, der Adresszähler inkrementiert, keine Displayschiebung
1	1	Cursor wird nach rechts geschoben, der Adresszähler inkrementiert, Displayverschiebung nach links

**Tab. 25.4** 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

Display als einzeiliges ( $N=0$ ) oder zweizeiliges ( $N=1$ ) Display initialisiert wird und ob 5x7-Punkte ( $F=0$ ) oder 5x10-Punkte ( $F=1$ ) große Zeichen verwendet werden.

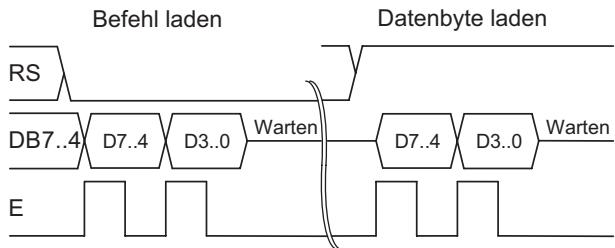
- **CGRAM-Adresse setzen** – Mit diesem Befehl wird der Cursor auf die gewünschte Adresse im CGRAM gesetzt;
- **DDRAM-Adresse setzen** – Mit dem Befehl wird die Display-Position festgelegt, an der das nächste Zeichen angezeigt werden soll.
- **Statusbit/Adresse lesen** – Beim Ausführen dieses Befehls 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).

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.

### 25.2.3 4-Bit-Kommunikation

Um I/O-Pins des Mikrocontrollers zu sparen, kann die Kommunikation mit dem Display-Controller auf vier 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

**Abb. 25.5** Display-Kommunikation über einen 4-Bit-Datenbus



D3...0, so wie in Abb. 25.5 dargestellt. Nach dem Hochfahren des Controllers wird der gesamte Inhalt des DDRAM-Speichers mit Leerzeichen gefüllt, die Kommunikation auf acht Bit gestellt, das Display wird abgeschaltet und der Cursor ausgebendet.

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. 25.6 dargestellt und hat folgende Einstellungen:

- zweizeiliges Display ( $N=1$ ),
- 5x7-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 ( $SH=0$ )

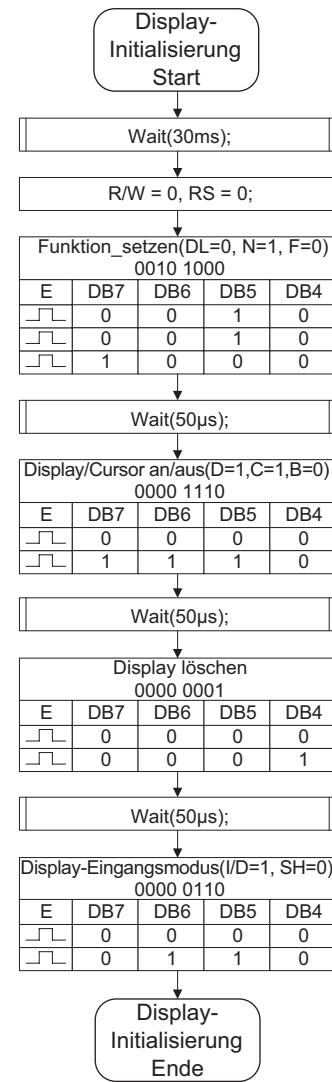
Jede übertragene Bytehälfte wird mit der fallenden Flanke des E-Signals in das Eingangsregister geladen.

#### 25.2.4 Generierung eines neuen Zeichens

Dem Anwender stehen die Zeichenadressen 0...7 zur Verfügung, um neue 5x8-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. 25.5 zu sehen.

Das Bitmuster eines zweiten Zeichens müsste man im CGRAM ab der Adresse 0x08 bis 0x0F speichern.

**Abb. 25.6** Display-Initialisierung im 4-Bit-Modus



Um den Programmcode zu vereinfachen, werden die für ein neues Zeichen berechneten CGRAM-Werte in ein Array gespeichert. Der Adresszähler des CGRAM-Speichers wird 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};

...
//0x00 = Adresse des 1. Datenbytes des Bitmusters im CGRAM
  
```

**Tab. 25.5** 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

```
Display_Set_CGRAMAddress(0x00);
for(uint8_t ucI = 0; ucI < 8; ucI++)
{
    Display_Write(ucOeZeichen[ucI]);
}
```

Nachdem die DDRAM-Adresse (Cursor-Position) im Hauptprogramm mit dem Aufruf:

```
//0x00 = die Zeichenadresse des neu definierten Zeichens
Display_Write(0x00);
```

gesetzt wurde, wird an der aktuellen Cursor-Position das Zeichen „ö“ dargestellt.

## 25.2.5 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:

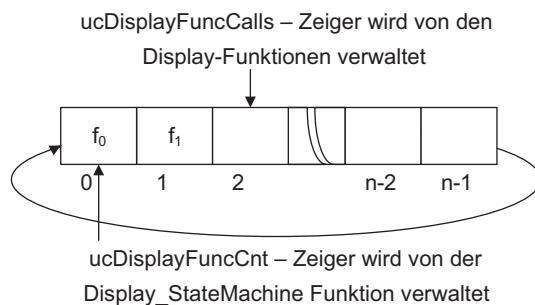
- das Ausgeben eines Bytes (Befehlscode oder Datenbyte) auf den 4-Bit-Datenbus. Das erfolgt in zwei Schritten gemäß der Abb. 25.5,
- das entsprechende Steuern der RS- und E-Eingänge des Displays und
- das Ausführen der weiteren Tasks des Hauptprogramms während das Display mit dem internen Ablauf beschäftigt 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. 25.7, auf den zwei Zeiger gerichtet sind, beide mit 0 initialisiert. Mit dem Aufruf einer Display-Funktion:

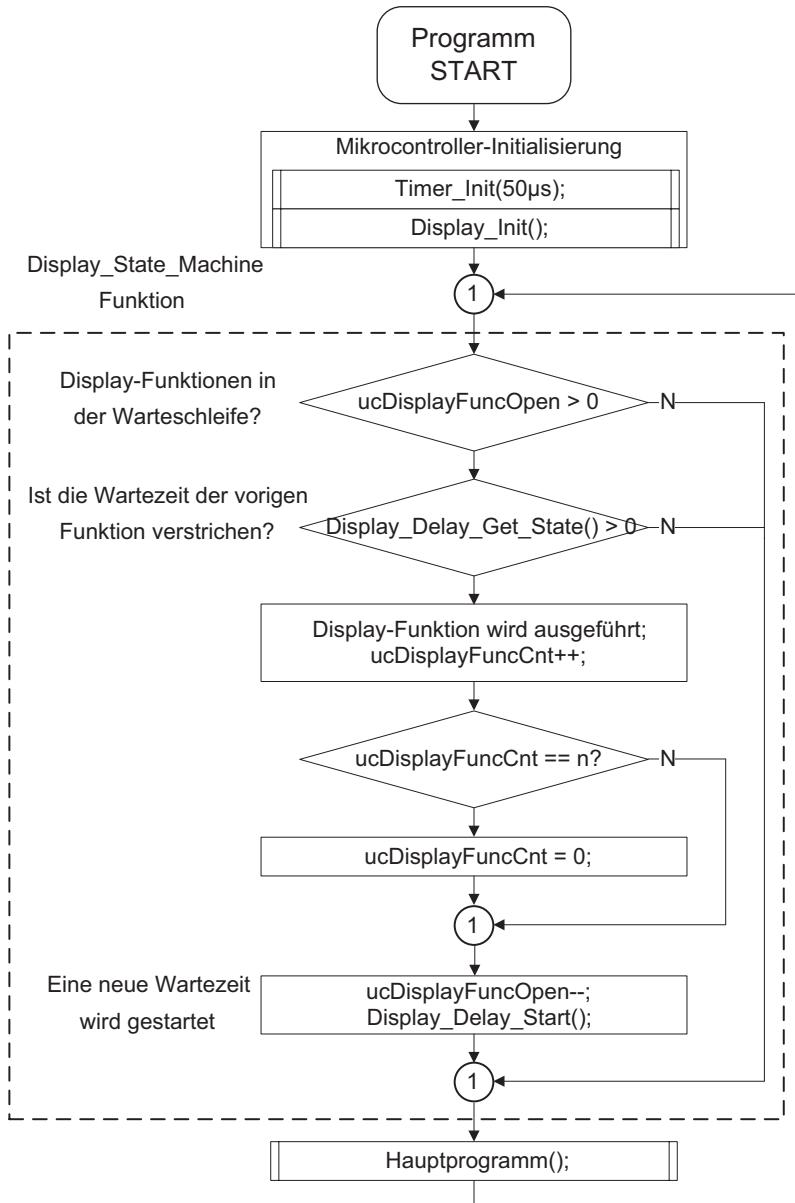
- wird das zu übertragende Byte (Befehlscode oder Datenbyte) in das Array `ucDisplayCommandByte` gespeichert zusammen mit einem zweiten Byte in das Array `ucDisplayDataByte` das den Zustand des RS-Eingangs (0/1) und die entsprechende Wartezeit (als Vielfache von 50 µs) codiert; der Zeiger `ucDisplayFuncCalls` 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. 25.8 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.



**Abb. 25.7** Displayfunktionen-Puffer

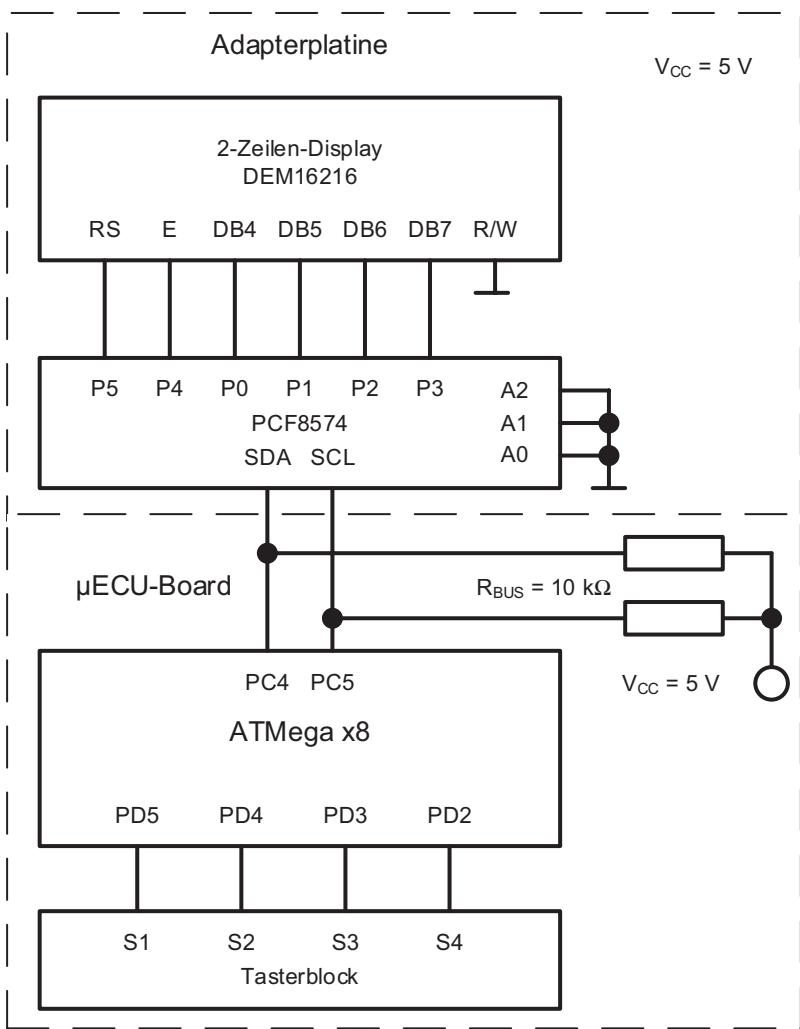


**Abb. 25.8** Zustandsautomat Displayfunktionen

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.

## 25.3 Serielle Ansteuerung eines parallelen LC-Displays

Ein LC-Display (siehe Abschn. 25.2), angeschlossen an einem im Kap. 24 Abschn. 1 beschriebenen Port-Expander-Baustein PCF8574, kann über I<sup>2</sup>C angesteuert werden, so wie in Abb. 25.9 dargestellt. Der Mikrocontroller benötigt für die Ansteuerung lediglich den I<sup>2</sup>C-Bus, an dem weitere Geräte oder Displays angeschlossen werden können.



**Abb. 25.9** Serielle Ansteuerung eines parallelen LC-Displays

### 25.3.1 Display-Ansteuerung über I<sup>2</sup>C

Der zeitliche Ablauf der Display-Ansteuerung und die Kommunikation mit dem Port-Expander wurden im Abschn. 25.2.3 beziehungsweise in Kap. 24 Abschn. 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. 25.6 dargestellt.

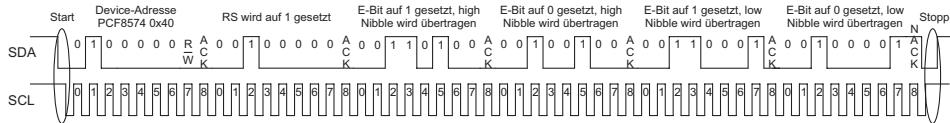
Der Baustein PCF8574 muss vor der Übertragung der fünf Bytes adressiert werden. Der gesamte Zeitverlauf der I<sup>2</sup>C-Kommunikation ist in der Abb. 25.10 dargestellt. Die Taktfrequenz des Busses beträgt in diesem Fall max. 100 kHz, was bedeutet, dass die Übertragung eines Bytes mindestens 90 µs 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 Zeichen anzuzeigen, werden in der gleichen I<sup>2</sup>C-Botschaft nur noch vier 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.

### 25.3.2 Software-Beispiel: Übertragung eines Datenbytes

Aus Portabilitätsgründen der Software werden die Verbindungen zwischen dem Display und dem Port-Expander wie folgt definiert:

**Tab. 25.6** 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 niedrigwertige Bytehälfte wird übertragen	0	0	1	1	0	0	0	1
Die niedrigwertige Bytehälfte wird in das Eingangsregister vom Display übernommen	0	0	1	0	0	0	0	1



**Abb. 25.10** I2C-Kommunikation – Anzeige des Zeichens ‚A‘ am Display

```
#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))
```

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 I<sup>2</sup>C-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. 25.6 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};
```

```
int8_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(); //Stopp
return TWI_OK;
}
```

## 25.4 DOGS102-6 – Graphisches Display mit serieller Ansteuerung

Die Displays der Reihe DOGS102-6 sind passive STN-Punktmatrix LC-Displays [3]. 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 [4] 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 65x132 Punkten. Eine Beispielschaltung ist in der Abb. 25.11 dargestellt.

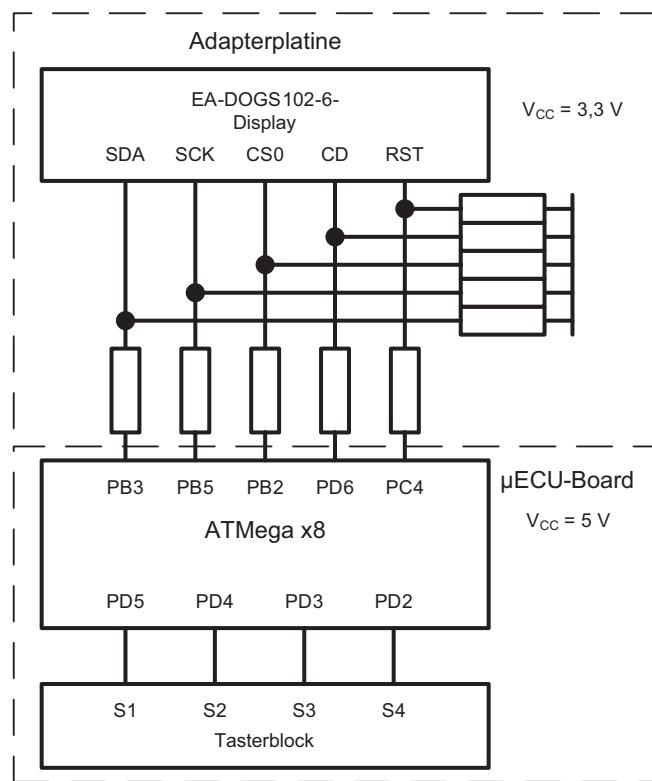
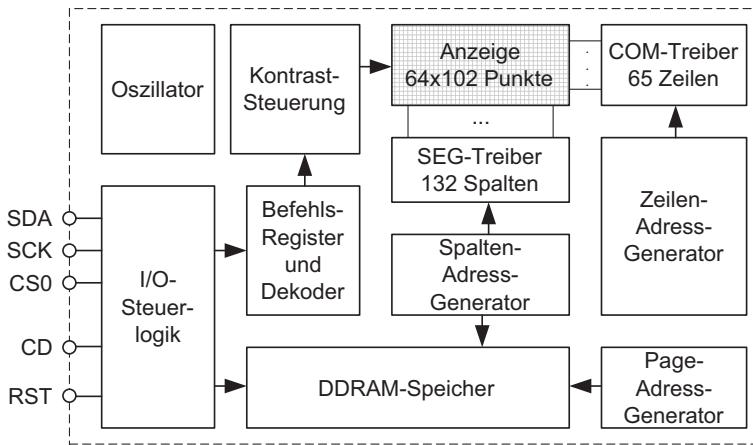


Abb. 25.11 DOGS-Display-Ansteuerung



**Abb. 25.12** Display – DOGS102-6 – Blockschaltbild

#### 25.4.1 Struktur des graphischen Displays DOGS 102-6

Ein grobes Blockschaltbild des Displays ist in der Abb. 25.12 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 Zeilenelektroden 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 CD<sup>11</sup> auf Low ist, in den Befehlsdecoder geladen und als Befehl dekodiert oder, wenn CD auf High ist in den Display-Data-RAM-Speicher gespeichert.

Der DDRAM-Speicher ist in acht Seiten mit jeweils 102 Bytes organisiert. Sein Inhalt wird vom Display abgebildet, so wie in der Tab. 25.7 dargestellt ist. Acht Displayzeilen bilden 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

<sup>11</sup>CD-Control Data.

**Tab. 25.7** Zusammenhang zwischen DDRAM-Inhalt und Pixel-Darstellung.

Dieser Zusammenhang gilt bei normalen Anzeige (keine Invertierung) und Startzeile gleich 0

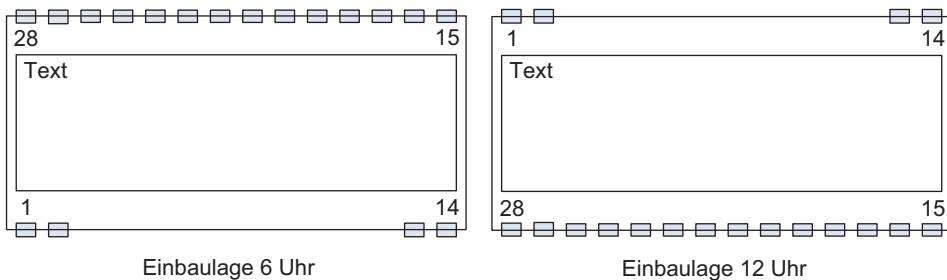
DDRAM		Display	
Page	Datenbyte m 0xD9	Spalte m	Zeilennummer
n	D0=1		8*n
	D1=0		8*n+1
	D2=0		8*n+2
	D3=1		8*n+3
	D4=1		8*n+4
	D5=0		8*n+5
	D6=1		8*n+6
	D7=1		8*n+7

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 des Displays macht sowohl eine 6-Uhr- als auch eine 12-Uhr-Einbaulage möglich, wie in Abb. 25.13 dargestellt. Die Page-Nummerierung beginnt immer von oben bei 0. Im linken Bild werden die Spalten beginnend von links von 0 bis 101 nummeriert, im rechten Bild von 30 bis 131.

### 25.4.2 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 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. 25.11 gezeigt wird.



**Abb. 25.13** DOGS – Display-Einbaulage

Ein Nachprogrammierung der folgenden Zeilen führt nur zum Erfolg, wenn der Abschnitt über die SPI-Schnittstelle verstanden wurde.

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. 25.14 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 Befehlsdekoder 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. 25.14 dargestellten Ablaufs zeigt folgender Codeausschnitt. In diesem Beispiel wird das gesamte Display durch das Speichern einer „0“ an jeder Adresse gelöscht.

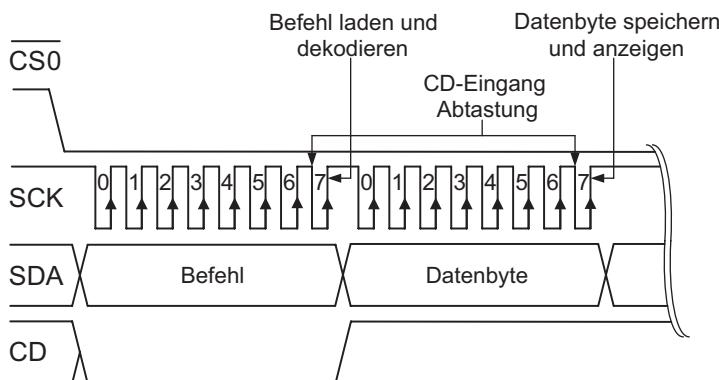


Abb. 25.14 DOGS – Display-SPI-Kommunikation

```

//die CS-Leitung wird auf Low gesetzt
SPI_Master_Start(sdevice_pins.DispDOGSSpi);
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);
    //die Seitenadresse wird gesetzt
    SPI_Master_Write(SET_PAGE_ADDRESS | ucPage);
    //die Anfangsspalte wird gesetzt
    DispDOGS_Set_Column(sdevice_pins, 0x00);
    //die folgenden Bytes sind Datenbytes
    DispDOGS_Set_CDHigh(sdevice_pins);
    for(ucColumn = 0; ucColumn < 102; ucColumn++)
    {
        SPI_Master_Write(0x00);
    }
}
SPI_Master_Stop(sdevice_pins.DispDOGSSpi); //Ende der Übertragung

```

### 25.4.3 Befehlssatz

Die Befehlscodes (siehe Tab. 25.8) werden über den SPI-Bus übertragen und der Zustand des CD-Eingangs bestimmt, ob das empfangene Byte in den DDRAM-Speicher oder in den Befehlsdecoder 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;

**Tab. 25.8** 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	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	

- **Ladungspumpe steuern** (Set Power Control, PC – Power Control) schaltet die Ladungspumpe für die Erzeugung einer LCD-Hochspannung bei PC2...0=111 ein oder bei PC2...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. 25.9);
- **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 SL5...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;

**Tab. 25.9** 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	

- **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-Uhr-Einbaulage kann die Initialisierungen durch folgende Schritte realisiert werden:

- Startzeile auf 0 setzen (SL5...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 (PC2...0=111b);
- Kontrast grobeinstellen (PC5...3=111b);
- 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
//1. Byte der Kontrast-Feineinstellung
#define SET_ELECTRONIC_VOLUME_1   0x81
//2. Byte der Kontrast-Feineinstellung
#define SET_ELECTRONIC_VOLUME_2   0x07
//Temperaturkoeffizient-Einstellung (1. Byte)
#define SET_ADVANCED_PROGRAM_CONTROL_0 0xFA
//Auswahl Temperaturkoeffizient (2. Byte)
#define TEMP_COMP_011              0x91
//Display in aktiv Modus versetzen
#define SET_DISPLAY_ON             0xAF

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 (Aus-  
    gang 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(ucDispDOGSIn  
    itValue[ucI]);  
    SPI_Master_Stopp(sdevice_pins.DispDOGSspi);  
    //der gesamte Inhalt des Speichers wird gelöscht  
    DispDOGS_Clear_Memory(sdevice_pins);  
}  
}
```

#### 25.4.4 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 5x7-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. 25.10 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 6x8-Pixel groß und auf dem gesamten Display können 8x17-Zeichen dieser Größe dargestellt werden.

**Tab. 25.10** Bitmuster des Zeichens „A“

Page		Spalte					
		m	m+1	m+2	m+3	m+4	m+5
n	2 <sup>0</sup>	0	1	1	1	0	0
	2 <sup>1</sup>	1	0	0	0	1	0
	2 <sup>2</sup>	1	0	0	0	1	0
	2 <sup>3</sup>	1	0	0	0	1	0
	2 <sup>4</sup>	1	1	1	1	1	0
	2 <sup>5</sup>	1	0	0	0	1	0
	2 <sup>6</sup>	1	0	0	0	1	0
	2 <sup>7</sup>	0	0	0	0	0	0
		0x7E	0x11	0x11	0x11	0x7E	0x00

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 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-Kommunikation wird gestartet
SPI_Master_Start(DispDOGS_1.DispDOGSspi);
DispDOGS_Set_CDLow(DispDOGS_1); //der CD-Eingang wird auf Low gesetzt
/* Seitenwahl */
SPI_Master_Write(SET_PAGE_ADDRESS | n);
/* Spaltenwahl - 2-Byte-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 Generierungbytes 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 neun 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 9x8 Bits von 18 µs. Für die Darstellung eines weiteren Zeichens rechts zum Vorigen, werden nur noch 12 µs gebraucht (Übertragungsdauer von sechs 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 à sechs 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. Samsung Electronics. KS0070B – Driver & Controller for dot matrix LCD. (2015). [www.datasheetcatalog.com](http://www.datasheetcatalog.com).
2. Display Elektronik. LCD Module DEM 16481 SYH-LY. (2015). [www.display-elektronik.de](http://www.display-elektronik.de).
3. Electronic Assembly. DOGS Graphic Series 102x64 Dots. (2015). [www.lcd-module.de](http://www.lcd-module.de).
4. UltraChip Inc. UC1701x 65x132 STN Controller Driver. (2015). [www.ultrachip.com](http://www.ultrachip.com).
5. Meroth, A., & Tolg, B. (2008) *Infotainmentsysteme im Kraftfahrzeug*. Friedr. Vieweg & Sohn Verlag | GWV Fachverlage GmbH.
6. Reisch, M. (2006). *Elektronische Bauelemente*. Springer.
7. Grünhaupt, G. (Hrsg.). (2006). *Handbuch der Mess- und Automatisierungstechnik im Automobil*. Springer-Verlag.



# Beispielprojekte

26

## Zusammenfassung

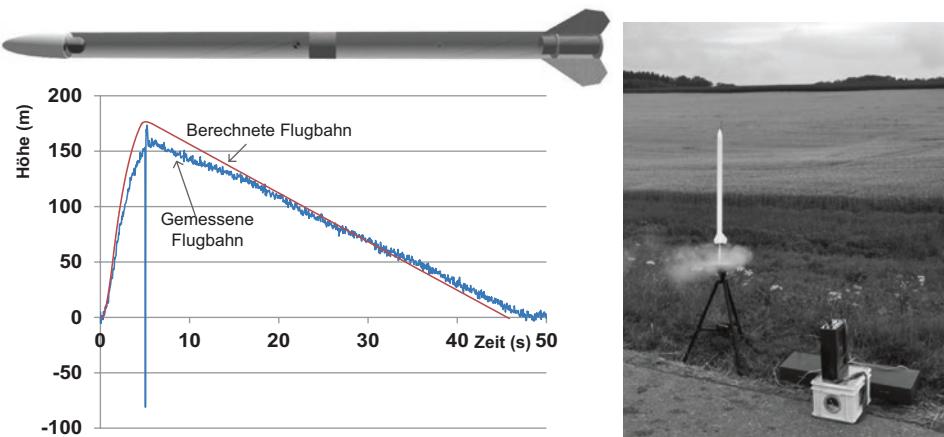
Das Kapitel beschreibt zwei Beispielprojekte, die auf dem Wissen im Buch aufbauen.

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. Ein weiteres Beispielprojekt ist eine verteilte Messstation mit CAN-Vernetzung, die hier ebenfalls vorgestellt wird.

## 26.1 Datenlogger

### 26.1.1 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,



**Abb. 26.1** Modellrakete mit Flugverlauf

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 Schoollab 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. 26.1 oben ist ein Modell der Rakete zu sehen. Deutlich erkennbar sind im Schnittbild der Motor und eines der beiden Fallschirmpakete. Die beiden Hauptrohre werden durch 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 mithilfe der barometrischen Höhenformel aus dem Luftdruck und der Temperatur ermittelt wurde, wird der Druckstoß beim Ausstoßen der Spitze als Höhenimpuls wahrgenommen. In Abschn. 26.1.4 werden auch die Beschleunigungsverläufe diskutiert.

### 26.1.2 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. 26.2 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 mm × 71 mm ausgelegt, damit es in den meisten Raketenspitzen Platz hat. Abb. 26.3 zeigt das Layout der doppelseitigen Platine, die in SMD-Technik von Hand beidseitig bestückt wurde, sowie die fertige Platine mit der Raketenspitze und dem Gehäuse. Die Öffnungen für den USB-Anschluss im Gehäuse und die Bohrung für den Druckausgleich in der Spitze sind gut zu erkennen.

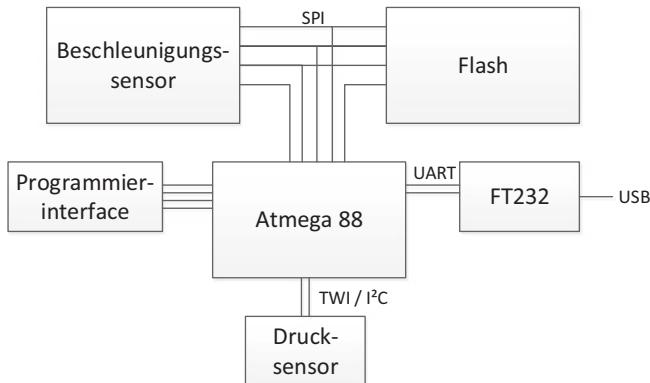
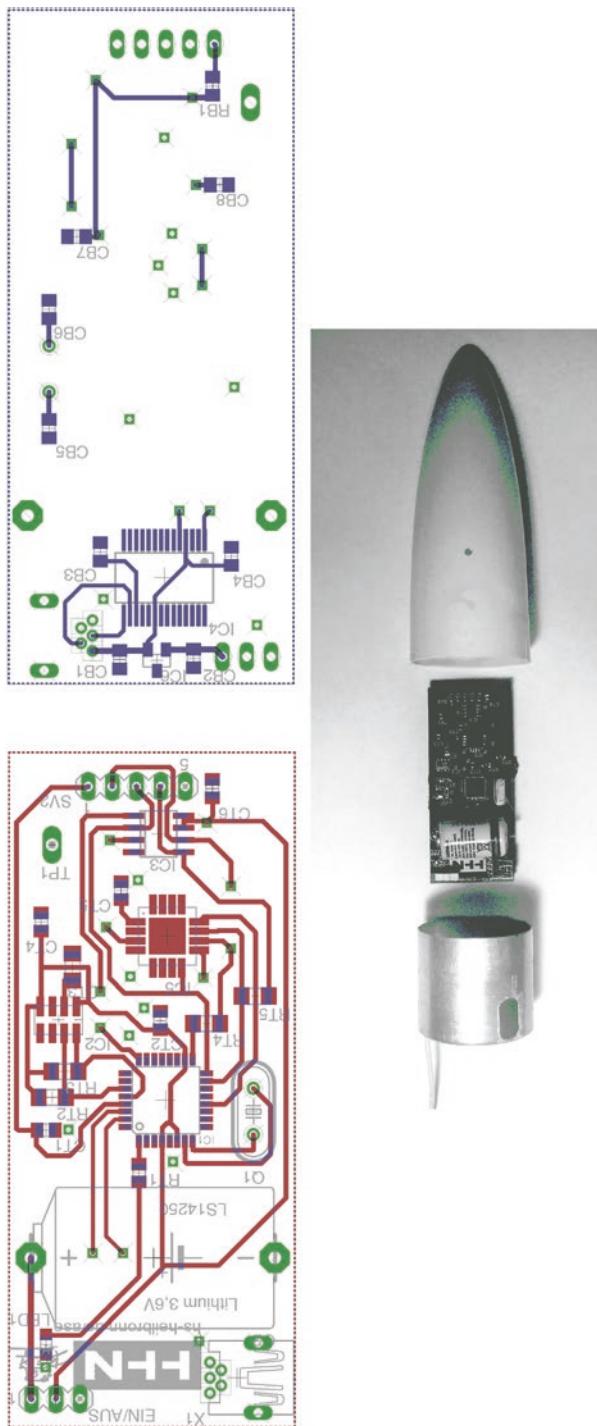


Abb. 26.2 Blockschaltbild des gesamten Projektes



**Abb. 26.3** Boardlayout und fertiges Board mit Raketenspitze

### 26.1.3 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.

#### 26.1.3.1 Der Zustandsautomat

Die gesamte Software ist als Zustandsautomat realisiert. Dabei existieren folgende Zustände:

- Warten
- Flash löschen
- Daten loggen
- Daten auslesen
- Stop

Der Zustandsautomat wird in einem 100 ms Task geschaltet. Folgende Ereignisse werden ausgewertet:

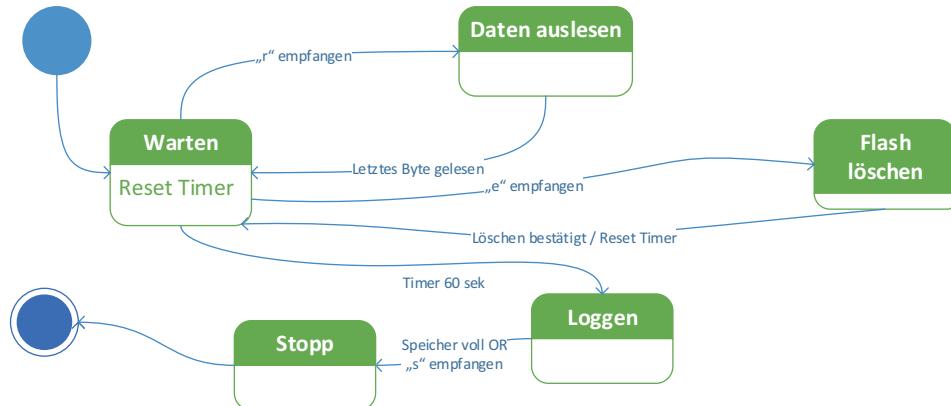
- 60 s abgelaufen
- Byte zum Flash löschen empfangen
- Byte zum Daten auslesen empfangen
- Flash voll (FLASH\_FULL) oder Byte zum Stoppen empfangen
- Flash komplett ausgelesen (SEND\_COMPLETE)

Damit kann der Automat wie folgt realisiert werden (Abb. 26.4).

#### 26.1.3.2 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:



**Abb. 26.4** Zustandsautomat des Raketenloggers

```

MPL3115_Write[ByteReg](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++;
}
    
```

### 26.1.3.3 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  $^{\circ}\text{C} \cdot 256$  vorliegt sowie den Beschleunigungen, die in  $2 \text{ m/s}^2$  vorliegen.

### 26.1.4 Auswertung

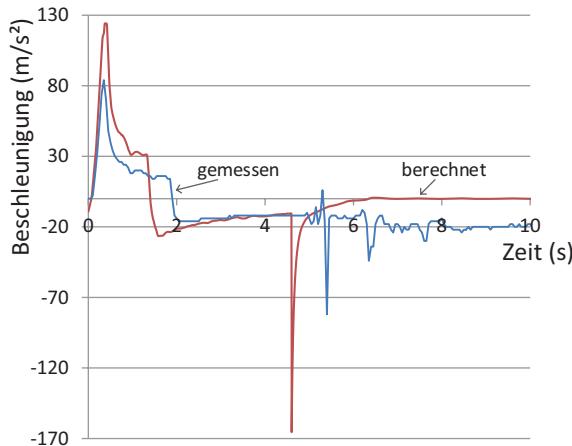
Die geflogene Höhe ergibt sich aus der bekannten und in jedem Physikbuch nachzuschlagenden barometrischen Höhenformel, die unter der Annahme einer isothermen Atmosphäre hergeleitet wurde. Bei den zu erwartenden Flughöhen ist diese Annahme noch zulässig und erweist sich auch bei der Temperaturmessung 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  $\Delta h = h_1 - h_0$

$$\Delta h = -\frac{R \cdot T}{M \cdot g} \ln\left(\frac{p(h_1)}{p(h_0)}\right) \quad (26.1)$$

Dies lässt sich in Excel aus den gemessenen Werten  $t$  (Temperatur in  $^{\circ}\text{C}$ ) und  $p$  bzw.  $p(h_0)$  am Boden gut berechnen, wobei für  $p$  nur das Verhältnis eine Rolle spielt. Durch



**Abb. 26.5** Beschleunigungsverlauf in Flugrichtung

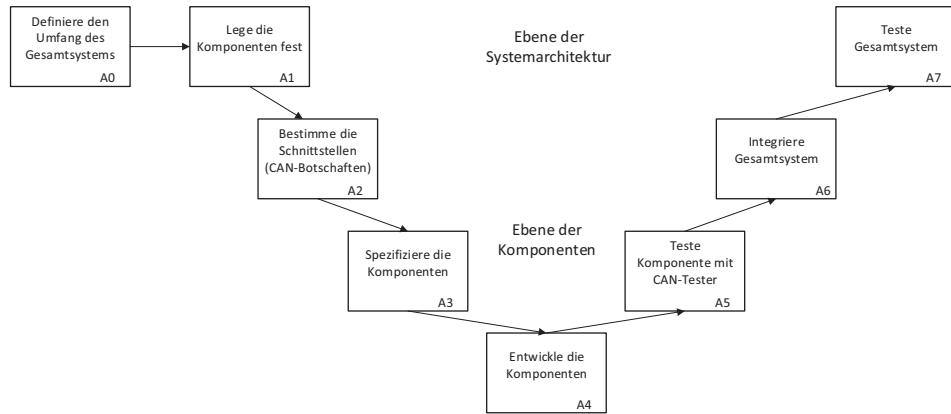
Anwendung der barometrischen Höhenformel berechnen sich Flugverläufe wie in Abb. 26.1 gezeigt.

Abb. 26.5 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. 26.1 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.

## 26.2 Smart Home mit CAN

In der Lehre werden immer wieder Beispiele für CAN-Netzwerke als Beispiele gesucht. Insbesondere bei Automobilkomponenten ist es oft schwierig, an eine geeignete Dokumentation heranzukommen, mit der die Komponente in ein Netzwerk integriert werden kann. Mit den Bausteinen, die in diesem Buch beschrieben sind, lassen sich jedoch vielfältige CAN-Netzwerke aufbauen, die sowohl einen praktischen als auch einen didaktischen Nutzen haben. Im folgenden Beispiel ist eine Anwendung skizziert, die mit Studierenden erfolgreich getestet wurden. Die Studierenden bekamen einen Schaltungsvorschlag, bestehend aus einem ATmega88 Controller, einem CAN-Controller vom Typ MCP2515 und einem CAN-Transceiver TJA1050 gemäß Kap. 10.

Die Aufgabe bestand darin, Komponenten für ein Smart Home zu entwickeln und mit CAN zu vernetzen. Jeder Sensor- oder Aktorknoten besteht also aus einem



**Abb. 26.6** Systementwicklungsmodell für das Beispielprojekt

Mikrocontroller, dem CAN-Controller/Transceiver und dem jeweiligen Sensor, Aktor oder den unten erwähnten Mikroschaltern zur Konfiguration, gemäß den Beschreibungen in den vorhergehenden Kapiteln. Der gesamte Entwicklungsablauf ist in Abb. 26.6 skizziert.

### 26.2.1 Aufbau

Das Beispielprojekt Smart Home besteht aus einem Codeschloss mit 12er Tastaturlblock, einem Feinstaubsensor, einem Funksender und –empfänger, einem Heizkörperthermostat, einem Bewegungsmelder, einem Rauchmelder, einem Rollladenantrieb mit PWM, einem Thermometer und einer Zentrale mit Display. Insgesamt ergeben sich dreizehn grundlegende CAN-Botschaften Tab. 26.1.

Diese sind:

- SollTemp: Die Solltemperatur wird von der Zentrale an den Heizkörper gesendet. Die Botschaft besteht aus der Solltemperatur zwischen 0 und 40 °C in Schritten von 0.5 K und der Kennung des Heizthermostats. Ist die Kennung des Heizthermostats 0xFF, werden alle Thermostate angesprochen, ist sie 0xFE, kann das Thermostat angelernt werden und dessen neue Kennung steht in Byte 3.
- HeizStatus: Die Ist-Temperatur (Auflösung wie Soll) und die Kennung werden vom Heizkörper übertragen
- CodeStatus: Das Codeschloss sendet eine Botschaft, wenn drei Fehleingaben stattgefunden haben, ebenso, wenn die Tür geöffnet wurde. Pro Schloss kann eine Kennung mitgeliefert werden (analog zum Heizkörper)
- CodeDoorLock: Eine Botschaft von der Zentrale, das Schloss zu öffnen oder zu schließen (Kennung des Schlosses plus Status). Ist die Kennung 0xFE muss in einem weiteren Byte eine neue Kennung übertragen werden.

**Tab. 26.1** CAN-Matrix des Smart Home

Bot-schaft	Name	CycleTime	CodeSchloss	Feinstaub	Funkalarm	Heizkörper	Bewegung	Rauch	Rollladen	Thermo/Hygro	Zentrale
0x100	SollTemp	1 s			Rx						Tx
0x101	HeizStatus	1 s			Tx						Rx
0x200	CodeStatus	100 ms	Tx					Rx			Rx
0x201	CodeDoorLock	Event	Rx								Tx
0x300	DustSensor	30 s		Tx							Rx
0x310	FunkBrickie	100 ms			Tx			Rx			Rx
0x400	Bewegung	1 s				Tx		Rx			Rx
0x401	BewegungScharf	100 ms				Rx					Tx
0xF01 n	RauchStatus	100 ms					Tx				Rx
0x500	GlobaleZeit	1 s						Rx			Tx
0x501	RollladenProg	100 ms						Rx			Tx
0x502	RollladenStatus	1 s						Tx			Rx
0x220	TempHum	1 s						Rx		Tx	Rx

- DustSensor: Der Feinstaubsensor sendet seine beiden Feinstaubwerte und eine Kennung (siehe Kap. 15)
- FunkBrücke: Ein 868 MHz Empfänger dient als drahtlose Brücke um Botschaften über CAN weiterzuleiten. Im Sendermodus leitet er eine CAN-Botschaft weiter.
- Bewegung: Ein PID Infrarotsensor ist am Eingang eines Mikrocontrollers angeschlossen. Über ein Bitmuster (DIP-Schalter) wird eine Kennung codiert.
- BewegungScharf: Die Zentrale sendet ein Byte und die Kennung des Bewegungsmelders. In einem weiteren Byte wird die Zeit bis zur Scharfschaltung in Sekunden angegeben, in einem vierten Byte die Verzögerungszeit bis zum Alarm in Sekunden
- RauchStatus: Jeder Rauchmelder sendet eine eigene CAN-Botschaft (über 4 Bit DIP-Schalter)
- GlobaleZeit: Die Uhrzeit je 1 Byte Stunde, Minute, Sekunde, Tag, Monat, Jahr, Wochentag wird von der Zentrale verteilt.
- RollladenProg: Die Zentrale versendet die Zeit in Stunde und Minute, sowie den Wochentag und eine Kennung der Rollladensteuerung. Ist die Kennung 0xFE kann eine neue Kennung über ein weiteres Byte angelernt werden.
- RollladenStatus: Die Rollladensteuerung sendet den aktuellen Status des Rolladens sowie ihre Kennung.
- TempHum: Ein Temperatur/Luftfeuchtefühler sendet seine Daten und eine Kennung (8 Bit DIP-Schalter)

Im Projekt müssen dann die jeweiligen Botschaften spezifiziert werden. Im Folgenden wird nur der Tastaturblock für das Codeschloss dargestellt, da sich die anderen Schaltungen jeweils aus den vorangegangenen Kapiteln ergeben. Es sei darauf hingewiesen, dass ein derart realisiertes System lediglich zu Demonstrationszwecken verwendet werden kann, insbesondere im Bereich der Rollladensteuerungen und der Heizventile müssen Einbauten nach den entsprechenden Normen umgesetzt werden und es besteht bei falschen Handhabung Lebensgefahr. Auch Rauchmelder aus solchen Laboraufbauten werden versicherungsrechtlich nicht anerkannt. Für die Ausbildung ist das Projekt jedoch sehr gut geeignet und generierte bei den Studierenden sehr viel Engagement. Insbesondere kann mit dem Projekt die Vorgehensweise einer Systementwicklung demonstriert werden. Die Studierenden lernen, zunächst den Gesamtsystemumfang zu beschreiben, anschließend eine Systemarchitektur festzulegen, bei der die Schnittstellen in Form von CAN-Botschaften beschrieben sind. Im Anschluss werden die Komponenten unabhängig voneinander entwickelt und getestet und schließlich zu einem Gesamtsystem integriert.

### 26.2.1.1 Tastaturblock

Der Tastaturblock wird mit als Matrix ausgeführt. Aus verschiedenen Gründen wurde entschieden, einen Portexpander PCA9534 (Abschn. 24.1) zur Ansteuerung zu verwenden. Im Codeschloss wird der korrekte Code (dreistellig) mit 000 vorbelegt. Die Eingabe soll nach folgenden Kriterien erfolgen:

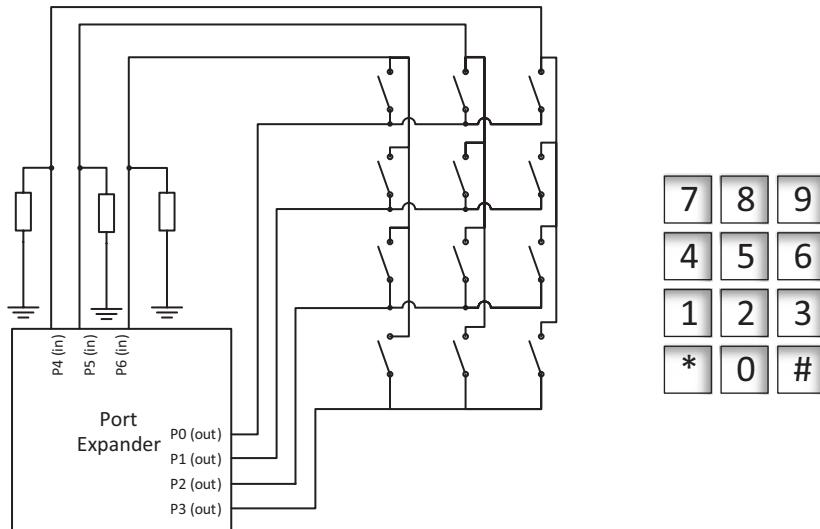
- Eingabe des korrekten Codes plus \* öffnet die Tür (Beispiel: 000\*)
- Neu programmieren erfolgt durch <alter Code>#<neuer Code>#, Beispiel 123#456#
- Nach drei Fehleingaben wird ein Alarm ausgelöst, indem eine CAN-Botschaft geschickt wird, die von der Zentrale und den Rauchmeldern empfangen wird, wodurch diese ihren Alarmton aktivieren.

Am Mikrocontroller sind eine rote und eine grüne LED angeschlossen, sowie ein Treiber, der einen Türsummer schaltet. Die Tür wird geöffnet, wenn der richtige Code eingegeben wurde oder durch eine CAN-Botschaft. Die grüne LED leuchtet, wenn die Tür geöffnet wird, die rote, wenn ein falscher Code eingegeben wurde.

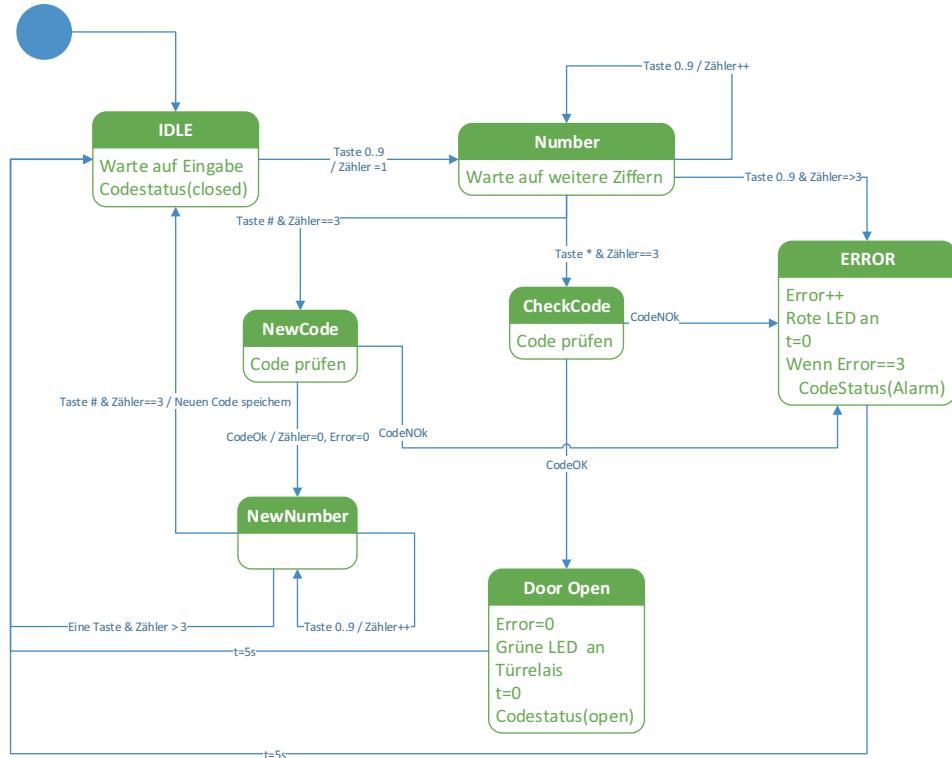
Das Auslesen der Tastatur erfolgt scannend wie in Abb. 26.7 skizziert. Je drei Taster werden zu einer Zeile zusammengefasst, die vier Zeilen liegen auf einem Port-Ausgang, der standardmäßig auf 0 liegt. Nun wird hintereinander je eine Zeile auf 1 gesetzt und in den Spalten ausgelesen. Ist ein Taster gedrückt, dann liegt an der entsprechenden Spalte eine 1 (Abb. 26.7).

Für den Ablauf ist ein Zustandsautomat (Abschn. 5.2) zu entwerfen.

Das Codeschloss sendet eine zyklische CAN-Botschaft, die angibt, ob die Tür geschlossen ist, geöffnet wurde, oder ob eine dreimalige Falscheingabe erfolgte, was ebenfalls zu einem Alarm führt (Abb. 26.8).



**Abb. 26.7** Anschluss eines Tastaturoblocks



**Abb. 26.8** Zustandsautomat des Codeschlosses

## Literatur

1. Open rocket. <http://openrocket.info/>. Zugriffen: Febr. 2021



---

## Erratum zu: Sensornetzwerke in Theorie und Praxis

---

### Erratum zu:

**A. Meroth und P. Sora, *Sensornetzwerke in Theorie und Praxis*,**  
**<https://doi.org/10.1007/978-3-658-31709-6>**

Die Autoren haben Zusatzmaterialien zu Kapitel 5, Kapitel 7, Kapitel 8, Kapitel 9, Kapitel 10, Kapitel 12, Kapitel 13, Kapitel 15, Kapitel 16, Kapitel 17, Kapitel 18, Kapitel 19, Kapitel 20, Kapitel 21, Kapitel 22, Kapitel 23, Kapitel 24 und Kapitel 25 nachträglich eingereicht. Diese Zusatzmaterialien sind nun Online verfügbar und die Information dazu ist in den einzelnen Kapiteln aktualisiert worden.

Weiterhin wurde der Buchumschlag auch korrigiert und die Inhaltsinformation angepasst.

---

Die korrigierte Version des Buches ist verfügbar unter  
<https://doi.org/10.1007/978-3-658-31709-6>

---

# Stichwortverzeichnis

- 1-Wire, 298
- A**
- Abtasttheorem, 368
    - Nyquist-Shannon, 101
  - Address Resolution Protocol (ARP), 185
  - Aerosol, 418
  - Aktivmatrix-LC-Anzeige (AMLCD), 632
  - Alarmzeit, 621
  - Aliasing, 371
  - ALU s. Arithmetic Logic Unit
  - Ambient Light Sensing, 497
  - Analog-Digitalwandler (AD-Wandler), 108, 110
  - Analogkomparator, 108, 109
  - Analogmultiplexer, 108
  - Annäherung, sukzessive, 526
  - Anpassung, 164
  - Anweisung, 11
    - #include, 36
    - else, 26
    - if, 25
    - switch, 28
  - Application Specific Integrated Circuit (ASIC), 374
  - Approximation, sukzessive, 110, 526
  - Arbeitsspeicher, 66, 71
  - Architektur, 129
  - Arithmetic Logic Unit (ALU), 64
  - Array, 40
  - ASCII (American Standard Code for Information Interchange), 41, 290
  - AT-Befehl, 362
  - AtmelStudio, 59
- Ausdruck, 11
- Automatic Repeat Request, 182
- AUTOSAR, 284
- AVR Libc (Funktionsbibliothek), 75
- B**
- Baudrate, 196
  - Bedingungsoperator, 27
  - Beschleunigungssensor, 428
  - Bezeichner, 10
  - Big-endian, 212
  - Block Size, 282
  - Bluetooth, 196, 351
  - Board Abstraction Layer, 379
  - Bootloader, 71
  - Brown-Out-Detection, 67
  - Brown-Out-Reset-Schaltung, 610
  - Bus, 169
  - Bytestuffing, 175
- C**
- Call-by-value-Prinzip, 32, 52
  - Carrier Sense Multiple Access (CSMA), 176
  - Central Processing Unit (CPU), 64
  - CISC s. Complex Instruction Set Computer
  - Clock-Streckung, 404
  - Codemultiplex, 334
  - Compiler, 60
  - Complex Instruction Set Computer (CISC), 65
  - Consecutive Frame, 282
  - Constraint Network, 191
  - Constraint Nodes, 190
  - Controller Area Network (CAN), 263

- CAN-Controller, 268  
 CAN-Frame, 265  
 CAN-Matrix, 670  
 CANopen, 284  
 CAN-Timing, 266  
 in Automation (CIA), 284  
 Coordinator-ZigBee, 357  
 CORDIC-Algorithmus, 485  
 Counter, 87  
 CPU s. Central Processing Unit  
 Crosstalk, 164  
 CSMA/CA, 264  
 Cyclic Redundancy Check (CRC), 305, 404
- D**
- D/A-Wandler, 516  
 Datenrichtungs-Register, 75  
 Datenstruktur, dynamische, 143  
 Datentyp, 15  
 Debugger, 60  
 Deklaration, 17  
 Digital-Analog-Wandler (DA-Wandler), 513  
 Digital Signal Processor (DSP), 338, 374  
 Domain Name System, 184  
 Doxygen, 12  
 DS18S20, 309
- E**
- Echtzeituhr, 618  
 EEPROM (Electrically Erasable Programmable Read-Only Memory), 66, 123, 319, 539, 551  
 Effektivwert, 120  
 Eingangsschnittstelle, analoge, 108  
 Einkopfsystem, 490  
 Empfänger 868 MHz, 671  
 End-Device, 357  
 Endian, 392, 434, 463, 480, 502  
 Endurance, 552, 563  
 Energie sparen, 109, 120, 122  
 Energy Harvesting, 190  
 Ereignis, 149
- F**
- Fehlerbehandlung, 179  
 Fehlererkennung, 179
- Fehlerkorrektur, 182  
 Fehlerspeicher, 123  
 Feinstaub, 418  
 Feinstaubsensor, 671  
 Festwertspeicher, 539  
 serieller, 542  
 FIFO (First in – First Out), 140  
 Filter, 273, 280  
 Finite-Impulse-Response (FIR)-Filter, 373  
 First Frame, 282  
 Flash, 562  
 Flashspeicher, serieller, 564  
 Floating-Gate-Feldeffekttransistor, 539  
 Flow Control Frame, 282  
 Flow State, 282  
 Flussdiagramm, 26, 30  
 Forward Error Correction, 182  
 Framebuffer, 634  
 free (Speicher), 50  
 Frequenzmultiplex, 334  
 FT232RL, 663  
 Full Function Device (FFD), 357  
 Funktion, 32  
     Definition, 33  
     free, 126  
     getter, 133  
     main(), 32  
     malloc, 126  
     setter, 133  
 Funktionsprototyp, 33  
 Funkübertragung, 333  
 Fuse, 69
- G**
- Ganzzahlarithmetik, 380  
 Ganzzahltyp, 15  
 Gateway, 183, 358  
 Gieren, 455  
 Gleitkommamatyp, 15  
 goto, 31  
 Guard, 150  
 Gyroskop, 456
- H**
- .h, 36  
 Halbduplex, 298  
 Hamming, 374

- Handle, 145  
Hanning, 374  
Hardware-Abstraktion, 131  
Harvard-Architektur, 66  
Header, 36, 327  
Heap, 50  
Heißleiter, 117  
High-Pegel, rezessiver, 167  
Hinterleuchtung, 632  
Historyzustand, 153  
Höhenformel, 667
- I**  
I<sup>2</sup>C, 298  
    Bus, 241  
IDE s. Integrated Development Environment  
Index, 40  
Inertialsensor, 427  
Infinite-Impulse-Response (IIR)-Filter, 376  
Inkrementoperator, 21  
Inline-Dokumentation, 12  
Input/Output, digitaler, 75  
Integrated Development Environment (IDE),  
    8, 59  
Intel-Format, 212  
Interrupt, 81  
    externe, 86  
    nested, 86  
    Service Routine, 81  
ISM, 191, 334  
ISO 15765, 281
- J**  
JavaScript Object Notation (JSON), 189  
Jitter, 178
- K**  
Kanalcodierung, 182  
Kommentar, 12  
Komponentensicht, 131  
Konkurrenzphase, 265  
Konstante, 18  
Konstantstromquelle, 604  
Kontextsicht, 130
- L**  
Latenzzeit, 178  
Leitung, 162  
Leitwerk, 64  
Letzte Meile, 190  
Linker, 60  
Liquid Crystal Display (LCD), 630  
Liste, verkettete, 140, 143  
Little-endian, 212, 271, 463  
Local Interconnect Network (LIN), 326  
Longitudinal Redundancy Check (LRC), 293  
Lookup-Tabelle (LUT), 119, 408, 485  
Low-Pegel, dominanter, 167  
LSB first, 212  
Luftdrucksensor, 387  
Luftfeuchtigkeit, 399
- M**  
MAC-Adresse, 185  
Magnetfeldsensor, 475  
Makro, 19, 34  
    #elif, 56  
    #endif, 56  
    #error, 57  
    #ifndef, 56  
malloc (Bibliotheksfunktion), 50  
Maschinencode, 9  
Master  
    In Slave Out (MISO), 220  
    Out Slave In (MOSI), 220  
Master-Slave-Bus, 297  
Master-Slave-Prinzip, 242  
Master-Slave-Protokoll, 176, 327  
Mehrfachauswahl, 27  
Mesh, 169  
Message Object, 274  
Message Queuing Telemetry Transport  
    (MQTT), 188  
Micro-Electro-Mechanical Systems (MEMS),  
    367, 389  
MMA6525 (Beschleunigungssensor), 663  
MODBUS, 287  
MOSFET, 539  
Motorola-Format, 212  
MPL3115 (Drucksensor), 663  
MSB first, 212

Multipoint, 166  
Multitasking, 137

## N

Nachrichtenkopf, 327  
Näherungssensor, 489  
    optischer, 497  
Neigungssensor, 439  
Nicken, 455  
Non-return-to-Zero, 196  
Nyquist-Kriterium, 119, 168, 368, 371

## O

Objektverzeichnis, 284  
OLED, 633  
Operator, 23  
    arithmetischer, 21  
    bitweiser, 21  
    logischer, 22  
    Speicher, 23  
Output Compare Register, 91  
Over-the-air (OTA)-Zugriff, 190

## P

Padding, 283  
Page, 551  
Pairing, 356  
Paritätsbit, 197  
PCA9534 (Port-Expander), 671  
PCI s. Pin Change Interrupt  
Peer-to-peer, 196  
PID, 327  
Pin Change Interrupt (PCI), 82  
Pointer, 50  
Point-to-Point, 166  
Polling, 82, 244, 251, 465, 557  
Port, 186  
Port-Expander, 603, 671  
Postfix, 19  
Potentiometer, digitales, 610  
Power Management, 120  
Power Reduction Register, 122  
Präcompiler, 56  
    #define, 19, 56  
Präprozessor, 56

Prescaler, 87  
Profil, Bluetooth, 352  
Programmzähler (program counter, PC), 65  
Protokolle, 160  
Proximity Sensor, 489  
Prozess  
    asynchroner, 140  
    isochroner, 140  
Prozessdatenobjekt (PDO), 284  
Pseudozustand, 151  
Puffer, 140  
Pullup-Widerstand, 75  
    externe, 77  
    interne, 80  
Pulsamplitudenmodulation (PAM), 369  
Pulsweitenmodulation, 94

## Q

Quantisierung, 371  
Quantisierungsrauschen, 372  
Quellcode, 60

## R

Radio, 584  
Real Time Clock, 618  
Receive-Puffer, 268  
Rechenwerk, 64  
Reduced Function Device (RFD), 357  
Reduced Instruction Set Computer (RISC), 65  
Redundanz, 179  
Referenz, 50  
Referenzspannung  
    ADC, 113  
    interne, 108  
Reflexionsfaktor, 165  
Regelwiderstand, digitaler, 609  
Remote-Terminal-Unit (RTU), 290  
    Modbus, 290  
Request for Comments (RFC), 162  
Reset, 69, 70  
Retention-Time, 563  
Ring, 169  
Ringpuffer, 140  
RISC s. Reduced Instruction Set Computer  
RMS s. Root Mean Square  
Roboterclub Aachen, 267

- Rollen, 455  
Rollover-Betrieb, 269, 273  
Root Mean Square (RMS), 120  
Roundtrip-Zeit, 178  
Router, 183, 357  
RS232, 195  
RT, 289
- S**  
Sample and Hold, 112, 369  
Sättigung, 384  
Schaltung  
    wired-AND, 167  
    wired-OR, 167  
Scheduler, 137  
Schleife  
    do-while-Schleife, 30  
    for-Schleife, 30  
    fußgesteuerte, 30  
    kopfgesteuerte, 29  
    while-Schleife, 29  
Schlüsselwort, 10  
    #endif, 37  
    #ifndef, 37  
    extern, 37  
Separation Time, 282  
Sequenzdiagramm, 166  
Serial Port Profile (SPP), 361  
Service ID (SID), 48  
Servicedatenobjekt (SDO), 284  
Skalierung, 385  
Sleep-Modus, 121, 330  
Socket, 187  
Speicher, 70  
    dynamischer, 126  
Sprite, 634  
SST25PF (Flashspeicher), 663  
Stampfen, 455  
Startzustand, 151  
State chart, 151  
State Machine, 149  
Status-Register, 381  
Stern, 169  
Steuerwerk, 64  
Störabstand, 372  
String, 41  
    nullterminierter, 41  
Strommessung, 530
- Struktur, 44  
Supercap, 618  
Symbol, 11
- T**  
Task, 134, 136  
Tastaturlblock, 669  
TCP/IP, 287  
Temperaturkoeffizient, negativer (NTC), 117  
Temperaturmessung, 409  
Temperatursensor, 309  
Thermometer, analoges, 117  
Thermostat, 414  
Thin Film Transistor (TFT), 632  
TIA/EIA-232, 195, 288  
TIA/EIA-485, 287, 288  
Tieppassfilter, 368  
Time Division Multiple Access (TDMA), 176  
Time Quantum, 266  
Timer, 87  
TJA1050, 668  
Topic, 188  
Transition, 149  
Transmit-Puffer, 268  
Transportprotokoll, 281  
Trust Center, 358  
TS5204, 663  
Tunneleffekt, 540  
TWI, 247  
    ISR, 251  
Typumwandlung  
    explizite, 25  
    implizite, 24
- U**  
Ultraschall-Näherungssensor, 489  
UML (Unified Modeling Language), 151  
UNI/O, 312  
Universal Asynchronous Receiver/Transmitter  
    (UART), 195, 327, 361
- V**  
Variable  
    extern, 132  
    globale, 36  
    lokale, 35

Vermittlungsschicht, 183  
Versorgung, 67  
Verteilungssicht, 130  
Verzweigung, 25  
Vollduplex, 197  
Von-Neumann-Architektur, 66

**W**

Wanken, 455  
Warteschlange, 140  
Watchdog, 70  
Wellenwiderstand, 164  
Wrapping, 131

**Z**  
Zählschleife, 30  
Zeichenkette, 41  
Zeiger, 50  
    auf Funktionen, 53  
Zeitmultiplex, 301, 334  
ZigBee, 357  
Zustandsautomat, 665  
Zustandsautomat, endlicher, 149  
Zustandsdiagramm, 151  
Zustandsübergangstabelle, 151  
Zuweisungsoperator, 20  
Zweierkomplement, 15  
Zweikopfsystem, 491  
Zyklus, 144