

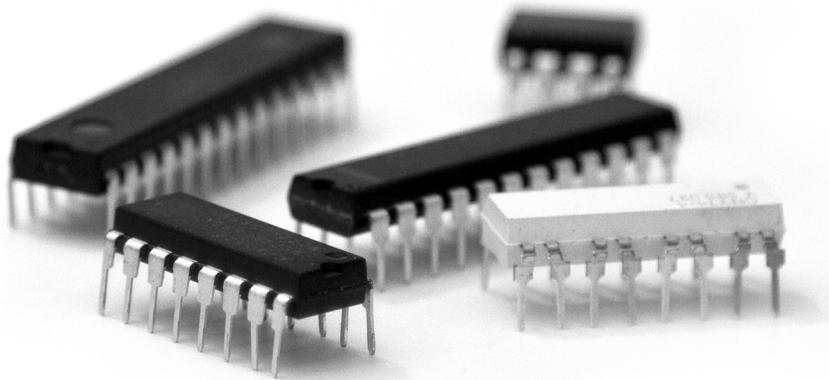


---

Skriptum zum Praktikum

**Reglerimplementierung auf  
Mikrocontrollern**

Stand: 6. September 2018



# Inhaltsverzeichnis

<b>Abbildungsverzeichnis</b>	<b>v</b>
<b>Tabellenverzeichnis</b>	<b>viii</b>
<b>Ablauf des Praktikums</b>	<b>ix</b>
<b>Literaturempfehlung</b>	<b>x</b>
<b>1. Grundlagen Mikrocontroller</b>	<b>2</b>
1.1. Was ist ein Mikrocontroller? . . . . .	2
1.2. Mikrocontroller, Mikroprozessoren und Mikrorechner . . . . .	2
1.3. Funktionsweise und Architektur eines typischen Mikrocontrollers . . . . .	3
1.3.1. Rechnerstrukturen . . . . .	3
1.3.2. CPU: Steuer- und Rechenwerk . . . . .	5
1.3.3. Speicheraufbau und -typen . . . . .	8
1.3.4. Peripherie . . . . .	9
1.3.5. Fuse- und Lockbits . . . . .	11
1.4. Basics aus Arithmetik und Logik . . . . .	11
1.4.1. Zahlensysteme und Datentypen . . . . .	11
1.4.2. Grundoperationen und Verknüpfungen für ganze Zahlen . . . . .	12
1.4.3. Einschub: Operatoren in C . . . . .	16
1.4.4. Einschub: ASCII-Code . . . . .	17
1.5. Minimalschaltung, Bauformen und Anschlüsse . . . . .	18
1.6. Programmierung, Flashen und Debugging . . . . .	20
1.6.1. Schnittstellen . . . . .	20
1.6.2. Programmieradapter und -software . . . . .	21
1.6.3. Programmiersprachen . . . . .	21
<b>2. I/O-Ports</b>	<b>22</b>
2.1. Treiberstufen . . . . .	22
2.2. I/O-Ports am AVR: DDR, PORT und PIN Register . . . . .	24
2.3. Leistungsfähigkeit . . . . .	25
<b>3. Serielle Schnittstelle UART</b>	<b>29</b>
3.1. Spezifikation und Protokoll . . . . .	29
3.2. UART im AVR . . . . .	31
3.2.1. Steuerregister . . . . .	31
3.2.2. Datentransfer (Empfang/Versand) . . . . .	32
3.2.3. Polling . . . . .	34
<b>4. Programmablauf und Interrupts</b>	<b>36</b>
4.1. Sprünge und elementare Programmstrukturen . . . . .	36
4.2. Assembler und höhere Programmiersprachen . . . . .	36
4.3. Interrupts . . . . .	38
4.3.1. Funktionsweise . . . . .	38

4.3.2. Interruptsteuerung . . . . .	39
4.3.3. Externe Interrupts am AVR . . . . .	40
4.3.4. Verschachtelte Interrupts . . . . .	40
4.3.5. Ist die ISR schnell genug? . . . . .	41
4.3.6. Interruptgesteuerte USART-Kommunikation . . . . .	42
4.3.7. ISR-Ablauf . . . . .	42
4.3.8. Race-Condition . . . . .	45
<b>5. Timer/Counter und Pulsweitenmodulation</b>	<b>50</b>
5.1. Was ist ein Timer/Counter . . . . .	50
5.2. Timer/Counter im AVR . . . . .	51
5.2.1. 8-bit Timer/Counter0 im ATmega8 . . . . .	51
5.3. Watchdog-Timer . . . . .	53
5.3.1. Was macht ein Watchdog-Timer? . . . . .	53
5.3.2. Watchdog-Timer im AVR . . . . .	53
5.4. Pulsweitenmodulation (PWM) . . . . .	54
5.4.1. Periode, Pulsweite und Tastverhältnis . . . . .	54
5.4.2. Anwendung und Einsatzbereiche . . . . .	55
5.4.3. PWM im AVR . . . . .	56
<b>6. Zahlendarstellung und Arithmetik</b>	<b>60</b>
6.1. Motivation . . . . .	60
6.2. Umwandlungsverfahren . . . . .	60
6.3. Datentypen . . . . .	62
6.3.1. Vorzeichenlose ganze Dualzahlen (unsigned integer) . . . . .	62
6.3.2. Vorzeichenbehaftete ganze Dualzahlen (signed integer) . . . . .	63
6.3.3. Festkommadarstellung (fixed point) . . . . .	64
6.3.4. Gleitkommadarstellung (floating point) . . . . .	64
6.4. Arithmetische Operationen . . . . .	65
6.4.1. Rechnung in der Festpunkttdarstellung . . . . .	65
6.4.2. Rechnung in Gleitpunkttdarstellung . . . . .	66
6.4.3. Ergebniss-Analyse mit dem Statusregister SREG . . . . .	66
<b>7. Analog/Digital-Wandlung</b>	<b>69</b>
7.1. Signalklassen . . . . .	69
7.2. Umrechnung, Auflösung und Genauigkeit . . . . .	69
7.2.1. Auflösung . . . . .	70
7.2.2. Fehler . . . . .	70
7.2.3. Geschwindigkeit . . . . .	72
7.3. A/D Wandler im AVR . . . . .	72
<b>8. Die H-Brücke</b>	<b>75</b>
8.1. Grundgerüst . . . . .	75
8.2. Betriebsmodi . . . . .	75
8.3. Realisierung mit Transistoren . . . . .	76
8.3.1. Was ist ein Transistor? . . . . .	76
8.3.2. Bipolartransistoren (BJTs) . . . . .	76

8.3.3. MOSFET . . . . .	77
8.3.4. MOSFET vs. BJT . . . . .	78
8.4. Single-Chip H-Brücken und Selbstbau . . . . .	78
8.5. Die H-Brücke und der Mikrocontroller . . . . .	79
<b>9. Zeitdiskrete Regelung</b>	<b>80</b>
9.1. Einleitung . . . . .	80
9.2. Zeitdiskrete Modellbeschreibung . . . . .	80
9.2.1. Differenzengleichung . . . . .	80
9.2.2. Z-Übertragungsfunktion . . . . .	80
9.2.3. Zeitdiskrete Zustandsraummodelle . . . . .	81
9.3. Abtastsysteme . . . . .	85
9.3.1. Grundsätzliches . . . . .	85
9.3.2. Zeitdiskrete Beschreibung abgetasteter kontinuierlicher Systeme . . . . .	85
9.4. Digitaler Regelkreis . . . . .	88
9.4.1. Reglerentwurf . . . . .	88
9.4.2. Zeitdiskrete Zustandsbeobachtung . . . . .	92
9.4.3. Das Separationsprinzip . . . . .	93
9.5. Echtzeitfähige Implementierung des Regelgesetzes . . . . .	94
<b>10. Drehgeber / Encoder</b>	<b>96</b>
10.1. Funktionsweise . . . . .	96
10.2. Auswertungsmöglichkeiten . . . . .	97
10.2.1. Timerbasiert (polling) . . . . .	98
10.2.2. Externe Interrupts . . . . .	98
10.2.3. Quadraturdecoder . . . . .	98
<b>11. Bus-Systeme</b>	<b>100</b>
11.1. I <sup>2</sup> C-Bus . . . . .	100
11.1.1. Protokoll, Geschwindigkeit und Verwendung . . . . .	100
11.1.2. Implementierungsdetails . . . . .	101
11.1.3. I <sup>2</sup> C Modul der Mega-Familie . . . . .	102
11.2. SPI-Bus . . . . .	103
11.2.1. Protokoll, Geschwindigkeit und Verwendung . . . . .	104
11.2.2. Implementierungsdetails . . . . .	104
11.2.3. SPI Modul der Mega-Familie . . . . .	106
<b>12. Flachheitsbasierte Steuerung und Regelung</b>	<b>108</b>
12.1. Grundlagen . . . . .	108
12.2. Beispiel: Trajektorienfolge . . . . .	109
12.2.1. Steuerung . . . . .	109
12.2.2. Regelung . . . . .	110
<b>A. Bootloader</b>	<b>113</b>
A.1. Integration des AVRBootloaders in das AtmelStudio . . . . .	113
A.1.1. Voraussetzungen . . . . .	113
A.1.2. Anpassen der Batch-Datei . . . . .	113

A.1.3. Einbinden in AtmelStudio . . . . .	114
A.1.4. Erstellen eines Buttons zum Bootloader . . . . .	114
A.1.5. Vorgehensweise zum Flashen des Mikrocontrollers . . . . .	116
<b>B. Ausgabe von Fließkommazahlen</b>	<b>117</b>
B.1. Der printf-Befehl . . . . .	117
B.1.1. Verwendung des sprintf-Befehls . . . . .	117
B.1.2. Nutzung der Fließkomma-Version des sprintf-Befehls . . . . .	118
<b>C. Ampelplatine</b>	<b>120</b>
C.1. Kurzbeschreibung Ampelplatine . . . . .	120
C.1.1. Komponenten . . . . .	120
C.1.2. Stromversorgung . . . . .	121
C.1.3. USB-UART-Brücke . . . . .	121
C.1.4. Pin-Belegung . . . . .	121
C.1.5. Jumper-Konfiguration . . . . .	123
C.1.6. Programmierung . . . . .	123
C.2. Interrupt-Vektoren im ATmega8 . . . . .	124
<b>D. Kleinroboter: Balancieren und Trajektorienfolge</b>	<b>125</b>
D.1. Aufbau KRT8 . . . . .	125
D.2. Bluetooth Verbindung . . . . .	126
D.3. H-Brücken . . . . .	128
D.4. Sensorik . . . . .	128
D.4.1. Quadraturzähler . . . . .	128
D.4.2. Beschleunigungssensoren . . . . .	128
D.4.3. Drehratensensoren . . . . .	129
D.4.4. Sensordatenfusion . . . . .	129
D.5. Die Codevorlage . . . . .	134
D.5.1. Module des Quellcodes . . . . .	134
D.5.2. Programmablauf . . . . .	134
D.6. Balancieren (Tag 3 bis 6) . . . . .	135
D.6.1. Motoransteuerung – Modul <code>motor</code> . . . . .	135
D.6.2. Auslesen der SPI Sensoren - Modul <code>sensorenSPI</code> . . . . .	136
D.6.3. Identifikation der Motorkennlinie – Modul <code>motor</code> . . . . .	137
D.6.4. Sensordatenfusion . . . . .	137
D.6.5. PID-Kaskade zum Balancieren – Modul <code>reglerBalancieren</code> . . . . .	140
D.7. Trajektorienfolge (Tag 7 bis 9) . . . . .	141
D.7.1. Beobachter und Steuerung – Modul <code>reglerTrajektorienfolge</code> . . . . .	141
D.7.2. Flachheitsbasierte Regelung – Modul <code>reglerTrajektorienfolge</code> . . . . .	143
D.7.3. Vergleich und Wettbewerb . . . . .	143
D.8. Interrupt-Vektoren im ATmega32 . . . . .	144
<b>Literaturverzeichnis</b>	<b>145</b>

# Abbildungsverzeichnis

1.1.	Mikrocontroller verschiedener Hersteller und Bauformen . . . . .	2
1.2.	Abgrenzung Mikrocontroller und Mikroprozessor. . . . .	3
1.3.	Schema einer Fernbedienung mit Mikrocontroller. . . . .	4
1.4.	Gegenüberstellung Von-Neumann- und Harvard-Architektur. . . . .	4
1.5.	Blockschatzbild ATmega8 . . . . .	6
1.6.	Blockschatzbild AT89LP214 . . . . .	7
1.7.	Zusammenwirken von Rechen- und Steuerwerk. . . . .	8
1.8.	CPU des ATmega8. . . . .	9
1.9.	Auszug aus dem Instructionset der AVR-Familie. . . . .	13
1.10.	Maximum Ratings ATmega8. . . . .	19
1.11.	ATmega8 in verschiedenen Bauformen. . . . .	20
1.12.	Verschiedene Programmiergeräte. . . . .	21
2.1.	Schema eines I/O-Ports. . . . .	22
2.2.	Ausgangstreiber. . . . .	23
2.3.	Modellschaltung eines Portanschlusses. . . . .	24
2.4.	I/O-Register von Port D des ATmega8. . . . .	24
2.5.	Input-Register von Port D des ATmega8. . . . .	25
2.6.	Elektrische Eigenschaften der Treiberstufen des ATmega8. . . . .	26
2.7.	Kennlinien verschiedener LEDs. . . . .	27
2.8.	Anschluss einer LED an den Mikrocontroller. . . . .	27
2.9.	I/O Pin Input Threshold Voltage vs. $V_{cc}$ . . . . .	28
3.1.	Synchrone serielle Datenübertragung. . . . .	29
3.2.	Vergleich von USART und UART Übertragung. . . . .	30
3.3.	USART Control und Status Register. . . . .	32
3.4.	UCSZ Bit Einstellungen. . . . .	33
3.5.	Einstellungen für Parität und Anzahl der Stop-Bits. . . . .	33
3.6.	UBRR Bit Einstellungen. . . . .	33
3.7.	Beispiele für UBRR-Einstellungen. . . . .	34
4.1.	Elementare Programmstrukturen. . . . .	36
4.2.	Aufruf des generierten Assemblercodes. . . . .	37
4.3.	Ablauf einer Programmunterbrechung. . . . .	38
4.4.	Kontrollregister zur Interruptsteuerung. . . . .	41
4.5.	USART Control and Status Register B. . . . .	42
4.6.	Worst-Case Szenario für eine Race Condition. . . . .	46
5.1.	Blockschatzbild 8-Bit Timer/Counter0 im ATmega8. . . . .	52
5.2.	Steuerregister für Timer0. . . . .	52
5.3.	Zählregister TCNT0. . . . .	52
5.4.	Interruptsteuerung für Timer0. . . . .	53
5.5.	Funktionsweise, eines Watchdog-Timers. . . . .	54
5.6.	Steuerregister des Watchdog-Timers. . . . .	54
5.7.	Pulsweitenmodulation. . . . .	55

5.8. FastPWM - Modus. . . . .	56
5.9. Steuerregister TCCR1. . . . .	57
5.10. Waveform Generation Modes. . . . .	57
5.11. Compare Output Mode. . . . .	58
5.12. Clock Select Bit-Beschreibung. . . . .	58
5.13. Wichtige Register des TIMER1. . . . .	59
6.1. Addition und Subtraktion vorzeichenloser Zahlen. . . . .	62
6.2. Addition und Subtraktion ganzer, vorzeichenbehafteter Zahlen. . . . .	64
6.3. Gleitkommadarstellung nach IEEE754. . . . .	65
6.4. Status-Register SREG. . . . .	67
7.1. Signalklassen. . . . .	69
7.2. Fehler bei der A/D-Wandlung. . . . .	71
7.3. ADC Control and Status Register ADCSRA. . . . .	73
7.4. ADMUX Register. . . . .	73
7.5. ADC Data Register. . . . .	74
8.1. Grundgerüst einer H-Brücke. . . . .	75
8.2. H-Brücke bei verschiedenen Drehrichtungen. . . . .	76
8.3. H-Brücke in weiteren Konfigurationen. . . . .	76
8.4. Bipolartransistor. . . . .	77
8.5. Metall Oxid Feldeffekt Transistor (MOSFET). . . . .	77
8.6. Schaltung einer H-Brücke mit MOSFETs und Freilaufdioden. . . . .	78
8.7. Typische H-Brücken Chips. . . . .	79
8.8. Signalfluss zwischen Mikrocontroller und H-Brücke. . . . .	79
9.1. Digitaler Regelkreis am Beispiel einer Ausgangsrückführung. . . . .	85
9.2. Zeitlicher Verlauf der Stellgröße $u(t)$ . . . . .	86
9.3. Ablauf des Regelzyklus . . . . .	95
10.1. Schematische Darstellung der Kodierung bei Drehgebern. . . . .	96
10.2. Schematische Darstellung eines photoelektrischen Drehgebers. . . . .	97
10.3. Signalverlauf A und B Kanal eines Drehgebers. . . . .	97
10.4. Quadraturdecoder LFLS7366R. . . . .	99
10.5. Logik-Schaltplan eines einfachen Quadraturdecoders. . . . .	99
11.1. Schematische Darstellung I <sup>2</sup> C. . . . .	100
11.2. Datentransfer auf einem I <sup>2</sup> C-Bus. . . . .	101
11.3. TWI Control Register. . . . .	103
11.4. TWI Status Register. . . . .	103
11.5. Schematische Darstellung SPI. . . . .	104
11.6. SPI Übertragung mit verschiedenen Modi. . . . .	105
11.7. SPI-Register des ATmega8. . . . .	107
12.1. Zustände und Eingangsgrößen des Kleinrobotermodells. . . . .	109
A.1. Anpassen des Batch-Skripts . . . . .	113
A.2. Erstellen eines neuen externen Werkzeugs . . . . .	114

A.3. Bearbeiten der Toolbar . . . . .	115
A.4. Fenster AddCommand . . . . .	115
A.5. Fertig integrierter Bootloader . . . . .	115
B.1. Verwendung der Fließkomma-Version des printf-Befehls. . . . .	119
C.1. Bild der Ampelplatine . . . . .	120
C.2. Layout der Ampelplatine . . . . .	121
C.3. Schaltplan der Ampelplatine . . . . .	122
C.4. Interrupt-Vektoren des ATmega8 . . . . .	124
D.1. KRT8-Roboter . . . . .	125
D.2. Schaltplan KRT8-Platine . . . . .	126
D.3. KRT8-Platine . . . . .	127
D.4. Drift des Drehratensensors und Auswirkung auf die Winkelbestimmung . .	130
D.5. Zusammenhang zwischen gemessenen Beschleunigungen und angenommene nem Winkel $\alpha_{Acc}$ . . . . .	130
D.6. Verrauschter Winkelwert für liegenden Kleinroboter . . . . .	131
D.7. Komplementärfilter (links: ursprünglich, rechts: vereinfacht)[14] . . . . .	131
D.8. Bodeplot für verschiedene Filterordnungen für die Grenzfrequenz von 1 rad/s	132
D.9. Filtercharakteristiken für IIR Filter 10. Ordnung . . . . .	132
D.10. Direct Form 1 / 2 [18] . . . . .	133
D.11. Motorkennlinie. Zu implementieren ist die Umkehrfunktion. . . . .	138
D.12. Kleinroboter, Zustände des Kleinroboters und Koordinatensystem der Sen- sormessungen . . . . .	138
D.13. DF2 Transposed des Tiefpassfilters 2. Ordnung, nach [14] . . . . .	139
D.14. Winkelmessungen und mit Komplementärfilter gefiltertes $\alpha$ . . . . .	140
D.15. Balancierregler . . . . .	142
D.16. Interrupt-Vektoren des ATmega16 . . . . .	144

# Tabellenverzeichnis

1.1.	Vergleich RISC und CISC . . . . .	5
1.2.	Datentypen in C . . . . .	12
1.3.	Binäre Grundoperationen . . . . .	15
1.4.	Arithmetische Operatoren in C . . . . .	16
1.5.	Inkrement- und Dekrement-Operatoren in C . . . . .	16
1.6.	Bitweise Operatoren in C . . . . .	16
1.7.	Logische Operatoren in C . . . . .	17
1.8.	Übersicht Zuweisungsoperatoren in C . . . . .	17
1.9.	Relationale Operatoren in C . . . . .	17
1.10.	Erweiterte ASCII-Code-Tabelle nach ISO-8859-1 (Westeuropäisch) . . . . .	18
2.1.	Ausgangsmodi eines ATmega . . . . .	25
2.2.	Beispiel I/O-Register . . . . .	25
3.1.	Bitdauer bei verschiedenen Baudaten . . . . .	31
5.1.	Overflow-Frequenz und -Periode für verschiedene Prescaler bei 4MHz . . . . .	51
11.1.	Verschiedene I <sup>2</sup> C Modi . . . . .	101
11.2.	Verschiedene SPI Modi . . . . .	106
B.1.	Wichtige Format-Specifier . . . . .	117
C.1.	Pinbelegung der LEDs . . . . .	123
C.2.	Pinbelegung der anderen Komponenten . . . . .	123
C.3.	Jumper-Konfiguration . . . . .	123
D.1.	LEDs des BTM222 Bluetooth Moduls . . . . .	127
D.2.	Status der Bluetooth Kopplung . . . . .	127
D.3.	Ansteuerung der H-Brücken . . . . .	128
D.4.	Fertige Module der Software auf dem Kleinroboter (Ordner <b>routines</b> ) . . . . .	134
D.5.	Vom Studenten zu implementierende Module . . . . .	135
D.6.	Initialwerte der Reglerverstärkungen (Einheiten: cm, s, °) . . . . .	141

# Ablauf des Praktikums

Das Praktikum umfasst insgesamt neun Tage. In den ersten fünf Tagen werden die Teilnehmer vormittags mit den benötigten Grundlagen vertraut gemacht. Zu den einzelnen Lerneinheiten werden nachmittags Übungsaufgaben bearbeitet, in denen die Teilnehmer sich in Zweiergruppen an der Umsetzung des Gelernten versuchen müssen. Ab dem vierten Tag wird an der Umsetzung zweier Regelungsaufgaben (Regelung zum Balancieren, Regler zur Trajektorienfolge) für einen mobilen Kleinroboter gearbeitet.

An Tag zwei, drei und vier wird der Stoff des Vortages anhand eines kleinen Testates (Dauer 10min) am Morgen abgefragt und bewertet. Ebenfalls werden die Übungsaufgaben der ersten Woche sowie die Programmierung des mobilen Kleinroboters in der zweiten Woche bewertet. Am Ende des Praktikums gibt es eine kleine mündliche Prüfung, in der Fragen zum eigenen Quellcode gestellt werden.

---

Tag 1	Kapitel 1: Grundlagen Mikrocontroller Kapitel 2: I/O-Ports Kapitel 3: Serielle Schnittstelle UART <i>Übung:</i> Ampelschaltung
Tag 2	Testat 1 Kapitel 4: Programmablauf und Interrupts Kapitel 5: Timer/Counter und Pulsweitenmodulation <i>Übung:</i> Stylophone
Tag 3	Testat 2 Kapitel 6: Zahlendarstellung und Arithmetik Kapitel 7: Analog/Digital-Wandlung Kapitel 8: Die H-Brücke <i>Übung:</i> Pimp-My-Stylophone
Tag 4	Testat 3 Kapitel 10: Drehgeber / Encoder Kapitel 11: Bus-Systeme Anhang D: Einführung Kleinroboter <i>Übung:</i> Anhang D.6.1 Kleinroboter – Motoransteuerung <i>Übung:</i> Anhang D.6.2 Kleinroboter – Auslesen der SPI Sensoren
Tag 5	Kapitel 9: Zeitdiskrete Regelung Anhang D.4.4: Kleinroboter – Sensordatenfusion <i>Übung:</i> Anhang D.6.3 Kleinroboter – Identifikation der Motorkennlinie <i>Übung:</i> Anhang D.6.4 Kleinroboter – Sensordatenfusion
Tag 6	<i>Übung:</i> Anhang D.6.5 Kleinroboter – PID-Kaskade zum Balancieren
Tag 7	Kapitel 12: Flachheitsbasierte Steuerung und Regelung <i>Übung:</i> Anhang D.7.1 Kleinroboter – Beobachter und Steuerung
Tag 8	<i>Übung:</i> Anhang D.7.2 Kleinroboter – Flachheitsbasierte Regelung
Tag 9	Abgabe der Übungen aus Anhang D.6 und D.7 Mündliches Testat Trajektorienregler-Wettbewerb

---

# Literaturempfehlung

Weiterführende Informationen zu Mikrocontrollern:

- U. Brinkschulte und T. Ungerer, *Mikrocontroller und Mikroprozessoren*, Springer
- G. Schmitt, *Mikrocomputertechnik mit Controllern der Atmel AVR-RISC-Familie*, Oldenbourg
- W. Trampert, *AVR-RISC Mikrocontroller*, Franzis Verlag
- K. Wüst, *Mikroprozessortechnik*, Vieweg + Teubner

Vertiefende Informationen zur digitalen Regelung:

- F. Gausch, A. Hofer und K. Schlacher, *Digitale Regelkreise*, Oldenbourg
- J. Lunze, *Regelungstechnik 2 - Mehrgrößensystem und Digitale Regelung*, Springer

## Ergänzende Links

Wissenswertes zu Embedded Systems:

- <http://www4.informatik.uni-erlangen.de/~wosch/Talks/040108HUB.pdf>

Wikipedia-Links:

- <http://de.wikipedia.org/wiki/Mikrocontroller>
- <http://de.wikipedia.org/wiki/Mikroprozessor>
- [http://de.wikipedia.org/wiki/Liste\\_von\\_Mikrocontrollern](http://de.wikipedia.org/wiki/Liste_von_Mikrocontrollern)

# Warum digital Regeln?

Zunächst soll geklärt werden, warum es überhaupt nötig sein kann einen Regler digital zu implementieren. Die digitale Ausführung eines Reglers bietet gegenüber der Analogvariante die folgenden **Vorteile**:

- Es sind komplexe Regelgesetze wie Zustandsregler, nichtlineare Regler und adaptive Regler einfacher umsetzbar.
- Die Reglerstruktur lässt sich leichter modifizieren.
- Reglerparameter können einfach und genau eingestellt werden.

Demgegenüber stehen aber die **Nachteile** einer digitalen Implementierung:

- Digitale Regler sind langsamer als analoge Lösungen (Abtastung, Verzögerungen)
- Das Regelgesetz muss zeitdiskret umgesetzt werden.
- Durch Quantisierungseffekte bei der A/D- und D/A-Wandlung können gewünschte Effekte auftreten, welche das Regelverhalten verschlechtern. 

Zur Implementierung eines digitalen Reglers bieten sich Mikrocontroller an, wie in den nächsten Kapiteln noch genauer erläutert wird.

# 1. Grundlagen Mikrocontroller

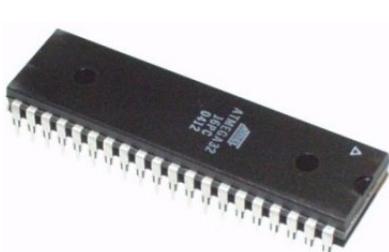


## 1.1. Was ist ein Mikrocontroller?

Mikrocontroller sind integrierte Schaltungen, d.h. Halbleiterchips, die neben einem Prozessor auch über peripherie Hardware-Funktionalität verfügen. Sie treten in unzähligen elektrischen Geräten in Gestalt von eingebetteten Systemen auf: Ihr Einsatzgebiet reicht von Armbanduhren über unterhaltungselektronische Geräte und Computer-Peripherie bis hin zu Automotive und industriellen Anwendungen wie z.B. Robotern. Ihre Entwicklung begann in den 1970er Jahren bei Intel und bewirkte nach und nach eine Revolution der Elektronik. Die Integration von RAM und ROM und immer neuer Peripherie-Einheiten auf einem einzigen Chip erweiterte dessen Einsatzgebiet im Lauf der Zeit immer weiter. Die Erfindung des wiederbeschreibbaren EEPROM-Speichers und Flash-Speichers ermöglichten einige Zeit später die Software-Entwicklung im Rapid Prototyping Verfahren und die Weiterentwicklung der Halbleitertechnik ließ die Preise der Chips bei steigender Leistungsfähigkeit immer weiter sinken, sodass sie heute bereits für deutlich weniger als 1€ zu bekommen sind und dabei komplizierte Aufgaben wahrnehmen können. Mikrocontroller existieren in den verschiedensten Bauformen und werden von zahlreichen Halbleiter-Produzenten hergestellt, darunter Atmel, Texas Instruments, Intel, Microchip, Infineon, Freescale, Toshiba und viele mehr. Und dies in Stückzahlen von vielen Milliarden jährlich!

## 1.2. Mikrocontroller, Mikroprozessoren und Mikrorechner

Ein Mikrorechner besteht aus einem (oder mehreren) Mikroprozessor, dem Speicher und den Ein-/Ausbabeschiftstellen. Ein Mikrocontroller ist im Prinzip ein Mikrorechner, welcher möglichst viele Steuerungs- und Kommunikationsaufgaben auf einem Chip lösen kann. Ein Mikroprozessor ist die Zentraleinheit (CPU, Central Processing Unit) eines Mikrorechnersystems. Die CPU beinhaltet den Prozessorkern, das Steuer- und das Rechenwerk (vgl. Abschnitt 1.3.2). Wie in Abbildung 1.2 dargestellt, bilden der Mikrocontroller und die daran angeschlossenen Peripherie-Geräte, wie z.B. Displays, zusammen ein Mikrorechnersystem.



(a) Atmel ATMEGA32



(b) Infineon PMA7110[1]



(c) Atmel ATMega1284P

Abbildung 1.1.: Mikrocontroller verschiedener Hersteller und Bauformen.

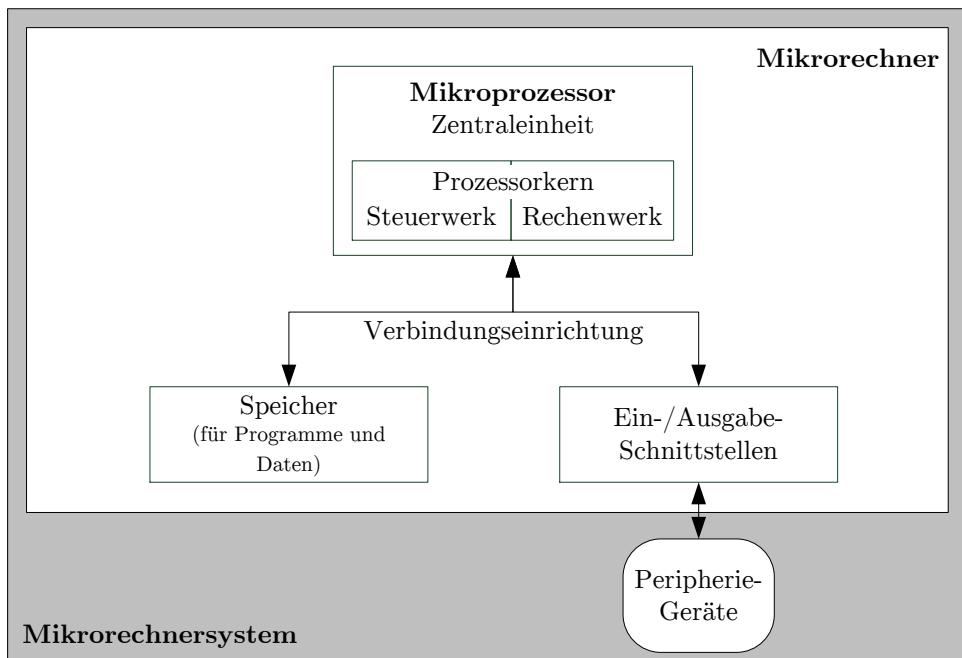


Abbildung 1.2.: Abgrenzung Mikrocontroller und Mikroprozessor [6].

### **Beispiel 1.1: Fernbedienung**

Das in Abbildung 1.3 dargestellte Schema einer Fernbedienung zeigt anschaulich die Bedeutung der oben definierten Begriffe:

- Datenverarbeitung über einen Mikrocontroller
- Tastaturmatrix als Eingabeperipherie
- IR-Leuchtdiode als Ausgabeperipherie

Idealerweise kann die Peripherie direkt an den Mikrocontroller angeschlossen werden, ohne zusätzliche Bausteine/Chips. Der Mikrocontroller stellt hierbei praktisch einen Ein-Chip-Mikrorechner mit speziell für Steuerungs- oder Kommunikationsaufgaben zugeschnittener Peripherie dar.

## **1.3. Funktionsweise und Architektur eines typischen Mikrocontrollers**

### **1.3.1. Rechnerstrukturen**

Bei der von-Neumann-Struktur liegen Befehle und Daten in einem gemeinsamen Arbeitsspeicher, der über ein Leitungssystem (Bus) mit der Prozessoreinheit (CPU) ver-

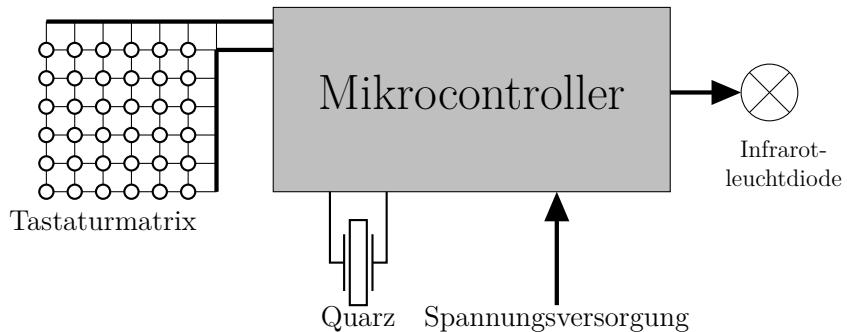


Abbildung 1.3.: Schema einer Fernbedienung mit Mikrocontroller (vgl. [6]).

bunden ist. Ein Steuerwerk setzt die Befehle, die über den (Adress-)Bus ausgelesen werden, in Steuersignale um und kontrolliert damit das Rechenwerk, das ebenfalls über den (Daten-)Bus mit Speicherwerk und den peripheren Schnittstellen kommuniziert. PCs und auch einige Mikrocontroller-Familien, wie z.B. der Serie MSP430 von Texas Instruments oder der sehr verbreitete MCS-51 von Intel (auch 8051 oder C51), basieren auf der von-Neumann-Architektur.

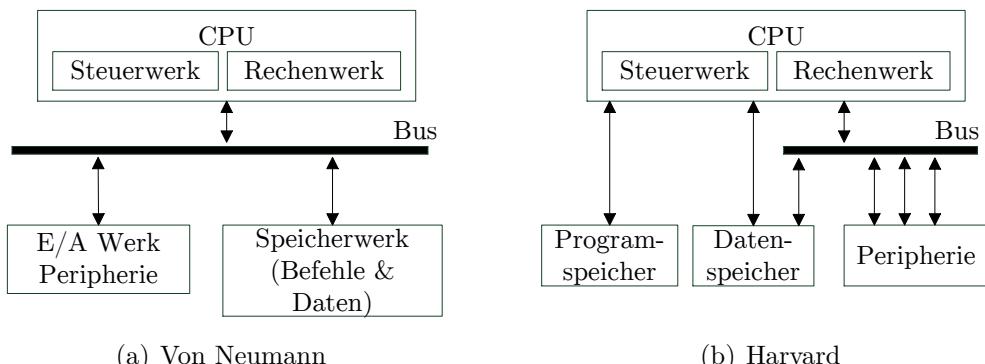


Abbildung 1.4.: Gegenüberstellung Von-Neumann- und Harvard-Architektur.

Bei den meisten Mikrocontrollern (z.B. von Atmel, Microchip) jedoch kommt die **Harvard-Struktur** zum Einsatz. Sie sieht vor, dass Befehle und Daten in getrennten Speicher- und Adressbereichen liegen. Dies ermöglicht das gleichzeitige Auslesen bzw. Schreiben von Befehlen und Daten, welches bei der von-Neumann-Architektur mindestens zwei Zyklen benötigt. Die Trennung in Daten- und Befehlsspeicher bewirkt zwar, dass der Datenspeicher nicht als Programmspeicher genutzt werden und das Programm seinen eigenen Speicherbereich nur mit Sonderbefehlen lesen kann; ein Überschreiben des Programmcodes bei Softwarefehlern ist dafür jedoch nicht möglich. Ein weiterer Vorteil der Harvard-Architektur besteht darin, dass Datenwortbreite und Befehlswortbreite unabhängig gewählt werden können.



## Ergänzende Links

What is the difference between a von Neumann architecture and a Harvard architecture?  
<http://www.arm.com/support/faqip/3738.html>

### 1.3.2. CPU: Steuer- und Rechenwerk

Die **CPU** (Central Processing Unit) ist die zentrale Komponente eines jeden Mikrocontrollers (Abb. 1.5 und 1.8). Sie arbeitet das in Maschinensprache (Binärkode) im Speicher befindliche Programm ab, führt Befehle und Berechnungen aus, verwaltet die Datenspeicher und interagiert mit den Peripherie-Einheiten des Controllers. Ihre Komponenten lassen sich zu zwei Einheiten zusammenfassen: dem **Steuerwerk** und dem **Rechenwerk** (Abb. 1.7). Erstes verfügt über einen **Befehlszähler**, der auf die Adresse des nächsten auszuführenden Befehls im Programmspeicher zeigt. Dieser wird vom Steuerwerk ausgelesen, mithilfe des Befehlsdecoders interpretiert und ausgeführt. Die Menge aller Befehle, die eine CPU abarbeiten kann, bezeichnet man als **Befehlssatz (Instruction Set)**. Dieser erlaubt eine Unterscheidung von Prozessoren in **RISC** (Reduced Instruction Set Computer) und **CISC** (Complex Instruction Set Computer). Letztere verfügen über verhältnismäßig mächtige Einzelbefehle, während erstere zugunsten eines kleineren **Decodieraufwands** im Wesentlichen auf komplizierte Befehle verzichten. Mikrocontroller weisen daher typischerweise RISC-Architektur auf. Die beiden Konzepte sind in Tabelle 1.1 noch einmal gegenübergestellt.

Tabelle 1.1.: Vergleich RISC und CISC.

RISC		CISC
<ul style="list-style-type: none"> <li>• Wenige, einfache Instruktionen</li> <li>• Befehle werden direkt über Logikschaltung ausgeführt</li> <li>• Meist nur ein Takt pro Befehl</li> <li>• Feste Länge der Befehle (typ. 5 Formate mit 32bit)</li> <li>• Viele Rechenregister (16-32)</li> <li>• Einfaches Chipdesign</li> <li>• Hohe Taktraten</li> </ul>		<ul style="list-style-type: none"> <li>• Viele, spezialisierte Instruktionen</li> <li>• Zerlegung des Befehls in einzelne Operationen („Mikrocode“)</li> <li>• Oft mehrere Takte pro Befehl notwendig</li> <li>• Unterschiedliche Befehlslänge</li> <li>• Komplexes Chipdesign</li> </ul>

Zur Ausführung eines Befehls wird dem Rechenwerk der Operationscode mitgeteilt sowie alle Operanden geladen und der **ALU (Arithmetic Logic Unit)** bereitgestellt; dies geschieht über so genannte **Register**, die einen Teil des Datenspeichers ausmachen. Die **ALU** führt die Berechnung durch und schreibt das Ergebnis in die dafür vorgesehene Adresse des Datenspeichers bzw. teilt sie dem Steuerwerk über ein gesondertes Status-

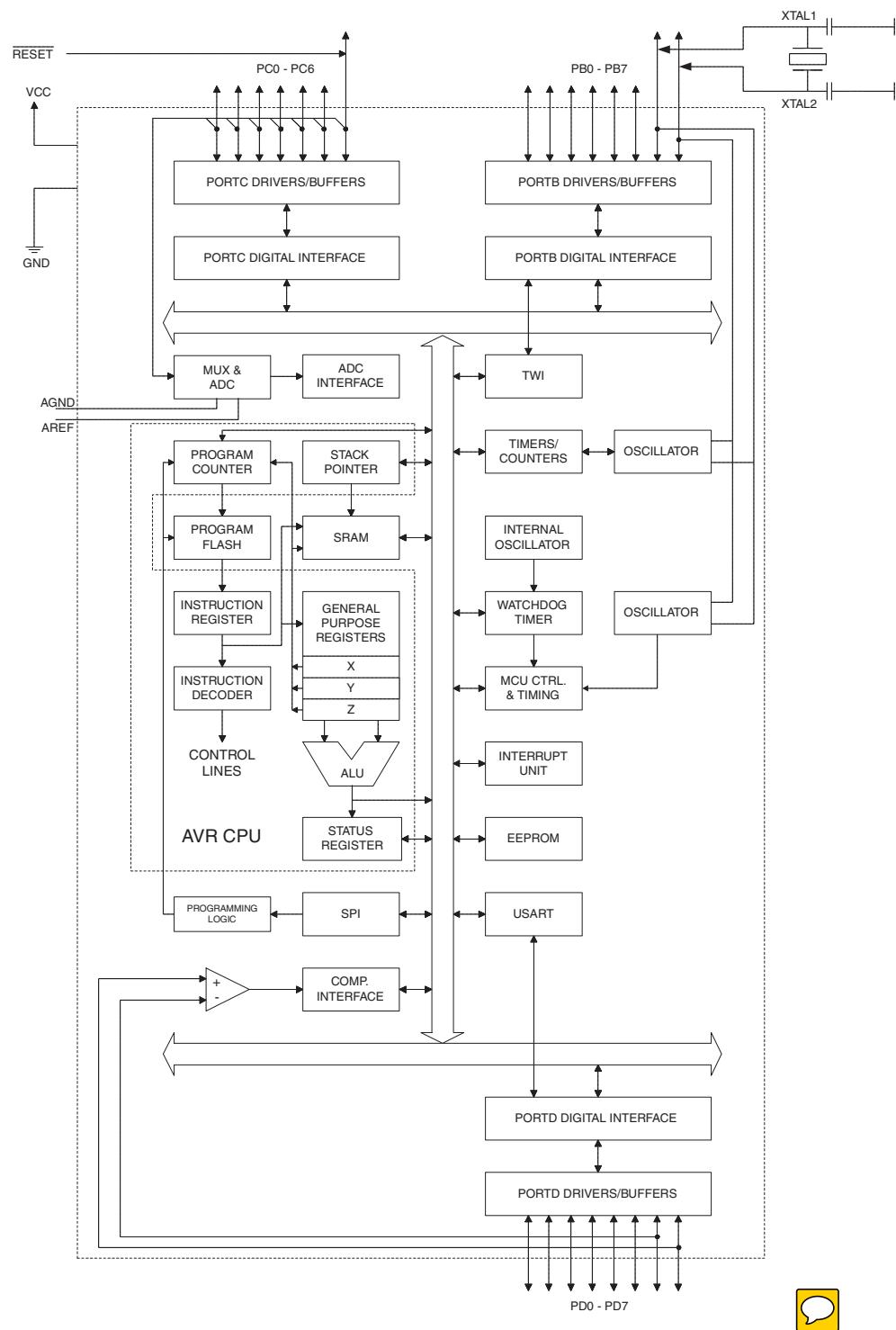


Abbildung 1.5.: Blockschaltbild ATmega8[5, S. 3].

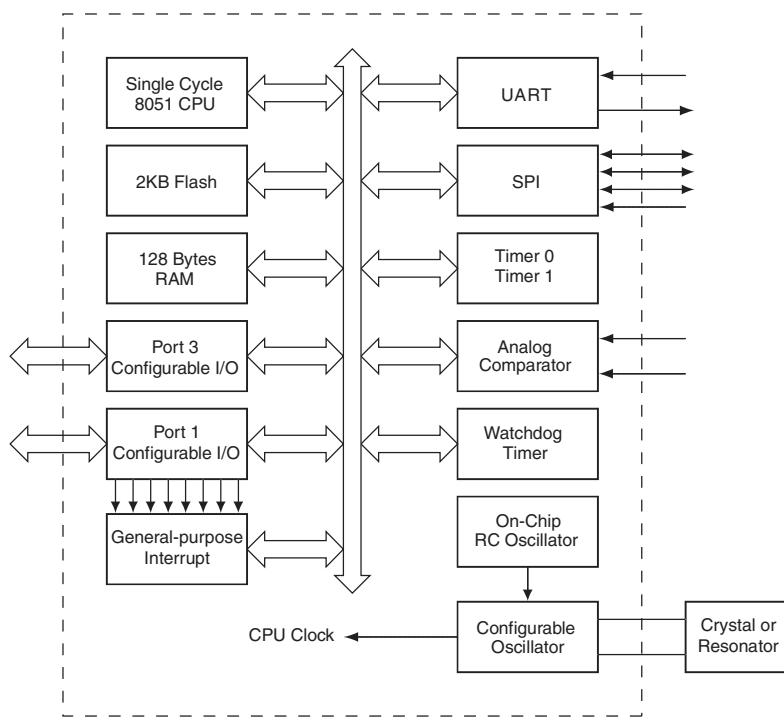


Abbildung 1.6.: Blockschaltbild AT89LP214 [2, S. 5].

Register mit. Operationen können einen oder mehrere Schritte erfordern (siehe z.B. AVR Instruction Set von Atmel).

Beispiele für Befehle sind Sprünge (bedingte oder unbedingte, Verzweigungen), Datenoperationen (Laden oder Speichern), arithmetische Befehle (Addieren, Inkrementieren, Multiplizieren), logische Befehle („oder“, „und“, „nicht“), Bit-Operationen (Shift, Bits im Statusregister lesen/schreiben) oder Mikrocontroller-Befehle (Sleep-Modus, Watchdog).

Die zeitliche Abarbeitung der Befehle wird durch einen **Takt** gesteuert, der entweder von einem **Mikrocontroller-internen Oszillator** oder einem **externen Quarz** vorgegeben wird. Die Frequenz liegt hierbei typischerweise im Megahertz-Bereich und ist entscheidend für die Geschwindigkeit, mit der ein Programm abläuft. Höhere Taktung führt meist zu höherem Stromverbrauch. Die maximale Frequenz, mit der ein Mikrocontroller betrieben werden kann, ist dem Datenblatt zu entnehmen.

Eine Besonderheit bei Mikrocontrollern besteht im Zugriff auf Peripherie-Komponenten (vgl. Abb. 1.6) mithilfe des sogenannten **Memory Mapped I/O** (eine gängige deutsche Bezeichnung existiert hierfür nicht). Die Kommunikation der CPU mit peripherer Hardware erfolgt hierzu über spezielle Register, die wie normaler Datenspeicher angesprochen (d.h. gelesen und beschrieben) werden können.

Eine Unterbrechung des linearen Programmflusses kann mithilfe von **Interrupts** erzielt werden. Falls ein bestimmtes Ereignis auftritt und eine definierte Bedingung erfüllt wird (z.B. Pegeländerung an einem Pin des Mikrocontrollers, Empfang über eine Schnittstelle, Timer-Ereignis), friert das Steuerwerk den aktuellen Zustand der CPU ein und führt anstelle des nächsten Befehls eine **ISR (Interrupt Service Routine)** aus, mit der auf das

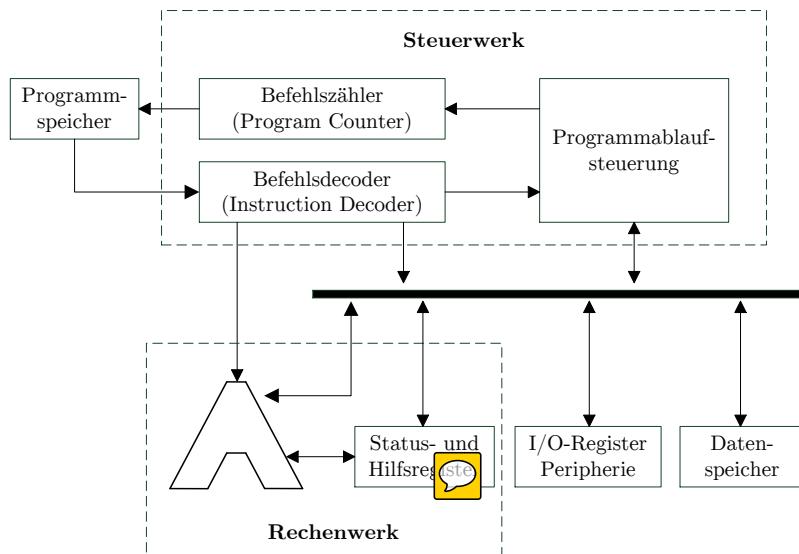


Abbildung 1.7.: Zusammenwirken von Rechen- und Steuerwerk.

Ereignis reagiert werden kann. Anschließend wird die Abarbeitung des Programms wieder aufgenommen. Interrupts können zur CPU-schonenden, schlanken Implementierung ungemein nützlich sein.

### 1.3.3. Speicheraufbau und -typen

Mikrocontroller verfügen meist über drei verschiedene Speicher, die aufgrund ihrer Speichermedien unterschiedliche Eigenschaften aufweisen. Der Programm speicher wird typischerweise als Flash-EEPROM, kurz: **Flash**, realisiert. Dabei handelt es sich um nicht-flüchtigen Speicher (d.h. die Information bleibt auch nach Entfernen der Betriebsspannung erhalten), der Information bitweise in MISFETs (Metall-Isolator-Halbleiter-Feldeffekt-Transistoren) speichert. Um den Zustand einer solchen Zelle zu ändern, bedarf es eines großen elektrischen Potentials, das aufgrund des quantenphysikalischen Tunneleffekts dazu führt, dass Ladung vom oder in das Gate des Transistor transportiert wird. Zur Programmierung des Flash-Speichers ist daher in der Regel ein **Programmieradapter (Programmer)** vornötig, ein externes Programmier tool, das das Programm über eine Programmierschnittstelle (IEEE Standard: JTAG, ISP oder proprietäre Standards der Hersteller) in den Speicher brennt. Manche Mikrocontroller sind jedoch in der Lage, einen Großteil ihres Flashes selbst zu beschreiben, was die Programmierung mittels Bootloader<sup>1</sup> ermöglicht.

Das **EEPROM** ist ebenfalls ein nicht-flüchtiger Speicher, dessen Inhalt im Gegensatz zum Flash-Speicher byte-weise beschrieben werden kann. Der Zugriff (Lesen/Schreiben) dauert spürbar länger als beim Flash. EEPROMs kommen in Mikrocontroller daher haupt-

<sup>1</sup>Ein Bootloader ist ein kleines Programm, das direkt nach dem Start des Controllers ausgeführt wird und dann das eigentliche Hauptprogramm lädt. Beim PC ermöglicht der Bootloader beispielsweise die Auswahl des zu startenden Betriebssystems.

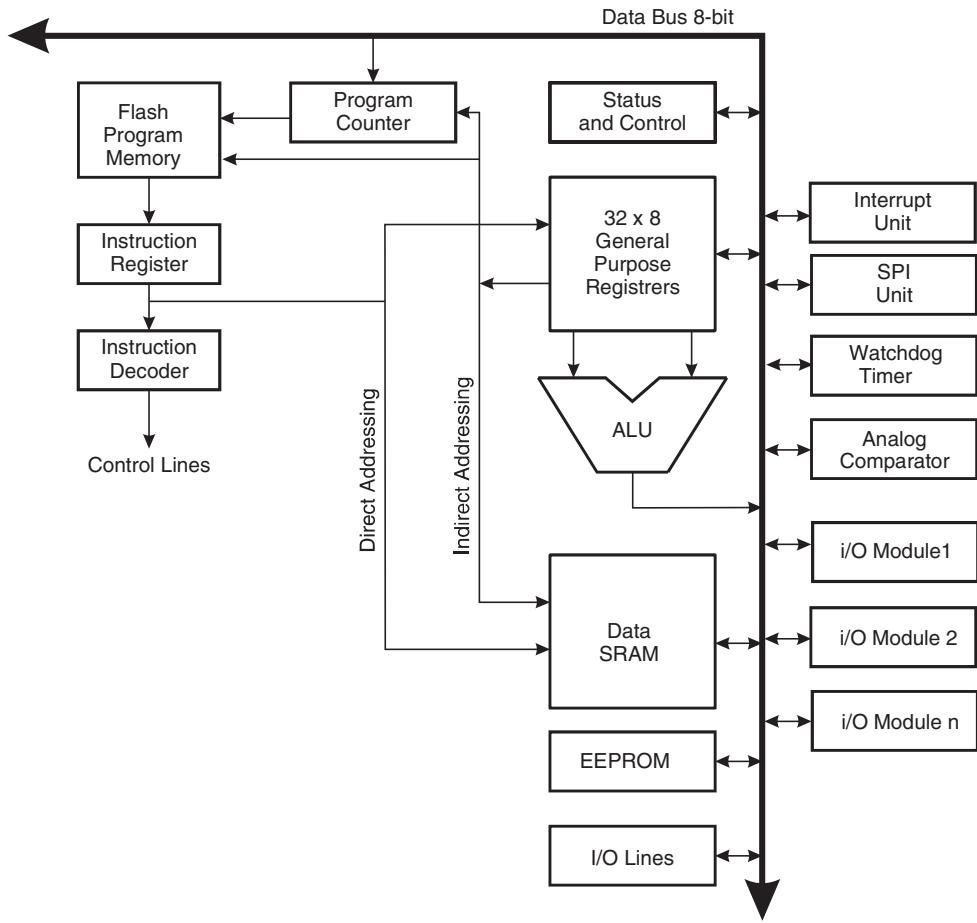


Abbildung 1.8.: CPU des ATmega8.

sächlich zum Einsatz, um kleinere Datenmengen in größeren Zeitabständen zu speichern, z.B. als Konfigurationsdaten oder Betriebsstundenzähler.

Der **SRAM** (Static Random-Access Memory) ist ein **flüchtiger (volatile)** Speichertyp, dessen Inhalt nur solange erhalten bleibt, wie eine **Betriebsspannung** anliegt. Er besteht aus Flipflop-Zellen (Kippstufen) in CMOS-Technologie und erlaubt **erheblich schnelleren Zugriff** als EEPROM-Speicher. Aus diesem Grunde kommt er bei Mikrocontrollern als **Arbeitsspeicher** und für die Register zum Einsatz.

### 1.3.4. Peripherie

Im Gegensatz zu Mikroprozessoren verfügen Mikrocontroller üblicherweise über eine Reihe peripherer Komponenten, die mittels Memory-Mapped I/O mit der CPU in Verbindung stehen. Sie können vom Programm meist **dynamisch aktiviert und deaktiviert** werden und sehr unterschiedliche Aufgaben erfüllen. Sie bestehen aus speziellen „in Hardware gegossenen“ Schaltungen, die mit dem Systemtakt versorgt werden und ihre Aufgabe fast unabhängig vom Prozessor erledigen. Nur ein Minimum an Information wird mit diesem ausgetauscht, beispielsweise zum An-/Abschalten und Konfigurieren der Komponente oder

zum Datentransfer. Dies ermöglicht eine dramatische Einsparung von Rechenkapazität, da beispielsweise ein Software-Timer die CPU zu 100% in einer Warteschleife auslasten würde, während ein interrupt-gestützter, peripherer Counter nach Ablauf des gegebenen Zeitintervalls den Prozessor informiert und dieser in der Zwischenzeit anderen Aufgaben nachgehen kann. Ein anderes Beispiel wäre eine serielle Kommunikations-Schnittstelle wie der UART: Eine Software-Implementierung würde die CPU voll auslasten, die periphere Hardware belastet den Prozessor dagegen fast gar nicht.

Die Art der enthaltenen Peripherie ist ein wesentliches Charakteristikum eines Mikrocontrollers und hat auf dessen Preis, Bauform (Größe), Stromverbrauch usw. Auswirkung. Aufgrund der großen Variation von Chips mit gleichem Prozessor aber unterschiedlicher Peripherie bieten die Hersteller auf ihren Websites Suchmasken an, mit denen der passende Controller gefunden werden kann. Die Auswahl des Controllers erfolgt beispielsweise anhand

- der geforderten Rechenleistung
- des Speicherbedarfs
- der benötigter Peripherie
- von Energie- und Anschlussdaten
- des geforderten Temperaturbereichs
- ...

Beispielhafte Peripheriekomponenten sind:

- **I/O-Ports**: Pins des Mikrocontrollers werden als digitale Ein- oder Ausgänge verwendet, d.h. die anliegende Spannung wird ausgelesen (digitaler Schmitt-Trigger) oder ein Logiksignal (Masse oder Versorgungsspannung) angelegt. Oft können Pins eines Mikrocontrollers wahlweise als Standard-I/O-Port verwendet werden oder nehmen Sonderfunktionen wahr (siehe folgende Beispiele). Die Pins sind meist zu je acht als Port organisiert und werden über gemeinsame Register angesteuert.
- **Externer Interrupt**: Pegeländerung am Pin kann als Interruptbedingung verwendet werden.
- **Timer und Counter**: können Rechtecksignale als Pulsweitenmodulation (PWM) erzeugen, Flanken an einem Pin zählen, regelmäßige Ereignisse auslösen, Zeitmessung durchführen uvm.
- **AD/DA-Wandler**: (Analog→Digital, Digital→Analog) dienen zur Übersetzung eines analogen Signals in eine Maschinenzahl bzw. zur Ausgabe einer bestimmten Spannung an einem Pin.
- **Komparatoren**: vergleichen die analogen Spannungen an zwei Pins.
- **U(S)ART**: (Universal (A)synchronous Receiver Transmitter). Die altbekannte serielle Schnittstelle (COM) RS232, die auch bei Modems zum Einsatz kam.

- **I<sup>2</sup>C-Interface** (Inter-Integrated Circuit, „I Square C“, „I Quadrat C“), auch TWI (Two Wire Interface). Ein serieller Datenbus mit zwei Leitungen, der typischerweise zur Kommunikation auf kurzen Strecken verwendet wird, z.B. zwischen zwei Mikrocontrollern auf einer Platine oder zwischen Mikrocontroller und Sensor.
- **SPI** (Serial Peripheral Interface) ist wie I<sup>2</sup>C ein Datenbus der verwendet wird um Peripherie-Geräte (z.B. Sensoren, externe Speicher) mit dem Mikrocontroller zu verbinden.
- **LCD-Controller** dienen zur Ansteuerung eines **LCD** Displays, mit dem Informationen angezeigt werden können.
- **USB-Controller** zum direkten Anschluss des Mikrocontrollers an den USB-Port eines PC.
- **CAN-Bus** (Controller Area Network) und **LIN-Bus** (Local Interconnect Network) sind verbreitete zwei- bzw. eindrahtige Feldbus-Systeme, die z.B. im Automobil und in der Automatisierungstechnik verwendet werden.
- **Ethernet-LAN-Controller** (Local Area Network) erlauben die Integration des Mikrocontrollers in ein bestehendes Netzwerk aus PCs.
- ...

### 1.3.5. Fuse- und Lockbits

Mikrocontroller verfügen über sogenannte Fuse- und Lock-Bits, die zur Konfiguration des Mikrocontrollers dienen und von der Software nicht verändert werden können. Sie enthalten beispielsweise Informationen über die Taktquelle (interner oder externer Oszillator), über den Bootloader oder die Brown-Out-Erkennung sowie Sicherheitseinstellungen, die ein Auslesen des Codes aus dem Controller verhindern.

## 1.4. Basics aus Arithmetik und Logik

### 1.4.1. Zahlensysteme und Datentypen

Wie aus Schulzeit und Vorlesungen bekannt, existieren neben dem gebräuchlichen dezimalen Zahlensystem weitere, die für die Arbeit mit Maschinen geeigneter sind. 2-basierter **Binärkode** besteht nur aus 1 und 0 und wird in der Programmiersprache C mit vorangestelltem **0b** gekennzeichnet.<sup>2</sup> **Hexadezimalzahlen** sind 16-basiert, reichen von 0 über 9 und A bis F und werden mit **0x** markiert. **Vier binäre Ziffern codieren eine Hexadezimalziffer**. Ein Byte besteht aus 8 Bits und kann z.B. die ganzen Zahlen von -128 bis 127 oder von 0 bis 255 codieren; dafür wird eine höchstens **8-stellige binäre bzw. 2-stellige Hexadezimalzahl** benötigt.

<sup>2</sup>Es gibt 10 Typen von Studenten: Die, die das Binäre verstehen, und die, die es nicht verstehen.

Tabelle 1.2.: Datentypen in C.

Typ	Größe (Bit)	Typische Namen
Ganzzahlig	8	char, Byte/byte
	16	Word, Short/short, Integer
	32	DWord/Double Word, int, long
	64	Int64, QWord, long long, Long/long
	128	Int128, Octaword, Double Quadword
Fließkomma	32	float, single
	64	double

**Vorsicht:**

Bezeichnungen wie *short*, *integer*, *long integer* usw. sind in unterschiedlichen Programmiersprachen bisweilen verschieden belegt.

Im Umgang mit Datentypen gilt es, grundsätzlich große **Vorsicht** walten zu lassen. Wird beispielsweise eine **vorzeichenlose** Größe im Speicher abgelegt und anschließend fälschlicherweise als vorzeichenbehaftete Variable wieder ausgelesen, führt dies zur Fehlinterpretation des gespeicherten Werts. Auch der **Wertebereich eines Datentyps** muss beachtet werden, um unabsichtlichen **Überlauf** zu vermeiden. Da der Mikrocontroller über kein Betriebssystem verfügt, wird der Programmierer über derartige Fehler nicht informiert - deren Auffinden kann dann unerfreulich lange dauern.

Wie man in Tabelle 1.2 sieht, gibt es eine **große Anzahl** möglicher Bezeichnungen für verschiedene ganzzahlige Datenformate. Die Bezeichnungen **unterscheiden** sich nicht nur je **nach Systemarchitektur** (PC, Workstation, Mikrocontroller...) sondern sogar je nach eingesetztem **Betriebssystem** (Windows, Linux) und **Compiler**. Auf dem AVR-Mikrocontroller bietet sich die Verwendung der vordefinierten eindeutigen Datentypen wie

```
int8_t  
uint8_t  
int16_t  
uint16_t
```

an. Diese können über die Datei *<stdint.h>* eingebunden werden.

### 1.4.2. Grundoperationen und Verknüpfungen für ganze Zahlen

Die Grundoperationen eines Mikrocontrollers für ganze Zahlen sind in Tabelle 1.3 anhand von Beispielen dargestellt. Daneben existieren in C die Symbole \* für Multiplikation, / für Division, ! für „Nicht“ und % für „modulo“. Die einfachen Rechenoperationen im C-Code setzt der Compiler in die entsprechenden Assembler-Befehle um. Je nach Funktionsumfang (Instruction Set) des Mikroprozessors müssen einfache C-Befehle in viele Assembler-Anweisungen (Abb. 1.9) umgesetzt werden.

Die folgenden Beispiele zeigen jeweils die gleiche Operation (Addition zweier 32-Bit Zahlen) auf verschiedenen Prozessoren. Hierbei wird der vom Compiler erzeugte Maschinencode (Assembler) verglichen. Die Berechnung benötigt auf dem **8-Bit Prozessor** etwa



Mnemonics	Operands	Description	Operation	Flags	#Clocks	#Clocks XMEGA
<b>Arithmetic and Logic Instructions</b>						
ADD	Rd, Rr	Add without Carry	$Rd \leftarrow Rd + Rr$	Z,C,N,V,S,H	1	
ADC	Rd, Rr	Add with Carry	$Rd \leftarrow Rd + Rr + C$	Z,C,N,V,S,H	1	
ADIW <sup>(1)</sup>	Rd, K	Add Immediate to Word	$Rd \leftarrow Rd + 1:Rd + K$	Z,C,N,V,S	2	
SUB	Rd, Rr	Subtract without Carry	$Rd \leftarrow Rd - Rr$	Z,C,N,V,S,H	1	
SUBI	Rd, K	Subtract Immediate	$Rd \leftarrow Rd - K$	Z,C,N,V,S,H	1	
SBC	Rd, Rr	Subtract with Carry	$Rd \leftarrow Rd - Rr - C$	Z,C,N,V,S,H	1	
SBCI	Rd, K	Subtract Immediate with Carry	$Rd \leftarrow Rd - K - C$	Z,C,N,V,S,H	1	
SBIW <sup>(1)</sup>	Rd, K	Subtract Immediate from Word	$Rd + 1:Rd \leftarrow Rd + 1:Rd - K$	Z,C,N,V,S	2	
AND	Rd, Rr	Logical AND	$Rd \leftarrow Rd \bullet Rr$	Z,N,V,S	1	
ANDI	Rd, K	Logical AND with Immediate	$Rd \leftarrow Rd \bullet K$	Z,N,V,S	1	
OR	Rd, Rr	Logical OR	$Rd \leftarrow Rd \vee Rr$	Z,N,V,S	1	
ORI	Rd, K	Logical OR with Immediate	$Rd \leftarrow Rd \vee K$	Z,N,V,S	1	
EOR	Rd, Rr	Exclusive OR	$Rd \leftarrow Rd \oplus Rr$	Z,N,V,S	1	
COM	Rd	One's Complement	$Rd \leftarrow \$FF - Rd$	Z,C,N,V,S	1	
NEG	Rd	Two's Complement	$Rd \leftarrow \$00 - Rd$	Z,C,N,V,S,H	1	
SBR	Rd,K	Set Bit(s) in Register	$Rd \leftarrow Rd \vee K$	Z,N,V,S	1	
CBR	Rd,K	Clear Bit(s) in Register	$Rd \leftarrow Rd \bullet (\$FFh - K)$	Z,N,V,S	1	
INC	Rd	Increment	$Rd \leftarrow Rd + 1$	Z,N,V,S	1	
DEC	Rd	Decrement	$Rd \leftarrow Rd - 1$	Z,N,V,S	1	
TST	Rd	Test for Zero or Minus	$Rd \leftarrow Rd \bullet Rd$	Z,N,V,S	1	
CLR	Rd	Clear Register	$Rd \leftarrow Rd \oplus Rd$	Z,N,V,S	1	
SER	Rd	Set Register	$Rd \leftarrow \$FF$	None	1	
MUL <sup>(1)</sup>	Rd,Rr	Multiply Unsigned	$R1:R0 \leftarrow Rd \times Rr (UU)$	Z,C	2	
MULS <sup>(1)</sup>	Rd,Rr	Multiply Signed	$R1:R0 \leftarrow Rd \times Rr (SS)$	Z,C	2	
MULSU <sup>(1)</sup>	Rd,Rr	Multiply Signed with Unsigned	$R1:R0 \leftarrow Rd \times Rr (SU)$	Z,C	2	
FMUL <sup>(1)</sup>	Rd,Rr	Fractional Multiply Unsigned	$R1:R0 \leftarrow Rd \times Rr << 1 (UU)$	Z,C	2	
FMULS <sup>(1)</sup>	Rd,Rr	Fractional Multiply Signed	$R1:R0 \leftarrow Rd \times Rr << 1 (SS)$	Z,C	2	
FMULSU <sup>(1)</sup>	Rd,Rr	Fractional Multiply Signed with Unsigned	$R1:R0 \leftarrow Rd \times Rr << 1 (SU)$	Z,C	2	
DES	K	Data Encryption	if (H = 0) then R15:R0 ← Encrypt(R15:R0, K) else if (H = 1) then R15:R0 ← Decrypt(R15:R0, K)			1/2

Abbildung 1.9.: Auszug aus dem Instructionset der AVR-Familie [3].

4 mal so viele Instruktionen wie auf dem 32-Bit Prozessor, und ist damit entsprechend langsamer.

### Beispiel 1.2: Addition zweier 32-Bit-Zahlen auf einem 8-Bit-Mikroprozessor

Wie man sieht, werden für jede Zahl 4 Register (32 Bit) im Prozessor belegt (*ldd* Befehl ersetzt Wert in einem Register, 2 Takte). Nun müssen also auch 4 Additionen durchgeführt werden (*add* Addition ohne Übertrag, 1 Takt bzw. *adc* Addition mit Übertrag, 1 Takt). Anschließend werden die 4 Byte wieder in den Speicher geschrieben (*std* Überschreibt Wert, 2 Takte).

```

        uint32_t A = 12500;
72: 84 ed      ldi      r24 , 0xD4      ; 212
74: 90 e3      ldi      r25 , 0x30      ; 48
76: a0 e0      ldi      r26 , 0x00      ; 0
78: b0 e0      ldi      r27 , 0x00      ; 0
7a: 89 83      std      Y+1, r24      ; 0x01
    
```

```

7c: 9a 83      std    Y+2, r25      ; 0x02
7e: ab 83      std    Y+3, r26      ; 0x03
80: bc 83      std    Y+4, r27      ; 0x04
          uint32_t B = 7500;
82: 8c e4      ldi    r24, 0x4C     ; 76
84: 9d e1      ldi    r25, 0x1D     ; 29
86: a0 e0      ldi    r26, 0x00     ; 0
88: b0 e0      ldi    r27, 0x00     ; 0
8a: 8d 83      std    Y+5, r24      ; 0x05
8c: 9e 83      std    Y+6, r25      ; 0x06
8e: af 83      std    Y+7, r26      ; 0x07
90: b8 87      std    Y+8, r27      ; 0x08

          uint32_t C;
          C = A + B;
92: 29 81      ldd    r18, Y+1      ; 0x01
94: 3a 81      ldd    r19, Y+2      ; 0x02
96: 4b 81      ldd    r20, Y+3      ; 0x03
98: 5c 81      ldd    r21, Y+4      ; 0x04
9a: 8d 81      ldd    r24, Y+5      ; 0x05
9c: 9e 81      ldd    r25, Y+6      ; 0x06
9e: af 81      ldd    r26, Y+7      ; 0x07
a0: b8 85      ldd    r27, Y+8      ; 0x08
a2: 82 0f      add    r24, r18
a4: 93 1f      adc    r25, r19
a6: a4 1f      adc    r26, r20
a8: b5 1f      adc    r27, r21
aa: 89 87      std    Y+9, r24      ; 0x09
ac: 9a 87      std    Y+10, r25     ; 0x0a
ae: ab 87      std    Y+11, r26     ; 0x0b
b0: bc 87      std    Y+12, r27     ; 0x0c

```

**Beispiel 1.3:** Addition zweier 32-Bit Zahlen auf einem 32-Bit Mikroprozessor

Auf einem 32-Bit Prozessor kann die gleiche Rechnung mit lediglich 8 Instruktionen durchgeführt werden, da die 32-Bit Zahlen nicht auf mehrere Register aufgeteilt werden müssen. Der prinzipielle Ablauf der Rechnung ist jedoch der gleiche.

```

          uint32_t A = 12500;
80000114:   e0 68 30 d4      mov    r8, 12500
80000118:   ef 48 ff f4      st.w  r7[-12], r8
          uint32_t B = 7500;
8000011c:   e0 68 1d 4c      mov    r8, 7500
80000120:   ef 48 ff f8      st.w  r7[-8], r8

```

	uint32_t C;	
	C = A + B;	
80000124:	ee f9 ff f4	ld.w r9, r7[-12]
80000128:	ee f8 ff f8	ld.w r8, r7[-8]
8000012c:	f2 08 00 08	add r8, r9, r8
80000130:	ef 48 ff fc	st.w r7[-4], r8

Tabelle 1.3.: Binäre Grundoperationen.

+	<b>Addition</b> Carry-Bit zeigt ggf. Überlauf an	$  \begin{array}{r}  1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\  + & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\  \hline  C = 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1  \end{array}  $
-	<b>Subtraktion</b> Carry-Bit zeigt ggf. Unterlauf an	$  \begin{array}{r}  0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\  - & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\  \hline  C = 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1  \end{array}  $
&	<b>Bitweise UND</b>	$  \begin{array}{r}  0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\  \& 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\  \hline  0 & 0 & 1 & 0 & 0 & 1 & 1 & 0  \end{array}  $
	<b>Bitweise ODER</b>	$  \begin{array}{r}  0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\    1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\  \hline  1 & 1 & 1 & 0 & 0 & 1 & 1 & 1  \end{array}  $
^	<b>Bitweise exklusives ODER (XOR)</b>	$  \begin{array}{r}  0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\  ^ 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\  \hline  1 & 1 & 0 & 0 & 0 & 0 & 0 & 1  \end{array}  $
~	<b>Bitweise Negierung</b>	$  \begin{array}{r}  \sim 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\  \hline  0 & 1 & 0 & 1 & 1 & 0 & 0 & 0  \end{array}  $
<<	<b>Links-Shift</b> Carry-Bit zeigt ggf. 9. bit an	$  \begin{array}{r}  1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & << 2 \\  \hline  1 & 0 & 0 & 1 & 1 & 1 & 0 & 0  \end{array}  $
>>	<b>Rechts-Shift</b> Carry-Bit zeigt ggf. 9. bit an	$  \begin{array}{r}  1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & >> 2 \\  \hline  0 & 0 & 1 & 0 & 1 & 0 & 0 & 1  \end{array}  $

### 1.4.3. Einschub: Operatoren in C

Wie jede Programmiersprache bietet C verschiedene Operatoren an, um Variablen zu manipulieren. Operatoren lassen sich in verschiedene Typen gliedern:

**Arithmetische Operatoren** beschreiben einfache mathematische Operationen, siehe Tab. 1.4.

Tabelle 1.4.: Arithmetische Operatoren in C

+	Addition	$a = b + c$
-	Subtraktion	$a = b - c$
*	Multiplikation	$a = b * c$
/	Division	$a = b / c$
%	Modulo	 $a = b \% c$

**Inkrement-Operatoren** erhöhen bzw. erniedrigen der Wert einer Variable um eine Einheit, siehe Tab. 1.5. Sie sind lediglich eine kürzere Schreibweise: Statt  $a = a + 1$  kann einfach  $a++$  geschrieben werden. In Verbindung mit einer Zuweisung erfolgt die Operation bei vorangestelltem  $++/-$  vor der Zuweisung, bei nachgestelltem Operator erst nach der Zuweisung.

Tabelle 1.5.: Inkrement- und Dekrement-Operatoren in C

++	Nachherige Werterhöhung	$a++$
++	Vorherige Werterhöhung	$++a$
--	Nachherige Werterniedrigung	$a--$
--	Vorherige Werterniedrigung	$--a$

**Bitweise Operatoren** führen bitweise Operationen durch, siehe Tab. 1.6 und Tab. 1.3. Dabei werden zwei Variablen bit für bit miteinander verarbeitet.

Tabelle 1.6.: Bitweise Operatoren in C

&	bitweises AND	$a = b \& c$
	bitweises OR	$a = b   c$
^	bitweises XOR	$a = a ^ c$
«	bitweises Linksschieben	$a = b « c$
»	bitweises Rechtsschieben	$a = b » c$
~	Einerkomplement (negation)	$a = \sim b$

**Logische Operatoren** führen logische Operationen wie AND/OR-Verknüpfungen durch, siehe Tab. 1.7. Meist werden sie im Zusammenhang mit if-Abfragen benötigt.

Tabelle 1.7.: Logische Operatoren in C

<code>&amp;&amp;</code>	logisches AND	if ( <code>a &amp;&amp; b</code> )...
<code>  </code>	logisches OR	if ( <code>a    b</code> )...
<code>!</code>	logisches NOT	if ( <code>!a</code> )...

**Zuweisungsoperatoren** weisen einer Variable einen Wert zu. Sie können wie in Tab. 1.8 gezeigt direkt mit anderen Operatoren verknüpft werden. So kann z.B. `a = a + b` auch einfach als `a += b` geschrieben werden. Diese Form der kombinierten Zuweisung funktioniert mit allen arithmetischen und binären Operatoren.

Tabelle 1.8.: Übersicht Zuweisungsoperatoren in C

<code>=</code>	Zuweisung	<code>a = b</code>
<code>+=</code>	Addition und Zuweisung	<code>a += b</code>
<code>-=</code>	Subtraktion und Zuweisung	<code>a -= b</code>
<code>*=</code>	Multiplikation und Zuweisung	<code>a *= b</code>
<code>&amp;=</code>	Bitweises AND und Zuweisung	<code>a &amp;= b</code>
<code> =</code>	Bitweises OR und Zuweisung	<code>a  = b</code>
<code>^=</code>	Bitweises XOR und Zuweisung	<code>a ^= b</code>

**Relationale Operatoren** vergleichen zwei Variablen und geben einen logischen Ausdruck (wahr/falsch) zurück, sieh Tab. 1.9. Meist werden sie im Zusammenhang mit if-Abfragen benötigt. Der Operator für Gleichheit (`==`) in Verbindung mit floating point Variablen sollte nur mit Bedacht gewählt werden.

Tabelle 1.9.: Relationale Operatoren in C

<code>&lt;</code>	Kleiner	if ( <code>a &lt; b</code> )...
<code>&lt;=</code>	Kleiner gleich	if ( <code>a &lt;= b</code> )...
<code>&gt;</code>	Größer	if ( <code>a &gt; b</code> )...
<code>&gt;=</code>	Größer gleich	if ( <code>a &gt;= b</code> )...
<code>==</code>	Gleich	if ( <code>a == b</code> )...
<code>!=</code>	Ungleich	if ( <code>a != b</code> )...

#### 1.4.4. Einschub: ASCII-Code

Für die Ein- oder Ausgabe von Text müssen die **einzelnen Zeichen binär kodiert** werden. Meist wird hierzu der **ASCII-Code** (American Standard Code for Information Interchange) verwendet, der bis zu 256 verschiedene Zeichen in einem Byte kodiert. Die

ASCII-Kodierung nach ISO-8859-1 für Westeuropäische Zeichen ist in Tabelle 1.10) dargestellt.

**Beispiel 1.4: ASCII Code für den Buchstaben A**

Um herauszufinden welche Zahl dem Buchstaben A entspricht genügt ein Blick in Tabelle 1.10. Aus der ersten Spalte bzw. Zeile ist die entsprechende Zahl im Hexadezimalsystem abzulesen. Für A liest man also die  $41_{16}$  aus der Tabelle, was im Dezimalsystem der  $4 * 16 + 1 = 65_{10}$  entspricht.

Tabelle 1.10.: Erweiterte ASCII-Code-Tabelle nach ISO-8859-1 (Westeuropäisch).

	.0	.1	.2	.3	.4	.5	.6	.7	.8	.9	.A	.B	.C	.D	.E	.F
0..	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1..	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2..	SP	!	"	#	\$	%	&	,	(	)	*	+	,	-	.	/
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
8..	€	,	f	"	...	†	‡	^	%	š	č	œ	ž	ž	ÿ	
9..	,	,	"	"	•	-	—	~	TM	š	›	œ	ž	ž	ÿ	
A..	í	€	£	¤	¥	¦	§	„	©	¤	«	¬	SHY	®		
B..	°	±	2	3	,	µ	¶	.	1	º	»	¼	½	¾	¸	
C..	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D..	Ð	Ñ	Ó	Ó	Ó	Ó	Ó	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E..	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F..	ð	ñ	ò	ó	ô	ö	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

## 1.5. Minimalschaltung, Bauformen und Anschlüsse

Zum Betrieb eines Mikrocontrollers benötigt man in minimaler Konfiguration nichts weiter als eine Stromversorgung. Dabei ist unbedingt auf eine bestmögliche **Stabilisierung der Versorgungsspannung** zu achten: Mikrocontroller, Sensoren und andere elektronische Komponenten können sehr empfindlich auf **Spannungsschwankungen** reagieren. Dies kann von verrauschten Messsignalen über Fehler in der Programmausführung bis hin zur Zerstörung der Komponente (z.B. durch Spannungsspitzen) führen. Unbedingt zu empfehlen ist daher z.B. die Anbringung eines schnellen „**Abblock-Kondensators**“ nahe an den Pins eines Halbleiters. Dieser lädt sich mit der Versorgungsspannung auf und kann bei plötzlichem Ladungsbedarf die Ladung zur Verfügung stellen. Bei Schaltungen mit induktiven Lasten ist eine **Schutzschaltung gegen Überspannung** unerlässlich (z.B. in Form von Freilaufdioden). Grundsätzlich ist zu beachten, dass Mikrocontroller verschiedene Betriebsspannungen (**VCC**) erfordern - diese sind dem Datenblatt zu entnehmen und betragen typischerweise 2V bis 5V (z.B. ATmega8: 2.7-5.5V, MSP430F543: 1.8-3.6V).

Jedes Datenblatt eines Halbleiters enthält zudem eine Tabelle mit „**Absolute Maximum Ratings**“, in der die **Grenzwerte** für bestimmte Belastungen angegeben sind (z.B. der

**Absolute Maximum Ratings\***

Operating Temperature .....	-55°C to +125°C
Storage Temperature .....	-65°C to +150°C
Voltage on any Pin except <u>RESET</u> with respect to Ground .....	-0.5V to $V_{CC}+0.5V$
Voltage on <u>RESET</u> with respect to Ground.....	-0.5V to +13.0V
Maximum Operating Voltage .....	6.0V
DC Current per I/O Pin .....	40.0mA
DC Current $V_{CC}$ and GND Pins.....	300.0mA

\*NOTICE: Stresses beyond those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only; functional operation of the device at these or other conditions beyond those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

Abbildung 1.10.: Maximum Ratings ATmega8 [5].

Betriebsspannung), außerhalb derer eine Beschädigung des Bauteils nicht ausgeschlossen werden kann. Diese Werte sind unbedingt zu beherzigen.

Besonders wichtig ist hierbei die maximale Spannung „Voltage on any pin“. Beim ATmega8 beispielsweise, darf die Spannung an keinem Pin größer sein als die Versorgungsspannung  $V_{CC}$  plus 0,5 V, da sonst der Mikrocontroller beschädigt wird (vgl. Abb. 1.10). Außerdem ist zu beachten, dass die Summe der Einzelströme an verschiedenen Pins kleiner sein muss als der maximal zulässige Strom der über den Versorgungs-Pin  $V_{CC}$  in den Chip fließt.

**Beispiel 1.5: Maximum ratings**

Wird der ATmega8 mit  $V_{CC} = 3,3\text{ V}$  betrieben, darf keiner der Pins mit einer Spannung größer als  $V_{CC} + 0,5\text{ V} = 3,8\text{ V}$  beschalten werden.

Die Beschaltung des Mikrocontrollers, also die Funktionen der einzelnen Pins, ist ebenfalls dem Datenblatt zu entnehmen; sie variiert innerhalb einer Mikrocontroller-Serie, gleiche **Chips** können sogar in **unterschiedlicher Bauform** vorliegen:

- **DIL bzw. DIP** (Dual In-Line Package): längliche und große Ausführung mit zwei Reihen von Pins in Drucksteckmontage. Industrieller Einsatz aufgrund der Baugröße und der hohen Montagekosten sehr selten, unter Bastlern jedoch aufgrund der Austauschbarkeit mithilfe von Fassungen und der leichten Verarbeitung (Rastermaß 2,54mm einfach zu löten) beliebt.
- **SO(IC)** (Small Outline Integrated Circuit): etwa halb so groß wie DIL und geeignet zur SMD (Surface Mounted Device) Verarbeitung. RM 1,27mm.
- **QFP** (Quad Flat Package): 32-200 Pins im RM 0,4 bis 1,0mm
- **MLF** (Micro Lead Frame): Bis zu 164 Pins, RM 0,4 bis 1,0mm. Die genauen Abmessungen der Bauform finden sich im Datenblatt.

Des weiteren weisen Mikrocontroller-Schaltungen für gewöhnlich einen Quarz als Taktgeber auf, da der interne Oszillator bei manchen Controllern nicht die erforderliche Präzision oder Frequenz besitzt.

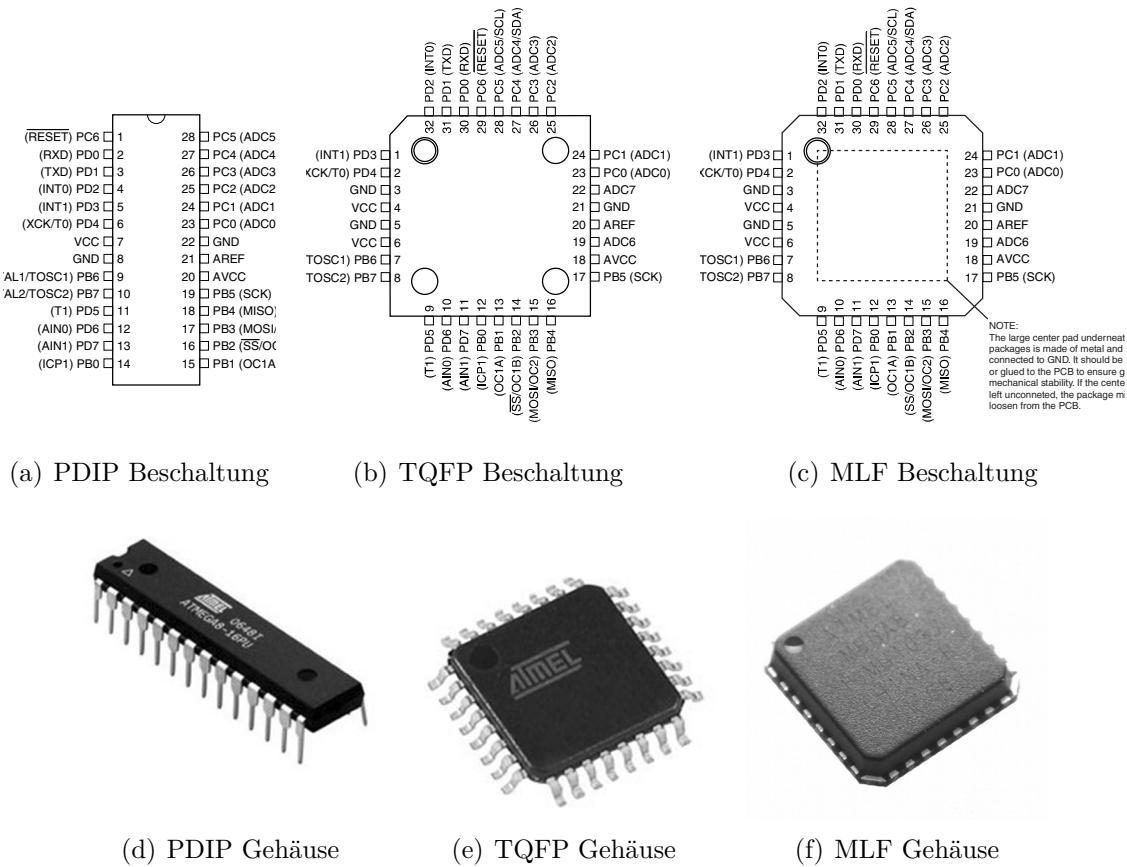


Abbildung 1.11.: ATmega8 in verschiedenen Bauformen [5, S. 2].

Mikrocontroller werden im Endprodukt meist auf **individuellen Schaltungen** eingesetzt. Zum Testen eines Mikrocontrollers und zur Vorentwicklung werden jedoch meist Testschaltungen aufgebaut bzw. **Experimentierboards** verwendet. Experimentier- und Evaluationsboards existieren von fast allen Halbleiterherstellern und sind bereits ab ca. 15€ erhältlich. Im Praktikum wird eine Ampelplatine sowie die Platine des Kleinroboters verwendet, welche beide am Lehrstuhl entwickelt wurden.



## 1.6. Programmierung, Flashen und Debugging

### 1.6.1. Schnittstellen

Die Programmierung der Mikrocontroller kann über diverse Schnittstellen erfolgen. Da es keinen genormten Standard gibt bzw. dieser von allen Herstellern unterschiedlich umgesetzt wird, wird zum Flashen und Debuggen in der Regel eine herstellerspezifische Interface-Hardware benötigt.

Eine Möglichkeit zur Programmierung ist das **ISP-Verfahren (In System Programming)**, das den Vorteil besitzt, Chips direkt in der Schaltung, in der sie zum Einsatz kommen, beschrieben werden können. Die Kommunikation mit dem Mikrocontroller erfolgt über **zwei Datenleitungen** (MISO: Master In Slave Out, MOSI: Master Out Slave In) und eine **Taktleitung** (SCK: Serial Clock); sie kann nur stattfinden, solange der Mikrocon-

troller über die Reset-Leitung angehalten wird. Außerdem können zusätzlich auch Masse und die Versorgungsspannung mit dem ISP-Interface zur Verfügung gestellt werden. In der Massenproduktion wird meist über ISP das Programm aufgespielt.

Im Praktikum wird die Programmierung über einen sog. **Bootloader** vorgenommen. Mit dem Bootloader, welcher auf dem Mikrocontroller installiert ist, kann der Mikrocontroller sich selbst ein neues Programm aufspielen, welches zuvor z.B. über die serielle Schnittstelle übertragen wurde.

Größere Mikrocontroller besitzen meistens ein **JTAG Interface**, das auch zum Debuggen verwendet werden kann. Mit diesem kann in den laufenden Mikrocontroller „geschaut“ werden, also im Betrieb schrittweise im Programmablauf fortgeschritten und dabei der Speicherinhalt betrachtet werden. Für die Entwicklung einer Mikroprozessoranwendung ist **JTAG** also bestens geeignet, während sich zur Programmierung eines fertigen Systems die **ISP** Schnittstelle anbietet.

Viele kleinere Controller (z.B. ATtiny) verfügen über die debugWIRE-Schnittstelle, da diese nur ein Pin benötigt und für Chips mit wenigen Pins damit weniger Einschränkung bedeutet. Diese Schnittstelle dient jedoch nur zum Debuggen, nicht zum Aufspielen eines Programmes auf den Mikrocontroller.

### 1.6.2. Programmieradapter und -software

**ISP-Interfaces** existieren in den **verschiedensten Bauformen** (siehe Abbildung 1.12). Im einfachsten Fall kann eine serielle Schnittstelle des PC in Kombination mit einem speziellen Programm ohne weitere Hardware verwendet werden. Eine Auflistung verschiedenster ISP-Adapter findet sich z.B. unter [http://www.mikrocontroller.net/articles/AVR\\_In\\_System\\_Programmer](http://www.mikrocontroller.net/articles/AVR_In_System_Programmer).



(a) USBProg 2.0    (b) Atmel AVRISP mkII    (c) Atmel JTAG Ice3

Abbildung 1.12.: Verschiedene Programmiergeräte.

### 1.6.3. Programmiersprachen

Für die Programmierung von Mikrocontrollern gibt es eine große Anzahl unterstützter Programmiersprachen mit den entsprechenden Compilern. Für die AVR Reihe von Atmel existieren beispielsweise neben den klassischen *Assembler* und *C* Compilern auch exotischere Varianten wie *Ada*, *BASIC*, *C++*, *Pascal* und sogar *graphische Programmierung* mittels Flussdarstellung.

Bei der Programmierung von Mikrocontrollern kommen aber hauptsächlich, die auch hier im Praktikum verwendeten Sprachen, *Assembler* und *C* zum Einsatz.

## 2. I/O-Ports

Die meisten Pins eines Mikrocontrollers besitzen die Funktionalität eines bidirektionalen Allzweck-Ports zur Ein- oder Ausgabe (**General Purpose Digital I/O Port**). Dabei handelt es sich typischerweise um Schaltungen aus mehreren Widerständen und Transistoren bzw. Operationsverstärkern, mit denen verschiedene Zustände am Pin erzeugt werden können.

### 2.1. Treiberstufen

Abbildung 2.1 zeigt schematisch eine Mikrocontroller-Treiberstufe. Die beiden Dioden auf der linken Seite dienen als Schutzschaltung gegen Überspannungen am Pin, da bei Über- bzw. Unterschreitung der Betriebsspannung bzw. Masse um mehr als ca. 0,7 V die jeweilige Diode durchbricht.

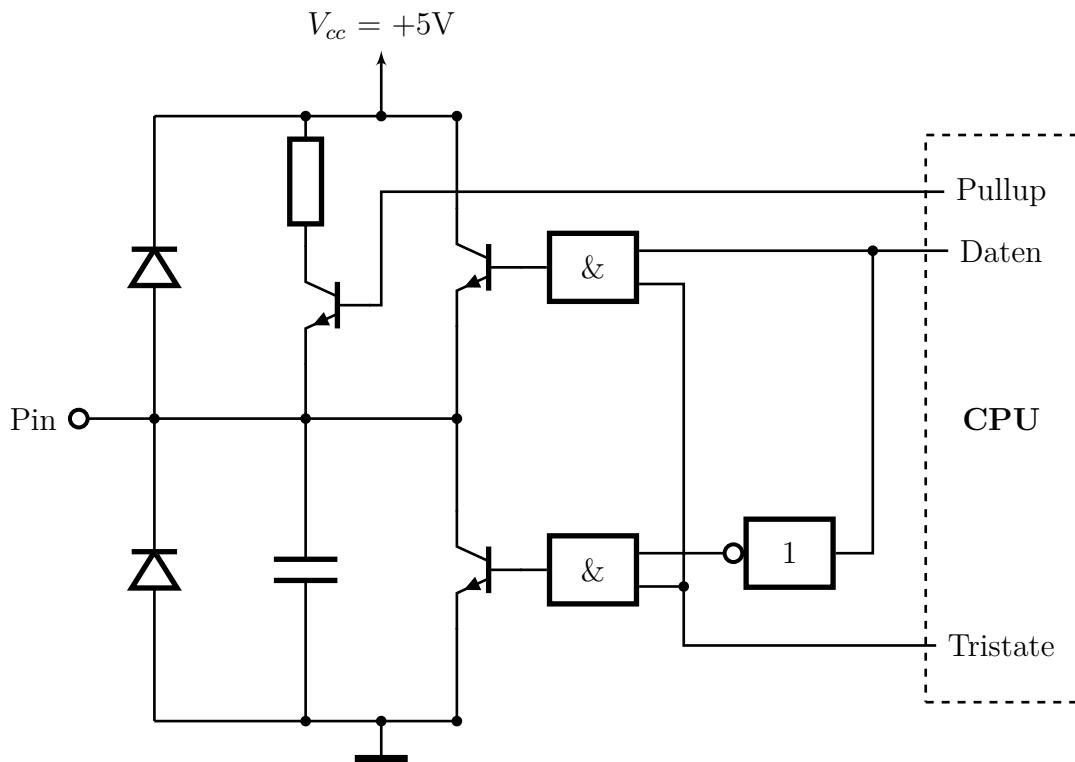


Abbildung 2.1.: Schema eines I/O-Ports.

Die verschiedenen Zustände des Pins werden erreicht, indem einer der obigen Schalter geschlossen wird. Dies geschieht durch Manipulation des entsprechenden Registers durch die CPU. Bei den oben vereinfacht dargestellten Schaltern handelt es sich in Wirklichkeit um integrierte Transistoren bzw. Operationsverstärker. In Abbildung 2.2 sind die drei in der **Transistor-Transistor-Logik (TTL)** wichtigsten Ausgangsschaltungen schematisch dargestellt.

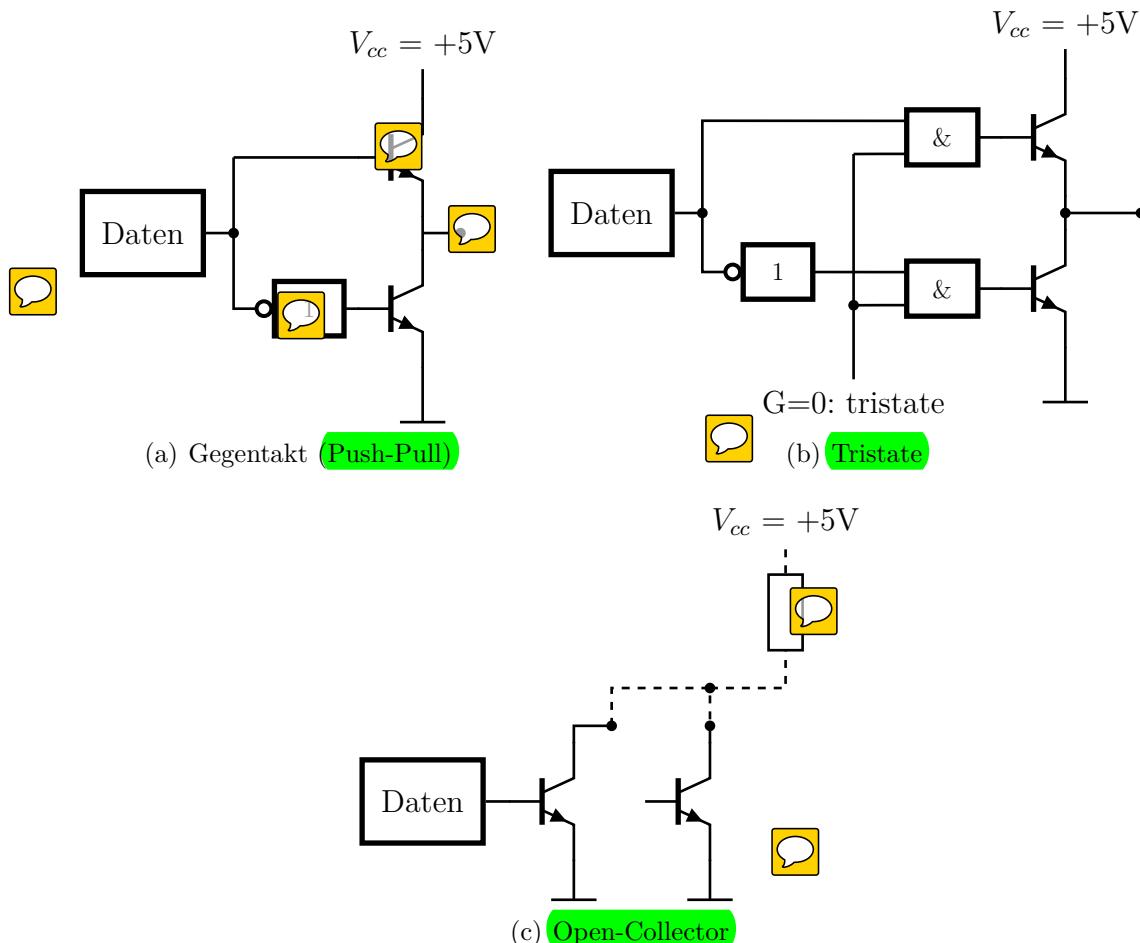


Abbildung 2.2.: Ausgangstreiber Vgl.[22, S. 281].

In der Gegentakt-Schaltung (*Push-Pull*, Abb. 2.2(a)) wird der **Ausgangspin** stets entweder auf **High** oder **Low** gezogen. Dies eignet sich für den Fall, dass der Pin als Ausgang verwendet wird, weil dann „große Ströme“ (im mA-Bereich, genauen Wert unter Absolute Maximum Ratings nachlesen und keinesfalls überschreiten!) durch den Pin fließen können. Im **Pull-Up- oder Pull-Down-Modus** wird eine **Sollspannung am Pin** erzeugt, die durch Aufbringung eines externen Signals leicht überschrieben werden kann. Daher eignet sich dieser, um den Pin als Eingang zu verwenden.

Daneben können jedoch auch alle vier Schalter offen gehalten werden; dann herrscht ein **undefinierter Zustand** am Pin, man spricht von **Tristate** oder **High-Impedance** (*High-Z*, Abb. 2.2(b)). Hierfür ist ein **zusätzliches Steuersignal** (hier *G*) nötig. Bei einem **Open-Collector-Ausgang** (Abb. 2.2(c)) hingegen wird der Transistor gegen *High* weggelassen. Die **Mikrocontroller** selber kann den Pin nur auf *Low* ziehen, für ein *High* Signal muss der Pin über einen **externen Arbeitswiderstand mit *High* verbunden** werden.

Nicht alle Mikrocontroller verfügen über alle Betriebsmodi; die genaue Bauweise unterscheidet sich nach Hersteller und Serie. Auch die Ansteuerung über Registern erfolgt auf unterschiedliche Art und Weise, was bei der Implementierung zu beachten ist.

## 2.2. I/O-Ports am AVR: DDR, PORT und PIN Register

Die Mikrocontroller der AVR-Serie von Atmel verfügen über die vier Betriebszustände *Push-Pull* nach *High* oder *Low*, *Pull-Up* und *Tristate*, deren Einstellung mithilfe der Register **DDR<sub>x</sub>** (Data Direction Register) und **PORT<sub>x</sub>** erfolgt. Die Verknüpfung der Registerinhalte mit dem Zustand des entsprechenden Pins ist in Abbildung 2.3 schematisch dargestellt.

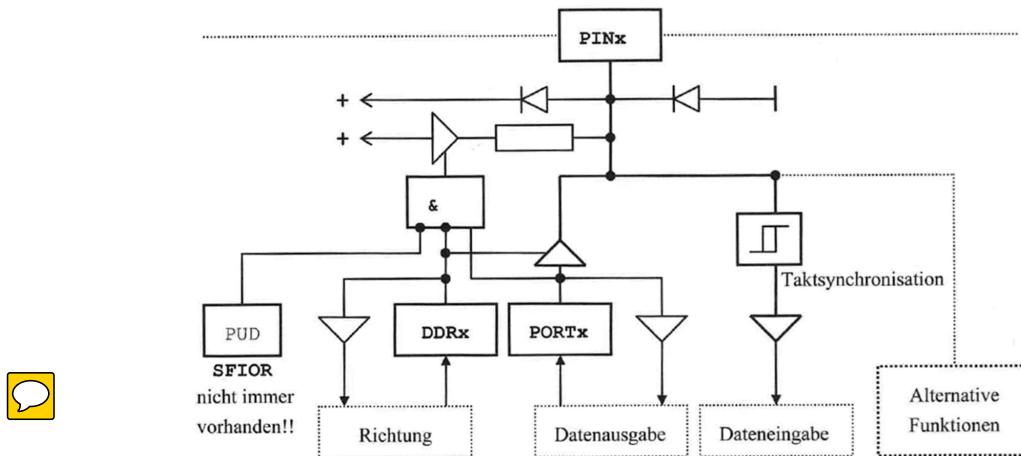


Abbildung 2.3.: Modellschaltung eines Portanschlusses [22, S. 284].

Die Pins des Mikrocontrollers sind zu den Ports A, B, C, und D gruppiert und nummeriert, PD3 steht beispielsweise für Pin 3 des Ports D (siehe Abb. 2.4). Das i-te Bit (von rechts gezählt, bei 0 beginnend) des Registers **DDR<sub>x</sub>** entscheidet darüber, ob der Pin Px<sub>i</sub> ein **Eingang** (0) oder **Ausgang** (1) ist.

The Port D Data Register – PORTD								
Bit	7	6	5	4	3	2	1	0
Read/Write	R/W							
Initial Value	0	0	0	0	0	0	0	0

The Port D Data Direction Register – DDRD								
Bit	7	6	5	4	3	2	1	0
Read/Write	R/W							
Initial Value	0	0	0	0	0	0	0	0

Abbildung 2.4.: I/O-Register von Port D des ATmega8.

### Beispiel 2.1: Ausgänge wählen mit DDRD

Hat das Register DDRD beispielsweise den Wert  $100_{10} = 64_{16} = 1100100_2$ , sind die Pins D0, D1, D3, D4 und D7 Eingänge, D2, D5 und D6 Ausgänge.

Die Funktion des Inhalts von **PORT<sub>x</sub>** hängt von den Werten in **DDR<sub>x</sub>** ab (siehe Tab. 2.1). Ist der Pin als Ausgang konfiguriert, bedeutet eine 1 in **PORT<sub>x</sub>**, dass der Pin aktiv

Tabelle 2.1.: Ausgangsmodi eines ATmega.

PORTx \ DDRx	0	1
0	Tri-State	Push-Pull Low
1	Pull-Up	Push-Pull High

auf *High* (Push-Pull-High, PPH) gehalten wird, bei 0 wird er aktiv auf *Low* gezogen (Push-Pull Low, PPL). Ist der Pin ein Eingang, so aktiviert eine 1 in PORTx den internen Pull-Up-Widerstand nach *High* (PU), eine 0 bedeutet *Tristate* (Tri). Einen internen Pull-Down-Widerstand (PD) gibt es im ATmega nicht.

### Beispiel 2.2: Beispielkonfiguration DDRD, PORTD

Setzt man  $DDRD = 0b1100100$  und  $PORTD = 0b10100110$ , so ergibt sich die in Tabelle 2.2 dargestellte Ausgangskonfiguration.

Tabelle 2.2.: Beispiel I/O-Register.

Pin-Nr	7	6	5	4	3	2	1	0
DDRD	0	1	1	0	0	1	0	0
PORTD	1	0	1	0	0	1	1	0
Zustand	PU	PPL	PPH	Tri	Tri	PPH	PU	Tri

(Tri = Tristate, PU = Pull-Up, PP H = Push-Pull High, PP L = Push-Pull Low)

Unabhängig von der Konfiguration eines Pins durch DDRx und PORTx, gibt das Register PINx Auskunft über die Spannung, die tatsächlich am Pin anliegt - es kann nur gelesen, nicht beschrieben werden und liest 1 für High, 0 für Low. Ist der Pin ein Ausgang, entspricht der Wert dem entsprechenden Eintrag von PORTx. Im Pull-Up-Modus ist der Pin *High*, sofern nicht durch externe Hardware (z.B. Taster) die Spannung nach Masse gezogen wird.

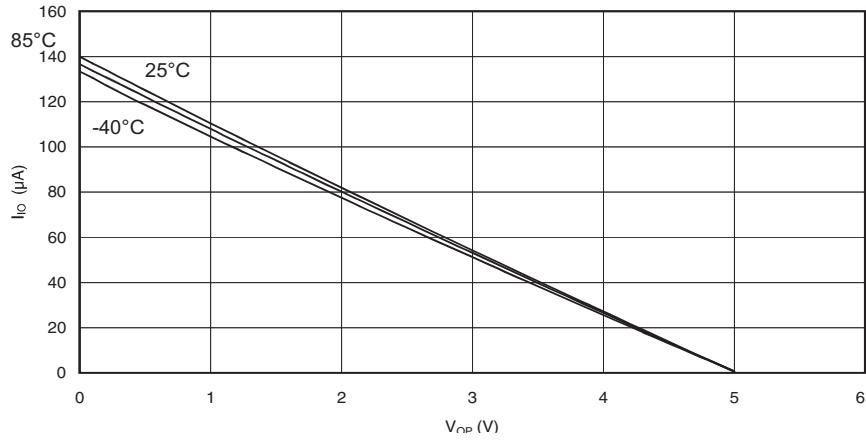
Address – PIND	Bit	7	6	5	4	3	2	1	0	PIND
Read/Write		PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	

Abbildung 2.5.: Input-Register von Port D des ATmega8.

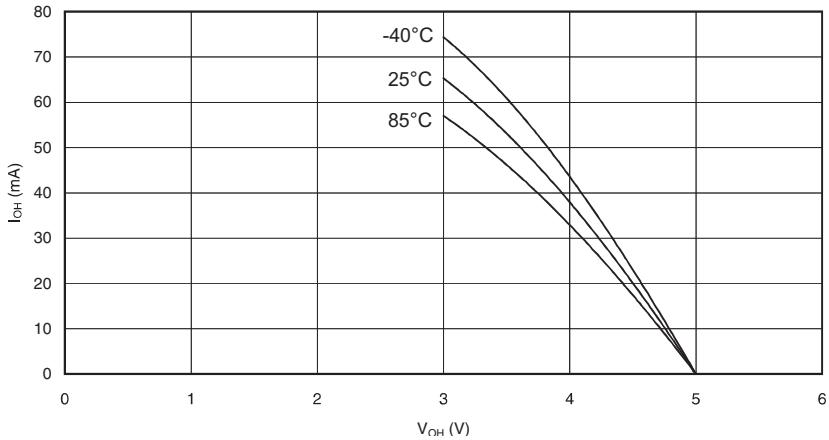
## 2.3. Leistungsfähigkeit

Wie bereits bei der Betrachtung der Absolute Maximum Ratings angesprochen, ist die elektrische Leistungsfähigkeit der Porttreiber begrenzt und ihre Überschreitung kann zur Zerstörung des Mikrocontrollers führen. Die folgenden Abbildungen aus dem Datenblatt des ATmega8A [5] zeigen exemplarisch die elektrischen Eigenschaften der Treiberstufe.

In Abbildung 2.6a) ist die Stromstärke durch einen Pin über der am Pin anliegenden Spannung für den Modus *Pull-Up* aufgetragen ( $V_{CC} = 5\text{ V}$ ).



(a) I/O Pin Pull-up Resistor Current vs. Input Voltage ( $V_{cc} = 5\text{V}$ ).



(b) I/O Pin Source Current vs. Output Voltage ( $V_{cc} = 5\text{V}$ ).

Abbildung 2.6.: Elektrische Eigenschaften der Treiberstufen des ATmega8 [5].

Wie zu erwarten, beträgt die Stromstärke bei  $5\text{ V}$  genau  $0\text{ }\mu\text{A}$  und steigt mit sinkender Spannung an, weil der Pin intern über einen hochohmigen Widerstand mit  $5\text{ V}$  verbunden ist. Aufgrund des hohen Innenwiderstands beträgt die maximale Stromstärke nur ca.  $0,13\text{ mA}$ . Abbildung 2.6b) zeigt die Stromstärke über der Pinspannung im *Push-Pull-Modus* nach *High*. Man sieht, dass auch hier die Spannung mit steigender Stromstärke abfällt, jedoch viel schneller: Beträgt die Spannung  $4,5\text{ V}$ , fließt bereits ein Strom von ca.  $20\text{ mA}$ . Der Innenwiderstand ist in diesem Betriebszustand also offensichtlich viel kleiner. Im Gegensatz zum *Pull-Up-Modus* kann hier also eine Schädigung des Controllers auftreten, da laut *Absolute Maximum Ratings* eine Stromstärke von  $40\text{ mA}$  keinesfalls überschritten werden darf. Daraus wird im Übrigen ersichtlich, dass der Mikrocontroller ohne weitere Verstärkerstufen lediglich Verbraucher mit relativ geringem Stromverbrauch ansteuern kann.

Stets gilt es, die maximal verträglichen Spannungen zu beachten, wie ein Blick in das Datenblatt einer Standard-Leuchtdiode (vgl. Abb. 2.7(a)) verrät: Typischerweise haben LEDs eine etwa parabolische Kennlinie. Unterhalb von  $1,5\text{ V}$  fließt fast kein Strom, dann

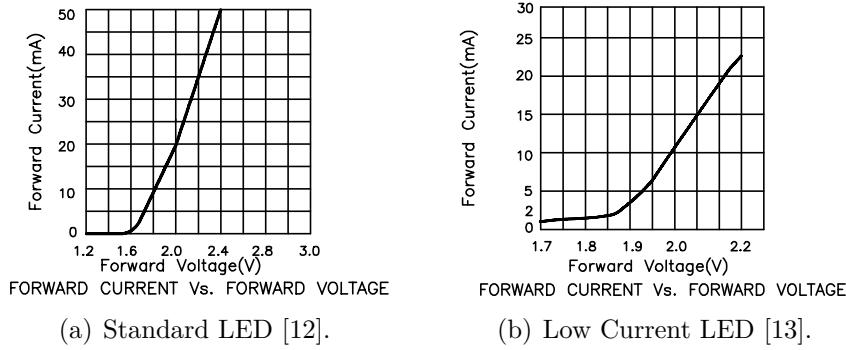


Abbildung 2.7.: Kennlinien verschiedener LEDs.

steigt die Stromstärke stark an und überschreitet bei ca. 2,3 V den laut Datenblatt zulässigen Grenzwert von 30 mA. Abhilfe kann man z.B. mit **Vorwiderständen** schaffen, die die Stromstärke limitieren.

**Vorsicht:**

LEDs dürfen niemals ohne Vorwiderstand an den Mikrocontroller angeschlossen werden!

**Beispiel 2.3: Vorwiderstand für Low Current LED**

Bei einem gewünschten Soll-Stromfluss von etwa 6 mA fallen an der LED 1,95 V ab (vgl. Abb. 2.7(b)), am Mikrocontroller-Pin liegen etwa 4,75 V an - die Differenz von 2,8 V müssen folglich am Widerstand abfallen, der also rund  $R = U/I \approx 470 \Omega$  betragen sollte.

Abbildung 2.8 zeigt den Anschluss einer LED an einen Ausgang des Mikrocontrollers. In diesem Beispiel leuchtet die LED, wenn der Pin D3 auf *Low* geschaltet ist.

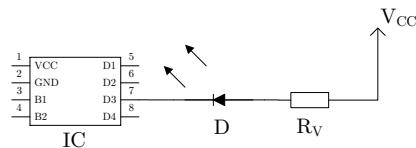


Abbildung 2.8.: Anschluss einer LED an den Mikrocontroller.

Verstärkerschaltungen, mit deren Hilfe der Mikrocontroller Motoren oder andere starke Verbraucher ansteuern kann, werden im Kapitel 8 behandelt.

Abbildung 2.9 betrifft das Verhalten des als Eingang konfigurierten Pins: Abhängig von der Versorgungsspannung wird die Pinspannung ab ca.  $1 - 2V$  als *High* erkannt, darunter als *Low*.

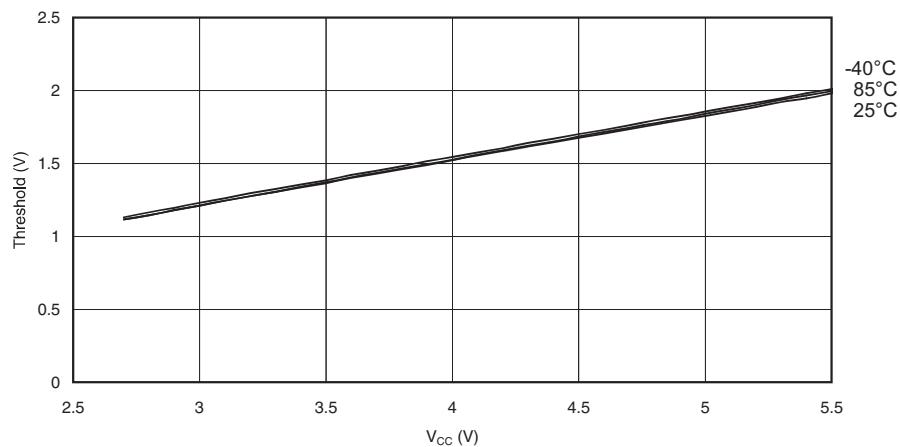


Abbildung 2.9.: I/O Pin Input Threshold Voltage vs.  $V_{cc}$  [5] .

# 3. Serielle Schnittstelle UART

UART bzw. USART (Universal Asynchronous/Synchronous Receiver Transmitter) bildet den Standard der seriellen Schnittstelle an PCs und Mikrocontrollern. Sie ist auch als Industriestandard unter dem Namen RS232 bekannt, die Kommunikation des PCs mit einem Modem erfolgte beispielsweise über die RS232-Schnittstelle des seriellen COM-Ports.

## 3.1. Spezifikation und Protokoll

Um Daten seriell, d.h. nacheinander, zu übertragen, werden sie zunächst parallel (d.h. 8 Bits = 1 Byte gleichzeitig) in ein Schieberegister kopiert und dann einzeln vom Sender im Sendetakt zum Empfänger geschoben. Dort werden sie über ein Schieberegister empfangen und können dann parallel wieder ausgelesen werden (siehe Abb. 3.1). Hierfür ist nur eine Datenleitung (gegenüber acht Leitungen bei parallelem Transfer) erforderlich, die Sendezeit verachtet sich dafür gegenüber paralleler Übertragung.

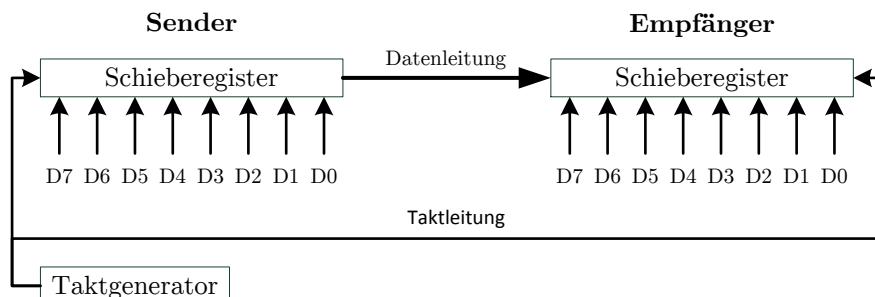
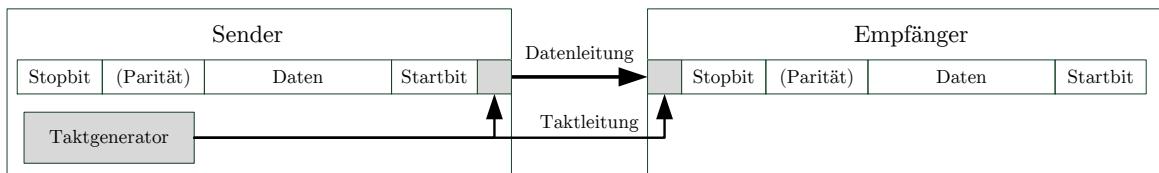


Abbildung 3.1.: Synchrone serielle Datenübertragung [22, S. 345].

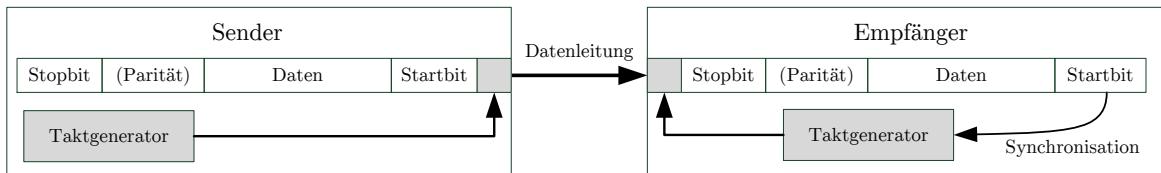
Der Unterschied zwischen *synchroner* und *asynchroner* Übertragung besteht in der Taktleitung. Im asynchronen Fall sind Sender und Empfänger nur über *Ground* und die Datenleitung miteinander verbunden - beide müssen auf die gleiche Taktfrequenz eingestellt sein und sich bei jedem Zeichen durch die Flanke eines sogenannten *Startbits* neu synchronisieren (vgl. Abb. 3.2) - die maximal zulässige Abweichung zwischen der tatsächlichen Baudrate des Senders und der des Empfängers beträgt üblicherweise etwa 2% vom Normwert. Bei synchroner Übertragung übermittelt der Sender über eine zusätzliche Leitung den Takt.

Bei der Übertragung eines Zeichens (z.B. 8 Bit) werden die Datenbits in einen Rahmen („Frame“) aus einem *Startbit* und einem *Stopbit* eingebettet. Optional kann die Übertragung des Zeichens um ein zweites *Stopbit* und/oder ein *Paritätsbit* zur Fehlererkennung erweitert werden. Für den am häufigsten verwendete Konfiguration hat sich die Kurzform **8-N-1** etabliert. Aufgeschlüsselt bedeutet dies:

- 8 Datenbits
- No Parity Bit
- 1 Stop Bit



(a) Synchronre Datenübertragung (USART)



(b) Asynchrone Datenübertragung (UART)

Abbildung 3.2.: Vergleich von USART und UART Übertragung.

Die Abkürzung kann noch um die Baudrate erweitert werden, wie zum Beispiel 9600/8-N-1.

### **Beispiel 3.1: Paritätsbit**

Ein *Paritätsbit* gibt an, ob die Anzahl der übertragenen Bits mit dem Wert 1 gerade oder ungerade ist. Der Sender berechnet die Parität der übertragenen Bitfolge und sendet diese direkt im Anschluss. Der Empfänger berechnet nun ebenfalls die Parität der empfangenen Bitfolge. Stimmt diese nicht mit dem empfangenen *Paritätsbit* überein ist bei der Übertragung ein Fehler unterlaufen. Je nach Einstellung steht ein *Paritätsbit* von 0 für eine gerade Anzahl (**even parity**) oder ungerade Anzahl (**odd parity**). Mit einem *Paritätsbit* kann genau ein falsch übertragenes Bit festgestellt werden.

7 Datenbits	Anzahl Einser	even parity	odd parity
0000000	0	0	1
1010001	3	1	0
1101001	4	0	1

Die **Geschwindigkeit** der Übertragung wird in der Einheit **baud** (Symbolrate) gemessen, was die Anzahl der Übertragungsschritte pro Sekunde beschreibt und hier der Datenübertragungsrate in **bps (Bits pro Sekunde)** entspricht.

Übliche Bitraten zeigt Tabelle 3.1. Ihr ist zu entnehmen, dass für hohe Baudraten die Bitdauer im Bereich weniger Mikrosekunden liegt. Betreibt man einen Mikrocontroller beispielsweise mit 1 MHz, so liegt dessen Schrittdauer bei 1 µs und der UART kann nur auf 8 µs oder 9 µs eingestellt werden - ein Betrieb mit 115 200 baud ist folglich nicht möglich, da beide Werte mehr als 2 % von 8,68 µs abweichen. Dies erklärt, warum Quarze mit „krummen“ Frequenzen von z.B. 18,432 MHz zum Einsatz kommen, siehe auch Beispiel 3.3.

Tabelle 3.1.: Bitdauer bei verschiedenen Baudraten.

Bitrate (baud)	Bitdauer
50	20 ms
300	3,3 ms
1.200	833 µs
2.400	417 µs
4.800	208 µs
9.600	104 µs
19.200	52 µs
38.400	26 µs
57.600	17 µs
115.200	8,68 µs
230.400	4,34 µs
460.800	2,17 µs
500.000	2,00 µs

Basierend auf der verwendeten Baudrate und des konfigurierten Übertragungsmodus (z.B. **8-N-1**) kann abgeschätzt werden, wie lange die Übertragung einer ASCII-Zeichenkette dauert:

**Beispiel 3.2:** Dauer einer Übertragung

Es sollen mittels UART 100 Zeichen (ASCII<sup>a</sup>) im Modus 9600/8N1 übertragen werden.

Bei 8N1 müssen pro Zeichen 1 Startbit + 8 Datenbits + 1 Stopbit = 10 Bits übertragen werden. Das Senden der 100 Zeichen dauert also insgesamt:

$$t = \frac{10 \frac{\text{Bits}}{\text{Zeichen}} \cdot 100 \text{Zeichen}}{9600 \frac{\text{Bits}}{\text{s}}} = 0,104 \text{s} \quad (3.1)$$

<sup>a</sup>Obwohl der ursprüngliche ASCII Datensatz nur 7bit umfasste, wird mittlerweile (und auch hier im Praktikum) der erweiterte ASCII-Datensatz mit 8bit verwendet. Dieser umfasst z.B. auch Sonderzeichen und Umlaute.

PC-seitig ist die Kommunikation über die serielle Schnittstelle beispielsweise mit den Programmen HyperTerminal, RealTerm o.a. möglich.

## 3.2. UART im AVR

### 3.2.1. Steuerregister

Im Folgenden betrachten wir die Ansteuerung der seriellen Schnittstelle in der AVR-Serie von Atmel. Die drei Register UCSRA, UCSRB und UCSRC (USART Control and Status

**USART Control and Status Register A – UCSRA**

Bit	7	6	5	4	3	2	1	0	UCSR
ReadWrite	RXC	TXC	UDRE	FE	DOR	PE	U2X	MPCM	R/W
	R	R/W	R	R	R	R	R/W	R/W	

(a) UCSRA

**Status Register B – UCSRB**

Bit	7	6	5	4	3	2	1	0	UCSR
ReadWrite	RXCIE	TXCIE	UDRIE	RXEN	TXEN	UCSZ2	RXB8	TXB8	R/W
Initial Value	0	0	0	0	0	0	0	0	

• Bit 7 – RXCIE: RX Complete Interrupt Enable

(b) UCSRB

**USART Control and Status Register C – UCSRC**

Bit	7	6	5	4	3	2	1	0	UCSR
ReadWrite	URSEL	UMSEL	UPM1	UPM0	USBS	UCSZ1	UCSZ0	UCPOL	R/W
Initial Value	1	0	0	0	0	1	1	0	

(c) UCSRC

Abbildung 3.3.: USART Control und Status Register [5, S. 148f].

Register A, B und C erlauben die Ansteuerung der Schnittstelle und das Auslesen von Statusinformationen.

### 3.2.2. Datentransfer (Empfang/Versand)

Das Register UCSRA dient im Wesentlichen zur Statusabfrage. Das Bit TXC (Transmit Complete) wird auf *high* gesetzt, wenn der Versand von Daten abgeschlossen ist. Befinden sich ungelesene Daten im Datenregister so ist das Bit RXC(Receive Complete) *high*. Das Bit UDRE (USART Data Register Empty) zeigt an, ob der Schreibpuffer leer ist und ein Zeichen zum Versand aufnehmen kann, die Bits FE (Frame Error) und PE (Parity Error), ob beim letzten Zeichenempfang ein Fehler registriert wurde, DOR (Data OverRun), ob beim Empfang ein Überlauf auftrat (der aus zwei Bytes bestehende Eingangspuffer wurde nicht ausgelesen, bevor das nächste Zeichen empfangen wurde). Mit der Wahl des Bits U2X lässt sich die Übertragungsgeschwindigkeit verdoppeln.

Mithilfe des Registers UCSRB lassen sich Receiver (RXEN-Bit) und Transmitter (TXEN-Bit) unabhängig voneinander ein- und ausschalten. RXCIE, TXCIE und UDRIE aktivieren Interrupts zu den Flags RXC, TXC und UDRE. Das Bit UCSZ2 charakterisiert gemeinsam mit UCSZ1 und UCSZ0 (im Register UCSRC) die Zahl der zu übertragenden Bits. Werden 9 Bits übertragen, so ist das Bit RXB8 für Empfang bzw. TXB8 für Versand zu aktivieren.

Beim Zugriff auf das Register UCSRC ist eine unangenehme Besonderheit zu beachten: Es wird über dieselbe Adresse angesprochen wie das Register UBRRH, obwohl die Register nicht identisch sind. Um festzulegen, welches der beiden Register gelesen oder beschrieben werden soll, muss das MSB URSEL auf *high* für UCSRC und auf *low* für UBRRH gesetzt werden. Mit dem UMSEL-Bit kann synchrone Operation eingestellt werden. Nur dann ist das Bit UCPOL von Bedeutung.

Die beiden Bits UPM1 und UPM0 (siehe Abb. 3.5(a)) legen fest, ob ein Paritätsbit

<b>UCSZ2</b>	<b>UCSZ1</b>	<b>UCSZ0</b>	<b>Character Size</b>
0	0	0	5-bit
0	0	1	6-bit
0	1	0	7-bit
0	1	1	8-bit
1	0	0	Reserved
1	0	1	Reserved
1	1	0	Reserved
1	1	1	9-bit

Abbildung 3.4.: UCSZ Bit Einstellungen [5, S. 151].

gesendet werden soll, und falls ja mit welcher Kodierung. Die Kommunikation über UART erlaubt entweder 1 oder 2 *stop bits* nach jedem übertragenen Zeichen. Dies wird über das USBS-Bit festgelegt (siehe Abb. 3.5(b)).

<b>UPM1</b>	<b>UPM0</b>	<b>Parity Mode</b>
0	0	Disabled
0	1	Reserved
1	0	Enabled, Even Parity
1	1	Enabled, Odd Parity

(a) UPM Bit Einstellungen

<b>USBS</b>	<b>Stop Bit(s)</b>
0	1-bit
1	2-bit

(b) USBS Bit Einstellungen

Abbildung 3.5.: Einstellungen für Parität und Anzahl der Stop-Bits [5, S. 151].

<b>USART Baud Rate Registers – UBRRH and UBRRHs</b>								
Bit	15	14	13	12	11	10	9	8
	URSEL	-	-	-	<b>UBRR[11:8]</b>			
					<b>UBRR[7:0]</b>			
	7	6	5	4	3	2	1	0
Read/Write	R/W	R	R	R	R/W	R/W	R/W	R/W
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0

Abbildung 3.6.: UBRR Bit Einstellungen [5, S. 152].

Die Register UBRRH (4LSB) und UBRL definieren die Baudrate nach der Formel [5, S. 132]

$$BAUD = \frac{f_{osc}}{16 \cdot (UBRR + 1)} \quad (3.2)$$

wobei  $f_{osc}$  die Taktfrequenz des Mikrocontrollers ist. Die **eigentlichen Daten** liegen im Register **UDR (UART Data Register)**, das sich die Sende- und die Empfangseinheit teilen. Beim Empfang wird das letzte Byte also aus UDR ausgelesen, zum Versand wird das Register mit einem Datenbyte beschrieben.

### Beispiel 3.3: Krumme Frequenzen

Formel 3.2 erklärt auch, warum meist Quarze mit „krummen“ Frequenzen wie z.B. 18,432 MHz verbaut werden: die Standard-Baudraten (siehe Tabelle 3.1) können nur dann genau eingestellt werden, wenn sich mit aus Formel 3.2 eine ganze Zahl für **UBBR** ergibt. Die durch die Beschränkung auf ganzzahlige Teiler entstehenden Fehler sind in Abbildung 3.7 beispielhaft aufgeführt. Wie man sieht, können mit den besagten 18,432 MHz fast alle Baudraten exakt gewählt werden.

Baud Rate (bps)	$f_{osc} = 16.0000\text{MHz}$				$f_{osc} = 18.4320\text{MHz}$				$f_{osc} = 20.0000\text{MHz}$			
	U2X = 0		U2X = 1		U2X = 0		U2X = 1		U2X = 0		U2X = 1	
	UBRR	Error	UBRR	Error	UBRR	Error	UBRR	Error	UBRR	Error	UBRR	Error
2400	416	-0.1%	832	0.0%	479	0.0%	959	0.0%	520	0.0%	1041	0.0%
4800	207	0.2%	416	-0.1%	239	0.0%	479	0.0%	259	0.2%	520	0.0%
9600	103	0.2%	207	0.2%	119	0.0%	239	0.0%	129	0.2%	259	0.2%
14.4k	68	0.6%	138	-0.1%	79	0.0%	159	0.0%	86	-0.2%	173	-0.2%
19.2k	51	0.2%	103	0.2%	59	0.0%	119	0.0%	64	0.2%	129	0.2%
28.8k	34	-0.8%	68	0.6%	39	0.0%	79	0.0%	42	0.9%	86	-0.2%
38.4k	25	0.2%	51	0.2%	29	0.0%	59	0.0%	32	-1.4%	64	0.2%
57.6k	16	2.1%	34	-0.8%	19	0.0%	39	0.0%	21	-1.4%	42	0.9%
76.8k	12	0.2%	25	0.2%	14	0.0%	29	0.0%	15	1.7%	32	-1.4%

Abbildung 3.7.: Beispiele für **UBRR**-Einstellungen[5, S. 156].

### 3.2.3. Polling

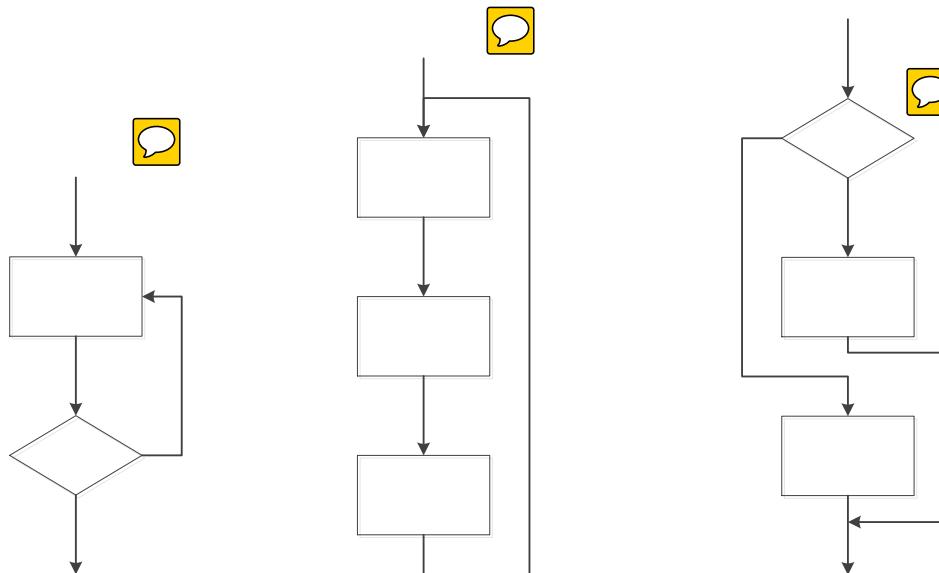
Da bei serieller Kommunikation immer nur ein Byte nach dem anderen übertragen werden kann und der ATmega nur einen sehr kleinen Sendepuffer besitzt, können zu sendende Daten meist nicht gleichzeitig an den **UART** übergeben werden, sondern müssen nach und nach vom Programm an den Transceiver übergeben werden. Neben der (in vieler Hinsicht vorteilhafteren) **interruptbasierten Steuerung**, die in Kapitel 4.3.6 behandelt wird, kann der Betriebsmodus „Polling“ zum Einsatz kommen. Dieser besteht darin, nach dem Versand eines Zeichens so lange zu warten, bis die **UART-Hardware** die Transmission abgeschlossen hat und das entsprechende Bit setzt/löscht. Dann erst wird das Programm fortgesetzt. Im Datenblatt des ATmega16 findet sich folgender beispielhafte C-Code:

```
void USART_Transmit( unsigned char data )
{
    // Wait for empty transmit buffer
    while ( !( UCSRA & (1<<UDRE) ) )
    {
        // do nothing
    }
    UDR = data;      // Put data into buffer , sends the data
}
```

# 4. Programmablauf und Interrupts

## 4.1. Sprünge und elementare Programmstrukturen

Wie wir bei der Einführung in die Funktionsweise der CPU gesehen hatten, besteht ein (sogenanntes imperatives) Programm, wie es auf einem Mikrocontroller ausgeführt wird, aus einer Folge von Befehlen, die nacheinander abgearbeitet werden. Bei Befehlen handelt es sich, wie erwähnt, z. B. um Datenoperationen, arithmetische oder logische Befehle. Jeder Befehlssatz sieht aber auch zahlreiche Sprungbefehle vor, mit deren Hilfe der Programmablauf beeinflusst werden kann.



(a) Bedingte Schleife

(b) Endlosschleife

(c) Bedingte Verzweigung

Abbildung 4.1.: Elementare Programmstrukturen.

Eine Endlosschleife lässt sich beispielsweise erzeugen, indem nach einer Reihe von Anweisungen ein Sprungbefehl kommt, der auf einen vorangegangenen Befehl zeigt. Eine Verzweigung wird generiert, indem basierend auf einer Aussage, die wahr oder falsch sein kann, auf zwei verschiedene Sprungmarken verwiesen wird. Die aus C bekannten for- und while-Schleifen lassen sich ebenfalls mithilfe von bedingten oder unbedingten Sprüngen realisieren, ebenso wie Unterprogramme (Funktionen oder Subroutinen).

## 4.2. Assembler und höhere Programmiersprachen

Mithilfe der Programmiersprache Assembler (der zugehörige Compiler heißt ebenfalls so) ist es möglich, ein Programm wie oben beschrieben Befehl für Befehl zu erzeugen, d. h.

jeder einzelne Befehl, der vom Mikrocontroller abgearbeitet werden soll, wird im Assembler explizit programmiert - anders als bei höheren Programmiersprachen, wo mehrere Prozessor-Befehle mithilfe einer einzigen Anweisungen ausgedrückt werden können (z.B. die while-Schleife). Ein Assembler-Codebeispiel aus dem Datenblatt des ATmega8 lautet:

USART\_Transmit :

```
; Wait for empty transmit buffer
sbis UCSRA,UDRE
rjmp USART_Transmit
; Copy ninth bit from r17 to TXB8
cbi UCSRB,TXB8
sbrc r17,0
sbi UCSRB,TXB8
; Put LSB data (r16) into buffer, sends the data
out UDR,r16
ret
```

Beim Assembly (Compile-Vorgang) werden diese Anweisungen anhand der Definitionen des Herstellers in den Code des Controllers überführt und dabei der (für den Menschen nicht mehr verständliche) Binärkode („Maschinensprache“) erzeugt. Aufgrund des 1-zu-1-Zusammenhangs zwischen Assembler-Befehlen und dem Prozessorcode wird auch der Assembler bisweilen umgangssprachlich als Maschinensprache bezeichnet.

Obwohl die Programmierung in Assembler zu viel effizienterem Maschinencode führen kann, als Compiler höherer Programmiersprachen ihn zu erzeugen in der Lage sind, ist sie für Anfänger mühselig und der Code anschließend schwer zu verstehen. In diesem Praktikum werden wir Assembler-Programmierung daher nicht weiterverfolgen, betonen aber, dass für Anwendungen, die hohe Ansprüche an die Effizienz des Codes stellen, Assembler dringend zu bevorzugen ist.

Dennoch kann es sehr hilfreich sein den vom C-Compiler generierten Assemblercode zu betrachten. Bei vielen Problemen hilft es zu schauen, wie der C-Code in Assembler umgesetzt wurde um vorher unerklärliche Fehler zu erklären. Im Atmel Studio findet man den vom Compiler erzeugten Assembler-Code in der (.lss)-Datei im Output Ordner des *Solution Explorers* (siehe Abb. 4.2).

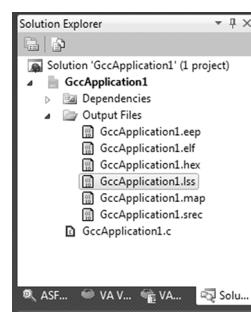


Abbildung 4.2.: Aufruf des generierten Assemblercodes.

## 4.3. Interrupts

### 4.3.1. Funktionsweise

Bei bestimmten Ereignissen kann im Prozessor ein Interrupt ausgelöst werden. Dabei wird das Programm unterbrochen und ein Unterprogramm, die sogenannte **Interrupt Service Routine (ISR)**, aufgerufen. Wenn diese beendet ist, läuft das Hauptprogramm weiter. Interrupts bieten damit den Peripherie-Komponenten eines Mikrocontrollers die Möglichkeiten, in den Programmfluss einzugreifen und auf externe Ereignisse zu reagieren. Sie werden beispielsweise ausgelöst

- bei Pegeländerungen an bestimmten Pins des Mikrocontrollers (sog. **externe Interrupts**),
- nach Ablauf einer vorher festgelegten Zeitspanne (**Timer**),
- beim Abschluss einer **Analog-Digital-Wandlung**,
- nach dem Versand oder Empfang eines Zeichens über **UART**,
- bei Aktivitäten auf dem **CAN-, LIN-, oder I<sup>2</sup>C-Bus, am LAN oder USB**

oder bei anderen Ereignissen in der Peripherie des Mikrocontrollers, die beliebig häufig, periodisch oder unregelmäßig auftreten.

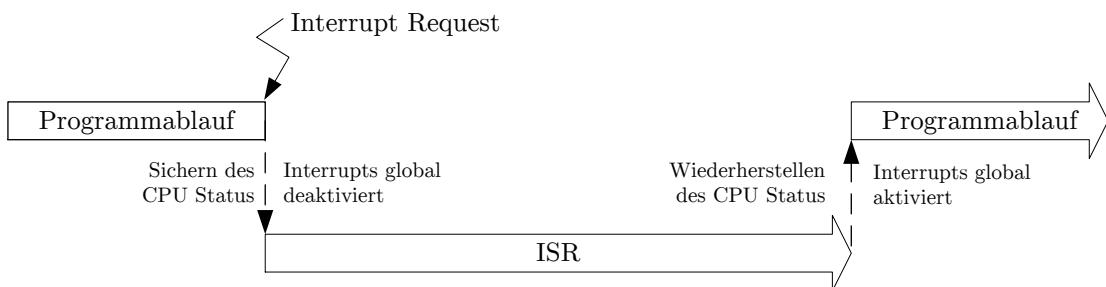


Abbildung 4.3.: Ablauf einer Programmunterbrechung [vgl. 6, S. 188].

Während der Ausführung einer ISR sind bei vielen Mikrocontrollern keine weiteren Interrupts möglich (s.u. verschachtelte Interrupts). Aus diesem Grund sollten ISRs so kurz wie irgend möglich gehalten und schnell beendet werden. Insbesondere muss bei periodisch auftretenden Interrupts die Interruptroutine kürzer sein als die Periodendauer des Ereignisses, anderenfalls können Interrupts „verschluckt“ werden, d.h. beim UART gehen Daten verloren, beim Timer Zählzyklen, beim AD-Wandler Messwerte etc. Programmfehler aufgrund solcher verschluckter Interrupts sind bisweilen schwer zu finden, weil sie schlecht reproduziert werden können. Langwierige Berechnungen, Auswertungen, Ausgaben (z.B. mittels `printf()` etc.) oder gar Warteschleifen haben in ISRs daher nichts zu suchen. Hier kommt sinnvollerweise eine andere Programmiertechnik zum Einsatz, nämlich die Übergabe von Parametern bzw. Steuersignalen an das Hauptprogramm, die wir anhand des UART später detaillierter erläutern. Hier sei noch angemerkt, dass

Variablen im C-Compiler als **volatile** deklariert werden müssen, wenn diese in einer ISR verändert werden können. Dadurch wird verhindert, dass der Compiler bei der Optimierung Vereinfachungen vornimmt, die darauf basieren, dass eine Variable im Hauptprogramm nicht verändert wird.

```
uint8_t volatile count = 0;
int main(void)
{
    while (1)
    {
        if (count == 5)
        {
            count = 0;
            // tu irgendwas
        }
    }
}
```

Im obigen Beispiel würde die if-Bedingung ohne **volatile** von Haus aus nie erfüllt, weil **count** sich im Programm nicht ändert. Der Compiler könnte den if-Zweig folglich entfernen, obwohl womöglich eine Interrupt-Routine existiert, die **count** inkrementiert:

```
ISR ( . . . )
{
    count++;
}
```

Indem man **count** also als **volatile** deklariert, teilt man dem Compiler mit, dass die Variable flüchtig ist und sich jederzeit in einer ISR ändern kann.

### 4.3.2. Interruptsteuerung

Interrupts müssen wie alle anderen Module und Funktionen eines Mikrocontrollers aktiviert und gesteuert werden. Dazu wird auf praktisch allen Mikrocontrollern ein zweistufiges System verwendet.

**Globale Interruptsteuerung:** Ein zentrales Steuerbit (beim AVR ist es das I-Bit im Statusregister SREG) wirkt wie ein Hauptschalter und kann global die Ausführung aller Interrupts ein- und ausschalten. Werden Interrupts ausgelöst und deren Interrupt-Flags gesetzt, solange die Interruptausführung global abgeschaltet ist, hängt es vom Typ des Interrupts ab, ob dieser verloren gehen kann oder nicht. Einige Interrupt-Flags sind nur solange gesetzt, wie die Bedingung für den Interrupt vorhanden ist, andere Interrupt-Flags wiederum bleiben gespeichert und sind also auch noch gesetzt wenn die eigentliche Bedingung für den Interrupt nicht mehr gegeben ist. Sobald die Interruptausführung global wieder freigegeben wird, wird für ein gesetztes Interrupt-Flag die zugehörige ISR ausgeführt. Verloren gehen Interrupts also lediglich wenn die Sperrzeit zu groß ist und währenddessen mehr als ein Interrupt vom selben Typ eintrifft und/oder wenn bei einem nicht-speicherndem Interrupt-Flag die Bedingung für den Interrupt nicht mehr vorliegt.

**Lokale Interruptsteuerung:** Zusätzlich kann jede einzelne Interruptquelle individuell ein- und ausgeschaltet werden. Dies geschieht mithilfe von Steuerregistern, die entsprechende Bits enthalten.

Will eine Peripherie-Einheit einen Interrupt auslösen, so setzt sie das zugehörige Flag im entsprechenden Statusregister auf 1. Ist der Interrupt lokal und global aktiviert, springt der Befehlszähler zum sog. Vektor der ISR. Eine ISR wird also nur dann ausgeführt, wenn

- die Interrupts global freigeschaltet sind, (*1. globale Aktivierung*)
- das individuelle Maskenbit gesetzt ist und (*2. lokale Aktivierung*)
- das Interrupt-Ereignis auftritt, also das Flag gesetzt wird. (*3. Interrupt-Ereignis tritt auf*)

Dieses System hat eine Reihe von Vorteilen. So können sehr schnell und einfach alle Interrupts kurzzeitig gesperrt werden, wenn beispielsweise atomare Operationen durchgeführt werden sollen, oder besonders zeitkritische Abläufe ausgeführt werden. Danach können alle konfigurierten Interrupts einfach wieder freigeschaltet werden, ohne dass die CPU viele verschiedene Interruptmaskenbits verwalten muss. Beim AVR existieren hierfür eigene Anweisungen im Befehlssatz des Prozessors, sodass die Sperrung oder Freischaltung des I-Bits nur einen einzigen Takt erfordert (zur Erinnerung: die Modifikation eines Registers mithilfe einer Maske dauert länger!). Der Befehl **SEI** (Global Interrupt Enable) aktiviert globale Interrupts, **CLI** (Global Interrupt Disable) sperrt sie.

### 4.3.3. Externe Interrupts am AVR

Die Beschreibung der wichtigsten Kontrollregister zur Interruptsteuerung ist in Abbildung 4.4 dargestellt.

### 4.3.4. Verschachtelte Interrupts

Einige Mikrocontroller, wie z.B. der AVR kennen nur zwei CPU-Zustände, Normale Programmausführung und Interruptausführung, gesteuert durch das I-Bit der CPU. Die normale Programmausführung kann jederzeit durch Interrupts unterbrochen werden. Die Interruptausführung kann nicht durch neue Interrupts unterbrochen werden. Die ISR wird erst zu Ende bearbeitet, zurück in die normale Programmausführung gesprungen und erst dann werden neue, wartende (engl. pending) Interrupts bearbeitet. Etwas komplexere Mikrocontroller oder große Prozessoren bieten verschiedene Interruptlevel (Stufen) an. Dabei gilt meist je niedriger die Zahl des Levels, um so höher die Priorität. Ein Interrupt mit höherer Priorität kann einen Interrupt mit niedriger Priorität unterbrechen. Ein Interrupt mit gleicher Priorität wie der gerade bearbeitete Interrupt kann das im Allgemeinen nicht. Das nennt man verschachtelte Interrupts (engl. nested interrupts). Klassische Vertreter hierfür sind 8051, PowerPC, X86 und Motorola 68000. Auf dem AVR kann man das Verhalten verschachtelter Interrupts sowohl in Assembler als auch in C durch Software emulieren, allerdings mit einigen Einschränkungen und Tücken, weshalb diese Technik Experten vorbehalten ist.

**MCU Control Register – MCUCR** The MCU Control Register contains control bits for interrupt sense control and general functions.

Bit	7	6	5	4	3	2	1	0	
	SE	SM2	SM1	SM0	ISC11	ISC10	ISC01	ISC00	MCUCR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

(a) MCU Control Register zur Steuerung externer Interrupts

ISC11	ISC10	Description
0	0	The low level of INT1 generates an interrupt request
0	1	Any logical change on INT1 generates an interrupt request
1	0	The falling edge of INT1 generates an interrupt request
1	1	The rising edge of INT1 generates an interrupt request

(b) Bitbeschreibung des MCU Control Registers

**General Interrupt Control Register – GICR**

Bit	7	6	5	4	3	2	1	0	
	INT1	INT0	–	–	–	–	IVSEL	IVCE	GICR
Read/Write	R/W	R/W	R	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

(c) General Interrupt Control Register

**General Interrupt Flag Register – GIFR**

Bit	7	6	5	4	3	2	1	0	
	INTF1	INTF0	–	–	–	–	–	–	GIFR
Read/Write	R/W	R/W	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	

(d) General Interrupt Flag Register

Abbildung 4.4.: Kontrollregister zur Interruptsteuerung [5, S. 49ff].

### 4.3.5. Ist die ISR schnell genug?

Um abschätzen zu können, ob eine ISR schnell genug abgearbeitet wird, um die oben beschriebenen Komplikationen zu vermeiden, sollte im Zweifelsfall eine Abschätzung vorgenommen werden, wie lange der Mikrocontroller maximal braucht, um sie zu durchlaufen. Hierfür bieten sich im Wesentlichen drei Techniken an:

**Simulation** Viele Hersteller bieten Simulatoren für ihre Controller an, mit deren Hilfe das CPU-Verhalten des Chips am PC emuliert werden kann. Mit deren Hilfe muss der längste Pfad in einer verzweigten ISR simuliert werden; alle beteiligten Variablen sind hierfür auf geeignete Werte zu setzen.

**Messung mit dem Oszilloskop** Dabei wird zum Beginn der ISR ein Pin auf *high* gesetzt und am Ende auf *low* - dies erfordert kaum Aufwand. Damit kann man in Echtzeit die Dauer der ISR messen. Die zusätzlichen Taktzyklen zum Aufruf und verlassen der ISR sind bekannt.

**Timer** Zu Beginn der ISR wird ein Timer (siehe folgender Abschnitt) gestartet. Am Ende der ISR wird der Zählerstand gespeichert. Dieser gibt abhängig von den Timer-Einstellungen den vergangene Zeit an.

### 4.3.6. Interruptgesteuerte UART-Kommunikation

Wir haben bereits einführend die Kommunikation zwischen Mikrocontroller und PC mit Hilfe der UART-Schnittstelle kennengelernt. Dabei haben wir gesehen, dass z.B. bei einer Übertragungsgeschwindigkeit von 9600 Baud etwa  $4\text{ }\mu\text{s}$  pro Bit vergehen. Ein Mikrocontroller, der mit einem 12 MHz-Quarz betrieben wird, durchläuft in dieser Zeit rund 1250 Takte, von denen er (je nach Komplexität der Anwendung) oft nur wenige für die Verarbeitung oder Bereitstellung der Daten benötigt. Den Rest der Zeit verbringt der Prozessor in einer Warteschleife, bis das nächste Zeichen empfangen oder gesendet werden kann. Man nennt diesen Modus Polling.

Interrupts erlauben eine viel effizientere Art z.B. des Versands von Daten: Der Prozessor wird vom UART informiert, sobald das letzte Zeichen versandt wurde, er unterbricht sein Programm um schnell das nächste Zeichen in den Puffer zu schieben, und fährt anschließend mit seinem Hauptprogramm fort. Analog kann so der Empfang erfolgen: In der ISR wird das empfangene Zeichen nur vom Prozessor abgeholt und in eine Puffervariable gespeichert. Im Hauptprogramm wird dann regelmäßig geprüft, ob Daten im Puffer liegen, und diese ggf. ausgewertet.

Der ATmega kann im Zusammenhang mit der UART Schnittstelle zwei verschiedene Interrupts auslösen, die über das UCSRB-Register aktiviert werden (Abb. 4.5). RXCIE löst einen Interrupt aus, sobald alle Daten im Schieberegister gesendet wurden, TXCIE entsprechend wenn alle Bits empfangen wurden.

Status Register B – UCSRB		Bit	7	6	5	4	3	2	1	0	
			RXCIE	TXCIE	UDRIE	RXEN	TXEN	UCSZ2	RXB8	TXB8	UCSRB
Read/Write			R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	
Initial Value			0	0	0	0	0	0	0	0	

Abbildung 4.5.: USART Control and Status Register B [5, S. 149].

Eine beispielhafte Implementierung der interruptgesteuerten UART-Kommunikation befindet sich in den Dateien `uart.c` und `uart.h`. Deren Verwendung und Funktionsweise ist in der Header-Datei dokumentiert.

### 4.3.7. ISR-Ablauf

Das folgende Beispiel soll verdeutlichen, welche Vorgänge beim Sprung in eine Interrupt-Routine geschehen, ohne dass dies im C-Code ersichtlich ist.

#### Beispiel 4.1: Einfaches Programm mit Timer ISR

Ein einfaches Programm besteht aus einer Hauptschleife und einer ISR, die durch einen Überlauf von Timer0 ausgelöst wird. In der Hauptschleife des Programms wird lediglich die Variable `x` hochgezählt. Innerhalb der ISR wird der Variable `y` der Wert  $x + 5$  zugewiesen.

Das obige Program lässt sich in C folgendermaßen schreiben:

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

```

volatile uint8_t x;
volatile uint8_t y;

int main(void)
{
    x = 0;
    y = 0;

    // Timer0 mit Prescaler 8 aktivieren
    TCCR0 |= (1<<CS01);

    // Timer0 Overflow Interrupt lokal aktivieren
    TIMSK |= (1<<TOIE0);

    // Globale Interrupt-Aktivierung
    sei();

    // Hauptschleife
    while(1)
    {
        // x um 1 hochzaehlen
        x += 1;
    }

    // Timer0 Overflow-Interrupt Service Routine
    ISR(TIMER0_OVF_vect)
    {
        y = 5 + x;
    }
}

```

Beim Kompilieren des Programms erstellt der GCC neben dem binären Programmcode eine \*.lss-Datei, die den C-Code und den daraus generierten Assembler-Code für Menschen lesbar enthält.

Für die Rechnung innerhalb der Hauptschleife erhält man folgenden Assembler-Code:

	x += 1;	
94:	80 91 60 00	lds r24 , 0x0060
98:	8f 5f	subi r24 , 0xFF ; 255
9a:	80 93 60 00	sts 0x0060 , r24

Zunächst wird der Wert der Variable  $x$  von der in  $Y$  gespeicherten Adresse in das Register 24 geladen (**ldd**). Nun wird der Inhalt von Register 24 mit 1 addiert und wieder in Register 24 geschrieben. Anschließend wird der Inhalt von Register 24 wieder auf die in  $Y$  hinterlegte Speicheradresse geschrieben (**std**). Statt der Addition um 1 verwendet der Compiler eine Subtraktion um 255 mit dem Befehl **subi** (subtract immediately). Dieser



Befehl führt direkt eine Subtraktion aus, ohne dass die Daten zunächst geladen werden müssen und ist somit die schnellste Möglichkeit die Rechnung auszuführen. Ein entsprechender **addi**-Befehl existiert auf dem AVR nicht. Dieses Beispiel zeigt dass das Verhalten des Compilers nicht immer auf den ersten Blick nachvollziehbar ist.

Wie man am folgenden Code erkennt, enthält die Interrupt Service Routine deutlich mehr Programmcode als die eigentliche Rechnung:

```
ISR(TIMER0_OVF_vect)
{
    a0: 1f 92      push   r1
    a2: 0f 92      push   r0
    a4: 0f b6      in     r0 , 0x3f      ; 63
    a6: 0f 92      push   r0
    a8: 11 24      eor    r1 , r1
    aa: 8f 93      push   r24
    ac: cf 93      push   r28
    ae: df 93      push   r29
    b0: cd b7      in     r28 , 0x3d      ; 61
    b2: de b7      in     r29 , 0x3e      ; 62
        y = 5 + x;
    b4: 80 91 60 00 lds    r24 , 0x0060
    b8: 8b 5f      subi   r24 , 0xFB      ; 251
    ba: 80 93 61 00 sts    0x0061 , r24

    be: df 91      pop    r29
    c0: cf 91      pop    r28
    c2: 8f 91      pop    r24
    c4: 0f 90      pop    r0
    c6: 0f be      out    0x3f , r0      ; 63
    c8: 0f 90      pop    r0
    ca: 1f 90      pop    r1
    cc: 18 95      reti
```

Zunächst werden die beiden Register R1 und R0 auf den Stack geschoben (**push**)<sup>1</sup>. Anschließend wird das Statusregister **SREG** in R0 geladen und auf dem Stack gesichert (a4 bis a6). Nach dem leeren des Register R1 (**eor R1, R1**) werden die Register R24, R28 und R29 gesichert. Als letzter Schritt in der Einsprungsequenz wird der Stackpointer (Adresse der aktuellen Stackposition) in die Register R28 und R29 geladen.

Die Berechnung selber nimmt lediglich 3 Befehle in Anspruch (b4 bis ba) und wird analog zur Rechnung in der Hauptschleife durchgeführt. Das Ergebnis der Rechnung wird anschließend wieder in den Speicher der Variable *y* geschrieben (**sts**: store direct to dataspace).

Nachdem die Berechnung in der ISR abgeschlossen ist, wird der Ursprüngliche Zustand der Register in Umgekehrter Reihenfolge gegenüber dem Sichern **pop** wieder hergestellt (be bis c4). Dabei wird der Zustand des Statusregisters vom Stack in R0 geladen und

---

<sup>1</sup>Der Stack ist ein Stapelspeicher, der nach dem Last-In-First-Out Prinzip arbeitet. Im Prinzip arbeitet er also wie ein Papierstapel auf dem Schreibtisch.

anschließend in das Statusregister geschrieben (*d0 bis d2*). Mit dem `reti`-Befehl wird der Befehlszähler auf den Stand vor Ablauf der ISR zurückgesetzt und der normale Programmablauf geht weiter.

**Beispiel 4.2:** Dauer der Ein- und Aussprungsequenz einer ISR

Zählt man die Zyklen der einzelnen im obigen ISR-Beispiel verwendeten Befehle zusammen (siehe [3]), so ergibt sich eine Gesamtdauer von 37 Taktzyklen. Zieht man die 4 Takte für die Berechnung ab, ergeben sich immer noch 33 Takte Overhead alleine für den Ein- und Aussprung in bzw. aus der ISR. Bei einem Takt von beispielsweise 16 MHz würden allein hierfür 2,06 µs benötigt.

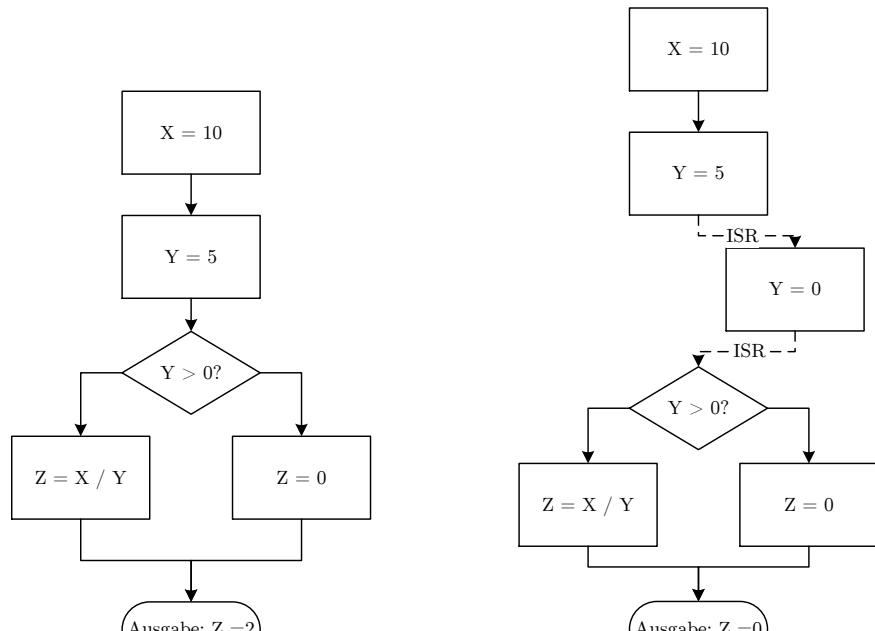
#### 4.3.8. Race-Condition

Beim AVR ist eine ISR die einzige Möglichkeit, dass der Programmablauf unterbrochen und eine andere Routine abgearbeitet wird. Werden in der ISR Daten verändert, die jedoch gerade im Hauptprogramm bearbeitet wurden, so kann es zu einer s.g. *Race Condition* (Wettlauf) kommen. Allgemein spricht man von einer *Race Condition*, wenn das Ergebnis einer Operation vom zeitlichen Verhalten einzelner Operationen abhängt. Dies ist zum Beispiel der Fall, wenn zwei Systeme für eine Rechnung auf die gleichen Daten zugreifen und diese dabei verändern. Je nachdem, welches System zuerst auf die Daten zugreift ändert sich das Ergebnis (Wettrennen um den Zugriff → *Race Condition*).

Das gleiche geschieht, wenn innerhalb einer ISR auf Daten der Hauptschleife zugegriffen wird. Abbildung 4.6 zeigt an einem konstruierten Beispiel welchen Einfluss das Ändern eines Wertes innerhalb einer ISR haben kann. Je nachdem, zu welchem Zeitpunkt die ISR aufgerufen wird, ändert sich am Ergebnis nichts oder aber die Rechnung schlägt komplett fehl.

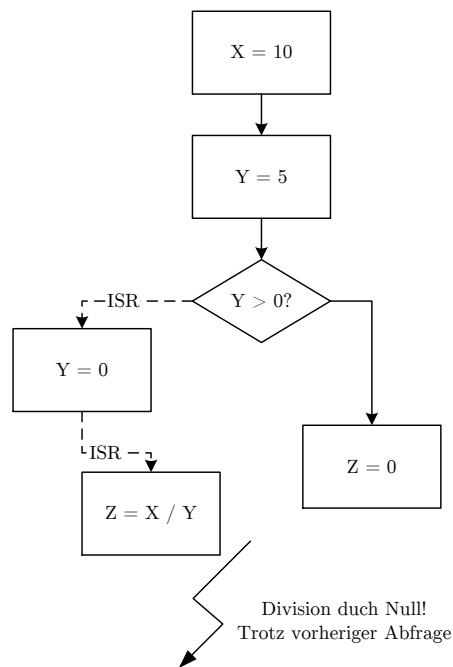
**Beispiel 4.3:** Race Condition durch ISR

In dem kurzen Programmausschnitt in Abbildung 4.6 werden zunächst die beiden Variablen  $X$  und  $Y$  auf 10 bzw. 5 initialisiert. Im Anschluss soll die Zahl  $Z = X/Y$  berechnet werden. Um eine Division durch Null zu verhindern wird vor der Rechnung geprüft ob der Nenner  $Y$  größer 0 ist. Ist er gleich 0, wird das Ergebnis einfach auf  $Z = 0$  gesetzt (Abb. 4.6(a)). Nun könnte in einer fiktiven ISR der Wert der Variable  $Y$  auf  $Y = 0$  gesetzt werden. Wird diese ISR vor der Abfrage  $Y > 0$  aufgerufen, so funktioniert das Programm weiterhin (Abb. 4.6(b)). Tritt die ISR jedoch direkt nach der Abfrage auf, schlägt die Rechnung  $Z = X/Y$  trotz der vorherigen Prüfung fehl (Abb. 4.6(b)).



(a) Ohne ISR

(b) ISR zu günstigem Zeitpunkt



(c) ISR zu ungünstigem Zeitpunkt

Abbildung 4.6.: Worst-Case Szenario für eine Race Condition.

Bereits an diesem simplen Beispiel erkennt man welche Probleme sich aus den s.g. Race Conditions ergeben können. Es können jedoch auch deutlich schwieriger auffindbare Probleme entstehen, wie das nächste Beispiel zeigt.

**Beispiel 4.4:** ISR während Addition von 32bit-Zahlen

Auf einem 8-Bit Mikroprozessor werden größere Zahlen (16-Bit, 32-Bit) jeweils auf mehrere Register aufgeteilt. Jede Operation mit dieser Zahl muss also prinzipiell für jedes Register einzeln ausgeführt werden (Ausnahme: ADIW-Befehl). Am folgenden Beispielcode sieht man, wie der Einschub einer ISR in die Addition zweier Zahlen zu Inkonsistenzen führt.



Das Programm besteht wie Beispiel 4.1 lediglich aus einer Hauptschleife, in der die Variable  $x$  jeweils um 1 erhöht wird und einer ISR die beim Überlauf des Timer0 ausgelöst wird. In der ISR wird die Variable  $x$  um 5 erhöht.

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

volatile uint32_t x;

int main(void)
{
    x = 0;

    // Timer0 mit Prescaler 8 aktivieren
    TCCR0 |= (1<<CS01);

    // Timer0 Overflow Interrupt lokal aktivieren
    TIMSK |= (1<<TOIE0);

    // Globale Interrupt-Aktivierung
    sei();

    // Hauptschleife
    while(1)
    {
        // x um 1 hochzaehlen
        x += 1;
    }

    // Timer0 Overflow-Interrupt Service Routine
    ISR(TIMER0_OVF_vect)
    {
        x += 5;
    }
}
```

 Wie man sieht belegt eine 32-Bit Zahl 4 Register (R24 - R27). Mithilfe des adiw-Befehls (Add Immediate to Word) wird zu den ersten beiden Bytes (R24 und R25) die 1 addiert. Anschließend wird der Übertrag zuerst auf R26 und der daraus resultierende Übertrag auf R27 addiert (adc: Add With Carry). Die Addition wird hier also in 3 einzelne Befehle aufgeteilt. Situationsabhängig kann der Compiler die Addition auch in andere Befehlsfolgen wie z.B. subi + adc + adc + adc umsetzen.

```
while (1)
{
    // x um 1 hochzaehlen
    x += 1;
    9c: 80 91 60 00      lds      r24 , 0x0060
    a0: 90 91 61 00      lds      r25 , 0x0061
    a4: a0 91 62 00      lds      r26 , 0x0062
    a8: b0 91 63 00      lds      r27 , 0x0063
    ac: 01 96            adiw     r24 , 0x01      ; 1
    ae: a1 1d            adc      r26 , r1
    b0: b1 1d            adc      r27 , r1
    b2: 80 93 60 00      sts      0x0060 , r24
    b6: 90 93 61 00      sts      0x0061 , r25
    ba: a0 93 62 00      sts      0x0062 , r26
    be: b0 93 63 00      sts      0x0063 , r27
}
```

In der ISR werden beim Einsprung zunächst, wie in Beispiel 4.1 bereits besprochen, die Register gesichert (*c4* bis *dc*). Anschließend werden die Variablen aus dem Speicher in die Register geladen (*de* - *ea*), die Berechnung durchgeführt (*ee* - *f2*) und das Ergebnis zurück in den Speicher geschrieben. Anschließend wird der Zustand vor der ISR wieder hergestellt (104 - 116) und das Programm normal fortgesetzt.

c4:	1f 92	push	r1
c6:	0f 92	push	r0
c8:	0f b6	in	r0 , 0x3f ; 63
ca:	0f 92	push	r0
cc:	11 24	eor	r1 , r1
ce:	8f 93	push	r24
d0:	9f 93	push	r25
d2:	af 93	push	r26
d4:	bf 93	push	r27
d6:	cf 93	push	r28
d8:	df 93	push	r29
da:	cd b7	in	r28 , 0x3d ; 61
dc:	de b7	in	r29 , 0x3e ; 62
	x = 5 + x;		
de:	80 91 60 00	lds	r24 , 0x0060
e2:	90 91 61 00	lds	r25 , 0x0061
e6:	a0 91 62 00	lds	r26 , 0x0062
ea:	b0 91 63 00	lds	r27 , 0x0063

ee :	05 96	adiw	r24 , 0x05	; 5
f0 :	a1 1d	adc	r26 , r1	
f2 :	b1 1d	adc	r27 , r1	
f4 :	80 93 60 00	sts	0x0060 , r24	
f8 :	90 93 61 00	sts	0x0061 , r25	
fc :	a0 93 62 00	sts	0x0062 , r26	
100:	b0 93 63 00	sts	0x0063 , r27	
104:	df 91	pop	r29	
106:	cf 91	pop	r28	
108:	bf 91	pop	r27	
10a:	af 91	pop	r26	
10c:	9f 91	pop	r25	
10e:	8f 91	pop	r24	
110:	0f 90	pop	r0	
112:	0f be	out	0x3f , r0	; 63
114:	0f 90	pop	r0	
116:	1f 90	pop	r1	
118:	18 95	reti		

Was passiert nun wenn die ISR genau innerhalb der Addition in der Hauptschleife, z.B. an der Stelle *ae* aufgerufen wird?

Wird *x* beispielsweise in der Hauptschleife gerade von 300 auf 301 erhöht, steht während der Addition (vor den **sts**-Befehlen) der Wert 300 im Speicher. Beim Aufruf der ISR lädt diese den Wert aus dem Speicher, berechnet daraus den neuen Wert *x* = 305 und schreibt ihn zurück in den Speicher. Nach der ISR wird die Hauptschleife aber genau an der Stelle fortgesetzt, an der sie unterbrochen wurde. Das heißt es wird weiter  $300 + 1 = 301$  berechnet, da ja genau diese Werte noch in den Registern stehen. Schließlich wird der Wert *x* = 301 in den Speicher geschrieben, obwohl er zuvor durch die ISR auf *x* = 305 erhöht wurde.

## Lösung des Problems

Um das Problem zu umgehen bieten sich **zwei Vorgehensweisen** an:

- Interrupts zu Beginn einer kritischen Berechnung deaktivieren, anschließend wieder aktivieren.
- Nutzung von Semaphoren, die angeben ob eine Variable verändert werden darf oder nicht. 

# 5. Timer/Counter und Pulsweitenmodulation

## 5.1. Was ist ein Timer/Counter

Timer/Counter sind periphere Hardwarekomponenten des Mikrocontrollers, die ein bestimmtes Register getriggert von einem Signal um 1 erhöhen (inkrementieren) oder erniedrigen (dekrementieren). Sie sind ein wichtiger Bestandteil fast aller Mikrocontroller, viele verfügen über mehr als einen Timer. Wählt man als *Trigger-Signal* z.B. den Systemtakt ist es möglich in regelmäßigen Zeitabständen Aktionen zu veranlassen. Aber Timer können noch mehr:

- Timer/Counter können mit einem externen Pin hoch/runter gezählt werden.
- Es gibt Möglichkeiten, bei bestimmten Zählerständen einen Interrupt auslösen zu lassen.
- Timer/Counter können aber auch zur selbstständigen Signalerzeugung an einem Ausgangspin verwendet werden.

Ein Hardware-Timer kann für verschiedene Funktionen konfiguriert und verwendet werden. Der Timer kann mit dem Systemtakt verbunden und folglich die Genauigkeit des Quarzes ausgenutzt werden, um ein Register regelmäßig und v.a. unabhängig von der CPU (d.h. vom restlichen Programmfluss) hochzählen zu lassen. Nach einem *Overflow*, also dem Überlauf des Zählregisters, beginnt der Timer wieder bei 0: ein 8-bit-Timer/Counter zählt also z.B. von 0 bis  $2^8 - 1 = 255$  und fängt von vorne an. Ein 16-bit-Timer/Counter zählt dementsprechend bis  $2^{16} - 1 = 65535$  hoch. Das Zählregister **TCNTx** kann von der CPU abgefragt (z.B. um die verstrichene Zeit zu messen) und auch beschrieben werden. Außerdem können bei bestimmten Zählerständen (z.B. beim *Overflow* oder beim Erreichen eines bestimmten Werts) Interrupts ausgelöst werden.

Um die Periode zwischen zwei Interrupts größenordnungsmäßig einzustellen, besitzt ein Timer typischerweise einen Verteiler (*Prescaler*). Mit dessen Hilfe wird der Timer nicht bei jedem Systemtakt, sondern z.B. nur bei jedem 8., 64., etc. Takt inkrementiert. Schwingt ein Quarz beispielsweise mit 12MHz, so läuft ein 8-bit-Timer alle  $21,3\text{ }\mu\text{s}$  über. Stellt man den *Prescaler* auf 1024, geschieht dies nur alle 21,8 ms.

Die exakte Periodendauer kann beispielsweise eingestellt werden, indem nach dem *Overflow* der Zählerstand **TCNTx** auf einen größeren Startwert als 0 gesetzt wird (siehe Bsp. 5.1).

Tabelle 5.1.: Overflow-Frequenz und -Periode für verschiedene Prescaler bei 4MHz. 

Vorteiler	$f$ Hz	$\Delta t$ [s]
1	15625	0,000064
8	1953,1	0,000512
64	244,14	0,004096
256	61,035	0,016384
1024	15,259	0,065536

**Beispiel 5.1:** Wie muss der Startwert  $TCNTx$  gewählt werden?

Die Frequenz mit der ein Ereignis durch den Timer/Counter ausgelöst wird, ergibt sich aus der Taktfrequenz der CPU, dividiert durch den Vorteiler (Prescaler) und die Anzahl der Zähler-Erhöhungen (jeweils 1 Takt) bis zum Überlauf (hier für 8-bit Timer):

$$f_{\text{Periode}} = \frac{f_{\text{CPU}}}{\text{Prescaler} \cdot (256 - TCNTx)} \quad (5.1)$$

Möchten wir auf einer 4 MHz-CPU also mit dem 8-Bit Timer/Counter mit einer Frequenz von 100 Hz (Prescaler 256) eine Aktion durchführen, so muss der Counter nach jedem Überlauf den Startwert

$$TCNTx = 256 - \frac{f_{\text{CPU}}}{\text{Prescaler} \cdot f_{\text{Periode}}} = 256 - \frac{4 \text{ MHz}}{256 \cdot 100 \text{ Hz}} \approx 100 \quad (5.2)$$

gesetzt werden.

## 5.2. Timer/Counter im AVR

Die Mikrocontroller der AVR Serie besitzen unterschiedliche und verschieden viele Timer. Der ATmega8 beispielsweise verfügt über zwei 8-bit-Timer/Counter (Timer0, Timer2) und einen 16-bit-Timer (Timer1). Der Aufbau des einfachsten Timer/Counters im ATmega8 ist in Abbildung 5.1 dargestellt.

### 5.2.1. 8-bit Timer/Counter0 im ATmega8

Die Aktivierung und Einstellung der Taktquelle des Timers erfolgt im Register **TCCR0**:

Das Zählregister heißt **TCNT0** (Abb. 5.3). In diesem wird der eigentliche Zählerwert hochgezählt. Es kann sowohl ausgelesen als auch überschrieben werden.

Mithilfe des Bits **TOIE0** (Timer Overflow Interrupt Enable 0) im **TIMSK**-Registers (Timer Interrupt Mask) kann der *Overflow-Interrupt* lokal aktiviert werden (Abb. 5.4a). Beim Overflow wird dann das Bit **TOV0** (Timer Overflow 0) im **TIFR**-Register (Timer

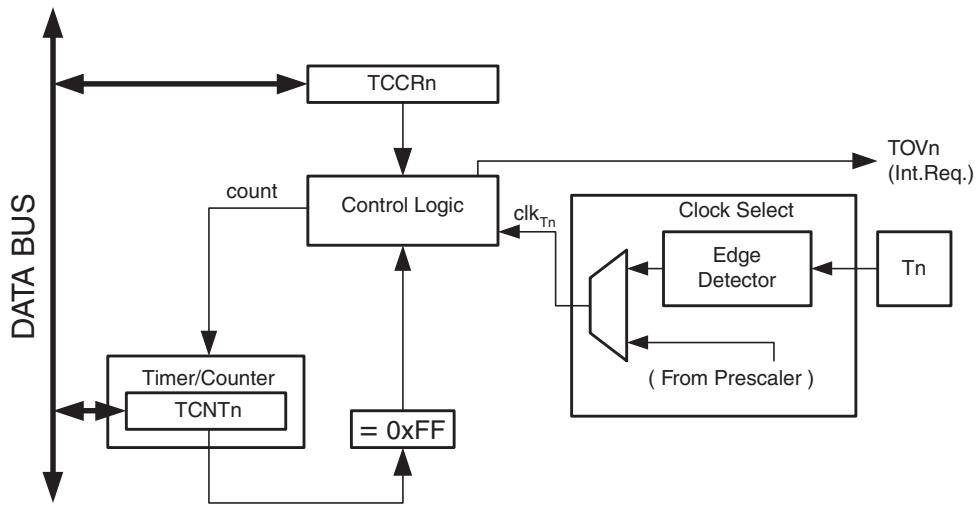


Abbildung 5.1.: Blockschaltbild 8-Bit Timer/Counter0 im ATmega8 [5, S. 29].

**Timer/Counter Control Register – TCCR0**

Bit	7	6	5	4	3	2	1	0	TCCR0
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

(a) Timer/Counter Control Register

CS02	CS01	CS00	Description
0	0	0	No clock source (Timer/Counter stopped)
0	0	1	clk <sub>I/O</sub> /(No prescaling)
0	1	0	clk <sub>I/O</sub> /8 (From prescaler)
0	1	1	clk <sub>I/O</sub> /64 (From prescaler)
1	0	0	clk <sub>I/O</sub> /256 (From prescaler)
1	0	1	clk <sub>I/O</sub> /1024 (From prescaler)
1	1	0	External clock source on T0 pin. Clock on falling edge
1	1	1	External clock source on T0 pin. Clock on rising edge

(b) Bitbeschreibung des TCCR0 Registers

Abbildung 5.2.: Steuerregister für Timer0 [5, S. 71].

**Timer/Counter Register – TCNT0**

Bit	7	6	5	4	3	2	1	0	TCNT0
Read/Write	R/W								
Initial Value	0	0	0	0	0	0	0	0	

Abbildung 5.3.: Zählregister TCNT0 [5, S. 72].

Interrupt Flag) gesetzt und die zugehörige ISR „**TIMER0\_OVF**“ aufgerufen (Abb. 5.4b).

<b>Timer/Counter Interrupt Mask Register – TIMSK</b>	Bit	7	6	5	4	3	2	1	0	
	Read/Write	R/W	<b>TIMSK</b>							
	Initial Value	0	0	0	0	0	0	0	0	

(a) Timer-Interrupt Steuerung TIMSK

<b>Timer/Counter Interrupt Flag Register – TIFR</b>	Bit	7	6	5	4	3	2	1	0	
	Read/Write	R/W	<b>TIFR</b>							
	Initial Value	0	0	0	0	0	0	0	0	

(b) Timer-Interrupt Flag Register TIFR

Abbildung 5.4.: Interruptsteuerung für Timer0 [5, S. 72].

## 5.3. Watchdog-Timer

### 5.3.1. Was macht ein Watchdog-Timer?

Durch Programmierfehler oder äußere Störungen kann es passieren, dass ein Mikrocontroller in eine „Sackgasse“ wie z.B. eine Endlosschleife gerät. Ein Gerät, das von einem solchen Mikrocontroller gesteuert wird, fällt aus („Hängt“) und ist somit im schlimmsten Fall unkontrollierbar.

Um einen solchen Fehlerzustand zu erkennen, gibt es den sogenannten **Watchdog-Timer**. Dieser funktioniert nach dem gleichen Prinzip wie ein Totmannschalter im Zug; Der Zugführer muss diesen regelmäßig betätigen, um zu zeigen dass er noch wach ist. Wird der Schalter innerhalb eines vorgegebenen Zeitraums nicht betätigt, wird eine Notbremsung eingeleitet.

Im Mikrocontroller zählt der *Watchdog*-Timer nach seiner Aktivierung solange hoch, bis er entweder durch das Programm zurückgesetzt wird oder er einen Überlauf produziert. Bei einem Überlauf wird der gesamte Mikrocontroller auf den Anfangszustand zurückgesetzt („Reset“). Im Programmablauf muss also der *Watchdog* in regelmäßigen Abständen zurückgesetzt werden. Geschieht dies nicht, z.B. weil der Mikrocontroller in einer Endlosschleife gefangen ist, löst der *Watchdog* einen Reset aus und startet den Mikrocontroller neu.

### 5.3.2. Watchdog-Timer im AVR

Im AVR verfügt der *Watchdog* über eine eigene Taktquelle, die je nach Versorgungsspannung, mit ca. 1 MHz schwingt. Die Steuerung des *Watchdog*-Timers erfolgt über das Register **WDTCR** (WatchDog-Timer Control Register). Neben der Aktivierung (**WDE** = 1) und Deaktivierung (**WDCE** = 1 und gleichzeitig **WDE** = 0), kann auch die Time-Out-Zeit durch verschiedene *Prescaler* angepasst werden.

Im C-Programm wird der Watchdog-Timer mithilfe des Befehls

```
_WDR();
```

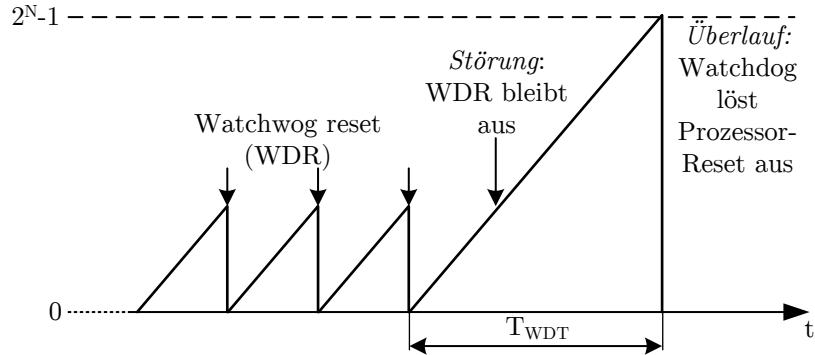


Abbildung 5.5.: Funktionsweise eines Watchdog-Timers [nach 24, S. 273].

Watchdog Timer Control Register – WDTCR								
Bit	7	6	5	4	3	2	1	0
Read/Write	R	R	R	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

(a) Watchdog Timer Control Register

WDP2	WDP1	WDP0	Number of WDT Oscillator Cycles	Typical Time-out at V <sub>CC</sub> = 3.0V	Typical Time-out at V <sub>CC</sub> = 5.0V
0	0	0	16K (16,384)	17.1ms	16.3ms
0	0	1	32K (32,768)	34.3ms	32.5ms
0	1	0	64K (65,536)	68.5ms	65ms
0	1	1	128K (131,072)	0.14s	0.13s
1	0	0	256K (262,144)	0.27s	0.26s
1	0	1	512K (524,288)	0.55s	0.52s
1	1	0	1,024K (1,048,576)	1.1s	1.0s
1	1	1	2,048K (2,097,152)	2.2s	2.1s

(b) Watchdog Timer Prescaler

Abbildung 5.6.: Steuerregister des Watchdog-Timers [5, S. 43f].

zurückgesetzt.

## 5.4. Pulsweitenmodulation (PWM)

### 5.4.1. Periode, Pulsweite und Tastverhältnis

Viele elektrische Verbraucher können in ihrer Leistung reguliert werden, indem die Versorgungsspannung in weiten Bereichen verändert wird. Ein normaler Gleichstrommotor läuft z.B. umso schneller, je größer die angelegte Versorgungsspannung ist. LEDs können ebenfalls gedimmt werden, indem mithilfe eines Vorwiderstands eine gewünschte Kombination aus Spannung und Stromstärke eingestellt wird; deren Produkt beschreibt die verbrauchte Leistung.

Jedoch gibt es noch eine weitere Möglichkeit, die Energie zu drosseln, die einem System zugeführt wird: Anstatt die Spannung abzusenken, ist es auch möglich die volle Versor-

gungsspannung über einen geringeren Zeitraum anzulegen. Glättet man den entsprechenden Spannungsverlauf (z.B. mithilfe von Kondensatoren), ergibt sich theoretisch wieder eine Gleichspannung. Diese ist gerade so groß, dass die Fläche unter den Spannungsverläufen gleich groß ist.

Diese Lösung wird als **Pulsweitenmodulation (Pulse Width Modulation, PWM)** bezeichnet und ist von großem Vorteil, da es wesentlich einfacher ist, ein **definiertes Rechtecksignal** zu erzeugen, als eine Spannung mithilfe einer analogen Verstärkerschaltung zu variieren.

Zur Erzeugung einer PWM wird also die Ein- und Ausschaltzeit eines Rechtecksignals bei fester Grundfrequenz variiert. Die **Einschaltzeit** wird als **Pulsbreite** bezeichnet. Das Verhältnis

$$DC = \frac{t_{ein}}{(t_{ein} + t_{aus})} \quad (5.3)$$

nennt man **Tastverhältnis** (engl. **Duty Cycle**, meist abgekürzt DC, nicht zu verwechseln mit Direct Current = Gleichstrom!); es liegt zwischen 0 und 1.

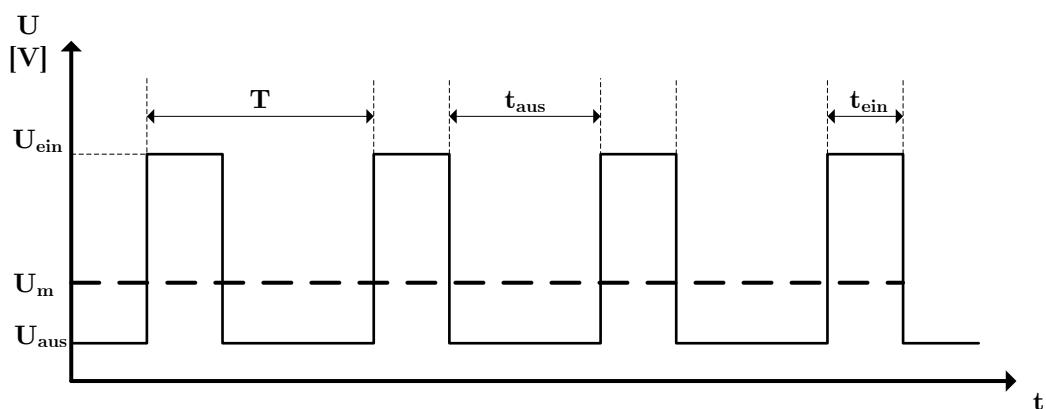


Abbildung 5.7.: Pulsweitenmodulation.

Wie leicht zu erkennen ist, gilt für den **Mittelwert der Spannung** mit der Periode  $T = t_{ein} + t_{aus}$ :

$$U_m = \frac{1}{T} \int_0^T u(t) dt = \frac{1}{T} \int_0^{t_{ein}} U_{ein} dt + \frac{1}{T} \int_{t_{ein}}^T U_{aus} dt = U_{aus} + (U_{ein} - U_{aus}) \cdot \frac{t_{ein}}{t_{ein} + t_{aus}} \quad (5.4)$$

$U_{aus}$  ist dabei normalerweise 0 V,  $U_{ein}$  die Betriebsspannung  $V_{cc}$ . Zur Berechnung der **Leistung des Verbrauchers** muss die Leistung während des ein- und ausgeschalteten Zustands getrennt betrachtet werden:

$$P = \frac{U_{ein}^2}{R} \frac{t_{ein}}{t_{ein} + t_{aus}} + \frac{U_{aus}^2}{R} \frac{t_{aus}}{t_{ein} + t_{aus}} \quad (5.5)$$

#### 5.4.2. Anwendung und Einsatzbereiche

**Motorsteuerung** Eine der Hauptanwendungen für PWM ist die Ansteuerung von (Gleichstrom-) Motoren. Der große Vorteil von PWM ist hier der **gute Wirkungs-**

grad. Würde man einen Digital-Analog-Wandler mit einem nachgeschalteten analogen Verstärker zur Ansteuerung verwenden, dann würde im Verstärker eine sehr hohe Verlustleistung in Wärme umgewandelt werden. Ein digitaler Verstärker mit PWM hat dagegen sehr geringe Verluste. Die verwendete Frequenz liegt meist im Bereich von einigen 10 kHz. Zur Berechnung der Drehzahl eines Motors kann im Normalfall der Mittelwert der PWM-Spannung als Betriebsspannung angenommen werden.

**DA-Wandlung** Die meisten Mikrocontroller haben keine DA-Wandler integriert, da diese relativ aufwändig sind. Allerdings kann man mittels eines PWM-Ausgangs auch eine DA-Wandlung vornehmen und eine Gleichspannung bereitstellen. Wird ein PWM-Signal über einen Tiefpass gefiltert (geglättet), entsteht eine Gleichspannung mit Wechselanteil, deren Mittelwert dem des PWM-Signals entspricht und dessen Wechselanteil von der Beschaltung abhängig ist. Nun bleibt das Problem der Dimensionierung des Tiefpasses.

**Datenübertragung** PWM kann auch verwendet werden, um ein analoges Signal zu kodieren. Sie kommt beispielsweise in RF-Fernsteuerungen bei der Kommunikation des Empfängers mit dem Servomotor zum Einsatz. Die Periode beträgt 20 ms, die Pulsbreite zwischen 1 ms und 2 ms. 1 ms codiert die linke Stellung, 2 ms die rechte Stellung des Motors, 1,5 ms die Mittelstellung usw.

### 5.4.3. PWM im AVR

Der 16-bit-Timer1 des ATmega8 unterstützt die automatische Erzeugung einer PWM an zwei seiner Pins. Die Besonderheit des Timer1 ist, dass er zwar nur ein 16-bit Zählregister TCNT1 besitzt, aber zwei unabhängige Kanäle, mit deren Hilfe zwei PWM-Signale gleicher Periode (aber unterschiedlicher Pulsbreite) am Mikrocontroller erzeugt werden können.

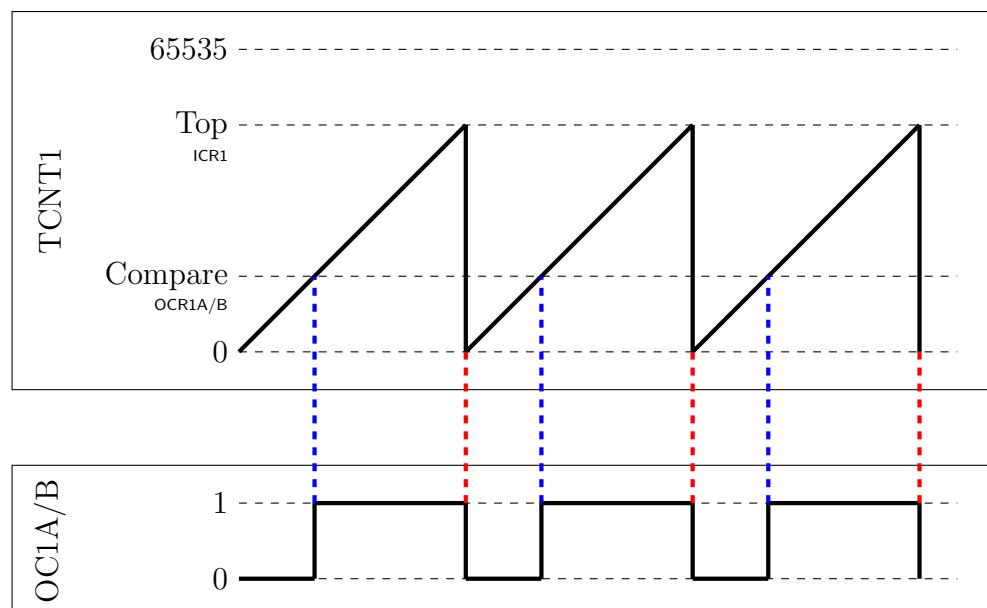


Abbildung 5.8.: FastPWM - Modus nach [5, S. 88].

Es existieren verschiedene Betriebsmodi, nämlich „Fast PWM“, „Phasen-korrekte PWM“ und „Phasen- und Frequenzkorrekte PWM“, deren genaue Bedeutung im Datenblatt ausführlich erklärt ist. Das Verhalten der einzelnen Register im Modus „Fast PWM“ zeigt Abbildung 5.8. Das Zählregister TCNT1 des Timer1 wird bei Erreichen des TOP-Wertes (gestrichelte Pfeile in der Abbildung) wieder auf Null gesetzt. Dabei wird der zugehörige Port OC1A/OC1B auf *high* gesetzt. Erreicht der Timer einen Vergleichswert (Register OCR1A/OCR1B) wird der zugehörige Port wieder auf *low* gesetzt. (Hinweis: Dieses Verhalten gilt so nur für den Modus „Fast PWM“. Der Timer kann - je nach Anwendung - auch in anderen Modi betrieben werden.)

Das Verhalten des Timer/Counter1 wird über die Register TCCR1A/TCCR1B (timer/-counter1 control register A/B, siehe Abb. 5.9) festgelegt. Zur Einstellung des Modus „Fast PWM“ werden beispielsweise die Bits WGM10-3 entsprechend Abbildung 5.10 gewählt (Beachte: unterschiedlicher TOP-Wert).

Timer/Counter 1 Control Register A – TCCR1A								
Bit	7	6	5	4	3	2	1	0
Read/Write	R/W	R/W	R/W	R/W	W	W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

(a) TCCR1A

Control Register B – TCCR1B								
Bit	7	6	5	4	3	2	1	0
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

(b) TCCR1B

Abbildung 5.9.: Steuererregister TCCR1 [5, S. 96ff].

Mode	WGM13	WGM12 (CTC1)	WGM11 (PWM11)	WGM10 (PWM10)	Timer/Counter Mode of Operation <sup>(1)</sup>	TOP	Update of OCR1x	TOV1 Flag Set on
7	0	1	1	1	Fast PWM, 10-bit	0x03FF	BOTTOM	TOP
8	1	0	0	0	PWM, Phase and Frequency Correct	ICR1	BOTTOM	BOTTOM
9	1	0	0	1	PWM, Phase and Frequency Correct	OCR1A	BOTTOM	BOTTOM
10	1	0	1	0	PWM, Phase Correct	ICR1	TOP	BOTTOM
11	1	0	1	1	PWM, Phase Correct	OCR1A	TOP	BOTTOM
12	1	1	0	0	CTC	ICR1	Immediate	MAX
13	1	1	0	1	(Reserved)	–	–	–
14	1	1	1	0	Fast PWM	ICR1	BOTTOM	TOP
15	1	1	1	1	Fast PWM	OCR1A	BOTTOM	TOP

Note: 1. The CTC1 and PWM11:0 bit definition names are obsolete. Use the WGM12:0 definitions. However, the functionality and location of these bits are compatible with previous versions of the timer

Abbildung 5.10.: Waveform Generation Modes [5, S. 97f].

Soll der Port OC1A/OC1B wie oben beschrieben geschaltet werden – bei Überlaufen des Timers auf *high* setzen, bei *compare match* auf *low* – dann entspricht das dem „*non-inverting*“ Modus wie in Abbildung 5.11 beschrieben.

Die Wahl des *Prescalers* erfolgt wie in Abbildung 5.12 dargestellt, analog zum Timer0.

COM1A1/ COM1B1	COM1A0/ COM1B0	Description
0	0	Normal port operation, OC1A/OC1B disconnected.
0	1	WGM13:0 = 9 or 14: Toggle OC1A on Compare Match, OC1B disconnected (normal port operation). For all other WGM1 settings, normal port operation, OC1A/OC1B disconnected.
1	0	Clear OC1A/OC1B on Compare Match when up-counting. Set OC1A/OC1B on Compare Match when downcounting.
1	1	Set OC1A/OC1B on Compare Match when up-counting. Clear OC1A/OC1B on Compare Match when downcounting.

Abbildung 5.11.: Compare Output Mode[5, S. 97].

CS12	CS11	CS10	Description
0	0	0	No clock source. (Timer/Counter stopped)
0	0	1	$\text{clk}_{\text{IO}}/1$ (No prescaling)
0	1	0	$\text{clk}_{\text{IO}}/8$ (From prescaler)
0	1	1	$\text{clk}_{\text{IO}}/64$ (From prescaler)
1	0	0	$\text{clk}_{\text{IO}}/256$ (From prescaler)
1	0	1	$\text{clk}_{\text{IO}}/1024$ (From prescaler)
1	1	0	External clock source on T1 pin. Clock on falling edge

Abbildung 5.12.: Clock Select Bit-Beschreibung[5, S. 99].

Die Arbeit mit Timer1 geschieht analog zum Timer0 (vgl. Abschnitt 5.2.1). Die wichtigsten Register zur Konfiguration des Timer1 sind (siehe Abb. 5.13):

- Im Register TCNT1 wird der Zählerstand des Timers gespeichert (Abb. .
- Die Register OCR1A/OCR1B legen den Wert des *compare match* fest.
- Das Register ICR1 (input capture register 1) kann die Zählerstände (TCNT1) bei externen Ereignissen speichern, oder den TOP-Wert festlegen.
- Die verschiedenen Interrupts (TOIE1: timer overflow interrupt enable, OCIE1A/B output compare match A/B interrupt enable) können im TIMSK Register lokal aktiviert / deaktiviert werden.
- Bei einem Interrupt werden die zugehörigen Flags im TIFR-Register gesetzt.

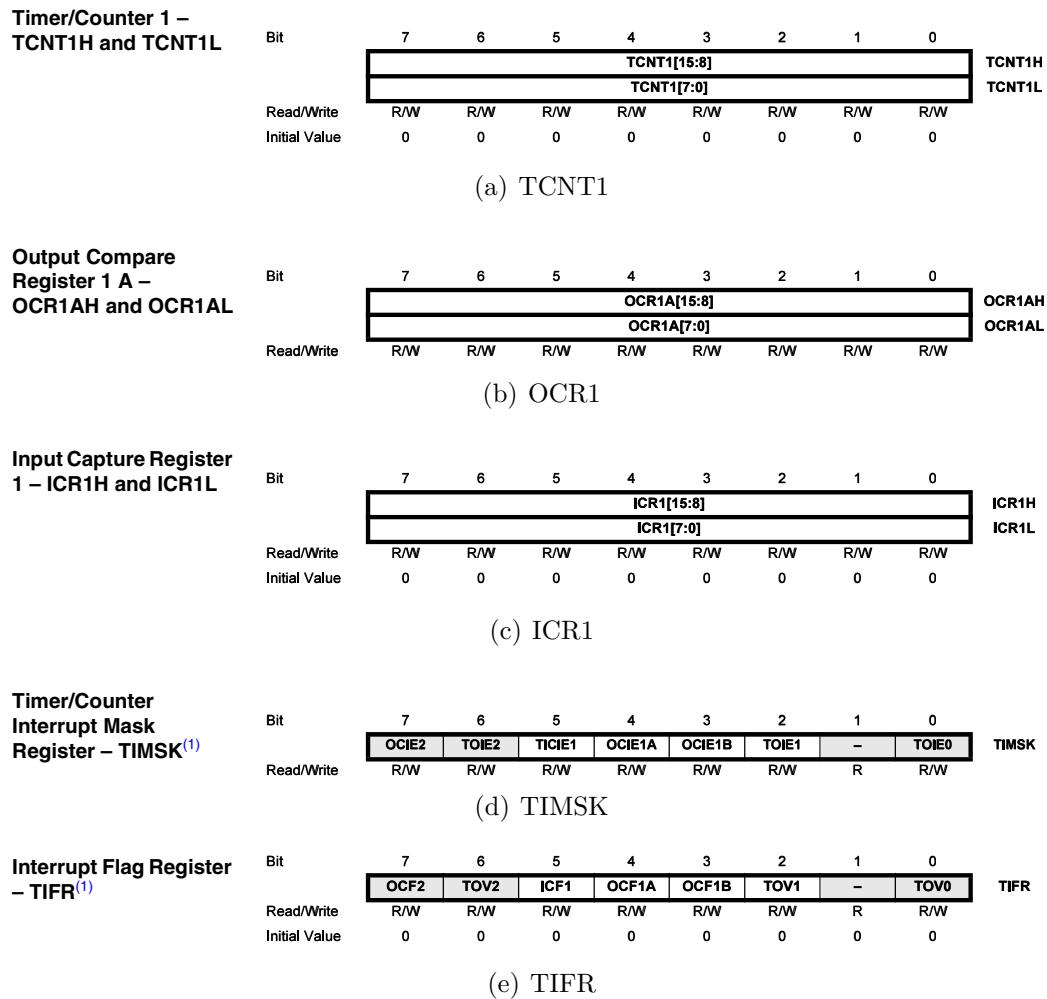


Abbildung 5.13.: Wichtige Register des TIMER1[5, S. 99ff].

# 6. Zahlendarstellung und Arithmetik

## 6.1. Motivation

Eine der Herausforderungen bei der Programmierung bzw. Implementierung auf Mikrocontrollern ist es, die richtige Darstellung der Zahlen im Binärcode so auszuwählen, dass alle benötigten arithmetischen Operationen durchführbar sind und ihre Ergebnisse dem wahren Ergebnis mit möglichst kleinem Fehler entsprechen. In diesem Kapitel werden die verschiedenen Zahlendarstellungen mit deren jeweiligen Eigenschaften sowie die Umwandlungsverfahren (dezimal/dual) eingeführt.

Die Darstellung der Zahlen im Speicher des Mikroprozessors und die Umsetzung einzelner Rechenoperationen hängt von der Registerbreite und dem Funktionsumfang des Prozessors ab (siehe Abschnitt 1.4.2).

## 6.2. Umwandlungsverfahren

Besonders wichtig für die Eingabe und Ausgabe von Dezimalzahlen auf einem Mikrocontroller sind Umwandlungsverfahren von dezimal nach dual und von dual nach dezimal. Ganze Zahlen bzw. die Vorpunktstellen werden nach dem Divisionrestverfahren fortlaufend durch 2 (Basis des dualen Zahlsystems) dividiert und die Reste ergeben die Ziffern des dualen Zahlsystems anfangend bei der wertniedrigsten<sup>1</sup> Stelle.

*Beispiel 6.1: Umwandlung von  $123_{10}$  in eine Dualzahl*

$\begin{array}{rcl} 123 & : & 2 = 61 \text{ Rest } 1 \text{ LSB} \\ 61 & : & 2 = 30 \text{ Rest } 1 \\ 30 & : & 2 = 15 \text{ Rest } 0 \\ 15 & : & 2 = 7 \text{ Rest } 1 \\ 7 & : & 2 = 3 \text{ Rest } 1 \\ 3 & : & 2 = 1 \text{ Rest } 1 \\ 1 & : & 2 = 0 \text{ Rest } 1 \end{array}$ $\Rightarrow 123_{10} = 01111011_2$
--

Nachpunktstellen werden fortlaufen mit 2 (Basis des dualen Zahlsystems) multipliziert. Die jeweilige Vorpunktstelle des Produkts ergibt die Ziffern der Dualzahl anfangend bei der höchsten Stelle<sup>2</sup>. Mit den Nachpunktstellen wird das Verfahren fortgesetzt, bis das Produkt Null ist oder die maximale Stellenzahl erreicht wurde.

<sup>1</sup>LSB: least significant bit

<sup>2</sup>MSB: most significant bit

**Beispiel 6.2:** Umwandlung von 0,687510 in eine Dualzahl

0,6875	*	2	=	1,3750	=	0,3750	+	1	MSB
0,3705	*	2	=	0,7500	=	0,7500	+	0	
0,7500	*	2	=	1,5000	=	0,5000	+	1	
0,5000	*	2	=	1,0000	=	0,0000	+	1	
0,0000	*	2	=	0,0000	=	0,0000	+	0	
$\Rightarrow 0,6875_{10} = 0,101100000_2$									

Muss das Verfahren beim Erreichen der maximalen Stellenzahl (hier 8) abgebrochen werden, so entsteht ein Umwandlungsfehler.

**Beispiel 6.3:** Umwandlungsfehler 1

0,4 <sub>10</sub> = 0,0110011001100110... <sub>2</sub>
Bei 8 Nachkommastellen (8 bits):
$0,4_{10} \approx 0,01100110_2 = 0,3984375_{10}$

**Beispiel 6.4:** Umwandlungsfehler 2

Darstellung des Werts 1<sub>10</sub> mit 8 Bits, Nachpunktstellen linksbündig, Dualpunkt vor der werthöchsten Stelle:

Bit 7	Bit 6	Bit 5	Bit 4
$2^{-1}$	$2^{-2}$	$2^{-3}$	$2^{-4}$
0,5	0,25	0,125	0,0625
<hr/>			
Bit 3	Bit 2	Bit 1	Bit 0
$2^{-5}$	$2^{-6}$	$2^{-7}$	$2^{-8}$
0,03125	0,015625	0,0078125	0,00390625

Die dezimale Summe aller acht Stellenwertigkeiten ergibt 0,9960937510, was einem Restfehler vom 0,0039062510 entspricht.

## 6.3. Datentypen

### 6.3.1. Vorzeichenlose ganze Dualzahlen (`unsigned integer`)

Bei diesem Datentyp werden die normalen Dualzahlen im Speicher abgelegt, jede Ziffer repräsentiert also ihre zugehörige Zweierpotenz.

**Beispiel 6.5:** 8-Bit Darstellung

Kleinste vorzeichenlose Zahl:	$0000000_2$	=	$0_{10}$
Größte vorzeichenlose Zahl:	$1111111_2$	=	$255_{10}$

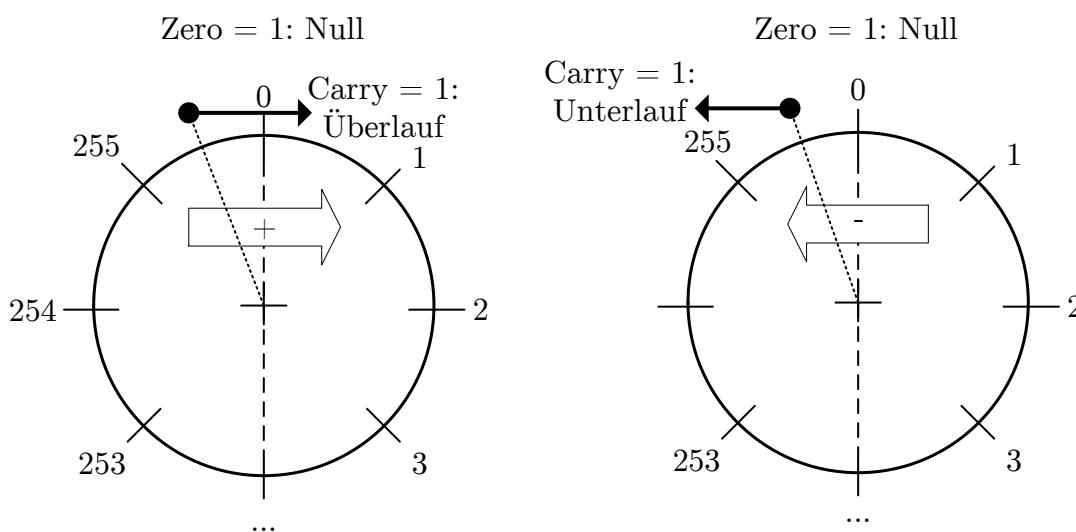


Abbildung 6.1: Addition und Subtraktion vorzeichenloser Zahlen [22, S. 19].

**Beispiel 6.6:** Überlauf vorzeichenloser ganzer Zahlen

Wird in 8-Bit Darstellung zur Zahl 250 beispielsweise die Zahl 8 addiert, übersteigt das Ergebnis den darstellbaren Bereich. Wie am Zahlenrad in Abbildung 6.1 dargestellt, beginnt dabei die Zählung wieder „von vorne“, so dass sich als Ergebnis die 2 ergibt. Dabei wird das so genannte **Carry-Bit** im Statusregister auf 1 gesetzt, um zu zeigen dass es bei der Addition zu einem Überlauf gekommen ist.

Auf den AVR-Mikrocontrollern stehen die folgenden Datentypen für vorzeichenlose ganze Zahlen zu Verfügung: `uint8_t` (`unsigned char`), `uint16_t`, `uint32_t` und `uint64_t`.



### 6.3.2. Vorzeichenbehaftete ganze Dualzahlen (signed integer)



Bei diesen Dualzahlen wird im MSB (ganz links) das Vorzeichen gespeichert: 0 falls positiv, 1 falls negativ. Besonders an dieser Zahlendarstellung ist, dass negative Dualzahlen im Zweierkomplement abgelegt werden, u. a. mit dem Ziel der Beseitigung des Vorzeichens. Das Zweierkomplement bekommt man nach der Addition von 1 zum Einerkomplement, das durch Negierung aller Bits entsteht.

**Beispiel 6.7:**  $-5_{10}$  als Dualzahl

	– 00000101
Einerkomplement	– 11111010
Weglassen des Vorzeichens	11111010
Addition von 1	+ 1
Zweierkomplement	<hr/> 11111011

Für die Rückkomplementierung negativer Dualzahlen ist das gleiche Verfahren anzuwenden d. h. Einerkomplement der negativen Zahl berechnen und 1 dazu addieren.

**Beispiel 6.8:**  $10010010_2$  als Dezimalzahl

	10010010
Einerkomplement	01101101
Addition von 1	+ 1
Vorzeichen zurück	<hr/> – 01101110 = $-110_{10}$

**Beispiel 6.9:** Extremwerte der 8-bit Darstellung

Kleinste positive Zahl:	$0_{10}$	=	00000000
Größte positive Zahl:	$+127_{10}$	=	01111111
Kleinste negative Zahl:	$-1_{10}$	=	11111111
Größte negative Zahl:	$-128_{10}$	=	10000000

**Beispiel 6.10:** Überlauf vorzeichenbehafteter ganzer Zahlen

Wird in 8-Bit Darstellung zur Zahl 120 beispielsweise die Zahl 10 addiert, übersteigt das Ergebnis den darstellbaren Bereich ( $-128$  bis  $+127$ ). Wie am Zahlenrad in Abbildung 6.2 dargestellt, gerät man also in den Bereich der negativen Zahlen und erhält als Ergebnis  $-126$ . Dabei wird das so genannte **Negative-Bit** im Statusregister auf 1 gesetzt, um zu zeigen dass es bei der Addition zu einem Überlauf und entsprechendem Vorzeichenwechsel gekommen ist.

Auf den AVR-Mikrocontrollern stehen die folgenden Datentypen für vorzeichenbehaftete ganze Zahlen zu Verfügung: `int8_t` (signed char), `int16_t`, `int32_t` und `int64_t`.

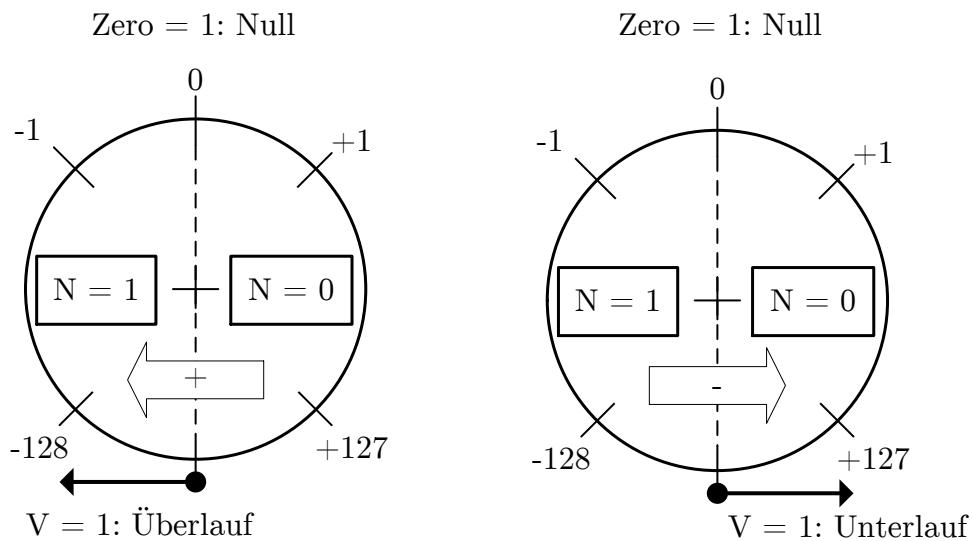


Abbildung 6.2: Addition und Subtraktion vorzeichenbehafteter Zahlen [22, S. 21].

### 6.3.3. Festkommadarstellung (fixed point)

In dieser Darstellung werden die Vorpunkt- und Nachkommastellen einer reellen Zahl hintereinander gelegt und man denkt sich den Punkt dazwischen.

**Beispiel 6.11:** 15,7510 mit 8 Nachkommastellen und 16 Dualstellen (2 Bytes)

15 in Dualzahl umwandeln:	$15_{10} = 1111_2$
0,75 in Dualzahl umwandeln:	$0,75_{10} = 0,11_2$
Gesamt:	$15,75_{10} = 1111,11_2$
Abgespeichert wird:	00001111 11000000

### 6.3.4. Gleitkommadarstellung (floating point)

In dieser Darstellung wird eine reelle Zahl mit einer normalisierten Mantisse (Absolutwert) und einem ganzzahligen Exponenten zur Basis des Zahlensystems (hier Dual) gespeichert.

**Beispiel 6.12:** Kommaverschiebung

$$15,75_{10} = 1111,11_2 = 1,11111 \cdot 2^3$$

Normalisieren bedeutet, das Komma so zu verschieben, dass es hinter der werthöchsten Ziffer steht. Der Exponent enthält dann die Anzahl der Verschiebungen. Bei Zahlen größer als 1 ist der Exponent positiv (Komma nach links schieben) und bei Zahlen kleiner 1 ist er negativ (Komma nach rechts).

Beim Datentyp float, der standardmäßig 32bits belegt, sieht die Gleitpunktdarstellung folgendermaßen aus (siehe Abb. 6.3):

- In der ersten Bitposition links: Vorzeichen der Zahl

- Folgende acht Bitpositionen: Charakteristik (Summe) aus dem Exponenten und einem Verschiebewert von  $127_{10} = 0111\ 1111_2$ , der das Vorzeichen des Exponenten beseitigt).
- Letzte 23 Bitpositionen: Mantisse (kein Zweierkomplement falls negativ).

Damit ergibt sich ein dezimaler Zahlenbereich von  $-3 \cdot 10^{38}$  bis  $+3 \cdot 10^{38}$ . Die 23-Bit lange Mantisse entspricht einer Genauigkeit von etwa sieben Dezimalstellen.

*Bemerkung:* Die führende 1 der normalisiert dargestellten Vorpunktstellen wird bei der Speicherung unterdrückt und muss bei allen Operationen wieder hinzugefügt werden.

31		24	23	16	15	8	7	0			
V	E	E	E	M	M	M	M	M			
Exponent				Mantisse							

(a) Datentyp single (32-Bit)

63		56	55	48	47	40	39	32			
V	E	E	E	M	M	M	M	M			
Exponent				Mantisse							

(b) Datentyp double (64-Bit)

Abbildung 6.3.: Gleitkommadarstellung nach IEEE754 [11].

**Beispiel 6.13:**  $10,75_{10}$  in Gleitkommadarstellung

$$\begin{aligned} +15\ 75_{10} &= +1111,1100\ 0000\ 0000\ 0000_2 \\ &= +1,111\ 1100\ 0000\ 0000\ 0000 \cdot 2^3 \text{ (normalisiert)} \end{aligned}$$

Vorzeichen: 0

Charakteristik:  $127_{10} + 3_{10} = 130_{10} = 1000\ 0010_2$

Mantisse: 1111 1000 0000 0000 000 (ohne Vorpunktstelle)

Speicher: 0 1000 0010 11110000000000000000000000000000

## 6.4. Arithmetische Operationen

### 6.4.1. Rechnung in der Festpunktdarstellung

Behandelt man in dieser Darstellung beide Anteile zusammen als eine ganze Zahl, so ergibt sich aus der Anzahl der Nachpunktstellen ein konstanter Skalenfaktor

$f = 2^{(\text{Anzahl der Nachpunktstellen})}$ , unter dessen Berücksichtigung sich reelle Festpunktzahlen mit den für ganze Zahlen vorgesehenen Operationen berechnen lassen. Es ist dabei zu

beachten, dass ein Korrekturfaktor benötigt wird um auf das Ergebnis in der Festpunkt-darstellung zurück zu kommen.

addieren:	$(a \cdot f) + (b \cdot f) = (a + b) \cdot f$	ohne Korrekturfaktor
subtrahieren:	$(a \cdot f) - (b \cdot f) = (a - b) \cdot f$	ohne Korrekturfaktor
multiplizieren:	$(a \cdot f) \cdot (b \cdot f) = (a \cdot b) \cdot f \cdot f$	Korrekturfaktor <i>Produkt/f</i>
dividieren:	$(a \cdot f)/(b \cdot f) = (a/b)$	Korrekturfaktor vor der Division:
	$(a \cdot f) \cdot f/(b \cdot f) = (a/b) \cdot f$	<i>Divident · f</i>

Da bei der Multiplikation und Division wegen des Mangels an Speicherlänge Nachpunktstellen verloren gehen können, kann der Übergang von einem festen zu einem variablen Skalenfaktor (der mit abgespeichert wird) sinnvoll sein.

### 6.4.2. Rechnung in Gleitpunktdarstellung

Dank der exponentenbasierte Darstellung lassen sich Zahlen in der Gleitpunktdarstellung unkompliziert addieren, subtrahieren, multiplizieren und dividieren. Es ist nur erforderlich die Mantissen der Ergebnisse wieder zu normalisieren.

Vor Additionen und Subtraktionen wird der Exponent der kleineren Zahl durch Verschiebung der Mantisse an den Exponenten der größeren Zahl angepasst.

**Beispiel 6.14:** Addition in Gleitpunktdarstellung

$$12 \cdot 10^4 + 34 \cdot 10^2 \Rightarrow 12 \cdot 10^4 + 0.034 \cdot 10^4 \Rightarrow 1234 \cdot 10^4$$

Bei Multiplikationen werden die Exponenten addiert und die Mantissen multipliziert.  
Bei Divisionen subtrahiert man die Exponenten und dividiert die Mantissen.

**Beispiel 6.15:** Multiplikation in Gleitpunktdarstellung

$$12 \cdot 10^4 * 12 \cdot 10^3 \Rightarrow (12 \cdot 12) \cdot 10^{2+3} \Rightarrow 144 \cdot 10^5$$

$$144 \cdot 10^5 / 12 \cdot 10^2 \Rightarrow (144/12) \cdot 10^{5-2} \Rightarrow 12 \cdot 10^3$$

Verfügt ein Mikroprozessor nicht über eine so genannte **Floating Point Unit (FPU)** für das Rechnen mit Fließkommazahlen, müssen die Fließkommaoperationen durch viele Einzeloperationen „nachgebaut“ werden. Dies ist sehr aufwändig, entsprechend langsam ist das Rechnen mit Fließkommazahlen auf solchen Prozessoren.

### 6.4.3. Ergebniss-Analyse mit dem Statusregister SREG

Arithmetische und logische Berechnungen werden von der ALU der CPU ausgeführt. Diese gibt dabei neben dem Ergebnis auch Statusinformationen im Register SREG (siehe Abb. 6.4) aus, die zur Bewertung und Analyse des Ergebnisses dienen.

Falls nach einer arithmetischen Operation beispielsweise ein Überlauf bzw. Unterlauf auftritt, wird dies mit einer 1 im **Carrybit (C)** angezeigt - dies kann beispielsweise als Zwischenübertrag verwendet werden oder auch zur Anzeige eines Fehlers (falls der Über-/Unterlauf unerwünscht war). Das **Zerobit (Z)** wird genau dann auf eins gesetzt, wenn das Operationsergebnis null ist, **Negative (N)**, wenn das Operationsergebnis

Bit	7	6	5	4	3	2	1	0	
	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W								
Initial Value	0	0	0	0	0	0	0	0	

Abbildung 6.4.: Status-Register SREG [5, S. 11].

negativ ist. Darüber hinaus existieren das **Two's Complement Overflow (V)**, **Sign (S)** und **Half Carry (H)** Bit, deren genaue Bedeutung bei Anwendung einer gegebenen Operation dem Befehlssatz der CPU zu entnehmen ist.

**Beispiel 6.16:** Rechnung mit Zero- und Carrybit

$$\begin{array}{rcl} 250_{10} & = & 1111\ 1010 \\ + \quad 8_{10} & = & 0000\ 1000 \\ & & 0000\ 0010 \quad \text{Zero} = 0, \text{Carry} = 1 \end{array}$$

$$\begin{array}{rcl} 250_{10} & = & 1111\ 1010 \\ + \quad 6_{10} & = & 0000\ 0110 \\ & & 0000\ 0000 \quad \text{Zero} = 1, \text{Carry} = 1 \end{array}$$

$$\begin{array}{rcl} 2_{10} & = & 0000\ 0010 \\ - \quad 3_{10} & = & 0000\ 0011 \\ & & 1111\ 1111 \quad \text{Zero} = 0, \text{Carry} = 1 \end{array}$$

**Beispiel 6.17:** Unterschiedliche Befehle für ganze Zahlen und Gleitkommazahlen

Auf dem Mikrocontroller existieren einige mathematische Befehle sowohl in einer ganzzahligen als auch einer Fließkommavariante. Ein Beispiel hierfür ist der Betrags-Befehl:

- **abs()**: Absolutwert einer ganzen Zahl
- **fabs()**: Absolutwert einer Gleitkommazahl

**Beispiel 6.18:** Vergleiche mit Fließkommazahlen

Durch die Begrenzte Rechengenauigkeit der Fließkommazahlen ist es schwierig einen exakten Zahlenwert in einem Vergleich zu treffen. Deshalb sollten Vergleiche einen kleinen Bereich um den Zielwert abdecken. Statt

if ( $a == 0$ )

sollte also z.B. besser

if ( $\text{fabs}(a) < 0.001$ )

verwendet werden.

# 7. Analog/Digital-Wandlung

Um den digitalen Mikrocontroller mit der analogen Welt außerhalb (Messungen, Aktoren...) zu verbinden, benötigt man eine Schnittstelle: den sogenannten Analog-Digital- (A/D-) bzw. Digital-Analog- (D/A-) Wandler. In diesem Kapitel werden die Grundlagen der A/D-Wandlung sowie ihre Implementierungsdetails bei der Atmel Mega-Familie präsentiert.

## 7.1. Signalklassen

Elektrische, informationstragende Signale lassen sich in vier Klassen kategorisieren:

- Ein Signal wird als **analog** bezeichnet, wenn es jeden Wert stufenlos zwischen einem Minimum und einem Maximum annehmen kann, d.h. die informationstragende Größe ist **wert- und zeitkontinuierlich**.
- Ein Signal heißt **zeitdiskret**, wenn sein stufenloser Wert nur zu **diskreten Zeitwerten** bekannt ist. Die **Abtastung** überführt ein zeitkontinuierliches Signal in ein **zeitdiskretes Signal**.
- Entsprechend nennt man ein Signal **wertdiskret**, wenn es nur **diskrete Werte** einnehmen kann, ein Wechsel des Wertes aber zu beliebigen Zeitpunkten erfolgen kann. Die **Quantisierung** wandelt ein wertkontinuierliches Signal in ein **wertdiskretes Signal** um.
- Ein **digitales** Signal ist **sowohl zeitdiskret also auch wertdiskret**.

## 7.2. Umrechnung, Auflösung und Genauigkeit

Die Aufgabe eines A/D-Wandlers ist es ein **analoges Signal zu digitalisieren** (also **abzutasten** und zu **quantisieren**), d. h. das analoge Signal wird zu jedem Abtastzeitpunkt gelesen

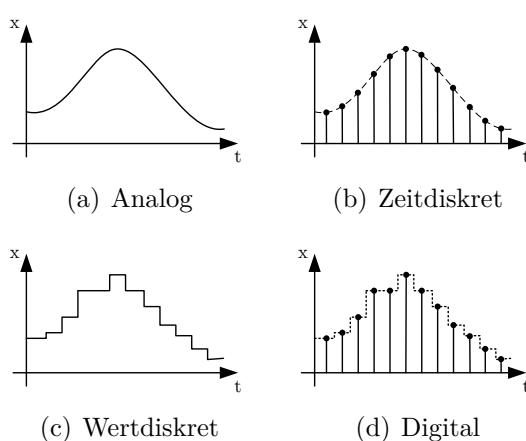


Abbildung 7.1.: Signalklassen.

und in eine Binärzahl, die der Auflösung des Wandlers entspricht, umgewandelt. Die drei wichtigsten Charakteristiken eines A/D-Wandlers sind seine Auflösung, sein Fehler und seine Geschwindigkeit.

### 7.2.1. Auflösung

Die Auflösung eines A/D-Wandlers ist die Anzahl der Bits, die er am Ausgang liefert. Typische Werte sind 8, 10, 12 oder 16 Bits. Sie beschreibt, wie fein gestuft wird, d. h. wie genau eine analoge Größe nach der Umwandlung digital dargestellt werden kann. Ein A/D-Wandler mit einer 8-Bit Auflösung bietet  $2^8 = 256$  Wertstufen zur Digitalisierung einer Größe.

Die Auflösung lässt sich auch in der Einheit des gewandelten Signals, also in Volt bei einer Spannung, ausdrücken. Sie entspricht der minimalen benötigten Spannung, die zu einer Wertänderung am Ausgang des A/D-Wandlers führt, also dem Wert des Least Significant Bit (LSB). Die Auflösung in Volt lässt sich durch Division des Spannungsbereichs durch die Zahl der Wertstufen berechnen.

#### *Beispiel 7.1: Kleinste darstellbare Spannung*

$$\begin{aligned} & \text{12-Bit Auflösung, } 0 \text{ V bis } 5 \text{ V Quelle} \\ \Rightarrow & Q = 5 \text{ V} / 2^{12} = 1,22 \text{ mV} = \text{Wert des LSB} \end{aligned}$$

$$\begin{aligned} & \text{10-Bit Auflösung, } 0 \text{ V bis } 5 \text{ V Quelle} \\ \Rightarrow & Q = 5 \text{ V} / 2^{10} = 4,88 \text{ mV} = \text{Wert des LSB} \end{aligned}$$

#### *Beispiel 7.2: ADC-Wert berechnen*

Welchen binären Wert liest ein 8-Bit ADC, wenn eine Spannung von 0,32 V anliegt? Die Referenzspannung beträgt  $V_{ref} = 2,56 \text{ V}$ .

Der ADC-Wert kann mittels Dreisatz bestimmt werden:

$$ADC = 255 \cdot \frac{0,32 \text{ V}}{2,56 \text{ V}} = 31,875 \quad (7.1)$$

Im ADC-Register steht also der Wert 31, oder binär geschrieben 00011111.  
a

<sup>a</sup>In diesem Fall kann man auch sehen, dass 0,32 V genau ein achtel der Referenzspannung ist. Dementsprechend würde man einfach den Maximalwert 11111111 um 3 Stellen nach rechts verschieben und erhält 00011111.

### 7.2.2. Fehler

Der durch die A/D-Wandlung entstehende Gesamtfehler zwischen dem analogen Wert und dem gewandelten Digitalwert hat im Wesentlichen drei Komponenten:

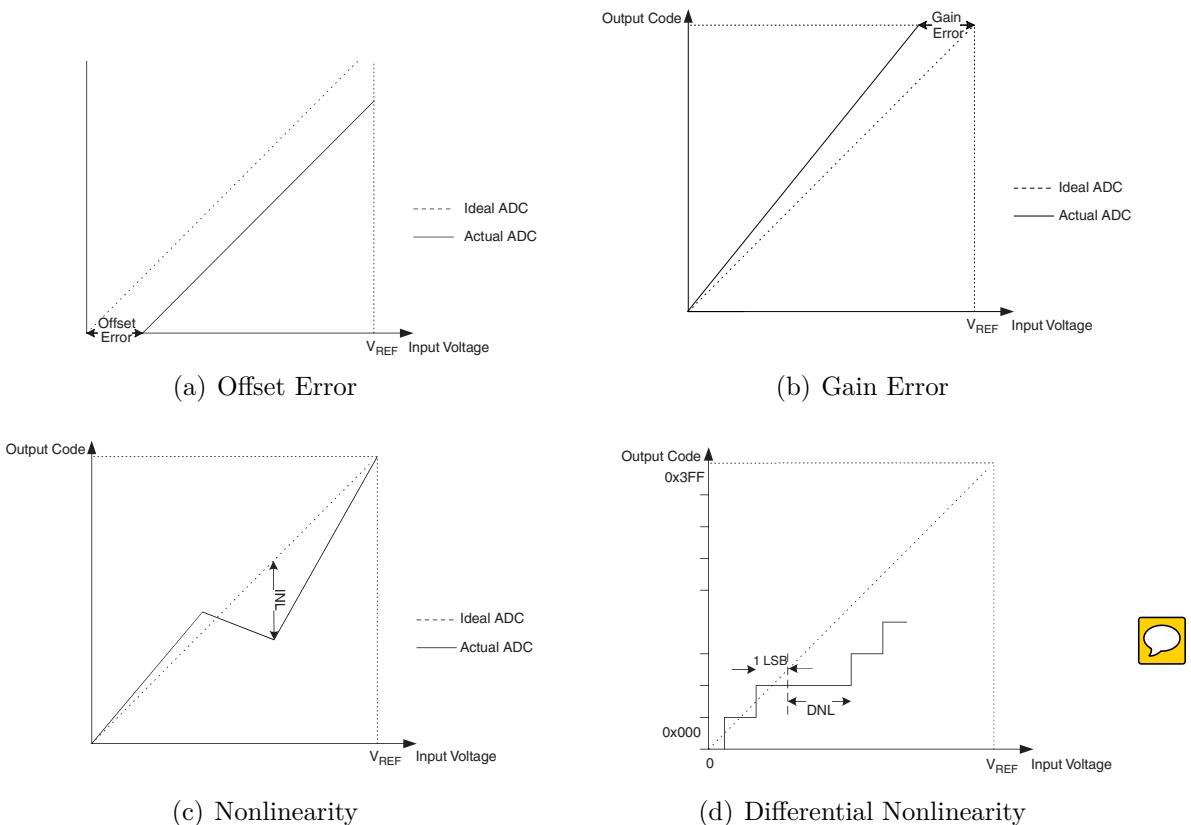


Abbildung 7.2.: Fehler bei der A/D-Wandlung [5, S. 197f].

- den Quantisierungsfehler, der dem Auflösungsfehler entspricht und normalerweise zwischen 0 und 0,5LSB variiert,
- den Effekt der Nichtlinearität der beteiligten Hardware-Komponenten und
- den Abtastfehler, der stark von der Frequenz des analogen Signals abhängig ist. Dieser Fehler spielt nur bei hochfrequenten Signalen eine Rolle und wird sonst vom Quantisierungsfehler überdeckt.

Eine graphische Darstellung der Umwandlungsfehler ist in Abbildung 7.2 gegeben.

**Beispiel 7.3:** Datenblatt ATmega16

„0.5 LSB Integral Non-linearity“ und „Absolut accuracy:  $\pm 2$  LSB“.

Man sagt dann salopp, dass zwei Bits „flackern“. Dieses Verhalten ist bei der Implementierung zu berücksichtigen.

Eine gewisse Glättung der Messergebnisse kann durch optimale Stabilisierung der Stromversorgung des Wandlers erzielt werden, z.B. durch einen zusätzlichen Abblockkondensator direkt an den Pins des A/D-Wandlers.

### 7.2.3. Geschwindigkeit

Die Geschwindigkeit eines A/D-Wandlers hängt sehr stark von seiner hardwaremäßigen Realisierung ab. Hauptfaktoren sind die Erfassungszeit (das Signal muss währenddessen unverändert bleiben), die Umwandlungszeit, d.h. die benötigte Zeit die für das Errechnen des digitalen Werts, und die Transferzeit, d.h. die benötigte Zeit, um den digitalen Wert im ADC-Register des Mikrocontrollers zu speichern. Diese drei Zeiten zusammen definieren die Abtastfrequenz eines A/D-Wandlers, welche der Anzahl der Messungen entspricht, die er pro Zeiteinheit verarbeiten kann. Meist ist die Geschwindigkeit im Datenblatt in KiloSamples Per Second (**kSPS**) angegeben.



**Beispiel 7.4:** Datenblatt ATmega16

„Up to 15 kSPS at maximum Resolution“ und  
„Conversion time: 13-260  $\mu$ s“.

Obwohl die meisten Mikrocontroller der AVR-Serie nur einen A/D-Wandler besitzen, erlauben sie durch Mehrfachnutzung (**Multiplexing**) die Wandlung von bis zu acht analogen Signalen bzw. Kanälen, indem immer nur ein Kanal mit dem Wandler verbunden wird und nach Abschluss der Messung ein anderer Kanal ausgewählt werden kann.

## 7.3. A/D Wandler im AVR

Das Analog-Digital-Converter Control and Status Register ADCSRA (Abb. 7.3) schaltet den Wandler ein (ADC Enable: ADEN-Bit), startet eine einzelne Wandlung (ADC Start Conversion: ADSC-Bit) oder versetzt den Konverter in den Free-Run-Modus (ADFR-Bit). Mit dem ADIE-Bit (ADC Interrupt Enable) wird der Interrupt lokal freigeschaltet, ADIF (*ADC Interrupt Flag*) ist das zugehörige Flag, das bei Abschluss einer Wandlung aktiviert wird. Mit den Bits ADPS0 bis ADPS2 wird ein *Prescaler* für die Geschwindigkeit der Wandlung festgelegt.

Mithilfe des Registers PrescalerADMUX wird dem *Multiplexer* mitgeteilt, welcher Kanal gemessen und welche Spannung als Referenz verwendet werden soll (siehe Abb. 7.4).

Die Register ADCH und ADCL beinhalten das Ergebnis der Messung, je nach Wert des ADLAR-Bits (Left Adjust Result) in den 10MSB oder 10LSB, siehe Abbildung 7.5. Werden nur 8-Bit Auflösung benötigt, ist es günstig ADLAR=1 zu wählen, da dann nur das ADCH-Register ausgelesen werden muss.

**ADC Control and Status Register A – ADCSRA**

Bit	7	6	5	4	3	2	1	0	ADCSRA
Read/Write	R/W								
Initial Value	0	0	0	0	0	0	0	0	

(a) ADCSRA Register

ADPS2	ADPS1	ADPS0	Division Factor
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

(b) ADC Prescaler Selections

Abbildung 7.3.: ADC Control and Status Register ADCSRA [5, S. 200].

**ADC Multiplexer Selection Register – ADMUX**

Bit	7	6	5	4	3	2	1	0	ADMUX
Read/Write	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

(a) ADMUX Register

MUX3..0	Single Ended Input
0000	ADC0
0001	ADC1
0010	ADC2
0011	ADC3
0100	ADC4
0101	ADC5

(b) Analog Chanel Selection Bits

0	0	AREF, Internal $V_{ref}$ turned off
0	1	$AV_{CC}$ with external capacitor at AREF pin
1	0	Reserved
1	1	Internal 2.56V Voltage Reference with external capacitor at AREF pin

(c) Voltage Reference Selection

Abbildung 7.4.: ADMUX Register [5, S. 199f].

*ADLAR = 0*

Bit	15	14	13	12	11	10	9	8	ADCH	ADCL
	—	—	—	—	—	—	ADC9	ADC8		
	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0		
	7	6	5	4	3	2	1	0		
Read/Write	R	R	R	R	R	R	R	R		
	R	R	R	R	R	R	R	R		
Initial Value	0	0	0	0	0	0	0	0		
	0	0	0	0	0	0	0	0		

*ADLAR = 1*

Bit	15	14	13	12	11	10	9	8	ADCH	ADCL
	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2		
	ADC1	ADC0	—	—	—	—	—	—		
	7	6	5	4	3	2	1	0		
Read/Write	R	R	R	R	R	R	R	R		
	R	R	R	R	R	R	R	R		
Initial Value	0	0	0	0	0	0	0	0		
	0	0	0	0	0	0	0	0		

Abbildung 7.5.: ADC Data Register [5, S. 201].

# 8. Die H-Brücke

Viele mechatronische Systeme weisen einen Elektromotor auf, der für die Bewegung des Systems verantwortlich ist und dessen Ansteuerung die zentrale Aufgabe der Regelungstechniker darstellt. Eine weit verbreitete Möglichkeiten für die Regelung bzw. Steuerung eines elektrischen Motors bietet die H-Brücke. Diese ermöglicht die Einstellung der Drehrichtung, in Verbindung mit einer PWM auch der Drehgeschwindigkeit, sowie das Bremsen des Motors auf einfache Art und Weise.

## 8.1. Grundgerüst

Eine H-Brücke besteht aus vier Schaltern mit dem zu steuernden Motor in der Mitte; die Herkunft der Bezeichnung ist leicht aus Abbildung 8.1 ersichtlich.

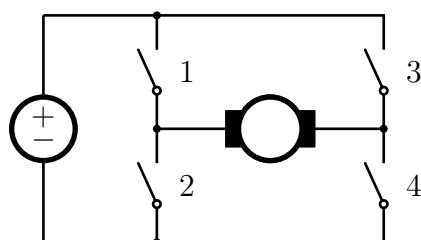


Abbildung 8.1.: Grundgerüst einer H-Brücke (Alle Schalter geöffnet, Leerlauf).

## 8.2. Betriebsmodi

Legt man wie in Abbildung 8.1 gezeigt eine Betriebsspannung an, geschieht zunächst gar nichts, solange die Schalter geöffnet sind. Dieser Betriebsmodus wird als Leerlauf (*Coast Mode*) bezeichnet.

Schließt man Schalter 1 und 4 wie in Abbildung 8.2(a), dann fließt Strom über den Motor und lässt ihn in eine Richtung drehen (hier rechts angenommen). In Abbildung 8.2(b) wurden die Schalter 1 und 4 geöffnet und die Schalter 2 und 3 geschlossen. Der durch den Motor fließende Strom wechselt seine Richtung und lässt den Motor nach links drehen.

Das gleichzeitige Schließen von Schalter 1 und 2 bzw. von 3 und 4 (siehe Abb. 8.3(a)) führt zu einem Kurzschluss, der die Schaltung zerstören und zu einer Explosion führen kann! Es muss daher sichergestellt werden, dass eine solche Kombination nie auftreten kann. Dazu wird oft eine Logikschaltung zwischen Steuereinheit (z.B. Mikrocontroller) und H-Brücke eingeführt, die nur die gewünschten Schalterkonstellationen zulässt.

Das Schließen von Schalter 1 und 3 bzw. von 2 und 4 schließt die Anschlüsse des Motors kurz, wobei die Stromquelle nicht zugeschaltet ist (siehe Abbildung 8.3(b)). Wird dann durch Drehung des Motors eine Spannung induziert (Generator), so kann diese direkt durch die Schleife abfließen, wobei aufgrund des ohmschen Widerstands des Motors die Leistung in Wärme umgewandelt und der Motor gebremst wird (*Brake Mode*).

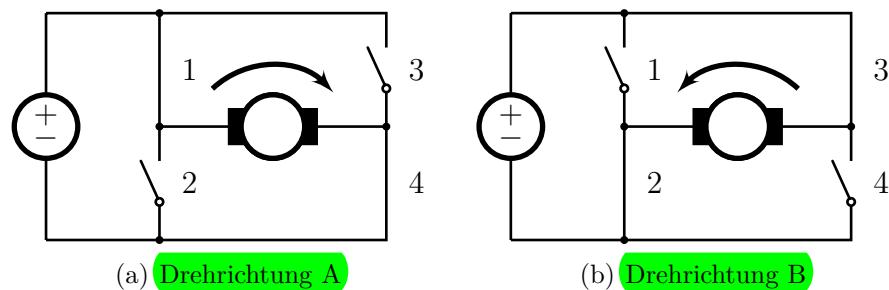


Abbildung 8.2.: H-Brücke bei verschiedenen Drehrichtungen.

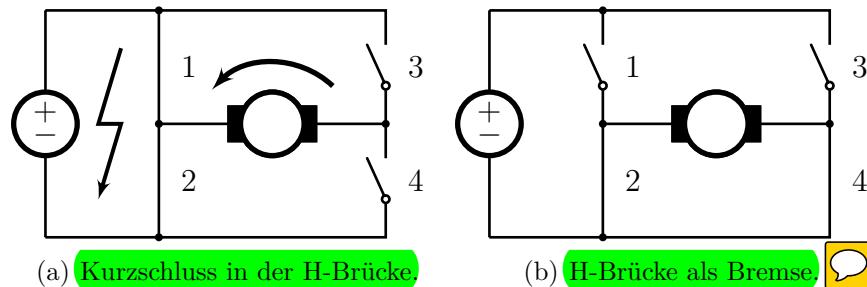


Abbildung 8.3.: H-Brücke in weiteren Konfigurationen.

Wichtig ist in jeder Schaltkombination, dass alle Bauteile die auftretenden Belastungen aushalten.

## 8.3. Realisierung mit Transistoren

Um von allen möglichen Vorteilen der H-Brücke zu profitieren, sollte man sie so in Hardware realisieren, dass das Öffnen und Schließen von den Schaltern beliebig kombinierbar und so schnell wie möglich ist. Dazu werden meist Transistoren als Schalter verwendet.

### 8.3.1. Was ist ein Transistor?

Ein Transistor (**Transfer Resistor**) ist ein elektronisches Bauelement zum Schalten und Verstärken von elektrischen Signalen und hat meist drei Pins. Durch das Anlegen einer ausreichenden Spannung an einem der Pins kann man den Stromfluss bzw. die Spannung zwischen den anderen zwei Pins beeinflussen. Folglich kann man ein solches Bauelement als Schalter oder als Verstärker betreiben, weshalb es zum Aufbau einer H-Brücke geeignet ist. Zwei Arten von Transistoren sind am häufigsten auf dem Markt anzutreffen: der Bipolartransistor (BJT) und der MOSFET.

### 8.3.2. Bipolartransistoren (BJTs)

Ein Bipolartransistor (**Bipolar Junction Transistor, BJT**) ist ein Transistor mit drei Pins, nämlich Kollektor (C), Emitter (E) und Basis (B). Er existiert in zwei Ausführungen, die

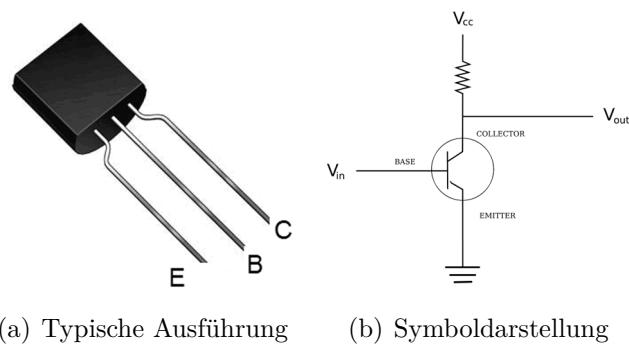


Abbildung 8.4.: Bipolartransistor.

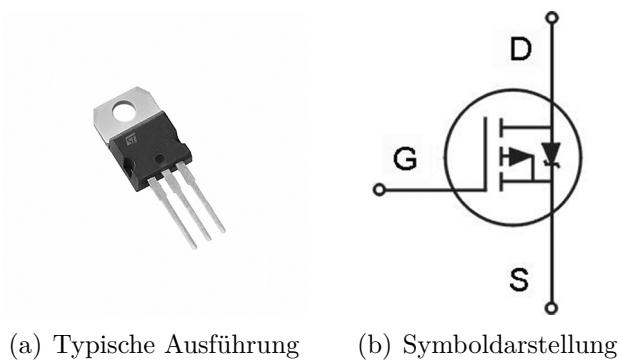


Abbildung 8.5.: Metall Oxid Feldeffekt Transistor (MOSFET).

sich in der Art und Anordnung der Halbleiter-Schichten unterscheiden und sich komplementär verhalten: *npn* und *pnp*. Die Basis entspricht stets dem Steuereingang des Transistors. Bei einem *npn*-Transistor führt ein höherer Stromfluss zwischen Basis und Emitter zu einem höheren Stromfluss zwischen Kollektor und Emitter; der Transistor „schaltet (durch)“. Dies ermöglicht einen Stromfluss zwischen Kollektor und Emitter  $I_{CE}$ , der den „Steuerstrom“  $I_{BE}$  durch Basis und Emitter um ein Vielfaches übertrifft. Idealisiert kann man die Kennlinie des Transistors im s.g. *Sättigungsbetrieb* als Schalter annehmen.

### 8.3.3. MOSFET

Der **Metalloxid-Halbleiter-Feldeffekttransistor** (Metal Oxide Semiconductor Field-Effect Transistor, MOSFET auch MOS-FET) ist ein Transistor mit 4 Pins, nämlich dem Gate (G), Drain (D), Source (S) und Bulk (B). Meist ist der Bulk jedoch intern mit Source verbunden, was zu einem 3-pin-Transistor führt. Die Spannung am Gate steuert den Stromfluss zwischen Source und Drain. Allerdings wird die Änderung des Transistorwiderstands im Gegensatz zum BJT nicht durch einen Stromfluss durchs Gate erzeugt, sondern nur durch das elektrische Feld, das die Gate-Spannung im Transistor erzeugt (daher der Name Feldeffekt-Transistor).

### 8.3.4. MOSFET vs. BJT

Die stromgesteuerten Bipolartransistoren erreichen teils wesentlich höhere Schaltgeschwindigkeiten als die spannungsgesteuerten MOSFETs und kommen deshalb in Hochfrequenzschaltungen bevorzugt zum Einsatz. Allerdings weisen sie typischerweise einen deutlich höheren Innenwiderstand auf und bewirken daher viel höhere Verluste.

Da die Verlustleistung in Wärme umgesetzt wird, ist eine Kühlung des Transistors eher notwendig als bei Einsatz eines MOSFETs. Aus diesem Grund sind letztere für den Aufbau einer H-Brücke meist zu bevorzugen, da die PWM zur Drosselung des Motors üblicherweise eine Frequenz aufweist, die ein MOSFET problemlos bewältigen kann.

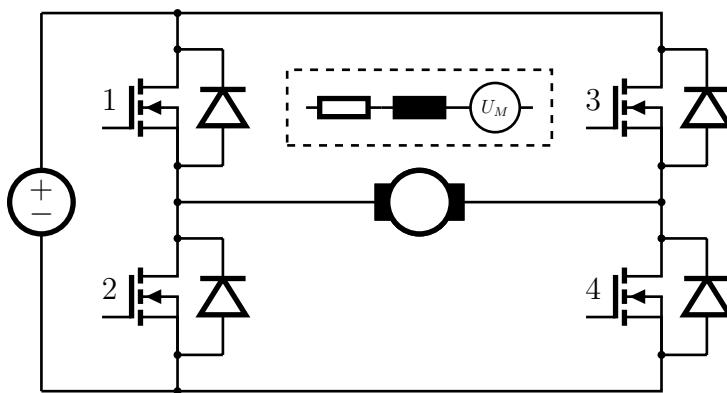


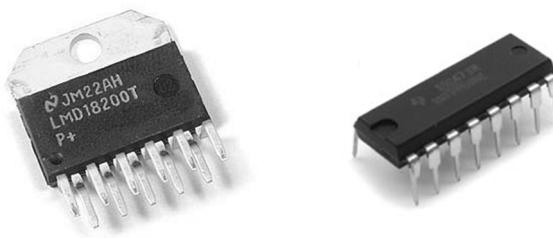
Abbildung 8.6.: Schaltung einer H-Brücke mit MOSFETs und Freilaufdioden.

Eine typische Schaltung einer H-Brücke mit MOSFETs ist in Abbildung 8.6 zu sehen. Wie in der Mitte der Abbildung 8.6 (gestrichelter Rahmen) zu sehen ist, enthält das Modell eines Motors eine Spule, deren Spannung proportional zu ihrer Induktivität und der Ableitung des durch sie fließenden Stroms ist. Falls im Betrieb des Motors ein Transistor ausgeschaltet wird (was bei einer PWM sehr häufig geschieht), induziert die Motorspule eine große Spannung falscher Polung, die den Transistor beschädigen kann. Um die Transistoren zu schützen, fügt man eine sogenannte Freilauf-Dioden ein (D1-D4 in Abbildung 8.6) ein, durch welche der induzierte Strom abfließen kann.

## 8.4. Single-Chip H-Brücken und Selbstbau

Es existieren zahlreiche fertige H-Brücken-Chips für die Ansteuerung von verschiedenen Arten (DC, Stepper...) und Größen (0,1 A bis 10 A, 4 V bis 36 V) von Motoren, z.B. LMD18200 (bis 6 A) von National Semiconductors oder SN754410 von Texas Instruments (1 A, bis 36 V).

Der Preis eines solchen Chips bewegt sich zwischen 5 und 20 €. Es ist jedoch auch ohne weiteres möglich, die H-Brücke selbst zu bauen. Die Kosten für vier MOSFET Transistoren (z. B. BUZ 11, IRF 1310, ...), Dioden (z. B. N4001), liegen bei 1-5€.



(a) LMD182200

(b) SN754410

Abbildung 8.7.: Typische H-Brücken Chips.

## 8.5. Die H-Brücke und der Mikrocontroller

Die Rolle des Mikrocontrollers bei der Steuerung eines Motors mit einer H-Brücke beschränkt sich auf die Ansteuerung der Gates der vier Transistoren. Durch das vom Mikrocontroller erzeugte PWM-Signal steuert man das Schalten der Transistoren (Reihenfolge und Geschwindigkeit), so dass ein Wunschverhalten des Motors erreicht wird, z.B. Linksdrehung mit 50% Geschwindigkeit.

Oft wird eine Logikschaltung zwischen den Mikrocontroller und die H-Brücke geschaltet, um die Ansteuerung der vier Transistoren beim Rechts- bzw. Linksdrehen mit überlagerter PWM bzw. beim Bremsen oder im Leerlauf zu erleichtern (diese vier Betriebsmodi lassen sich über zwei Leitungen codieren) und überdies Kurzschlüsse durch fehlerhafte geschaltete Pins oder bei einem Ausfall des Mikrocontroller auszuschließen (siehe Abb. 8.8).

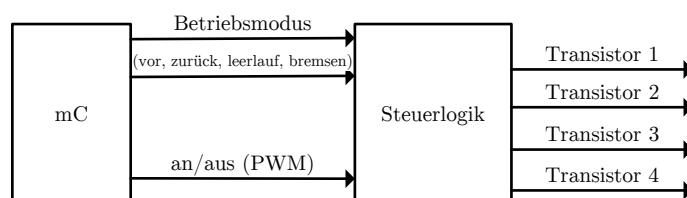


Abbildung 8.8.: Signalfluss zwischen Mikrocontroller und H-Brücke.

# 9. Zeitdiskrete Regelung

## 9.1. Einleitung

Inhalt dieses Praktikums ist die Realisierung zeitdiskreter Steuerungen und Regelungen auf digitaler Hardware. Dieses Kapitel versucht die in diesem Zusammenhang wichtigsten systemtheoretischen Grundlagen zu wiederholen bzw. kompakt zusammenzufassen, um die praktische Umsetzung einer zeitdiskreten Regelung auf einem Mikrocontroller zu ermöglichen. Insbesondere sollen grundsätzliche Gemeinsamkeiten und Unterschiede zeitkontinuierlicher und -diskreter Systeme, sowie Besonderheiten bei der zeitdiskreten Regelung kontinuierlicher Systeme herausgestellt werden. Im Folgenden wird ausschließlich auf **lineare** Systeme und deren Regelung eingegangen.

*Finally, we make some remarks on why linear systems are so important. The answer is simple: Because we can solve them!*

-Richard Feynman<sup>1</sup>

## 9.2. Zeitdiskrete Modellbeschreibung

In diesem Abschnitt wird auf wichtige Beschreibungsformen der Dynamik zeitdiskreter Systeme eingegangen.

### 9.2.1. Differenzengleichung

Im zeitkontinuierlichen Fall lassen sich Systeme im Zeitbereich durch Differentialgleichungen beschreiben. Als Pendant bieten sich im zeitdiskreten Fall Differenzengleichungen zur Systembeschreibung an:

$$y(k+N) + a_{N-1}y(k+N-1) + \dots + a_0y(k) = b_M u(k+M) + b_{M-1}u(k+M-1) + \dots + b_0u(k) \quad (9.1)$$

Diese Gleichung beschreibt die **lineare, zeitinvariante Dynamik** einer (Ausgangs-)Größe  $y$  abhängig von der (Eingangs-)Größe  $u$ . Dabei entspricht  $y(k)$  dem Wert von  $y$  zum Zeitpunkt  $k \cdot \Delta t$ , wobei  $k \in \mathbb{N}^+$ ,  $a_i$  und  $b_i \in \mathbb{R}$  und  $\Delta t \in \mathbb{R}^+$  das Zeitintervall zwischen zwei aufeinander folgenden diskreten Zeitpunkten darstellt.

### 9.2.2. Z-Übertragungsfunktion

Ähnlich wie im zeitkontinuierlichen Fall ist es auch im zeitdiskreten möglich eine **Systembeschreibung im Frequenzbereich** anzugeben. Bei ersterem nutzt man die Laplace-Transformation um die Differentialgleichung in den Frequenzbereich zu transformieren

<sup>1</sup> Richard Phillips Feynman (geboren am 11. Mai 1918 in Queens, New York; gestorben am 15. Februar 1988 in Los Angeles) war ein US-amerikanischer Physiker und Nobelpreisträger des Jahres 1965.

und erhält dadurch die komplexe Übertragungsfunktion  $G(s)$ . Im Zeitdiskreten erledigt dies die Z-Transformation, die für eine Folge  $x(k)$ ,  $k = 0, \dots, \infty$  wie folgt definiert ist:<sup>2</sup>

$$X(z) = \sum_{k=0}^{\infty} x(k)z^{-k} \quad (9.2)$$

Wendet man diese Transformation auf die Differenzengleichung (9.1) an so erhält man die Z-Übertragungsfunktion des zeitdiskreten Systems:

$$G(z) = \frac{Y(z)}{U(z)} = \frac{b_M z^M + b_{M-1} z^{M-1} + \dots + b_0}{z^N + a_{N-1} z^{N-1} + \dots + a_0} \quad (9.3)$$

Mit der Z-Übertragungsfunktion lassen sich wichtige Aussagen über die Eigenschaften des zeitdiskreten Systems gewinnen. Beispielsweise kann man (fast) genau wie im zeitkontinuierlichen Fall an den Polen die E/A-Stabilität des Systems erkennen. Damit das System stabil ist, müssen die Pole nun allerdings nicht in der linken komplexen Halbebene, sondern im Einheitskreis um den Ursprung der komplexen Ebene liegen. Außerdem erhält man für  $z = 1$  die stationäre Verstärkung des Systems  $V = G(z = 1)$ .<sup>3</sup>

Für weitere Eigenschaften der Z-Transformation und Z-Übertragungsfunktion, die an dieser Stelle nicht alle behandelt werden sollen, wird auf entsprechende Literatur, z.B. [17], verwiesen. Als nächstes wollen wir auf zeitdiskrete Zustandsraummodelle und deren Systemeigenschaften eingehen.

### 9.2.3. Zeitdiskrete Zustandsraummodelle

Im zeitkontinuierlichen stellen Zustandsraummodelle v.a. für die Regelungstechnik eine sehr hilfreiche Form der Systembeschreibung dar. Zeitdiskrete Systeme lassen sich auch durch Zustandsraummodelle beschreiben, mit dem Unterschied, dass die Dynamik des Zustandsvektors nun durch eine Differenzengleichung anstelle einer Differentialgleichung beschrieben wird. Wir betrachten im Folgenden ein zeitdiskretes SISO-System, um die grundsätzlichen Zusammenhänge möglichst leicht zu vermitteln:

$$\begin{aligned} \mathbf{x}_{k+1} &= \mathbf{A} \cdot \mathbf{x}_k + \mathbf{b} \cdot u_k \\ y_k &= \mathbf{c}^T \mathbf{x}_k \end{aligned} \quad (9.4)$$

mit:  $\mathbf{A} \in \mathbb{R}^{n \times n}$ ,  $\mathbf{b} \in \mathbb{R}^n$  und  $\mathbf{x}_k := \mathbf{x}(k \cdot \Delta t) \in \mathbb{R}^n$

Für den Mehrgrößenfall wird auf die reichhaltige Literatur verwiesen, z.B. [7, 9, 20, 17].

#### Stabilität

Die wichtigste Struktureigenschaft des obigen dynamischen Systems ist seine Stabilität. Um zu beurteilen, ob das System eine stabile Eigendynamik aufweist, setzen wir den

<sup>2</sup>In [17] Kap. 12.1.1 wird auf den Zusammenhang zur Laplace-Transformation eingegangen und gezeigt, dass die Z-Transformation eine spezielle Form der Laplace-Transformation ist. Außerdem wird hier gezeigt, dass zwischen  $z$  und  $s$  der Zusammenhang  $z = e^{s\Delta t}$  besteht.

<sup>3</sup>Vergleich: Im zeitkontinuierlichen Fall muss man  $s = 0$  wählen um die stationäre Verstärkung zu erhalten →  $V = G(s = 0)$

Eingang zu null und betrachten die Zustands-Differenzengleichung:

$$\mathbf{x}_{k+1} = \mathbf{A} \cdot \mathbf{x}_k \quad (9.5)$$

Nun beurteilen wir die Stabilität des Systems anhand der Frage, ob die Lösung von (9.5) asymptotisch gegen den Ursprung strebt. Eine äquivalente Forderung ist, dass die Norm der Lösung  $\|\mathbf{x}_k\|_2^2 = \mathbf{x}_k^T \cdot \mathbf{x}_k$  gegen null strebt, wenn  $k$  wächst. Wann diese Bedingung erfüllt ist, ist dem folgendem Satz zu entnehmen:

**Satz 1** Das zeitdiskrete System (9.5) ist **asymptotisch stabil**, wenn alle Eigenwerte von  $\mathbf{A}$  innerhalb des Einheitskreises liegen.

$$|\lambda_i(\mathbf{A})| < 1, \quad i = 1, 2, \dots, n \quad \Rightarrow \lim_{k \rightarrow \infty} \mathbf{x}_k^T \cdot \mathbf{x}_k \rightarrow 0 \quad (9.6)$$

Wenn lediglich gilt

$$|\lambda_i(\mathbf{A})| \leq 1, \quad i = 1, 2, \dots, n \quad (9.7)$$

so ist das System **grenzstabil**.

Ist wenigstens ein Eigenwert betragsmäßig größer eins, so ist das System **instabil** und die Norm der Lösung  $\mathbf{x}_k^T \cdot \mathbf{x}_k \xrightarrow{i.A.} \infty$  wenn  $k \rightarrow \infty$ .

Ein einfaches Beispiel veranschaulicht diesen Satz: Wir betrachten ein System mit nur einem Zustand, d.h. es sei  $x_k \in \mathbb{R}$  und das System (9.5) gleich  $x_{k+1} = a \cdot x_k$ . Das System besitzt damit genau einen Eigenwert  $\lambda = a$ . Es ist nun einfach zu erkennen, dass für alle  $|a| < 1$ ,  $|x|$  kleiner wird, für  $|a| = 1$  die Lösung betragsmäßig konstant bleibt  $\forall k$  und für  $|a| > 1$ ,  $|x|$  wächst. Für die Norm  $\|x_k\|_2^2 = x_k^2$  gilt das gleiche.

Analog zum zeitkontinuierlichen Fall lässt sich im zeitdiskreten folgendes zur Übertragungsstabilität sagen:

**Satz 2** Ist das System asymptotisch stabil, so ist es auch sicher übertragungsstabil<sup>4</sup> vom Eingang  $u$  auf den Ausgang  $y$ .

## Steuerbarkeit

Die Systeme (9.5) und (9.4) werden auch als iterative Abbildungen bezeichnet. Ist eine Anfangsbedingung  $(AB)$   $\mathbf{x}_0 \in \mathbb{R}^n$  gegeben, so ist die Lösung des zeitdiskreten Systems (9.4) leicht zu bestimmen, wir beginnen mit dem Zeitpunkt  $k = 1$ :

$$\mathbf{x}_1 = \mathbf{A} \cdot \mathbf{x}_0 + \mathbf{b} \cdot u_0$$

Für den Zeitpunkt  $k = 2$  und  $k = 3$  erhalten wir

$$\mathbf{x}_2 = \mathbf{A} \cdot \mathbf{x}_1 + \mathbf{b} \cdot u_1 = \mathbf{A} \cdot (\mathbf{A} \cdot \mathbf{x}_0 + \mathbf{b} \cdot u_0) + \mathbf{b} \cdot u_1 = \mathbf{A}^2 \cdot \mathbf{x}_0 + \mathbf{A} \cdot \mathbf{b} \cdot u_0 + \mathbf{b} \cdot u_1$$

$$\mathbf{x}_3 = \mathbf{A} \cdot \mathbf{x}_2 + \mathbf{b} \cdot u_2 = \mathbf{A}^3 \cdot \mathbf{x}_0 + \mathbf{A}^2 \cdot \mathbf{b} \cdot u_0 + \mathbf{A} \cdot \mathbf{b} \cdot u_1 + \mathbf{b} \cdot u_2$$

---

<sup>4</sup>bounded-input-bounded-output

Für einen beliebigen Zeitpunkt  $k$  gilt

$$\mathbf{x}_k = \mathbf{A}^k \cdot \mathbf{x}_0 + \mathbf{A}^{k-1} \cdot \mathbf{b} \cdot u_0 + \mathbf{A}^{k-2} \cdot \mathbf{b} \cdot u_1 + \dots + \mathbf{b} \cdot u_{k-1} = \mathbf{A}^k \cdot \mathbf{x}_0 + \sum_{i=0}^{k-1} \mathbf{A}^{k-1-i} \cdot \mathbf{b} \cdot u_i \quad (9.8)$$

In diesem Zusammenhang beantworten wir die Frage, ob das zeitdiskrete dynamische System (9.4) steuerbar ist. Unter Steuerbarkeit versteht man im zeitdiskreten Fall dasselbe wie im Zeitkontinuierlichen:

**Definition 1** Ein System ist vollständig steuerbar, wenn ein beliebiger Anfangszustand  $\mathbf{x}_0 = \mathbf{x}(t_0)$  in endlicher Zeit, durch eine geeignet gewählte Stellgröße, in einen beliebigen Endzustand  $\mathbf{x}_e = \mathbf{x}(t_e)$  überführt werden kann.

Im Zeitkontinuierlichen kann man zeigen, dass diese Forderung bereits erfüllt ist, wenn sich das System in endlicher Zeit von einem beliebigen Anfangszustand in den Endzustand  $\mathbf{x}_e = \mathbf{0}$  überführen lässt.<sup>5</sup> Daher wird die Steuerbarkeit im zeitkontinuierlichen Fall teilweise auch anhand dem letztgenannten Kriteriums definiert (vgl. Vorlesung „Systemtheorie in der Mechatronik“). Dieses Kriterium ist allerdings im Zeitdiskreten nicht mehr hinreichend, um Steuerbarkeit nach Definition 1 sicher zu erfüllen.

Allerdings lässt sich für den zeitdiskreten Fall zeigen, dass die Forderung aus Definition 1 bereits erfüllt ist, wenn sich das System in endlicher Zeit vom Anfangszustand  $\mathbf{x}_0 = \mathbf{0}$  in einen beliebigen Endzustand  $\mathbf{x}_e = \mathbf{x}(t_e)$  überführen lässt.<sup>6</sup> Um dies zu prüfen schreiben wir Gleichung (9.8) für den Endzeitpunkt  $k = e$  um zu

$$\mathbf{x}_e = \overbrace{\mathbf{A}^e \cdot \mathbf{x}_0}^{\mathbf{0}} + \underbrace{\left[ \begin{array}{cccc} \mathbf{b} & \dots & \mathbf{A}^{e-3} \cdot \mathbf{b} & \mathbf{A}^{e-2} \cdot \mathbf{b} & \mathbf{A}^{e-1} \cdot \mathbf{b} \end{array} \right]}_{(*)} \cdot \begin{bmatrix} u_{e-1} \\ u_{e-2} \\ u_{e-3} \\ \vdots \\ u_0 \end{bmatrix} \quad (9.9)$$

Das System ist genau dann steuerbar, wenn die Spalten der Matrix (\*) den  $n$ -dimensionalen Zustandsraum aufspannen, d.h. der Rang von (\*) muss gleich  $n$  sein. Da sich alle höheren Potenzen von  $\mathbf{A}$  als Linearkombination von  $\mathbf{A}^0, \mathbf{A}^1, \dots, \mathbf{A}^{n-1}$  darstellen lassen<sup>7</sup>, können die für  $e > n$  zusätzlich zu den ersten  $n$  Spalten  $\mathbf{b}, \mathbf{Ab}, \dots, \mathbf{A}^{n-1}\mathbf{b}$  auftretenden Spalten keine Rangerhöhung der Matrix bewirken. Deshalb reicht es aus, die Matrix (\*) für  $e = n$  zu betrachten. Dies stellt gerade die Steuerbarkeitsmatrix  $Q_S$  dar, wie man sie aus dem kontinuierlichen Fall bereits kennt. Die hieraus resultierende Erkenntnis, halten wir im folgenden Satz fest.

**Satz 3** Das zeitdiskrete System (9.4) ist steuerbar, wenn die Matrix  $Q_S$ ,

$$Q_S = \left[ \begin{array}{ccccc} \mathbf{b} & \dots & \mathbf{A}^{n-3} \cdot \mathbf{b} & \mathbf{A}^{n-2} \cdot \mathbf{b} & \mathbf{A}^{n-1} \cdot \mathbf{b} \end{array} \right] \quad (9.10)$$

vollen Rang  $n$  hat. Dann existiert eine Steuerung

$$\left[ \begin{array}{ccccc} u_{n-1} & u_{n-2} & \dots & u_0 \end{array} \right]^T = Q_S^{-1} \cdot \mathbf{x}_e \quad (9.11)$$

die das System vom Ursprung in den Endzustand  $\mathbf{x}_e$  überführt.

<sup>5</sup>Dies gilt für lineare, zeitinvariante Systeme.

<sup>6</sup>Diese Systemeigenschaft wird in der Literatur auch als Erreichbarkeit bezeichnet.

<sup>7</sup>Info: Dies lässt sich mit dem Satz von Cayley-Hamilton zeigen.

**Bemerkung 1** Anhand von Gleichung (9.9) kann man sich überlegen, dass es dann auch möglich ist von jedem beliebigen Anfangszustand in jeden beliebigen Endzustand zu gelangen.

### Beobachtbarkeit

Als nächstes wollen wir auf die Beobachtbarkeit von zeitdiskreten Systemen eingehen. Von großem Interesse ist an dieser Stelle die Frage, ob wir die Anfangsbedingung  $\mathbf{x}_0$  des zeitdiskreten Systems (9.4) anhand von  $n$  Messungen  $y_0, y_1, \dots, y_{n-1}$  bestimmen können. Es reicht aus, das autonome System ohne Eingang zu betrachten. Entwickeln wir den Ausgang in diskreter Zeit, so erhalten wir die folgenden Gleichungen

$$\begin{aligned} y_0 &= \mathbf{c}^T \cdot \mathbf{x}_0 \\ y_1 &= \mathbf{c}^T \cdot A \cdot \mathbf{x}_0 \\ y_2 &= \mathbf{c}^T \cdot A^2 \cdot \mathbf{x}_0 \\ y_3 &= \mathbf{c}^T \cdot A^3 \cdot \mathbf{x}_0 \\ &\vdots \\ y_{n-1} &= \mathbf{c}^T \cdot A^{n-1} \cdot \mathbf{x}_0 \end{aligned}$$

Die obigen Gleichungen lassen sich wie folgt zusammenfassen:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix} = Q_B \cdot \mathbf{x}_0 \quad (9.12)$$

Was wir in obiger Gleichung erkennen, halten wir im folgenden Satz fest.

**Satz 4** Das zeitdiskrete System (9.4) ist beobachtbar anhand des Ausgangs  $y$ , wenn die Matrix  $Q_B$ ,

$$Q_B = \begin{bmatrix} \mathbf{c}^T \\ \mathbf{c}^T \cdot A \\ \mathbf{c}^T \cdot A^2 \\ \vdots \\ \mathbf{c}^T \cdot A^{n-1} \end{bmatrix} \quad (9.13)$$

vollen Rang  $n$  hat. Dann ist der Anfangszustand aus den Messwerten rekonstruierbar:

$$\mathbf{x}_0 = Q_B^{-1} \cdot \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix} \quad (9.14)$$

Wie wir gesehen haben, sind die wichtigen Struktureigenschaften einer Regelstrecke, die Steuerbarkeit und die Beobachtbarkeit, im Falles eines zeitdiskreten Systems leicht abzuleiten. Im folgenden Abschnitt stellen wir den Zusammenhang zwischen der zeitkontinuierlichen Welt und der zeitdiskreten Welt her.

## 9.3. Abtastsysteme

In diesem Abschnitt wird auf Besonderheiten der rechnergestützten zeitdiskreten Regelung von zeitkontinuierlichen Systemen eingegangen.

### 9.3.1. Grundsätzliches

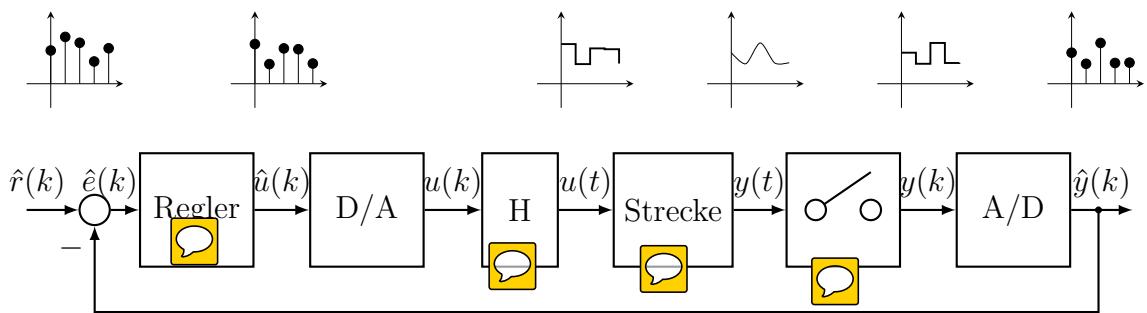
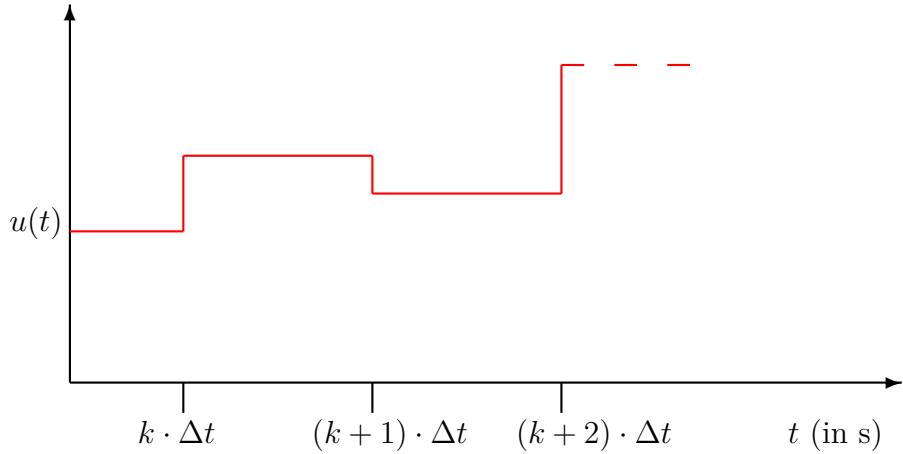


Abbildung 9.1.: Digitaler Regelkreis am Beispiel einer Ausgangsrückführung (nach: [17]).

In den Ingenieurwissenschaften liegen i.d.R. Systeme vor, die sich zeitkontinuierlich entwickeln. Da wir unsere Regelungsalgorithmen auf einer digitalen Hardware implementieren, ist es uns unmöglich, den kontinuierlichen Verlauf der Ausgänge zu erfassen. Aus Sicht der digitalen Signalprozessoren entwickelt sich das kontinuierliche System immer von einem Abtastzeitpunkt  $k \cdot \Delta t$  zum nächsten  $(k + 1) \cdot \Delta t$ . Die Wahrnehmung der realen Welt durch den digitalen Signalprozessor geschieht über den Analog-Digital-Wandler, der alle  $\Delta t$  Zeiteinheiten eine Wandlung durchführt und dann ausgelesen wird. Auch die Stellgröße kann sich nur im Takt der Abtastung ändern, da sie in diesem Takt vom digitalen Regler berechnet und dann über einen Digital-Analog-Wandler aufgeschalten wird. Zwischen den Abtastzeitpunkten wird die Stellgröße mit Hilfe eines Halteglieds konstant gehalten, weswegen sich der in Abb. 9.2 gezeigte treppenförmige Verlauf einstellt. Abb. 9.1 zeigt den Aufbau eines digitalen Regelkreises am Beispiel einer Ausgangsrückführung.

### 9.3.2. Zeitdiskrete Beschreibung abgetasteter kontinuierlicher Systeme

Um nun für ein zeitkontinuierliches System eine zeitdiskrete Regelung (oder Steuerung) zu entwerfen, kann es sinnvoll sein, eine zeitdiskrete Beschreibung des zeitkontinuierlichen Systems zu bestimmen. Damit können dann, wie im zeitkontinuierlichen Fall, Aussagen über wichtige Eigenschaften des ungeregelten und geregelten Systems (z.B. Stabilität,

Abbildung 9.2.: Zeitlicher Verlauf der Stellgröße  $u(t)$ .

Steuer- und Beobachtbarkeit, etc.) generiert werden. Wir betrachten daher nun ein zeitkontinuierliches, lineares und zeitinvariantes Zustandsraummodell (SISO).

$$\dot{\mathbf{x}} = \mathbf{A}_c \mathbf{x} + \mathbf{b}_c u \quad (9.15)$$

$$y = \mathbf{c}_c^T \mathbf{x} \quad (9.16)$$

Der Index  $c$  soll verdeutlichen, dass es sich um Systemmatrizen/-vektoren des kontinuierlichen Systems handelt. Aus der Systemtheorie (siehe z.B. Vorlesung „Systemtheorie in der Mechatronik“) ist bekannt, dass sich die Lösung der Zustandsdifferentialgleichung durch den folgenden analytischen Ausdruck angeben lässt:

$$\mathbf{x}(t) = e^{\mathbf{A}_c(t-t_0)} \mathbf{x}(t_0) + \int_{t_0}^t e^{\mathbf{A}_c(t-\tau)} \mathbf{b}_c u(\tau) d\tau \quad (9.17)$$

Mit Hilfe dieses Ausdrucks lässt sich nun eine zeitdiskrete Beschreibung des abgetasteten kontinuierlichen Systems ableiten. Die konstante Abtastzeit sei  $\Delta t$ . Wir wählen den Anfangszeitpunkt  $t_0$  gleich der Zeit zum Abtastschritt  $k$ , und die Zeit  $t$  gleich der Zeit zum nächsten Abtastschritt  $k + 1$ . Es gilt somit  $t_0 = k\Delta t$  und  $t = (k + 1)\Delta t$ . Außerdem nehmen wir an, dass sich die Einganggröße  $u(t)$  in dem Zeitintervall zwischen zwei Abtastungen nicht ändert und erhalten somit  $u(t) = u_k = \text{const}, \forall t \in [k\Delta t, (k + 1)\Delta t]$ . Damit lässt sich der Systemzustand  $\mathbf{x}_{k+1} = \mathbf{x}(t=(k+1)\Delta t)$  zum nächsten Abtastzeitpunkt  $k + 1$  berechnen:

$$\mathbf{x}_{k+1} = e^{\mathbf{A}_c \Delta t} \mathbf{x}_k + \int_{k\Delta t}^{(k+1)\Delta t} e^{\mathbf{A}_c((k+1)\Delta t-\tau)} \mathbf{b}_c u(\tau) d\tau \quad (9.18)$$

Mit der Substitution  $\alpha = (k + 1)\Delta t - \tau$  und der Annahme, dass die Stellgröße innerhalb eines Abtastschritts konstant gehalten wird, lässt sich das Integral weiter vereinfachen und es ergibt sich

$$\mathbf{x}_{k+1} = e^{\mathbf{A}_c \Delta t} \mathbf{x}_k + \int_0^{\Delta t} e^{\mathbf{A}_c \alpha} d\alpha \mathbf{b}_c u_k \quad (9.19)$$

Definiert man nun

$$\mathbf{A}_d := e^{\mathbf{A}_c \Delta t} \quad \text{und} \quad (9.20)$$

$$\mathbf{b}_d := \int_0^{\Delta t} e^{\mathbf{A}_c \alpha} d\alpha \mathbf{b}_c \quad (9.21)$$

erhält man schließlich die diskrete Zustandsdifferenzengleichung

$$\mathbf{x}_{k+1} = \mathbf{A}_d \mathbf{x}_k + \mathbf{b}_d u_k \quad (9.22)$$

die die Dynamik des kontinuierlichen Systems an den Abtastzeitpunkten **exakt** beschreibt. Die Matrix  $\mathbf{A}_d$  und der Vektor  $\mathbf{b}_d$  sind die Dynamikmatrix und der Eingangsvektor des abgetasteten Systems. Der Ausgangsvektor des abgetasteten Systems entspricht dem der kontinuierlichen Systembeschreibung,  $\mathbf{c}_d^T = \mathbf{c}_c^T$ . Damit lässt sich auch die Ausgangsgleichung des abgetasteten Systems angeben:

$$\mathbf{y}_k = \mathbf{c}_d^T \mathbf{x}_k \quad (9.23)$$

**Bemerkung 2** Ist die Dynamikmatrix des kontinuierlichen Systems regulär, d.h.  $\det(\mathbf{A}_c) \neq 0$ , kann die Lösung des Integrals zur Bestimmung von  $\mathbf{b}_d$  analytisch angegeben werden.

$$\mathbf{b}_d = \mathbf{A}_c^{-1} (e^{\mathbf{A}_c \Delta t} - I) \mathbf{b} = \mathbf{A}_c^{-1} (\mathbf{A}_d - I) \mathbf{b}$$

**Bemerkung 3** Die Exponentielle  $e^{\mathbf{A}_c \Delta t}$  ist definiert als

$$e^{\mathbf{A}_c \Delta t} = \sum_{i=0}^{\infty} \frac{(\Delta t)^i}{i!} \mathbf{A}_c^i$$

Für sehr kleine Abtastzeiten kann man die Reihe nach dem linearen Glied abbrechen und folgenden Approximation für  $e^{\mathbf{A}_c \Delta t}$  benutzen:

$$e^{\mathbf{A}_c \Delta t} \approx \mathbf{I} + \mathbf{A}_c \Delta t$$

Anhand dieser zeitdiskreten Darstellung und den Erkenntnissen aus Abschnitt 9.2.3 kann das abgetastete System nun auf Stabilität, Steuer- und Beobachtbarkeit untersucht werden.

In diesem Zusammenhang ist es interessant, in welcher Weise sich diese Eigenschaften vom zeitkontinuierlichen auf das durch Abtastung entstehende zeitdiskrete System übertragen. Die folgenden Aussagen treffen zu:<sup>8</sup>

- Eine stabile Eigendynamik des zeitkontinuierlichen Systems ist eine notwendige und hinreichende Bedingung für die Stabilität des zeitdiskreten. Das heißt, ist eines der beiden stabil so ist es auch das andere. Dies wird klar, wenn man den Zusammenhang zwischen den Eigenwerten der beiden Systeme betrachtet:  $\lambda_i(\mathbf{A}_d) = e^{\lambda_i(\mathbf{A}_c) \Delta t}$ .

<sup>8</sup>Der Nachweis dieser Aussagen wird der Kürze halber untergeschlagen, kann aber z.B. in [17] nachgelesen werden.

- Die Übertragungsstabilität des zeitkontinuierlichen Systems ist eine hinreichende, aber nicht notwendige Bedingung für die Übertragungsstabilität des zeitdiskreten. D. h. das zeitdiskrete System ist übertragungsstabil, wenn das kontinuierliche System übertragungsstabil ist. Die Umkehrung gilt nur dann, wenn die steuerbaren und beobachtbaren Eigenvorgänge des kontinuierlichen Systems auch beim zeitdiskreten System steuerbar und beobachtbar sind. Andernfalls ist es denkbar, dass instabile Eigenvorgänge des kontinuierlichen Systems das Ein-/Ausgangsverhalten des zeitdiskreten Systems nicht beeinflussen und das abgetastete instabile System deshalb E/A-stabil ist.
- Die Steuerbarkeit und Beobachtbarkeit des kontinuierlichen Systems sind notwendig, aber nicht hinreichend für die Steuerbarkeit und Beobachtbarkeit des zeitdiskreten Systems.

## 9.4. Digitaler Regelkreis

Die Regelkreisstrukturen der digitalen Regelung unterscheiden sich nicht von denen der kontinuierlichen Regelung. Es können auch hier beispielsweise Zustands- und Ausgangsrückführungen, PI- und PID-Regler realisiert werden. Da die mathematische Beschreibung der zeitdiskreten Regelung der kontinuierlichen sehr ähnlich ist, können die wichtigsten Aussagen über die Regelkreiseigenschaften und Analysemethoden direkt übernommen werden. So kann zum Beispiel die Stabilität des geregelten Systems, genau wie im zeitkontinuierlichen Fall, anhand dessen Eigenwerten/Polen beurteilt werden.

### 9.4.1. Reglerentwurf

Für den Entwurf von Abtastreglern gibt es zwei Wege, die in diesem Kapitel behandelt werden. Ist die Abtastzeit sehr klein im Vergleich zu den maßgebenden Zeitkonstanten des Regelkreises, so kann der Regler als kontinuierlicher Regler entworfen und dann als zeitdiskreter Regler realisiert werden. In [17] wird hierzu als Fausregel angegeben, dass die Abtastfrequenz ca. 20-30 mal größer sein sollte als die Grenzfrequenz<sup>9</sup> des Regelkreises. Allerdings besteht bei dieser Variante immer noch das Problem, dass das zeitkontinuierliche Regelgesetz zeitdiskret approximiert werden muss. Andererseits gibt es Verfahren zum Entwurf zeitdiskreter Regler für eine zeitdiskret modellierte Regelstrecke. Der Entwurf verläuft nach denselben Methoden wie bei kontinuierlichen Reglern. Beide Verfahren werden im Folgenden näher betrachtet.

#### Approximation eines zeitkontinuierlichen Regelgesetzes durch einen zeitdiskreten Regler

In diesem Abschnitt gehen wir davon aus, dass die Abtastung schnell genug erfolgt. In diesem Fall kann man einen kontinuierlichen Regler anhand eines kontinuierlichen Modells auslegen. Für die digitale Umsetzung muss das zeitkontinuierliche Regelgesetz nun jedoch durch ein zeitdiskretes approximiert werden. Besteht der Regler nur aus einer konstanten

<sup>9</sup>Die Grenzfrequenz beschreibt, bis zu welcher Frequenz ein Regelkreis dem Sollwert folgt bzw. bis zu welcher Frequenz er Störungen unterdrückt.

Rückführung (Zustandsrückführung oder Ausgangsrückführung mit P-Regler), so kann er im Zeitdiskreten eins zu eins implementiert werden. Anders sieht es aus, falls der Regler eine eigene Dynamik besitzt, wie zum Beispiel ein PID-Regler der Form:

$$u(t) = k_P e(t) + k_I \int_0^t e(\tau) d\tau + k_D \frac{de(t)}{dt}, \quad \text{mit } e(t) = r(t) - y(t) \quad (9.24)$$

Das Integral und die Ableitung der Regelabweichung stehen nicht wie bei einem analogen Regelkreis kontinuierlich zur Verfügung. Da die Regelabweichung dem digitalen Regler nur an den Abtastzeitpunkten bekannt ist, muss sowohl das Integral als auch die Ableitung numerisch berechnet werden. Damit kann die zeitdiskrete Umsetzung lediglich eine Approximation des zeitkontinuierlichen Regelgesetzes darstellen.

### Approximation der Ableitung:

Die Ableitung der Regelabweichung

$$\dot{e}(t) = \frac{de(t)}{dt} \quad (9.25)$$

kann z.B. über eine Finite Differenz der Form

$$\rightarrow \dot{e}(k\Delta t) = \dot{e}_k \approx \frac{e_k - e_{k-1}}{\Delta t} \quad (9.26)$$

näherungsweise berechnet werden. Im Frequenzbereich bedeutet dies, dass die Übertragungsfunktion eines D-Glieds

$$R(s) = k_D s \quad (9.27)$$

durch die Z-Übertragungsfunktion

$$R(z) = k_D \frac{1 - z^{-1}}{\Delta t} = \frac{k_D}{\Delta t} \frac{z - 1}{z} \quad (9.28)$$

ersetzt wird. Vergleicht man Gleichung (9.27) und (9.28) so fällt auf, dass zur Bestimmung des diskreten Reglers die komplexe Frequenz  $s$  durch

$$s = \frac{1}{\Delta t} \frac{z - 1}{z} \quad (9.29)$$

ersetzt wurde. An dieser Stelle sind zwei Dinge bemerkenswert:

1. Im Gegensatz zum kontinuierlichen D-Regler, der eine (evtl. stabilisierende) Phasenvordrehung erzeugt, erzeugt der diskrete D-Regler ein Phasennacheilen. Der Grund hierfür ist, dass der Regler jeweil einen „Takt“ warten muss bevor er die Differenzbildung ausführen kann.
2. Der diskrete D-Regler ist, im Gegensatz zum kontinuierlichen, technisch von vornherein realisierbar, da das Regelgesetz nur tatsächlich verfügbare Informationen benötigt.

### Approximation des Integrals:

Um ein I-Glied  $R(s) = k_I/s$  zeitdiskret zu realisieren, muss das Integral der Regelabweichung numerisch berechnet werden. Benutzt man hierzu die **Rechteckregel**

$$\int_0^{k\Delta t} e(\tau) d\tau \approx \Delta t \sum_{i=1}^k e_i \quad (9.30)$$

so kann das zeitdiskrete Regelgesetz rekursiv in der Form

$$u_k = u_{k-1} + k_I \Delta t e_k \quad (9.31)$$

angegeben werden. Im Frequenzbereich ergibt sich daraus, dass die Übertragungsfunktion

$$R(s) = \frac{k_I}{s} \quad (9.32)$$

durch die Z-Übertragungsfunktion

$$R(z) = k_I \Delta t \frac{z}{z-1} \quad (9.33)$$

und die komplexe Frequenz  $s$  durch

$$s = \frac{1}{\Delta t} \frac{z-1}{z} \quad (9.34)$$

ersetzt wird. Es besteht hier also derselbe Zusammenhang zwischen  $s$  und  $z$  wie bei Approximation der Ableitung durch eine Finite Differenz (vgl. Gl. (9.29)).

Benutzt man zur numerischen Integration alternativ die **Trapezregel**

$$\int_0^{k\Delta t} e(\tau) d\tau \approx \Delta t \left( \frac{e_0}{2} + \sum_{i=1}^{k-1} e_i + \frac{e_k}{2} \right) \quad (9.35)$$

so ergibt sich folgende rekursive Form des Stellgesetzes

$$u_k = u_{k-1} + \frac{k_I \Delta t}{2} (e_{k-1} + e_k) \quad (9.36)$$

und damit im Frequenzbereich die Z-Übertragungsfunktion

$$R(z) = \frac{k_I \Delta t}{2} \frac{z+1}{z-1} \quad (9.37)$$

Man erkennt also, dass hier die komplexe Frequenz  $s$  durch

$$s = \frac{2}{\Delta t} \frac{z-1}{z+1} \quad (9.38)$$

ersetzt wird. Der letzte Zusammenhang ist in der Literatur auch als Tustin-Transformation bekannt. Wie sich durch nachrechnen leicht zeigen lässt, besitzt diese Transformation die folgende schöne Eigenschaft: Eine komplexe Frequenz  $s$  aus der linken komplexen Halbebene wird in den Einheitskreis abgebildet und umgekehrt. Das heißt, aus stabilen Eigenwerten des kontinuierlichen Reglers werden stabile Eigenwerte des zeitdiskreten Reglers.

Die angegebene Approximation ist also auch aus der Sicht der Stabilitätseigenschaften des Reglers zweckmäßig.<sup>10</sup> Die Tustin-Transformation bietet sich daher zur Approximation beliebiger dynamischer Regelgesetze an.

In der folgenden Übersicht sind die Ergebnisse dieses Abschnitts noch einmal kompakt zusammengefasst:

<b>kont. Regelgesetz:</b> $u(t) = k_P e(t) + k_I \int_0^t e(\tau) d\tau + k_D \frac{de(t)}{dt}$ $\downarrow$ Approximation <b>disk. Regelgesetz:</b> $u_k = k_P x_{P,k} + k_I x_{I,k} + k_D x_{D,k}$	$\swarrow$ $\searrow$
<b>Rechteckregel bzw. Finite Differenz</b> $x_{P,k} = e_k$ $x_{I,k} = \Delta t e_k + x_{I,k-1}$ $x_{D,k} = \frac{1}{\Delta t} (e_k - e_{k-1})$	<b>Trapezregel bzw. Tustin-Transformation</b> $x_{P,k} = e_k$ $\mathbf{x}_{I,k} = \frac{\Delta t}{2} (\mathbf{e}_k + \mathbf{e}_{k-1}) + \mathbf{x}_{I,k-1}$ $x_{D,k} = \frac{2}{\Delta t} (e_k - e_{k-1}) - x_{D,k-1}$

In dieser Form kann der PID-Regler zeitdiskret implementiert werden, wobei die Hilfszustände  $x_{I,k}$  und  $x_{D,k}$  zeitlich mitentwickelt werden müssen. Der durch die Tustin-Transformation approximierte D-Anteil ( $x_{D,k}$ ) weist eine instabile Dynamik auf. Daher wird der D-Anteil i.d.R. über die Finite-Differenz und der I-Anteil über die Tustin-Transformation approximiert.

Auf eine wichtige Eigenschaft, die alle diskreten Umsetzungen gemein haben, soll nun noch hingewiesen werden: Bei allen zeitdiskret umgesetzten Regelgesetzen taucht die Abtastzeit  $\Delta t$  als Parameter im Regelgesetz auf. **Wird also die Abtastzeit verändert, so muss auch das Regelgesetz neu berechnet werden!**

### Direkter Entwurf eines zeitdiskreten Reglers

Liegt eine zeitdiskrete Beschreibung des Abtastsystems vor, so kann auch direkt ein diskreter Regler entworfen werden. Wir werden, um es kurz zu halten, an dieser Stelle nur auf den Entwurf einer zeitdiskreten Zustandsrückführung eingehen.

#### Zustandsrückführung:

Dazu gehen wir davon aus, dass ein zeitdiskretes Zustandsraummodell der Strecke gemäß (9.4) vorliegt. Wir setzen ein Regelgesetz der Form

$$u_k = -\mathbf{r}^T \mathbf{x}_k + f r_k \quad (9.39)$$

an. Die Größen  $\mathbf{r}^T$ ,  $f$  und  $r_k$  sind die Rückführmatrix, ein Führungsfilter zum Erreichen stationärer Genauigkeit und die Sollgröße zum aktuellen Zeitpunkt. Daraus ergibt sich folgende Darstellung für den geschlossenen Regelkreis:

$$\begin{aligned} \mathbf{x}_{k+1} &= \mathbf{A}_r \cdot \mathbf{x}_k + \mathbf{b} f r_k \\ y_k &= \mathbf{c}^T \mathbf{x}_k \end{aligned} \quad (9.40)$$

mit:  $\mathbf{A}_r = (\mathbf{A} - \mathbf{b} \mathbf{r}^T) \in \mathbb{R}^{n \times n}$

<sup>10</sup>Damit ist jedoch nicht gesichert, dass auch der Regelkreis seine Stabilitätseigenschaften beibehält!

Nun wollen wir  $f$  so wählen, dass für  $k \rightarrow \infty$  die Regelgröße einer konstanten Führunggröße stationär genau folgt. Wir betrachten dazu die Z-Übertragungsfunktion

$$G(z) = \frac{y(z)}{r(z)} = \mathbf{c}^T(z\mathbf{I} - \mathbf{A} + \mathbf{b}\mathbf{r}^T)^{-1}\mathbf{b} f \quad (9.41)$$

und fordern, dass diese für  $z = 1$  (s. Endwertsatz der Z-Transformation) den Wert 1 annimmt  $\rightarrow G(1) \stackrel{!}{=} 1$ . Daraus ergibt sich der Vorfilter  $f$ :

$$f = \frac{1}{\mathbf{c}^T(\mathbf{I} - \mathbf{A} + \mathbf{b}\mathbf{r}^T)^{-1}\mathbf{b}} \quad (9.42)$$

Ist das System nun noch vollständig steuerbar, so können wir mit der Zustandsrückführung alle Eigenwerte der Dynamikmatrix  $A_r$  beliebig vorgeben. Dazu können die aus dem Zeitkontinuierlichen bekannten Methoden der Polplatzierung benutzt werden.

### Zeitdiskrete optimale Regelung

Im Zeitdiskreten lässt sich außerdem auch ein LQ-Regler umsetzen, der das folgende Gütemaß minimiert:

$$J(\mathbf{x}_0, u_k) = \sum_{k=0}^{\infty} (\mathbf{x}_k^T \mathbf{Q} \mathbf{x}_k + R u_k^2) \quad (9.43)$$

Wie im Zeitkontinuierlichen erhält man die Lösung des Optimierungsproblem über eine Riccatigleichung, die im Zeitdiskreten wie folgt lautet:<sup>11</sup>

$$\mathbf{P} = \mathbf{Q} + \mathbf{A}^T \mathbf{P} \mathbf{A} - \mathbf{A}^T \mathbf{P} \mathbf{b} (R + \mathbf{b}^T \mathbf{P} \mathbf{b})^{-1} \mathbf{b}^T \mathbf{P} \mathbf{A} \quad (9.44)$$

Mit der positiv definiten Lösung  $\mathbf{P}$  ergibt sich die optimale Rückführung:

$$\mathbf{r}^T = (R + \mathbf{b}^T \mathbf{P} \mathbf{b})^{-1} \mathbf{b}^T \mathbf{P} \mathbf{A} \quad (9.45)$$

### 9.4.2. Zeitdiskrete Zustandsbeobachtung

Nun wollen wir noch einen zeitdiskreten Zustandsbeobachter entwerfen, um die Zustandsrückführung realisieren zu können. Wir gehen davon aus, dass nicht der gesamte Zustandsvektor gemessen wird, sondern nur die Ausgangsgröße  $y$ . Wie gelangen wir auf alternativem Wege zu den notwendigen Zustandsgrößen? Da wir bereits ein Modell nutzen, um eine Rückführung zu entwerfen, liegt es nahe das Modell auch zu nutzen, um die Zustände dem digitalen Signalsprozessor bereitzustellen. Zuerst betrachten wir einen *trivialen* Beobachter, oder auch Simulator, der den Zustand  $\bar{x}_k$  erzeugt, indem die iterative Abbildung

$$\bar{x}_{k+1} = \mathbf{A} \cdot \bar{x}_k + \mathbf{b} \cdot u_k, \quad \mathbf{A} \in \mathbb{R}^{n \times n}, \quad \mathbf{b} \in \mathbb{R}^n \quad \text{und} \quad \bar{x}_k := \bar{x}(k \cdot \Delta t) \in \mathbb{R}^n \quad (9.46)$$

in jedem Abtastschritt ausgewertet wird. Dieses dynamische System gleicht dem diskreten System (9.4), wenn Dynamikmatrix und Eingangsvektoren der beiden Systeme übereinstimmen und die Stellgröße  $u_k$  bekannt ist.

---

<sup>11</sup>In Matlab steht der Befehl `dare` zum Lösen einer zeitdiskreten Riccatigleichung zur Verfügung.

Allerdings wird ein Anfangszustand  $\bar{x}_0$  für die erste Auswertung von Gleichung (9.46) benötigt. Dieser ist jedoch im Allgemeinen nicht, oder nur teilweise, bekannt. Allein mit Gleichung (9.46) lässt sich der Systemzustand also selbst bei bekannter Systemdynamik nicht exakt rekonstruieren. Deshalb wird (9.46) um eine Ausgangsrückführung erweitert:

$$\bar{x}_{k+1} = \mathbf{A} \cdot \bar{x}_k + \mathbf{b} \cdot u_k + \mathbf{k} \cdot (y_k - \mathbf{c}^T \cdot \bar{x}_k), \quad \mathbf{k} \in \mathbb{R}^n \quad (9.47)$$

Wir betrachten nun den Schätzfehler  $\mathbf{e}$ , definiert als Differenz der Zustände des Beobachters (9.47) und des realen Prozesses (9.4):

$$\mathbf{e}_k = \mathbf{x}_k - \bar{x}_k, \quad \mathbf{e}_k \in \mathbb{R}^n \quad (9.48)$$

Seine Fehlerdynamik ergibt sich zu

$$\begin{aligned} \mathbf{e}_{k+1} &= \mathbf{x}_{k+1} - \bar{x}_{k+1} = \\ \mathbf{A} \cdot \mathbf{x}_k + \mathbf{b} \cdot u_k - \mathbf{A} \cdot \bar{x}_k - \mathbf{b} \cdot u_k - \mathbf{k} \cdot (y_k - \mathbf{c}^T \cdot \bar{x}_k) &= \\ \mathbf{A} \cdot \mathbf{e}_k - \mathbf{k} \cdot (\mathbf{c}^T \cdot \mathbf{x}_k - \mathbf{c}^T \cdot \bar{x}_k) &= (\mathbf{A} - \mathbf{k} \cdot \mathbf{c}^T) \mathbf{e}_k \end{aligned} \quad (9.49)$$

mit dem Anfangsfehler  $\mathbf{e}_{k=0} = \mathbf{e}_0 \in \mathbb{R}^n$ . Ist das System beobachtbar (Satz 4 gilt), so können wir durch Wahl der Einträge im Vektor  $\mathbf{k}$  die Eigenwerte der Fehlerdynamik (9.49), das sind die Eigenwerte von  $\mathbf{A} - \mathbf{k} \cdot \mathbf{c}^T$ , frei vorgeben [17]. Wir wollen, dass die Fehlerdynamik stabil ist, d.h. der Fehler soll abklingen. Nach Satz 1 müssen die Eigenwerte von  $\mathbf{A} - \mathbf{k} \cdot \mathbf{c}^T$  deshalb innerhalb des Einheitskreises plaziert werden. Der Beobachter ist dann durch Gleichung (9.47) gegeben.

Bei einer realen Umsetzung dieses Beobachters liegt im Abtastschritt  $k$  der Zustand  $\bar{x}_k$  im Speicher. Der A/D-Wandler liest in diesem Abtastschritt die Sensoren aus. Anschließend wird nach (9.47) der Zustand für den Abtastschritt  $k + 1$  berechnet,  $\bar{x}_{k+1}$ , und schließlich die Stellgröße  $u_{k+1}$  nach Gleichung (9.39) mit dem Zustand  $\bar{x}_{k+1}$ . Da der Anfangsfehler konvergiert, wird der Regler asymptotisch mit dem tatsächlichen Zustand arbeiten. Die Konvergenzgeschwindigkeit kann durch geschickte Wahl von  $\mathbf{k}$  in (9.49) eingestellt werden!

Eine wichtige Anmerkung zur Wahl der Ausgangsrückführung  $\mathbf{k}$  in (9.47): Da die Matrix  $\mathbf{A} - \mathbf{k} \cdot \mathbf{c}^T$  die gleichen Eigenwerte besitzt wie die Matrix  $[\mathbf{A} - \mathbf{k} \cdot \mathbf{c}^T]^T = \mathbf{A}^T - \mathbf{c} \cdot \mathbf{k}^T$ , kann die Aufgabe, die Ausgangsrückführung  $\mathbf{k}$  zu parametrisieren in eine Entwurfsproblem für eine Zustandsrückführung überführt werden: Dazu wählt man  $\mathbf{k}$  so, dass das duale System

$$\tilde{x}_{k+1} = \mathbf{A}^T \cdot \tilde{x}_k + \mathbf{c} \cdot \tilde{u}_k, \quad \mathbf{A}^T \in \mathbb{R}^{n \times n}, \quad \mathbf{c} \in \mathbb{R}^n \quad \text{und} \quad \tilde{x}_k := \tilde{x}(k \cdot \Delta t) \in \mathbb{R}^n \quad (9.50)$$

durch die Zustandsrückführung  $\tilde{u}_k = -\mathbf{k}^T \cdot \tilde{x}_k$ ,  $\tilde{u}_k \in \mathbb{R}$  stabilisiert wird [17].

### 9.4.3. Das Separationsprinzip

Wie beantworten wir nun die Frage, ob es zulässig ist, in der Zustandsrückführung an Stelle des realen Zustandsvektors seine Schätzung zu verwenden. Dabei analysieren wir

den geschlossenen Regelkreis in diskreten Zeit. Gegeben ist nun

$$\text{die Strecke: } \mathbf{x}_{k+1} = \mathbf{A} \cdot \mathbf{x}_k + \mathbf{b} \cdot u_k, \quad \mathbf{x}_{k=0} = \mathbf{x}_0 \quad (9.51a)$$

$$\text{das Regelgesetz: } u_k = -\mathbf{r}^T \cdot \bar{\mathbf{x}}_k + f \cdot r_k \quad (9.51b)$$

$$\text{der Beobachterzustand: } \bar{\mathbf{x}}_k = \mathbf{x}_k - \mathbf{e}_k \quad (9.51c)$$

$$\text{und die Fehlerdynamik: } \mathbf{e}_{k+1} = [\mathbf{A} - \mathbf{k} \cdot \mathbf{c}^T] \cdot \mathbf{e}_k, \quad \mathbf{e}_{k=0} = \mathbf{e}_0 \quad (9.51d)$$

mit der Reglerverstärkung  $\mathbf{r} \in \mathbb{R}^n$  und der Beobachterverstärkung  $\mathbf{k} \in \mathbb{R}^n$ . Setzen wir das Regelgesetz in die Strecke ein, so erhalten wir die Dynamik des geschlossenen Kreises:

$$\begin{bmatrix} \mathbf{x}_{k+1} \\ \mathbf{e}_{k+1} \end{bmatrix} = \begin{bmatrix} \mathbf{A} - \mathbf{b} \cdot \mathbf{r}^T & \mathbf{b} \cdot \mathbf{r}^T \\ 0 & \mathbf{A} - \mathbf{k} \cdot \mathbf{c}^T \end{bmatrix} \cdot \begin{bmatrix} \mathbf{x}_k \\ \mathbf{e}_k \end{bmatrix} + \begin{bmatrix} \mathbf{b} \cdot f \\ \mathbf{0} \end{bmatrix} \cdot r_k \quad (9.52)$$

Die Eigenwerte der Dynamikmatrix dieses dynamischen Systems müssen natürlich im Einheitskreis zu liegen kommen, damit der geschlossene Kreis stabil ist. Wegen der Diagonalstruktur des geschlossenen Kreises gilt das folgende Separationstheorem

**Satz 5** *Die Eigenwerte des Regelkreises (9.52), in dem eine Zustandsrückführung mit einem Beobachter realisiert ist, setzen sich aus den Eigenwerten der Matrix  $\mathbf{A} - \mathbf{b} \cdot \mathbf{r}^T$ , die einen Regelkreis mit Zustandsrückführung und ohne Beobachter beschreibt, und den Eigenwerten der Systemmatrix  $\mathbf{A} - \mathbf{k} \cdot \mathbf{c}^T$  des Beobachters zusammen.*

Wegen dem obigen Satz, dürfen wir die Zustandsrückführung getrennt von der Zustandsbeobachtung entwerfen. Sind beide Systeme stabil, so ist auch der geschlossene Kreis stabil. Aus praktischen Gründen empfiehlt es sich jedoch, die Fehlerdynamik eine bis mehrere Größenordnungen schneller abklingen zu lassen als den Regelfehler.

## 9.5. Echtzeitfähige Implementierung des Regelgesetzes

Abschließend gehen wir nun noch auf die echzeitfähigkeit unseres Regelgesetzes ein. Egal ob wir uns für einen PID-Regler oder eine Zustandsrückführung entscheiden, müssen wir sicher stellen, dass alle vom Mikrocontroller **auszuführenden Operationen innerhalb eines Abtastschritts zu bewältigen** sind. Zur Veranschaulichung dient der Zeitstrahl in Abb. 9.3.

Zu Beginn und vor dem ersten Einlesen der Sensoren (für  $k = 0$ ) müssen **alle Größen, die in das Regelgesetz eingehen, geeignet initialisiert** werden. Für einen PID-Regler sind dies die Hilfszustände des Regelgesetzes (s. Übersicht in Abschnitt 9.4.1) und für eine Zustandsrückführung mit Zustandsbeobachter ist dies  $\bar{\mathbf{x}}_0$ . Damit ist die Stellgröße  $u_0$  im Abtastschritt  $k = 0$  definiert. Wir gehen nun also davon aus, dass der Signalprozessor initialisiert und  $u_0$  aufgeschaltet wurde. **Als erstes sind die Sensoren auszulesen und im Falle eines analogen Sensors muss eine Analog-Digital-Wandlung** durchgeführt werden (vergleiche Abbildung 9.3). Nun steht  $y_0$  zur Verfügung. Im Falle eines PID-Reglers berechnen wir nun die Regelabweichung  $e_0$  und werten damit das Regelgesetz aus.<sup>12</sup> Im

<sup>12</sup>Eigentlich erhalten wir damit die Stellgröße  $u_0$ , die wir aber erst zum Zeitpunkt  $u_1$  aufschalten. Die Stellgröße wird also um eine Abtastzeit  $\Delta t$  verzögert aufgeschalten. Ein weiterer Grund warum  $\Delta t$  klein sein sollte.

Falle einer Zustandsrückführung wird die Beobachtergleichung (9.47) ausgewertet und damit der geschätzte Zustandsvektor  $\bar{x}_1$  berechnet. Anschließend kann die neue Stellgröße  $u_1 = -\mathbf{r}^T \cdot \bar{x}_1 + f \cdot r_1$  berechnet werden, die im nächsten Abtastschritt  $k = 1$  aufgeschaltet wird. Danach haben wir wieder  $\Delta t$  Zeiteinheiten um (i) die Sensoren auszulesen und (ii) die Stellgröße  $u_2$  zu bestimmen und abzuspeichern.

Die Balken in Abbildung 9.3 zeigen qualitativ die belegte Prozessorzeit der jeweiligen Aufgabe an.  $u_k$  ist i.d.R. ein PWM-Signal, das Schreiben der Daten in das PWM-Register benötigt fast keine Rechenzeit, weswegen der Balken sehr dünn ist. Die A/D-Wandlung benötigt eine gewisse Zeit, die dem Datenblatt des Signalprozessors entnommen werden kann. Selbstverständlich benötigt der Prozessor auch Zeit um die Stellgröße für den nächsten Zeitschritt zu berechnen. Bei einer Zustandsrückführung muss hier beispielsweise der Beobachter aktualisiert (Gl. (9.47)  $\rightarrow \bar{x}_{k+1}$ ) und anschließend das Regelgesetz (9.39,  $\rightarrow u_{k+1}$ ) ausgewertet werden. Alle Prozesse müssen innerhalb von  $\Delta t$  abgeschlossen sein.

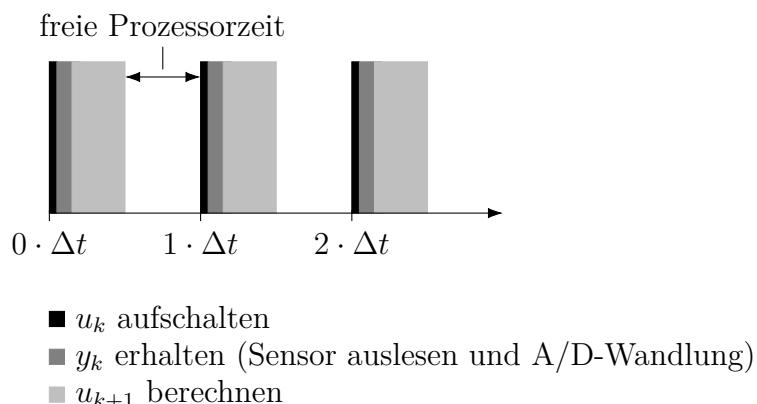


Abbildung 9.3.: Ablauf des Regelzyklus

# 10. Drehgeber / Encoder

Bei elektromechanischen Systemen ist es häufig nötig, die Position oder Lage eines Elementes zu messen. Hier im Praktikum ist dies beispielsweise der **Drehwinkel** der Räder des Kleinroboters. Hierfür werden im allgemeinen so genannte Drehgeber (engl. rotary encoder) verwendet. Man unterscheidet **zwei Typen** von Drehgebern:

- **Absolut-Drehgeber** messen direkt die absolute Position (z.B. Winkel)
- **Inkrementalgeber** erfassen lediglich **Lageänderungen**. Diese Daten können anschließend zur absoluten Position weiterverarbeitet werden.

## 10.1. Funktionsweise

Absolut-Drehgeber kodieren direkt die **verschiedenen diskreten Positionen**. Dies kann beispielsweise mithilfe von **Schleifkontakte** oder auch **optisch** erfolgen. Mithilfe der Kodierung in Abbildung 10.1(a) lassen sich beispielsweise mit drei Kontakten  $2^3 = 8$  verschiedene Winkelbereiche auf dem Kreis kodieren.

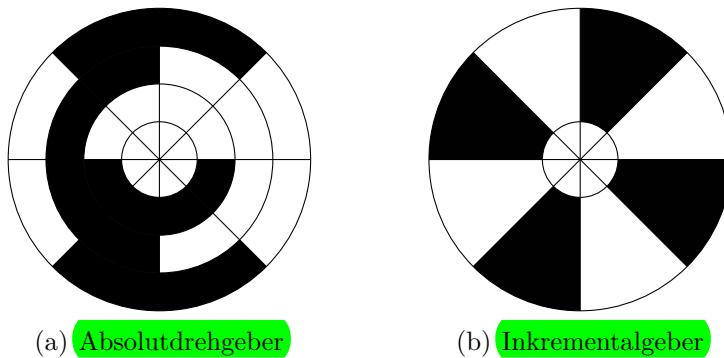


Abbildung 10.1.: Schematische Darstellung der Kodierung bei Drehgebern.

Da die Räder des Roboters jedoch mehr als eine Umdrehung absolvieren sollen, bieten sich für die Messung der **Rad-Drehwinkel** Inkrementalgeber an. Diese verfügen über sich periodisch wiederholende **Teilstriche** (siehe Abb. 10.1(b)), die über verschiedene Mechanismen **erfasst** werden können:

- Schleifkontakte
- Photoelektrische Abtastung (abbildend oder interferentiell)
- Magnetische Abtastung

Aufgrund der hohen Genauigkeit bei vergleichsweise einfacherem Aufbau werden häufig photoelektrische Drehgeber verwendet. Bei diesen wird eine mit Schlitten versehene Scheibe von einer **Lichtquelle (LED)** beleuchtet. Auf der anderen Seite der Scheibe befinden sich **zwei leicht versetzte Photodioden**, die je nach Position der Scheibe abwechselnd angeleuchtet werden (siehe Abb. 10.2).

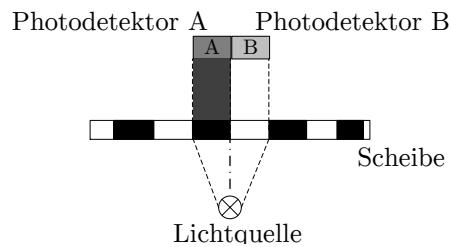


Abbildung 10.2.: Schematische Darstellung eines photoelektrischen Drehgebers.

Bei einer Drehung sind die beiden Signale von Photodetektor A und B entsprechend um  $\pm 90^\circ$  verschoben (Abb. 10.3). Entsprechend der jeweiligen Phasenverschiebung wird der Strichzähler erhöht oder erniedrigt. Mit der Anzahl der Striche kann nun der gesamte Winkel berechnet werden:

$$\gamma = 2\pi \frac{n}{N} \quad (10.1)$$

wobei  $n$  die aktuelle Anzahl der gezählten Striche und  $N$  die Anzahl der Striche auf der Scheibe ist.

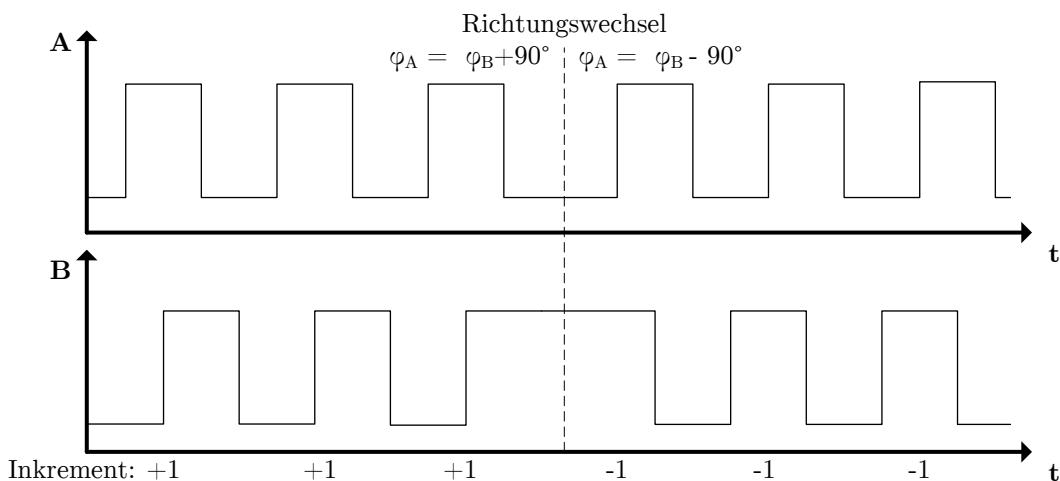


Abbildung 10.3.: Signalverlauf A und B Kanal eines Drehgebers.

Um den Nullpunkt der Winkelmessung festzulegen, verfügen manche Drehgeber über eine weitere Spur (*index signal*), die genau dort einen einzelnen Puls erzeugt.

## 10.2. Auswertungsmöglichkeiten

Für die Aufnahme und Auswertung der Encoder-Signale mit dem Mikrocontroller gibt es verschiedene Möglichkeiten, die im Folgenden kurz vorgestellt werden.

### 10.2.1. Timerbasiert (polling)

Um den Drehwinkel zu bestimmen, müssen wir die beiden Signale A und B des Encoders erfassen. Die einfachste Herangehensweise ist es, die mit den Kanälen verbundenen Pins des Mikrocontrollers als Eingang zu schalten und in regelmäßigen Abständen abzufragen (Polling). Wenn das Ausgangssignal des Encoders auf Logic-Level<sup>1</sup> liegt, kann es genügen einfache Input-Pins zu verwenden. Ansonsten können die Signale auch erst mit dem A/D-Wandler des Chips erfasst und anschließend mit einem Grenzwert verglichen werden. Eine A/D-Wandlung ist jedoch vergleichsweise zeitaufwändig.

Eine entscheidende Einschränkung der Polling-Methode ist die korrekte Wahl der Abtastfrequenz. Diese muss zwei gegeneinander wirkende Anforderungen erfüllen:

- Die Abtastfrequenz muss hoch genug sein um alle Flankenwechsel im Signalverlauf zu erfassen.
- Sie soll aber auch möglichst gering sein, da die CPU während der Abtastung nicht anderweitig genutzt werden kann.

Um die notwendige Abtastfrequenz zu bestimmen, muss die maximale Drehgeschwindigkeit der Scheibe vorher bekannt sein.

### 10.2.2. Externe Interrupts

Je nach Wahl der Abtast-Frequenz werden bei der Polling-Methode viele unnötige Abtastungen durchgeführt oder aber Signale nicht erfasst. Beides kann man durch die Verwendung von externen Interrupts vermeiden. Wie bereits in Abschnitt 4.3 erklärt, können bestimmte Eingangspins eines Mikrocontrollers für so konfiguriert werden, dass sie bei einer Zustandsänderung einen Interrupt auslösen. Werden solche Pins mit den Photodetektoren verbunden, werden die Signale tatsächlich nur zu den Zeitpunkten ausgewertet, an denen Sie uns interessieren, nämlich an den Flankenwechseln. Für den Fall, dass das Encoder-Signal nicht auf dem Logic-Level des Mikrocontrollers liegt, kann auch der Analog-Comparator als Interrupt-Quelle verwendet werden (siehe [5, S. 186ff]).

Eine große Drehgeschwindigkeit bedeutet viele Flankenwechsel und entsprechend viele Interrupts die ausgelöst werden. Dies kann den Programmablauf negativ beeinflussen.

Sowohl bei der Polling-Methode als auch bei der Nutzung externer Interrupts muss die Interpretation und Auswertung der Signale in Software erfolgen. Hierfür muss Rechenzeit eingeplant werden.

### 10.2.3. Quadraturdecoder

Neben den oben gezeigten Möglichkeiten den Encoder selber auszuwerten, gibt es hierfür auch fertige Bausteine.

<sup>1</sup>Logic-Level: Spannungsbereich in dem ein Mikrocontroller die Zustände High und Low darstellt, z.B.  $V_{lo} \leq 0,8\text{ V}$  und  $V_{hi} \geq 2,0\text{ V}$

### Intern

Einige Mikrocontroller mit vielen Peripherie-Module (z.B. Xmega-Familie) besitzen einen eingebauten Quadraturdecoder. Dieser übernimmt eigenständig und parallel zum Programmablauf die Auswertung der Encoder-Signale, und stellt die gezählten Striche in einem Register bereit.

### Externer IC

Des Weiteren gibt es spezielle Quadratur-Decoder-ICs. Sie können wie jede andere Peripherie über SPI oder I<sup>2</sup>C mit dem Mikrocontroller verbunden werden. Sie bieten gegenüber den selbstgebauten oder integrierten Decodern unter anderem folgende Vorteile:

- Höhere Taktraten (erlaubt die Messung höherer Drehzahlen)
- Gleichzeitige Auswertung mehrerer Drehgeber in einem Chip
- Eingebaute Rausch-Filter

Abbildung 10.4 zeigt beispielhaft einen Quadraturdecoder mit Eingängen für die Kanäle A, B und Index, sowie einem SPI Interface.

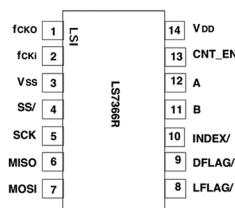


Abbildung 10.4.: Quadraturdecoder LFLS7366R [16].

### FPGA

Eine weitere Möglichkeit Quadraturencoder auszuwerten bieten FPGAs (Field Programmable Gate Array). Diese sind spezielle ICs in denen eine logische Schaltung programmiert werden kann. Ein einfacher Quadraturencoder lässt sich bereits mit der in Abbildung 10.5 dargestellten Schaltung auf einem FPGA implementieren.

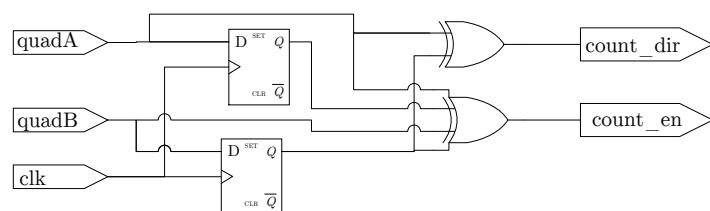


Abbildung 10.5.: Logik-Schaltplan eines einfachen Quadraturdecoders [10].

Weitere Details zur Implementierung eines Quadraturdecoder auf einem FPGA finden sich z.B. hier: <http://www.fpga4fun.com/QuadratureDecoder.html>

# 11. Bus-Systeme

Ein Bus ist ein System zur Datenübertragung zwischen mehreren Teilnehmern über einen gemeinsamen Übertragungsweg, bei dem die Teilnehmer nicht an der Datenübertragung zwischen anderen Teilnehmern beteiligt sind. Dabei unterscheidet man zwischen zwei Arten:

- Interne Bus-Systeme, z.B. innerhalb eines Mikrocontrollers, wie etwa der Daten- und der Steuer-Bus aber auch Bus-Systeme zur Kommunikation zwischen dem Mikrocontroller und Peripherie.
- Externe Bus-Systeme, z.B. zur Kommunikation zwischen PC und Drucker (USB, FireWire, ...).

In diesem Kapitel werden zwei bedeutsame interne Bus-Systeme zum Anschluss von Peripherie an den Mikrocontroller vorgestellt.

## 11.1. I<sup>2</sup>C-Bus

I<sup>2</sup>C ist ein von Philips Semiconductors (heute NXP Semiconductors) entwickelter serieller Zweidraht-Bus mit je einer Daten- und Takteleitung zur Kommunikation zwischen ICs. I<sup>2</sup>C steht für Inter-Integrated Circuit und wird als „I-Quadrat-C“ oder „I-squared-C“ gesprochen. Einige Hersteller verwenden die Bezeichnung TWI (Two-Wire Interface).

Das Konzept des Busses basiert auf einer Master-Slave Architektur, d. h. die Kontrolle der Kommunikation liegt in der Hand eines Masters, während die anderen Teilnehmer (z.B. Sensor, Speicher-ICs) nur auf die Befehle reagieren. Ein Slave kann also nie selbstständig Daten senden. In einem I<sup>2</sup>C-Bus gibt es mindestens einen Master. Konfigurationen mit mehreren Mastern sind aber möglich (Multi-Master).

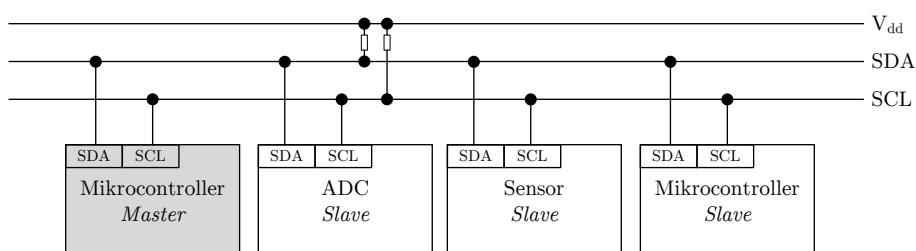


Abbildung 11.1.: Schematische Darstellung I<sup>2</sup>C.

### 11.1.1. Protokoll, Geschwindigkeit und Verwendung

Um einen Datenaustauschvorgang zu initiieren übernimmt der Master, der Daten senden oder empfangen möchte, den Bus und gibt die Adresse des Slaves, mit dem er kommunizieren möchte, aus. Im nächsten Schritt teilt er mit, ob er Daten senden oder empfangen möchte. Danach werden die Austauschdaten auf die Datenleitung gelegt - bei einem

Schreibkommando vom Master, bei einem Lesebefehl vom Slave. Sobald der Transfer vom Master abgeschlossen wird, gibt er den Bus frei. Die Übertragungsrate beträgt beim *Standard Mode* bis zu 100 kBit/s, beim *Fast Mode* bis zu 400 kBit/s (Ver.:1.0, 1992) und beim *High-Speed Mode* bis zu 3,4 MBit/s (Ver.: 2.0, 1998, Strom- und Spannungsanforderungen wurden gesenkt). Die aktuelle Version 3.0 von 2007 führte einen „extra schnellen“ Modus (*Fast Mode Plus*) mit bis zu 1 MBit/s ein, der im Gegensatz zum „Hochgeschwindigkeitsmodus“ dasselbe Protokoll verwendet wie die 100 kBit/s- und 400 kBit/s-Modi.

Der Bustakt wird immer vom Master ausgegeben. Tabelle 11.1 listet die maximal erlaubten Taktraten nochmals auf.

Tabelle 11.1.: Verschiedene I<sup>2</sup>C Modi.

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

Der Hauptvorteil von I<sup>2</sup>C ist, dass man nur einen Mikrocontroller braucht, um ein ganzes Netzwerk an integrierten Schaltungen zu kontrollieren. Außerdem werden für das sehr einfache Protokoll nur zwei I/O-Pins benötigt. Der Bus eignet sich für die Kommunikation zwischen ICs über kleine Distanzen mit geringer Übertragungsgeschwindigkeit. Hauptnachteil des Systems ist die Störanfälligkeit, was seine Verwendung auf störungssame Anwendungsbereiche einschränkt, d. h. Bereiche ohne Übersprechen (Crosstalk), Rauschen, EMV- und Kontaktprobleme (Stecker, Buchsen).

### 11.1.2. Implementierungsdetails

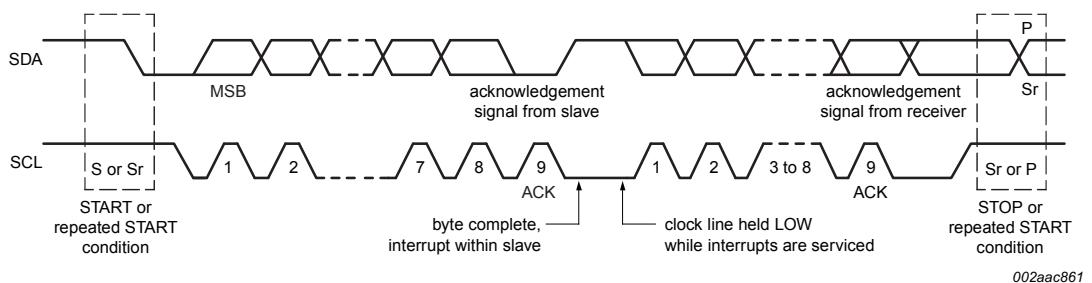


Abbildung 11.2.: Datentransfer auf einem I<sup>2</sup>C-Bus [19, S. 10].

Abbildung 11.2 zeigt den Transfer eines Datenbytes über den Bus: Der Master stellt zunächst eine Start-Bedingung am Bus her, um den Beginn einer Kommunikation anzugeben. Dann sendet er als erstes Byte die Adresse seines Partners, wobei die ersten sieben Bit die eigentliche Adresse des Slaves darstellen und das achte Bit (R/W-Bit) den Lese- oder Schreibwunsch festlegt. Das I<sup>2</sup>C-Protokoll verfügt folglich über einen Adressraum von 7 Bit, was (abzüglich 16 reserverter Adressen wie z.B. dem „General Call“ zum Senden an

alle Teilnehmer) bis zu 112 Teilnehmer auf einem Bus erlaubt.<sup>1</sup> Abhängig vom R/W-Bit werden die Daten byteweise vom Slave auf den Bus geschrieben oder aus dem Bus ausgelesen. Nach jedem Byte wird ein ACK (Acknowledgement) vom Slave (Schreibvorgang) bzw. vom Master (Lesevorgang) geschickt, wobei das letzte Byte eines Lesezugriffs vom Master mit einem NAK quittiert wird, um das Ende der Übertragung anzudeuten. Eine Übertragung wird durch das Stop-Signal beendet oder durch ein Repeated-Start-Signal von einer weiteren Übertragung fortgesetzt.

**Beispiel 11.1:** Daten aus einem externen EEPROM auslesen

Der Mikrocontroller erzeugt eine Start-Bedingung, sendet die I<sup>2</sup>C-Adresse des EEPROM mit Schreibwunsch und dann die zu lesende Speicher-Adresse im EEPROM. Dann überträgt er nach einem Repeated-Start erneut die Adresse des EEPROM, diesmal mit Lesewunsch, dieses legt den Inhalt der angeforderten Datenzelle auf den Bus. Abschließend beendet der Master die Kommunikation mit Stop.

### 11.1.3. I<sup>2</sup>C Modul der Mega-Familie

Die genaue Ansteuerung des I<sup>2</sup>C-Controllers im Mikrocontroller kann wie immer dem Datenblatt entnommen werden. Wir nennen hier nur die Namen der Register und deren grobe Funktion.

Das TWI Bit Rate Register TWBR dient gemeinsam mit dem Prescaler in TWSR (s.u.) zur genauen Einstellung der I<sup>2</sup>C-Busgeschwindigkeit mithilfe des internen Takts gemäß der Formel

$$\text{SCL frequency} = \frac{\text{CPU Clock frequency}}{16 + 2(\text{TWBR}) \cdot 4^{\text{TWPS}}} \quad (11.1)$$

Das TWI Control Register TWCR schaltet das Interface mithilfe des Bits TWEN (TWI Enable) ein, erlaubt die Aktivierung des lokalen I<sup>2</sup>C-Interrupts und beinhaltet das zugehörige Flag (TWIE und TWINT). Des Weiteren kann die Erzeugung einer *Start-* bzw. *Stop-Bedingung* angeordnet werden (Bits TWSTA und TWSTO) und festgelegt werden, ob beim Empfang der eigenen Adresse bzw. des nächsten Bytes vom *Slave* ein *Acknowledge* (ACK) oder *Non-Acknowledge* (NACK) gesendet werden soll.

Das Status-Register TWSR enthält in den Bits TWS3-7 den Status-Code des Interfaces und codiert den Prescaler zu TWBR (siehe Abb. 11.1.3). Das TWI Data Register TWDR enthält das letzte empfangene Byte bzw. muss mit dem zu sendenden Byte beschrieben werden.

Das TWI Address Register TWAR des Controllers im *Slave-Modus* sollte seine Adresse beinhalten. Das TWI General Call Recognition Enable Bit TWGCE legt dabei fest, ob sich der Mikrocontroller von *General Calls*<sup>2</sup> angesprochen fühlt oder nicht.

<sup>1</sup>Für eine Erweiterung des Adressraums wurde später eine 10 Bit-Adressierung eingeführt. Sie ist abwärtskompatibel zum 7-bit-Standard durch Nutzung von 4 der 16 reservierten Adressen. Beide Adressierungsarten sind gleichzeitig verwendbar, was bis zu 1136 Knoten auf einem Bus erlaubt.

<sup>2</sup>Daten werden für alle teilnehmenden Slaves zusammen verschickt, als Adresse wird 0 gewählt

**TWI Bit Rate Register – TWBR**

Bit	7	6	5	4	3	2	1	0	
	TWBR7	TWBR6	TWBR5	TWBR4	TWBR3	TWBR2	TWBR1	TWBR0	TWBR
Read/Write	R/W								

(a) TWBR

**TWI Control Register – TWCR**

Bit	7	6	5	4	3	2	1	0	
	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	–	TWIE	TWCR
Read/Write	R/W	R/W	R/W	R/W	R	R/W	R	R/W	

(b) TWCR

Abbildung 11.3.: TWI Control Register.

**TWI Status Register – TWSR**

Bit	7	6	5	4	3	2	1	0	
	TWS7	TWS6	TWS5	TWS4	TWS3	–	TWPS1	TWPS0	TWSR
Read/Write	R	R	R	R	R	R	R/W	R/W	

(a) TWSR

TWPS1	TWPS0	Prescaler Value
0	0	1
0	1	4
1	0	16
1	1	64

(b) TWPS Bit Beschreibung

Abbildung 11.4.: TWI Status Register.

## 11.2. SPI-Bus

Der SPI-Bus (**Serial Peripheral Interface Bus**) ist ein von Motorola entwickelter synchrone, serieller Datenbus. Die Datenübertragung geschieht über drei Leitungen, an die jeder Bus-Teilnehmer angeschlossen ist:

- **SDO** (*Serial Data Out*) bzw. **MOSI** (*Master Out / Slave In*)
- **SDI** (*Serial Data In*) bzw. **MISO** (*Master In / Slave out*)
- **SCK** (*Serial Clock*)

Die wichtigsten Unterschiede zum TWI (I<sup>2</sup>C) Bus sind:

- Jeder Slave-Teilnehmer ist mit einer eigenen *Chip-Select*-Leitung ( $\overline{CS}$  /  $\overline{SS}$  /  $\overline{STE}$ ) mit dem Master verbunden. Über diese steuert der Master, mit welchem Slave er kommunizieren möchte.
- Da sowohl eine *Data Out* als auch eine *Data In* Leitung vorhanden sind, ist der SPI-Bus **vollduplexfähig** (gleichzeitige Senden und Empfangen von Daten möglich).

Abbildung 11.5 zeigt beispielhaft die Verschaltung eines Masters mit drei Slaves.

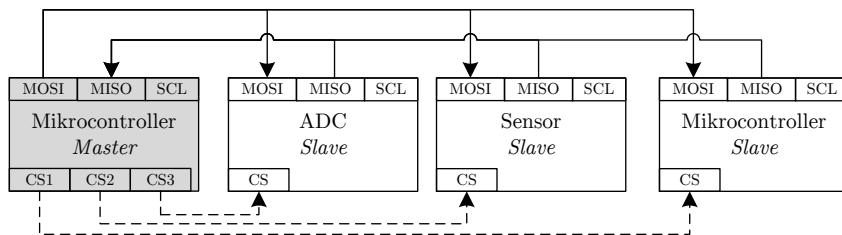


Abbildung 11.5.: Schematische Darstellung SPI.

### 11.2.1. Protokoll, Geschwindigkeit und Verwendung

Im Gegensatz zum I<sup>2</sup>C-Bus, gibt es für den SPI keinen offizieller Standard. In der Praxis bedeutet dies zwar, dass man bei der Implementierung eines Protokolls große Gestaltungsspielraum hat, andererseits aber auch viele verschiedene Implementierungen existieren, die nicht zwingend kompatibel sind. Da jeder Slave direkt über die SS-Leitung ausgewählt wird, benötigen die Geräte am SPI keine Adressen und kein festes Protokoll zur Kommunikation. Der SPI-Bus unterstützt theoretisch beliebig hohe Taktraten, die in der Praxis jedoch durch die Taktfrequenz der Mikrocontrollers oder aber häufiger noch der angeschlossenen Peripherie begrenzt wird. SPI wird nahezu überall verwendet, wo ein Controller mit Peripherie kommuniziert. Wenn die zusätzlichen Leitungen kein Problem darstellen, ist der SPI durch seine einfache Implementierung und hohe Datenraten dem I<sup>2</sup>C-Bus vorzuziehen. Typische Anwendungen des SPI-Bus sind beispielsweise:

- Auslesen von Sensoren; z.B. Temperatur, Druck, A/D-Wandler
- Verbindung mit anderen Bus-Controllern; z.B. USB, CAN
- Lesen und Schreiben auf externe Speicher; z.B. EEPROM, Flash, MMC, SD-Karten

### 11.2.2. Implementierungsdetails

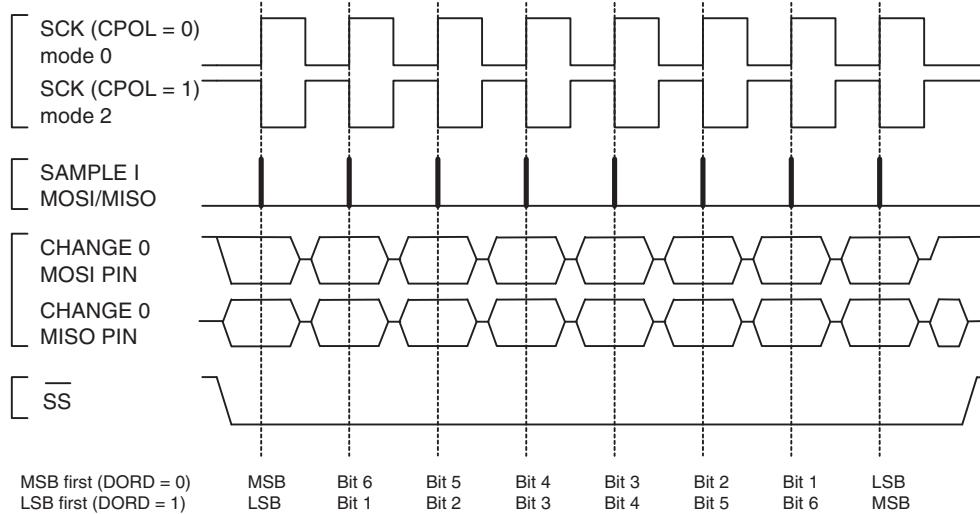
Beim SPI-Bus darf es stets nur einen Master geben. Dieser wählt über die Chip-Select-Leitungen den Slave aus, mit dem er kommunizieren möchte. Es sind auch Schaltungen möglich, bei denen stets alle Slaves aktiv sind (s.g. „Daisy Chain Mode“). In diesem Fall spricht der Master nur den ersten Slave an. Dieser wiederum gibt alle Daten an den zweiten Slave weiter usw. Die Daten werden also durch alle Slaves durchgereicht. Diese müssen dann allerdings wieder softwareseitig entscheiden, ob die Daten an sie adressiert sind oder nicht.

Möchte der Master kommunizieren, so zieht er die zum Slave führende SS-Leitung nach unten, und gibt über die SCK-Leitung ein Taktsignal aus. Im Takt der SCL-Leitung werden nun MOSI bzw. MISO geschalten um die Daten zu übertragen. Bei der Interpretation des gesendeten Signals werden verschiedene Möglichkeiten unterschieden:

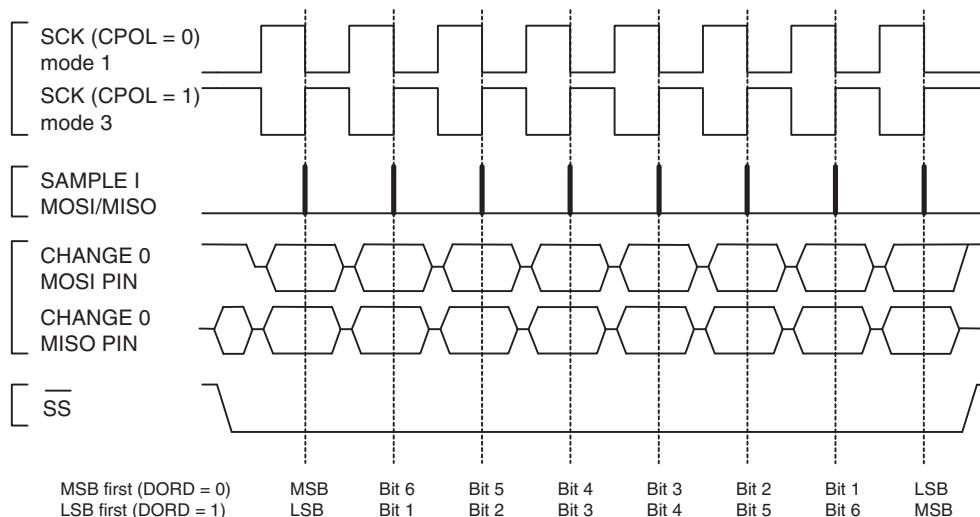
- Die Clock-Polarity (CPOL) legt fest, ob *high* oder *low* auf SCL der Referenzzustand ohne Takt (Clock-Idle) ist. Bei CPOL = 0 ist der Clock Idle *low*, bei CPOL = 1 entsprechend *high*.

- Die *Clock-Phase* (CPHA) legt nun fest, wann die Zustand der MOSI- bzw. MISO-Leitung als neues Bit in das Schieberegister übernommen werden soll. Bei CPHA = 0 geschieht dies bei der ersten Flanke (*leading edge*) eines Taktsignals, bei CPHA = 1 entsprechend bei der zweiten Flanke (*trailing edge*).

Möchte der Master die Kommunikation beenden, so lässt er einfach die SS-Leitung „los“, und beendet sein Taktsignal. Die zeitliche Abfolge für die verschiedenen Modi ist in Abbildung 11.6 noch einmal graphisch dargestellt.



(a) CPHA = 0



(b) CPHA = 1

Abbildung 11.6.: SPI Übertragung mit verschiedenen Modi [5, S. 128].

Obwohl es keinen offiziellen Standard für die Übertragung über SPI gibt, haben sich die vier möglichen Kombinationen von CPHA und CPOL als Quasi-Standard etabliert (siehe Tabelle 11.2) und finden sich so auch beispielsweise im ATmega-Handbuch [5].

Mode	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

Tabelle 11.2.: Verschiedene SPI Modi.

**Beispiel 11.2:** SPI Übertragung in Modus 0

1. Zunächst zieht der Master die  $\overline{SS}$ -Leitung zum entsprechenden Slave auf *low*
2. Nun legt er ein Taktsignal auf die SCK Leitung. Im Modus 0 ist diese ohne Takt auf *low* und bei einem Takt auf *high* (da CPOL = 0).
3. Gleichzeitig ändert er bei jedem Takt den Zustand der MOSI-Leitung entsprechend dem zu übertragenden Bit. Bei CPHA = 0 geschieht das Sampling (Aufnehmen) des Bits bei der ersten Flanke eines Taktes, in diesem Fall also bei einer steigenden Flanke. Zu diesem Zeitpunkt muss die Datenleitung MOSI den gewünschten Zustand eingenommen haben.
4. Jetzt wird Bit für Bit aus dem Schieberegister über die Datenleitung übertragen. Dies kann, je nach Einstellung, mit dem höchstwertigen Bit beginnen (MSB first) oder entsprechend andersherum (LSB first).
5. Nach der Übertragung des Bytes setzt der Master die  $\overline{SS}$ -Leitung wieder auf *high*

**11.2.3. SPI Modul der Mega-Familie**

Wie bei der Beschreibung des I<sup>2</sup>C soll hier nur eine kurze Übersicht über die wichtigsten und grundlegenden Konzepte des SPI-Moduls im ATmega gegeben werden.

Wie üblich erfolgt die grundlegende Einstellung des SPI-Moduls über das Control Register SPCR (siehe Abb. 11.7(a)). Das SPIE-Bit aktiviert den SPI-Interrupt, der nach einer erfolgreichen Übertragung aufgerufen wird. Das SPE-Bit dient der Aktivierung des SPI-Moduls, während das MSTR-Bit festlegt ob der Controller als *Master* oder *Slave* fungiert. Mithilfe der Bits DORD, CPOL und CPHA werden die Datenreihenfolge sowie die Signalinterpretation (Bedeutung der Flankenwechsel) festgelegt. Die Taktrate des SPI-Bus wird über die Bits SPI2X, SPI1 und SPR0 eingestellt (Abb. 11.7(b)).

**SPI Control Register – SPCR**

Bit	7	6	5	4	3	2	1	0	SPCR
Read/Write	R/W								
Initial Value	0	0	0	0	0	0	0	0	

(a) SPI Control Register (SPCR)

SPI2X	SPR1	SPR0	SCK Frequency
0	0	0	$f_{osc}/4$
0	0	1	$f_{osc}/16$
0	1	0	$f_{osc}/64$
0	1	1	$f_{osc}/128$
1	0	0	$f_{osc}/2$
1	0	1	$f_{osc}/8$
1	1	0	$f_{osc}/32$

(b) SPI Taktraten-Einstellungen

**SPI Status Register – SPSR**

Bit	7	6	5	4	3	2	1	0	SPSR
Read/Write	R	R	R	R	R	R	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

(c) SPI Status Register (SPSR)

**SPI Data Register – SPDR**

Bit	7	6	5	4	3	2	1	0	SPDR
Read/Write	R/W								
Initial Value	X	X	X	X	X	X	X	X	Undefined

(d) SPI Data Register (SPDR)

Abbildung 11.7.: SPI-Register des ATmega8 [5, S. 125ff].

Das SPI Status Register (SPSR, Abb. 11.7(c)) enthält wichtige Informationen zur Steuerung der SPI-Kommunikation: Das SPIF-Flag wird automatisch auf 1 gesetzt, sobald die Übertragung abgeschlossen ist. Werden die Daten im Ausgaberegister während einer Übertragung geändert, wird eine „*Write Collision Flag*“ (WCOL) gesetzt. Das SPI Data Register (SPDR, Abb. 11.7(d)) dient der Übertragung der Daten vom Speicher in das SPI-Schieberegister. Nach dem Beschreiben des Registers werden die Daten automatisch über SPI übertragen. Beim Lesen des Registers hingegen, werden die Daten aus dem Schieberegister gelesen.

Im ATmega8 Handbuch [5, S. 123f] findet sich ein Beispiel-Code zur Übertragung von Daten, sowohl auf Master- als auch Slave-Seite. Das Senden von Daten geschieht beispielsweise folgendermaßen:

```
void SPI_MasterTransmit(char cData)
{
    /* Write Data SPDR register , starts the transmission */
    SPDR = cData;
    /* Wait for transmission complete */
    while (!(SPSR & (1<<SPIF)));
}
```

# 12. Flachheitsbasierte Steuerung und Regelung

## 12.1. Grundlagen

Das Konzept der *Flachheit* wurde 1992 von Fliess, Lévine, Martin und Rouchon eingeführt [8]. Es beschreibt eine Klasse nichtlinearer Systeme mit mindestens einer Steuergröße, die eine Verallgemeinerung der linearen steuerbaren Systeme darstellen [21]. Die Flachheit eines nichtlinearen Systems ermöglicht einen systematischen Entwurf von Steuerungen und Regelungen zur Trajektorienfolge [21].

**Definition:** *Flaches System* [21]

Ein nichtlineares System

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}) \quad (12.1)$$

mit  $\mathbf{x} \in \mathbb{R}^n$  und  $\mathbf{u} \in \mathbb{R}^m$  heißt *flach*, falls ein Ausgang

$$\mathbf{y} = \mathbf{h}(\mathbf{x}, \mathbf{u}, \dots, \mathbf{u}^{(\alpha)}) \quad (12.2)$$

mit  $\mathbf{y} \in \mathbb{R}^m$  existiert, der die folgenden Bedingungen erfüllt:

Der Zustandsvektor  $\mathbf{x}$  und der Eingangsvektor  $\mathbf{u}$  können als Funktionen des Ausgangsvektors und einer endlichen Anzahl seiner Zeitableitungen eindeutig ausgedrückt werden:

$$\mathbf{x} = \psi_1(\mathbf{y}, \dot{\mathbf{y}}, \dots, \mathbf{y}^{(\beta)}) = \psi_1(y_1, \dot{y}_1, \dots, y_1^{(\beta_1)}, \dots, y_m, \dot{y}_m, \dots, y_m^{(\beta_m)}) \quad (12.3)$$

$$\mathbf{u} = \psi_2(\mathbf{y}, \dot{\mathbf{y}}, \dots, \mathbf{y}^{(\beta+1)}) = \psi_2(y_1, \dot{y}_1, \dots, y_1^{(\beta_1+1)}, \dots, y_m, \dot{y}_m, \dots, y_m^{(\beta_m+1)}) \quad (12.4)$$

Der Ausgang  $\mathbf{y}$  wird als flacher Ausgang bezeichnet.

Der *flache Ausgang ist nicht eindeutig*, d.h. für ein flaches System existieren beliebig viele verschiedene solcher Ausgänge.

Das größte Problem bei der Anwendung einer flachheitsbasierten Steuerung oder Regelung ist das Auffinden eines flachen Ausgangs, wofür es bisher keine allgemein anwendbare Methode gibt [21].

Ist aber ein solcher Ausgang gefunden, kann eine Trajektorienfolgeregelung mit wenigen Schritten entworfen werden [15]. Dies wird im Folgenden am Beispiel einer Steuerung bzw. Regelung für den Kleinroboter vorgestellt.

## 12.2. Beispiel: Trajektorienfolge

Für den Kleinroboter soll eine Trajektorienfolgeregelung entworfen werden. Der Kleinroboter soll hierbei fest vorgegebene Trajektorien abfahren. Angetrieben wird der Roboter nur durch die einzeln angesteuerten Hinterräder. Abbildung 12.1(a) zeigt die Kinematik des einfachen (siehe Kapitel 12.2.1) und erweiterten (siehe Kapitel 12.2.2) Systemmodells.

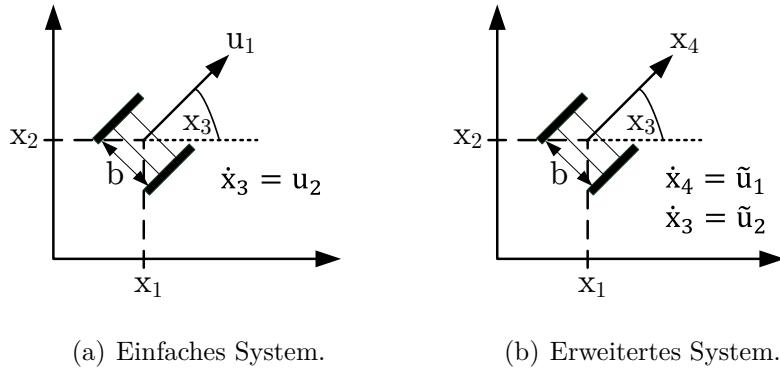


Abbildung 12.1.: Zustände und Eingangsgrößen des Kleinrobotermodells.

### 12.2.1. Steuerung

Zur Beschreibung des Kleinroboter-Systems wurden die in Abbildung 12.1 dargestellten Zustandsgrößen und Eingänge gewählt. Das in Abb. 12.1(a) dargestellte System wird durch die folgenden Differentialgleichungen beschrieben [23]:

$$\dot{x}_1 = u_1 \cdot \cos x_3 \quad \text{Position } x \quad (12.5)$$

$$\dot{x}_2 = u_1 \cdot \sin x_3 \quad \text{Position } y \quad (12.6)$$

$$\dot{x}_3 = u_2 \quad \text{Fahrtrichtung} \quad (12.7)$$

wobei die Stellgrößen  $u_1$  (Längsgeschwindigkeit) und  $u_2$  (Drehrate) über

$$u_1 = \frac{v_l + v_r}{2} \quad (12.8)$$

$$u_2 = \frac{v_r - v_l}{b} \quad (12.9)$$

mit den Geschwindigkeiten  $v_r$  und  $v_l$  der Räder verknüpft sind. Die Konstante  $b$  beschreibt hierbei den Abstand der beiden Räder. [23]. Da wir die Trajektorie für die Position  $[x_1, x_2]$  des Roboters vorgeben wollen, werden die Ausgänge

$$y_1 = x_1 \quad (12.10)$$

$$y_2 = x_2 \quad (12.11)$$

gewählt. Die Ausgänge werden nun so oft nach der Zeit abgeleitet, bis die Ableitungen von den beiden Eingängen  $u_1$  und  $u_2$  abhängen:

$$\dot{y}_1 = \dot{x}_1 = u_1 \cdot \cos x_3 \quad (12.12)$$

$$\dot{y}_2 = \dot{x}_2 = u_1 \cdot \sin x_3 \quad (12.13)$$

$$\ddot{y}_1 = \dot{u}_1 \cdot \cos x_3 - u_1 \cdot \sin x_3 u_2 \quad (12.14)$$

$$\ddot{y}_2 = \dot{u}_1 \cdot \sin x_3 + u_1 \cdot \cos x_3 u_2 \quad (12.15)$$

Nun hat man genügend Gleichungen, um die Zustände  $x_i$  und die Eingänge  $u_i$  als Funktionen der Ausgänge und ihrer Ableitungen auszudrücken [23]:

$$x_1 = y_1 \quad (12.16)$$

$$x_2 = y_2 \quad (12.17)$$

$$x_3 = \text{atan}\left(\frac{\dot{y}_2}{\dot{y}_1}\right) \quad (12.18)$$

$$u_1 = \sqrt{\dot{y}_1^2 + \dot{y}_2^2} \quad (12.19)$$

$$u_2 = \dot{x}_3 = \frac{\ddot{y}_2 \dot{y}_1 - \dot{y}_2 \ddot{y}_1}{\dot{y}_1^2 + \dot{y}_2^2} \quad (12.20)$$

Es handelt sich also bei den interessierenden Ausgängen um flache Ausgänge. Werden nun die Solltrajektorie  $[y_{1,d}, y_{2,d}]$  und ihre Zeitableitungen in die Gleichungen (12.19) und (12.20) eingesetzt, erhält man direkt das gesuchte Steuergesetz.

### 12.2.2. Regelung

Um auftretenden Störungen entgegenzuwirken, wird eine Zustandsregelung für das System entworfen. Eine Voraussetzung für die im Praktikum verwendete Reglerentwurfsmethodik ist, dass die Summe über alle  $\beta_i + 1$  gleich der Systemordnung ist [23]:

$$\sum_{i=1}^m (\beta_i + 1) = n \quad (12.21)$$

Für das in Abb. 12.1(a) dargestellte System ist dies leider nicht der Fall. Deshalb führt man mit  $x_4 = u_1$ ,  $\dot{x}_4 = \tilde{u}_1$  und  $\tilde{u}_2 = u_2$  den neuen Zustand  $x_4$  und die neuen Eingänge  $\tilde{u}_1$  und  $\tilde{u}_2$  ein. Damit erhält man die Darstellung

$$\dot{x}_1 = x_4 \cdot \cos x_3 \quad (12.22)$$

$$\dot{x}_2 = x_4 \cdot \sin x_3 \quad (12.23)$$

$$\dot{x}_3 = \tilde{u}_2 \quad (12.24)$$

$$\dot{x}_4 = \tilde{u}_1 \quad (12.25)$$

für das System. Man beachte, dass der neue Eingang  $\tilde{u}_2$  identisch zum Eingang  $u_2$  in der Systemdarstellung (12.5) - (12.7) ist, und lediglich der einheitlichen Notation halber mit einer Tilde versehen wurde.

Statt der Längsgeschwindigkeit  $u_1$ , wird jetzt die Längsbeschleunigung  $\tilde{u}_1 = \dot{u}_1$  vorgegeben, so dass  $(\beta_1 + 1) + (\beta_2 + 1)$  nun der Systemordnung (jetzt  $n = 4$ ) entspricht

[23]:

$$\dot{y}_1 = x_4 \cos x_3 \quad (12.26)$$

$$\dot{y}_2 = x_4 \sin x_3 \quad (12.27)$$

$$\ddot{y}_1 = \tilde{u}_1 \cos x_3 - x_4 \sin x_3 \tilde{u}_2 \quad (12.28)$$

$$\ddot{y}_2 = \tilde{u}_1 \sin x_3 + x_4 \cos x_3 \tilde{u}_2 \quad (12.29)$$

Die beiden Eingänge  $\tilde{u}_1$  und  $\tilde{u}_2$  können wie folgt in Abhängigkeit des flachen Ausgangs und dessen Zeitableitungen dargestellt werden:

$$\tilde{u}_1 = \frac{d}{dt} \left( \sqrt{\dot{y}_1^2 + \dot{y}_2^2} \right) = \frac{\ddot{y}_1 \dot{y}_1 + \ddot{y}_2 \dot{y}_2}{\sqrt{\dot{y}_1^2 + \dot{y}_2^2}} \quad (12.30)$$

$$\tilde{u}_2 = \frac{d}{dt} \left( \operatorname{atan} \left( \frac{\dot{y}_2}{\dot{y}_1} \right) \right) = \frac{\ddot{y}_2 \dot{y}_1 - \ddot{y}_1 \dot{y}_2}{\dot{y}_1^2 + \dot{y}_2^2} \quad (12.31)$$

Nun sollen zunächst  $\tilde{u}_1$  und  $\tilde{u}_2$  so gewählt werden, dass  $\ddot{y}_1$  und  $\ddot{y}_2$  über die neuen Eingänge

$$v_1 = y_2^{(\beta_1+1)} = \ddot{y}_1 \quad (12.32)$$

$$v_2 = y_2^{(\beta_2+1)} = \ddot{y}_2 \quad (12.33)$$

unabhängig von einander vorgegeben werden können. Dazu setzt man (12.32) und (12.33) in (12.30) ein, und (12.31) und erhält die Stellgrößen

$$\tilde{u}_1 = \frac{v_1 \dot{y}_1 + v_2 \dot{y}_2}{\sqrt{\dot{y}_1^2 + \dot{y}_2^2}} \quad (12.34)$$

$$\tilde{u}_2 = \frac{v_2 \dot{y}_1 - v_1 \dot{y}_2}{\dot{y}_1^2 + \dot{y}_2^2} \quad (12.35)$$

durch die das System in die beiden Integratorketten  $\ddot{y}_1 = v_1$  und  $\ddot{y}_2 = v_2$  zerlegt und folglich entkoppelt wird. Dieselben Ausdrücke für  $\tilde{u}_1$  und  $\tilde{u}_2$  erhält man in dem man die rechten Seiten von (12.28) und (12.29) gleich  $v_1$  bzw.  $v_2$  setzt, nach  $\tilde{u}_1$ ,  $\tilde{u}_2$  auflöst und die Zustände  $x_3$  und  $x_4$  durch den flachen Ausgang darstellt.

Nun soll dem Folgefehler  $e_i = y_{i,d} - y_i$  eine lineare (asymptotisch stabile) Dynamik zugewiesen werden. Dazu wählt man

$$v_i = \ddot{y}_{i,d} + a_{i,1}(\dot{y}_{i,d} - \dot{y}_i) + a_{i,0}(y_{i,d} - y_i) \quad (12.36)$$

was zu einer Fehlerdynamik zweiter Ordnung

$$\ddot{e}_i + a_{i,1}\dot{e}_i + a_{i,0}e_i = 0 \quad (12.37)$$

führt, wie sich leicht durch Einsetzen von (12.36) in die Integratorketten  $\ddot{y}_i = v_i$  überprüfen lässt. Für eine stabile Fehlerdynamik müssen alle  $a_{i,j} > 0$  gewählt werden (Hurwitz-Kriterium) [23]. Setzt man den Zusammenhang 12.36 in Gl. (12.34) und (12.35) ein, erhält man das Regelgesetz für die neuen Eingänge:

$$\begin{aligned} \tilde{u}_1 &= \cos x_3 (\ddot{y}_{1,d} + a_{11}(\dot{y}_{1,d} - x_4 \cos x_3) + a_{10}(y_{1,d} - x_1)) + \dots \\ &\dots + \sin x_3 (\ddot{y}_{2,d} + a_{21}(\dot{y}_{2,d} - x_4 \sin x_3) + a_{20}(y_{2,d} - x_2)) \end{aligned} \quad (12.38)$$

$$\begin{aligned}\tilde{u}_2 &= \frac{\cos x_3}{x_4} (\ddot{y}_{2,d} + a_{21}(\dot{y}_{2,d} - x_4 \sin x_3) + a_{20}(y_{2,d} - x_2)) - \dots \\ &\dots - \frac{\sin x_3}{x_4} (\ddot{y}_{1,d} + a_{11}(\dot{y}_{1,d} - x_4 \cos x_3) + a_{10}(y_{1,d} - x_1))\end{aligned}\quad (12.39)$$

Man beachte hierbei die Singularität bei  $x_4 = 0$  in (12.39).

**Der Winkel (Fahrtrichtung) kann nicht ohne eine Längsbewegung ( $x_4 \neq 0$ ) geändert werden.** Bei der Implementierung des Regelgesetzes muss der Fall  $x_4 = 0$  gesondert behandelt werden. Die Eingänge  $\tilde{u}_i$  können nun wieder in die ursprünglichen Eingänge  $u_i$  umgerechnet werden [23]:

$$u_1 = \int_0^t \tilde{u}_1 d\tau \quad (12.40)$$

$$u_2 = \tilde{u}_2 \quad (12.41)$$

# A. Bootloader

## A.1. Integration des AVR rootloaders in das AtmelStudio

### A.1.1. Voraussetzungen

Für die Integration des Bootloaders in das AtmelStudio ist es nötig, dass folgende Dateien auf dem Computer gespeichert sind:

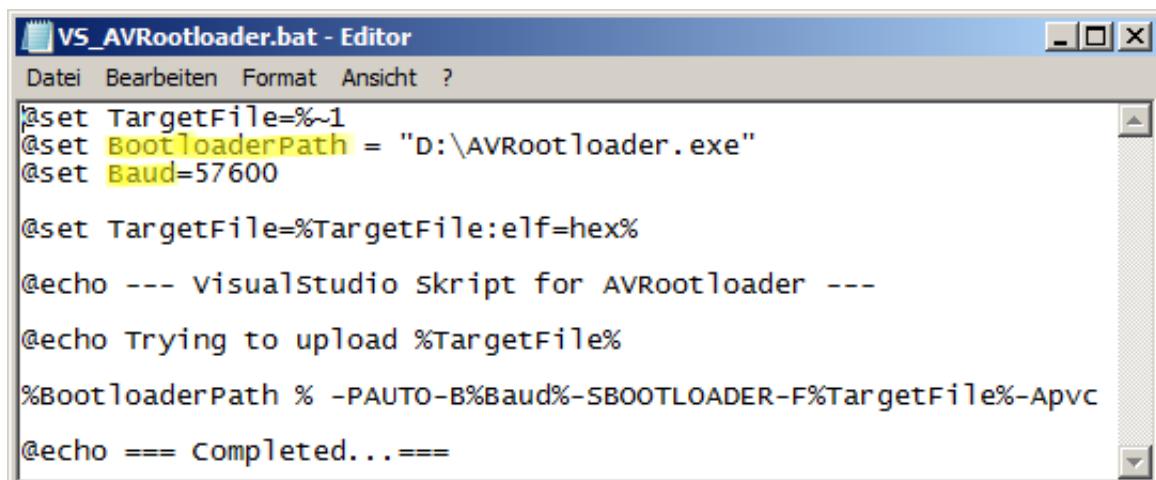
1. Die AVR rootloader Windows Anwendung (AVR rootloader.exe) sowie dazugehörige Dateien
2. Das Batch-Skript VS\_AVRootloader.bat

Wichtig ist, dass immer die modifizierte *AVRootloader.ini* verwendet wird. Diese ist notwendig, da nur dort die Timings für eine erfolgreiche Kommunikation über Bluetooth enthalten sind.

Der AVR rootloader kann nicht mit „-“-Zeichen im Pfad und in Dateinamen umgehen. Aus diesem Grund darf der Pfad zum AVR rootloader und auch der Pfad zu allen Atmel Studio Projekten und auch der Projektnamen kein „-“-Zeichen enthalten!

### A.1.2. Anpassen der Batch-Datei

Zunächst muss die Batch-Datei an die jeweilige Umgebung angepasst werden. Dies bedeutet, dass unter *BootloaderPath* der vollständige Pfad zur AVR rootloader.exe angegeben wird. Zusätzlich kann die gewünschte Baudrate unter *Baud* geändert werden.



```
VS_AVRootloader.bat - Editor

Datei Bearbeiten Format Ansicht ?

@set TargetFile=%~1
@set BootloaderPath = "D:\AVR rootloader.exe"
@set Baud=57600

@set TargetFile=%TargetFile:elf=hex%
@echo --- visualstudio skript for AVR rootloader ---
@echo Trying to upload %TargetFile%
%BootloaderPath% -PAUTO-B%Baud%-SBOOTLOADER-F%TargetFile%-Apvc
@echo === Completed...==
```

Abbildung A.1.: Anpassen des Batch-Skripts

### A.1.3. Einbinden in AtmelStudio

Nun kann der Bootloader als externes Werkzeug ins Atmel Studio eingebunden werden. Dazu wählt man in der Menüleiste den Eintrag *Tools → External Tools*. Falls die Option *External Tools* nicht zur Verfügung steht, muss Atmel Studio zuvor über *Tools → Select Profile* in den „Advanced UI Modus“ umgeschaltet werden. Im folgenden Fenster wird mit *Add* ein neues Werkzeug erstellt und mit den in Abb. A.2 gezeigten Einstellungen versehen. Im Feld *Command* muss dabei der Pfad zum Batch-Skript VS\_AVRootloader.bat angegeben werden. Jetzt erscheint der AVRootloader bereits im Menü *Tools*.

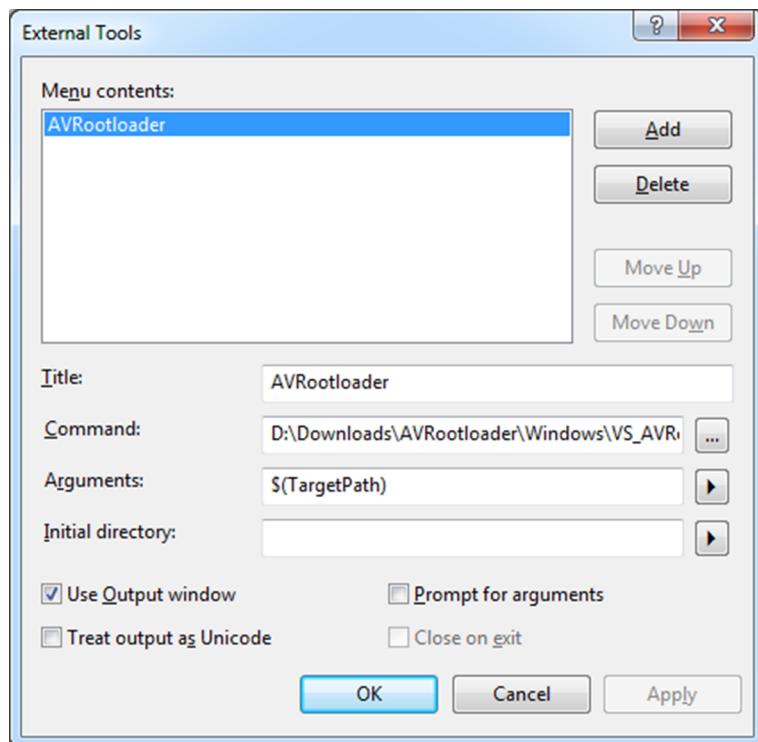


Abbildung A.2.: Erstellen eines neuen externen Werkzeugs

### A.1.4. Erstellen eines Buttons zum Bootloader

Zunächst wird mit einem Rechtsklick auf die Toolbar der Eintrag *Customize* aufgerufen (Abb. A.3(a)). Dort wird im Reiter *Commands* unter *Toolbar* die gewünschte Toolbar ausgewählt (Abb. A.3(b)). Hier bietet sich *Build* an, da der Bootloader Button dann neben den Buttons zum erstellen des Programms erscheint. Durch klicken auf *AddCommand* wird der Leiste ein neuer Befehl hinzugefügt. Hier wählt man unter *Tools* den Eintrag *External Command 1* aus (ggf. anpassen).

Nun ist der Bootloader in das AtmelStudio integriert und es erscheint ein Button *AVRootloader* in der Toolbar. Durch klicken auf den Button *AVRootloader* wird die aktuelle kompilierte Datei (\*.hex) auf den Mikrocontroller überspielt. **Das heißt, das Projekt muss vorher durch Klicken auf Erstellen kompiliert werden.**

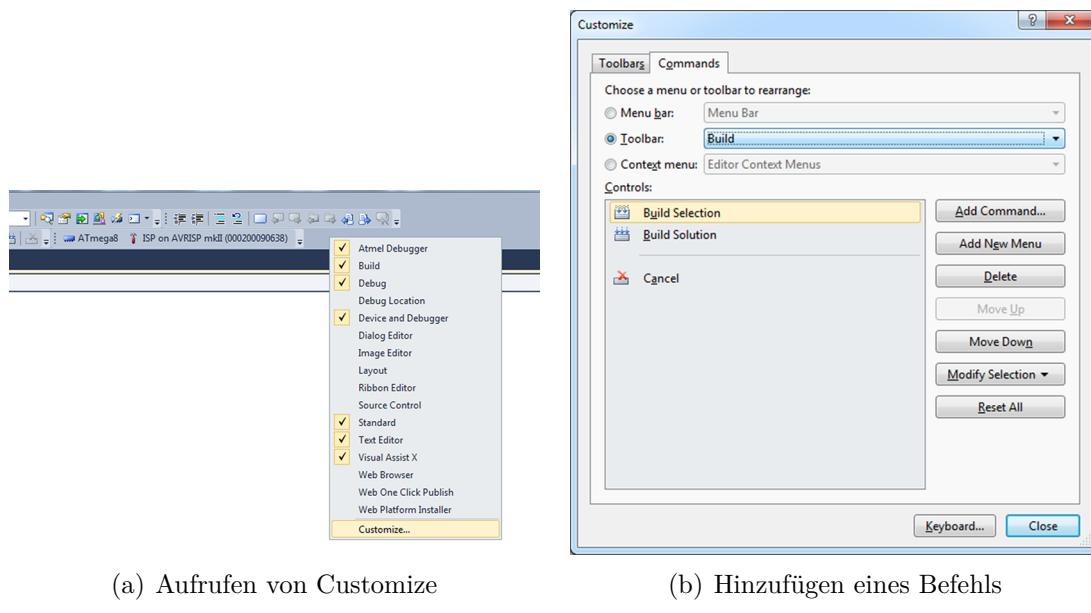


Abbildung A.3.: Bearbeiten der Toolbar

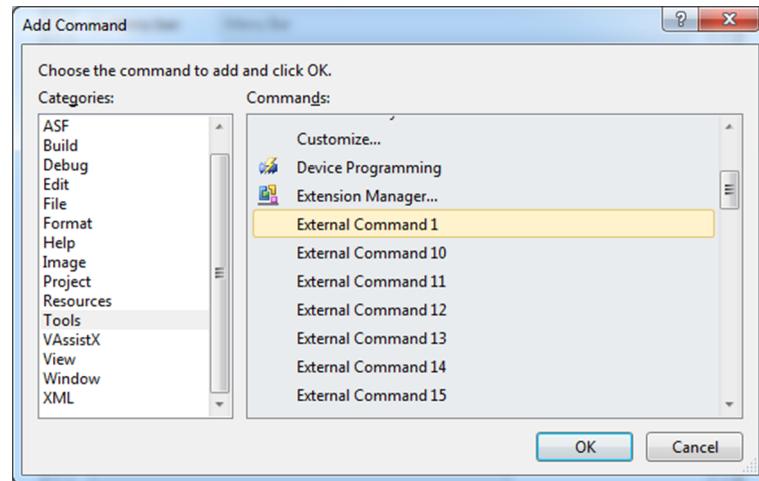


Abbildung A.4.: Fenster AddCommand



Abbildung A.5.: Fertig integrierter Bootloader

### A.1.5. Vorgehensweise zum Flashen des Mikrocontrollers

Zusammenfassend noch einmal das Vorgehen, wie der Code auf den Mikrocontroller kommt:

1. Kompilieren des Codes durch klicken auf *Projekt Erstellen* bzw. *Build Solution (F7)*.
2. Ausgabe prüfen ob Kompilieren fehlerfrei abgelaufen ist.
3. Klicken auf *AVRootloader*. Der Bootloader-Client wird geöffnet und verbindet sich mit dem Mikrocontroller. Wenn das Programm erfolgreich überspielt wurde, schließt sich der Bootloader-Client automatisch und in der Ausgabezeile erscheint das Wort *Completed*.

Falls der Mikrocontroller mit einem Programm bespielt ist, dass nicht auf den **BOOTLOADER**-Befehl reagiert, wird das AVRootloader-Programm keine Verbindung zum Bootloader herstellen können. In diesem Fall muss der Mikrocontroller von Hand resetted werden:

1. AVRootloader starten
2. Mikrocontroller durch drücken der Reset-Taste zurücksetzen.
3. Nun hat man 1s Zeit um in der Bootloader-GUI den Button *Connect* zu drücken.
4. Wird der *Connect*-Button rechtzeitig gedrückt wird die Verbindung erfolgreich hergestellt und das Programm kann hochgeladen werden.

# B. Ausgabe von Fließkommazahlen

## B.1. Der printf-Befehl

### B.1.1. Verwendung des sprintf-Befehls

Mithilfe des `printf`-Befehls können verschiedenste Zahlen formatiert ausgegeben werden. Soll das Ergebnis statt auf der Konsole (die der Mikrocontroller ja nicht hat) in einer Zeichenfolge (*String*) gespeichert werden, verwendet man den Befehl `sprintf`:

```
int sprintf ( char * str , const char * format , ... );
```

Hierfür muss zunächst eine Variable definiert werden, in die die Zeichenfolge geschrieben wird. Hierzu wird ein Array von `char` mit ausreichender Größe definiert und der `sprintf` als erster Parameter (`str`) übergeben. Der zweite Parameter ist der s.g. Formatstring, der die Formatierung der Zahlenwerte festlegt. Anschließend werden alle auszugebenden Zahlenwerte als weitere Parameter angefügt.

Im folgenden Beispiel werden die beiden Zahlen `a` und `b` formatiert und in die Zeichenfolge `buffer` geschrieben. Die Variable `buffer` enthält dann den Satz: „Eine Zahl ist 5, die andere Zahl ist 3“.

```
#include <stdio.h>

int main ()
{
    char buffer [50];
    int a=5;
    int b=3;
    sprintf(buffer , "Eine Zahl ist %d , die andere ist %d" , a , b);
}
```

Der Format-String enthält beliebigen Text, wobei die Stellen, an denen eine Zahl eingefügt werden soll durch das `%` markiert sind. Die darauf folgenden Zahlen und Buchstaben (**specifier**) legen fest, welcher Datentyp an dieser Stelle eingefügt werden soll, und wie er ausgegeben wird (siehe Tab. B.1).

Tabelle B.1.: Wichtige Format-Specifier

specifier	Format	Beispiel
d	Vorzeichenbehaftete Dezimalzahl	-392
u	Vorzeichenlose Dezimalzahl	392
s	Zeichenfolge	Hallo
e	Wissenschaftliche Notation	3.923e4
f	Fließkommazahl	234.13

Bei Fließkommazahlen (specifier `f`) kann zusätzlich noch die Anzahl der Nachkommastellen definiert werden. Die Anweisung

```
sprintf(buffer, "Meine Zahl ist %.2f", 67.456345);
```

würde beispielsweise den Satz „Meine Zahl ist 67.46“ ausgeben.

Ein vollständige Beschreibung der verschiedenen Formatierungsmöglichkeiten findet man beispielsweise unter <http://wwwcplusplus.com/reference/cstdio/printf/>

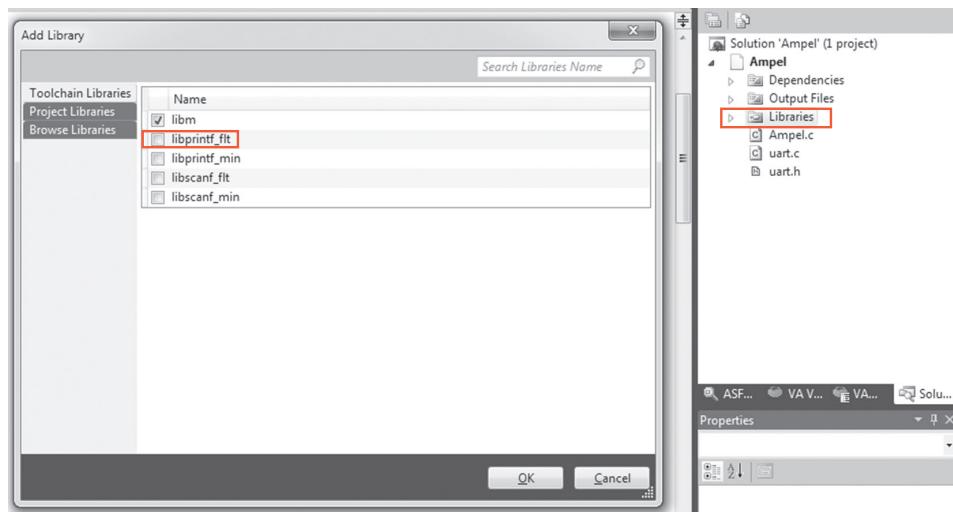
Die Nutzung von `sprintf()` benötigt viel Speicher. Bei der Fließkomma-Variante ist der Speicherbedarf noch einmal deutlich größer. Integer-Werte können auch mit der ressourcensparenderen Funktion `itoa()` in eine Zeichenfolge konvertiert werden.

### B.1.2. Nutzung der Fließkomma-Version des `sprintf`-Befehls

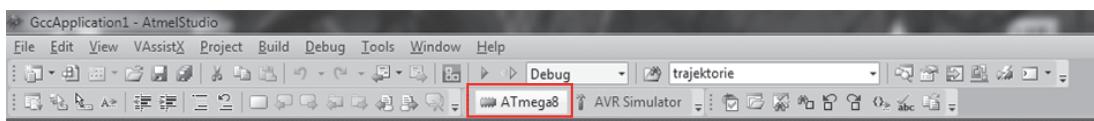
Das AVR-Studio verwendet standardmäßig eine Version der `printf`-Bibliothek, die keine Fließkommazahlen unterstützt.

Um die Nutzung der (größeren) Fließkomma-Version zu erzwingen, muss die Bibliothek `libprintf_flt` eingebunden, sowie die Linker-Option `-Wl, -u, vfprintf` aktiviert werden. Das Einbinden der Bibliothek geschieht durch Rechtsklick auf den Ordner **Libraries** im **Solution Explorer** (siehe Abb. B.1(a)). Die Linker-Optionen werden, wie in Abbildung B.1(b) und B.1(c) gezeigt, über die Projekteigenschaften eingestellt.

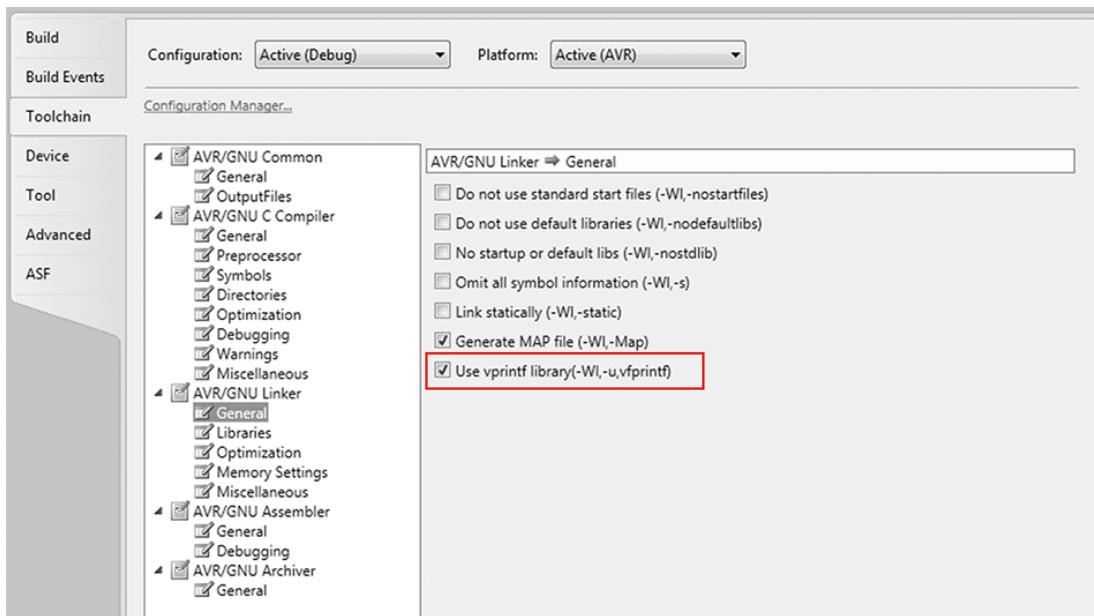
Der GCC-Compiler des AVR-Studios erfordert, dass `float`-Variablen zuerst nach `double` gecastet werden, bevor sie mit der Float-Variante von `sprintf` ausgegeben werden.



(a) Einbinden der libprintf\_flt-Bibliothek.



(b) Öffnen der Projekteigenschaften



(c) Aktivieren der Linker-Option „-Wl, -u, vfprintf“.

Abbildung B.1.: Verwendung der Fließkomma-Version des printf-Befehls.

# C. Ampelplatine

## C.1. Kurzbeschreibung Ampelplatine



Abbildung C.1.: Bild der Ampelplatine

### C.1.1. Komponenten

Die Ampelplatine besteht aus den folgenden in Abb. C.2 dargestellten Hauptkomponenten:

1. Atmel ATmega8 Mikrocontroller
2. 16MHz Quarz als Taktquelle für den Mikrocontroller
3. 2 Darlingtonarrays als Treiber für die LEDs
4. 4 „Ampeln“ mit je drei LEDs
5. Reset-Taster
6. 2 Taster (S2, S3)
7. Zwei-Achsen-Potentiometer (Joystick) mit integriertem Taster (S1)
8. Phototransistor als Lichtsensor
9. Lautsprecher (Summer)
10. UART-USB-Brückenchip zur Kommunikation mit dem PC
11. 2 Jumper zur Konfiguration

Der vollständige Schaltplan ist in Abbildung C.3 dargestellt.

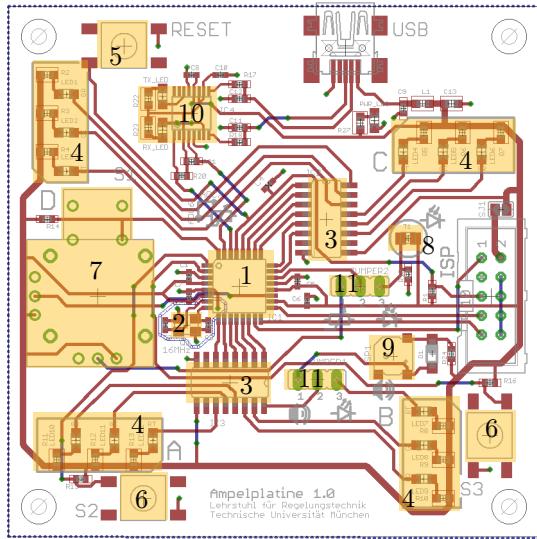


Abbildung C.2.: Layout der Ampelplatine

### C.1.2. Stromversorgung

Die Stromversorgung der Ampel-Platine erfolgt über den USB-Anschluss. Die grüne LED unterhalb des USB-Anschlusses leuchtet sobald die Platine an die Stromversorgung angeschlossen ist.

### C.1.3. USB-UART-Brücke

Der ATmega8 selbst verfügt lediglich über eine UART-Schnittstelle. Um eine einfache Kommunikation mit dem Computer zu ermöglichen verfügt die Platine über einen USB-UART-Brückenchip von FTDI (FT230X). Dieser erscheint am Computer als virtuelle serielle Schnittstelle (COM-Port) und ermöglicht so die einfache Kommunikation mit dem Mikrocontroller (z.B. über Realterm oder MATLAB). Die benötigten Treiber werden unter Windows 7 automatisch installiert (Internetverbindung nötig), oder können alternativ unter <http://www.ftdichip.com/Drivers/VCP.htm> heruntergeladen werden. Die beiden LEDs direkt an der UART-Brücke leuchten auf, wenn Daten empfangen bzw. versendet werden.

### C.1.4. Pin-Belegung

Die einzelnen LEDs der Ampeln sind an die in Tabelle C.1 dargestellten Pins des Mikrocontrollers angeschlossen. Die Pinbelegung aller anderen Komponenten kann Tabelle C.2 entnommen werden. Die Ports B2 sowie ADC7 sind doppelt belegt. Die jeweilige Funktion muss mithilfe der Jumper ausgewählt werden.

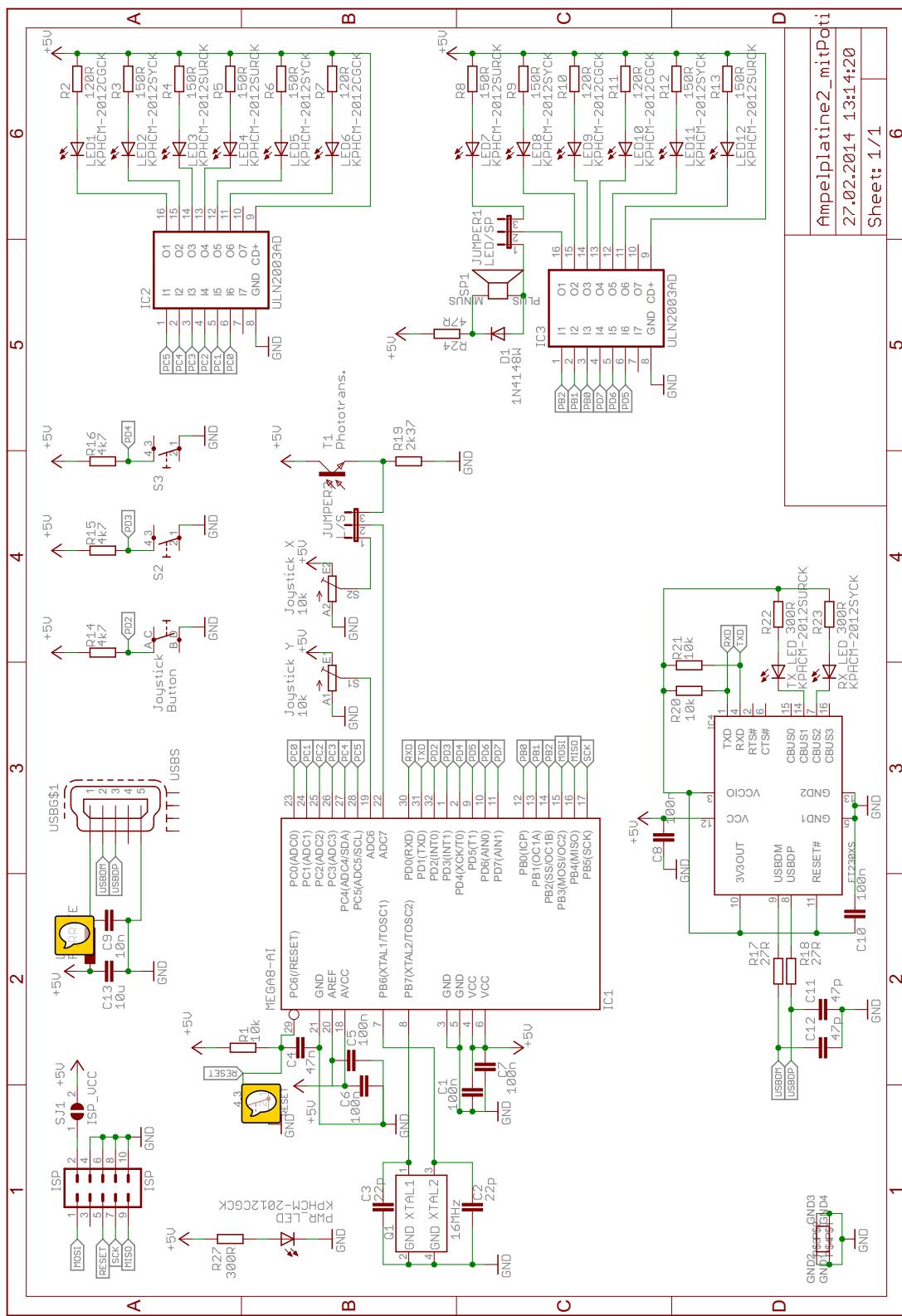


Abbildung C.3.: Schaltplan der Ampelplatine

Tabelle C.1.: Pinbelegung der LEDs

	Ampel A	Ampel B	Ampel C	Ampel D
<b>Rot</b>	D5	B2	C2	C3
<b>Gelb</b>	D6	B1	C1	C4
<b>Grün</b>	D7	B0	C0	C5

Tabelle C.2.: Pinbelegung der anderen Komponenten

<b>Taster S1</b>	D2 (INT0)	<b>Joystick X</b>	ADC6
<b>Taster S2</b>	D3 (INT1)	<b>Joystick Y</b>	ADC7
<b>Taster S3</b>	D4	<b>Lichtsensor</b>	ADC7
<b>Summer</b>	B2		

### C.1.5. Jumper-Konfiguration

Da sowohl der Anschluss B2 als auch der Anschluss ADC7 doppelt belegt sind, muss die jeweils gewünschte Funktion mithilfe der Jumper festgelegt werden. Die jeweiligen Funktionen sind unterhalb der Jumper symbolisch dargestellt, und in Tabelle C.3 erklärt.

### C.1.6. Programmierung

Der verbaute ATmega8 Mikrocontroller kann direkt über die ISP<sup>1</sup>-Schnittstelle programmiert werden. Da hierfür jedoch zusätzliche Hardware (*Programmer*) nötig ist, wird im Praktikum eine andere elegante Möglichkeit verwendet.

Alle Platinen sind mit einem Bootloader-Programm bespielt. Mithilfe dieses kleinen Programmes kann sich der Mikrocontroller selbst programmieren. Beim Starten des Mikrocontrollers wird zunächst der Bootloader ausgeführt. Dieser wartet eine Sekunde, ob ihm per UART ein neues Programm gesendet wird. Ist dies der Fall, wird das neue Programm in den Speicher geschrieben und anschließend gestartet. Empfängt der Bootloader keine Daten, so startet er direkt das im Speicher hinterlegte Programm.

Aufgrund der einfach zu bedienenden Benutzeroberfläche verwenden wir im Praktikum den Bootloader AVR-Rootloader<sup>2</sup>. Dieser besteht aus dem Bootloader, welcher auf dem Mikrocontroller ausgeführt wird und einer Windows-Anwendung zum Übertragen neuer

<sup>1</sup>In System Programming

<sup>2</sup>[http://www.mikrocontroller.net/articles/AVR-Bootloader\\_mit\\_Verschl%C3%BCsselung\\_von\\_Hagen\\_Re](http://www.mikrocontroller.net/articles/AVR-Bootloader_mit_Verschl%C3%BCsselung_von_Hagen_Re)

Tabelle C.3.: Jumper-Konfiguration

Verbunden	1-2	2-3
Jumper 1	Summer an B2	Rote Ampel B an B2
Jumper 2	Joystick X-Achse an ADC7	Lichtsensor an ADC7

Programme auf den Mikrocontroller.

Da der Bootloader nur beim Start des Mikrocontrollers aktiv ist, kann auch nur zu diesem Zeitpunkt ein neues Programm eingespielt werden. Prinzipiell ist es nötig, den Reset-Taster der Platine zu drücken und innerhalb von einer Sekunde den Programmiervorgang zu starten.

Es gibt jedoch auch eine elegantere Variante: Der Bootloader-Client sendet beim Verbindungsversuch den Text **BOOTLOADER** an den Mikrocontroller. Die vom Lehrstuhl bereitgestellte UART-Bibliothek `uart.h` reagiert auf diesen Text und führt einen softwareseitigen Neustart aus, so dass der Mikrocontroller neu startet und ein neu-programmieren ermöglicht.

## C.2. Interrupt-Vektoren im ATmega8

Vector No.	Program Address <sup>(2)</sup>	Source	Interrupt Definition
1	0x000 <sup>(1)</sup>	RESET	External Pin, Power-on Reset, Brown-out Reset, and Watchdog Reset
2	0x001	INT0	External Interrupt Request 0
3	0x002	INT1	External Interrupt Request 1
4	0x003	TIMER2 COMP	Timer/Counter2 Compare Match
5	0x004	TIMER2 OVF	Timer/Counter2 Overflow
6	0x005	TIMER1 CAPT	Timer/Counter1 Capture Event
7	0x006	TIMER1 COMPA	Timer/Counter1 Compare Match A
8	0x007	TIMER1 COMPB	Timer/Counter1 Compare Match B
9	0x008	TIMER1 OVF	Timer/Counter1 Overflow
10	0x009	TIMER0 OVF	Timer/Counter0 Overflow
11	0x00A	SPI, STC	Serial Transfer Complete
12	0x00B	USART, RXC	USART, Rx Complete
13	0x00C	USART, UDRE	USART Data Register Empty
14	0x00D	USART, TXC	USART, Tx Complete
15	0x00E	ADC	ADC Conversion Complete
16	0x00F	EE_RDY	EEPROM Ready
17	0x010	ANA_COMP	Analog Comparator
18	0x011	TWI	Two-wire Serial Interface
19	0x012	SPM_RDY	Store Program Memory Ready

- Notes:
1. When the BOOTRST Fuse is programmed, the device will jump to the Boot Loader address at reset, see "["Boot Loader Support – Read-While-Write Self-Programming"](#) on page 202
  2. When the IVSEL bit in GICR is set, Interrupt Vectors will be moved to the start of the boot Flash section. The address of each Interrupt Vector will then be the address in this table added to the start address of the boot Flash section

Abbildung C.4.: Interrupt-Vektoren des ATmega8 [5, S.46].

# D. Kleinroboter: Balancieren und Trajektorienfolge

## D.1. Aufbau KRT8

Der Aufbau des KRT8 ist in Abbildung D.1 visualisiert. Angetrieben wird der Roboter von zwei Gleichstrommotoren. Der Radabstand beträgt etwa 9.9 cm, der Durchmesser der Räder 6.6 cm. Als Spannungsversorgung dient ein 1000 mAh Lithium-Polymer Akku. Auf den Motorwellen sitzt jeweils eine Encoderscheibe mit 900 Ticks pro Umdrehung. Die Auswertung der Encoder-Signale erfolgt auf der Platine durch zwei Quadraturzähler LSI LS7366R (Datenblatt siehe Moodle). Die Kommunikation zwischen Mikrocontroller und den Quadraturzählern geschieht über den SPI Bus. Weitere Peripherie sind die Drehraten- und Beschleunigungssensoren. Erstere sind über den I<sup>2</sup>C Bus mit dem Mikrocontroller verbunden, letztere über den SPI Bus. Als Mikrocontroller kommt ein ATmega32 zum Einsatz. Da auf einen externen Quarz als Taktgeber verzichtet wurde, läuft der Mikrocontroller mit den 8 MHz der internen Clock.

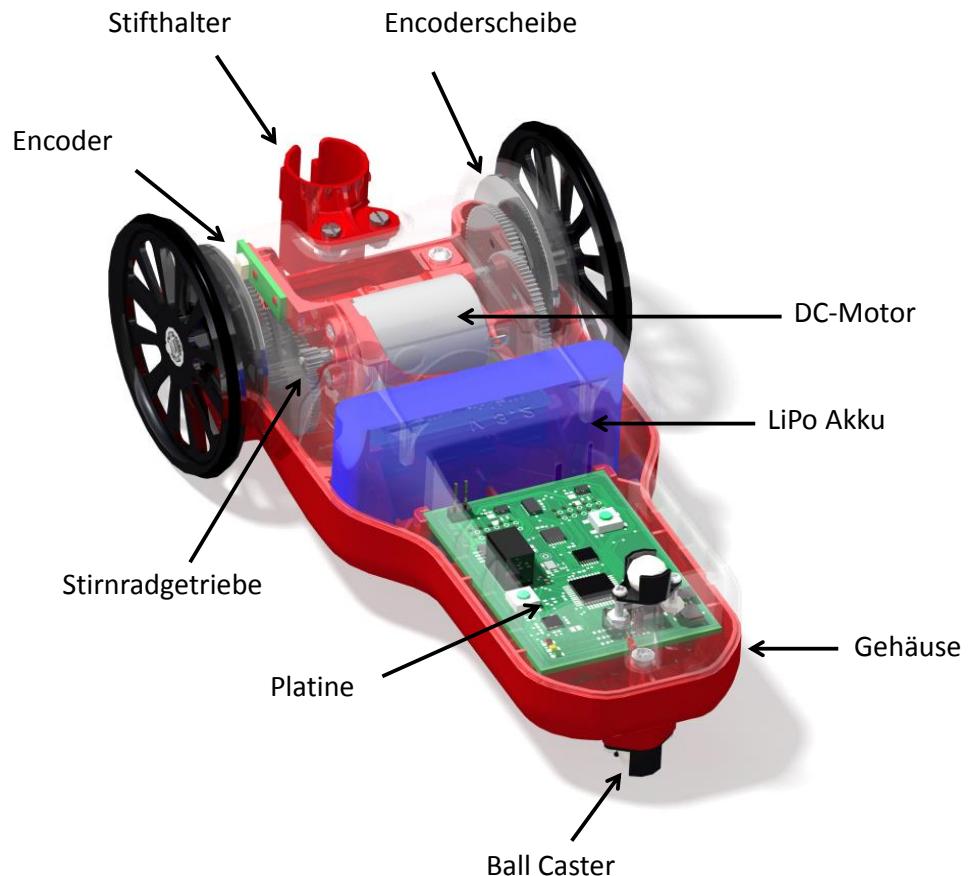


Abbildung D.1.: KRT8-Roboter

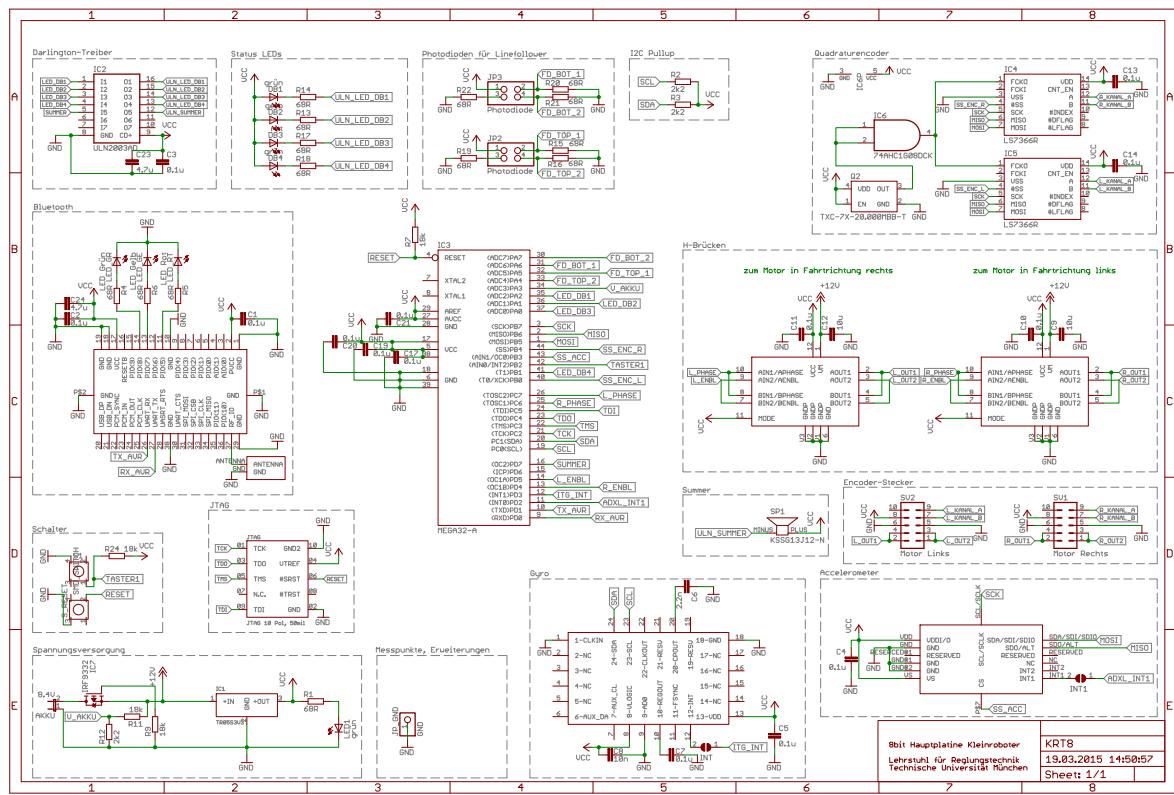


Abbildung D.2.: Schaltplan KRT8-Platine

## D.2. Bluetooth Verbindung

Zur einfachen und drahtlosen Datenübertragung besitzt der KRT8 ein Bluetooth Modul, um über die serielle Schnittstelle mit dem PC kommunizieren zu können. Das Modul des Kleinroboters wurde so konfiguriert, dass es den Namen des Kleinroboters trägt, also z.B. KRT8-1 für den Kleinroboter Nummer 1. Als PIN wird beim Pairing einfach 19222 angegeben. Die Baudrate beträgt 38400 bps. Schritt für Schritt erfolgt der Aufbau der Bluetooth Verbindung folgendermaßen:

- Stromversorgung herstellen (Akku einstecken)
- Bluetooth-Dongle am PC einstecken, bzw. Bluetoothmodul des PC aktivieren
- Neues Bluetooth-Gerät hinzufügen
- KRT8-X auswählen (X durch die eigene(!) Roboter-Nummer ersetzen)
- Als Schlüssel 19222 eingeben
- Der KRT8 erscheint nun als COM-Port und kann wie zuvor die Ampelplatine, nun aber mit der Baudrate 38400 bps verwendet werden

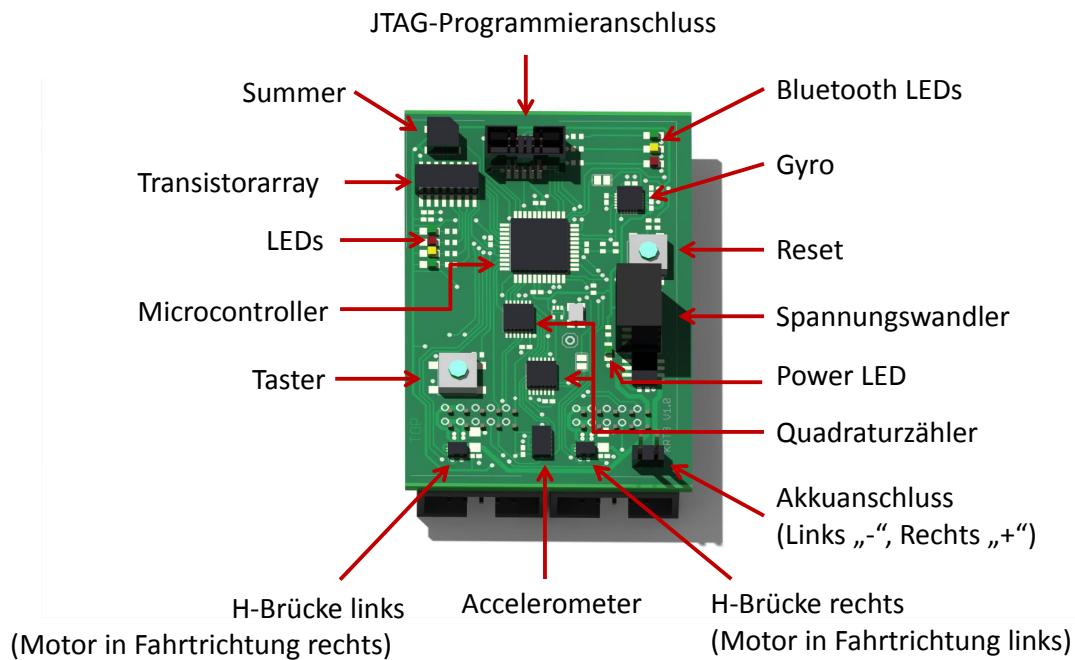


Abbildung D.3.: KRT8-Platine

Tabelle D.1.: LEDs des BTM222 Bluetooth Moduls

<b>Farbe</b>	<b>PIO</b>	<b>Pin</b>	<b>Funktion</b>
rot	5	11	signalisiert Austausch von Daten per Bluetooth
gelb	7	13	signalisiert Status der Bluetoothkopplung

Tabelle D.2.: Status der Bluetooth Kopplung

<b>Gelbe LED</b>	<b>Bedeutung</b>
Aus	keine Bluetoothverbindung (Modul vermutlich abgestürzt)
An	Bluetoothverbindung besteht (Pairing)
blinkend (0,1 s)	wartet, auf Verbindung, Slave Mode
blinkend (0,3 s)	Erkennbar, wartet auf Verbindung, Slave Mode
blinkend (0,9 s)	Anfrage, Master Mode
blinkend (1,2 s)	Verbinde, Master Mode

## D.3. H-Brücken

Zur Motoransteuerung werden zwei H-Brücken DRV8835 von Texas Instruments eingesetzt (Datenblatt siehe Moodle). Die H-Brücken werden im PHASE/ENABLE Mode betrieben. Die vier Transistoren der H-Brücke müssen nicht einzeln geöffnet und geschlossen werden, sondern werden von der internen Logik des Bauteils angesteuert. Aus dem Datenblatt und dem Schaltplan des KRT8 lässt sich die Ansteuerung der Modi entnehmen (siehe Tabelle D.3).

Tabelle D.3.: Ansteuerung der H-Brücken

Rechtes Rad	PD5(OC1A)	PC7
1	0	vorwärts
1	1	rückwärts
0	0/1	Bremse
Linkes Rad	PD4(OC1B)	PC6
1	0	vorwärts
1	1	rückwärts
0	0/1	Bremse

*Hinweis:* Die linke H-Brücke steuert den in Fahrtrichtung rechten Motor an, die rechte H-Brücke den Fahrtrichtung linken Motor. Unterschiede im Motorverhalten machen es notwendig, im Verlauf der Reglerentwicklung individuelle Motorkennlinien zu implementieren (siehe Aufgabe D.6.3).

## D.4. Sensorik

### D.4.1. Quadraturzähler

Die Quadraturzähler des KRT8 kommunizieren mit dem ATmega32 über den SPI Bus, dessen Einrichtung im Laufe des Praktikums erfolgt. Das *Manual* (Lesen dringend empfohlen) der Quadraturzähler LS7366R beschreibt detailliert das Kommunikationsprotokoll zwischen Mikrocontroller und dem Quadraturzähler und die auf dem Chip vorhandenen Register. Im Praktikum werden lediglich die Register MDR0 und MDR1 zur Konfiguration und das Register CNTR verwendet, welches den aktuellen Zählerstand beinhaltet. Makros (zu finden im *Manual*) erleichtern wie auch beim Mikrocontroller das Senden von Instructionbytes und das Setzen der entsprechenden Bits in den Konfigurationsregistern.

### D.4.2. Beschleunigungssensoren

Ebenso wie die Quadraturzähler kommuniziert der Beschleunigungssensor über den SPI Bus mit dem ATmega32. Im *Quick Start Guide* des Beschleunigungssensor ADXL345 ist beschrieben, wie das Auslesen der Beschleunigung in x-, y- und z-Richtung funktioniert.

Zugehörige Makros (siehe auch das *ADXL345 Manual*) sind bereits in der Header-Datei `sensorenSPI.h` definiert.

### D.4.3. Drehratensensoren

Die Drehratensensoren des KRT8 sind über den I<sup>2</sup>C Bus mit dem ATmega32 verbunden. Dieser ist bereits eingerichtet, sodass Sie die Messwerte über die Funktionen im `sensorenI2C` Modul bequem abfragen können. Ausgegeben werden können die Drehraten um die x-, y- und z-Achse (siehe Abbildung D.12).

### D.4.4. Sensordatenfusion

Stehen mehrere Sensoren in einem Aufbau zur Verfügung, so lassen sich unter Umständen gewisse Größen mit verschiedenen Sensoren messen. Dies kann ausgenutzt werden um mittels einer Sensordatenfusion einen besseren (sprich genaueren, rauschärmeren, offset-freien, ...) Messwert zu erhalten. Beim Kleinroboter ist dies für den Kippwinkel  $\alpha$  der Fall (siehe Abbildung D.12). Eine Möglichkeit, diesen zu bestimmen, ist über den verbauten Drehratensensor (Gyro). Um von der Drehrate  $\dot{\alpha}$  auf den Winkel  $\alpha_{Gyro}$  zu schließen, muss diese lediglich aufintegriert werden:

$$\alpha_{Gyro} = \int_0^t (\dot{\alpha} + \Delta\dot{\alpha}_{Gyro}) dt. \quad (\text{D.1})$$

Wichtig ist zu beachten, dass der Messwert immer fehlerbehaftet ( $\Delta\dot{\alpha}_{Gyro}$ ) ist. Im Falle des Drehratensensors setzt sich der Fehler aus Rauschen und einer langsamem Drift zusammen. Durch die Integration wird zwar das Rauschen geglättet, der Drift führt jedoch dazu, dass durch Integration die Abweichung vom echten Wert immer größer wird. Abbildung D.4 zeigt die gemessene Drehrate  $\dot{\alpha}_{Gyro}$  und den Winkel  $\alpha_{Gyro}$  für den Kleinroboter in unveränderter aufrechter Position. Gut zu erkennen sind der langsame Drift der Drehrate, dessen Auswirkung auf den Winkel sowie der rauscharme Verlauf des Winkels. Schon bereits nach 20 Sekunden weicht der gemessene Winkel vom tatsächlichen um mehr als 5° ab. Die zweite Möglichkeit zur Bestimmung des Winkels  $\alpha$  ist über den Beschleunigungssensor (Accelerometer). Durch geeignete trigonometrische Betrachtungen kann aus der gemessenen Beschleunigung in x- und in z-Richtung auf den Winkel geschlossen werden (siehe Abbildung D.5). Dies geschieht unter Annahme, dass lediglich die Erdbeschleunigung gemessen wird, also  $\mathbf{f}_{\text{meas}} = -\mathbf{g}$  gilt. Das Problem bei dieser Methode ist, dass die Beschleunigungsmesswerte stark verrauscht und entgegen unserer Annahme von kinematischen Beschleunigungen überlagert sind, und so zwar ein drift- und offsetfreier, jedoch verrauschter Winkelwert entsteht (siehe Abbildung D.6).

Es liegen also zwei Messungen für  $\alpha$  vor, deren Fehler in unterschiedlichen Zeitskalen angesiedelt sind. Die Winkelmessung aus dem Beschleunigungssensor hat ein stabiles Langzeitverhalten (kein Drift, kein Offset), ist jedoch stark verrauscht.  $\alpha_{Gyro}$  hingegen hat ein gutes Kurzzeitverhalten (kein Rauschen), driftet aber bei längeren Zeiten stark ab. Liegt eine solche Konstellation an Messwerten vor, eignet sich ein Komplementärfilter um die positiven Aspekte beider Signale zu vereinen und die Nachteile zu eliminieren.

Das Komplementärfilter in seiner ursprünglichen Form besteht aus einem Tiefpass- und einem Hochpassfilter (siehe Abbildung D.7). Die Messwerte mit gutem Langzeit- und

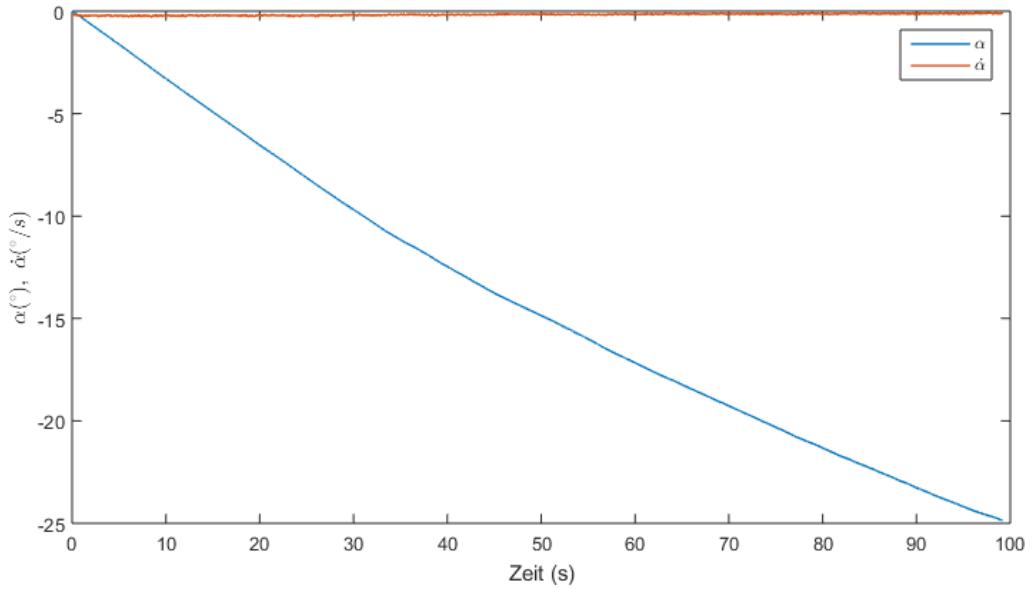
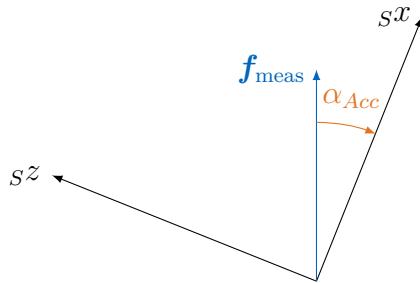


Abbildung D.4.: Drift des Drehratensensors und Auswirkung auf die Winkelbestimmung

Abbildung D.5.: Zusammenhang zwischen gemessenen Beschleunigungen und angenommenem Winkel  $\alpha_{Acc}$ 

schlechtem Kurzzeitverhalten werden mit dem Tiefpassfilter gefiltert, sodass nur noch das gute Langzeitverhalten übrig bleibt. Die Werte mit schlechtem Langzeit- und guten Kurzzeitverhalten werden analog dazu mit einem Hochpass gefiltert. Um nun sowohl bei langsamem als auch bei schnellen Winkeländerungen eine gute Schätzung für den Winkel zu erhalten, werden beide Signale nach der Filterung zusammenaddiert. Das Besondere beim Komplementärfilter ist, dass zusätzlich die Bedingung  $G_{TP}[z] + G_{HP}[z] = 1$  an das Übertragungsverhalten der Filter gestellt wird. Dadurch wird garantiert, dass eine konstante Amplitudenverstärkung über alle Frequenzbereiche gegeben ist. Wird nun diese Bedingung umgeformt und in die Gleichung für die Berechnung des geschätzten Winkels  $\alpha_{Komp}$  eingesetzt, so ergibt sich:

$$\begin{aligned}\alpha_{Komp} &= G_{TP}[z]\alpha_{Acc} + G_{HP}[z]\alpha_{Gyro} = G_{TP}[z]\alpha_{Acc} + (1 - G_{TP}[z])\alpha_{Gyro} \\ &= \alpha_{Gyro} + G_{TP}[z](\alpha_{Acc} - \alpha_{Gyro}).\end{aligned}\quad (\text{D.2})$$

Das Komplementärfilter lässt sich also allein durch einen Tiefpassfilter realisieren, wodurch Rechenzeit und Speicherplatz gespart werden kann (siehe Abbildung D.7 rechts).

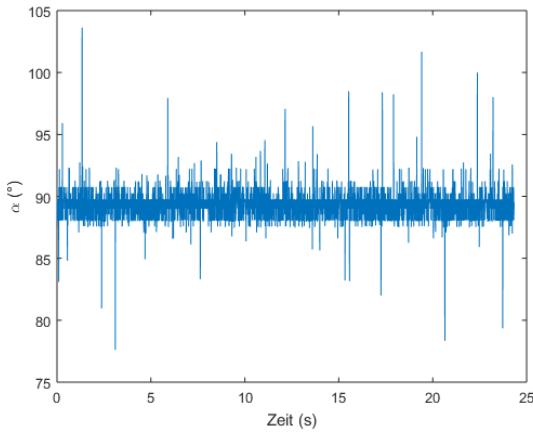


Abbildung D.6.: Verrauschter Winkelwert für liegenden Kleinroboter

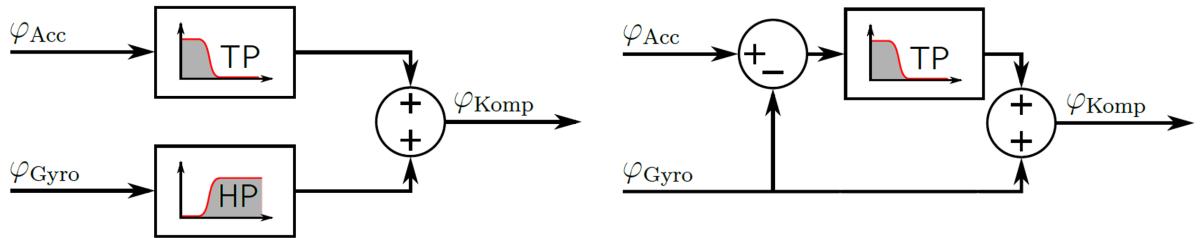


Abbildung D.7.: Komplementärfilter (links: ursprünglich, rechts: vereinfacht)[14]

Allgemein hat ein zeitdiskretes Filter die Form:

$$F[z] = \frac{b_1 + b_2 z^{-1} + \dots + b_{n+1} z^{-n}}{a_1 + a_2 z^{-1} + \dots + a_{m+1} z^{-m}}. \quad (\text{D.3})$$

Es wird in der Regel zwischen zwei Filtertypen unterschieden, den Infinite-Impulse-Response Filtern (IIR) und Finite Impulse Response Filtern (FIR). FIR Filter zeichnen sich dadurch aus, dass alle ihre Pole in 0 liegen ( $a_1, \dots, a_m = 0$ ) und sie somit immer stabil sind. Ihr Nachteil gegenüber den IIR Filtern ist, dass für den gleichen Amplitudenabfall eine deutlich höhere Ordnung benötigt wird, was bei der Implementierung mit höherem Rechen- und Speicherbedarf einhergeht und zu einer größeren Phasenverschiebung führt. Aus diesem Grund ist es empfehlenswert einen IIR-Filter für den Einsatz auf dem Mikrocontroller zu verwenden. Bei der Auslegung des IIR Filters gilt es nun noch die Filterordnung sowie die Zähler- und Nenner Koeffizienten zu bestimmen. Eine höhere Filterordnung bedeutet im Allgemeinen einen stärkeren Anstieg bzw. Abfall des Amplitudenganges an der Grenzfrequenz (siehe Abbildung D.8). Gleichzeitig erhöht sich jedoch dadurch auch die Phasenverschiebung, was sich negativ auf den geschätzten Wert und somit auch negativ auf die Regelung ausüben kann. Es ist daher mit der Wahl der Filterordnung immer ein Kompromiss zwischen steilem Anstieg bzw. Abfall des Amplitudenganges und der unerwünschten Phasenverschiebung zu schließen.

Durch die Wahl der Filterkoeffizienten kann der Verlauf des Amplitudenganges festgelegt werden. Für einen IIR Tiefpassfilter gibt es verschiedene Methoden diese zu bestimmen, wodurch gewisse charakteristische Amplitudengänge erreicht werden. Die bekannt-

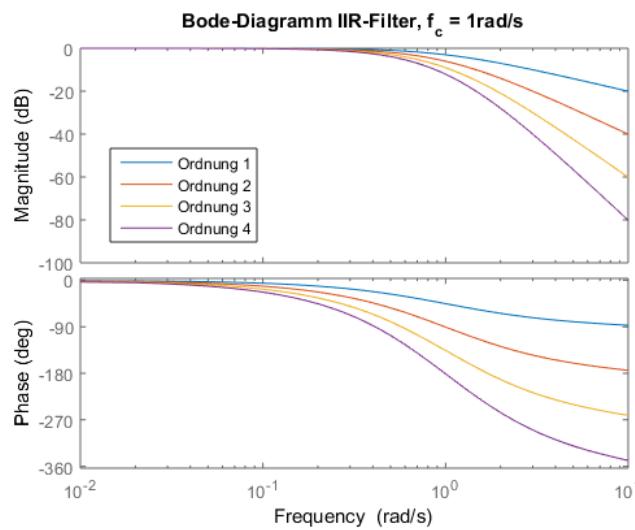


Abbildung D.8.: Bodeplot für verschiedene Filterordnungen für die Grenzfrequenz von 1 rad/s

testen Charakteristiken sind Butterworth, Chebyshev Typ 1/Typ 2 oder der Elliptische Filter (siehe Abbildung D.9). Diese unterscheiden sich in der Glattheit des Amplitudengangs vor bzw. nach der Grenzfrequenz sowie der Steigung des Amplitudenabfalls. Je nach Anwendung ist eine passende Charakteristik zu wählen. Für die Berechnung der Filterkoeffizienten hat jede Charakteristik ihre eigene Vorschrift. Im „fdatool“ von MATLAB sind diese Vorschriften implementiert, sodass eine einfach Bestimmung der Filterkoeffizienten ermöglicht wird. Es müssen lediglich Filtertyp, Ordnung, Charakteristik und die Grenzfrequenzen vorgegeben werden. Auch für die Implementierung des FIR-Filters gibt

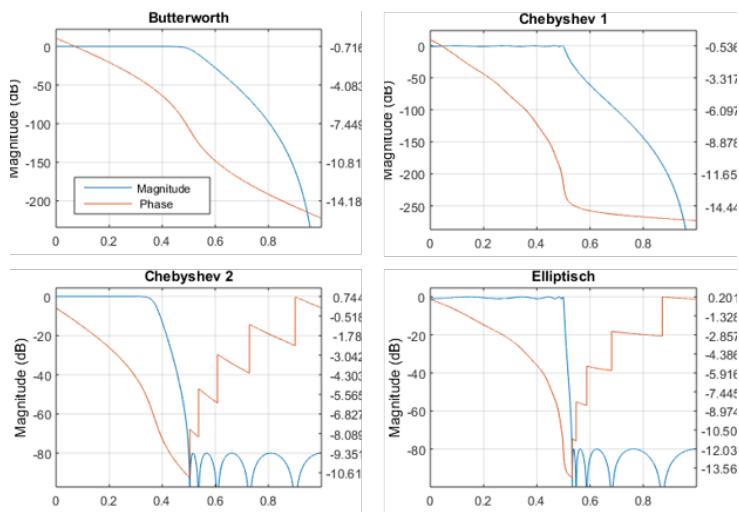


Abbildung D.9.: Filtercharakteristiken für IIR Filter 10. Ordnung

es verschiedene Varianten. Die Implementierung in der Form von Gleichung D.3 ist jedoch zu vermeiden. Durch Quantisierungsfehler im Mikrocontroller kann es bei Filtern höherer Ordnung in dieser Darstellung leicht zur Verschiebung von Nullstellen und Polen

kommen, was im schlimmsten Fall zu Instabilität führt. Besser ist die Zerlegung in ein Produkt sogenannter Second Order Sections (SOS):

$$F[z] = \prod_{i=1}^{n_{filt}} g_i \frac{b_{1i} + b_{2i}z^{-1} + b_{3i}z^{-2}}{1 + a_{2i}z^{-1} + a_{3i}z^{-2}}. \quad (\text{D.4})$$

Diese sind weniger anfällig für Quantisierungseffekte und liefern deshalb bessere Ergebnisse. Die Second Order Sections selbst, können wiederum auf verschiedenem Wege implementiert werden. Abbildung D.10 zeigt die Direct Form 1 (DF1) und die Direct Form 2 (DF2). Wie leicht erkenntlich ist, benötigt die Direct Form 2 weniger Delays, also bei der Implementierung weniger Variablen und ist deshalb die am meist genutzte Variante. Neben DF1 und DF2 gibt es noch weitere Möglichkeiten eine SOS zu implementieren (z.B. Lattice-Struktur, DF1/DF2 Transposed) auf die hier nicht näher eingegangen werden soll.

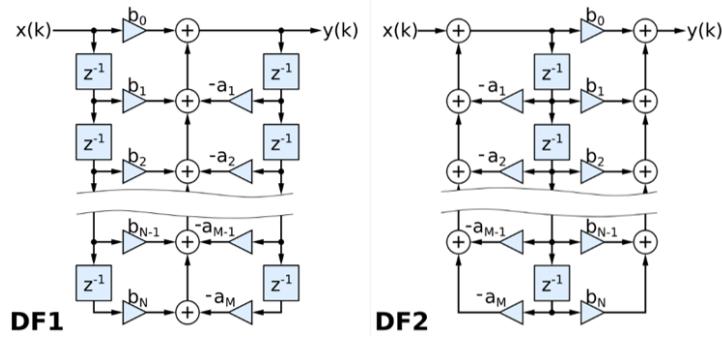


Abbildung D.10.: Direct Form 1 / 2 [18]

## D.5. Die Codevorlage

### D.5.1. Module des Quellcodes

Da die Aufgabe der Steuerung und Regelung des Kleinroboters mit seiner Vielzahl an Sensoren eine recht komplexe Programmstruktur bedingt, ist eine *Modularisierung* des Codes entscheidend, um im Programmcode die Übersicht zu behalten zu können.

Ein *Modul* besteht hierbei aus jeweils einer Header-Datei (\*.h) und einer C-Datei (\*.c) mit gleichem Namen. Die Header-Datei verweist auf sämtliche Funktionen, Typen und Variablen, die das Modul zur Verfügung stellt, ohne diese zu definieren oder zu implementieren. Wenn ein Modul verwendet werden soll, so bindet man im jeweiligen File einfach die Header-Datei des Moduls ein.

Für den Kleinroboter werden die Module, die nicht vom Studenten verändert werden müssen, im Ordner **routines** zur Verfügung gestellt. Genaue Informationen zur Verwendung finden sich im jeweiligen Quellcode.

Tabelle D.4.: Fertige Module der Software auf dem Kleinroboter (Ordner **routines**)

<i>Modulname</i>	<i>Inhalt</i>
<b>akku</b>	Funktionen zur Überprüfung der Akkuspannung
<b>sensorenI2C</b>	Funktionen zur Kommunikation mit den Drehratesensoren
<b>trajektorienVorgabe</b>	Definition der Trajektorie für den Trajektorienfolgeregel und Funktionen zum Abruf der Trajektorie
<b>uart</b>	Kommunikation über die serielle Schnittstelle

Im Laufe der Tage 3–9 implementieren Sie die Module in Tabelle D.5 selbst. Die Signatur einiger Funktionen (Typen und Anzahl der Argumente und der Rückgabewert der Funktion) und der Wert einiger Konstanten sind hierbei bereits vorgegeben. **Bevor Sie an einem Modul arbeiten, empfiehlt sich daher der Blick in die entsprechende Header-Datei!**

### D.5.2. Programmablauf

Jedes Modul besitzt eine (oder mehrere) Funktionen mit der Bezeichnung `*_init`. Diese werden direkt nach Start des Mikrocontrollers aus der `main.c` aufgerufen. Diese Funktionen beinhalten die jeweiligen Schritte zur Initialisierung der Module. Während der Initialisierung leuchtet die rote LED um dem Benutzer zu signalisieren, den Roboter in dieser Zeit nicht zu bewegen. Dies ist notwendig, da in den `init`-Funktionen zum Teil Sensoren kalibriert werden.

Ist die Kalibrierung abgeschlossen, signalisiert die gelbe LED, dass per Tastendruck die Hauptschleife gestartet werden kann. Die Grüne LED signalisiert, dass die Hauptschleife läuft. Die zweite grüne LED wird bei Aufruf der Interrupt-Routine getoggelt und zeigt somit an, dass korrekt in den Interrupt gesprungen wird.

Tabelle D.5.: Vom Studenten zu implementierende Module

Modulname	Funktionalität	Bearbeitung
main	Hauptprogramm. Hier wird mittels setzen der Variable <code>activeController</code> zwischen manueller Steuerung, Balancieren und Trajektorienfolge gewählt. <b>Vom Studenten müssen hier allenfalls zu Testzwecken Veränderungen vorgenommen werden.</b>	
motor	Funktionen zur Ansteuerung der Motoren und lesen der Radgeschwindigkeiten. Funktion zur manuellen Steuerung.	Tag 4
sensorenSPI	Kommunikation über SPI allgemein. Kommunikation mit den Quadraturdecodern. Kommunikation mit den Accelerometern.	Tag 4–5
reglerBalancieren	Komplementärfilter und PID-Kaskade des Balancierreglers	Tag 5–6
reglerTrajektorienfolge	Beobachter, Steuerung und Regelung zur Trajektorienfolge	Tag 7–8

Bei jedem Sprung in die ISR wird die Variable `sampleFlag` auf 1 gesetzt, und somit der Regler ausgeführt. Welcher Regler aktiv ist wird durch Setzen der Variablen `activeController` bestimmt.

Wenn der Code in Hauptschleife nicht innerhalb der Zykluszeit von 10 ms durchlaufen wird, wird über UART der string `missed` gesendet. Gründe für das Verletzen der Zykluszeit sind z.B. delays in der Hauptschleife oder den Unterfunktionen.

## D.6. Balancieren (Tag 4 bis 6)

### D.6.1. Motoransteuerung – Modul motor

Als erster Schritt soll die Ansteuerung der H-Brücken implementiert werden. Beginnen Sie mit der Ansteuerung des linken Rades. Benutzen Sie hierzu den `Timer1` im *Fast-PWM Modus*. Konfigurieren Sie den Timer außerdem für den *non-inverting* Betrieb. Setzen Sie den *Prescaler* so, dass sich mittels geeignetem Wert des 16bit Registers `ICR1` eine PWM Frequenz von ca. 10kHz einstellt. *Tipp:* Verwenden Sie das Makro `MAX_PWM` für die Einstellung des `ICR1` Wertes, indem Sie in der Header-Datei `motor.h` die Präprozessoranweisung

```
#define MAX_PWM <<TOP-WERT>>
```

hinzufügen. So lässt sich später im Code leicht auf den aktuell eingestellten Wert zurückgreifen. Die eben beschriebenen Anweisungen zur Initialisierung des Timers werden

sinnvollerweise in der Funktion

```
void motor_init(void)
```

untergebracht.

Wenn die Ansteuerung des linken Rades problemlos funktioniert, erweitern Sie Ihr Programm so, dass beide Motoren mittels einer Funktion

```
void motor_pwm(uint8_t sideId, float pwm_duty)
```

angesprochen werden können. Das Argument `sideId` ist hierbei ein *unsigned integer*, der die anzusprechende Seite identifiziert, z.B. 0 für links und 1 für rechts. Das Argument `pwm_duty` gibt den kommandierten PWM-duty-cycle an und hat einen Wertebereich von -1 (volle Rückwärtsfahrt) bis +1 (volle Vorwärtsfahrt).

Funktioniert die Ansteuerung beider Motoren, implementieren Sie die Funktion

```
void motor_manualCtrl(void)
```

mit der Sie den Roboter möglichst komfortabel vom PC aus fernsteuern können (z.B. mit Ziffernblock, Joystick?). Unternehmen Sie erste Fahrversuche.

**Abnahme der Motoransteuerung** Zeigen Sie Ihrem Betreuer wie Sie koordiniert Rechts- und Linkskurven und gerade Strecken sowohl vorwärts als auch rückwärts zurücklegen.

### D.6.2. Auslesen der SPI Sensoren - Modul sensorenSPI

Der SPI Bus wird zur Kommunikation mit den Quadraturdecodern und Beschleunigungssensoren verwendet. **Zur Konfiguration ist das Studium des Datenblatts des Mikrocontrollers und der Sensoren unbedingt erforderlich!**

**Kommunikation über SPI** Zunächst implementieren Sie die Funktionen mit dem Präfix `spi_*` um den Bus zu initialisieren (`spi_init`), einen Slave auszuwählen (`spi_select`) und Daten zu senden und zu empfangen (`spi_sendAndRead`). Nehmen Sie für die komfortable Auswahl der Slaves die Makros in der Header-Datei zu Hilfe.

**Auslesen der Quadraturdecoder** Verwenden Sie nun Ihre `spi_*`-Funktionen um die weiteren Funktionen

```
void qdec_writeRegister(uint8_t spiSensorId, uint8_t registerByte, uint8_t data)
void qdec_writeCommand(uint8_t spiSensorId, uint8_t command)
```

entsprechend mit Leben zu füllen. Beide Funktionen werden zum Initialisieren der Quadraturdecoder benötigt (vgl. `qdec_init`). Schreiben Sie nun den Körper von

```
int32_t qdec_getCounts(uint8_t spiSensorId)
```

um die gezählten Ticks des mittels `spiSensorId` ausgewählten Decoders zu erhalten.

**Auslesen der Beschleunigungssensoren** Schreiben Sie die Funktion

```
void acc_writeRegister(uint8_t registerByte, uint8_t data)
```

welche während der Initialisierung des Sensors (`acc_init`) verwendet wird. Füllen Sie nun den Körper von

```
void acc_getData(int16_t* accData)
```

mit den nötigen Anweisungen, um an die übergebene Adresse `accData` die gemessenen Werte zu schreiben. Hierbei ist `accData` die Adresse eines Integer-Arrays mit drei Elementen für die Messungen in  $x$ ,  $y$ , und  $z$ -Richtung im Sensorkoordinatensystem.

**Abnahme des Auslesens der Sensordaten** Geben Sie über UART zyklisch die folgenden Daten aus: Die gemessenen Ticks rechts und links sowie die Beschleunigungswerte in  $x$ ,  $y$  und  $z$ -Richtung.

### D.6.3. Identifikation der Motorkennlinie – Modul motor

Schreiben Sie die Funktion

```
void motor_getVel(float* vMessLinks, float* vMessRechts)
```

um die aktuellen Radgeschwindigkeiten (in cm/s) zu erhalten. Überprüfen Sie die korrekte Funktion (auf die Vorzeichen achten!) indem Sie mit der manuellen Fernsteuerung herumfahren und die Geschwindigkeitswerte über UART ausgeben. Identifizieren Sie nun die Motorkennlinie für beide Räder so genau wie möglich (siehe Fig. D.11). Die Kennlinie soll pro Rad lediglich zwei Parameter (Steigung  $m$  und Offset  $c$ ) besitzen und durch die Gleichung

$$D_{\text{PWM}} = \frac{u_{\text{Akku,nom}}}{u_{\text{Akku}}} (mv_{\text{soll}} + c \operatorname{sgn}(v_{\text{soll}})) \quad (\text{D.5})$$

beschrieben werden. Hierbei bezeichnet  $D_{\text{PWM}}$  den aufzuschaltenden PWM-duty-cycle,  $v_{\text{soll}}$  die gewünschte Geschwindigkeit,  $u_{\text{Akku,nom}}$  die Akkuspannung bei Aufzeichnung der Kennlinie und  $u_{\text{Akku}}$  die aktuelle Akkuspannung.

Wenn Sie die Kennlinie identifiziert haben, implementieren Sie diese im Körper der Funktion

```
void motor_setVel(float vSollLinks, float vSollRechts)
```

um den beiden Rädern die Soll-Geschwindigkeiten aufprägen zu können.

**Abnahme zur Identifikation der Motorkennlinie** Geben Sie über UART zyklisch die folgenden Daten aus: Soll- und Ist-Geschwindigkeit links und rechts.

### D.6.4. Sensordatenfusion

Um die Qualität der Kippwinkelmessung  $\alpha$  (siehe Abbildung D.12) zu verbessern, soll ein Komplementärfilter genutzt werden. Dieser fusioniert die Messwerte aus Accelerometer und Gyroskop so, dass ein besserer geschätzter Winkel  $\alpha$  entsteht.

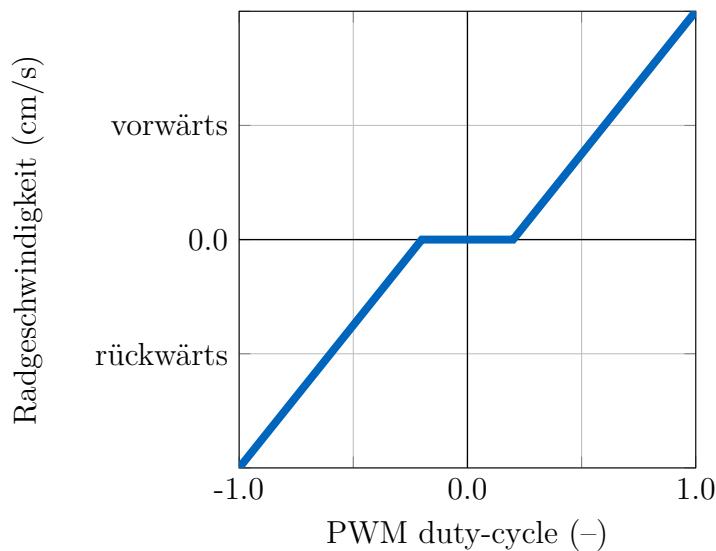


Abbildung D.11.: Motorkennlinie. Zu implementieren ist die Umkehrfunktion.

**Tiefpassfilter 2. Ordnung** Implementieren Sie dafür zunächst die Funktion

```
float reglerBalancieren_PT2 (float x)
```

welche den benötigten Tiefpassfilter 2. Ordnung für das Signal x darstellt. Achten Sie darauf, dass Sie leicht die Filterkoeffizienten zur Optimierung des Filters ändern können. Das Filter soll in der Direct Form 2 Transposed implementiert werden (siehe Abbildung D.13).

**Komplementärfilter** Füllen Sie nun die Funktion

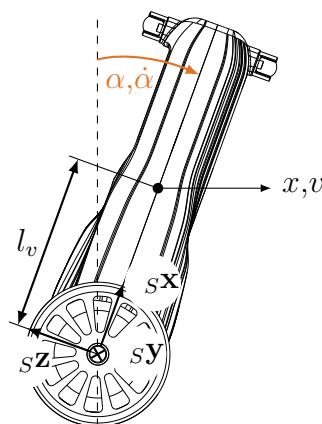


Abbildung D.12.: Kleinroboter, Zustände des Kleinroboters und Koordinatensystem der Sensormessungen

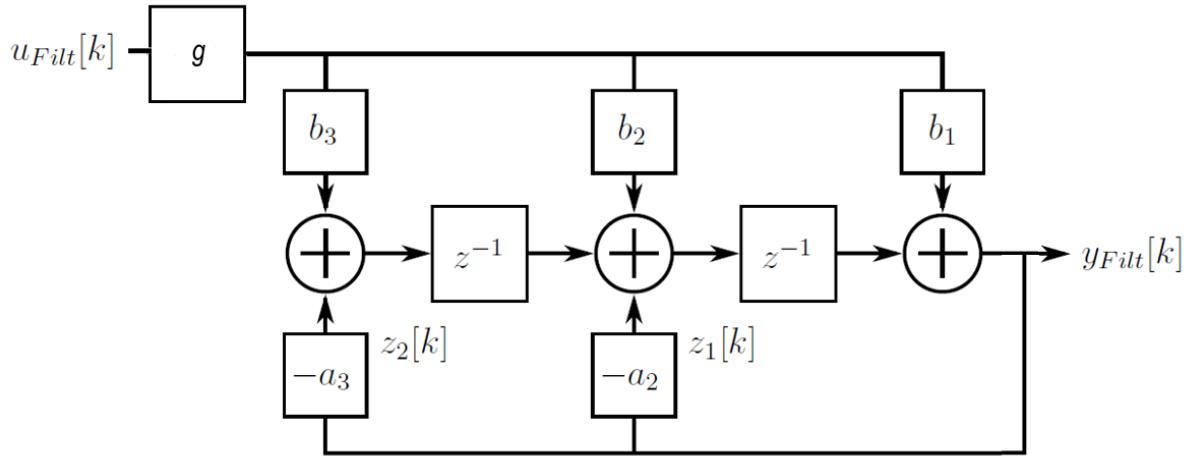


Abbildung D.13.: DF2 Transposed des Tiefpassfilters 2. Ordnung, nach [14]

```
void reglerBalancieren_komplementaerFilter(float* alphaFiltered, float* alpha_dot)
```

mit Leben. Diese soll Gyroskop- und Accelerometermesswerte auslesen und für diese jeweils einen Winkel  $\alpha$  bestimmen. Anschließend sollen diese beiden Winkelwerte mithilfe des Komplementärfilters zu einem geschätzten Winkel `alphaFiltered` fusioniert werden. Zusätzlich soll die Funktion auch noch die gemessene Winkelgeschwindigkeit `alpha_dot` ausgeben.

Für die Bestimmung der Filterkoeffizienten des Tiefpasses nutzen Sie bitte das `fdatool` von MATLAB in Kombination mit dem Befehl `filter`. Zeichnen Sie hierzu eine Messreihe von Winkelmessungen über das Accelerometer und das Gyroskop für den Kleinroboter in aufrechter Lage stehend auf. Nutzen Sie diese, um Ihr Komplementärfilter vorab in Matlab zu entwerfen. Ziel ist es den glatten Verlauf des Gyroskops *ohne* Drift zu erhalten (siehe  $\alpha_{Komp}$  in Abbildung D.14).

**Initialisierung der Filter** Die Anweisungen zur Initialisierung der Filter sollen in der Funktion

```
void reglerBalancieren_init(void)
```

untergebracht sein. So werden zu jedem Programmstart — und wenn gewünscht auch später — die Filter neu initialisiert.

**Abnahme des Komplementärfilters** Zur Abnahme Ihres Komplementärfilters durch den Betreuer, zeichnen Sie bitte den Winkel  $\alpha_{Acc}$  berechnet aus den Accelerometerwerten,  $\alpha_{Gyro}$  berechnet aus den Gyroskopwerten und das mit dem Komplementärfilter gefilterte  $\alpha_{Komp}$  auf, während Sie den Kippwinkel des Roboters beliebig verändern. Visualisieren Sie diese Messwerte anschließend mit Hilfe von Matlab und zeigen Sie den so entstandenen Plot Ihrem Betreuer.

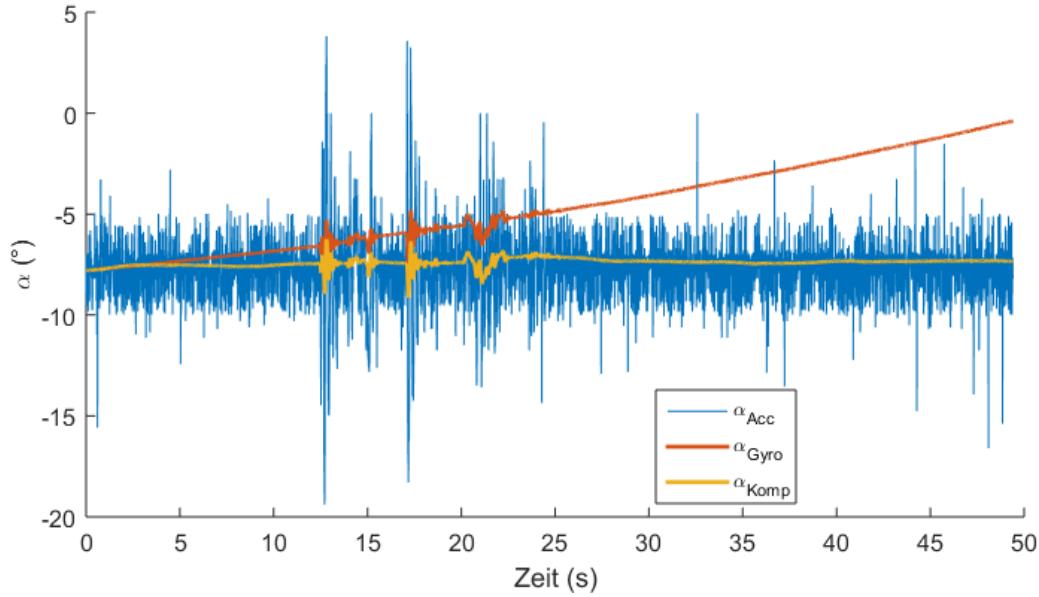


Abbildung D.14.: Winkelmessungen und mit Komplementärfilter gefiltertes  $\alpha$

### D.6.5. PID-Kaskade zum Balancieren – Modul reglerBalancieren

Sie sind dank der vorherigen Aufgaben bereits in der Lage, alle relevanten Zustände des Kleinroboters ( $\alpha$ ,  $\dot{\alpha}$ ,  $v_{\text{Rad}}$ ) zu messen beziehungsweise mittels des Komplementärfilters zu schätzen. Nun gilt es eine geeignete Rückführung dieser Messdaten zu entwerfen, um die instabile aufrechte Ruhelage des Roboters zu stabilisieren.

**Allgemeines zum Balancierregler** Es soll eine PID-Kaskade mit der (idealen, nicht realisierbaren) Struktur aus Abbildung D.15 erstellt werden. Beachten Sie, dass im Folgenden die Drehgeschwindigkeit um die  $sx$  und  $sy$ -Achsen zu null angenommen wird, was bei gleichen Motorkommandos, idealem Modellverhalten und den entsprechenden Anfangsbedingungen auch tatsächlich der Fall ist.

Die innere Kaskade soll den Winkelfehler  $\tilde{\alpha} = \alpha_{\text{soll}} - \alpha$  ausregeln und verwendet als Stellgröße die Soll-Geschwindigkeit der Räder, welche von der bereits implementierten Motorkennlinie in eine Motorspannung  $u_{\text{Mot}}$  umgesetzt wird.

Die äußere Kaskade regelt den Geschwindigkeitsfehler  $\tilde{v} = v_{\text{soll}} - v$  zu null. Wie der Abbildung D.12 zu entnehmen beschreibt  $l_v$  die Entfernung zwischen Radachse und dem Punkt auf dem Roboter, dessen Geschwindigkeit  $v$  geregelt werden soll. Sie kann mittels der Kinematik wie in Abbildung D.15 dargestellt aus den Messdaten ermittelt werden.

Beachten Sie bei der Implementierung der Kaskade, dass das Blockschaltbild D.15 so nicht realisierbar ist, da es Differenzierglieder enthält. Durch geeignete Umformung des Blockschaltbilds lassen sich einige D-Glieder jedoch entfernen. Wo dies nicht möglich ist, soll wie in Kapitel 9 beschrieben ein approximativer Differenzierer realisiert werden.

Beginnen Sie das Tuning des Reglers mit den Reglerverstärkungen in Tabelle D.6

**Schritt für Schritt** Schreiben Sie zunächst die Funktion

Tabelle D.6.: Initialwerte der Reglerverstärkungen (Einheiten: cm, s, °)

<i>Winkelregler</i>	<i>Geschwindigkeitsregler</i>
$K_{\alpha,P} = 3,8843$	$K_{v,P} = 0,3531$
$K_{\alpha,I} = 0,0144$	$K_{v,I} = 0,6757$
$K_{\alpha,D} = 0,0021$	$K_{v,D} = 0,0006$
$l_v = 5,0 \text{ cm}$	

```
void reglerBalancieren_winkelRegler (float alpha, float
                                     alpha_dot, float alpha_soll)
```

und überprüfen Sie, ob diese korrekt funktioniert. Implementieren Sie anschließend die äußere Kaskade in der Funktion

```
void reglerBalancieren_geschwindigkeitsRegler (float vRad
                                               , float alpha, float alpha_dot)
```

Versuchen Sie dafür zunächst die vorgegebene Schaltlogik zu verstehen, ehe Sie den Code verändern.

Wie zuvor soll `reglerBalancieren_init` genutzt werden um den Regler zu initialisieren und `reglerBalancieren_regelung` als Wrapper für alle nötigen Funktionsaufrufe zur Ausführung der Reglerkaskade.

**Abnahme des Balancierreglers** Lassen Sie den Roboter balancieren und geben Sie über UART zyklisch die folgenden Daten aus: Gemessene Geschwindigkeit  $v$ , im Abstand  $l_v$  über der Radachse, gemessener Winkel  $\alpha$ .

## D.7. Trajektorienfolge (Tag 7 bis 8)

An den Tagen 7 bis 9 wird ein weiterer Regler für den Kleinroboter implementiert, der es erlaubt liegend eine vorgegebene Trajektorie abzufahren. Besagte Trajektorie (eine Acht) wird vom Modul `trajektorienVorgabe` bereitgestellt.

### D.7.1. Beobachter und Steuerung – Modul `reglerTrajektorienfolge`

Erstellen Sie einen Beobachter, der regelmäßig aufgerufen wird und anhand der gemessenen Radbewegungen die Position und Winkellage des Roboters berechnet:

```
void reglerTrajektorienfolge_beobachter (float* u1_B,
                                         float* u2_B, float* x1_B, float* x2_B, float* x3_B)
```

Implementieren Sie dann eine flachheitsbasierte Steuerung, die die vorgegebenen kartesischen Geschwindigkeiten und Beschleunigungen in Radgeschwindigkeiten übersetzt.

```
void reglerTrajektorienfolge_steuering (void);
```

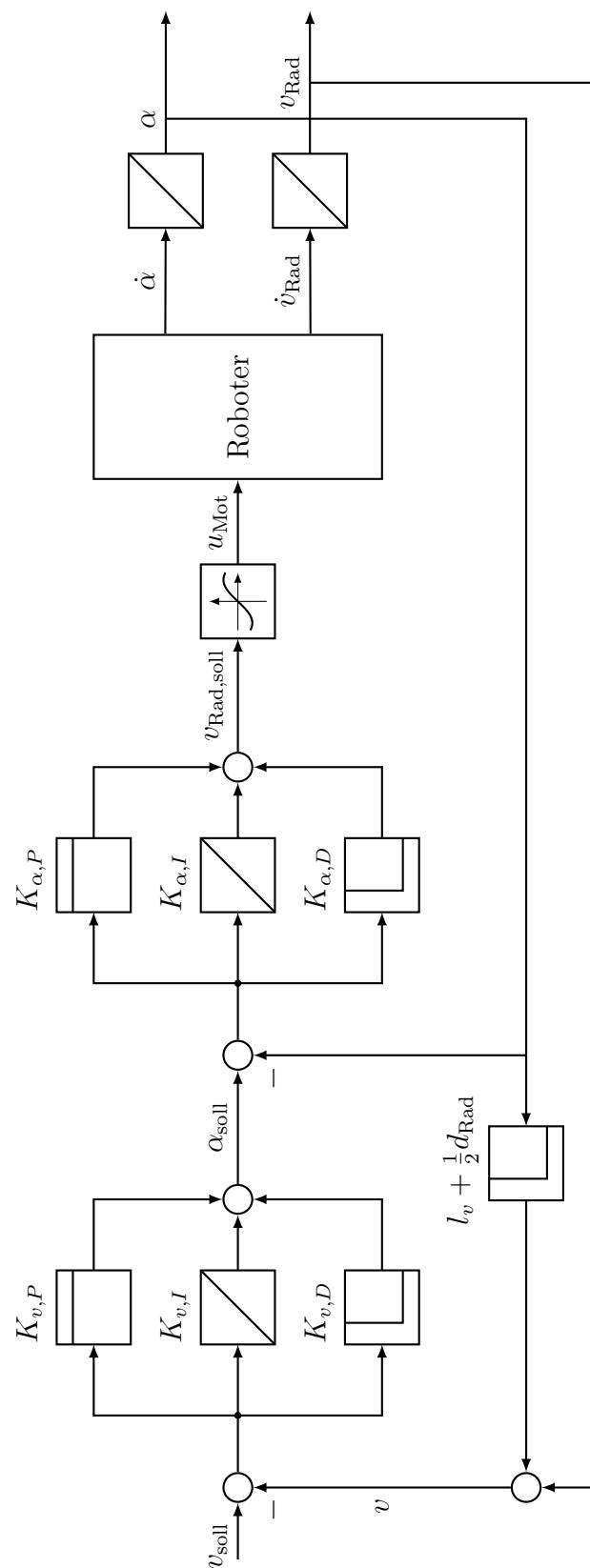


Abbildung D.15.: Balancierregler

Lassen Sie den Roboter damit die Solltrajektorie abfahren und tunen Sie ggf. die Parameter (Geometrie, Kennlinie). Übertragen Sie die gemessenen Positionen zum PC, plotten Sie diese (z.B. in MATLAB) und vergleichen Sie die Messung mit der Realität.

**Abnahme des Beobachters und der Steuerung** Lassen Sie den Roboter mehrfach die Acht gesteuert abfahren. Geben Sie alle nötigen Daten zyklisch über UART aus und plotten diese in geeigneter Weise in MATLAB. Über den MATLAB-Befehl `quiver` lässt sich neben der Position auch die Geschwindigkeit grafisch darstellen.

### D.7.2. Flachheitsbasierte Regelung – Modul `reglerTrajektorienfolge`

Erweitern Sie die Steuerung um eine Rückführung und programmieren Sie so mithilfe dynamischer Linearisierung einen flachheitsbasierten Regler.

```
void reglerTrajektorienfolge_regelung (float u1_B, float
u2_B, float x1_B, float x2_B, float x3_B);
```

Platzieren Sie die Pole geeignet, um optimales Folgeverhalten zu erzielen.

**Abnahme der flachheitsbasierten Regelung** Evaluieren Sie die Reglerperformance in MATLAB durch Vergleich von Ist und Soll-Trajektorie.

### D.7.3. Vergleich und Wettbewerb

Ihr Roboter wird mit einem Stift versehen und tritt im Wettbewerb gegen seine Kameraden an. Wer kann die Trajektorie zweimal oder öfter abfahren, trifft die Sollkurve dabei am besten und erzeugt die kleinste Abweichung zwischen den Runden?

## D.8. Interrupt-Vektoren im ATmega32

Vector No.	Program Address <sup>(2)</sup>	Source	Interrupt Definition
1	\$000 <sup>(1)</sup>	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	\$002	INT0	External Interrupt Request 0
3	\$004	INT1	External Interrupt Request 1
4	\$006	INT2	External Interrupt Request 2
5	\$008	TIMER2 COMP	Timer/Counter2 Compare Match
6	\$00A	TIMER2 OVF	Timer/Counter2 Overflow
7	\$00C	TIMER1 CAPT	Timer/Counter1 Capture Event
8	\$00E	TIMER1 COMPA	Timer/Counter1 Compare Match A
9	\$010	TIMER1 COMPB	Timer/Counter1 Compare Match B
10	\$012	TIMER1 OVF	Timer/Counter1 Overflow
11	\$014	TIMER0 COMP	Timer/Counter0 Compare Match
12	\$016	TIMER0 OVF	Timer/Counter0 Overflow
13	\$018	SPI, STC	Serial Transfer Complete
14	\$01A	USART, RXC	USART, Rx Complete
15	\$01C	USART, UDRE	USART Data Register Empty
16	\$01E	USART, TXC	USART, Tx Complete
17	\$020	ADC	ADC Conversion Complete
18	\$022	EE_RDY	EEPROM Ready
19	\$024	ANA_COMP	Analog Comparator
20	\$026	TWI	Two-wire Serial Interface
21	\$028	SPM_RDY	Store Program Memory Ready

Notes:

- When the BOOTRST fuse is programmed, the device will jump to the Boot Loader address on reset, see "Boot Loader Support – Read-While-Write Self-Programming" on page 244.
- When the IVSEL bit in GICR is set, interrupt vectors will be moved to the start of the Flash section. The address of each Interrupt Vector will then be the address in this table + to the start address of the Boot Flash section.

Abbildung D.16.: Interrupt-Vektoren des ATmega32 [4, S.44].

# Literaturverzeichnis

- [1] Infineon AG. Infineon pma 7110, May 2013. URL [http://www.infineon.com/export/sites/default/media/press/Image/press\\_photo/PMA7110.jpg](http://www.infineon.com/export/sites/default/media/press/Image/press_photo/PMA7110.jpg).
- [2] *AT89LP213 Data Sheet*. Atmel, 11 2010. URL <http://www.atmel.com/Images/doc3538.pdf>.
- [3] *8-bit AVR Instruction Set*. Atmel, 07 2010. URL <http://www.atmel.com/Images/doc0856.pdf>.
- [4] *ATMega8(L) Data Sheet*. Atmel, 02 2011. URL [www.atmel.com/images/doc2503.pdf](http://www.atmel.com/images/doc2503.pdf).
- [5] *ATMega8(L) Data Sheet*. Atmel, 02 2013. URL [http://www.atmel.com/Images/Atmel-2486-8-bit-AVR-microcontroller-ATmega8\\_L\\_datasheet.pdf](http://www.atmel.com/Images/Atmel-2486-8-bit-AVR-microcontroller-ATmega8_L_datasheet.pdf).
- [6] U. Brinkschulte and T. Ungerer. *Mikrocontroller und Mikroprozessoren*. Springer, 3. edition, 2010.
- [7] J. B. Burl. *Linear Optimal Control -  $\mathcal{H}_2$  and  $\mathcal{H}_\infty$  Methods*. Addison Wesley Longman, Inc., Menlo Park, CA, 1999.
- [8] M. Fliess, J. Lévine, P. Martin, and P. Rouchon. *Nonlinear Control System Design*. Pergamon Press, 1992.
- [9] O. Föllinger. *Optimale Regelung und Steuerung*. Oldenbourg, München, dritte verbesserte Auflage, 1994.
- [10] *Quadrature Decoder*. fpga4fun.com, 05 2013. URL <http://www.fpga4fun.com/QuadratureEncoder>.
- [11] *IEEE 754-2008: Standard for Floating-Point Arithmetic*. IEEE Standards Association, 2008.
- [12] *Bright Red L-934HD*. Kingbright, 2013.
- [13] *L-934LI HIGH EFFICIENCY RED*. Kingbright, 2013.
- [14] C. Leonhardt. Konstruktion und Regelung eines multifunktionalen Kleinroboters. Master's thesis, Semesterarbeit, Lehrstuhl für Regelungstechnik, TUM, 2014.
- [15] B. Lohmann and M. Buttelmann. Flache Systeme: Einführung, Anwendung und Realisierung einer Regelung für mobile Plattformen.
- [16] *LF7366R 32-BIT QUADRATURE COUNTER WITH SERIAL INTERFACE*. LSI Computer Systems, 2007.
- [17] J. Lunze. *Regelungstechnik 2 - Mehrgrößensysteme und Digitale Regelung*. Springer, Berlin, 2006.

- [18] Mswiki3141. Direkt-form 1/2 iir, creative-commons-lizenz „namensnennung – weitergabe unter gleichen bedingungen 4.0 international“, 12 2015. URL [https://de.wikipedia.org/wiki/Filter\\_mit\\_unendlicher\\_Impulsantwort](https://de.wikipedia.org/wiki/Filter_mit_unendlicher_Impulsantwort).
- [19] *UM10204 I2C-bus specification and user manual*. NXP Semiconductor, 09 2012.
- [20] M. Papageorgiou. *Optimalierung - Statische, Dynamische und Stochastische Verfahren für die Anwendung*. Oldenbourg, Wien, 1991.
- [21] R. Rothfuß, J. Rudolph, and M. Zeitz. Flachheit: Ein neuer Zugang zur Steuerung und Regelung nichtlinearer Systeme. *Automatisierungstechnik*, 1997.
- [22] G. Schmitt. *Mikrocomputertechnik mit Controllern der Atmel AVR-RISC-Familie*. Oldenbourg, 4. edition, 2008.
- [23] M. Stampfl. Trajektorienfolgeregelung mit Asuro. Master's thesis, Semesterarbeit, Lehrstuhl für Regelungstechnik, TUM, 2011.
- [24] K. Wüst. *Mikroprozessortechnik*. Vieweg + Teubner, 4. edition, 2011.