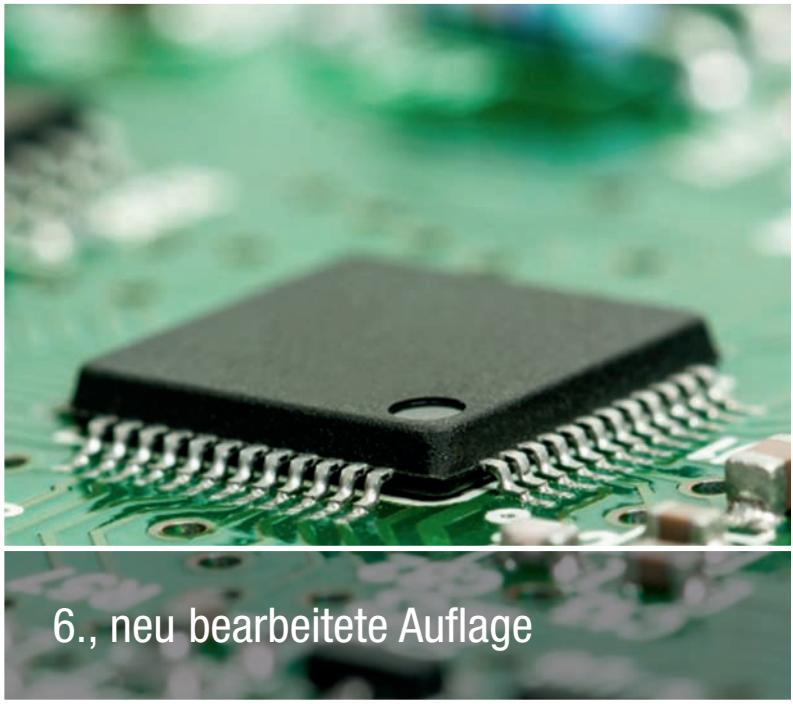
Bernd-Dieter Schaaf
Stephan Böcker

Mikrocomputer-technik

Aktuelle Controller 8051: Funktionsweise, äußere Beschaltung und Programmierung



6., neu bearbeitete Auflage

HANSER

Schaaf/Böcker

Mikrocomputertechnik

Lernbücher der Technik

herausgegeben von Dipl.-Gewerbelehrer Manfred Mettke, Oberstudiendirektor a. D.

Bisher liegen vor:

Bauckholt, Grundlagen und Bauelemente der Elektrotechnik, 6. Auflage
Felderhoff/Freyer, Elektrische und elektronische Messtechnik, 8. Auflage
Felderhoff/Busch, Leistungselektronik, 4. Auflage
Fischer/Hofmann/Spindler, Werkstoffe in der Elektrotechnik, 6. Auflage
Freyer, Nachrichten-Übertragungstechnik, 6. Auflage
Heiderich/Meyer, Probleme lösen mit C/C++, 1. Auflage
Knies/Schierack, Elektrische Anlagentechnik, 6. Auflage
Schaaf, Mikrocomputertechnik, 6. Auflage
Seidel/Hahn, Werkstofftechnik, 9. Auflage

Bernd-Dieter Schaaf, Stephan Böcker

Mikrocomputertechnik

Aktuelle Controller 8051: Funktionsweise, äußere Beschaltung und Programmierung

unter Mitarbeit von Peter Wissemann

6., neu bearbeitete Auflage

Mit 256 Bildern, 44 Tabellen, 31 Übungen und Beispielen

Fachbuchverlag Leipzig
im Carl Hanser Verlag

OStR Dipl.-Ing. Bernd-Dieter Schaaf
StR Dipl.-Ing. Stephan Böcker
Heinz-Nixdorf-Berufskolleg Essen
OStR Dipl.-Ing. Peter Wissemann
Heinrich-Hertz-Berufskolleg Düsseldorf

Bibliografische Information der Deutschen Nationalbibliothek

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

ISBN: 978-3-446-43078-5
E-Book-ISBN: 978-3-446-43348-9

Internet: http://www.hanser-fachbuch.de

Lektorat: Dr. Martin Feuchte
Herstellung: Dipl.-Ing. Franziska Kaufmann
Satz: Satzherstellung Dr. Steffen Naake, Brand-Erbisdorf
Coverconcept: Marc Müller-Bremer, www.rebranding.de, München
Coverrealisierung: Stephan Rönigk
Druck und Bindung: Friedrich Pustet KG, Regensburg
Printed in Germany

Vorwort des Herausgebers

Was können Sie mit diesem Buch lernen?

Wenn Sie dieses Lernbuch durcharbeiten, dann erwerben Sie umfassende Erkenntnisse und Qualifikationen, die Sie zur **Handlungsfähigkeit** in der **Mikrocomputertechnik mit Mikrocontrollern** führen.

Der Umfang dessen, was wir Ihnen anbieten, orientiert sich an
- den Studienplänen der Fachhochschulen für Technik,
- den Lehrplänen der Fachschulen für Technik in den Bundesländern,
- den Anforderungen der beruflichen Praxis,
- dem Stand der Technik.

Sie werden systematisch und schrittweise mit der spezifischen Hard- und Software eines Mikrocomputersystems mit Mikrocontrollern vertraut gemacht. Sie können Programme konstruieren und Anwenderprogramme der Automatisierungstechnik nutzen.

Dabei gehen Sie folgenden Fragen nach:
- Welche Struktur der Baugruppen und ihrer Programmierung kennzeichnet einen Mikrocontroller?
- Wie kann man ein Entwicklungssystem konstruieren?
- Welche Erweiterungen lässt das Mikrocontrollersystem zu?
- Wie programmiert man das System mit der Hochsprache „C“?

Wer kann mit diesem Buch lernen?

Jeder, der
- sich weiterbilden möchte,
- die Grundlagen der Datenverarbeitung kennt,
- Kenntnisse in den Grundlagen der Elektrotechnik besitzt.

Das können sein:
- Studenten an Fachhochschulen und Berufsakademien,
- Studenten an Fachschulen für Technik,
- Schüler an beruflichen Gymnasien und Berufsoberschulen,
- Schüler in der Assistentenausbildung,
- Facharbeiter, Gesellen und Meister während und nach der Ausbildung,
- Umschüler und Rehabilitanden,
- Teilnehmer an Fort- und Weiterbildungskursen,
- Autodidakten,

vor allem in den Bereichen:
- Elektrische Energietechnik, Prozessautomatisierung, Prozessleittechnik
- Informations- und Kommunikationstechnik.

Wie können Sie mit diesem Buch lernen?

Ganz gleich, ob Sie mit diesem Buch in Hochschule, Schule, Betrieb, Lehrgang oder zu Hause im „stillen Kämmerlein“ lernen, es wird Ihnen Freude machen.

Warum?

Ganz einfach, weil Ihnen hier ein Buch empfohlen wird, das in seiner Gestaltung die **Grundgesetze des menschlichen Lernens beachtet**.

– Ein Lernbuch also! –

Sie setzen sich kapitelweise mit den Lerninhalten, Lehrstoffen auseinander, schrittweise dargestellt, in überschaubaren Lernsequenzen. Wo es möglich ist, wird der Lehrstoff ausführlich beschrieben auf der linken Spalte der Buchseite und umgesetzt in die technisch-wissenschaftliche Darstellung auf der rechten Spalte der Buchseite. Die weitgehende Zuordnung der behandelten Lerninhalte in den beiden Spalten erleichtert das Lernen wesentlich, Umblättern ist zum Beispiel in der Regel nicht nötig. An Beispielen konkretisiert und veranschaulicht der Autor die neuen Lerninhalte.

– Ein unterrichtsbegleitendes Lehrbuch mit Beispielen! –

Jetzt können und sollten Sie sofort die Übungsaufgaben lösen, um das Neugelernte zu festigen, zu vertiefen und mit bisher Gelerntem zu verknüpfen. Die wesentlichen Schritte der Lösung und das Ergebnis der jeweiligen Übung sind am Ende des Buches vom Autor für Sie aufgeschrieben.

– Also auch ein Arbeitsbuch mit Übungen und Lösungen! –

Für das Aufsuchen entsprechender Kapitel steht Ihnen das Inhaltsverzeichnis am Anfang des Buches zur Verfügung. Für die Suche bestimmter Begriffe hat der Autor für Sie am Ende des Buches das Sachwortregister angelegt.

– Selbstverständlich mit Inhaltsverzeichnis und Sachwortregister! –

Sicherlich werden Sie durch die intensive Arbeit mit dem Buch Ihre „Bemerkungen zur Sache“ unterbringen wollen und die Lösungen der Übungen an den jeweiligen Stellen zuordnen, um so ein individuelles Arbeitsmittel an der Hand zu haben.

– Am Ende ist Ihr Buch entstanden! –

Möglich wurde dieses Lernbuch für Sie durch die Bereitschaft des Autors und die intensive Unterstützung des Verlages mit seinen Mitarbeitern. Ihnen sollten wir herzlich danken.

Beim Lernen wünsche ich Ihnen nun viel Freude und Erfolg.

Ihr Herausgeber

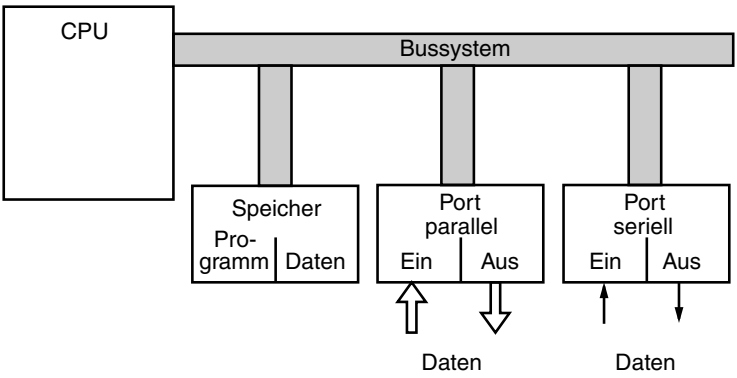
Manfred Mettke

Inhaltsverzeichnis

1 Der Mikrocomputer

In diesem Kapitel werden der grundsätzliche Aufbau und die Arbeitsweise eines Mikrocomputers beschrieben. Damit soll die Grundlage für das Verständnis der darauf folgenden Inhalte gelegt werden.

1.1 Der Aufbau eines Mikrocomputers



Ein Mikrocomputer besteht aus einer zentralen Prozess-Einheit CPU, die über ein Bussystem mit dem Speicher und den Eingabe- und Ausgabeeinheiten verbunden ist.

Die CPU wird über das Programm gesteuert. Sie verarbeitet die Daten, die sie aus dem Speicher oder den Eingabe-Ports liest. Die Ergebnisse schreibt sie wieder in den Speicher oder in die Ausgabe-Ports. Der Transport der Daten erfolgt über eine Anzahl paralleler Leitungen, die als Datenbus bezeichnet werden. Der Datenbus ist ein Teil des Bussystems.

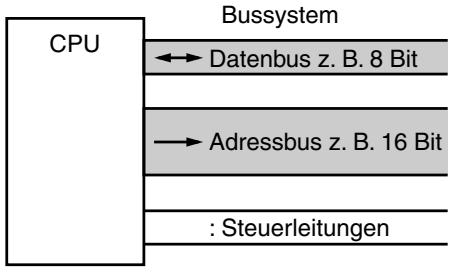
Das Programm und die Daten liegen im Speicher in bestimmten Adressen. Auch die Ports werden durch Adressen voneinander unterschieden. Die CPU muss die Adressen ausgeben, aus denen sie das Programm oder die Daten lesen will oder in die sie Daten schreiben will. Die Adresse wird von der CPU als Dualzahl über den Adressbus ausgegeben. Der Adressbus ist ein weiterer Teil des Bussystems.

Da z. B. beim Speicher Daten in die gleiche Adresse geschrieben oder aus ihr gelesen werden, ist ihm mitzuteilen, ob es sich um einen Lese- oder Schreibvorgang handelt. Das gleiche gilt für die Ein- und Ausgabeports, falls sie gleiche Adressen haben. Zur Veranlassung dieser und ähnlicher Schaltvorgänge hat die CPU noch einige Steuerleitungen. Sie werden unter der Bezeich-

nung Steuerbus zusammengefasst. Der Steuerbus ist der dritte Teil des Bussystems. Während sich die Leitungen von Daten- und Adressbus immer in ihrer Gesamtheit betrachten lassen, da sie zum gleichen Zeitpunkt in einen neuen Zustand übergehen, muss beim Steuerbus jede Leitung im Einzelnen betrachtet werden. Die Steuerleitungen geben die Zeitpunkte vor, zu denen die Schaltvorgänge im System stattfinden.

Insgesamt besteht das Bussystem, über das die CPU mit den angeschlossenen Baugruppen korrespondiert, also aus dem Datenbus, dem Adressbus und dem Steuerbus.

An das Bussystem sind die Baugruppen angeschlossen, die die CPU für ihre Arbeit benötigt.

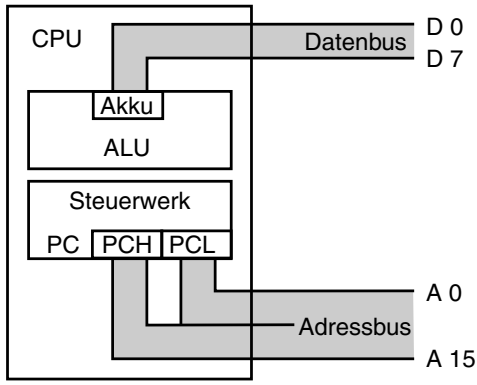


Die CPU-Baugruppe

Das zentrale Register in der CPU ist für den Programmierer der Akkumulator. Der Akku ist das Eingabe- und Ausgaberegister der Arithmetik-Logik-Einheit (Arithmetical Logical Unit) ALU. Zu Beginn einer arithmetischen oder logischen Operation steht im Akku einer der beiden Operanden. Am Ende der Operation steht darin das Ergebnis.

Der Operand muss durch Befehle des Programms vor der Operation in den Akku transportiert und danach wieder in den Datenspeicher zurückgebracht werden. Damit ist der Akku auch die Übergabestation für die Daten, die in die CPU hinein oder heraus transportiert werden.

Die CPU gibt über den Adressbus die Adressen der Speicherstellen oder der Ports aus, deren Inhalte sie lesen oder überschreiben will. Diese Adressen werden in der CPU in einem Doppelregister, dem Programmzähler (Programm Counter) PC vorbereitet. Im PC steht immer die Adresse, die als nächstes ausgegeben wird.



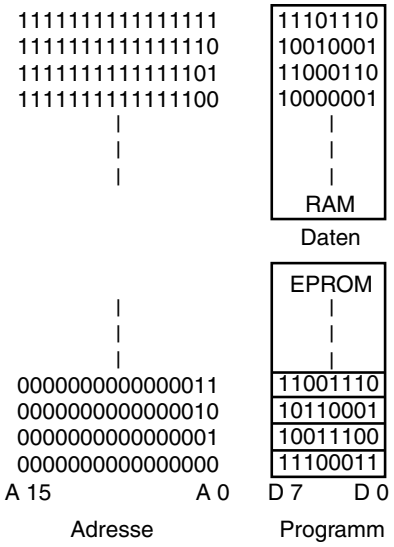
Der Speicher

Im Speicher stehen das Programm und die zu verarbeitenden Daten. Die Daten können entweder konstante oder variable Werte sein. Die Variablen werden über die Ports eingelesen oder ausgegeben.

Das Programm beinhaltet die Arbeitsanweisungen, die die CPU der Reihe nach liest und ausführt.

Die CPU beginnt die Programmbearbeitung nach einem Reset ab der Adresse 0000h. Ab dieser Adresse müsssen die Befehle gespeichert sein. Als Speicherbaustein für das Anfangsprogramm lässt sich nur ein EPROM verwenden, damit nach dem Einschalten der Stromversorgung mit anschließendem Reset sofort das Programm vorhanden ist.

Die variablen Daten lassen sich nur in einem RAM-Baustein speichern.

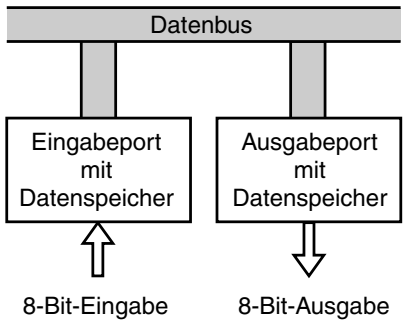


Die parallelen Ports

Über die parallelen Ports werden die Daten zu je acht Bit eingelesen oder ausgegeben.

Die CPU schreibt die Ausgabedaten über den 8-Bit-Datenbus in die Ausgabeports. Darin werden die Daten bis zum nächsten Überschreiben gespeichert.

Die an den Ports anliegenden Eingabedaten werden von der CPU über den Datenbus gelesen.

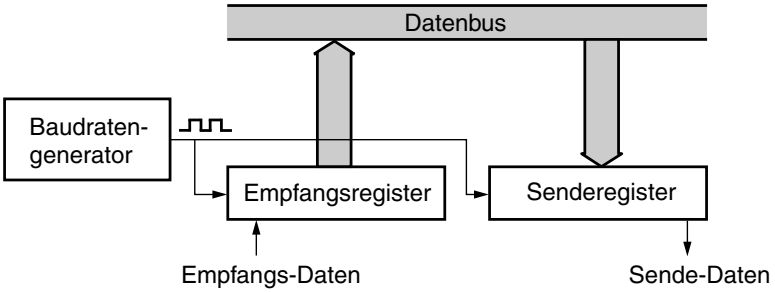


Die seriellen Ports

Serielle Ports senden oder empfangen die Daten nacheinander, Bit für Bit. Deshalb wird für das Senden oder das Empfangen der Daten nur je eine Leitung benötigt. Dazu kommt bei einem Minimalausbau nur noch das Bezugspotenzial 0 V. Der serielle Datenstrom wird eingerahmt von einem Startbit zu Beginn und einem Stopbit am Ende der Übertragung. In der Regel wird ein Byte seriell übertragen. Die Übertragung erfolgt mit einer bestimmten Geschwindigkeit, der Baudrate. Die Baudrate gibt an, wie viele Bits pro Sekunde übertragen werden. Bei einer Baudrate von z. B. 9600 werden 9600 Bits pro Sekunde übertragen. Die Übertragung wird getaktet von einem Baudratengenerator.

Für die CPU stellt sich der serielle Port genauso dar wie der parallele Port. Bei der Datenausgabe wird die Umwandlung in den seriellen Bitstrom einschließlich Start- und Stopbit in der seriellen Schnittstelle selbst vorgenommen.

Die entsprechende Serien-Parallel-Wandlung erfolgt auch bei den Empfangsdaten. Die interne Portschaltung stellt der CPU die Daten parallel zur Verfügung.

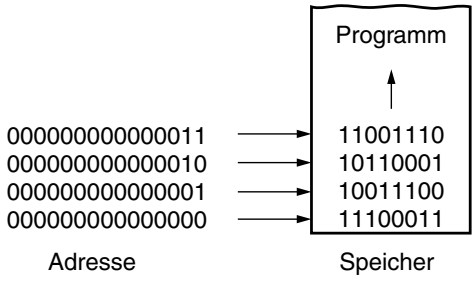


1.2 Die Arbeitsweise eines Mikrocomputers

Die CPU führt die Befehle aus, die ihr vom Programm vorgegeben werden. Dabei liest sie erst den Befehl, um ihn anschließend auszuführen. Auf diese Weise wird das Programm Befehl für Befehl abgearbeitet.

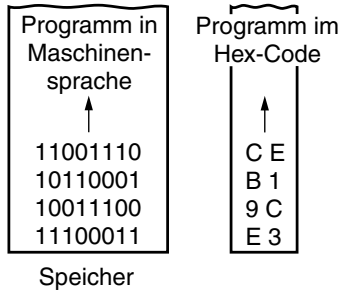
Das Programm

Das Programm ist eine Folge von Befehlen oder Instruktionen. Es liegt in aufeinander folgenden Adressen im Speicher. Jede Adresse markiert einen Speicherplatz mit acht Bit bzw. einem Byte Inhalt. Die Adressen sind z. B. 16-Bit-Dualzahlen.



Die Instruktionen des Programms sind in Bytes aufgeteilt. Jede Operation, die die CPU ausführen soll, wird durch ein bestimmtes Bitmuster in einem Byte dargestellt. Das so gespeicherte Programm, welches die CPU lesen und verstehen kann, wird Maschinenprogramm genannt.

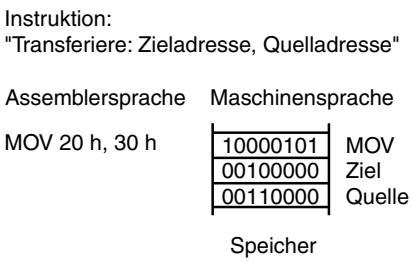
Fasst man die Bitmuster zur leichteren Lesbarkeit zu je vier Bit zusammen, lassen sie sich als Hex-Zahlen darstellen.



Ein Programm in Maschinensprache lässt sich, selbst wenn es im Hexcode dargestellt wird, kaum lesen oder schreiben. Deshalb stellt man die Operationen symbolisch durch Abkürzungen dar.

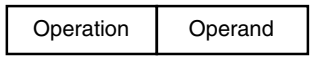
Diese Abkürzungen werden Mnemonics genannt. Ein so geschriebenes Programm ist ein Assemblerprogramm; die Programmiersprache heißt Assembler.

Ein in Assemblersprache geschriebenes Programm wird von einem Übersetzerprogramm, Makro-Assembler genannt, in Maschinensprache übersetzt und dann in den Speicher des Mikrocomputers geladen.



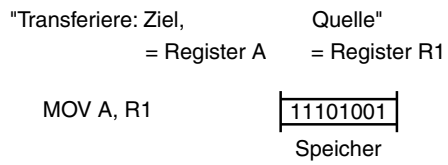
Aufbau einer Instruktion

Eine Instruktion besteht prinzipiell aus der Operation und dem Operanden. Es gibt einfache und „mächtige“ Befehle. Entsprechend kurz oder lang sind die Instruktionen oder Befehle. Zur Unterbringung im Speicher müssen sie byteweise aufgeteilt werden. Einfache Befehle sind nur ein Byte lang, mächtige Befehle können drei Byte lang sein.



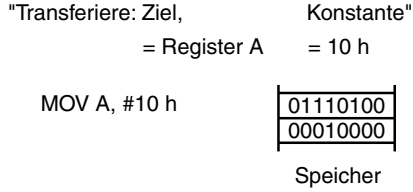
Bei 1-Byte-Befehlen sind Operation und Operand in acht Bit zusammengefasst. Dabei befindet sich der Operand in der Regel in einem Register in der CPU.

1-Byte-Befehl:



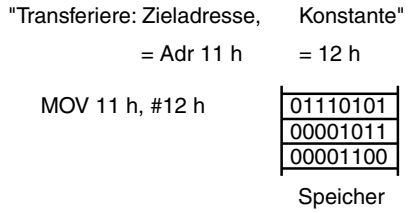
Bei 2-Byte-Befehlen steht die Operation im ersten und der Operand im zweiten Byte.

2-Byte-Befehl:



3-Byte-Befehle enthalten die Operation und in den nächsten beiden Bytes den Operanden mit Adressen oder Konstanten.

3-Byte-Befehl:



1.3 Programmbearbeitung durch die CPU

Die CPU beginnt ihre Arbeit nach einem Reset bei der Adresse 0000h. An dieser Adresse erwartet sie den Operationscode der ersten Instruktion des Programms.

Das erste Byte eines Befehls enthält immer den Operationscode.

Der Operationscode sagt dem Controller, welche Operationen er ausführen soll und wie viele Bytes der Befehl lang ist.

Operationscode:
1. welche Operation
2. Länge des Befehls

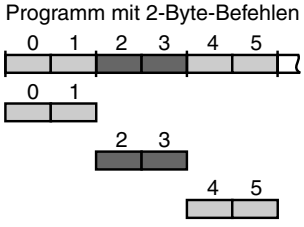
Ist der gesamte Befehl aus dem Speicher gelesen, führt der Controller ihn aus.

Das Lesen der Befehle aus dem Programmspeicher erfolgt immer nach dem gleichen Schema.

Die CPU liest während eines Zyklus immer zweimal aus dem Programmspeicher. Zu Beginn des Befehls liest sie stets aus zwei aufeinander folgenden Adressen. Das erste Byte des Befehls beinhaltet den Operationscode. Dieser sagt der CPU, was sie machen soll und wie viele Bytes der Befehl lang ist. Ist die Endadresse des Befehls erreicht oder schon überschritten, wird bei den nächsten Zyklen die Adresse nicht mehr erhöht. Die CPU liest dann ein- oder mehrmals die gleiche Adresse aus.

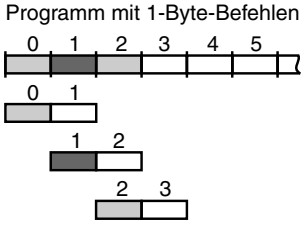
2-Byte-Befehle

Die meisten Befehle der CPU sind 2-Byte-Befehle. Bei ihnen erfolgt das Einlesen und die Ausführung in einem Zyklus. In Bezug auf diese Befehle arbeitet die CPU optimal.



1-Byte-Befehle

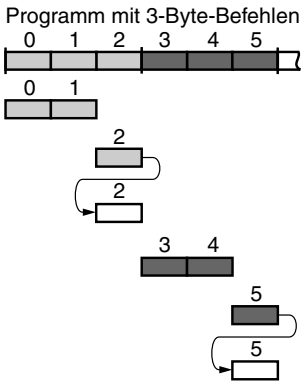
Auch bei 1-Byte-Befehlen liest der Controller zu Beginn des Befehls immer zwei aufeinanderfolgende Adressen ein. Da das zweite Byte nicht benötigt wird, wird es verworfen. Es wird beim nächsten Zyklus noch einmal plus dem folgenden Byte eingelesen.



3-Byte-Befehle

Zu Beginn des Befehls liest der Controller wieder zwei aufeinander folgende Adressen ein. Da der Operationscode ihm sagt, dass es sich um einen 3-Byte-Befehl handelt, erhöht er die Adresse jetzt nur noch bis zum dritten Byte. Er liest deshalb beim zweiten Zyklus das dritte Byte zweimal ein.

Dauert das Lesen und Bearbeiten eines Befehls länger als zwei Zyklen, wird in den folgenden Zyklen immer die Adresse des dritten Bytes eingelesen, zweimal pro Zyklus.



Die Bearbeitung eines Programms von der CPU mit unterschiedlich langen Befehlen wird im Folgenden verdeutlicht. Dabei ist die Länge der Befehle fest vorgegeben und kann der Befehlsliste entnommen werden. Ein Ausschnitt aus der Befehlsliste ist auf der folgenden Seite dargestellt. Der Mikrocomputer soll folgendes Assemblerprogramm bearbeiten:

```
MOV R0,#90     2 Byte, 1 Zyklus
MOV A,F8       2 Byte, 1 Zyklus
MOV @R0,A      1 Byte, 1 Zyklus
MOV E8,#FF     3 Byte, 2 Zyklen
```

Das Assemblerprogramm wird in Maschinensprache übersetzt und gespeichert. Mithilfe der Hex- Befehlsliste (siehe folgende Seiten), sieht das Programm dann so aus:

Adresse	Inhalt	Befehl
0000	78	MOV R0,#
0001	90	90
0002	E5	MOV A,
0003	F8	F8
0004	F6	MOV @R0, A
0005	75	MOV
0006	E8	E8
0007	FF	#FF

Auszug aus der Befehlsliste für den Datentransfer:

Mnemonic		Description	Byte	Cycle
Data Transfer				
MOV	A,Rn	Move register to accumulator	1	1
MOV	A,direct [1)]	Move direct byte to accumulator	2	1
MOV	A,@Ri	Move indirect RAM to accumulator	1	1
MOV	A,#data	Move immediate data to accumulator	2	1
MOV	Rn,A	Move accumulator to register	1	1
MOV	Rn,direct	Move direct byte to register	2	2
MOV	Rn,#data	Move immediate data to register	2	1
MOV	direct,A	Move accumulator to direct byte	2	1
MOV	direct,Rn	Move register to direct byte	2	2
MOV	direct,direct	Move direct byte to direct byte	3	2
MOV	direct,@Ri	Move indirect RAM to direct byte	2	2
MOV	direct,#data	Move immediate data to direct byte	3	2
MOV	@Ri,A	Move accumulator to indirect RAM	1	1
MOV	@Ri,direct	Move direct byte to indirect RAM	2	2
MOV	@Ri,#data	Move immediate data to indirect RAM	2	1
MOV	DPTR,#data 16	Load data pointer with a 16-bit constant	3	2
MOVC	A,@A+DPTR	Move code byte relative to DPTR to accumulator	1	2
MOVC	A,@A+PC	Move code byte relative to PC to accumulator	1	2
MOVX	A,@Ri	Move external RAM (8-bit addr.) to A	1	2
MOVX	A,@DPTR	Move external RAM (16-bit addr.) to A	1	2
MOVX	@Ri,A	Move A to external RAM (8-bit addr.)	1	2

[1)] MOV A,ACC is not a valid instruction

Auszug aus der Befehlsliste für die arithmetischen Operationen:

Mnemonic		Description	Byte	Cycle
Arithmetic Operations				
ADD	A,Rn	Add register to accumulator	1	1
ADD	A,direct	Add direct byte to accumulator	2	1
ADD	A,@Ri	Add indirect RAM to accumulator	1	1
ADD	A,#data	Add immediate data to accumulator	2	1
ADDC	A,Rn	Add register to accumulator with carry flag	1	1
ADDC	A,direct	Add direct byte to A with carry flag	2	1
ADDC	A,@Ri	Add indirect RAM to A with carry flag	1	1
ADDC	A,#data	Add immediate data to A with carry flag	2	1
SUBB	A,Rn	Subtract register from A with borrow	1	1
SUBB	A,direct	Subtract direct byte from A with borrow	2	1
SUBB	A,@Ri	Subtract indirect RAM from A with borrow	1	1
SUBB	A,#data	Subtract immediate data from A with borrow	2	1
INC	A	Increment accumulator	1	1
INC	Rn	Increment register	1	1
INC	direct	Increment direct byte	2	1
INC	@Ri	Increment indirect RAM	1	1

Befehlsliste in Hex-Folge

Hex Mnemonic	Hex Mnemonic	Hex Mnemonic	Hex Mnemonic	Hex Mnemonic	Hex Mnemonic	Hex Mnemonic	Hex Mnemonic
00 NOP	20 JB badr,rel	40 JC rel	60 JZ rel	80 SJMP rel	A0 ORL C,/badr	C0 PUSH dadr	E0 MOVX A,@DPTR
01 AJMP page 0	21 AJMP page 1	41 AJMP page 2	61 AJMP page 3	81 AJMP page 4	A1 AJMP page 5	C1 AJMP page 6	E1 AJMP page 7
02 LJMP adr16	22 RET	42 ORL dadr,A	62 XRL dadr,A	82 ANL C,badr	A2 MOV C,badr	C2 CLR badr	E2 MOVX A,@R0
03 RR A	23 RL A	43 ORL dadr,#k8	63 XRL dadr,#k8	83 MOVC A,@A+PC	A3 INC DPTR	C3 CLR C	E3 MOVX A,@R1
04 INC A	24 ADD A,#k8	44 ORL A,#k8	64 XRL A,#kt8	84 DIV AB	A4 MUL AB	C4 SWAP A	E4 CLR A
05 INC dadr	25 ADD A,dadr	45 ORL A,dadr	65 XRL A,dadr	85 MOV dadr,dadr	A5 reserviert	C5 XCH A,dadr	E5 MOV A,dadr
06 INC @R0	26 ADD A,@R0	46 ORL A,@R0	66 XRL A,@R0	86 MOV dadr,@R0	A6 MOV @R0,dadr	C6 XCH A,@R0	E6 MOV A,@R0
07 INC @R1	27 ADD A,@R1	47 ORL A,@R1	67 XRL A,@R1	87 MOV dadr,@R1	A7 MOV @R1,dadr	C7 XCH A,@R1	E7 MOV A,@R1
08 INC R0	28 ADD A,R0	48 ORL A,R0	68 XRL A,R0	88 MOV dadr,R0	A8 MOV R0,dadr	C8 XCH A,R0	E8 MOV A,R0
09 INC R1	29 ADD A,R1	49 ORL A,R1	69 XRL A,R1	89 MOV dadr,R1	A9 MOV R1,dadr	C9 XCH A,R1	E9 MOV A,R1
0A INC R2	2A ADD A,R2	4A ORL A,R2	6A XRL A,R2	8A MOV dadr,R2	AA MOV R2,dadr	CA XCH A,R2	EA MOV A,R2
0B INC R3	2B ADD A,R3	4B ORL A,R3	6B XRL A,R3	8B MOV dadr,R3	AB MOV R3,dadr	CB XCH A,R3	EB MOV A,R3
0C INC R4	2C ADD A,R4	4C ORL A,R4	6C XRL A,R4	8C MOV dadr,R4	AC MOV R4,dadr	CC XCH A,R4	EC MOV A,R4
0D INC R5	2D ADD A,R5	4D ORL A,R5	6D XRL A,R5	8D MOV dadr,R5	AD MOV R5,dadr	CD XCH A,R5	ED MOV A,R5
0E INC R6	2E ADD A,R6	4E ORL A,R6	6E XRL A,R6	8E MOV dadr,R6	AE MOV R6,dadr	CE XCH A,R6	EE MOV A,R6
0F INC R7	2F ADD A,R7	4F ORL A,R7	6F XRL A,R7	8F MOV dadr,R7	AF MOV R7,dadr	CF XCH A,R7	EF MOV A,R7
10 JBC badr,rel	30 JNB badr,rel	50 JNC rel	70 JNZ rel	90 MOV DPTR,#k16	B0 ANL C,/badr	D0 POP dadr	F0 MOVX @DPTR,A
11 ACALL page 0	31 ACALL page 1	51 ACALL page 2	71 ACALL page 3	91 ACALL page 4	B1 ACALL page 5	D1 ACALL page 6	F1 ACALL page 7
12 LCALL adr16	32 RETI	52 ANL dadr,A	72 ORL C,badr	92 MOV badr,C	B2 CPL badr	D2 SETB badr	F2 MOVX @R0,A
13 RRC A	33 RLC A	53 ANL dadr,#k8	73 JMP @A+DPTR	93 MOVC A,@A+DPTR	B3 CPL C	D3 SETB C	F3 MOVX @R1,A
14 DEC A	34 ADDC A,#k8	54 ANL A,#k8	74 MOV A,#k8	94 SUBB A,#k8	B4 CJNE A,#k8,rel	D4 DA A	F4 CPL A
15 DEC dadr	35 ADDC A,dadr	55 ANL A,dadr	75 MOV dadr,#k8	95 SUBB A,dadr	B5 CJNE A,dadr,rel	D5 DJNZ dadr,rel	F5 MOV dadr,A
16 DEC @R0	36 ADDC A,@R0	56 ANL A,@R0	76 MOV @R0,#k8	96 SUBB A,@R0	B6 CJNE @R0,#k8,rel	D6 XCHD A,@R0	F6 MOV @R0,A
17 DEC @R1	37 ADDC A,@R1	57 ANL A,@R1	77 MOV @R1,#k8	97 SUBB A,@R1	B7 CJNE @R1,#k8,rel	D7 XCHD A,@R1	F7 MOV @R1,A
18 DEC R0	38 ADDC A,R0	58 ANL A,R0	78 MOV R0,#k8	98 SUBB A,R0	B8 CJNE R0,#k8,rel	D8 DJNZ R0,rel	F8 MOV R0,A
19 DEC R1	39 ADDC A,R1	59 ANL A,R1	79 MOV R1,#k8	99 SUBB A,R1	B9 CJNE R1,#k8,rel	D9 DJNZ R1,rel	F9 MOV R1,A
1A DEC R2	3A ADDC A,R2	5A ANL A,R2	7A MOV R2,#k8	9A SUBB A,R2	BA CJNE R2,#k8,rel	DA DJNZ R2,rel	FA MOV R2,A
1B DEC R3	3B ADDC A,R3	5B ANL A,R3	7B MOV R3,#k8	9B SUBB A,R3	BB CJNE R3,#k8,rel	DB DJNZ R3,rel	FB MOV R3,A
1C DEC R4	3C ADDC A,R4	5C ANL A,R4	7C MOV R4,#k8	9C SUBB A,R4	BC CJNE R4,#k8,rel	DC DJNZ R4,rel	FC MOV R4,A
1D DEC R5	3D ADDC A,R5	5D ANL A,R5	7D MOV R5,#k8	9D SUBB A,R5	BD CJNE R5,#k8,rel	DD DJNZ R5,rel	FD MOV R5,A
1E DEC R6	3E ADDC A,R6	5E ANL A,R6	7E MOV R6,#k8	9E SUBB A,R6	BE CJNE R6,#k8,rel	DE DJNZ R6,rel	FE MOV R6,A
1F DEC R7	3F ADDC A,R7	5F ANL A,R7	7F MOV R7,#k8	9F SUBB A,R7	BF CJNE R7,#k8,rel	DF DJNZ R7,rel	FF MOV R7,A

k8 ≙ konst8 k16 ≙ konst16

Die Ausführung kann, je nach Befehl, innerhalb der CPU erfolgen oder externe Baugruppen ansprechen. Erfolgt die Ausführung intern, ist sie auf dem externen Bus nicht sichtbar. Wird auf externe Baugruppen wie Speicher oder Ports zugegriffen, können die Signale auf den Busleitungen verfolgt werden. Ist die Instruktion ausgeführt, gibt die CPU die nächste Adresse auf dem Adressbus aus und das Steuersignal zum Lesen des Speicherinhaltes. Dieser Inhalt ist der Operationscode der nächsten Instruktion. Die Instruktion wird wieder komplett gelesen und dann ausgeführt.

So arbeitet die CPU Befehl für Befehl des Programms ab.

Übung 1.1

1. Aus welchen Leitungen besteht das Bussystem eines Mikrocomputers?
2. Was steht im Programm Counter?
3. Wie viele Adressen lassen sich mit 16 Bit darstellen?
4. Mit welcher Anzahl Leitungen kann eine serielle Datenübertragung arbeiten?
5. Erklären Sie den Begriff „Baudrate“.
6. Stellen Sie zwei Befehle in Assemblersprache und in Maschinensprache dar.
7. Was steht im ersten Byte jedes Befehls?
8. In welchem Speicherbaustein ist das Programm ab Adresse 0000h gespeichert?
9. In welchem Speicherbaustein werden variable Daten gespeichert?

Übung 1.2

Folgendes Assemblerprogramm soll vom Mikrocomputer bearbeitet werden:

Assemblerbefehle:

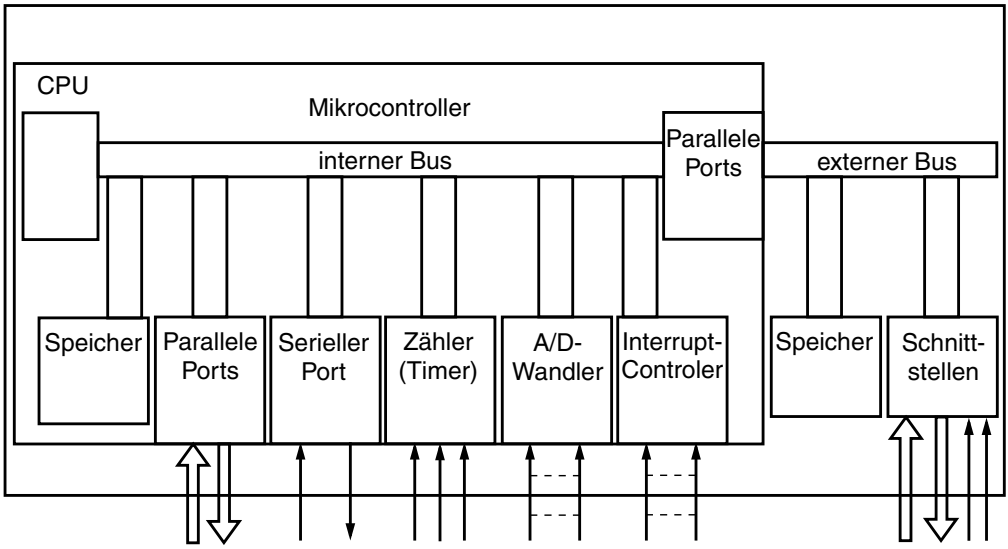
```
MOV R0,E8
MOV A, 90
ADD A, R0
ADD A, #10
MOV F0, A
```

1. Schreiben Sie hinter die Assemblerbefehle die Befehlslänge in Bytes und die Anzahl der Zyklen, die der Controller zur Bearbeitung benötigt.
2. Übersetzen Sie das Programm in Maschinensprache. Listen Sie die Adressen und die Inhalte auf (siehe vorhergehendes Beispiel). Zur Übersetzung verwenden Sie die „Befehlsliste in Hexform“.

2 Der Mikrocontroller

Der Mikrocontroller beinhaltet auf einem Chip einen kompletten Mikrocomputer, wie in Kapitel 1 beschrieben. Auf dem Chip sind die CPU, ein ROM-Speicher für das Programm, ein RAM-Speicher für die variablen Daten sowie parallele und serielle Ein- und Ausgabeports integriert. Die CPU ist über ein internes Bussystem mit dem Speicher und den Schnittstellen-Baugruppen verbunden. Der Controller wird hauptsächlich im Bereich der Automatisierungs-, Steuerungs- und Antriebstechnik eingesetzt. Speziell für diese Anwendungsgebiete sind außer den aufgeführten Standard-Baugruppen noch eine Reihe zusätzlicher Funktionseinheiten in den Controller integriert. Solche Funktionseinheiten sind z. B. schnelle Zähler (Timer), A/D-Wandler oder Interrupt-Controller.

Werden zusätzliche Funktionseinheiten benötigt, lässt sich ein externes Bussystem aufbauen. Daran lassen sich dann weitere Speicher oder Ports anschließen.

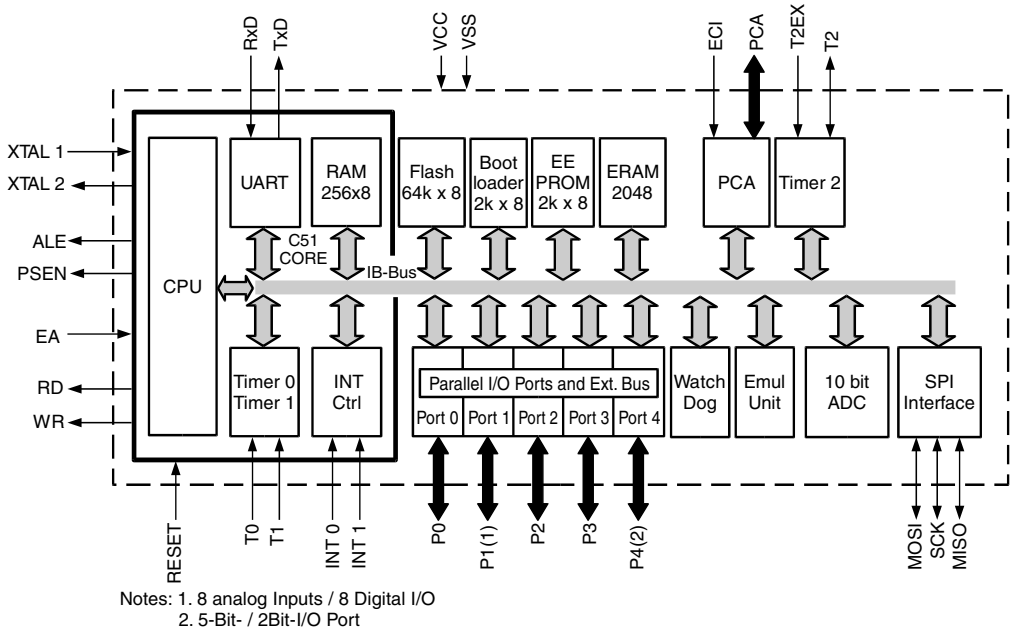


Dieses Buch stellt Controller der 8051-Familie bzw. deren Derivate vor. Obwohl die Architektur des 8051-Controllers schon 1980 von Intel vorgestellt wurde, gibt es auch heute noch von fast jedem namenhaften Hersteller ein oder mehrere Controller, die diese Architektur verwenden. Dabei verwenden die Controller dann einen gleichen Befehlsatz, und unterscheiden sich nur in den zusätzlichen Funktionseinheiten, wie zum Beispiel dem „In-System-Programming“ oder einem zusätzlichen internen Speicher. Zudem haben die heutigen 8051-er Derivate optimierte Befehlsausführungszeiten und können mit einem höheren externen Takt betrieben werden.

In diesem Buch soll beispielhaft ein 8051-er Derivat von Atmel, der AT89C51AC3 vorgestellt werden. Zudem werden einige Vergleiche zwischen unterschiedliche Realisierungen (zum Beispiel beim A/D-Wandler) zu dem älteren nicht mehr erhältlichen Typ SAB 80C535 gemacht.

2.1 Das Blockschaltbild des Mikrocontrollers

Folgendes Blockschaltbild gilt für den Controller AT89C51AC3 von Atmel, welches mit kleinen Abweichungen anderen 8051er-Controllern gleicht.



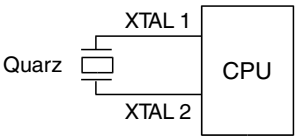
Notes: 1. 8 analog Inputs / 8 Digital I/O
2. 5-Bit- / 2Bit-I/O Port

2.1.1 Der C51-Core

Im linken umrahmten Teil des Blockschaltbildes ist die klassische 8051-Architektur mit der CPU, dem CPU-nahen 256 KByte RAM-Speicher, der seriellen Kommunikation (UART-Schnittstelle), zwei speziellen Timern und einem Interruptblock dargestellt. Dieser Teil entspricht genau dem Urtypen der 8051-Architektur und lässt sich ohne Abweichungen gleich programmieren.

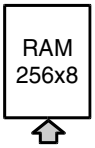
CPU

Die CPU ist über das interne Bussystem mit den übrigen Funktionseinheiten verbunden. Der Systemtakt der CPU wird über einen extern zugeschalteten Quarz an den Anschlüssen XTAL1 und XTAL2 erzeugt.



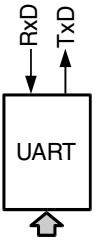
Speicher

Der CPU-nahe Speicher dient zum schnellen Abspeichern und Laden von variablen Daten. Meist sind dies einzelne Merkerbits, Zählvariablen oder Ähnliches.



UART-Schnittstelle

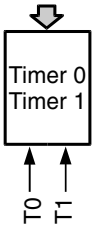
Der Controller ist mit einer seriellen Schnittstelle ausgestattet, die vollduplex arbeitet. Die Sendeleitung TxD (Transmit Data) ist der Anschluss 1 von Port 3. Die Empfangsleitung RxD (Receive Data) ist der Anschluss 0 des Portes 3. Der Schnittstelle können die Daten von der CPU parallel übergeben werden.



Das serielle Senden und Empfangen mit einstellbaren Übertragungsprotokoll wird von der Funktionseinheit selbstständig durchgeführt. Dasselbe gilt für die seriell empfangenen Bits, die von der CPU parallel gelesen werden können.

Timer

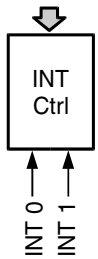
Die Timer 0 und 1 sind zwei unabhängig voneinander arbeitende Zähler. Sie lassen sich als Ereigniszähler oder als Zeitgeber einsetzen.



Bei der Verwendung als Ereigniszähler werden externe Impulse gezählt, die an bestimmten Eingängen der parallelen Ports eintreffen. Bei Verwendung als Zeitgeber werden interne Impulse gezählt, die von der Oszillatorfrequenz abgeleitet werden. Die Zeiten ergeben sich aus der Periodendauer des internen Taktes und dem Zählwert.

Interrupt

Mithilfe der Interrupt-Control-Funktion kann ein laufendes Programm durch ein internes oder externes Ereignis unterbrochen werden und eine entsprechende Funktion ausgeführt werden. Zum Beispiel, wenn ein Not-Aus betätigt wird.

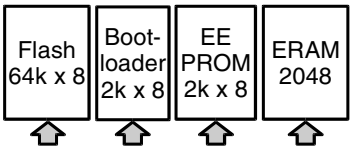


2.1.2 Zusätzliche Funktionen

Rechts neben dem C51 Core sind in dem Blockschaltbild zusätzliche Blöcke angeordnet, die über den internen Bus des Controllers verbunden sind. Diese Blöcke sind von Atmel dem 8051-Kern hinzugefügt bzw. in bestimmten Bereichen optimiert worden, um den Mikrocontroller-Chip für den Anwender komfortabler und vielseitiger nutzbar zu machen. Die Funktionsweise und Architektur dieser Blöcke weicht teilweise von den Urtypen der 8051-Architektur ab. Viele Teile sind auch hinzugefügt worden, wie zum Beispiel mehrere zusätzliche Speicher auf dem Chip.

Speicher

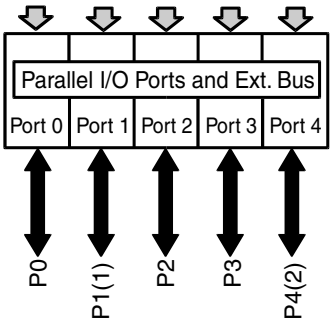
Zum zusätzlichen Speicher gehören ein 64 KByte Flash-Speicher, in dem das eigentliche Programm gespeichert wird, ein Bootloader, mit dem das Programm direkt über die serielle Schnittstelle in den Flash-Speicher geschrieben werden kann, ein 2 KByte EEPROM, in dem Daten auch nach dem Ausschalten des Controllers erhalten bleiben und ein 2 KByte ERAM-Speicher für die kurze Speicherung von Daten während des Betriebes.



Parallele Ports

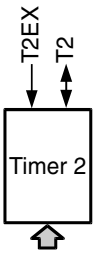
Dieser Controller verfügt über fünf digitale Ports mit jeweils 8 Bit. Mithilfe dieser Ports können digitale Eingänge ausgelesen, bzw. digitale Ausgänge gesetzt und gelöscht werden.

Einige Ports haben zusätzliche Alternativfunktionen, auf die per Programm umgeschaltet werden kann.



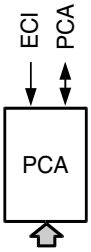
Timer 2

Der Timer 2 funktioniert wie die Timer 0 und Timer 1, ist allerdings mit zusätzlichen Funktionen ausgestattet, wie zum Beispiel mit einem programmierbaren Takt-Ausgang.



PCA

Die PCA-Einheit (Programmable Counter Array) ist ein komplexer Timer, der für verschiedene Anwendungen, wie zum Beispiel der Pulsweitenmodulation genutzt wird. Mit diesem Timer kann die CPU sehr entlastet werden, und spezielle zeitgesteuerte Anwendungen können selbständig von diesem Modul durchgeführt werden.

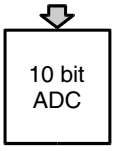


Watch Dog

Der Watch Dog ist ebenfalls ein spezieller Zähler, der ständig überprüft, ob das Programm in einer festen Zeit zyklisch arbeitet. Wenn das Programm an einer Stelle nicht mehr weiter bearbeitet wird, so stellt dies der Watch Dog fest, und führt zum Beispiel selber einen Reset des Controller durch.

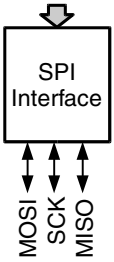
10-Bit ADC

Der 10-Bit ADC (Analog Digital Converter) wandelt analoge Eingangssignale in 10-Bit-Digitalwerte um. Hierzu stehen acht analoge Eingänge bereit. Die Wandlung erfolgt nacheinander für jeden Analogeingang.



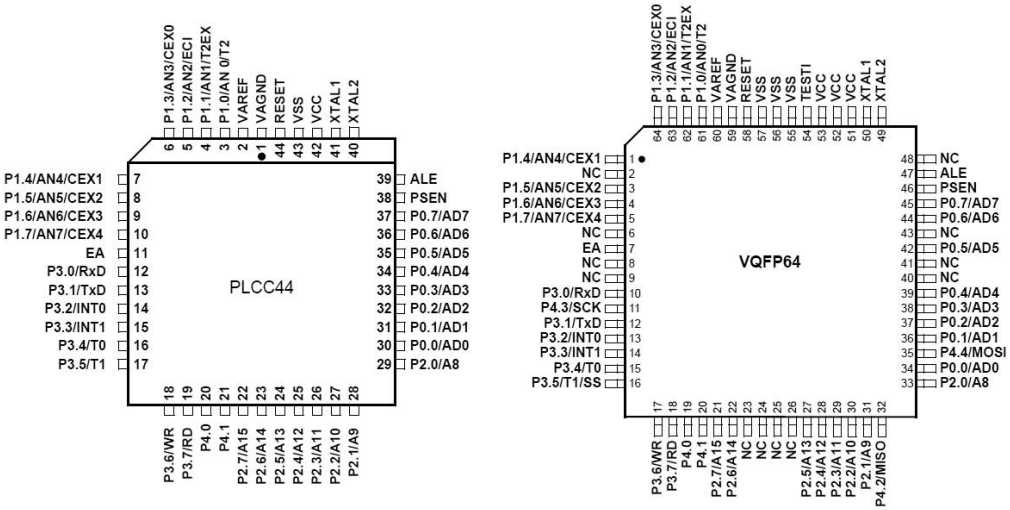
SPI-Interface (nur Gehäuse über 52 Pins)

Das „Seriell Port Interface“ ist eine spezieller Baustein für die serielle Datenkommunikation zwischen mehreren Geräten. Hierbei können sowohl Peripherie-Geräte als auch andere CPUs, seriell kommunizieren und somit Daten austauschen. Die komplette Steuerung hierfür übernimmt dieser Block.



2.2 Anschlussbezeichnungen und Funktionen

Die auf dem heutigen Markt erhältlichen Controller werden in unterschiedlichen Gehäuseformen angeboten. So gibt es auch den Atmel AT89C51AC3 zum Beispiel in vier verschiedenen Gehäuseformen. Zwei Gehäuseformen werden in der SMD-Bauweise angeboten, zwei andere lassen sich in einem PLCC-Sockel einsetzen. Das kleinste PLCC- Gehäuse weist 44 Pins auf, das größere 52 Pins. Die SMD-Gehäuse haben jeweils 44 Pins und 64 Pins. Bei dem Gehäuse mit mehr Pins wird zusätzlich noch der Portpin P4.2, P4.3 und P4.4 nach außen geführt. In diesen Gehäusen steht dann das SPI-Interface zusätzlich zur Verfügung. Die anderen übrig gebliebenen Pins sind meistens nicht angeschlossen („Not Connected"-NC).



Genaue Anschlussbezeichnungen und Funktionen

Symbolische Bezeichnungen		Funktionen	(E) = Eingang (A) = Ausgang	
Hauptfunktion	**Alternativ-funktionen**	**Hauptfunktion**	**Alternativ-funktionen**	
P0.0 ⋮ P0.7		Port 0.0 (E/A) ⋮ Port 0.7 (E/A)	Adressbus und Datenbus	A0/D0 ⋮ A7/D7
P1.0	 AN0 T2	Port 1.0 (E/A)	 Analogeingang 0 Takteingang Timer 2	 (E) (E)
P1.1	 AN1 T2EX	Port 1.1 (E/A)	 Analogeingang 1 Triggereingang Timer 2	 (E) (E)

Symbolische Bezeichnungen		Funktionen	(E) = Eingang (A) = Ausgang	
Hauptfunktion	**Alternativ-funktionen**	**Hauptfunktion**	**Alternativ-funktionen**	
P1.2		Port 1.2 (E/A)		
	AN2		Analogeingang 2	(E)
	ECI		Takteingang PCA	(E)
P1.3		Port 1.3 (E/A)		
	AN3		Analogeingang 3	(E)
	CEX0		PCA Modul 0 Eingang	(E)
			PWM Ausgang 0	(A)
P1.4		Port 1.4 (E/A)		
	AN4		Analogeingang 4	(E)
	CEX1		PCA Modul 1 Eingang	(E)
			PWM Ausgang 1	(A)
P1.5		Port 1.5 (E/A)		
	AN5		Analogeingang 5	(E)
	CEX2		PCA Modul 2 Eingang	(E)
			PWM Ausgang 2	(A)
P1.6		Port 1.6 (E/A)		
	AN6		Analogeingang 6	(E)
	CEX3		PCA Modul 3 Eingang	(E)
			PWM Ausgang 3	(A)
P1.7		Port 1.7 (E/A)		
	AN7		Analogeingang 7	(E)
	CEX4		PCA Modul 4 Eingang	(E)
			PWM Ausgang 4	(A)
P2.0 ⋮ P2.7		Port 0.0 (E/A) ⋮ Port 0.7 (E/A)	Adressbus	A8 ⋮ A15
P3.0		Port 3.0 (E/A)		
	RxD		serielle Schnittstelle RxD	(E)
P3.1		Port 3.1 (E/A)		
	TxD		serielle Schnittstelle TxD	(A)
P3.2		Port 3.2 (E/A)		
	$\overline{\text{INT0}}$		Interrupt 0	(E)
P3.3		Port 3.3 (E/A)		
	$\overline{\text{INT1}}$		Interrupt 1	(E)
P3.4		Port 3.4 (E/A)		
	T0		Timer 0 Impuls	(E)
P3.5		Port 3.5 (E/A)		
	T1		Timer 1 Impuls	(E)
P3.6		Port 3.6 (E/A)		
	WR		Schreiben (Ext. Speicher)	(A)
P3.7		Port 3.7 (E/A)		
	RD		Lesen (Ext. Speicher)	(A)

Symbolische Bezeichnungen		Funktionen	(E) = Eingang (A) = Ausgang
Hauptfunktion	**Alternativ-funktionen**	**Hauptfunktion**	**Alternativ-funktionen**
P4.0		Port 4.0 (E/A)	
P4.1		Port 4.1 (E/A)	
P4.2	MISO	Port 4.2 (E/A)	Master Input/Slave Output vom SP-Interface (E/A)
P4.3	SCK	Port 4.3 (E/A)	Serieller Takt (SPI) (A)
P4.4	MOSI	Port 4.4 (E/A)	Master Output/Slave Input vom SP-Interface (E/A)
VAREF VAGND		Obere Referenzspannung für die A/D-Wandlung (max. 3 V) Untere Referenzspannung für die A/D-Wandlung	
ALE RESET EA PSEN		Adress Latch Enable Reset External Access Programm Store Enable	
XTAL 1 XTAL 2		Oszillator	
VSS GND VCC TESTI		Masse 0 V Spannungversorgung 5 V Verbindung zu VCC	

Übung 2.1

Fragen zu den Controller-Anschlüssen:
1. Welche Ports werden für den externen Bus benötigt?
2. Welche Portpins sind reine Eingabe-/Ausgabeports ohne Alternativfunktion?
3. An welchen Portpins findet die serielle Datenein- und -ausgabe statt?
4. Timer 1 soll als Ereigniszähler genutzt werden. Wo sind die Eingangsimpulse anzuschließen?
5. Sie wollen analoge Eingangspannungen erfassen. Wie viele Spannungen können Sie an dem Baustein anschließen und wie groß dürfen diese Spannungen maximal sein?

Übung 2.2

Fragen zu den internen Controller-Baugruppen/Gehäusen:
1. Welcher Unterschied besteht zwischen dem 52-Pin- und 44-Pin-Gehäuse?
2. Welche Timer enthält der Controller AT89C51AC3?
3. In welchem Speicher wird das eigentliche Programm gespeichert?
4. Sie wollen Daten über eine längere Zeit speichern, auch wenn der Controller ausgeschaltet wird. In welchem Speicher sollten Sie die Daten abspeichern?

3 Externe Speicherorganisation

Der Mikrocontroller erhält seine Arbeitsanweisungen von einem Programm. Das Programm bearbeitet die Daten, die in den Controller eingegeben werden. Programm und Daten sind zu speichern.

Der Mikrocontroller AT89C51AC3 enthält einen 64 KByte großen Flash-Speicher für das Programm. Zudem enthält der Controller einen 2KByte großen ERAM-Speicher (extended RAM), um Daten abzuspeichern. Sollte dieser Speicher nicht ausreichen, so lässt sich ein externer Datenspeicher am Controller anschließen.

Zu diesem Speicher enthält der 8051er-Core noch einen integrierten RAM-Speicher für 256 variable, schnell verfügbare Bytes.

An den Urtypen der 8051er-Controllern war meist nur der 256 Byte große Speicher im Mikrocontroller integriert. Sowohl der Programmspeicher, als auch der Datenspeicher mussten dann extern angeschlossen werden.

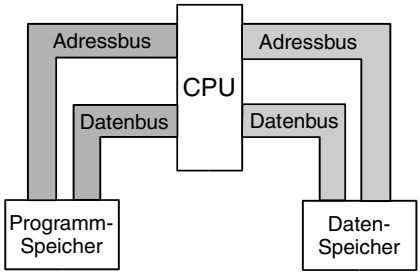
3.1 Speicher-Architekturen

In der Datenverarbeitung wird zwischen der Harvard- und der Von-Neuman-Architektur unterschieden.

Harvard-Architektur

Die Harvard-Architektur teilt den Speicher in Programm- und Datenspeicher auf. Beide Speicher liegen im gleichen Adressbereich. Dabei wird der Datenspeicher und der Programm- speicher konsequent voneinander getrennt. Der Datenspeicher und der Programmspeicher haben getrennte Busse. Dies lässt eine besonders schnelle Datenverarbeitung zu.

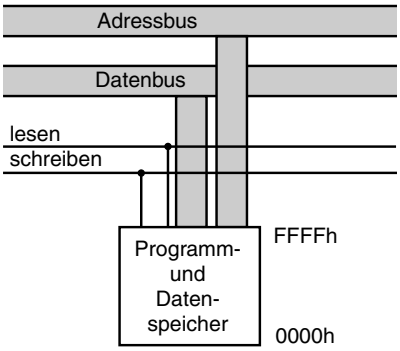
Harvard-Architektur:



Von-Neuman-Architektur

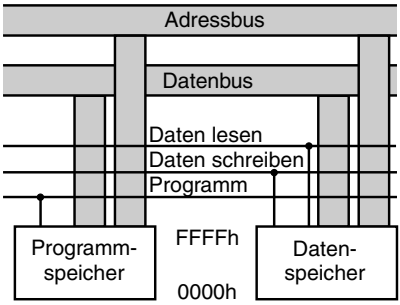
Bei der Von-Neuman-Architektur liegen Programm und Daten abwechselnd oder hintereinander im gleichen Speicher. Programm und Daten werden nur über die Adresse angewählt.

Von-Neuman-Architektur:

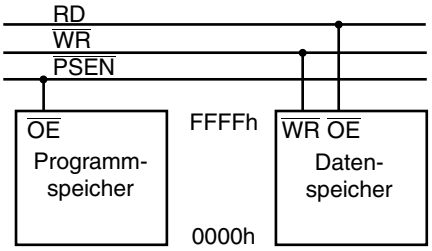


Speicher-Architektur der 8051-Familie

Die Controller der 8051-Familie mixen diese beiden Architekturen. Sie haben wie bei der „Von-Neumann- Architektur“ nur einen Adress- und Datenbus, und wie bei der „Harvard-Architektur“ einen getrennten Programm- und Datenspeicher. Der Programm- und Datenspeicher liegt im gleichen Adressbereich, beginnend bei der Adresse 0000 (hex) bis FFFF (hex).



Über das Steuersignal PSEN wird der Programmspeicher aktiviert. Da aus diesem Speicher nur gelesen werden muss, reicht eine Steuerleitung aus. Über die jeweilige Aktivierung der Leitung RD (Read = Daten lesen) bzw. WR (Write = Daten schreiben) wird dem Datenspeicher von der CPU mitgeteilt, ob gelesen oder geschrieben werden soll. Da nur ein Adress- und Datenbus vorhanden ist, muss hintereinander entweder aus dem Programmspeicher gelesen oder aus dem Datenspeicher gelesen bzw. geschrieben werden. Es können alledings dieselben Adressen sowohl für den Programm als auch für den Datenspeicher verwendet werden.



$\overline{\text{PSEN}}$ = Program STORE Enable
$\overline{\text{WR}}$ = Write
$\overline{\text{RD}}$ = Read

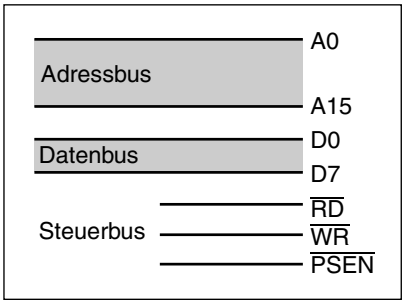
3.2 Aufbau eines externen Bussystems

Wie bereits erwähnt, lässt sich auch bei dem AT89C51AC3 Mikrocontroller ein externer Programm- und Datenspeicher anschließen. Hierzu kann der entsprechende interne Speicher deaktiviert werden.

Für den Anschluss der Speicherbausteine an den Mikrocontroller wird ein externes Bussystem benötigt.

Adressbus :16 Bit
Datenbus :8 Bit
Steuerbus :PSEN
= Programm Store Enable
= Freigabe Programmspeicher
:WR
= Write
= Freigabe Schreiben
:RD
= Read
= Freigabe Lesen

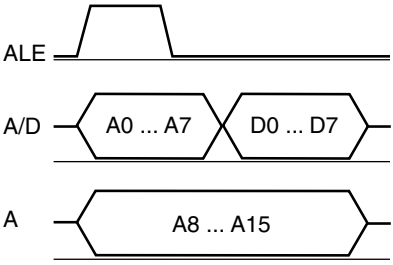
Externes Bussystem:



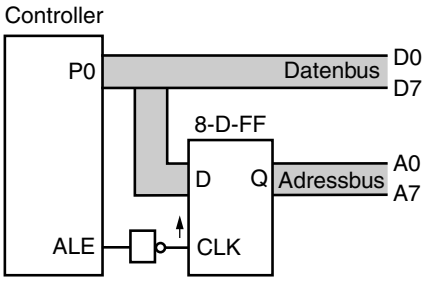
Die Bussignale für Adress- und Datenbus liefert der Controller nicht in der benötigten Form. Um Anschlusspins zu sparen, gibt er das Daten- und das niederwertige Adressbyte nacheinander über die gleichen acht Anschlussstifte aus.

Zuerst erscheint das Adress- und dann das Datenbyte. Da die Adresse aber über die gesamte Zykluszeit benötigt wird, ist sie in acht D-Flipflops zu speichern. Um das Speichern der Adressen zu veranlassen, liefert der Controller das Steuersignal ALE (Adress-Latch-Enable).

Controller-Signale:

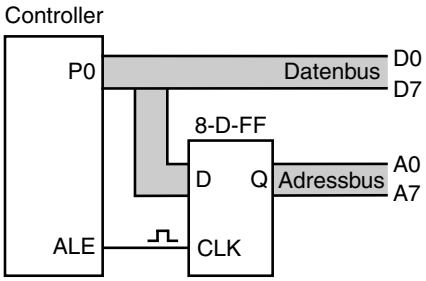


Die Adresssignale A0 bis A7 sind bei einem flankengesteuerten 8-D-Flipflop zum Zeitpunkt der negativen Flanke von ALE zu speichern.



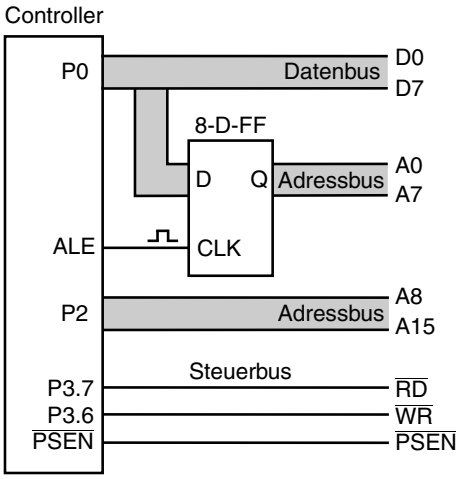
Wird ein Transparent-Latch eingesetzt, sind die Eingangssignale mit ALE = 1 in den Flipflops zu speichern. Mit ALE = 0 sind die Eingänge der Flipflops zu sperren.

Die Signale für das höherwertige Adressbyte gibt der Controller an seinem Port P2 aus.



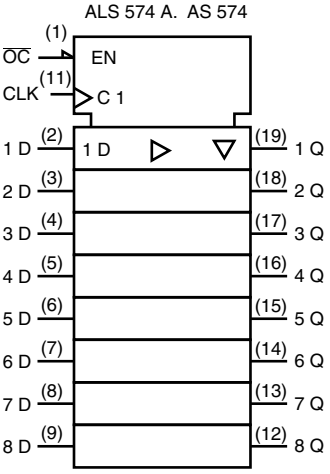
Controller mit externem Bussystem

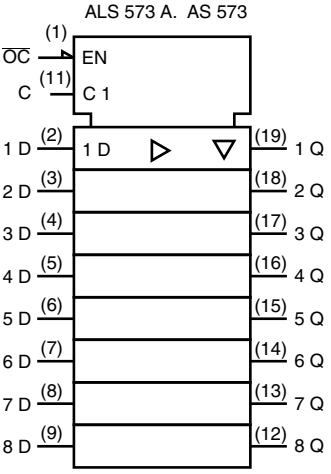
Leider gehen für das externe Bussystem 18 Portleitungen als Ein- und Ausgabeleitungen für die Peripherie verloren. An den externen Bus lassen sich jedoch nicht nur Speicherbausteine anschließen, sondern auch zusätzliche Eingabe- und Ausgabeports. Diese externen Ports werden von der CPU wie Speicheradressen angesprochen. Jeder 8-Bit-Port hat eine Adresse. Ein Eingabeport wird über das Steuersignal RD gelesen, ein Ausgabeport mit Hilfe des Steuersignals WR beschrieben.



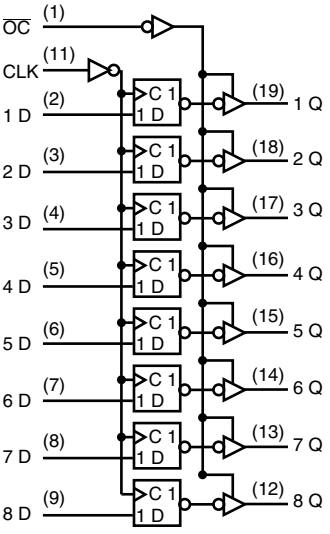
Übung 3.1

Setzen Sie folgende 8-D-Flipflops in die Schaltung für das externe Bussystem ein. Skizzieren Sie die Schaltung einmal unter Verwendung des Bausteins 74AC574 und und ein zweites Mal mit dem Baustein 74AC573.

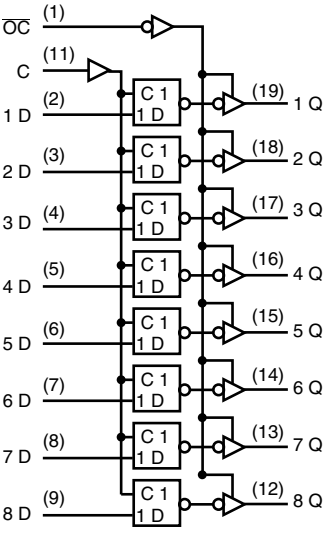




logic diagrams (positive logic)



logic diagrams (positive logic)



ALS 574 A. AS 574
(EACH FLIP-FLOP)

INPUTS			OUTPUT
$\overline{OC}$	CLK	D	Q
L	↑	L	H
L	↑	L	L
L	L	X	Q_0
H	X	X	Z

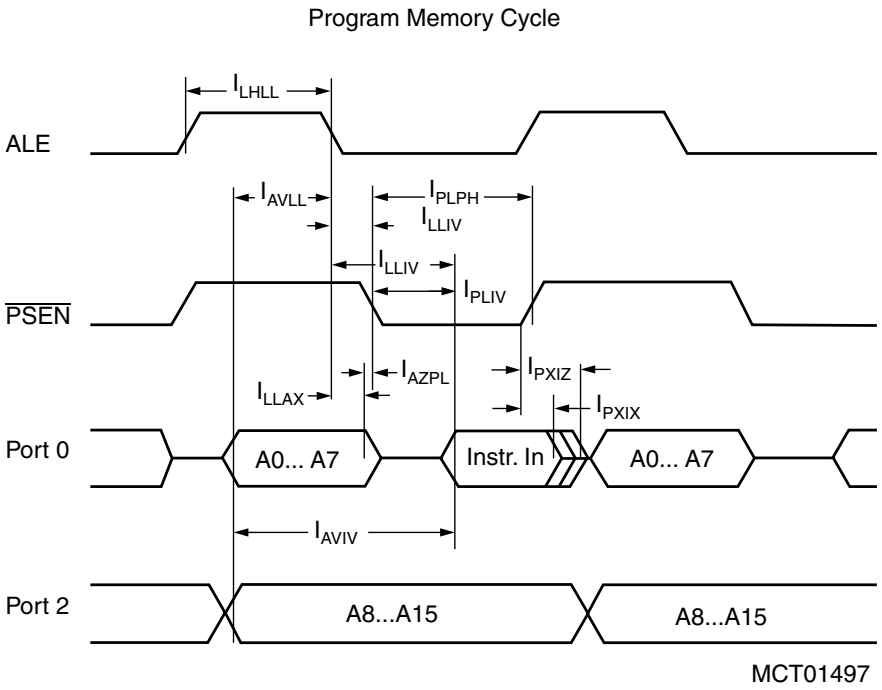
ALS 574 A. AS 574
(EACH FLIP-FLOP)

INPUTS			OUTPUT
ENABLE			Q
$\overline{OC}$	C	D	
L	H	H	H
L	H	L	L
L	L	X	Q_0
H	X	X	Z

3.3 Lesen aus dem Programmspeicher

Die genaue Arbeitsweise soll anhand des Signaldiagrammes für einen Programmspeicher-Lesezyklus erläutert werden.

Programmspeicher-Lesezyklus

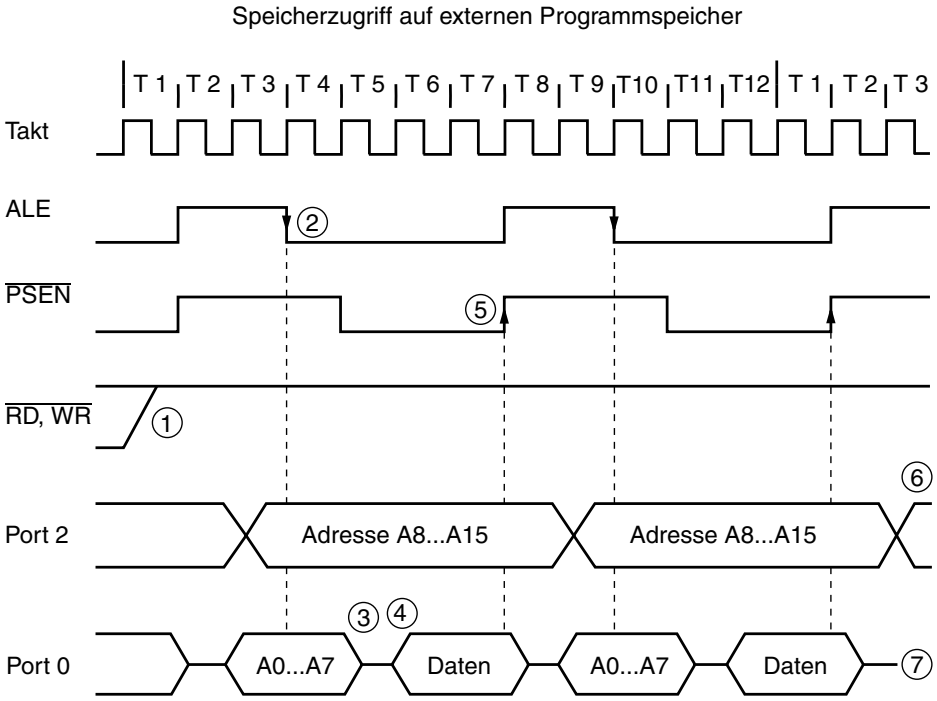


Ein Zyklus hat eine Länge von 12 Oszillatortakten. Bei unserer Betrachtung soll der Zyklus mit dem Wechsel der Adresse beginnen. Auf Port 0 gibt der Controller die Adresse A0 bis A7 aus, auf Port 2 die Adresse A8 bis A15. Mit der fallenden Flanke von ALE müssen die Adressen A0 bis A7 in acht D-Flipflops (Latch) gespeichert werden. Damit liegen dann alle Signale für den Adressbus an. Jetzt nimmt der Controller die Adressen A0 bis A7 wieder weg und schaltet den Port 0 auf Datenempfang. Die Daten sind die aus dem Programmspeicher gelesenen Inhalte. Der Programmspeicher legt den Inhalt der angewählten Adresse auf den Datenbus, wenn sein Ausgang durch das Steuersignal PSEN freigegeben wird, welches der Controller aktiviert. Es wechselt auf Low. Die jetzt ankommenden Daten speichert der Controller intern ab. Danach schaltet der Controller das Signal PSEN wieder auf High. Damit ist der erste Lesevorgang aus dem Programmspeicher beendet.

Das Signaldiagramm zeigt, dass daran sofort ein zweiter Lesevorgang anschließt. Während eines Zyklus mit 12 Oszillatortakten wird also zweimal aus dem Programmspeicher gelesen. Die meisten Befehle werden sofort intern ausgeführt, ehe der nächste Zyklus beginnt.

Übung 3.2

Beschreiben Sie, was an den markierten Punkten des Signaldiagramms geschieht.



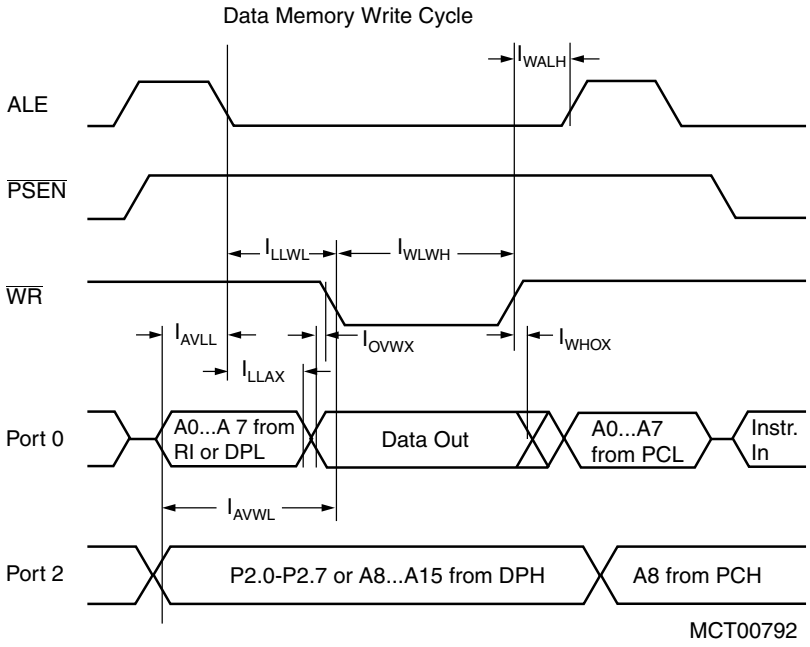
Reicht der interne Datenspeicher des Controllers nicht aus, muss ein externer Datenspeicher angeschlossen werden. Liest der Controller aus dem Programmspeicher einen Transferbefehl, bei dessen Ausführung er auf den externen Datenspeicher zugreifen muss, geschieht dies wie in folgenden Signaldiagrammen beschrieben.

3.4 Zugriff auf den externen Datenspeicher

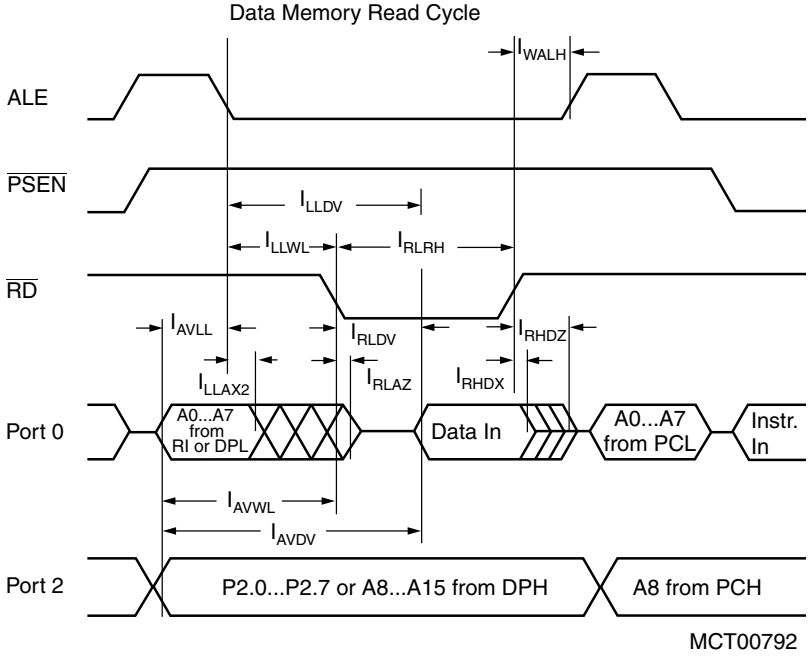
In den externen Datenspeicher können Daten geschrieben und wieder ausgelesen werden.

Beim Zugriff auf den Datenspeicher wird, im Gegensatz zum Zugriff auf den Programmspeicher, pro Zyklus nur eine Speicheradresse beschrieben oder gelesen. Deshalb wird auch das Steuersignal ALE nur einmal pro Zyklus aktiv.

Datenspeicher-Schreibzyklus



Datenspeicher-Lesezyklus



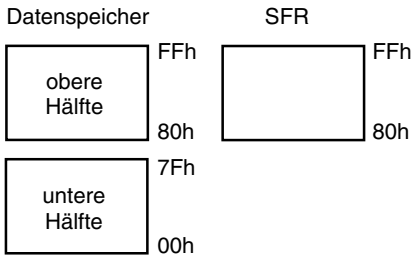
4 Die interne Speicher-organisation im C51-Core

Zu den Besonderheiten der 8051-Architektur gehört der interne 256 Byte Datenspeicher. Er ist in einzelne Bereiche aufgeteilt. Hervorzuheben ist der 16 Byte große bitadressierbare Bereich, in dem jeweils 128 Bits einzeln verarbeitet werden können. Dies macht einen großen Vorteil bei der Programmierung von Verknüfungssteuerungen, wie sie meistens in der Automatisierungstechnik benötigt werden.

Die internen Speicherbereiche

Der 256 Byte große Speicher besteht aus einem Datenspeicher mit einer unteren und einer oberen Hälfte. Parallel zur oberen Hälfte des Speichers existiert ein Spezial-Funktions-Register, welcher die gleichen Adressen aufweist. Mithilfe des Spezial-Funktions-Register (SFR) können über spezielle Bytes die Zusatzfunktionen, wie zum Beispiel die Ports, der A/D-Wandler usw. konfiguriert und angesteuert werden.

Interne Speicherbereiche:



Durch die Art der Adressierung wird dem Controller gesagt, auf welchen Bereich er zugreifen soll. Die obere Hälfte des Datenspeichers wird indirekt adressiert, die SFR werden direkt adressiert.

4.1 Die untere Hälfte des Datenspeichers

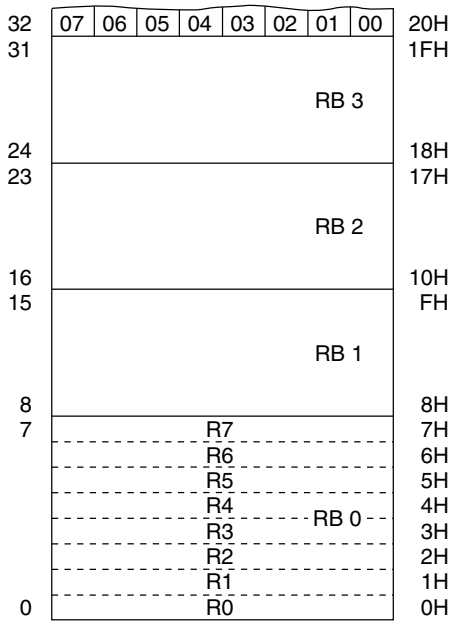
Die untere Hälfte des Datenspeichers ist aufgeteilt in Registerbänke, Bit-Speicher und Byte-Speicher.

Registerbänke RB0 bis RB3

Die vier Registerbänke liegen ab Adresse 0000h an aufwärts. Der Controller arbeitet immer nur mit einer der Registerbänke. Nach einem Reset ist Registerbank null eingeschaltet. Die Register R0 und R1 sind für die indirekte Adressierung des internen Datenspeichers einzusetzen. In die anderen Register lassen sich Zwischenergebnisse ablegen.

Bei einem Sprung in ein Unterprogramm kann die Registerbank gewechselt werden. Beim Rücksprung in das alte Programm lässt sich wieder auf die alte Registerbank umschalten. Damit sind auch die alten Zwischenergebnisse wieder vorhanden. Sie müssen beim Umschalten auf ein Unterprogramm also nicht einzeln auf dem Stack gerettet werden.

Registerbänke:



Das Umschalten der Registerbänke erfolgt über die Bits drei und vier in dem Spezial-Funktions-Register PSW (Programm Status Wort). Die in diese beiden Stellen gesetzte Dualzahl ist die Nummer der benutzten Registerbank.

Bit-Speicher

Bei den Bytes 20h bis 2Fh des internen Datenspeichers lässt sich jedes Bit einzeln adressieren. Die 128 Bits sind von 00h bis 7Fh durchnummeriert. In diesem Speicherbereich können sowohl Bits als auch die Bytes mit ihrer Hex-Adresse angesprochen werden. Ob es sich um eine Bit-oder Byte-Adresse handelt, erkennt der Controller an dem jeweiligen Befehl.

Der Befehlssatz enthält extra eine Gruppe von Befehlen für die Einzelbit-Verarbeitung.

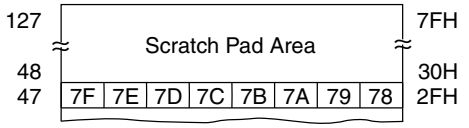
Bit-Speicher:

48									30H
47	7F	7E	7D	7C	7B	7A	79	78	2FH
46	77	76	75	74	73	72	71	70	2EH
45	6F	6E	6D	6C	6B	6A	69	68	2DH
44	67	66	65	64	63	62	61	60	2CH
43	5F	5E	5D	5C	5B	5A	59	58	2BH
42	57	56	55	54	53	52	51	50	2AH
41	4F	4E	4D	4C	4B	4A	49	48	29H
40	47	46	45	44	43	42	41	40	28H
39	3F	3E	3D	3C	3B	3A	39	38	27H
38	37	36	35	34	33	32	31	30	26H
37	2F	2E	2D	2C	2B	2A	29	28	25H
36	27	26	25	24	23	22	21	20	24H
35	1F	1E	1D	1C	1B	1A	19	18	23H
34	17	16	15	14	13	12	11	10	22H
33	0F	0E	0D	0C	0B	0A	09	08	21H
32	07	06	05	04	03	02	01	00	20H
31									1FH
									RB 3

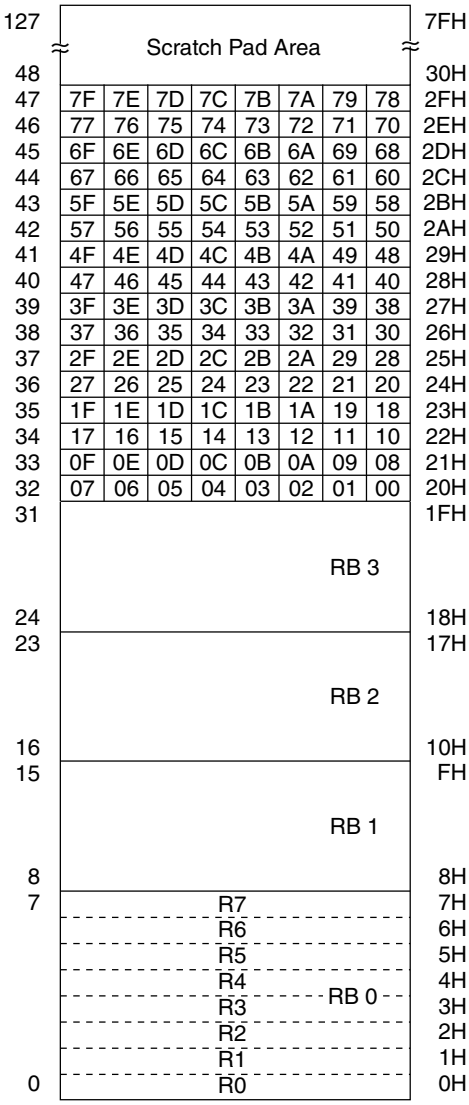
Byte-Speicher

Ab Adresse 30h beginnt der Byte-Speicher. Er umfasst 80 Bytes und geht bis Adresse 7Fh.

Byte-Speicher:



Die komplette untere Hälfte des internen Datenspeichers



4.2 Die obere Hälfte des Datenspeichers

Die obere Hälfte des Datenspeichers umfasst 128 Bytes und geht von Adresse 80h bis FFh. Die Bytes sind mithilfe der Register R0 oder R1 indirekt adressierbar.

Beispiel 4.1

Beispiel für die indirekte Adressierung

Es soll der Hex-Wert 34 in die Speicherzelle128 geschrieben werden.

Zunächst muss z. B. in den Register R0 die Speicheradresse geschrieben werden (Adresse 128 ist 80hex).

```
MOV R1, #80h
```

Anschließend kann über R1 der Wert in die gewünschte Speicherzelle geschrieben werden. Hierzu wird @ vor das entsprechende Register geschrieben.

```
MOV @R1, #34h
```

4.3 Spezial-Funktions-Register

Die Spezial-Funktions-Register enthalten alle Register, die der Controller für seine Arbeit, für die Ein- und Ausgabe und für interne Einstellungen benötigt.

Auf die Bedeutung der Register oder einzelner Bits wird eingegangen, wenn sie für die Realisierung bestimmter Funktionen benötigt werden.

Die Spezial-Funktions-Register befinden sich im Adressbereich 80h bis FFh. Sie sind über direkte Adressierung anzusprechen. Sie lassen sich mit ihrer absoluten Hexadresse benennen oder, bei einem geeigneten Übersetzerprogramm, auch mit ihrer symbolischen Adresse. Einige SFR sind nur als komplettes Byte anzusprechen, andere sind bitadressierbar. Bei geeignetem Übersetzerprogramm sind auch diese einzelnen Bits mit ihrem symbolischen Namen zu benennen.

Beispiele:

Setzen des Timer-Modus-Registers auf den Wert 0Fh (auf die Funktion der Register soll hier nicht eingegangen werden):

Symbolische Adresse des SFR: TMOD
Absolute Adresse des SFR: 89h
nicht bitadressierbar

```
MOV TMOD,#0F
```

oder

```
MOV 89,#0F
```

Freigabe des Timer 0 durch Setzen des Timer Run Flags TR0 auf 1:

Symbolische Adresse des SFR: TCON
Absolute Adresse des SFR: 88h
bitadressierbar
Symbolische Adresse des Bit: TR0
Absolute Adresse des Bit: 8Ch

```
SETB TR0
```

oder

```
SETB 8C
```

Liste der SFR-Register

Block	Symbol	Funktion	Adresse	Bitadr.
CPU	ACC	Akkumulator	E0h	ja
	B	B-Register	F0h	ja
	PSW	Programmstatuswort	D0h	ja
	SP	Stack Pointer	81h	
	DPL	Data Pointer, Low-Byte	82h	
	DPH	Data Pointer, High-Byte	83h	
Ports	P0	Port 0	80h	ja
	P1	Port 1	90h	ja
	P2	Port 2	A0h	ja
	P3	Port 3	B0h	ja
	P4	Port 4	C0h	ja
Timer	TH0	Timer-0-Register, High-Byte	8Ch	
	TL0	Timer-0-Register, Low-Byte	8Ah	
	TH1	Timer-1-Register, High-Byte	8Dh	
	TL1	Timer-1-Register, Low-Byte	8Bh	
	TH2	Timer-2-Register, High-Byte	CDh	
	TL2	Timer-2-Register, Low-Byte	CCh	
	TCON	Steuerungsregister für T0 und T1	88h	ja
	TMOD	Modusauswahl für T0 und T1	89h	
	T2CON	Timer-2-Steuerregister	C8h	ja
	RCAP2H	Timer-2-Reload/Capture, High-Byte	CBh	
	RCAP2L	Timer-2-Reload/Capture, Low-Byte	CAh	
	WDTRST	WatchDog Timer Reset	A6h	
	WDTPRG	WatchDog Timer Programm	A7h	
Serielle Schnittstelle	SCON	Steuerungsregister der seriellen Schnittstelle	98h	ja
	SBUF	Buffer der seriellen Schnittstelle	99h	
	SADEN	Maske der Slave Adresse	B9h	
	SADDR	Slave Adresse	A9h	

Block	Symbol	Funktion	Adresse	Bitadr.
PCA	CCON	PCA-Timer-Steuerungsregister	D8h	ja
	CMOD	PCA-Timer Modusauswahl	D9h	
	CL	PCA-Timer, Low-Byte	E9h	
	CH	PCA-Timer, High-Byte	F9h	
	CCAPM0	PCA-Timer- Modus 0	DAh	
	CCAPM1	PCA-Timer- Modus 1	DBh	
	CCAPM2	PCA-Timer- Modus 2	DCh	
	CCAPM3	PCA-Timer- Modus 3	DDh	
	CCAPM4	PCA-Timer- Modus 4	DEh	
	CCAP0H	PCA Compare Capture Module 0H	FAh	
	CCAP1H	PCA Compare Capture Module 1H	FBh	
	CCAP2H	PCA Compare Capture Module 2H	FCh	
	CCAP3H	PCA Compare Capture Module 3H	FDh	
	CCAP4H	PCA Compare Capture Module 4H	FEh	
	CCAP0L	PCA Compare Capture Module 0L	EAh	
	CCAP1L	PCA Compare Capture Module 1L	EBh	
	CCAP2L	PCA Compare Capture Module 2L	ECh	
	CCAP3L	PCA Compare Capture Module 3L	EDh	
	CCAP4L	PCA Compare Capture Module 4L	EEh	
Interrupt-System	IEN0	Interrupt-Freigaberegister 0	A8h	ja
	IEN1	Interrupt-Freigaberegister 1	E8h	ja
	IPL0	Interrupt-Priorität-Steuerung Low 0	B8h	ja
	IPH0	Interrupt-Priorität-Steuerung High 0	B7h	
	IPL1	Interrupt-Priorität-Steuerung Low 1	F8h	ja
	IPH1	Interrupt-Priorität-Steuerung Low 1	F7h	
A/D-Wandler	ADCON	A/D-Wandler-Steuerregister	F3h	
	ADCF	A/D-Wandler- Konfiguration	F6h	
	ADCLK	A/D-Wandler Takt	F2h	
	ADDH	A/D-Wandler Ergebnis High-Byte	F5h	
	ADDL	A/D-Wandler Ergebnis Low-Byte	F4h	
SPI	SPCON	SPI-Steuerregister	D4h	
	SPSCR	SPI-Status- und Steuerregister	D5h	
	SPDAT	SPI - Daten	D6h	
System	PCON	Steuerung Stromaufnahme	87h	
	AUXR	Auxiliary Register 0	8Eh	
	AUXR1	Auxiliary Register 1	A2h	
	CKCON0	Taktsteuerung 0	8Fh	
	CKCON1	Taktsteuerung 1	9Fh	
	FCON	Flash-Steuerung	D1h	
	EECON	EEPROM Steuerung	D2h	
	FSTA	Flash-Status	D3	

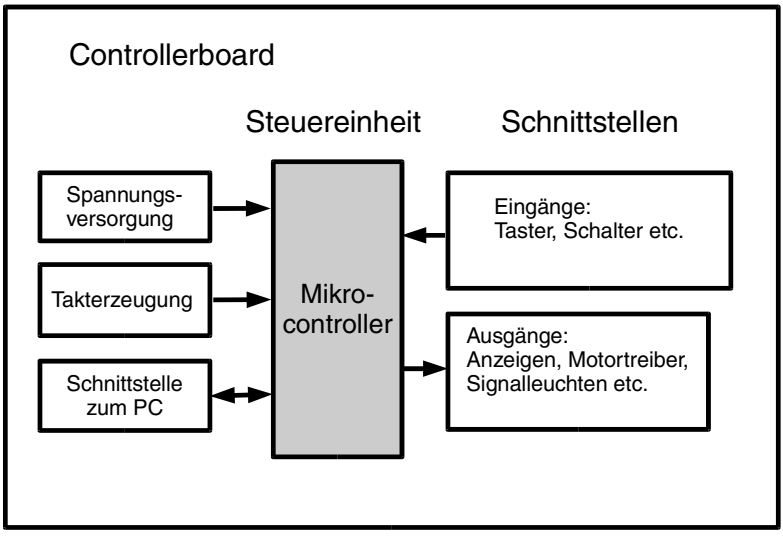
Übung 4.1

1. Wie lässt sich der interne Speicher des Controllers gliedern? Wie lassen sich die einzelnen Speichergruppen ansprechen? Machen Sie eine Skizze.
2. Registerbänke:
   Wie viele sind vorhanden? Wozu werden sie eingesetzt? Wie erfolgt die Umschaltung der Registerbänke?
3. Wie viele direkt adressierbare Bits lassen sich speichern? Unter welchen Adressen?
4. Wie viele direkt adressierbare Bytes lassen sich neben den Registern und dem Bitspeicher im internen RAM ablegen?
5. In welchem Adressbereich liegen die „Spezial-Funktions-Register“? Wie werden sie adressiert?
6. Wie viele Bytes lassen sich in der oberen Hälfte des internen RAMs speichern? Wie werden sie adressiert?

5 Konstruktion eines Controllerboards

Um elektrische Kleinsteuerungen zu realisieren, wird eine Platine benötigt, die den Mikrocontroller mit einer zuverlässige Spannungsversorgung und einem Systemtakt versorgt. Zudem müssen die entsprechenden Portpins des Controllers so verschaltet werden, dass die gewünschten Schnittstellen, wie z. B. Taster, Schalter, Anzeigen, Motortreiber und Schützschaltungen ohne Zerstörung des Controllers angesteuert und abgefragt werden können.

Wenn ein Entwicklungsboard entwickelt wird, werden zum Programmieren und Testen noch eine Schnittstelle zum PC benötigt, Taster und Schalter als Eingänge zur Simulation von realen Sensoren bzw. Benutzereingaben und Signalleuchten zur Anzeige von Ausgangssignalen.



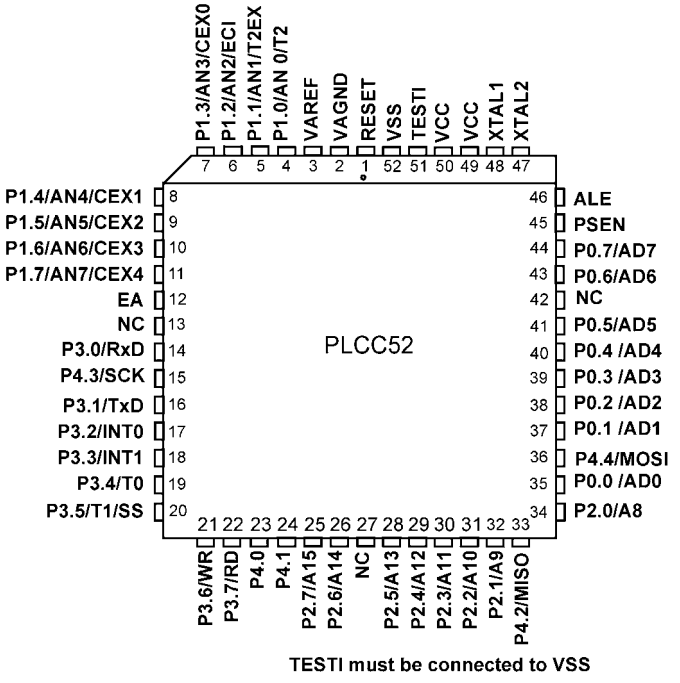
Um die Schnittstelle zum Computer zu schaffen, hat der Atmel AT89C51AC3 einen Bootloader über die serielle Schnittstelle zur direkten „In-System-Programmierung“, so dass für die Kommunikation mit dem PC nur ein Treiberbaustein für die Pegelanpassung auf die zu erstellende Platine nötig ist. Ein passendes Betriebssystem ist schon mit dem Bootloader im Chip integriert, so dass sich im Chip ein kleines Entwicklungssystem befindet. Der Controller kann dann ohne von der Platine genommen zu werden direkt programmiert und im fertigen System getestet werden. Bei Controllern ohne ISP (In Sytem Programming)-Schnittstelle muss das Programm z. B. in einen EPROM-Baustein (teilweise mit im Chip integriert) mithilfe eines Programmiergerätes geschrieben werden, der dann anschließend über den externen Bus (falls der

Speicher nicht auf dem Chip implementiert ist) an den Controller angeschlossen wird. Wenn man Programme direkt im System testen will, ohne den EPROM-Speicherchip immer zu wechseln, so benötigte man ein Hardware-Entwicklungssytem. Im Programmspeicher muss dann ein besonderes Betriebssystem mithilfe des Mikrocontrollers mit dem PC kommunizieren. Die Programmdaten werden dann entweder oberhalb des Adressbereiches vom Betriebssystem geschrieben oder es kann zwischen dem Betriebssystemspeicher und dem Progammspeicher mit einem Schalter auf dem Board umgeschaltet werden. Bei der Umschaltung zwischen Betriebssystem und Programmspeicher kann ab Adresse 0000h der Speicher beschrieben werden, ansonsten wird oberhalb der Adresse vom Betriebssystem das Programm gespeichert und gestartet.

5.1 Steuereinheit

Die Anschlüsse des Controllers

In diesem Teil werden exemplarisch am Mikrocontroller Atmel AT89C51AC3 die relevanten Daten aus dem „Users Manual“ dargestellt.



Pin Name	Type	Description
VSS	GND	**Circuit ground**
TESTI	I	**Must be connected to VSS**
VCC		**Supply Voltage**
VAREF		Reference Voltage for ADC
VAGND		Reference Ground for ADC
P0.0:7	I/O	**Port 0**: Is an 8-bit open drain bi-directional I/O port. Port 0 pins that have 1's written to them float, and in this state can be used as high-impedance inputs. Port 0 is also the multiplexed low-order address and data bus during accesses to external Program and Data Memory. In this application it uses strong internal pull-ups when emitting 1's. Port 0 also outputs the code Bytes during program validation. External pull-ups are required during program verification.
P1.0:7	I/O	**Port 1**: Is an 8-bit bi-directional I/O port with internal pull-ups. Port 1 pins can be used for digital input/output or as analog inputs for the Analog Digital Converter (ADC). Port 1 pins that have 1's written to them are pulled high by the internal pull-up transistors and can be used as inputs in this state. As inputs, Port 1 pins that are being pulled low externally will be the source of current (IIL, see section „Electrical Characteristic“) because of the internal pull-ups. Port 1 pins are assigned to be used as analog inputs via the ADCCF register (in this case the internal pull-ups are disconnected). As a secondary digital function, port 1 contains the Timer 2 external trigger and clock input; the PCA external clock input and the PCA module I/O. P1.0/AN0/T2 Analog input channel 0, External clock input for Timer/counter2. P1.1/AN1/T2EX Analog input channel 1, Trigger input for Timer/counter2. P1.2/AN2/ECI Analog input channel 2, PCA external clock input. P1.3/AN3/CEX0 Analog input channel 3, PCA module 0 Entry of input/PWM output. P1.4/AN4/CEX1 Analog input channel 4, PCA module 1 Entry of input/PWM output. P1.5/AN5/CEX2 Analog input channel 5, PCA module 2 Entry of input/PWM output. P1.6/AN6/CEX3 Analog input channel 6, PCA module 3 Entry of input/PWM output. P1.7/AN7/CEX4 Analog input channel 7, PCA module 4 Entry ot input/PWM output. Port 1 receives the low-order address byte during EPROM programming and program verification. It can drive CMOS inputs without external pull-ups.

Pin Name	Type	Description
P2.0:7	I/O	**Port 2**: Is an 8-bit bi-directional I/O port with internal pull-ups. Port 2 pins that have 1's written to them are pulled high by the internal pull-ups and can be used as inputs in this state. As inputs, Port 2 pins that are being pulled low externally will be a source of current (IIL, see section Ëlectrical Characteristic") because of the internal pull-ups. Port 2 emits the high-order address byte during accesses to the external Program Memory and during accesses to external Data Memory that uses 16-bit addresses (MOVX @DPTR). In this application, it uses strong internal pull-ups when emitting 1's. During accesses to external Data Memory that use 8 bit addresses (MOVX @Ri), Port 2 transmits the contents of the P2 special function register. It also receives high-order addresses and control signals during program validation. It can drive CMOS inputs without external pull-ups.
P3.0:7	I/O	**Port 3**: Is an 8-bit bi-directional I/O port with internal pull-ups. Port 3 pins that have 1's written to them are pulled high by the internal pull-up transistors and can be used as inputs in this state. As inputs, Port 3 pins that are being pulled low externally will be a source of current (IIL, see section Ëlectrical Characteristic") because of the internal pull-ups. The output latch corresponding to a secondary function must be programmed to one for that function to operate (except for TxD and WR). The secondary functions are assigned to the pins of port 3 as follows: P3.0/RxD: Receiver data input (asynchronous) or data input/output (synchronous) of the serial interface P3.1/TxD: Transmitter data output (asynchronous) or clock output (synchronous) of the serial interface P3.2/INT0: External interrupt 0 input/timer 0 gate control input P3.3/INT1: External interrupt 1 input/timer 1 gate control input P3.4/T0: Timer 0 counter input P3.5/T1/SS: Timer 1 counter input SPI Slave Select P3.6/WR: External Data Memory write strobe; latches the data byte from port 0 into the external data memory P3.7/RD: External Data Memory read strobe; Enables the external data memory. It can drive CMOS inputs without external pull-ups.
P4.0:4	I/O	**Port 4**: Is an 2-bit bi-directional I/O port with internal pull-ups. Port 4 pins that have 1's written to them are pulled high by the internal pull-ups and can be used as inputs in this state. As inputs, Port 4 pins that are being pulled low externally will be a source of current (IIL, on the datasheet) because of the internal pull-up transistor. The secondary functions are assigned to the 5 pins of port 4 as follows: P4.0: Regular Port I/O

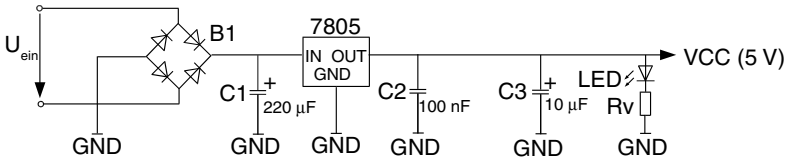
Pin Name	Type	Description
		P4.1: Regular Port I/O P4.2/MISO: Master Input Slave Output of SPI controller P4.3/SCK: Serial Clock of SPI controller P4.4/MOSI: Master Ouput Slave Input of SPI controller It can drive CMOS inputs without external pull-ups.
RESET	I/O	**Reset**: A high level on this pin during two machine cycles while the oscillator is running resets the device. An internal pull-down resistor to VSS permits power-on reset using only an external capacitor to VCC.
ALE	O	**ALE**: An Address Latch Enable output for latching the low byte of the address during accesses to the external memory. The ALE is activated every 1/6 oscillator periods (1/3 in X2 mode) except during an external data memory access. When instructions are executed from an internal Flash (EA = 1), ALE generation can be disabled by the software.
PSEN	O	**PSEN**: The Program Store Enable output is a control signal that enables the external program memory of the bus during external fetch operations. It is activated twice each machine cycle during fetches from the external program memory. However, when executing from of the external program memory two activations of PSEN are skipped during each access to the external Data memory. The PSEN is not activated for internal fetches.
EA	I	**EA**: When External Access is held at the high level, instructions are fetched from the internal Flash. When held at the low level, AT89C51AC3 fetches all instructions from the external program memory.
XTAL1	I	**XTAL1**: Input of the inverting oscillator amplifier and input of the internal clock generator circuits. To drive the device from an external clock source, XTAL1 should be driven, while XTAL2 is left unconnected. To operate above a frequency of 16 MHz, a duty cycle of 50 % should be maintained.
XTAL2	O	**XTAL2**: Output from the inverting oscillator amplifier.

5.2 Die Beschaltung des Controllers

Spannungsversorgung

Um den Controller mit Spannung zu versorgen, wird eine stabilisierte 5 V Gleichspannung benötigt. Diese Spannung kann entweder direkt aus einem stabilisierten Netzteil erfolgen oder sie wird mit einem Spannungsregler und einer eventuellen Gleichrichterschaltung erzeugt.

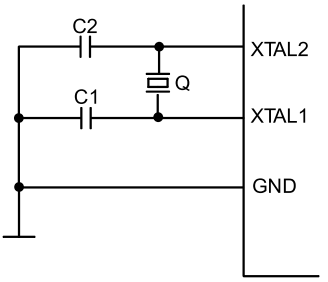
Meistens ist es sinnvoll als Schutz eine entsprechende Schaltung vor den Mikrocontroller zu setzen.



Mit einer optionalen LED kann gezeigt werden, ob Spannung am Board anliegt. Die 5 V Spannung muss am Pin VCC und am Pin TESTI (falls vorhanden) des Controllers angeschlossen werden, Ground wird am Pin GND angeschlossen.

Takterzeugung

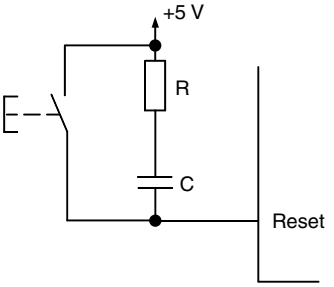
Für die Erzeugung des Systemtaktes dient ein integrierter Oszillator, der über die Anschlüsse XTAL1 und XTAL2 extern beschaltet werden muss. XTAL1 ist dabei der Eingang und XTAL2 ist der Ausgang eines invertierenden Verstärkers. Mithilfe eines Quarzes und zwei Kondensatoren kann die Schwingung erzeugt werden.



Soll der Systemtakt über ein externes Taktsignal eingespeist werden, so wird nur der Eingang XTAL1 beschaltet.

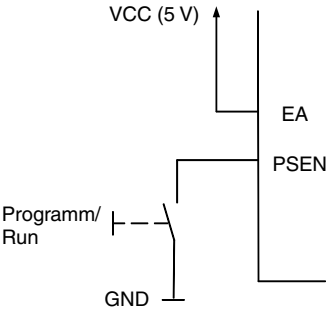
Reset

Über den Pin Reset kann der Controller zurückgesetzt werden. Dies ist erforderlich, um nach der Programmierung zum Beispiel das Programm zu starten oder den Controller für die Übertragung des Programmes vorzubereiten. Der Reset wird durchgeführt, wenn der Pin für zwei Systemtakte auf 5 V gezogen wird. Durch einen zusätzlichen externen Kondensator am Reset-Eingang lässt sich der Controller automatisch starten.



Interner und externer Programmspeicher

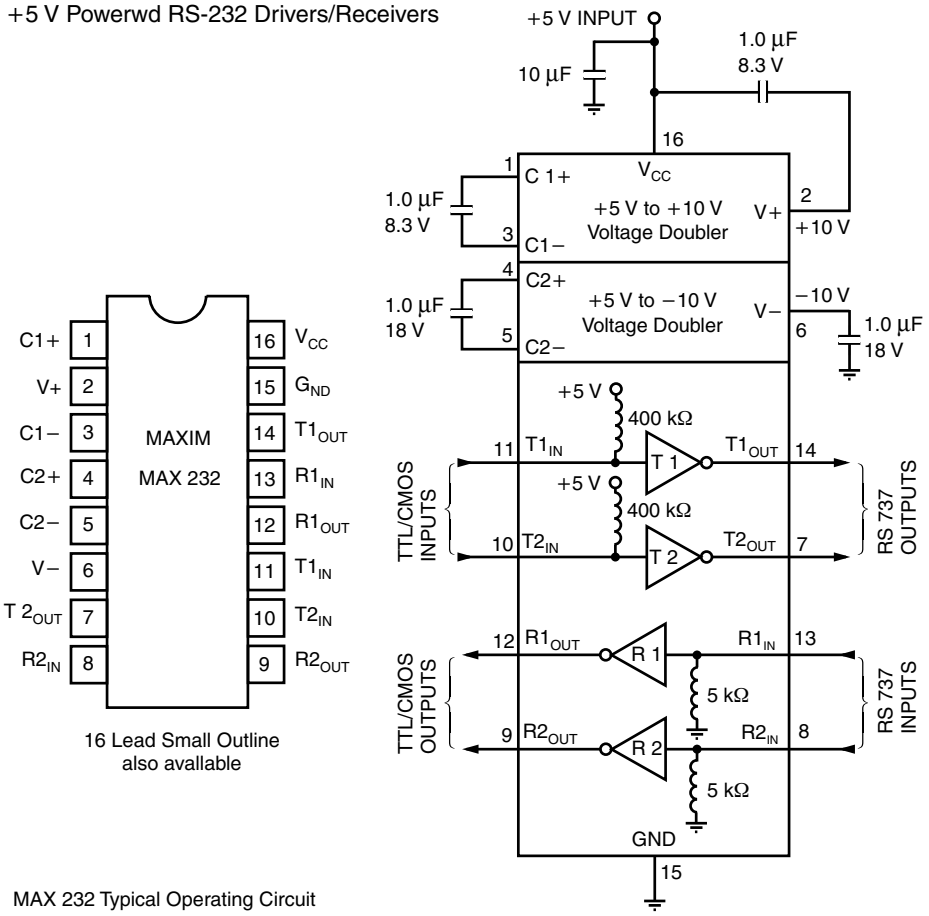
Über den Pin EA (External Access) kann zwischen dem externen Programmspeicher und dem internen Programmspeicher umgeschaltet werden. Wird der Pin am „High-Level“ (5 V) angeschlossen, so wird der interne Programmspeicher verwendet. Wird ein „Low-Level“ (Ground) angeschlossen, so wird ein externer Programm-Speicher verwendet.



PSEN gibt die Steuersignale für den externen Programmspeicher aus. Wenn der interne Programmspeicher verwendet wird, dann muss während der Programmierung des internen Flash-Speichers der Pin auf Ground (0 V) liegen. Aufgrund dessen sollte hier ein Schalter angeschlossen werden. Es kann zusätzlich noch eine LED angeschlossen werden, um zu überprüfen, ob sich der Controller im Programmier- oder Run-Modus befindet.

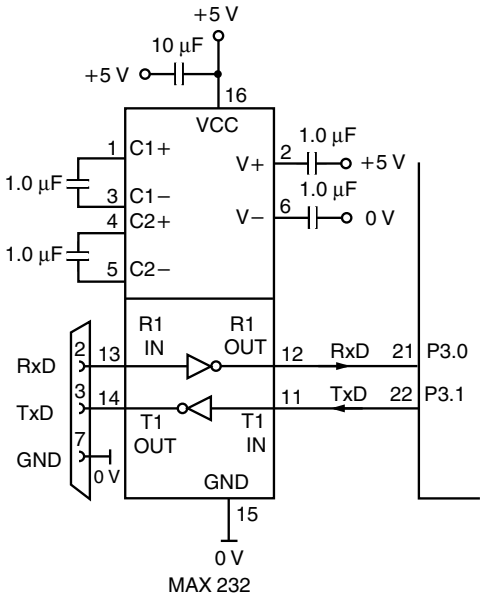
Schnittstelle zum PC

Mithilfe der seriellen Schnittstelle wird eine Verbindung zwischen Mikrocontroller und PC hergestellt. Das Problem liegt allerdings darin, dass die Datenübertragung der seriellen Schnittstelle mit Signalpegeln von -12 V und $+12$ V arbeitet. Die serielle Schnittstelle des Controllers liefert aber TTL-Pegel von $+5$ V und 0 V. Für die Pegelwandlung und Verstärkung liefert die Firma Maxim einen speziellen Treiberbaustein, der nur eine Betriebsspannung von 5 V benötigt und die höheren Übertragungsspannungen intern selbst erzeugt.



MAX 232 Typical Operating Circuit

An den Pins 13 und 14 des MAX232 wird die serielle Schnittstelle vom PC kommend angeschlossen. Pin 12 und 11 werden entsprechend des Schaltplans mit den Portpins P3.0 und P3.1 des Mikrocontrollers verbunden.



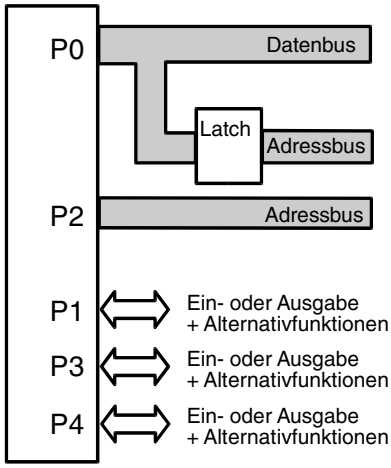
5.3 Schnittstelle

Der Controller AT89C51AC3 hat vier digitale Ports. Beim Port 4 werden nur die ersten 5 Bits (großes Gehäuse) bzw. 2 Bits (kleines Gehäuse) nach außen geführt. Der Port 1 kann zusätzlich auch analoge Signale einlesen.

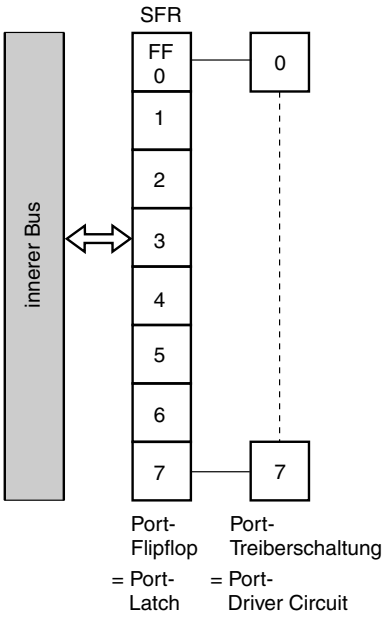
Wird ein externes Bussystem aufgebaut, sind dafür die Ports P0 und P2 zu verwenden. Dann bleiben als Ein- und Ausgabeports nur P1, P3 und P4 übrig.

Alle Ports lassen sich bitweise zur Ein- und Ausgabe digitaler Signale nutzen. Die Ports P1 und P3 haben darüber hinaus auch noch weitere Alternativfunktionen.

Jedem Port ist ein Spezial-Funktions-Register (SFR) zugeordnet, welches 8 einzelne Bits steuert oder abfragt.



Die Flipflops der SFR steuern die Port-Treiber. Ausgaben erfolgen immer über das dem Port zugeordnete SFR-Register!

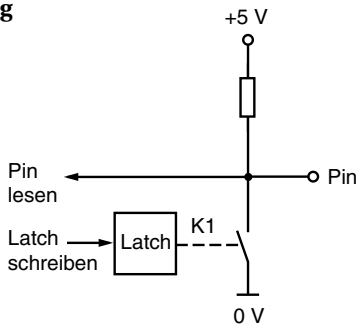


Die Schaltung der Ports

An den Portanschlüssen lassen sich Daten einlesen und ausgeben. An einer vereinfachten Schaltung soll gezeigt werden, wie das an den Portpins realisiert wird.

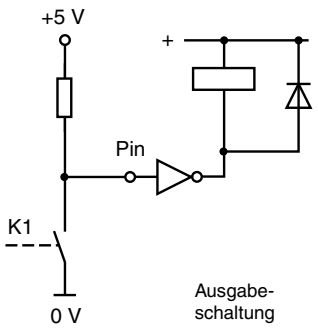
Port-Treiberschaltung, vereinfachte Darstellung

Der Schalter K1 wird invertiert angesteuert.
Latch = 0 → K1 geschlossen.
Latch = 1 → K1 geöffnet.



Datenausgabe

Latch = 0; K1 geschlossen; Pin = 0
Latch = 1; K1 geöffnet ; Pin = 1

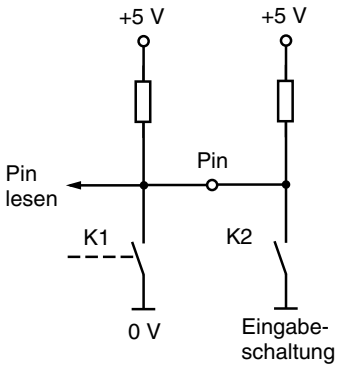


Dateneingabe

K2 geschlossen; Pin = 0
K2 geöffnet ; Pin = 1

Die Daten können jedoch nur eingelesen werden, wenn das Latch 1-Signal hat und damit K1 öffnet! Hat das Latch 0-Signal, wird der Portpin auf 0 festgehalten. Ein 1-Signal der Eingabeschaltung wird nach 0 kurzgeschlossen.

Wichtig:
Soll ein Portpin als Eingabe genutzt werden, ist das Latch durch eine Ausgabe auf 1 zu setzen!



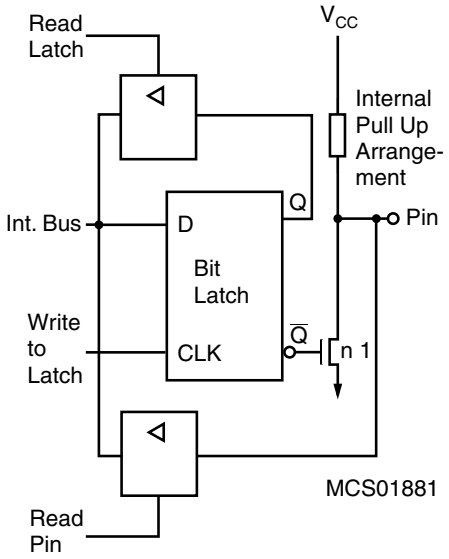
Basisschaltung der Ports

Datenausgabe

Wird ein Befehl zur Ausgabe eines Bits ausgeführt, liegt das Signal auf dem internen Datenbus. Das Steuerwerk aktiviert den Impuls „Write to Latch“, und das Bit wird im Latch gespeichert. Das Latch steuert den Treiber, der das Signal am Portpin ausgibt.

Zwischen dem Ausgang des Latches und dem Treibereingang befindet sich allerdings noch ein UND-Gatter, mit dem der Ausgang über spezielle alternative Funktionen gesteuert werden kann. Hierzu muss am Port-Latch-Ausgang ein 1-Signal anliegen. Diese Alternativfunktionen werden zum Beispiel bei der Puls-Weiten-Modulation benötigt, an dem der Portpin durch eine spezielle eigenständige Funktionseinheit schnell geschaltet wird ohne die CPU zu belasten.

Ausgangsbeschaltung der Ports P1, P3 und P4:



Dateneingabe

Wird ein Befehl zum Einlesen eines Pin-Signales ausgeführt, erzeugt das Steuerwerk den Impuls „Read Pin“ und das Signal, welches am Portpin anliegt, geht auf den internen Datenbus.

Zudem kann für die zusätzlichen Funktionseinheiten über den Abgriff „Alternate Input Funktion“ der Portpin eingelesen werden. Dies wird zum Beispiel bei dem A/D-Wandler benötigt.

Port 0 und Port 2

Port 0 und Port 2 stellen eine Besonderheit dar, weil diese für den externen Daten und Adressbus verwendet werden können. Die Ports lassen sich aber ähnlich handhaben wie die übrigen Ports. Es ist nur eine zusätzliche interne Beschaltung integriert, um diese für den Adress- und Datenbus zu nutzen, der Anwender merkt hiervon nichts.

Achtung:
Port 0 hat zudem, wenn dieser nicht für den Adress- und Datenbus genutzt wird, keine internen Pull-Up-Widerstände, so dass diese bei Bedarf extern beschaltet werden müssen. So können die Eingänge von Port 0 sehr hochohmig einlesen.

5.4 Die elektrischen Daten

Folgende Daten sind aus dem Datenblatt von dem Microcontroller Atmel AT89C51AC3 entnommen.

DC Parameters for Standard Voltage

$T_A = -40\,^{\circ}\text{C}$ to $+85\,^{\circ}\text{C}$; $V_{SS} = 0\,\text{V}$;
$V_{CC} = 2.7\,\text{V}$ to $5.5\,\text{V}$ and $F = 0$ to $40\,\text{MHz}$ (both internal and external code execution)
$V_{CC} = 4.5\,\text{V}$ to $5.5\,\text{V}$ and $F = 0$ to $60\,\text{MHz}$ (internal code execution only)

Symbol	Parameter	Min	Typ	Max	Unit	Test Conditions
V_{IL}	Input Low Voltage	-0.5		$0.2V_{CC} - 0.1$	V	
V_{IH}	Input High Voltage except XTAL1, RST	$0.2V_{CC} + 0.9$		$V_{CC} + 0.5$	V	
V_{IH1}	Input High Voltage, XTAL1, RST	$0.7V_{CC}$		$V_{CC} + 0.5$	V	
V_{OL}	Output Low Voltage, ports 1, 2, 3 and 4(1)			0.3 0.45 1.0	V V V	$I_{OL} = 100\,\mu\text{A}$ $I_{OL} = 1.6\,\text{mA}$ $I_{OL} = 3.5\,\text{mA}$
V_{OL1}	Output Low Voltage, port 0, ALE, PSEN (1)			0.3 0.45 1.0	V V V	$I_{OL} = 200\,\mu\text{A}$ $I_{OL} = 3.2\,\text{mA}$ $I_{OL} = 7.0\,\text{mA}$
V_{OH}	Output High Voltage, ports 1, 2, 3, and 4	$V_{CC} - 0.3$ $V_{CC} - 0.7$ $V_{CC} - 1.5$			V V V	$I_{OH} = -10\,\mu\text{A}$ $I_{OH} = -30\,\mu\text{A}$ $I_{OH} = -60\,\mu\text{A}$ $V_{CC} = 3\,\text{V}$ to $5.5\,\text{V}$
V_{OH1}	Output High Voltage, port 0, ALE, PSEN	$V_{CC} - 0.3$ $V_{CC} - 0.7$ $V_{CC} - 1.5$			V V V	$I_{OH} = -200\,\mu\text{A}$ $I_{OH} = -3.2\,\text{mA}$ $I_{OH} = -7.0\,\text{mA}$ $V_{CC} = 5\,\text{V} \pm 10\,\%$
R_{RST}	RST Pulldown Resistor	50	100	200	kΩ	
I_{IL}	Logical 0 Input Current, ports 1, 2, 3 and 4			-50	µA	$V_{in} = 0.45\,\text{V}$

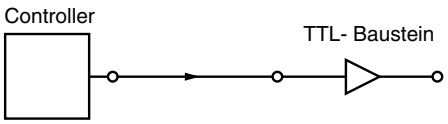
Symbol	Parameter	Min	Typ	Max	Unit	Test Conditions
I_{LI}	Input Leakage Current			±10	µA	0.45 V < V_{in} < V_{CC}
I_{TL}	Logical 1 to 0 Transition Current, ports 1, 2, 3 and 4			−650	µA	V_{in} = 2.0 V
C_{IO}	Capacitance of I/O Buffer			10	pF	F_c = 1 MHz T_A = 25 °C
I_{PD}	Power-down Current		75	150	µA	3 V < V_{CC} < 5.5 V
I_{CC}	Power Supply Current	I_{CCOP} = 0.4 Frequency (MHz) +8 I_{CCIDLE} = 0.2 Frequency (MHz) +8				V_{CC} = 5.5 V

Note:
Under steady state (non-transient) conditions, IOL must be externally limited as follows:
Maximum I_{OL} per Port-Pin: 10 mA
Maximum I_{OL} per 8-bit Port:
Port 0: 26 mA
Ports 1, 2, 3 and 4: 15 mA
Maximum total IOL for all output pins: 71 mA
If I_{OL} exceeds the test condition, V_{OL} may exceed the related specification. Pins are not guaranteed to sink current greater than the listed test conditions.

Port 1 bis 4 als Ausgang

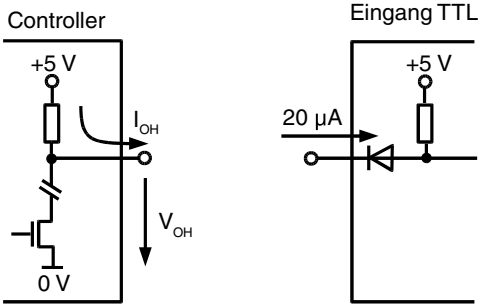
Wenn Port 1 bis Port 4 als Ausgang verwendet werden, so sind die maximalen Werte laut Datenblatt einzuhalten. Wichtig ist hierbei die Information in der Bemerkung (Note), in der genau die maximale Stromentnahme der Ports definiert ist. Zudem sollte geprüft werden, bis zu welcher Spannung das anzuschließende Bauteil sicher den High- und Low-Pegel schaltet.

Beispiel für ein TTL-Baustein



Sichere Erkennung des High-Pegels

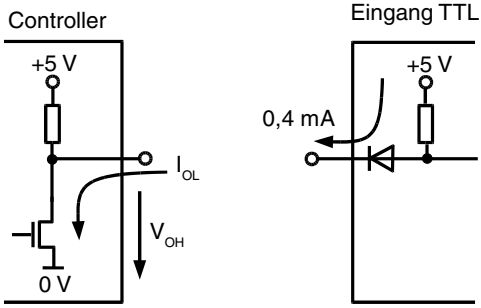
Wie im Datenblatt unter V_{OH} angegeben, wird bei einer Stromentnahme von 60 µA eine Spannung von $V_{CC} - 1{,}5$ V (hier: 3,5 V) garantiert. Bei kleineren Stromentnahmen ist die Spannung höher. Ein TTL High-Pegel benötigt mindestens 2 V. Da hier nur 20 µA benötigt werden, liegt die Spannung V_{OH} höher, so dass der High-Pegel sicher erkannt wird. Es lassen sich somit auch mehrere Eingänge parallel am Port anschließen.



Sichere Erkennung des LOW-Pegels

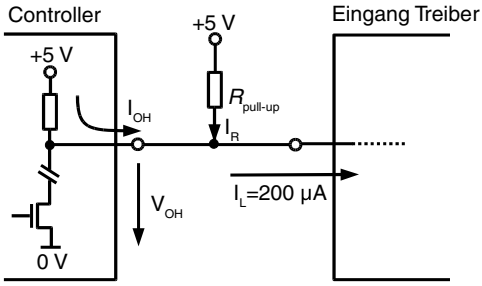
Im Datenblatt muss nun unter VOL geschaut werden. Hier steht, dass bei einem Strom von $I_{OL} = 1{,}6\,\text{mA}$ eine Spannung von maximal $V_{OH} = 0{,}45\,\text{V}$ am Ausgang anliegt. Bei kleineren Strömen ist die Spannung niedriger.

Da der TTL-Pegel bei unter 0,8 V als LOW definiert ist und hier nur ein Strom von 0,4 mA fließt, wird der LOW-Pegel sicher erkannt.



Ein Betrachten der Ströme und Schaltbilder zeigt, dass das Risiko für eine Zerstörung des Controllers nur beim LOW-Pegel besteht. Wenn mehrere Bauteile parallel geschaltet werden, darf ein maximaler Strom pro Port (8 Portpins) von 15 mA fließen, pro Portpin maximal 10 mA (siehe Bemerkungen zu den elektrischen Daten).

Bei manchen Treiberbausteinen kommt es vor, dass zum Schalten einer Last am Eingang des Bausteins ein höherer Strom benötigt wird, wie zum Beispiel bei dem Treiberbaustein ULN2803. Hier kann mit einem zusätzlichen externen Pull-UP-Widerstand Abhilfe geschaffen werden.



Beispiel:

Zum Schalten wird eine Spannung von 3,5 V und ein Strom von mindestens 200 µA benötigt.

geg.: $V_{OH} = 3{,}5\,\text{V}$; $I_{OH} = 60\,\mu\text{A}$; $I_L = 200\,\mu\text{A}$

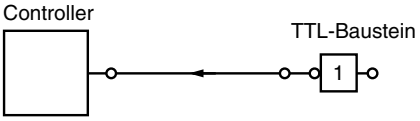
$$\begin{aligned} R_{\text{pull-up}} &= \frac{5\,\text{V} - V_{OH}}{I_L - I_{OH}} \\ &= \frac{5\,\text{V} - 3{,}5\,\text{V}}{200\,\mu\text{A} - 60\,\mu\text{A}} \\ &= 10{,}7\,\text{k}\Omega \end{aligned}$$

Aus der Rechnung ergibt sich, dass mit einem 10-kΩ-Pull-Up-Widestand ein Strom von mindestens 200 µA erreicht wird. Zu Beachten ist dann aber, dass nun beim Abschalten ein zusätzlicher Strom fließt. In diesem Beispiel fließt für den Fall, dass kein Strom über den Treiberbaustein fließen würde und $V_{OH} = 0\,\text{V}$ wären ein zusätzlicher Strom von 500 µA. Wenn alle 8 Portpins beschaltet werden, fließt dann insgesamt ein Strom von 4 mA über den gesamten Port, was kein Problem darstellt. Erst bei niederohmigeren Pull-Up-Widerständen können die kritischen Grenzwerte erreicht werden.

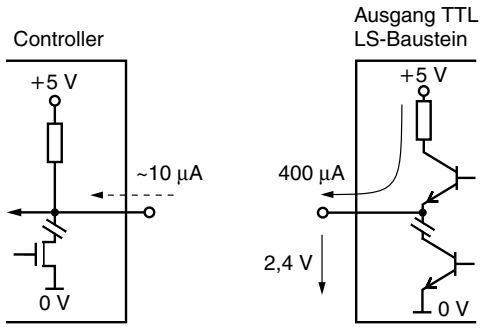
Port 1 bis 4 als Eingang

Es sind die Grenzwerte eingetragen.

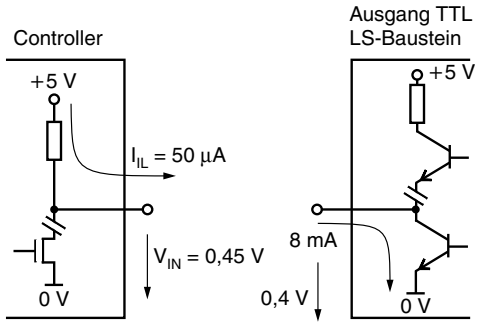
Als Beispiel dient ein TTL Inverter.



Der TTL-Ausgang liefert 1-Signal:



Der TTL-Ausgang liefert 0-Signal:

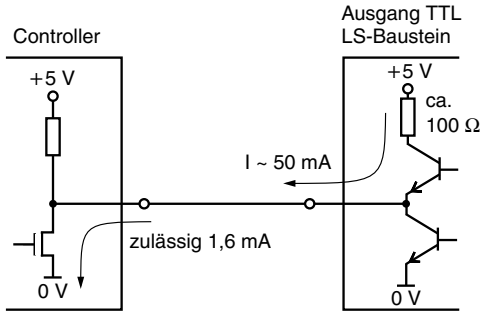


Die Signalpegel des TTL-Bausteins werden vom Controller sicher als High- und Low-Signal erkannt. Der Ausgang des TTL-Bausteins kann wesentlich mehr Strom nach 0 V ziehen, als für den Controller-Anschluss erforderlich.

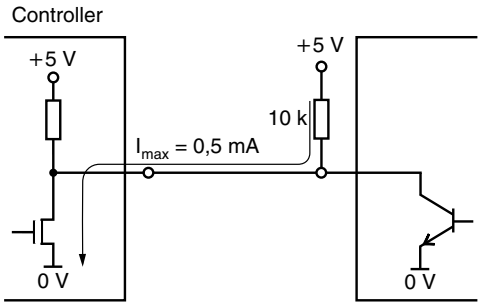
Achtung:

Es muss auch der Fall berücksichtigt werden, dass ein Programmierfehler passiert und der Portlatch ein Low-Signal hat. Aufgrund dessen wird folgende Betrachtung gemacht.

Voraussetzung bei der Lastbetrachtung war, dass der Controller richtig als Eingangsport programmiert wurde, d. h. das Portlatch 1-Signal hat und damit der Ausgangstransistor des Controllers sperrt. Wurde in das Latch eine 0 programmiert, leitet der Ausgangstransistor des Controllers, und es fließt ein viel zu hoher Strom, da der Kollektorwiderstand des TTL-Ausgangs so niederohmig ist. Der Controller-Ausgang wird zerstört!



Um den Controller bei falscher Programmierung vor Zerstörung zu bewahren, ist ein Baustein mit Open Kollektor zu verwenden. Ein Kollektorwiderstand von 10 k ist extern anzuschließen.

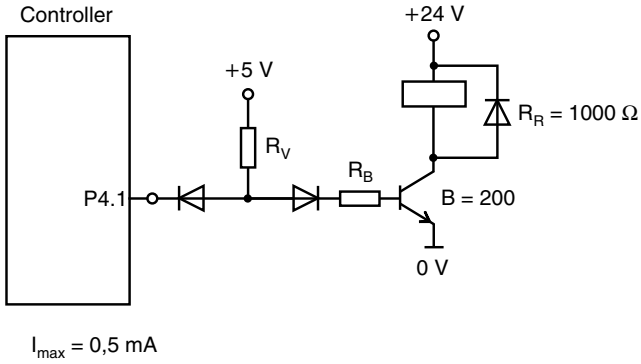


Übung 5.1

Ein Eingangssignal ist über ein Flipflop zu entprellen und an Port 4.0 anzuschließen. Zeichnen Sie die Schaltung. Wie ist das Portlatch zu programmieren, wenn der Port als Eingang genutzt werden soll?

Übung 5.2

An Portpin 4.1 ist ein 24-V-Relais anzuschließen. Dazu verwenden Sie folgende Schaltung:



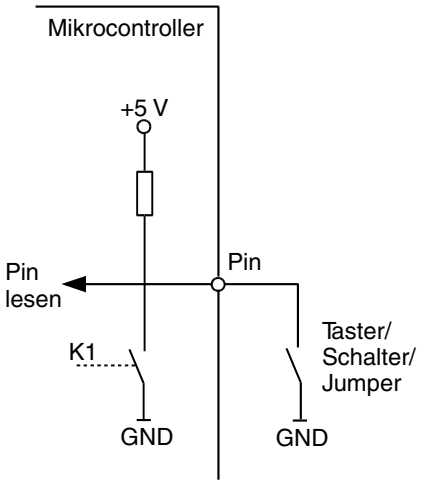
Der Transistor hat eine Verstärkung von 200. Er soll vom Basisstrom zweifach übersteuert werden. Das Relais hat einen Innenwiderstand von 1 kΩ. Berechnen Sie die Widerstände R_V und R_B. Am Portpin des Controllers soll der Strom $I_{OL} = 1{,}6\,\text{mA}$ fließen.

5.5 Hardware zum Testen

In diesem Abschnitt soll eine mögliche Hardwaretestumgebung aufgezeigt werden, um die zu erstellenden Programme mit dem Mikrocontroller zu prüfen.

Eingänge

Zum Testen der Programme wird der Port 1 für Eingänge verwendet. Hierzu können Schalter, Taster oder eine einfache Jumperleisten genutzt werden, die den entsprechenden Portpin des Controllers auf Masse schalten. Die im Controller verbauten Pull-UP Widerstände sorgen dafür, dass bei Nichtbeschaltung ein High-Pegel am Pin-Eingang anliegt. Bei der Beschaltung auf Masse liegt ein LOW-Pegel an.



Ein als Schließer eingesetzter Taster oder Einschalter macht also, wenn er betätigt wird, ein LOW-Signal.

Ein als Öffner eingesetzter Signalgeber bringt bei Betätigung ein High-Signal.

Daraus ergibt sich, dass die Eingänge beim Taster/Schließer invertiert eingelesen werden.

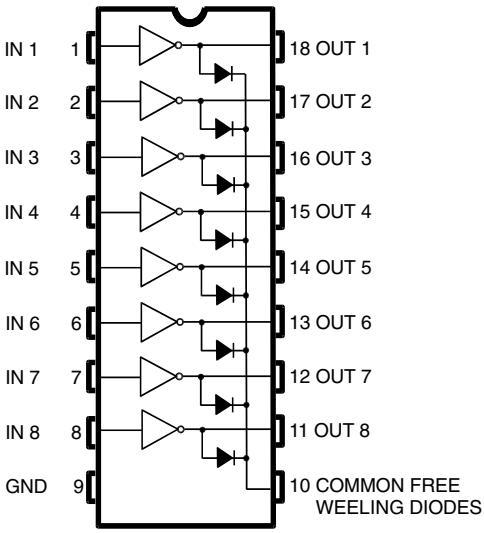
Ausgänge

Als Ausgang soll Port 2 dienen, der zum Beispiel über den Treiberbaustein ULN2803A Aktoren und LEDs schaltet.

Dieser IC ist weit verbreitet und kann Lasten von bis zu 500 mA schalten.

Die Eingänge sind direkt gegenüber den Ausgängen angeordnet, damit eine leichte Verschaltung möglich ist. Zudem sind die Ausgänge zur Beschaltung von induktiven Lasten mit Dioden abgesichert.

Treiber ULN 2803A:

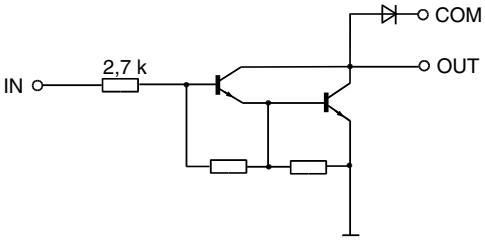


Der Treiber ULN 2803A

Damit der Treiberbaustein korrekt am Mikrocontroller angeschlossen wird, soll dieser kurz vorgestellt werden.

Beim ULN 2803A handelt es sich um eine integrierte Schaltung, die acht Darlington Transisitoren beinhaltet. Diese Transistoren können einen Strom von bis zu 500 mA auf Masse schalten und brauchen hierzu nur einen kleinen Eingangsstrom. Damit die Transistoren bei 5 V Eingangsspannung schalten können, ohne zerstört zu werden, ist ein Vorwiderstand von $R_V = 2{,}7\,k\Omega$ integriert.

Innere Beschaltung:



Anschluss an den Controller

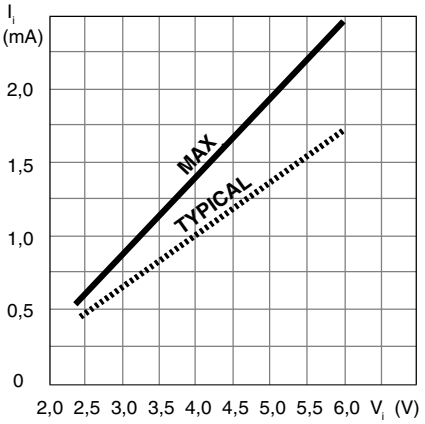
Wenn nun ein Ausgang am Controller einen Eingang des Treibers schalten soll, so muss ein genügend hoher Strom fließen. Der Strom ist abhängig von der zu schaltenden Last.

Bei einer Last von 100 mA (I_C) muss ein Eingangsstrom von 250 µA (I_i) fließen, um sicher durchzuschalten. (Siehe Kennlinie I_i zu I_C)

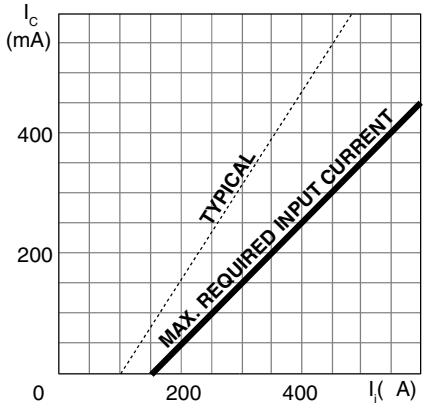
Dies wird schon bei kleinen Eingangsspannungen U_i erreicht (siehe Kennlinie U_i zu I_i). Wenn bei einer angenommenen Ausgangsspannung von 2,5 V beim Mikrocontroller ein Ausgangsstrom von 250 µA überschritten wird, schaltet der Treiber also sicher durch. Dies kann mit dem integrierten Pull-Up-Widerstand allerdings nicht garantiert werden. Laut Datenblatt fließt bei der Ausgangsspannung von 3,5 V ein Strom von 60 µA.

Um die Stromentnahme sicherzustellen, sollte ein zusätzlicher Pull-Up-Widerstand extern beschaltet werden.

Eingangsspannung U_i zu Eingangsstrom I_i (ULN 2803A)



Eingangsspannung U_i zu Eingangsstrom I_i

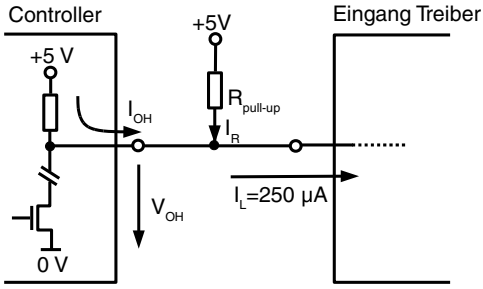


Der Pull-Up-Widerstand lässt sich dann wie folgt berechnen:

I_{OH} wird hier nicht berücksichtigt, da der Wert nicht bekannt ist. Der zur Verfügung stehende Strom wird dadurch nur größer, wodurch der Transistor mehr übersteuert wird, was nicht schadet.

$$R_{\text{pull-up}} = \frac{5\,\text{V} - V_{OH}}{I_L - I_{OH}} = \frac{5\,\text{V} - 2{,}5\,\text{V}}{250\,\mu\text{A}} = 10\,\text{k}\Omega$$

Schaltung mit externen Pull-Up-Widerstand:



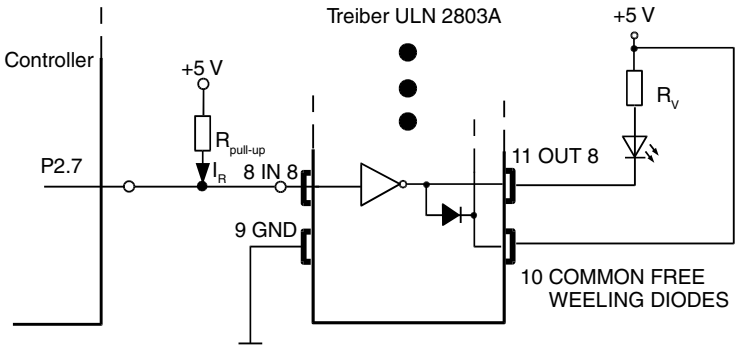
Wenn die benötigte Spannung am Eingangstreiber niedriger ist, so fällt mehr Spannung am Pull-Up-Widerstand ab und der Strom nimmt zu. Zudem fließt noch der zusätzliche Strom I_{OH}, der größer als 60 µA ist. Dann lassen sich sogar im typischen Fall höhere Lasten als 300 mA schalten (siehe Kennlinienfeld I_i zu I_C).

Aufzupassen ist nur, dass der Pull-Up-Widerstand nicht zu klein wird, dann fließt nämlich unter Umständen ein zu hoher Strom durch den Controller, wenn der Portpin „Low"-Signal hat.

$$I = \frac{5\,\text{V}}{R_{\text{pull-up}}} = \frac{5\,\text{V}}{10\,\text{k}\Omega} = 500\,\mu\text{A}$$

In diese Fall fließt ein unkritischer Strom von 500 µA pro Portpin.

Eine beispielhafte Schaltung für Port 2.7, die über den Treiberbaustein eine LED ansteuert, sieht dann wie folgt aus:



Die anderen Ports werden dann gleich beschaltet. Natürlich kann anstelle der LED ein stärkerer Verbraucher angeschlossen werden, wie z. B. ein Relais.

Ein weiterer großer Vorteil dieser Treiberstufe liegt in der Invertierung. Wenn der Port mit High angesteuert wird, dann wird der entsprechende Aktor auch eingeschaltet.

Übung 5.3

Sie wollen mit dem Treiberbaustein ULN 2803A eine Last mit einem Stromverbrauch von 300 mA sicher schalten. Dimensionieren Sie den Pull-Up-Widerstand.

6 Methode der Programmentwicklung

Ein Gerät soll mithilfe eines Mikrocontrollers gesteuert werden. In diesem Gerät muss sich das Mikrocontrollerboard befinden mit einem Steuerprogramm in Maschinensprache.

Zur Entwicklung des Maschinenprogramms dient meist eine integrierte Entwicklungsumgebung (IDE) auf dem PC. Je nach Entwicklungsumgebung kann zum Beispiel das Programm in Assembler, in der Hochsprache C, in Basic oder auch in einer anderen Programmiersprache geschrieben werden. Das benötigte Maschinenprogramm kann in der IDE erzeugt, getestet und über ein entsprechendes Programm in den Mikrocontroller übertragen werden.

6.1 Erzeugen des Maschinencodes

Damit der Prozessor des Controllers die Programmanweisungen versteht, muss das geschriebene Programm in die entsprechende Maschinensprache übersetzt werden. Nach dem Übersetzen werden meist drei Dateien erstellt.

Das reine Maschinenprogramm, Objektprogramm genannt, besteht aus Bitmustern, die sich nicht als ASCII-Zeichen auf dem Bildschirm oder Drucker darstellen lassen. Bei einer Übertragung dieser Datei zu einem anderen Gerät besteht keine Möglichkeit, auf Übertragungsfehler zu prüfen.

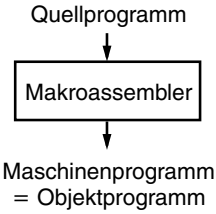
Deshalb werden Maschinenprogramme, die angezeigt und übertragen werden sollen, im Intel-Hex-Format erstellt. Das Programm wird dabei in ASCII-Zeichen hexadezimal dargestellt. Es enthält die Anfangsadresse des Programms sowie die Anzahl der Datenbytes mit einer Checksumme. Damit wird eine Fehlererkennung bei der Übertragung möglich.

Drei relevante Dateien:

1. Ein komplettes Programmlisting:
   `Name.LST`
2. Ein Maschinenprogramm im Intel-Hex-Format:
   `Name.HEX`
3. Ein reines Maschinenprogramm:
   `Name.OBJ`

Assembler

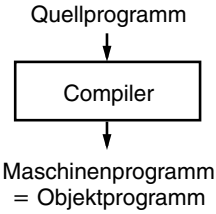
Beim Assemblerprogramm geschieht die Umsetzung in ein Maschinencode durch den Makroassembler. Die Anweisungen des Makroassemblers ähneln den Assemblerbefehlen des Mikrocontrollers und werden deshalb Pseudobefehle genannt. Die Programmierung ist somit sehr Hardware-nah. Der Makroassembler bietet dem Programmierer allerdings eine Menge Hilfen an.



Es lassen sich z. B. absoluten Adressen oder Daten symbolische Namen zuweisen. Ziele von Sprungbefehlen lassen sich mit symbolischen Bezeichnungen als Sprungmarken angeben. Die relative Sprungweite wird dann beim Übersetzen berechnet. Register und einzelne Bits lassen sich symbolisch adressieren. Zahlen sind in verschiedenen Codes darstellbar. Texte lassen sich in ASCII-Zeichen eingeben und werden automatisch in ihr Bitmuster übersetzt.

Hochsprachen

Bei den übrigen Programmiersprachen wird das geschriebene Programm mithilfe eines Compilers in die Maschinensprache übersetzt. Teilweise ist es möglich, dass der Compiler als Zwischencode aus der Hochsprache eine Assemblerdatei erzeugt.



Der Vorteil einer Hochsprache liegt darin, dass Befehlssequenzen mit umgangssprachlichen Konstrukten beschrieben werden können, die meist noch nicht einmal auf den Prozessortyp bezogen sind. Zudem bieten Hochsprachen wesentlich komplexere Datenstrukturen verglichen mit dem Assembler, z. B. Arrays, Strings und Strukturen. Die Programmierung ist wesentlich intuitiver.

Der Nachteil gegenüber der Assemblerprogrammierung liegt darin, dass meist der erzeugte Maschinencode größer ist. Im Laufe der Weiterentwicklungen wird der erzeugte Code aber immer optimierter und effizienter, sodass kaum noch etwas am erzeugten Code optimiert werden kann. Ein weiteres Problem liegt darin, dass zeitkritische Anwendungen nicht exakt programmiert werden können, da die Umsetzung des zu erzeugenden Maschinencodes vom Programmierer nicht beeinflussbar ist. Deshalb ist es auch möglich, in Hochsprachen zum Beispiel kleinere Unterprogramme in Assembler zu schreiben.

6.2 Übertragen des Maschinencodes auf das Mikrocontrollerboard

Nach der Programmentwicklung auf dem PC muss das Maschinenprogramm in der Hardwareumgebung getestet werden. Dies geschieht, indem entweder über ein Terminalprogramm oder ein spezielles In-System-Programm die erzeugte HEX-Datei vom PC zum Controller geschickt wird. Der Controller schreibt dann mithilfe seines Betriebssystems die Daten in den RAM- bzw. Flashspeicher. Anschließend kann der Programmspeicher in den RUN-Modus geschaltet und das Programm in der Hardwareumgebung getestet werden. Alternativ kann das Programm mit einem EPROMMER in das EPROM (entweder einzeln oder im Controller integriert) gebrannt werden, welches anschließend in das Board des Zielsystems gesteckt wird.

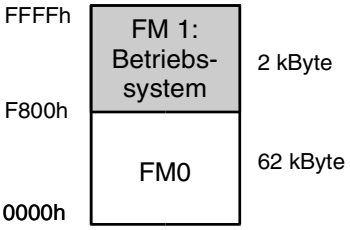
Beim Atmel AT89C51AC3-Mikrocontroller kann die Übertragung in den internen Flash-Speicher entweder über die In-System-Schnittstelle geschehen oder parallel über ein Programmiergerät.

Wird die Übertragung über die In-System-Schnittstelle durchgeführt, so benötigt der Controller ein Betriebssystem, welches über ein Hardwarebyte aktiviert werden kann. Bei Aktivierung wird das Betriebssystem ab Adresse F800h in den Flash-Speicher geschrieben. Dieses Betriebssystem benötigte 2KByte, sodass dieser Speicherbereich dann nicht mehr für das Programm zur Verfügung steht.

Bei der Parallelprogrammierung steht der komplette Flash-Speicher für das Programm zur Verfügung.

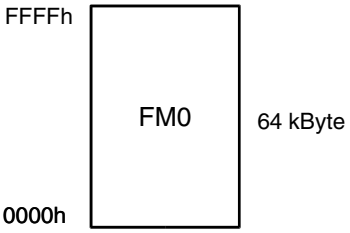
Programmspeicher mit In-System-Programmierung:

BLJB=0



Programm-Speicher mit Parallelprogrammierung:

BLJB=1



In-System-Programmierung des Controllers AT89C51AC3

Zur In-System Programmierung wird das von Atmel kostenlos zur Verfügung gestellte Programm „Atmel Flip“ genutzt. Hierzu muss zunächst über die serielle Schnittstelle eine Verbindung zum Controller aufgebaut werden. Besteht diese Verbindung, so zeigt das Programm zum Beispiel die Bootloader-Version des entsprechenden Controllers an. Zudem lässt sich das sogenannte Hardware-Byte einstellen. In diesem wird definiert, wie lang die Befehlszykluszeit sein soll, oder an welcher Stelle der Controller nach einem Reset das Programm starten soll. Wird die In-System-Programmierung gewählt, so beginnt das Programm (PSEN = Low-Signal)

nach einem Reset ab der Speicherstelle F800 hexadezimal. Ab dieser Stelle steht, wie oben dargestellt, das interne Betriebssystem des Controllers. Wenn der Controller parallel programmiert wird, beginnt das Programm nach einem Reset ab der Speicherstelle 0000 hexadezimal.

Eine Besonderheit dieses Atmel Controllers liegt zusätzlich darin, dass dieser über das Hardwarebit X2 entweder auf eine Befehlszykluszahl von 12 Takten eingestellt werden kann, wie es bei dem ursprüngliche 8051er-Controllertypen war, oder auf einen verkürzten Zyklus von nur 6 Takten. Wird eine verkürzte Zykluszahl gewählt, so arbeitet der Controller doppelt so schnell, wie der ursprüngliche 8051er-Controller.

In diesem Buch wird immer die ursprüngliche Maschinenzykluszahl von 12-Takten vorausgesetzt.

Sind alle Daten eingestellt, so muss der HEX-File geladen werden. Anschließend wir über den RUN-Button der Hex-File und die entsprechenden Einstellungen in den Mikrocontroller geladen.

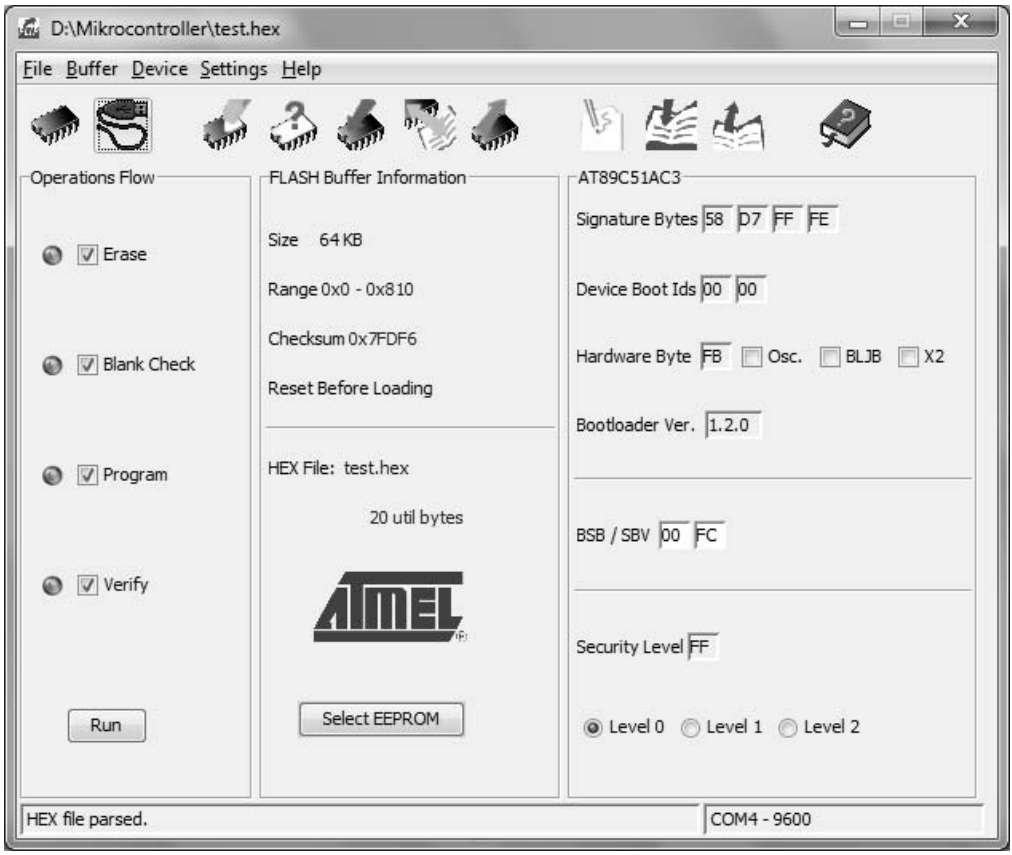
Register des Hardware Security Bytes (HSB)

HSB

X2	BLJB	-	-	-	LB2	LB1	LB0
7							0

Bit	Funktion
X2	X2=1 (Auslieferungszustand): Standard-Modus: 12 Takte pro Maschinenzyklus X2=0: 6 Takte pro Maschinenzyklus
BLJB	**Boot Loader Jump Bit:** **BLJB=1 (Auslieferungszustand):** Wird bei der Paralleprogrammierung benötigt. Nach dem Reset des Controllers ist die Startadresse des Flash-Speichers 0000h (Programm Counter: 0000h) (ENBOOT=0) **BLJB=0:** Wird bei der In-System-Programmierung benötigt. Nach dem Reset des Controllers ist die Startadresse des Flash-Speichers F800h (Programm Counter: F800h) (ENBOOT=1) Ab dieser Adresse steht das Betriebssystem.
LB2-0	**General Memory Lock Bits:** Diese werden nur für die Parallelprogrammierung benötigt.

Atmel Flip-Programm



6.3 Strukturiertes Programmieren

Um ein Steuerungsprogramm zu entwickeln, sollte strukturiert vorgegangen und eine ordentliche Dokumentation angefertigt werden. Dann lässt sich auch später noch das Programm nachvollziehen und kleine Änderungen lassen sich gezielt in das bestehende Programm einpflegen.

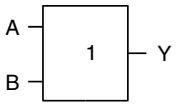
Hierzu sollen zwei Möglichkeiten dargestellt werden. Entweder wird das Programm, wie es in der Automatisierungstechnik üblich ist, mit logischen Verknüpfungssteuerungen entworfen, oder es werden sogenannte Programmablaufpläne entwickelt. In beiden Fällen kann zunächst unabhängig von der gewählten Programmiersprache eine Lösung gefunden werden. Diese Lösung lässt sich dann einfach mit der entsprechenden Programmiersprache umsetzen.

Logische Verknüpfungssteuerungen

Logische Verknüpfungssteuerungen sind aus der Digitaltechnik bekannt. Einzelne Bits lassen sich über entsprechende Gatter miteinander verknüpfen. Das zu erstellende Programm enthält dann die notwendige Logik, die zyklisch vom Mikrocontroller abgearbeitet wird.

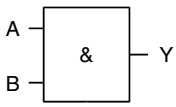
Zur Übersicht und zum Nachschlagen sind die wichtigsten Verknüpfungen mit deren Funktionstabelle im Folgenden dargestellt.

Oder-Verknüpfung:



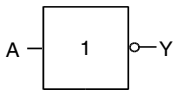
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

UND-Verknüpfung:



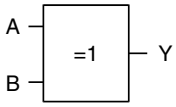
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

NICHT-Verknüpfung / Inverter:



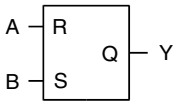
A	Y
0	1
1	0

XOR-Verknüpfung:



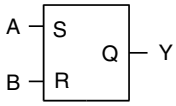
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

SR-Speicher (setzdominant):



S	R	Q
0	0	speichern
0	1	rücksetzen
1	0	setzen
1	1	**setzen**

RS-Speicher (rücksetzdominant):



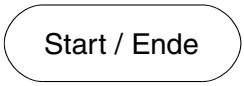
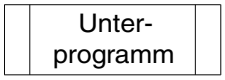
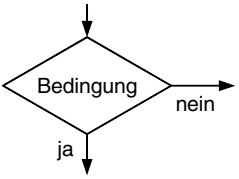
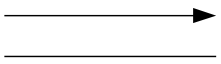
S	R	Q
0	0	speichern
0	1	rücksetzen
1	0	setzen
1	1	**rücksetzen**

Der Programmablaufplan

Der Programmablaufplan ist eine andere sehr flexible Möglichkeit, Programmstrukturen darzustellen. Grafisch lassen sich die Algorithmen und Programme mit Folgen von Operationen, Verzweigungen, Ausgaben, Eingaben und Unterprogrammen beschreiben. Die benötigten Symbole sind in der DIN 66 001 (ISO 5807) genormt.

Besonders in der Mikrocontrollerprogrammierung eignen sich derartige Pläne sehr, da in beliebigen Kombinationen und Reihenfolgen die Symbole angewendet werden können, und somit eine sehr flexible Möglichkeit der Programmentwicklung gegeben ist, die im Gegensatz zu den Verknüpfungssteuerungen alle möglichen Strukturen des Programms darstellen kann.

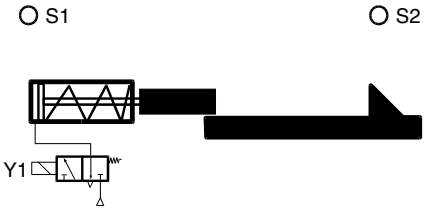
Es sind folgende für die Programmierung wichtigen Symbole definiert:

Bedeutung	Symbol
Oval: Ein Programm wird mit dem Start- und Stoppzeichen begonnen bzw. beendet. Zudem werden weiter Grenzpunkte mit diesem Zeichen dargestellt, z. B. eine Unterbrechung des Programms.	
Rechteck: Mit dem Rechteck werden alle möglichen Verarbeitungen gekennzeichnet, wie Operationen oder Ein- und Ausgaben. In der alten Norm gab es zusätzliche Zeichen für die Ein- und Ausgabe, die hier aber nicht verwendet werden sollen.	
Rechteck mit vertikalen doppelten Linien: Ein Rechteck mit vertikalen doppelten Linien kennzeichnet ein Aufruf eines Unterprogramms.	
Raute: Eine Raute stellt eine Verzweigung oder Fallunterscheidung dar. Bei einer Verzweigung stehen an den jeweiligen Verbindungen die Entscheidungen ja/nein.	
Pfeil/Linie: Mit einem Pfeil oder einer Linie wird die Verbindung zum nächstfolgenden Element gekennzeichnet.	

Programmbeispiel Holzspalter

Als Beispiel für die Entwicklung eines Programms soll die Steuerung eines Holzspalters dienen.

Technologieschema

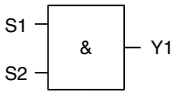


Ein Hydraulikstössel zum Spalten des Holzes fährt aus, wenn sowohl der Taster S1 als auch der Taster S2 betätigt wird (Zweihandbetätigung zur Vermeidung von Unfallgefahren).

Die Lösung soll einmal mithilfe von Verknüpfungssteuerungen und einmal mithilfe des Programmablaufplans gezeigt werden.

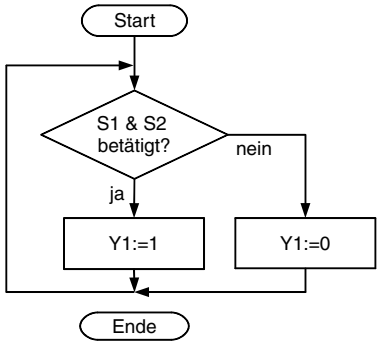
Lösung mit der Verknüpfungssteuerung

Durch eine UND-Verknüpfung von S1 und S2 lässt sich das Problem lösen:



1. Lösungsmöglichkeit mit dem Programmablaufplan

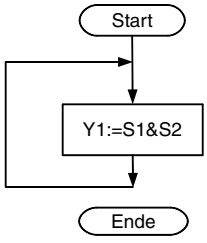
Nach dem Start des Programms wird abgefragt, ob S1 und S2 gleichzeitig betätigt sind. Wenn „ja“ wird das Magnetventil Y1 betätigt, wenn „nein“ dann wird es deaktiviert. Anschließend führt der Pfeil wieder zur Abfrage. Das Programm wird somit zyklisch durchlaufen und niemals beendet werden.



2. Lösungsmöglichkeit mit dem Programmablaufplan

In dieser 2. Lösungsmöglichkeit wird in nur einer Operation der Wert von Y1 zugeordnet. Die Funktion des Programms ist identisch, wird nur anders programmiert.

Alleine in diesem einfachen Beispiel wird deutlich, dass der Programmablaufplan sehr vielfältig eingesetzt werden kann, und auch unterschiedliche Lösungen hervorbringt, die dokumentiert werden sollten.



7 Programmierung in der Hochsprache C

In diesem Buch wird sowohl die Programmierung in Assembler als auch die Programmierung in der Hochsprache C beschrieben.

Da die Programmierung von Controllern immer mehr in der Hochsprache C durchgeführt wird, soll zunächst mit dieser begonnen werden. Zur Vertiefung der Architektur des Mikrocontrollers wird am Ende des Buches auf die Programmierung in Assembler eingegangen.

7.1 Die Programmiersprache C

Die Programmiersprache C ist sehr umfangreich, sodass in diesem Buch nur die grundlegenden Elemente von C beschrieben und erläutert werden können. Diese reichen aber in der Regel aus, um einfache Steuerungsprogramme zu realisieren.

Leider unterscheiden sich die C-Compiler teilweise in der Syntax der Bitprogrammierung, da diese eine Besonderheit der 8051er-Mikrocontrollerfamilie sind und somit nicht genormt sind. In diesem Buch werden die Beispielprogramme mit der IDE µVision von Keil erstellt. Sie beinhaltet den verbreiteten C-Compiler C51 und den Assembler A51. Eine aktuelle Demoversion ist beim Anbieter Keil aus dem Internet zu beziehen. Diese ist lediglich auf ein 2-KByte-HEX-File beschränkt, welches aber für die hier beschriebenen Projekte ausreicht.

Bei der Verwendung eines anderen Compilers müssen lediglich die Deklarationen der Bitverarbeitung angepasst werden.

Grundlegender Aufbau eines C-Programms

Ein C-Programm beginnt, falls notwendig, mit dem Einbinden von Bibliotheken, welche entweder Standardbibliotheken sind oder auch selbst erstellte. In diesen sind häufig benötigte Routinen und Definitionen enthalten, die nicht jedes mal neu programmiert werden sollen. So gibt es meist für jeden Mikrocontroller eine Bibliothek, in der zum Beispiel die Adressen der Spezialfunktionsregister symbolischen Variablen zugeordnet sind.

Im Anschluss können globalen Variablen definiert werden, die für alle Programmteile inklusive der Unterprogramme zur Verfügung stehen sollen. Dann kommt die Hauptfunktion main mit dem eigentlichen Programm. Da ein C-Programm immer in Funktionen gegliedert ist, wird auch das Hauptprogramm als Funktion definiert. Diese Funktion muss einmal vorkommen. In

C besitzt jede Funktion einen Datentyp für den Rückgabewert, der vor dem Namen der Funktion steht. Soll sie keinen Wert zurückgeben, ist das Schlüsselwort void stattdessen einzusetzen. Unter dem Funktionskopf folgen in einem Block, begrenzt durch geschweifte Klammern, die entsprechenden Anweisungen. Das Hauptprogramm beinhaltet immer eine Endlosschleife „`while(1)`“ in der zyklisch die programmierten Operationen abgearbeitet werden.

Kommentare können mit `//` eingeleitet werden. Damit wird die gesamte Zeile als Kommentar behandelt. Sollen mehrere Zeilen als Kommentar dienen, so kann ein ganzer Block mit `/*` begonnen und mit `*/` beendet werden.

An der Umsetzung des Steuerungsprogramms des Holzspalters aus Kapitel 6 soll der Grundaufbau eines C-Programms erläutert werden:

```
/*
     Steuerungsprogramm für den Holzspalter:
     Der Zylinder soll nur ausfahren, wenn gleichzeitig
     S1 und S2 betätigt werden.
     Umsetzung der Verknüpfungssteuerung.

     Datei: holzspalter.c
*/

#include<t89C51ac2.h>   // Einbinden der Bibliothek des Controllers.
                        // Da in der aktuellen Version noch nicht der
                        // aktuelle Controller ac3 eingebunden ist, wird
                        // hier und im Folgenden die Bibliothek des
                        // Vorgängers verwendet.

// Symbolische Zuordnung der Ports:

sbit S1=P1^0;           // S1 wird zum Beispiel dem Port P1.0 zugeordnet.
sbit S2=P1^1;           // S2 wird zum Beispiel dem Port P1.1 zugeordnet.
sbit Y1=P2^0;           // Y1 wird zum Beispiel dem Port P2.0 zugeordnet.

// Hauptprogramm:

void main()             // Der Funktionskopf des Hauptprogramms.
{                       // Beginn des Blocks vom Hauptprogramm.

     while (1)          // Endlosschleife für die zyklische
                        // Programmbearbeitung.
     {                  // Beginn des Blocks der zyklischen
                        // Programmbearbeitung.

          Y1=S1&&S2;    // Operation der UND-Verknüpfung.

     }                  // Ende Block Endlosschleife.
}                       // Ende Block main.
```

Hinweis: In diesem Programm wurde für ein besseres Verständnis nicht berücksichtigt, dass alle Eingänge und Ausgänge am Mikrocontroller invertiert angesteuert werden (siehe Beschaltung Kapitel 5). Zudem ist die Drahtbruchsicherheit nicht beachtet worden.

7.2 Grundlagen von C

Dieser Abschnitt stellt die gebräuchlichen Datentypen, Definitionen und Befehle in C für die Mikrocontrollerprogrammierung dar. Weitere Ergänzungen können in der jeweiligen Online-Hilfe des Compilers entnommen werden.

Datentypen

Gebräuchliche Datentypen bei Controllern sind: int, unsigned int, char, unsigned char und float. Zeichenketten (strings) besitzen keinen eigenen Datentyp und werden als Feld (array) vom Datentyp char deklariert. Die entsprechenden Wertebereiche sind rechts in der Tabelle dargestellt.

Datentyp	Bits	Zahlenbereich
char	8	$-128 \ldots 127$
unsigned char	8	$0 \ldots 255$
int	16	$-32768 \ldots 32767$
unsigned int	16	$0 \ldots 65536$
float	32	$-10^{38} \ldots 10^{38}$ Genauigkeit: 8 Stellen

Datentypen der Bitverarbeitung

Eine Sonderrolle spielen die Datentypen sfr, bit und sbit. Sie wurden speziell für die 8051-Controller zur Verfügung gestellt. Beim Wechsel des Compilers muss der Quellcode dementsprechend angepasst werden. In der nebenstehenden Tabelle sind die speziellen Datentypen für die Bitverarbeitung des verwendeten C51-Compilers der Keil IDE µVision zusammengestellt.

Für die Bitverarbeitung beim C51-Compiler der Keil IDE µVision:

Datentyp	Bits	Zahlenbereich
bit	1	0 oder 1
sbit	1	0 oder 1
sfr	8	$0 \ldots 255$

Konstanten- und Variablendefinitionen und -deklarationen

Konstanten und Variablen müssen im Programm festgelegt werden. Dabei wird jedem Namen ein jeweilige Datentyp zugeordnet. Bei einer direkten Wertfestlegung wird dies Deklaration genannt, ansonsten Definition. Die verwendeten Namen können aus Buchstaben und Ziffern mit zusätzlichen Unterstrichen „_“ bestehen, wobei nicht mit einer Ziffer begonnen werden darf. Umlaute und Sonderzeichen sind ebenfalls nicht erlaubt.

Zudem unterscheidet C zwischen Groß- und Kleinbuchstaben.

Beispiele für Konstanten:

```
const int wert = 134;
const float wert_2 = 2.645;
```

Beispiele für Variablendefinitionen:

```
int wert_3;
unsigned int counter, i;
float temp, pres;
char a;
```

Beispiele für Variablendeklarationen:

```
int zaehler = 0;
float druck = 3.54;
char Temp = 'T';
```

Einige gebräuchliche C-Befehle

Zuweisungen von Werten

Mithilfe des einfachen Gleichheitszeichens (auch Zuweisungsoperator genannt) wird der linksstehenden Variablen der rechts stehende Wert zugeordnet.

Rechts vom Zuweisungsoperator können beliebige Variablen, Konstanten oder auch arithmetische Ausdrücke stehen.

Unterschiedliche Datentypen werden dabei automatisch umkonvertiert.

```
wert_1 = 134;
druck = ADDH/255.0*80;
x = y = z - 100;
```

Falls `druck` und `ADDH` Integerwerte sind, wird in der Rechnung mit Gleitpunktzahlen gerechnet, da 255.0 angegeben ist. Die Variable `druck` erhält aber wieder einen Integerwert.

Operatoren

Beschreibung:	**Umsetzung in C:**
Inkrementieren (den Wert um Eins erhöhen)	`i++;` identisch mit: `i = i+1;`
Dekrementieren (den Wert um Eins erniedrigen)	`i--;` identisch mit: `i = i-1;`
Addition	`a+b`
Subtraktion	`a-b`
Multiplikation	`a*b`
Division	`a/b`
Modulodivision (Der Rest der Division ist das Ergebnis)	`a%b`
Logisches Nicht (Auslesen der invertierten Variablen)	`!a`
Logische UND-Verknüpfung von Bits	`a&&b`
Logische ODER-Verknüpfung von Bits	`a||b`
Vergleicher (liefert 1, wenn die Bedingung erfüllt ist)	
< kleiner	`a<b`
> größer	`a>b`
<= kleiner gleich	`a<=b`
>= größer gleich	`a>=b`
== gleich	`a==b`
! = ungleich	`a!=b`
Byte A um b Stellen nach rechts schieben	`A>>b`
Byte A um b Stellen nach links schieben	`A<<b`

Bitweise Negierung (Von Bits und Bytes)	`~a`
Bytes bitweise UND-Verknüpfen	`A&B`
Bytes bitweise ODER-Verknüpfen	`A|B`
Bytes bitweise XOR-Verknüpfen	`A^B`

Programmierung von Wiederholungen/Schleifen

Kopfgesteuerte Schleife

Solange die Bedingung wahr ist, wird Block 1 ausgeführt.

Sollte beim erstmaligen Aufruf die Bedingung nicht erfüllt sein, wird Block 1 nicht ausgeführt und übersprungen.

```
while (Bedingung erfüllt)
{
    Block1;
}
```

Fußgesteuerte Schleife

Solange die Bedingung wahr ist, wird wie bei der kopfgesteuerten Schleife Block 1 ausgeführt.

Sollte jedoch beim erstmaligen Aufruf die Bedingung nicht erfüllt sein, wird trotzdem einmal Block 1 ausgeführt.

```
do
{
    Block1;
}
while (Bedingung erfüllt);
```

Zählergesteuerte Schleife

Es wird zuvor festgelegt, wie oft Block 1 aufgerufen werden soll. In dem nebenstehenden Beispiel 50 mal.

z. B.:

```
for (i=0; i<50; i++)
{
    Block1;
}
```

Abfragen/Kontrollstrukturen

If-Abfrage:
Wenn die Bedingung wahr ist, wird Block 1 ausgeführt.

```
if (Bedingung)
{
    Block1;
}
```

If-Else-Abfrage:
Wenn die Bedingung erfüllt ist, wird Block 1 ausgeführt, ansonsten Block 2.

```
if (Bedingung)
{
    Block1;
}
else
{
    Block2;
}
```

7.3 Programmieren in Funktionen

Die Programmiersprache C wird für eine bessere Übersichtlichkeit und für eine größtmögliche Strukturierung in Funktionen programmiert.

Funktionen sind in sich abgeschlossene Programmteile, die beliebig oft aus anderen Funktionen heraus aufgerufen werden können. Es ist möglich, den Funktionen Werte zum Bearbeiten zu übergeben. Zusätzlich können diese auch selber Werte zurückliefern.

Es ist sinnvoll, Funktionen bei der Programmierung zu verwenden, da
- gleiche Programmabschnitte an unterschiedlichen Teilen im Programm nur einmal programmiert werden müssen.
- das Programm in kleinere Teilprogramme zerlegt werden kann, und dadurch dieses besser verständlich wird.
- eine arbeitsteilige Programmierung möglich ist, in der Teams nur für wenige Funktionen zuständig sind.
- bereits programmierte Funktionen auch von anderen benutzt werden können, ohne dass diese wissen, wie die einzelne Funktionen in sich funktionieren.

Funktionen ohne Übergabe- und ohne Rückgabewert

Beim Aufruf wird der Block1 bearbeitet. Anschließend wird wieder an die vorherige Stelle des Programms zurückgesprungen.

Funktion:

```
void Funktion1()
{
    Block1;
}
```

Aufruf der Funktion:

```
Funktion1();
```

Funktionen mit einem Übergabe-, aber ohne Rückgabewert

In der Klammer nach dem Funktionsnamen muss dann eine lokale Variable definiert werden. Diese Variable steht anschließend in der Funktion zur Verfügung.

Funktion:

```
void Wertuebergabe (char Ausgabe)
{
  P2=Ausgabe; // Auf Port P2 wird das
              // Byte Ausgabe ausgegeben.
}
```

Beim Aufruf der Funktion wird entsprechend des Datentyps der zu verarbeitende Wert übergeben.

Aufruf der Funktion:

```
Wertuebergabe(0x3F); // Die HEX-Zahl 3F
                     // wird der Funktion
                     // wertuebergabe
                     // übergeben.
```

Funktionen mit Rückgabewert

Soll eine Funktion einen Wert zurückgeben, so muss der Datentyp des rückzugebenen Wertes vor dem Funktionsnamen angegeben werden.

Mithilfe des Befehls `return` wird der Wert dann an die Funktion übergeben.

Zusätzlich kann die Funktion Übergabewerte bekommen oder nicht.

Funktion:

```
int addition (int a, int b)
{
    int e;
    e=a+b;
    return e;
}
```

Aufruf der Funktion:

```
z=addition (5,3);
```

Dem Wert z wird der Wert 8 zugeordnet.

7.4 Binärkombinationen verwalten

Häufig kommt es vor, dass komplette Binärkombinationen auf einen Port, in ein Spezialfunktionsregister oder in einen Speicher geschrieben, oder einzelne Bits aus einem kompletten Byte gelesen werden sollen.

Eingabe der Binärkombinationen

In den meisten Compiler können allerdings keine Binärkombinationen, sondern nur Hexadeximalzahlen eingegeben werden. Dies hat den Vorteil, dass der Programmcode übersichtlicher bleibt und weniger Tippfehler passieren.

Beispiel:

Soll zum Beispiel die komplette Binärkombination 0010 1101 auf dem Port 2 ausgegeben werden, so werden die ersten vier Stellen zu einer HEX-Zahl zusammengefasst und anschließend die nächsten vier Stellen zur einer weiteren Hex-Zahl.

0010 $\rightarrow$ 2 (Hexadezimal)
1101 $\rightarrow$ D (Hexadezimal)

Der Befehl in C lautet dann:

```
P2=0x2D; // mit 0x werden HEX-Zahlen
         // in C gekennzeichnet
```

Tabelle HEX-Zahlen Binärzahlen und Dezimalzahlen:

Binärkombinationen				HEX-Zahl	Dezimalzahl
0	0	0	0	0	0
0	0	0	1	1	1
0	0	1	0	2	2
0	0	1	1	3	3
0	1	0	0	4	4
0	1	0	1	5	5
0	1	1	0	6	6
0	1	1	1	7	7
1	0	0	0	8	8
1	0	0	1	9	9
1	0	1	0	A	10
1	0	1	1	B	11
1	1	0	0	C	12
1	1	0	1	D	13
1	1	1	0	E	14
1	1	1	1	F	15

Maskierung

Wenn ein Byte nicht bitadressierbar ist, aber trotzdem einzelne Bits verändert werden sollen, so muss dies über eine sogenannte Maskierung geschehen.

Bits schreiben

1. Beispiel: Ein einzelnes Bit von 0 auf 1 setzen.

Byte Test hat folgende Binärkombination: 001 0 1101

Es soll nun das Bit 4 von 0 auf 1 gesetzt werden, ohne die anderen Bits zu verändern.

Umsetzung:

Das komplette Byte wird mit einer binären Oder-Verknüpfung mit einem Byte, welches nur an dem vierten Bit eine 1 hat, zusammengesetzt. Dann ändert sich nur Bit 4. Das Ergebnis wird dann in Test gespeichert.

```
   001 0 1101
>= 000 1 0000 (Oder-Verknüpfung)
   001 1 1101 (Ergebnis)
```

In C sieht das wie folgt aus:

```
Byte=Byte|0x10;
```

2. Beispiel: Ein einzelnes Bit von 1 auf 0 setzen.

Byte Test hat folgende Binärkombination: 00 1 0 1101

Es soll nun das Bit 5 von 1 auf 0 gesetzt werden, ohne die anderen Bits zu verändern .

Umsetzung:

Das komplette Byte wird mit einer binären UND-Verknüpfung mit einem Byte, welches nur an dem fünften Bit eine 0 hat, zusammengesetzt. Dann ändert sich nur Bit 5. Das Ergebnis wird dann in Test gespeichert.

```
   0010 1101
&  1101 1111 (UND-Verknüpfung)
   0000 1101 (Ergebnis)
```

In C sieht das wie folgt aus:

```
Byte=Byte&0xDF;
```

Bits lesen

Soll ein einzelnen Bits aus einer Binärkombination gelesen werden, so geschieht dies ebenfalls über eine Maskierung.

Beispiel:

Es soll zum Beispiel geprüft werden, ob Bit 3 des Bytes Test aktiviert ist.

Umsetzung:

Prüfen, ob Bit 3 = 0.

Das komplette Byte wird mit einer binären UND-Verknüpfung mit einem Byte, welches nur an dem dritten Bit eine 1 hat, verglichen. Ist Bit 3 gleich 0 gewesen, so ist das gesamte Ergebnis 0.

1010 1 0 10 (Bit 3 ist 0)
& 0000 0 1 00 (UND-Verknüpfung)
0000 0000 (Ergebnis = 0)

Prüfen, ob Bit 3 = 1.

Das komplette Byte wird mit einer binären UND-Verknüpfung mit einem Byte, welches nur an dem dritten Bit eine 1 hat, verglichen. Ist Bit 3 gleich 1 gewesen, so ist das gesamte Ergebnis ungleich 0.

1010 1 1 10 (Bit 3 ist 1)
& 0000 0 1 00 (UND-Verknüpfung)
0000 0100 (Ergebnis <> 0)

In C sieht dies wie folgt aus:

1. Es soll ein Programmblock 1 durchlaufen werden, wenn Bit 3 gleich 0 ist:

```
if ((Test&0x04)==0)
{
    //Block 1
}
```

2. Es soll ein Programmblock 1 durchlaufen werden, wenn Bit 3 gleich 1 ist:

```
if (!((Test&0x04)==0))
{
    //Block 1
}
```

8 C-Programme für Controller-Grundfunktionen

Alle in diesem Buch vorgestellten Programme sind mit dem Mikrocontroller Atmel AT89C51AC3 und der IDE µVision von Keil umgesetzt worden. Bei Abweichungen zu anderen 8051er-Controllern werden jeweils Anmerkungen oder Vergleiche durchgeführt, sodass die Programmierungen auch auf diesen 8051er-Architekturen lauffähig sind.

Als Bibliothek wird hier die Headerdatei „t89C51AC2“ des Vorgängers Atmel AT89C51AC2 verwendet. Bei der Nutzung eines anderen 8051er-Controllers muss diese Bibliothek gegen eine entsprechende ausgetauscht werden.

Falls eine andere IDE verwendet wird, so müssen die Deklarationen für die Bitverarbeitung dem jeweiligen Compiler angepasst werden (siehe Kapitel 6).

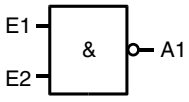
Zudem können die Programme mit der im Kapitel 5 beschriebenen Hardwarebeschaltung getestet werden.

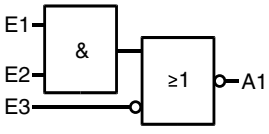
8.1 Verknüpfungssteuerungen mit Bitverarbeitung

Dieses Kapitel zeigt anhand von Beispielen, wie Programme mit Verknüpfungssteuerungen in die Programmiersprache C umgesetzt werden. Grundlagen zu den Verknüpfungssteuerungen wurden bereits im Kapitel 6 gezeigt. Zum Testen dient die oben beschriebene Entwicklungshardware.

Beispiele für die Programmierung von Verknüpfungssteuerungen

Verknüpfungssteuerung:



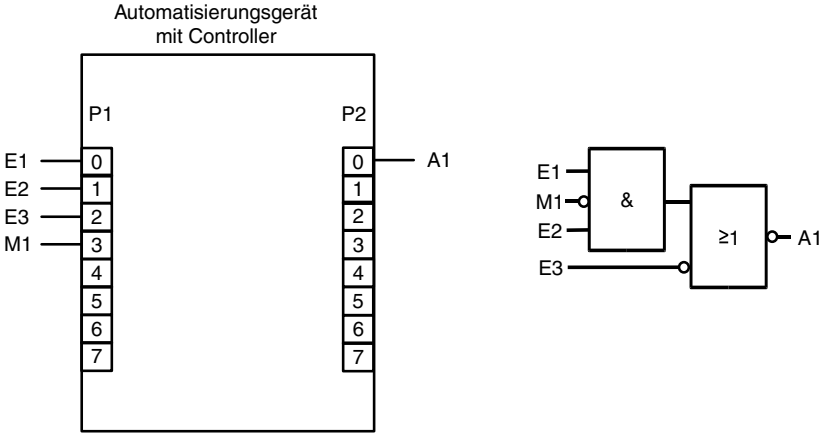


C-Programm:

```
A1=!(E1&&E2); // Und-Verknüpfung
              // mit logischem
              // Invertierer !

A1=!((E1&&E2)||!E3);
```

Beispiel 8.1

Schreiben Sie für folgende logische Verknüpfungssteuerung mit der Zuordnung der jeweiligen Ein- und Ausgangssignale ein vollständiges C-Programm.



Lösung:

Zu beachten ist, dass in der oben beschriebenen Hardware die Eingänge des Controllers invertiert angesteuert werden!

```
#include<t89c51ac2.h> // Einbinden der Controller-Bibliothek

// Den Portpins symbolische Namen zuordnen:

sbit E1=P1^0;        // E1 wird Port P1.0 zugeordnet
sbit E2=P1^1;        // E2 wird Port P1.1 zugeordnet
sbit E3=P1^2;        // E3 wird Port P1.2 zugeordnet
sbit M1=P1^3;        // M1 wird Port P1.3 zugeordnet
sbit A1=P2^0;        // A1 wird Port P2.0 zugeordnet

void main (void)     // Hauptprogramm
{
   while(1)          //Endlosschleife für zyklische Programmbearbeitung
   {
         A1=!((!E1&&!E2&&M1)||E3); // Logische Funktion mit
                                   // Invertierung der Eingänge
                                   // (siehe Hardwarebeschaltung)
   }
}
```

Übung 8.1

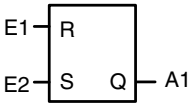
Erweitern Sie das obige Programm dahingehend, dass der Ausgang des UND-Gatters zusätzlich auf dem Port P2.7 ausgegeben werden soll.

Programmierung von Flipflops

Für die Programmierung von Flipflops werden, wie in der Automatisierungstechnik, verschiedene Merkerbits benötigt. Diese sind mit dem Datentyp bit am Anfang des Programms zu deklarieren.

Verknüpfungssteuerung:

Flipflop (setzdominant)

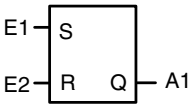


C-Programm:

```
if (E1==1)   // Wenn E1=1 ist,
{            // wird der Merker M0=0
     M0=0;   // gesetzt.
}
if (E2==1)   // Wenn E2=1 ist,
{            // wird der Merker M0=1
     M0=1;   // gesetzt.
}
A1=M0;       // Der Merker M0 wird
             // dem Ausgang
             // zugeordnet.
```

Verknüpfungssteuerung:

Flipflop (rücksetzdominant)

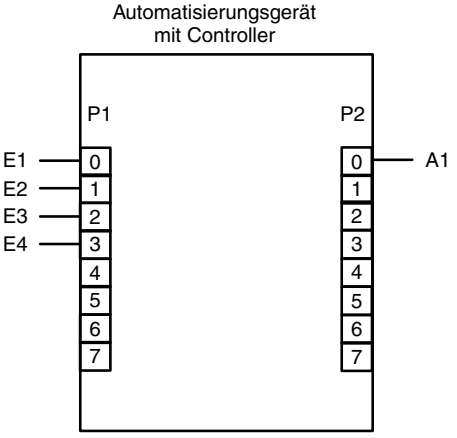


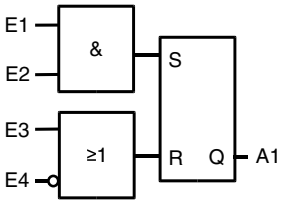
C-Programm:

```
if (E1==1)   // Wenn E1=1 ist,
{            // wird der Merker M0=1
     M0=1;   // gesetzt.
}
if (E2==1)   // Wenn E2=1 ist,
{            // wird der Merker M0=0
     M0=0;   // gesetzt.
}
A1=M0;       // Der Merker M0 wird
             // dem Ausgang
             // zugeordnet.
```

Beispiel 8.2

Programmieren Sie folgende Verknüpfungssteuerung.





Lösung:

```
#include<t89c51ac2.h>    // Einbinden der Controller-Bibliothek

// Den Symbolische Namen die Adressen der Portpins zuordnen:

sbit E1=P1^0;          // SFR-Adresse von Port P1.0
sbit E2=P1^1;          // SFR-Adresse von Port P1.1
sbit E3=P1^2;          // SFR-Adresse von Port P1.2
sbit E4=P1^3;          // SFR-Adresse von Port P1.3
sbit A1=P2^0;          // SFR-Adresse von Port P2.0

bit M0;                // Merkerbit M0 deklarieren

void main (void)       // Hauptprogramm
{
   while(1)            // Endlosschleife für die zyklische Programmbearbeitung
   {
        if ((!E1&&!E2)==1) // Abfrage Seteingang (Eingänge wegen der
                           // Hardwareverschaltung invertiert)
      {
         M0=0;
      }
      if ((!E3|E4)==1)    // Abfrage Reseteingang (Eingänge wegen der
                          // Hardwareverschaltung invertiert)
      {
         M0=1;
      }
      A1=M0;
   }
}
```

Übung 8.2

Ändern Sie das Programm so um, dass der Flipflop bei den gleichen Ein- und Ausgängen setzdominant arbeitet.

Signalabbilder erzeugen

Das Problem bei der zyklischen Programmbearbeitung von mehreren Verknüpfungssteuerungen besteht darin, dass sich während der Auswertung die Eingangssignale ändern können, und es so für einen Programmzyklus zu falschen Ausgangssignalen kommen kann. Ähnlich verhält es sich bei den Ausgangssignalen. Wenn mehrere Verknüfungssteuerungen hintereinander programmiert werden, so ist es möglich, dass zwei Ausgänge eingeschaltet werden, die eigentlich nicht zusammen eingeschaltet sein dürfen.

Aufgrund dieser Problematik soll zunächst vom Eingangsport als auch vom Ausgangsport ein Signalabbild erstellt werden. Am Anfang des Programms sind dann die Eingangssignale in das Signalabbild zu kopieren, um anschließend mit den kopierten Signalen die Verknüpfungssteuerungen auszuführen. Am Ende der Bitverarbeitung lassen sich die Ausgänge dann in den Ausgangsport kopieren.

Signalabbilder von Ports realisieren

Es soll beispielhaft ein Signalabbild von Port1 als Eingang erstellt werden.

Die Deklaration geschieht in C durch die Definition eines Bytes (Datentyp char). Mit dem Zusatz bdata (bei Keil µVision) kann das definierte Byte in den bitadressierbaren Bereich geschoben werden. Dabei ist beispielsweise das Bit 3 des definierten Bytes EB1 mit EB1^3 anzusteuern. Oft erscheint es sinnvoll, jedem einzelnen Bit mit sbit einen symbolischen Namen zuzuordnen.

```
// Signalabbild definieren:
char bdata EB1;
sbit Ein1 =EB1^0;
sbit Ein2 =EB1^1;
sbit Ein3 =EB1^2;
sbit Ein4 =EB1^3;
sbit Ein5 =EB1^5;
sbit Ein6 =EB1^6;
sbit Ein7 =EB1^7;
...
```

Im Hauptprogramm wird zyklisch am Anfang der Verknüpfungssteuerung dem Byte EB1 der Signalzustand von P1 übertragen.

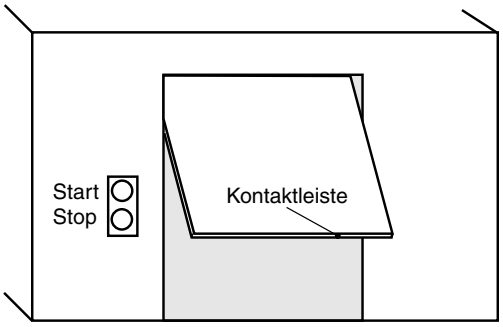
```
Hauptprogramm: EB1=P1;
```

Wenn, wie in unserem Fall, die Eingänge hardwarebedingt invertiert sind, so kann die Invertierung hier direkt vorgenommen werden. Die einzelne Invertierung im Programm entfällt.

```
Hauptprogramm: EB1=~P1; // EB1 wird der invertierte Signalzustand von P1
                        // übertragen
```

8.1.1 Steuerung eines Hallentores

Technologieschema



Funktion

Ein Hallentor soll sich von außen und von innen öffnen oder schließen lassen. Zum Einleiten der Bewegung ist an der Außenwand und in der Halle je ein Start-Taster angebracht. Wird

die Start-Taste bei geöffnetem Tor gedrückt, dann schließt das Tor. Eine Betätigung bei geschlossenem Tor soll ein Öffnen bewirken. Ob das Tor offen oder geschlossen ist, wird durch Grenztaster erfasst.

Das Tor wird durch einen Motor bewegt.

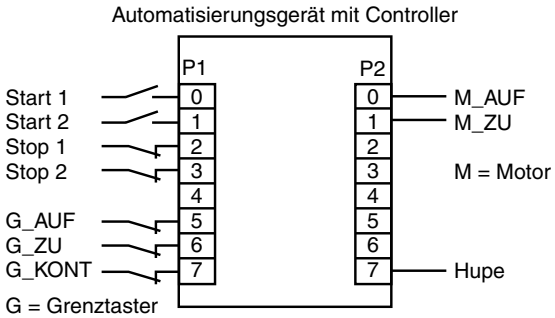
Die eingeleitete Bewegung lässt sich durch je einen Stop-Taster an der Außenwand und in der Halle anhalten. Bei erneutem Starten aus mittlerer Stellung muss das Tor immer öffnen.

Schließt das Tor, ertönt eine Hupe zur Warnung.

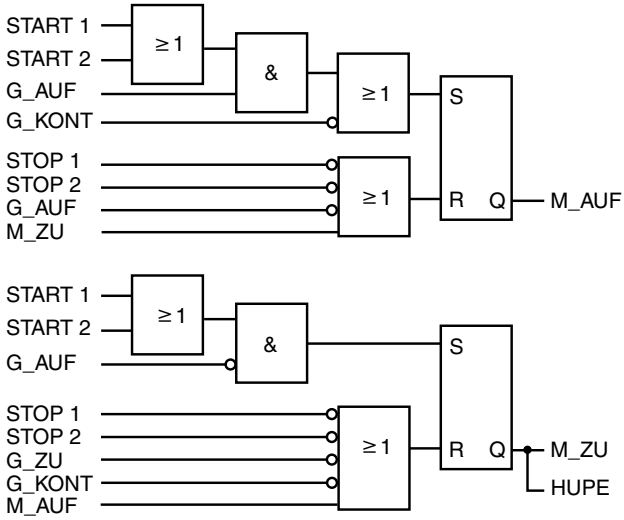
An der unteren Kante des Tores ist eine Kontaktleiste angebracht. Wird sie betätigt, muss das Tor sofort wieder öffnen.

Anschlussplan:

Die Eingangs- und Ausgangssignale lassen sich wie folgt den Ports des Controllers zuordnen:



Funktionsplan:



Lösung

Da hier mehrere Verknüfungssteuerungen verarbeitet werden, muss zunächst ein Signalabbild der Ein- und Ausgänge erzeugt werden. Erst anschließend kann die Bitverarbeitung durchgeführt werden. Am Ende des Programmzyklus wird dann das Ergebnis in den Port P2 übertragen. So kann es nicht vorkommen, dass das Tor gleichzeitig auf- und zu fährt, welches ohne Erzeugung der Signalabbilder passieren könnte.

C-Programm:

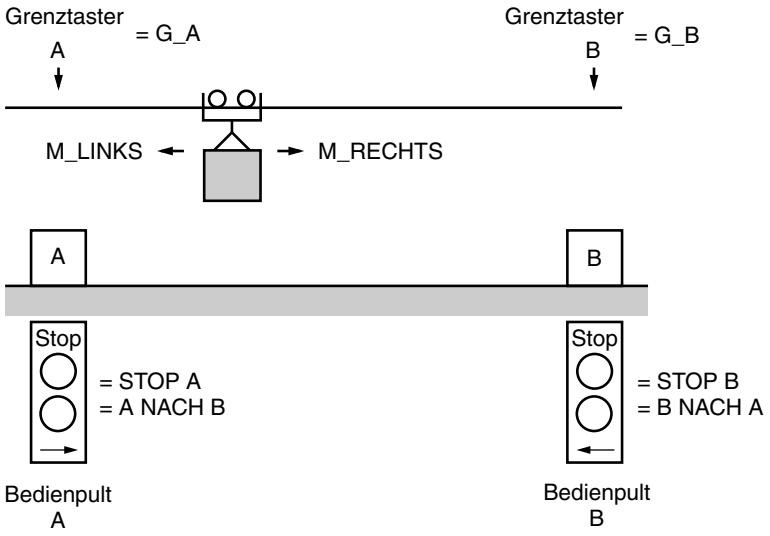
```
#include<t89c51ac2.h>        // Einbinden der Controller-Bibliothek

// Signalabbilder und symbolische Namen zuordnen:
// Eingaenge (Signalabbild P1):

char bdata EB1=0x00;         // Deklarieren des Signalabbildes von P1 und
                             // gleichzeitig alle Ports von EB1 Nullsetzen.
sbit Start1 =EB1^0;          // Abbild von Port P1.0
sbit Start2 =EB1^1;          // Abbild von Port P1.1
sbit Stop1  =EB1^2;          // Abbild von Port P1.2
sbit Stop2  =EB1^3;          // Abbild von Port P1.3
sbit G_AUF  =EB1^5;          // Abbild von Port P1.5
sbit G_ZU   =EB1^6;          // Abbild von Port P1.6
sbit G_KONT =EB1^7;          // Abbild von Port P1.7

// Ausgaenge (Signalabbild P2):

char bdata AB2=0x00;         // Deklarieren des Signalabbildes von P2 und
                             // gleichzeitig alle Ports von AB2 Null setzen.
sbit M_AUF =AB2^0;           // Abbild von Port P2.0
sbit M_ZU  =AB2^1;           // Abbild von Port P2.1
sbit Hupe  =AB2^7;           // Abbild von Port P2.7

// Merker definieren:
bit M0;                      // Merkerbit M0 deklarieren
bit M1;                      // Merkerbit M1 deklarieren

void main (void)             // Hauptprogramm
{
    P1=0xFF;                 // Vorbereitung, um über P1 Daten einzulesen

    while(1)                 // Endlosschleife für zyklische Programmbearbeitung
    {
        EB1=~P1;             // Eingangssignale lesen
                             // EB1 wird der invertierte Signalzustand von P1
                             // übertragen. (bei verwendeter Hardwarebeschaltung)

// Flipflop M_Auf programmieren:

        if ((((Start1||Start2)&&G_AUF)||!G_KONT)==1)     // Abfrage Seteingang
        {
            M0=1;
        }
```

```
        if ((!Stop1||!Stop2||!G_AUF||M_ZU)==1)          // Abfrage Reseteingang
        {
            M0=0;
        }
        M_AUF=M0;                                       // Zuweisung von M_AUF

// Flipflop M_ZU und Hupe programmieren:


        if (((Start1||Start2)&&!G_AUF)==1)              // Abfrage Seteingang
        {
            M1=1;
        }
        if ((!Stop1||!Stop2||!G_ZU||!G_KONT||M_Auf)==1) // Abfrage Reseteingang
        {
            M1=0;
        }

        M_ZU=M1;                                        // Zuweisung von M_ZU
        Hupe=M1;                                        // Zuweisung von Hupe

// Zuweisungen auf Port 2 übertragen:
        P2=AB2;

       }
}
```

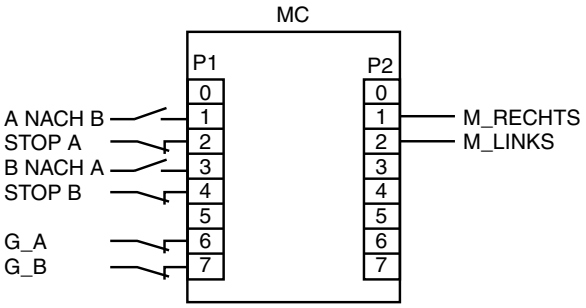
Übung 8.3

Schreiben Sie das Programm für folgende Transportsteuerung.

Technologieschema:



Anschlussplan:



Funktion:

Ein Laufkran in einer Fabrikhalle soll Teile zwischen Station A und Station B transportieren. Das Ziel darf nur von der Station vorgegeben werden, an der der Kran steht. Nach dem Drücken der Zieltaste fährt der Kran zum gegenüberliegenden Ziel.

Die Bewegung kann mit den Stopp-Tasten beider Stationen angehalten werden. Bei einem Stopp zwischen den Haltestellen, fährt er nach anschließendem Start mit einer der Zieltasten immer erst zum vorher gewählten Ziel, gleich welche der Zieltasten zum Starten gedrückt wird.

Aufgabe:

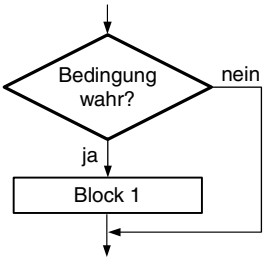
Entwerfen Sie die Verknüpfungssteuerung als Funktionsplan mit Logiksymbolen.

Schreiben Sie das C-Programm einschließlich Zuweisungen und Initialisierung.

8.2 Programmablaufpläne in C umsetzen

Zeitliche Verzögerungen und gleichzeitige Bitveränderungen an einem Port lassen sich mit Bitverknüpfungen nur schwer darstellen. Aufgrund dessen wird häufig bei komplexeren Abläufen auf die Programmablaufpläne zurückgegriffen. In diesem Kapitel soll gezeigt werden, wie Programmstrukturen von Programmablaufplänen in C umgesetzt werden können.

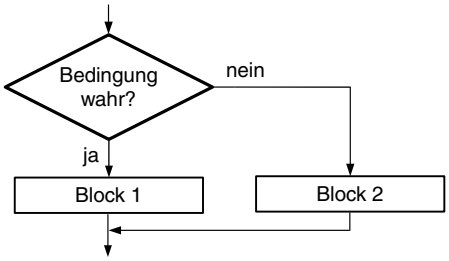
Programmablaufplan:



Zugehörige C-Syntax:

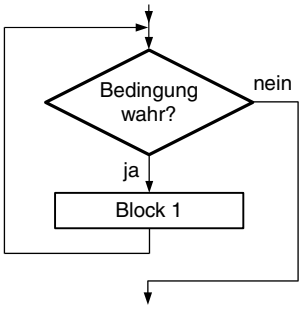
IF-Abfrage

```
if (Bedingung)
{
    // Block1
}
```



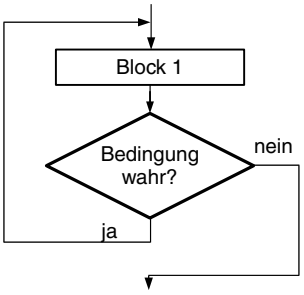
IF-ELSE-Abfrage

```
if (Bedingung)
{
    // Block1
}
else
{
    // Block2
}
```



Kopfgesteuerte WHILE-Schleife

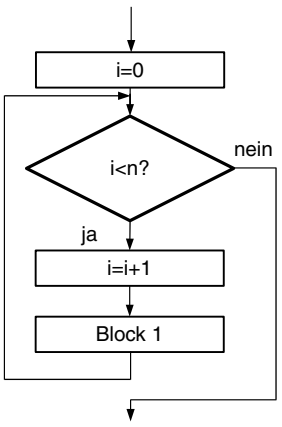
```
while(Bedingung)
{
    // Block1
}
```



Fußgesteuerte DO-WHILE-Schleife

```
do
{
    // Block1
}
while (Bedingung);
```

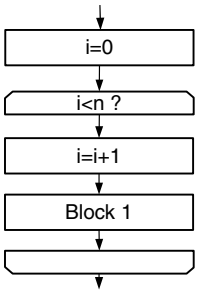
1. Möglichkeit:



FOR-Schleife

```
for (i=0;i<n;i++)
{
    // Block1
}
```

2. Möglichkeit:



Schleifen können auch mit dem Wiederholungssymbol im Programmablaufplan gekennzeichnet werden.

8.2.1 Lichteffekte mit Progammablaufplänen

Zum Einstieg wird anhand von unterschiedlichen Lichteffekten gezeigt, wie diese Programme mithilfe des Programmablaufplans dargestellt und umgesetzt werden können.

Beispiel 8.3

An Port 2 sollen alle angeschlossenen LEDs im gleichmäßigen sichtbaren Rhythmus blinken.

Vorüberlegungen:

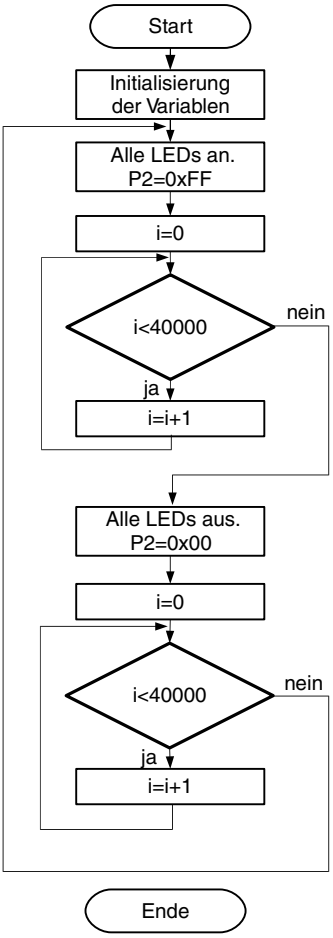
Um das Blinken der LEDs sichtbar zu machen, wird eine Zeitverzögerung benötigt. Die Zeitverzögerung kann mit einer einfachen Zählschleife realisiert werden. Dabei braucht der Controller eine gewisse Zeit, um eine Variable bis zum Beispiel 40 000 hochzuzählen.

Zudem muss überlegt werden, wie alle LEDs an Port 2 eingeschaltet und ausgeschaltet werden können. Am einfachsten geschieht dies, wenn alle Portausgänge auf einmal mit der Hexadezimal 0xFF (ein) und 0x00 (aus) aktiviert bzw. deaktiviert werden.

Für die Zeitverzögerung soll der Controller bis 40 000 hochzählen. Hierfür wird eine Variable vom Datentyp unsigned int benötigt, da int z. B. nur Ganzzahlen bis 32 767 beinhaltet.

Umsetzung:

PAP (Programmablaufplan):

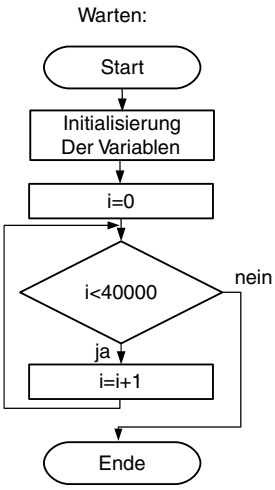


C-Programm:

```
#include<t89C51ac2.h>

void main() // Funktionskopf main
{
     unsigned int i;
     while (1)// Endlosschleife
{
          P2=0xFF;


          for (i=0;i<40000;i++)
          {
          }




P2=0x00;


          for (i=0;i<40000;i++)
          {
          }



     }    // Ende Endlosschleife
}         // Ende main
```

Optimierung

Das oben beschriebene Programm kann nun noch optimiert werden. Zum Beispiel wird zweimal dieselbe for-Schleife verwendet. Hierfür lässt sich eine Funktion realisieren.




```
void warten ()
{

     unsigned int i;

     for (i=0;i<40000;i++)
     {
     }
}
```

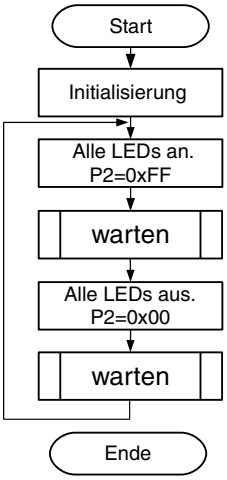
Anstelle der leeren geschweiften Klammern kann auch einfach ein Semikolon gesetzt werden.

Anstatt:

```
for (i=0;i<40000;i++)
{
    // leerer Block
}
```

```
for (i=0;i<40000;i++);
```

Optimiertes Programm:




```
#include <t89C51ac2.h>

void warten();  // Funktionsprototyp

void main()     // Funktionskopf main
{
     while (1) // Endlosschleife
     {
          P2=0xFF;

          warten();

          P2=0x00;

          warten();

     }    // Ende Endlosschleife
}         // Ende main
```

Unterprogramm warten:
PAP siehe oben.

```
void warten ()
{
     unsigned int i;
     for (i=0;i<40000;i++);
}
```

Bei der Verwendung von Funktionen müssen diese vor der Hauptfunktion als Funktionsprototyp stehen. Dann können diese nach der Hauptfunktion main geschrieben werden.

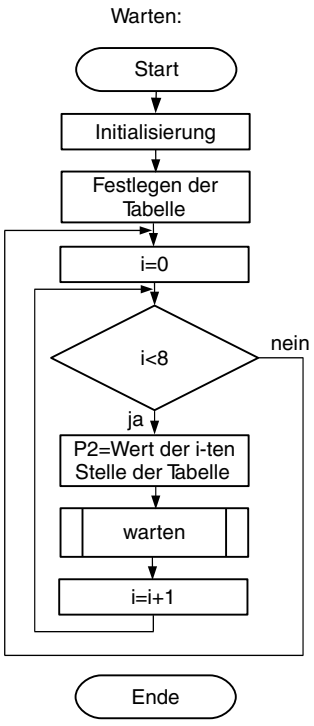
Beispiel 8.4

Es soll mithilfe eines Arrays ein Lauflicht an Port 2 erzeugt werden.

Zunächst sind die notwendigen HEX-Zahlen zur Ausgabe von P2 zu ermitteln.

P2.7	P2.6	P2.5	P2.4	P2.3	P2.2	P2.1	P2.0	HEX
1	0	0	0	0	0	0	0	0x80
0	1	0	0	0	0	0	0	0x40
0	0	1	0	0	0	0	0	0x20
0	0	0	1	0	0	0	0	0x10
0	0	0	0	1	0	0	0	0x08
0	0	0	0	0	1	0	0	0x04
0	0	0	0	0	0	1	0	0x02
0	0	0	0	0	0	0	1	0x01

PAP:



C-Programm:

```
#include<t89C51ac2.h>

void warten();    // Funktionsprototyp

void main()       // Funktionskopf main
{
     int i;

     char Tabelle[8] = {0x80, 0x40,
     0x20, 0x10, 0x08, 0x04, 0x02, 0x01};

     while (1)    // Endlosschleife
     {
          for (i=0;i<8;i++)
          {
             P2=Tabelle[i];
             warten();
          }

     }    // Ende Endlosschleife
}         // Ende main
```

Tabelle: 0x80;0x40;0x20;0x10;0x08;0x04;0x02;0x01

Die Wartefunktion wird von der obigen Aufgabe übernommen. Es wird nur die Wartezeit verändert auf 10 000, damit ein gleichmäßiger Durchlauf zu sehen ist.

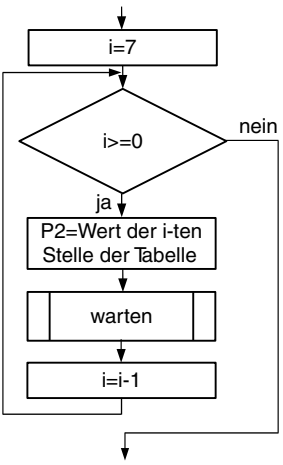
```
void warten()
{
     unsigned int i;
     for (i=0;i<10000;i++);
}
```

Beispiel 8.5

Es soll ein Lauflicht realisiert werden, welches erst von rechts nach links und dann von links nach rechts usw. arbeitet.

Dies ist nun einfach zu realisieren. Es muss, nachdem die obige Tabelle durchlaufen wurde, diese wieder rückwärts durchlaufen werden.

Ergänzung:



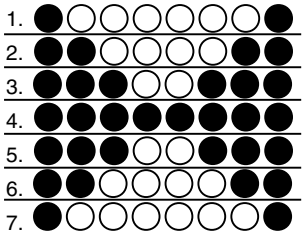

```
for (i=7;i>=0;i--)
{
     P2=Tabelle[i];
     warten();
}
```

Übung 8.4

Realisieren Sie die Ergänzung des Programms! Das Lauflicht scheint außen immer stehen zu bleiben. Ändern Sie dies!

Übung 8.5

Es soll ein Programm „Wischer“ erzeugt werden. Dabei sollen die LEDs zyklisch nacheinander wie folgt leuchten:



schwarz: LED an

Erstellen Sie ein C-Programm!

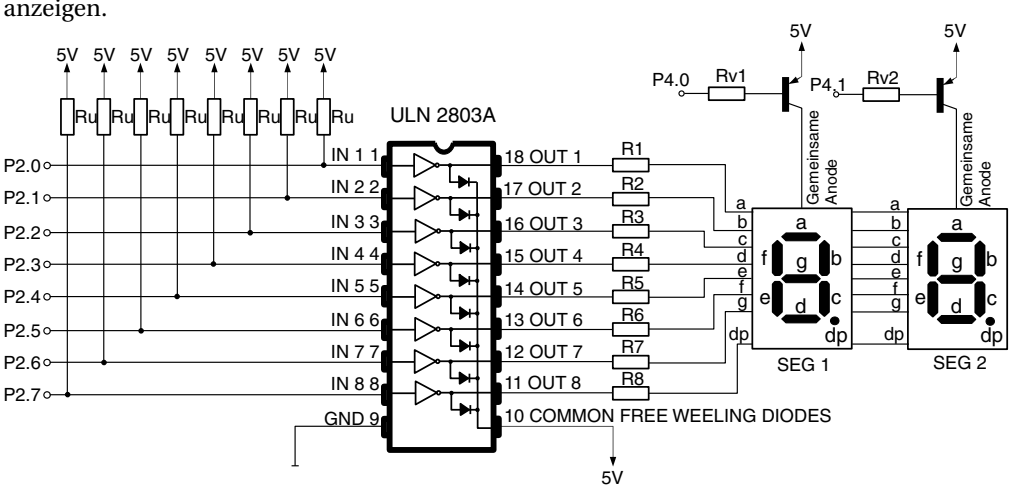
Übung 8.6

Schreiben Sie jeweils eine Funktion für das Programm Lauflicht und Wischer! In dem Hauptprogramm sollen dann nacheinander die Funktionen aufgerufen werden.

8.2.2 Ansteuern von zwei Siebensegmentanzeigen

Häufig werden in Anwendungen 7-Segmenanzeigen verwendet, da diese preislich sehr günstig sind, und für die meisten Anzeigen ausreichen. Damit nicht zu viele Ports am Controller verbraucht werden, ist es möglich, die Segmente über Transistoren schnell hin- und herzuschalten. Das menschliche Auge kann diesen Schaltvorgang nicht wahrnehmen, sodass es aussieht, als wären beide Segmente zur selben Zeit eingeschaltet.

Der folgende Schaltplan zeigt eine mögliche Schaltung zur Ansteuerung von zwei 7-Segmentanzeigen.



Mithilfe einer Tabelle können die auszugebenden Hex-Zahlen an Port 2 für die entsprechenden Ziffern bestimmt werden.

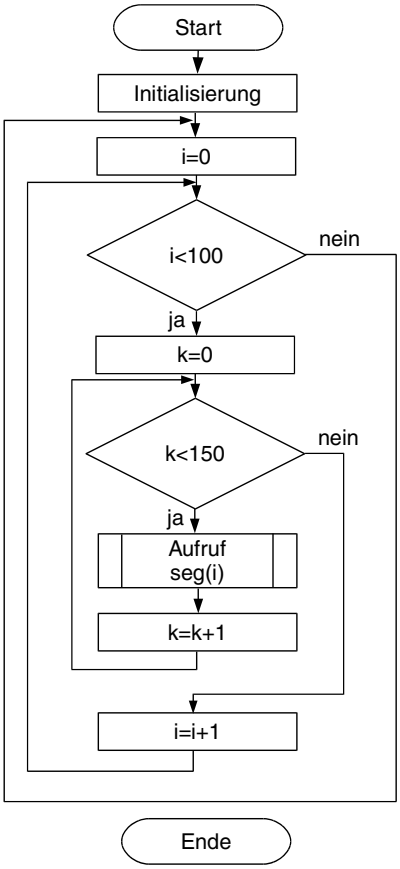
Ziffer	P2.7-dp	P2.6-g	P2.5-f	P2.4-e	P2.3-d	P2.2-c	P2.1-b	P2.0-a	HEX
0	0	0	1	1	1	1	1	1	0x3F
1	0	0	0	0	0	1	1	0	0x06
2	0	1	0	1	1	0	1	1	0x5B
3	0	1	0	0	1	1	1	1	0x4F
4	0	1	1	0	0	1	1	0	0x66
5	0	1	1	0	1	1	0	1	0x6D
6	0	1	1	1	1	1	0	1	0x7D
7	0	0	0	0	0	1	1	1	0x07
8	0	1	1	1	1	1	1	1	0x7F
9	0	1	1	0	1	1	1	1	0x6F

Es soll hier nun beispielhaft ein Zähler erzeugt werden, der Dezimal von 0 bis 99 zählt. Die Zahlen sollen auf den zwei 7-Segmentanzeigen dargestellt werden. Anschließend beginnt der Zähler wieder mit 0.

Das Hauptprogramm sieht wie folgt aus:

PAP:

Zähler mit 7-Segmentanzeige:



Erläuterungen:

Initialisierung:

Zur Initialisierung gehören notwendige Zählvariablen und das Einbinden der Funktionsprototypen.

Zyklischer Programmablauf

FOR Schleife:

In der Schleife wird die Variable i von 0 bis 99 gezählt. Die entsprechenden Zahlen sollen auf den 7-Sementanzeigen dargestellt werden.

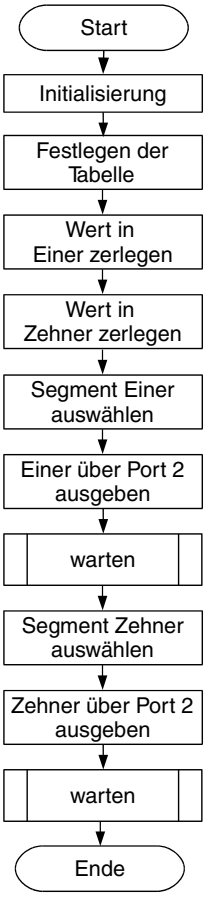
Aufruf der Funktion „Anzeige“:

Die Funktion „Anzeige“ wird mithilfe einer weiteren for-Schleife 150 mal aufgerufen. Dabei wird jedes Mal der anzuzeigende Wert (hier i) an die Funktion übergeben. Durch den mehrmaligen Aufruf ergibt sich eine Verzögerungszeit. Würde die Verzögerung in einer einzelnen Funktion aufgerufen, so wäre in der Zeit die 7-Segmentanzeige deaktiviert. Die Anzeige würde flackern.

Das Unterprogramm „Seg“ sieht wie folgt aus:

PAP:

Unterprogramm Seg:



Erläuterungen:

Initialisierung:

Zur Initialisierung gehören die notwendigen Variablen für den Zehner und Einer.

Festlegen der Tabelle:

Hier werden die oben hergeleiteten HEX-Werte der 7-Seg.-Anzeige in einer Tabelle bzw. in einem Array-Feld gespeichert.

Wert in Einer/Zehner zerlegen:

Da der darzustellende Wert eine zweistellige Zahl ist, muss diese in den Einer und Zehner zerlegt werden. So kann jede Ziffer einzeln auf einer 7-Seg.-Anzeige dargestellt werden.

Segement (Einer) auswählen:

Hier wird über Port P4.0 das zweite Segment aktiviert.Port P4.1 wird deaktiviert.

Einer ausgeben:

Über Port 2 wird die anzuzeigende Ziffer ausgegeben. Dazu wird der jeweilige Tabelleneintrag benötigt.

Warten:

Kurze Wartezeit, damit die LEDs voll durchschalten.

Segment1 (Zehner) auswählen:

Hier wird über Port P4.1 das erste Segment aktiviert. P4.2 wird deaktiviert.

Zehner ausgeben:

Über Port 2 wird die anzuzeigende Ziffer ausgegeben. Dazu wird der jeweilige Tabelleneintrag benötigt.

Warten:

Kurze Wartezeit, damit die LEDs voll durchschalten.

C-Programm:

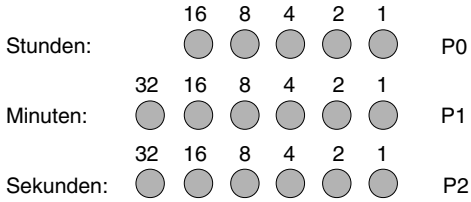
```
#include <t89c51ac2.h>                  // Einbinden der Mikrocontroller-Bibliothek

sbit P40=P4^0;                          // Port Zehner (low active)
sbit P41=P4^1;                          // Port Einer (low active)

void seg(unsigned char zahl);           // Funktionprototyp 7-Segmentanzeige
void warten(void);                      // Funktionsprototyp warten

void main(void)                         // Hauptprogramm
{
      int i,j;                          // Variablendefinition für Zählvariablen
      while (1)                         // Endlosschleife
      {
            for (i=0;i<100;i++)         // Von 0 bis 100 zählen: Die Zählwerte
                                        // sollen auf den 7-Seg.-Anzeigen
                                        // dargestellt werden.
            {
                  for (k=0;k<150;k++)   // Wartezeit mit Aufruf der Funktion seg,
                  {                     // um eine flackerfreie Anzeige zu erhalten.
                        seg(i);
                  }
            }
      }                                 // Ende Endlosschleife
}                                       // Ende main

void seg(unsigned char zahl)            // 7-Segment-Ansteuerung
{
      unsigned char z=0;                // Variable z für Zehner
      unsigned char e=0;                // Variable e für Einer
                                        // Tabelle / Array-Feld der Anzeigewerte für
                                        // die 7 Segmente:
      unsigned char ziffer[10]={0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F};

      e=zahl%10;                        // Einer ermitteln mittels Modulo-Operator
      z=(zahl-e)/10;                    // Zehner ermitteln
      P40=1;                            // Display Einer wählen
      P41=0;
      P2=ziffer[e];                     // Einer ausgeben
      warten();                         // Funktion warte zur Stabilisierung der Anzeige

      P40=0;                            // Display Zehner wählen
      P41=1;
      P2=ziffer[z];                     // Zehner ausgeben
      warten();                         // Funktion warte zur Stabilisierung der Anzeige
}

void warten(void)                       // Funktion warten
{
      unsigned int j;               // Eine Zählvariable wird von 0 bis 300 hochgezählt.
      for(j=0;j<300;j++);
}
```

8.2.3 Programmieren einer Binäruhr mit einem externen Taktgenerator

In diesem Beispiel soll über einen externen genauen Taktgeber, welcher jede Sekunde eine steigende Flanke hat, eine Binäruhr programmiert werden.

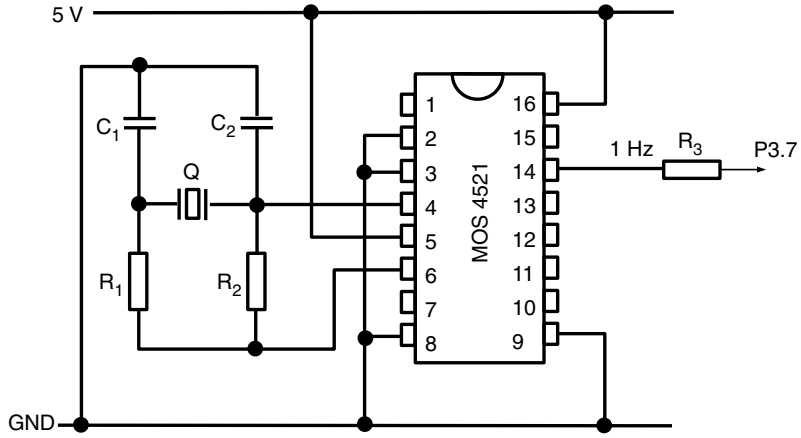
Die Binäruhr zeigt die Stunden, Minuten und Sekunden nicht wie herkömmlich als Zahl an, sondern mit einzelnen LEDs in der Binärcodierung.

Die Anzeige der zu programmierenden Binäruhr soll wie folgt aussehen:



Zusätzliche Hardware

Als Taktgenerator dient ein genauer Quarzoszillator in Kombination mit dem Frequenzteiler MOS 4521. Dieser schaltet den Ausgang zwischen 0 V und 5 V. Damit der Controller bei einer fehlerhaften Programmierung nicht zerstört wird, wird der Strom durch den Widerstand R_3 auf maximal 1,1 mA begrenzt (wenn der Eingang des Controllers versehentlich auf Ground programmiert ist).



An Port 0 und Port 1 wird wie bei Port 2 der Treiberbaustein ULN2803A mit entsprechenden LEDs zur Anzeige beschaltet.

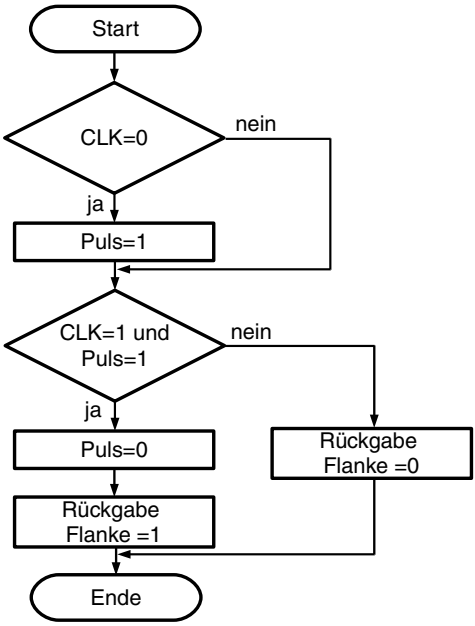
Flankenerkennung

Das genaue 1-Hz-Signal wird an Port 3.7 eingespeist. Bei jeder positiven Flanke soll die Variable für die Sekunden um eins erhöht werden.

Um Flanken zu erkennen, muss ein momentaner Zustand mit einem vorherigen Zustand verglichen werden. In dem rechts dargestellten Programmablaufplan wird das Bit Puls gleich 1 gesetzt, wenn das Eingangssignal Low-Zustand hat.

Anschließend erfolgt eine Abfrage, ob das Eingangssignal High und der vorherige Zustand Puls = 1 ist. Wenn nicht, trat keine Flanke auf. Wechselt jetzt das Eingangssignal von Low auf High, so ist das Bit Puls noch 1 und das Eingangssignal auch 1. Es wird also erkannt, dass ein Flankenwechsel von Low auf High auftrat. Das Bit Puls wird dann ebenfalls wieder zu 0 gesetzt. Wenn beim nächsten Aufruf immer noch das Eingangssignal 1 ist, ist aber das Bit Puls wieder 0, sodass erkannt wird, dass keine Flanke auftrat.

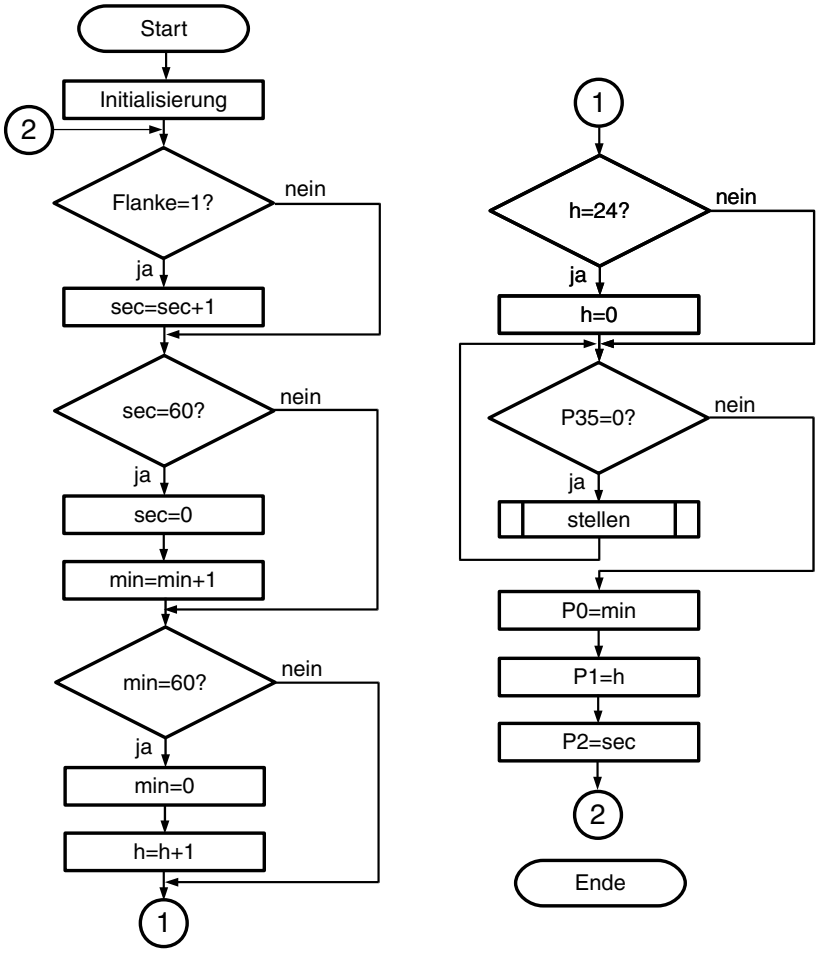
Funktion Flankenerkennung:



Realisierung der Funktion in C:

```
bit flanke ()
{
      if(!CLK)                  // Abfragen, ob Eingangssignal=0
      {
            Puls=1;             // Puls =1 setzen
      }
      if ((CLK==1)&Puls)        // Positive Flanke erkennen
      {
            Puls=0;             // Puls=0 setzen
            return 1;           // Die Funktion meldet, dass eine Flanke auftrat
      }
      else
      {
            return 0;           // Es ist keine Flanke aufgetreten.
      }
}
```

Programmablaufplan für die Uhr

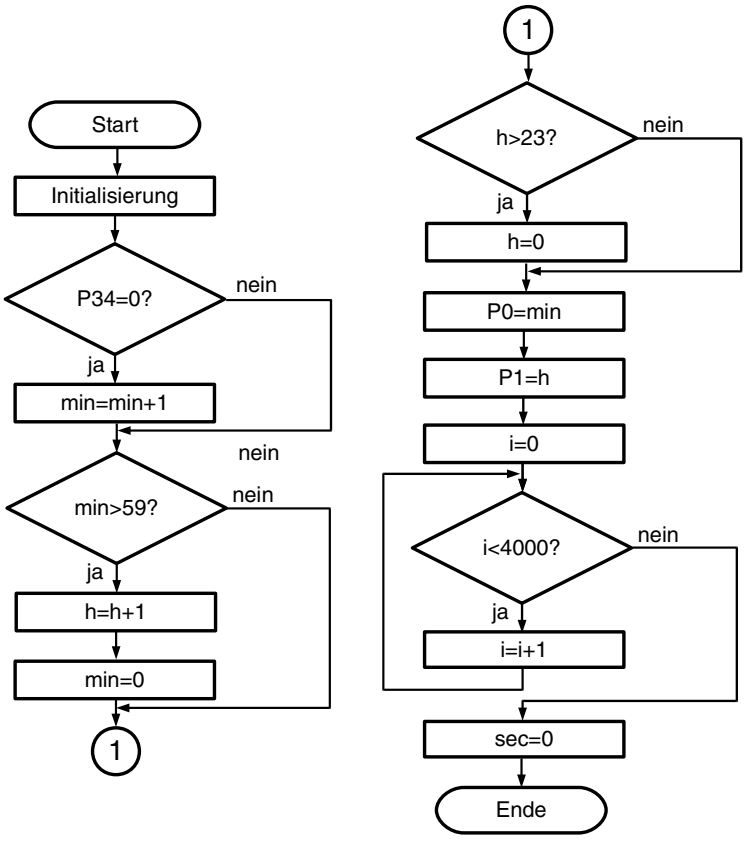


Verwendete Symbole:
h: Stunden
sec: Sekunden
min: Minuten

Einstellen der Uhrzeit

Mithilfe von zwei Tastern an P3.5 und P3.4 soll die Uhrzeit eingestellt werden. Sobald P3.5 betätigt wird, hält die Uhrzeit an. Dann kann mit P3.4 die Uhrzeit eingestellt werden. Es muss die gesamte Zeit P3.5 betätigt sein. Die Uhrzeit läuft dann mit P3.4 sehr schnell auf die einzustellende Zeit. Zum Schluss werden die Sekunden auf null gesetzt.

Programmablaufplan zum Uhrstellen



Übung 8.7

Realisieren Sie das Programm mithilfe der Programmablaufpläne in C!

8.2.4 Ansteuern eines LC-Displays

Ein Gerät, welches über einen Mikrocontroller gesteuert wird, benötigt eine Vielzahl von Eingangssignalen zu seiner Bedienung. Außerdem gibt es eine Reihe von Rückmeldungen aus dem Gerät oder Prozess sowie Anzeigen des Betriebszustandes.

Um diese Ein- und Ausgaben des Controllers für den Bediener übersichtlich darzustellen, wird eine der Funktion des Gerätes entsprechende Bedienoberfläche benötigt. Sie muss Schalter, Taster und Anzeigelampen enthalten und mit erklärenden Texten unterlegt sein. Solch eine Bedienoberfläche ist im Verhältnis zur Controller-Steuerung sehr kostspielig, wenig flexibel und benötigt viel Platz. Außerdem wird dafür eine große Anzahl von Controller-Ports benötigt. Dieser Aufwand ist insbesondere für kleine und preiswerte Geräte viel zu hoch.

Es wird nach einer standardisierten Lösung gefragt, die variabel der Funktion angepasst werden kann. Sie soll wenig Platz benötigen und mit einer geringen Anzahl Ports des Controllers auskommen.

Solch eine Möglichkeit bietet ein LC-Display mit wenigen angepassten Bedientasten. Das LC-Display ermöglicht erklärende Texte und falls notwendig eine Bedienerführung in Form eines Menüs. Es lassen sich sowohl Signaleingaben als auch Zahlen- und Texteingaben vornehmen. Diese können kontrolliert und anschließend quittiert werden. Es wird nur eine geringe Anzahl von Portanschlüssen am Controller benötigt, da die wenigen Bedienelemente, je nach Menüpunkt, immer neue Bedeutungen haben und andere Funktionen auslösen.

Das LC-Display kommuniziert mit dem Controller über eine geringe Anzahl paralleler Portanschlüsse. Über diese Anschlüsse kann der Controller Steuersignale austauschen und ASCII-Zeichen für Buchstaben oder Ziffern ausgeben, die auf dem Display erscheinen sollen.

Das zeichenbasierte LC-Display mit dem Standardchipsatz

Hier soll die Ansteuerung eines LC-Zeichendisplays mit dem Standardchipsatz HD44780 von Hitachi oder einem kompatiblen Chipsatz beschrieben werden.

Die komplett zu beziehende Einheit „LC-Display“ besteht aus einer gerahmten Flüssig-Kristall-Anzeige mit Hintergrundbeleuchtung und einer Platine mit dem eigenem Standard-Controller HD44780. Der interne RAM-Speicher für die Datenanzeige des Controllers besteht aus 80 Speicherplätzen mit je 8-Bit. Jede Speicherstelle entspricht einem möglichen darzustellenden Zeichen.

Es gibt Flüssig-Kristall-Anzeigen mit mehreren Zeilen, z. B. maximal 4 Zeilen mit jeweils 20 Zeichen. Sind weniger Zeilen vorhanden, können die Zeilen länger sein.

Die Adressen der Speicherzellen werden meist wie in der nebenstehenden Tabelle aufgeteilt. Bei unterschiedlichen Größen sind die Adressen ähnlich angeordnet. Wird z. B. ein 2x16 Display eingesetzt, so sind die Adressen wie beim 2×40-Display festgelegt, es werden allerdings nur die ersten 16 Zeichen der jeweiligen Zeile angezeigt. Der übrige Speicher kann trotzdem beschrieben werden und z. B. über eine Schiebefunktion in den darstellbaren Bereich verschoben werden.

Aufteilung des Datenspeichers am Beispiel eines zwei und vierstelligen Displays:

Display-größe	1. Zeile (HEX)	2. Zeile (HEX)	3. Zeile (HEX)	4. Zeile (HEX)
2×40/ 2×16	00-27	40-67		
4×20	00-13	40-53	14-27	54-67

Jedes Zeichen besteht aus einer Punktmatrix von 5×7 Punkten. Die Ansteuerung der gebräuchlichen Zeichen erfolgt im ASCII-Code.

Bedeutung und Funktion der Pins

In der Regel haben die Zeichendisplays 14 bzw. 16 Anschlüsse. Bei 16 Anschlüssen werden zwei Pins für die Hintergrundbeleuchtung verwendet. Werden diese nicht angeschlossen, funktioniert das Display auch ohne Hintergrundbeleuchtung.

Spannungsversorgung

An den beiden Pins 1 und 2 wird die Spannungsversorgung des Displays angeschlossen. Hier 5V.

Kontrastspannung

Mithilfe dieser Spannung lässt sich der Kontrast anpassen. Diese Spannung kann zum Beispiel mit dem Mittelabgriff eines Potenziometers ($> 5\,\text{k}\Omega$), welches an 0 V und 5 V angeschlossen ist, eingestellt werden.

Registerauswahl

Mit der Registerauswahl wird dem Display mitgeteilt, ob ein Befehl oder darzustellende Daten an das Display geschickt bzw. vom Display gelesen werden sollen.

Schreiben/Lesen

Beim „Low"-Signal werden Daten bzw. Befehle an das Display geschickt. Beim „High-Signal" lassen sich Daten oder Befehle vom Display auslesen. Sinnvolle zu empfangende Befehle sind zum Beispiel, ob das Display den vorherigen Befehl schon abgearbeitet hat, oder es damit noch beschäftigt ist.

Enable

Mit einer fallenden Flanke wird die entsprechende Aktion ausgelöst, z. B. jetzt die Daten in das Datenregister schreiben, oder jetzt die Daten ins Befehlsregister schreiben. Beim Lesen wird der Wert auf „High" gesetzt.

DB0-DB7

Über diese Pins werden die 8-Bit langen Daten bzw. Befehle übertragen. Es ist auch möglich das Display im sogenannten 4-Bit Betrieb zu nutzen. Dann werden zuerst die höheren vier Bit und anschließend die niederwertigen vier Bits des zu sendenden Bytes mit DB4 bis DB7 übertragen. In diesem Modus werden die Pins DB0 bis DB3 nicht benötigt und somit notwendige Pins am Controller eingespart.

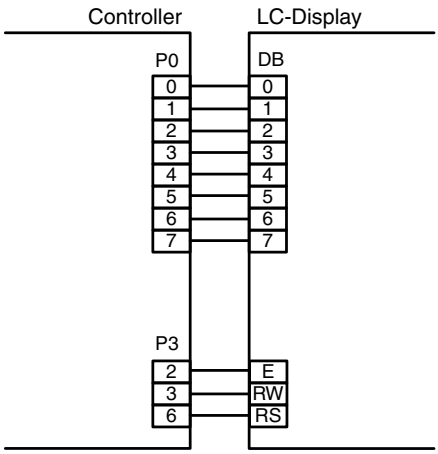
PIN	Bezeichnung	Funktion
	A_{LED}	Anode der Hintergrundbeleuchtung
	K_{LED}	Katode der Hintergrundbeleuchtung
1	GND	Versorgungsspannung Masse, 0 V
2	VDD	Versorgungsspannung $+5$ V
3	V_C	Kontrastspannung 0 V $\rightarrow$ maximaler Kontrast, $+5$ V $\rightarrow$ minimaler Kontrast
4	RS	Registerauswahl: Befehl: RS=0 Daten: RS=1
5	R/W	Schreiben/Lesen Schreiben: R/W=0 Lesen: R/W=1
6	E	Enable Fallende Flanke: Aktion wird ausgelöst
7	DB0	(Bit 0 eines Befehls oder Bit 0 der Daten)
8	DB1	(Bit 1 eines Befehls oder Bit 1 der Daten)
9	DB2	(Bit 2 eines Befehls oder Bit 2 der Daten)
10	DB3	(Bit 3 eines Befehls oder Bit 3 der Daten)
11	DB4	Bit 4 eines Befehls oder Bit 4 der Daten
12	DB5	Bit 5 eines Befehls oder Bit 5 der Daten
13	DB6	Bit 6 eines Befehls oder Bit 6 der Daten
14	DB7	Bit 7 eines Befehls oder Bit 7 der Daten

Anschluss des Displays am Mikrocontroller

In einem ersten Schritt soll das Display parallel über die 8-Bit Datenleitung angeschlossen werden.

Hierzu sind beispielhaft die Datenleitungen des Displays mit Port 0 verbunden und die Steuerleitungen mit Port 3, wie nebenstehend zu sehen. Die benötigten Spannungspegel sind TTL und Controller-Port kompatibel.

Das Display ist zusätzlich mit der Spannungsversorgung und der Kontrastspannung zu versorgen.



Die Befehlsliste des Displays

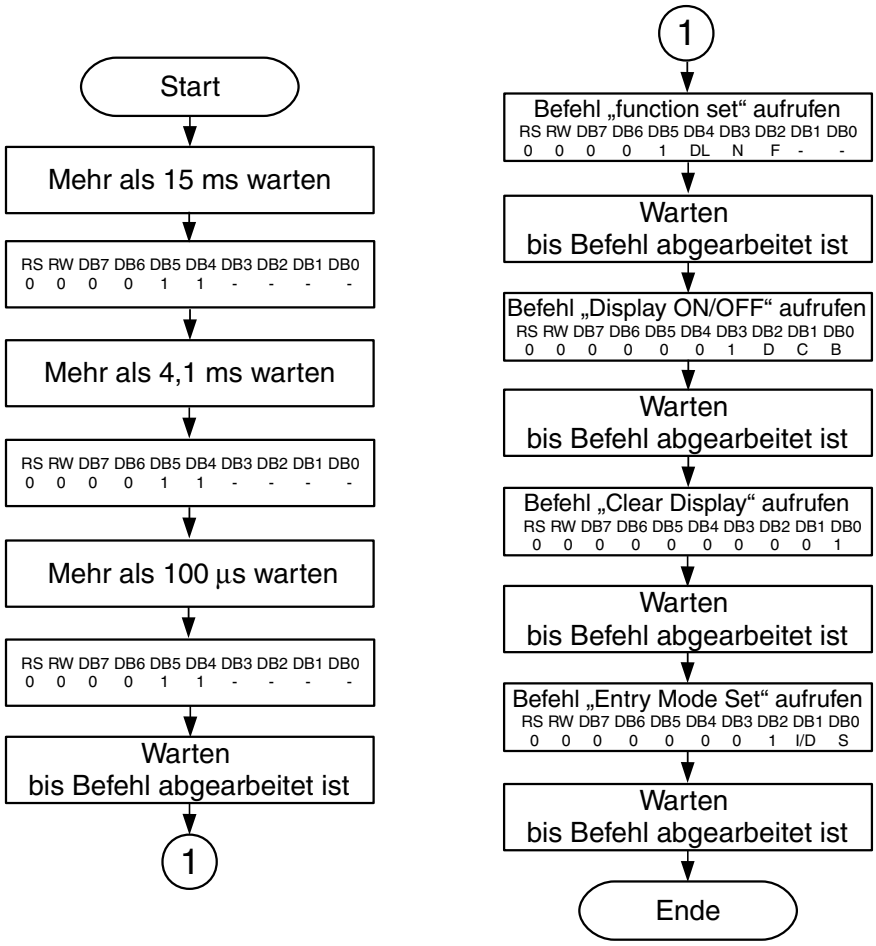
Befehl	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Funktion
Clear display	0	0	0	0	0	0	0	0	0	**1**	Anzeige löschen
Return Home	0	0	0	0	0	0	0	0	**1**		Platziert den Cursor an Adresse 0 des Datenspeichers
Entry Mode set	0	0	0	0	0	0	0	**1**	I/D	S	S=0: Der Cursor wird nach dem Schreiben/Lesen eines Zeichens jeweils um eine Stelle entsprechend von I/D verschoben. I/D=1: Cursor nach rechts verschieben (inkrementieren) I/D=0: Cursor nach links verschieben (dekrementieren)
Display ON/OFF control	0	0	0	0	0	0	**1**	D	C	B	B=0/1: Blinkender Cursor aus/ein C=0/1: Cursor-Unterstrich aus/ein D=0/1: Display aus/ein
Cursor/ Display Shift	0	0	0	0	0	**1**	S/C	R/L	–	–	Verschiebt die Anzeige (S/C=1) oder den Cursor (S/C=0) um eine Stelle nach links (R/L=0) oder rechts (R/L=1), ohne die Inhalte des Datenspeichers zu ändern.
Funktion Set	0	0	0	0	**1**	DL	N	F	–	–	F=0: 5×7 Punktmatrix F=1: 5×10 Punktmatrix N=0: eine Displayzeile N=1: 2 oder 4 Displayzeilen DL=0/1: 4-Bit / 8-Bit-Ansteuerung

Befehl	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Funktion
Set CG RAM Adress	0	0	0	**1**	A5	A4	A3	A2	A1	A0	Setzt die Adresse des Speichers für die frei definierbaren Zeichen.
Set DDRAM Adress	0	0	**1**	A6	A5	A4	A3	A2	A1	A0	Setzt die Schreibadresse des Datenspeichers fest.
Read Busy-Flag Adresse	0	1	BF	A6	A5	A4	A3	A2	A1	A0	BF=1: Display beschäftigt BF=0: Display bereit A0..A6: aktuelle Adresse
Write Data	1	0	D7	D6	D5	D4	D3	D2	D1	D0	Schreibt Daten in den Datenspeicher (DDRAM) oder in den Speicher für frei programmierbare Zeichen (CGRAM).
Read Data	1	1	D7	D6	D5	D4	D3	D2	D1	D0	Liest Daten aus dem Datenspeicher (DDRAM) oder aus dem Speicher für frei programmierbare Zeichen (CGRAM).

Initialisierung des Displays

Normalerweise initialisiert sich das Display nach dem Einschalten der Spannungsversorgung automatisch. Es kann allerdings vorkommen, dass die Initialisierung fehlschlägt. Zudem muss das Display auch noch konfiguriert werden. Aufgrund dessen empfiehlt es sich, bevor das Display benutzt wird, eine Neuinitialisierung durchzuführen. Dazu gibt der Hersteller eine bestimmte Routine vor. Es soll dreimal hintereinander die Kombination 0x30 übertragen werden. Anschließend muss mit den oben dargestellten Befehlen das richtige Display konfiguriert werden. Zwischen den Befehlen ist eine gewisse Wartezeit einzuhalten.

Programmablaufplan für die Initialisierungsphase nach Herstellerangaben



Die Initialisierungsphase kann in einer Funktion beschrieben werden. Es müssen die sechs Befehle an das LC-Display übertragen werden.

Hier soll ein zweizeiliges Display mit jeweils 16 Zeichen und einer 5x7 Punktmatrix initialisiert werden. Das Display soll eingeschaltet und der Cursor aktiviert sein.

Befehle:
1. Das dreimal zu übertragende Muster entspricht dem HEX-Wert 0x30 an P0.
2. Der Befehl „Funktion SET“ entspricht hier (F=0; N=1; DL=1) dem HEX-Wert 0x38 an P0.
3. Der Befehl „Display ON/OFF“ entspricht hier (Display an: D=1; Cursor an C=1; blinken an B=1) dem HEX-Wert 0x0F an P0.
4. Der Befehl „Clear Display“ entspricht dem HEX-Wert 0x01 an P0.
5. Der Befehl „Entry Mode Set“ entspricht hier (Cursor nach rechts verschieben I/D=1; S=0) dem HEX-Wert 0x06.

Der Aufruf der Befehle lässt sich wie beim Lauflicht mit einem Array-Feld realisieren. Nach jedem Befehl muss eine Wartezeit eingehalten werden, die hier mit der Funktion warten realisiert wird. Je nach Taktfrequenz des Controllers variiert die Zählvariable für die leere Zählschleife der Verzögerung. Da Befehle gesendet werden, müssen RS und RW „Low“ sein. Zur Übertragung muss an EN eine negative Flanke auftreten, die durch die Umschaltung von EN=1 auf EN=0 realisiert wird.

Funktion der Initialisierung in C:

```
void init2x16()
{
      char k;                  // interne Zählvariable
                               // Befehlsliste:

      unsigned char befehl[7]={0x30,0x30,0x30,0x38,0x0F,0x01,0x06};

      warten();                // mehr als 15 ms warten

      for (k=0;k<7;k++)        // alle Befehle auf P0 übertragen
      {
            RS=0; RW=0; EN=1;
            warten();
            P0=befehl[k];
            EN=0;
      }
      warten();
}

void warten()                  // Funktion für die Verzögerungszeit
{
      int i;
      for(i=0;i<100;i++);
}
```

Nach einer erfolgreichen Initialisierungsphase blinkt der Cursor an Position 0.

Datenausgabe auf dem Display

Daten können nun auf das Display übertragen werden, indem anstatt des Befehls die Daten als ASCII-Code gesendet werden.

Der gültige ASCII-Code des Controllers HD44780 ist rechts dargestellt. Wenn zum Beispiel das Zeichen „A“ auf dem Display erscheinen soll, so muss auf dem Port 0 folgende HEX-Zahl übertragen werden:

```
P0=0x41;
```

In C lassen sich die Zeichen automatisch in den ASCII-Code übertragen, wenn die Zeichen in Hochkommata gesetzt werden.

So lässt sich das Zeichen „A“ auch wie folgt übertragen:

```
P0='A';
```

Um die Zeichen nun an das Display zu senden, ist RS=1 und RW=0 zu schalten. Zur Übertragung muss ebenfalls eine fallende Flanke an EN auftreten und eine Wartezeit für die Bearbeitung des Befehls eingehalten werden.

ASCII-Code Tabelle für HD44780:

DB 4-7

DB 0-3	2	3	4	5	6	7	A	B	C	D	E	F
0		0	@	P	`	p		–	タ	ミ	α	p
1	!	1	A	Q	a	q	。	ア	チ	ム	ä	q
2	"	2	B	R	b	r	「	イ	ツ	メ	ß	Θ
3	#	3	C	S	c	s	」	ウ	テ	モ	ε	∞
4	$	4	D	T	d	t	、	エ	ト	ヤ	μ	Ω
5	%	5	E	U	e	u	・	オ	ナ	ユ	σ	ü
6	&	6	F	V	f	v	ヲ	カ	ニ	ヨ	ρ	Σ
7	'	7	G	W	g	w	ァ	キ	ヌ	ラ	q	π
8	(	8	H	X	h	x	ィ	ク	ネ	リ	ɾ	x̄
9	)	9	I	Y	i	y	ゥ	ケ	ノ	ル	⁻¹	y
A	*	:	J	Z	j	z	ェ	コ	ハ	レ	i	千
B	+	;	K	[	k	{	ォ	サ	ヒ	ロ	ˣ	万
C	,	<	L	¥	l	|	ャ	シ	フ	ワ	Φ	円
D	–	=	M	]	m	}	ュ	ス	ヘ	ン	ŧ	÷
E	.	>	N	^	n	→	ョ	セ	ホ	゛	ñ	
F	/	?	O	_	o	←	ッ	ソ	マ	°	ö	■

Im folgenden Beispiel wird „Hallo Welt“ auf das Display geschrieben.

```
#include <t89c51ac2.h>                  // Einbindern der Controllerbibliothek

sbit RS=P3^6;                           // Definition von RS,RW und EN
sbit RW=P3^3;
sbit EN=P3^2;
                                        // Funktionsprototypen
void init2x16();
void schreiben();
void warten();

void main()                             // Hauptprogramm
{
      init2x16();                       // Aufruf der Initialisierungsfunktion
      schreiben();                      // Aufruf der Schreibfunktion
      while(1);                         // Endlosschleife
}

void init2x16()                         // Funktionskopf der Initialisierungsfunktion
{
      char k;                           // interne Zählvariable
                                        // Befehlsliste:
```

```
      unsigned char befehl[7]={0x30,0x30,0x30,0x38,0x0F,0x01,0x06};

      warten();                        // mehr als 15 ms warten

      for (k=0;k<7;k++)                // alle Befehle hintereinander auf P0
      {                                // übertragen
            RS=0; RW=0; EN=1;
            warten();
            P0=befehl[k];
            EN=0;
      }
      warten();
}
void schreiben()                       // Funktionskopf der Schreibfunktion
{
      char k;                          // interne Zählvariable
            // Auszugebene Zeichen. Durch die
            // Anführungsstriche werden diese in den ASCII-Code gewandelt.
      unsigned char text[10]={'H','a','l','l','0',' ','W','e','l','t'};

      for (k=0;k<10;k++)               // alle Zeichen hintereinader auf P0
      {                                // übertragen
            RS=1; RW=0; EN=1;
            warten();
            P0=text[k];
            EN=0;
      }
}

void warten()                          // Funktion für die Verzögerungszeit
{
      int i;
      for(i=0;i<100;i++);
}
```

Übung 8.8

Deaktivieren Sie den blinkenden Cursor am Ende der Ausgabe!

Optimierung der Funktion schreiben

Da es sehr aufwändig und wenig flexibel ist, in der Funktion schreibe ein Array mit dem darzustellenden Text zu definieren, empfiehlt es sich die Funktion allgemein gültig umzuschreiben. Im unten dargestellten Quelltext erwartet die Funktion ein Datenfeld vom Typ char. Der Aufruf der Funktion lautet dann beispielsweise wie folgt:

```
schreiben("Hallo Welt!");
```

Am Ende eines solchen Datenfeldes wird immer \0 generiert. Dies kann genutzt werden, um das Ende des Datenfeldes zu prüfen. Im Beispiel wird solange ein Zeichen nach dem anderen übertragen, bis das Ende des Datenfeldes erreicht ist.

```
void schreiben(char text[])
```

```
{
      int k=0;
      while(text[k]!='\0')
      {
               RS=1; RW=0;EN=1;
               P0=text[k];
            EN=0;
            fertig();
               k++;
      }
}
```

Eine weitere Optimierung liegt darin, anstelle der einfachen Wartefunktion die Funktion fertig() zu verwenden. Diese Funktion fragt das Display ab, ob es noch beschäftigt ist, oder schon bereit ist für die nächste Übertragung (siehe Befehl „Read Busy-Flag Adresse“. Zudem kann der Port P0.7 am Anfang des Programms mit der Variable BSY definiert werden.

```
sbit BSY=P0^7;

void fertig()
{
      RS=0;RW=1;EN=1;
      while(BSY);
}
```

Übung 8.9

Realisieren Sie das Programm „Hallo Welt“ mit den oben beschriebenen Optimierungen.

Ausgabe von zweistelligen Zahlen

Bei der Ausgabe von Integer- oder Char-Zahlen kann der Compiler die Daten nicht automatisch in den ASCII-Code umwandeln, da sie einen anderen Datentyp haben. Der HEX-Code muss dann berechnet und ausgegeben werden. Da die Zahlen in der ASCII-Tabelle (siehe oben) ab 0x30 beginnen, wird zu dieser HEX-Zahl die entsprechende Ziffer addiert. Die einzelnen Ziffern können wie bei der 7-Segmenanzeige bestimmt werden. Die folgende Funktion zeigt die Ausgabe von Zahlen mit zwei Ziffern/Stellen.

```
void ausgabe_zahl_2Stellen(char zahl)      // Funktionskopf
{
      char zehner,einer;                   // Deklaration der Variablen
      fertig();                            // Warten bis LCD fertig ist.

      zehner=zahl/10;                      // Umwandeln in einzelne Ziffern
      einer=zahl-(10*zehner);

      RS=1; RW=0; EN=1;                    // LCD für den Datentransfer vorbereiten

      P0=zehner+0x30;                      // Zehner übertragen
      EN=0;

      warten();
```

```
      RS=1; RW=0; EN=1;                    // LCD für den Datentransfer vorbereiten

      P0=einer+0x30;                       // Einer übertragen
      EN=0;

      fertig();                            // Warten bis LCD fertig ist.
}
```

Übung 8.10

Lassen Sie auf dem LC-Display wie bei dem Beispielprogramm der Siebensegmentanzeige Zahlen von 0 bis 99 hochzählen.

Hinweis: Sie können das Display nach jeder Ausgabe neu initialisieren, damit die Zahlen immer an den Anfang geschrieben werden.

Positionierungen auf dem Display

Damit das Display bei jeder Ausgabe nicht immer neu initialisiert wird, kann mit dem Display Befehl „Set DDRAM Adress“ die Schreibposition der Datenausgabe für eine beliebige Positionierung festgelegt werden.

Befehl	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Funktion
Set DDRAM Adress	0	0	**1**	A6	A5	A4	A3	A2	A1	A0	Setzt die Schreibadresse des Datenspeichers fest.

Über die Adresse A0 bis A6 (7 Bit) ist die Schreibposition des Datenspeichers festgelegt. Beim zweizeilige Display ist die Adressaufteilung (Positionierung) der ersten Zeile von 0x00 bis 0x27 und bei der zweiten Zeile von 0x40 bis 0x67 festgelegt.

Die 0. Position der zweiten Zeile (0x40 Hex) lautet binär codiert also 0100 0000.

Da der Datenspeicher nur 7 Bit lang ist, wird das höchste Bit verwendet, um dem LC-Display mitzuteilen, dass es sich um den Befehl „Set DDRAM Adress“ handelt.

Um die Positionierung nun in die zweite Zeile zu setzen, muss dem Display die Kombination 1100 0000 mitgeteilt werden. Dies geschieht, indem zu der Adresse der HEX-Wert 0x80 (Binär 100 0000) addiert wird.

Positionierung 0. Pos in Zeile 2:

```
  0100 0000 (0x40)
+ 1000 0000 (0x80)
  ---------------
  1100 0000 (0xC0)
```

Die weiteren Stellen können einfach auf die Adresse addiert werden.

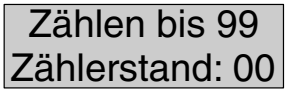
In C sieht eine solche Funktion beispielshaft so aus:

```
void pos(char zeile, char stelle)
{
      if (zeile==2)              // Wenn die Zeile = 2 sein soll,
      {
            zeile=0x40;          // wird die Variable zeile mit der
      }                          // Adresse 0x40 vorbelegt.
```

```
      else
      {
            zeile=0x00;          // ansonsten mit 0x00;
      }
      RS=0; RW=0; EN=1;          // Vorbereitung, um einen Befehl zu senden.

      P0=0x80+zeile+stelle;      // Befehl berechnen und übertragen.

      EN=0;
      warten();                  // Warten, bis Befehl übertragen ist.
}
```

Übung 8.11

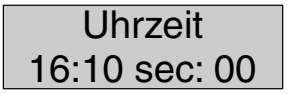
Ändern Sie das Zählprogramm dahingehend ab, dass auf dem Display folgendes ausgegeben wird:



Entwerfen Sie zusätzlich eine Funktion void sonderzeichen(char zeichen), um das Sonderzeichen „ä“ darzustellen.

Übung 8.12

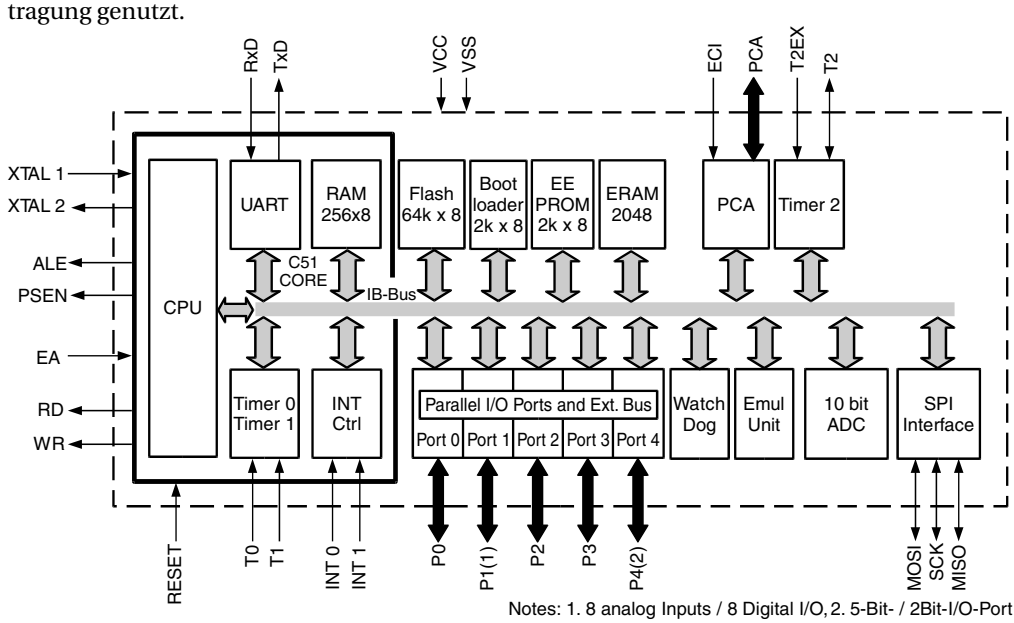
Schreiben Sie ein Programm, an dem die Uhrzeit auf dem Display dargestellt wird, unter Verwendung des Programms Binäruhr.



9 Controller Erweiterungen

Zur Übersicht sei nochmals das Blockschaltbild von Kapitel 2 dargestellt:

Bis jetzt wurden am Mikrocontroller die CPU, der RAM-Speicher, der Flash-Speicher für das Programm, der Boot-Loader für die Programmübertragung, die parallelen Ports (Parallel I/O Ports) für die Ein- und Ausgabe und die serielle Schnittstelle (UART) für die Programmübertragung genutzt.



Notes: 1. 8 analog Inputs / 8 Digital I/O, 2. 5-Bit- / 2Bit-I/O-Port

Im Weiteren soll nun der Timer 0-/Timer 1-Block, der Analog-Digitalwandler (10 bit ADC)-Block, der Interrupt (INT Ctrl)-Block und weitere Teile des seriellen-Schnittstellen (UART)-Blocks erläutert werden.

Hinweis: Bei allen Erweiterungen wurde davon ausgegangen, dass der Controller 12 Takte für einen Befehlszyklus benötigt. Wird über X2 beim Controller AT89C51AC3 eine Befehlszykluszahl von 6 Takten eingestellt, so werden alle Funktionserweiterungen mit einem doppelten Takt gespeist.

Erweiterungen:

Timer 0 / Timer 1
Zähler bzw. Zeitgeber

10 Bit ADC
Analog Digital Wandler

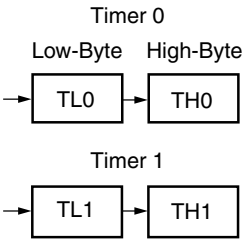
INT Ctrl
Interrupts

UART
Serielle Schnittstelle

10 Der Zähler/Zeitgeber Timer 0 und Timer 1

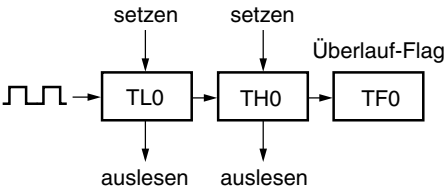
Timer 0 und Timer 1 bestehen aus jeweils zwei 8-Bit-Vorwärtszählern. Die Zähler lassen sich, je nach gewähltem Modus, als 8-Bit oder als 16-Bit-Zähler verwenden. Die 16-Bit-Zähler ergeben sich aus der Reihenschaltung zweier 8-Bit-Zähler.

Die 8-Bit-Zähler-Register sind die Spezial-Funktions-Register TL0 und TH0 für Timer 0 und TL1 und TH1 für Timer 1.



Die Zähler-Register lassen sich auf Anfangswerte setzen. Ab diesen Werten zählen sie nach dem Starten des Zählers aufwärts. Läuft der Zähler über, wird das Überlauf-Flag gesetzt. Nach einem Überlauf lässt sich der Zähler wieder auf einen Anfangswert setzen. Geschieht dies nicht, beginnt er bei dem Wert null.

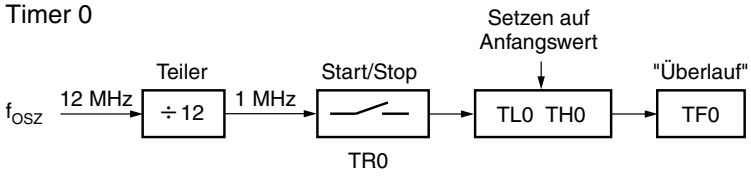
Timer 0 als 16-Bit-Zähler:



Das Überlauf-Flag kann einen Interrupt auslösen oder vom Programm abgefragt werden. Bei einer Abfrage per Programm ist es auch durch das Programm wieder zu löschen.

Der jeweilige Zählerstand lässt sich auch aus den Registern auslesen.

10.1 Einsatz der Timer als Zeitgeber



Bei einer Verwendung der Timer als Zeitgeber werden sie vom Systemtakt hochgezählt. Die Periodendauer des Systemtaktes wird über den Zähler addiert. Je nach Anfangswert des Zählers

ergibt sich bis zum Überlauf eine bestimmte Zeitverzögerung. Diese Zeitverzögerung ist nicht abhängig vom laufenden Programm. Der Timer läuft, getaktet vom Systemtakt, als eigenständige Hardware-Baugruppe. Die Zeitverzögerung hängt also nur von der Quarzfrequenz ab.

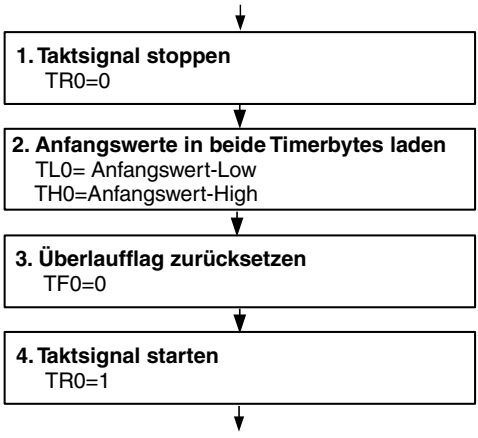
Wird beim Controller Atmel AT89C51AC3 über X2 eine Befehlszykluszahl von 6 Takten gewählt, dividiert der vorgeschaltete Teiler anstelle durch 12 durch 6.

Läuft der Zähler über, setzt er sein Überlauf-Flag TF0 oder TF1 auf 1. Damit lässt sich ein Interrupt auslösen, mit dem das laufende Programm unterbrochen und ein Unterprogramm als Interrupt-Service-Routine gestartet werden kann. Weiteres hierzu im Kapitel 13 (Das Interrupt-System).

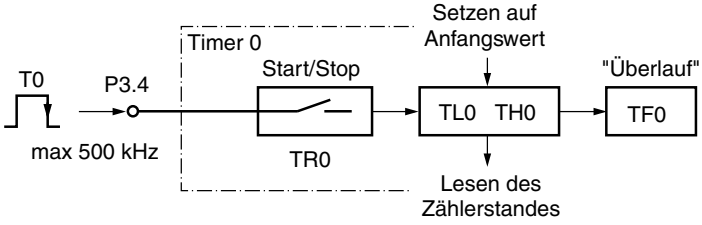
Wird nicht im Interruptbetrieb gearbeitet, wird das Überlauf-Flag bei jedem Programmzyklus abgefragt. Ist es gesetzt, wird darauf reagiert und das Flag zurückgesetzt.

Die Überlauf-Flags sind Bits in bestimmten Spezial-Funktions-Registern. Sie sind bitadressierbar.

Einstellen der Zeit und starten: z. B. Timer 0



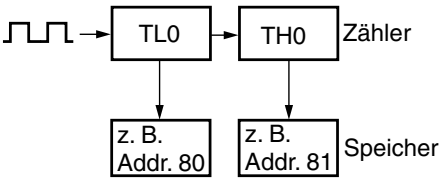
10.2 Einsatz der Timer als Ereigniszähler



Werden die Timer als Ereigniszähler eingesetzt, erhalten sie ihre Zählimpulse von einem externen Geber. Es werden die negativen Flanken der Eingangsimpulse gezählt. Dabei arbeiten die Zähler unabhängig vom laufenden Programm.

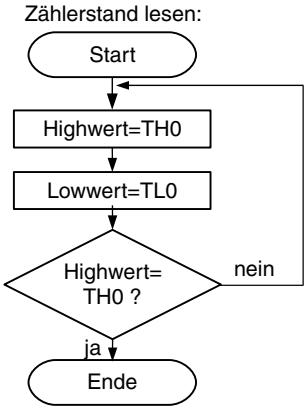
Lesen des Zählerstandes

Der 16-Bit-Zähler wird aus zwei Registern gebildet, die nacheinander auszulesen sind. Das ist kein Problem, wenn der Zähler während des Auslesens über TR0 angehalten werden kann.



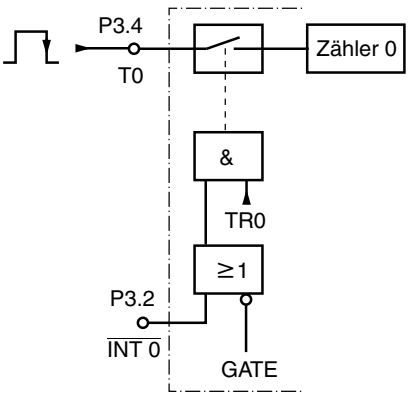
Will man bei laufendem Zähler den Zählerstand auslesen, ist zu berücksichtigen, dass während des Auslesens Zählimpulse eintreffen können, die den Zählerstand verändern. Es ist daher zu kontrollieren, ob während des Auslesens ein Übertrag in das höherwertige Byte aufgetreten ist. Ist das der Fall, ist das Lesen zu wiederholen.

Programm zum Auslesen des Zählerstandes des Timers 0:



Freigabe des Zählers

Am Beispiel des Timers 0:
Der Zähler lässt sich über die Bits TR0 und GATE in Spezial-Funktions-Registern oder über ein externes Signal an Portpin P3.2 freigeben.

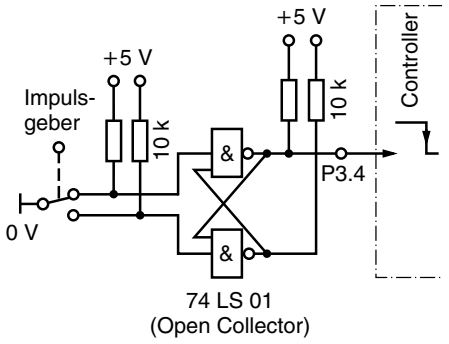


Interne Freigabe: `GATE = 0; TR0 = 1`
Externe Freigabe über P3.2 = 1: `GATE = 1; TR0 = 1`

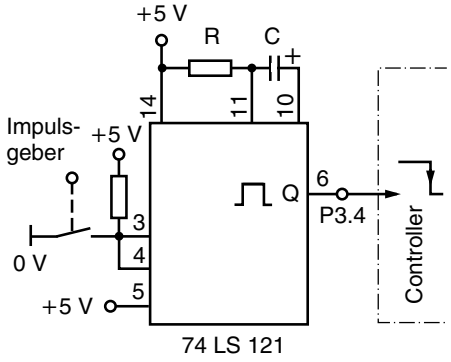
Entprellen des Zählimpulses

Kommt der Eingangsimpuls für den Zähler von einem mechanischen Kontakt, muss er entprellt werden. Das kann nicht per Software geschehen, sondern durch eine Hardware vor dem Controller-Eingang. Damit lassen sich auch kurze Störimpulse herausfiltern.

Entprellen mittels eines Flipflops:



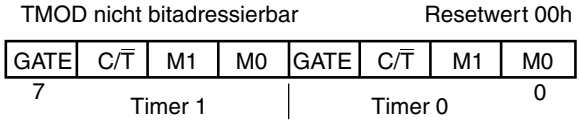
Entprellen über eine monostabile Kippstufe:



10.3 Einstellen der Timer-Funktion

Die Timer-Funktion wird mithilfe des Timer-Modus-Registers TMOD eingestellt.

Timer-Modus-Register TMOD:

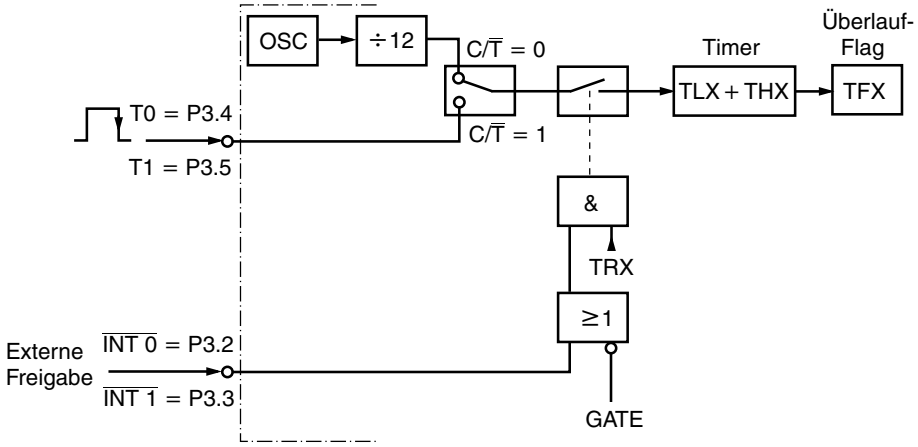


Bedeutung der einzelnen Bits des Timer-Modus-Registers:

Bit		Funktion
GATE		Umschaltung interne/externe Freigabe:
0		Interne Freigabe über TR0 bzw. TR1
1		Externe Freigabe über P3.2 bzw. P3.3 plus interne Freigabe über TR0 bzw. TR1.
C/$\overline{T}$		Umschaltung Zeitgeber oder Zähler:
0		Funktion als Zeitgeber
1		Funktion als Zähler
M1	M0	Wahl des Arbeitsmodus: (Es werden nur die zwei gebräuchlichen Arbeitsmodi beschrieben)
0	1	Modus 1: TL0 und TH0 bzw. TL1 und TH1 bilden einen 16-Bit-Zähler
1	0	Modus 2: TL0 bzw. TL1 bilden einen 8-Bit Auto-Reload-Zähler. Bei Überlauf wird der im High-Byte TH0 bzw. TH1 stehende Wert in das Low-Byte kopiert. Das High-Byte bleibt unverändert.

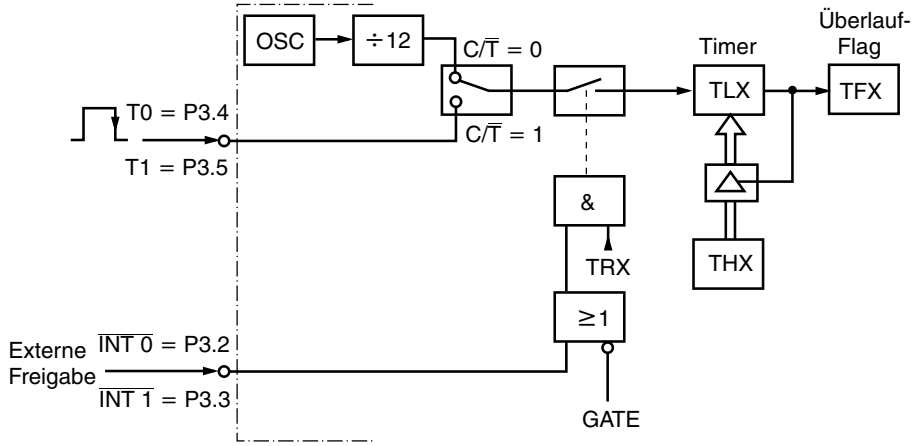
Blockschaltbild der Timer 0 und 1 in Modus 1

16-Bit-Timer/Counter:



Blockschaltbild der Timer 0 und 1 in Modus 2

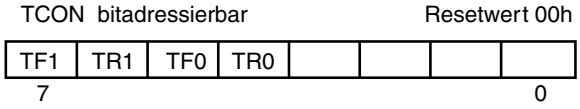
8-Bit-Timer/Counter mit Auto-Reload:



10.4 Steuern der Timer

Die Timer werden über die Bits TF und TR im Timer-Control-Register TCON gesteuert.

Timer-Control-Register TCON:



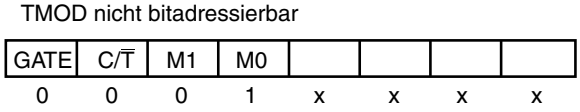
Bedeutung der einzelnen Bits des Timer-Control-Registers:

Bit	Funktion
TF0	Timer-Überlauf-Flag:
TF1	Das Flag wird beim Überlauf des Timers gesetzt. Beim Einsprung in die zugehörige Interrupt-Adresse wird es automatisch rückgesetzt. Bei Abfrage im Programmzyklus muss es durch das Programm gelöscht werden.
TR0	Timer-Run-Flag = Timer Freigabe:
TR1	Das Flag muss durch das Programm gesetzt oder gelöscht werden.
1	Timer startet
0	Timer stopt

Beispiel 10.1

Initialisieren Sie den Timer 1 als 16-Bit-Zeitgeber. Timer 0 ist nicht zu beeinflussen.

Timer-Modus einstellen:



Da TMOD nicht bitadressierbar ist, und trotzdem einzelne Bits geändert werden sollen, ohne die untersten vier Bit zu beeinflussen, muss die Zuordnung über eine Maskierung geschehen.

Es werden zunächst die oberen vier Bit komplett auf Null gesetzt. Dies geschieht über eine bitweise UND-Verknüpfung mit 0x0F (Hex). Dabei bleiben die unteren 4 Bit erhalten. (x kennzeichnet beliebige Bitkombinationen)

```
    xxxx xxxx
&   0000 1111 (UND-Verknüpfung)
    ---------
    0000 xxxx (Ergebnis)
```

Anschließend wird dem Byte durch eine bitweise ODER-Verknüfung mit 0x10 (Hex) die endgültige Bitkombination zugeordnet.

```
    0000 xxxx
>=  0001 0000 (ODER-Verknüpfung)
    ---------
    0001 xxxx (Ergebnis)
```

In C sieht das wie folgt aus:

```
TMOD=((TMOD&0x0F)|0x10)
```

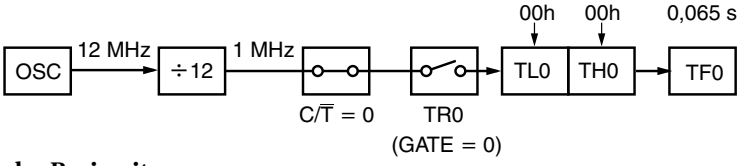
Übung 10.1

Initialisieren Sie Timer 0 als 16-Bit-Zähler. Dabei darf Timer 1 nicht beeinflusst werden. Es sollen externe Impulse gezählt werden. Die Eingangsimpulse vom Geber sind über eine monostabile Kippstufe entprellt.

10.5 Anwendung als Zeitgeber

Timer 0 soll als Zeitgeber für ein Lauflicht eingesetzt werden. Die Zeiten sollen einstellbar sein.

Die Basiszeit liefert der 16-Bit-Timer 0.



Berechnung der Basiszeit:

Der Takt (hier 12 MHz) wird durch zwölf geteilt. Die resultierende Periodendauer T dieses Signal beträgt $T = 1/f = 1/1\,\text{MHz} = 1\,\mu\text{s}$. Dies bedeutet, dass jede µs der Zähler um Eins erhöht wird. Da es sich um einen 16-bit-Zähler handelt, läuft dieser bei $2^{16} = 65\,536$ über.

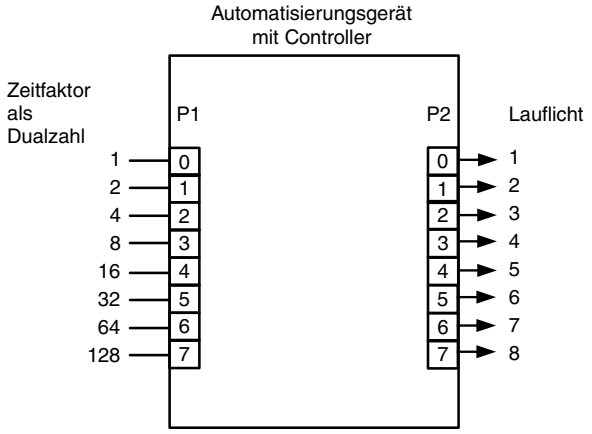
Beim Laden des Timers mit der Zahl 0000 (HEX) ergibt sich somit bis zum Überlauf eine Zeit von 0,065 536 s, wenn der Controller mit einem 12-MHz-Quarz arbeitet.

Dieser Basiswert wird mit einem einstellbaren Zeitfaktor multipliziert.

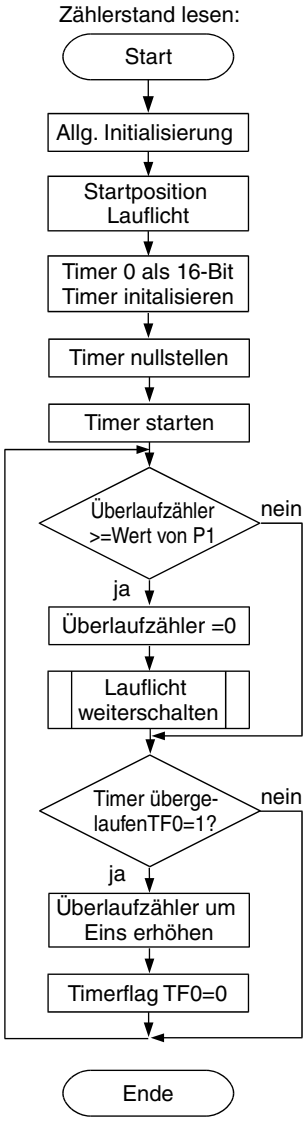
$$\text{Zeit} = \text{Zeitfaktor} \times \text{Basiswert}$$

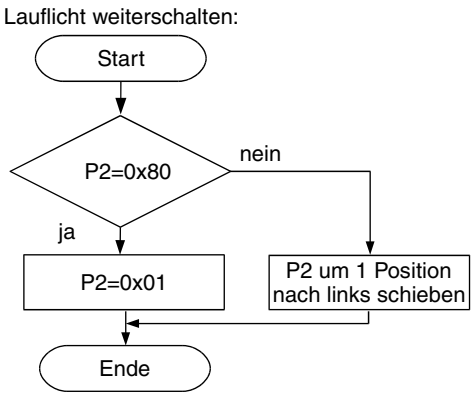
Dazu müssen die Timer-Überläufe gezählt werden. Stimmt der Zählerstand des Überlaufzählers mit dem eingestellten Zeitfaktor überein, ist die Zeit erreicht, und das Lauflicht schaltet um eine Stelle weiter.

Anschlussplan:



Programmablaufplan:





Erläuterung zum Unterprogramm Lauflicht weiterschalten:

Zuerst wird überprüft, ob die letzte LED (Bit 7 von Port 2) leuchtet.

Ist dies der Fall, wird das Lauflicht wieder in die Startposition geschrieben.

Wenn es noch nicht am Ende angekommen ist, wird das komplette Byte um eine Stelle nach links verschoben.

Umsetzung in C:

```
/*
      Einstellbares Lauflicht an Port 2
      Datei: Lauflicht_BSP_9_1_5.c
*/

#include<t89C51ac2.h>                  // Einbinden der Bibliothek des
                                       // Controllers.
```

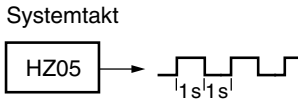
```
void Lauflicht_weiterschalten();       // Funktionsprototyp zum Weiterschalten
                                       // des Lauflichts

void main()                            // Funktionskopf des Hauptprogramms
{
      unsigned char UZaehler=0x00;     // Variablendeklaration des
                                       // Überlaufzählers

      P1=0xFF;                         // Port 1 zum Einlesen vorbereiten
      P2=0x01;                         // Startposition Lauflicht
      TMOD=((TMOD&0xF0)|0x01);         // Timer 0 als 16-Bit Timer
                                       // initialisieren
      TL0=0x00;                        // Timer nullstellen
      TH0=0x00;

      TR0=1;                           // Timer starten
      while (1)                        // Endlosschleife für die zyklische
                                       // Programmbearbeitung
      {
            if (UZaehler>=P1)          // Wenn der Überlaufzähler >= dem Wert
                                       // des Ports 1 ist,
            {
                  UZaehler=0;          // wird der Überlaufzähler auf 0
                                       // gesetzt.
                  Lauflicht_weiterschalten();  // Lauflicht eine Position
                                       // weiterschalten
            }

            if (TF0==1)                // wenn der Timer0 überläuft,
            {
                  UZaehler++;          // wird der Überlaufzähler erhöht
                  TF0=0;               // und das Timerflag 0 gelöscht.
            }
      }
}


void Lauflicht_weiterschalten()        // Unterprogramm Lauflicht_weiterschalten
{
      if (P2==0x80)                    // Wenn Lauflicht am Ende:
      {
            P2=0x01;                   // Lauflicht zurücksetzen
      }
      else                             // sonst
      {
            P2=P2<<1;                  // um eine Stelle nach links schieben.
      }
}
```

Beispiel 10.2

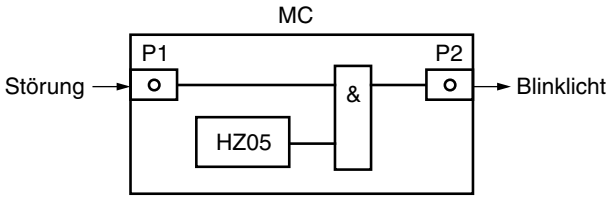
Systemtakt 0,5 Hertz

Bei vielen Automatisierungsgeräten wird dem Anwender ein Systemtakt mit einer bestimmten Frequenz zur Verfügung gestellt. Damit lassen sich Blinklichter, Warnhupen oder Zeitzähler ansteuern. Der Systemtakt wird von einem bestimmten Merkerbit ausgegeben, welches seinen Zustand mit einer vorgegebenen Frequenz ändert.

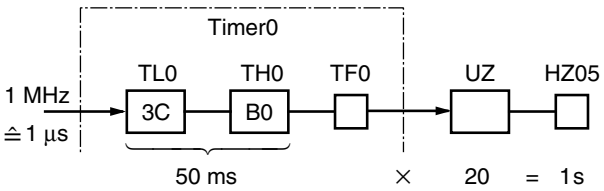
Hier soll ein Systemtakt von 0,5 Hz erzeugt werden. Das Merkerbit mit dem symbolischen Namen HZ05 muss dazu seinen Zustand jede Sekunde ändern.



Als Anwendung soll der Systemtakt HZ05 bei einer eintreffenden Störung am Port P1.0 ein Blinklicht an P5.0 einschalten.



Erzeugung des Systemtaktes HZ05

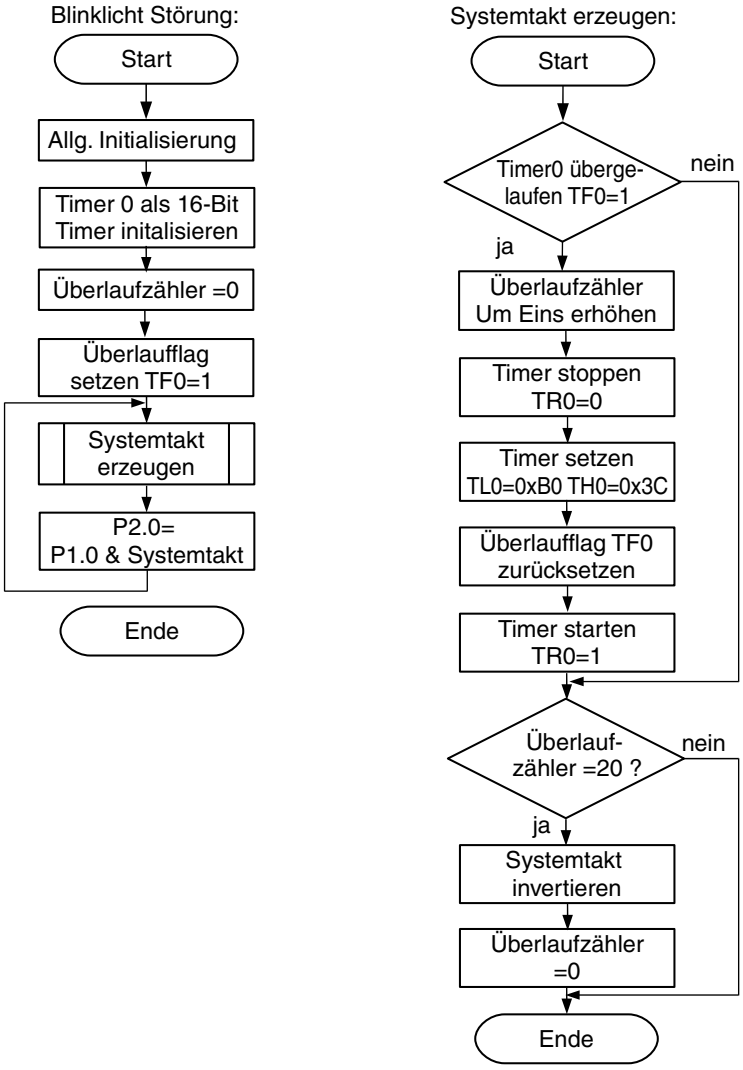


Um eine Zeit von 50 ms zu erhalten, muss der Timer auf die Dezimalzahl 15 536 dezimal oder 3C B0 hex gesetzt werden.

$$65\,536 - 50\,000 = 15\,536$$

Die 50 ms sind mit 20 zu multiplizieren, um auf eine Sekunde zu kommen. Dazu wird der Überlaufzähler mit dem symbolischen Namen UZ vom Überlaufflag TF0 von 0 bis 20 hochgezählt. Ist die 20 erreicht, wird das Bit HZ05 invertiert.

Programmablaufplan für das Programm „Blinklicht Störung“ und das Unterprogramm „Systemtakt erzeugen“:



Im Hauptprogramm wird zu Beginn softwaremäßig das Timerflag TF0 gesetzt, damit der Timer direkt im Unterprogramm starten kann. Ansonsten würde der Timer nie gestartet werden.

Übung 10.2

Setzen Sie die beiden Programmablaufpläne in C um. Die Variable Systemtakt soll dabei global sein und im gesamten Programm zur Verfügung stehen.

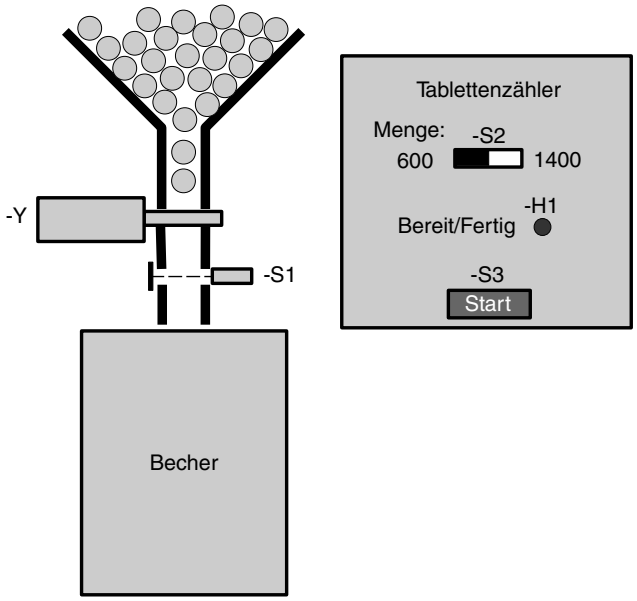
10.6 Anwendung als Ereigniszähler

Manuelle Tablettenabfüllanlage

In einer Chemiefabrik für medizinische Tabletten werden für den Großkunden zwei unterschiedliche Größen von Verpackungen angeboten. Die kleine Packung enthält 600 Tabletten und die große Packung 1400 Tabletten.

Da es sehr aufwändig ist, die Tabletten einzeln zu zählen, soll eine manuelle Tablettenabfüllanlage mithilfe eines Mikrocontrollers programmiert werden.

Technologieschema der Abfüllanlage:



Funktionsweise:

Zunächst stellt der Anwender einen Becher unter die Abfüllanlage. Mit -H1 (P2.1) wird angezeigt, dass die Anlage bereit ist. Anschließend wählt er über -S2 (P1.1) aus, ob 600 (S2=0) oder 1400 Tabletten (S2=1) abgefüllt werden sollen. Im nächsten Schritt betätigt der Anwender die Start-Taste -S3 (P1.0). Sobald die Anlage im Zählmodus ist, erlischt die Signalleuchte -H1. Dann wird automatisch der Zylinder -Y (P2.0=0) geöffnet. Der Lichtsensor -S1 gibt für jede Tablette einen Impuls zum Port P3.5 des Mikrocontrollers. Sobald 1400 bzw. 600 Tabletten gezählt wurden, wird der Zylinder -Y geschlossen (P2.0=1). Mit -H1 (P2.1) wird angezeigt, ob der Zählvorgang vorbei ist.

Die oben beschriebene Situation soll unter Verwendung der Timer-/Zähler-Funktionseinheit realisiert werden.

Timer/Zähler einstellen und auswählen:

Es muss Timer 1 als 16-Bit-Zähler gewählt werden, da dieser hardwaretechnisch mit Port P3.5 verbunden ist und mehr als 256 Zählvorgänge nötig sind (8-bit-Zähler reicht nicht aus). TMOD ist somit wie folgt zu initialisieren:

TMOD nicht bitadressierbar

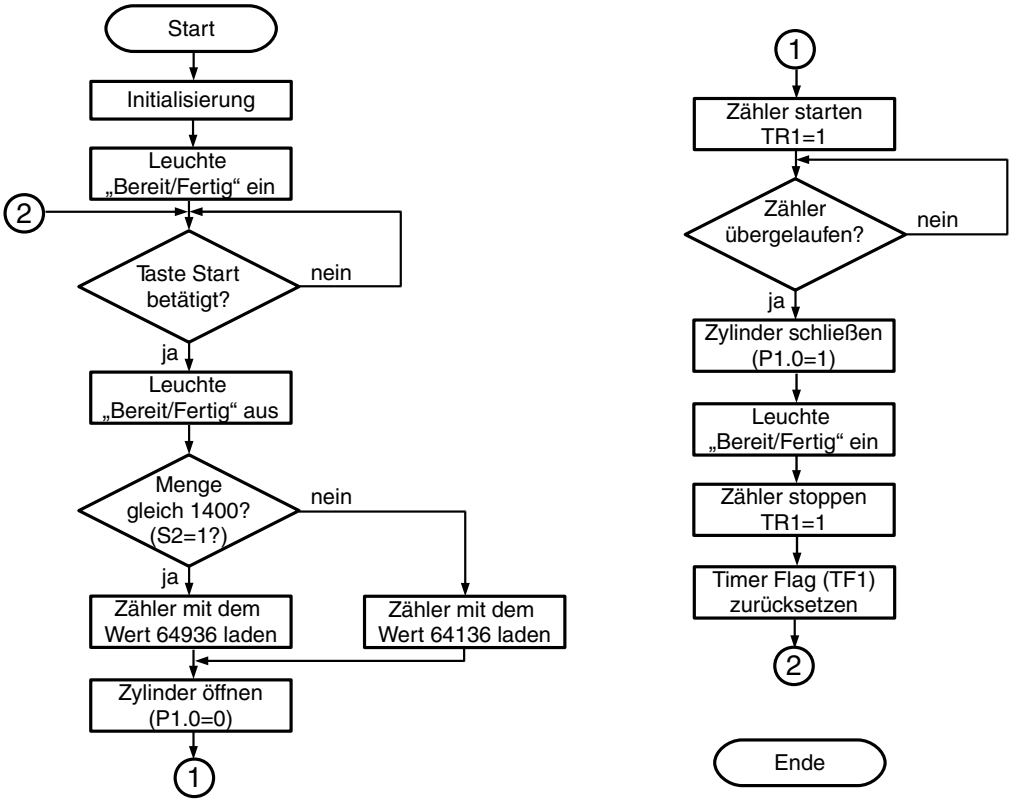
GATE	C/T	M1	M0				
0	1	0	1	x	x	x	x

Zudem müssen die Anfangswerte berechnet werden:

600 Tabletten: Zähler vorladen mit $65\,536 - 1400 = 64\,136 \rightarrow$ Hex: FA88

1400 Tabletten: Zähler vorladen mit $65\,536 - 600 = 64\,936 \rightarrow$ Hex: FDA8

Programmablaufplan:



Realisierung in C:

```
#include <t89c51ac2.h>              // Einbinden der Bibliothek

sbit S3=P1^0;                       // Initialisierung der benötigten
sbit S2=P1^1;                       // Ein- und Ausgänge
```

```
sbit y=P2^0;
sbit H1=P2^1;


void main(void)                    // Hauptprogramm
{
      TMOD=(TMOD&0x0F)|0x50;       // Initialisierung des Zählers
                                   // ohne T0 zu beeinflussen
      H1=1;                        // Lampe ein



      while(1)                     // Endlosschleife
      {
            while(S3==0);          // Warten bis Taster S3 betätigt wird
            H1=0;                  // Lampe aus
            if (S2==0)             // Wenn S2 =1,
            {
                  TH1=0xFD;        // Zähler mit 64936 (0xFDA8 HEX) laden
                  TL1=0xA8;
            }
            else                   // ansonsten
            {
                  TH1=0xFA;        // Zähler mit 64136 (0xFA88 Hex) laden
                  TL1=0x88;
            }
            y=0;                   // Zylinder einfahren
            TR1=1;                 // Timer1 starten
            while(TF1==0);         // Warten bis Timer1 überläuft
            y=1;                   // Zylinder wieder ausfahren
            H1=1;                  // Lampe ein
            TR1=0;                 // Timer1 stoppen
            TF1=0;                 // Timer-Flag 1 wieder zurücksetzen.
      }
}
```

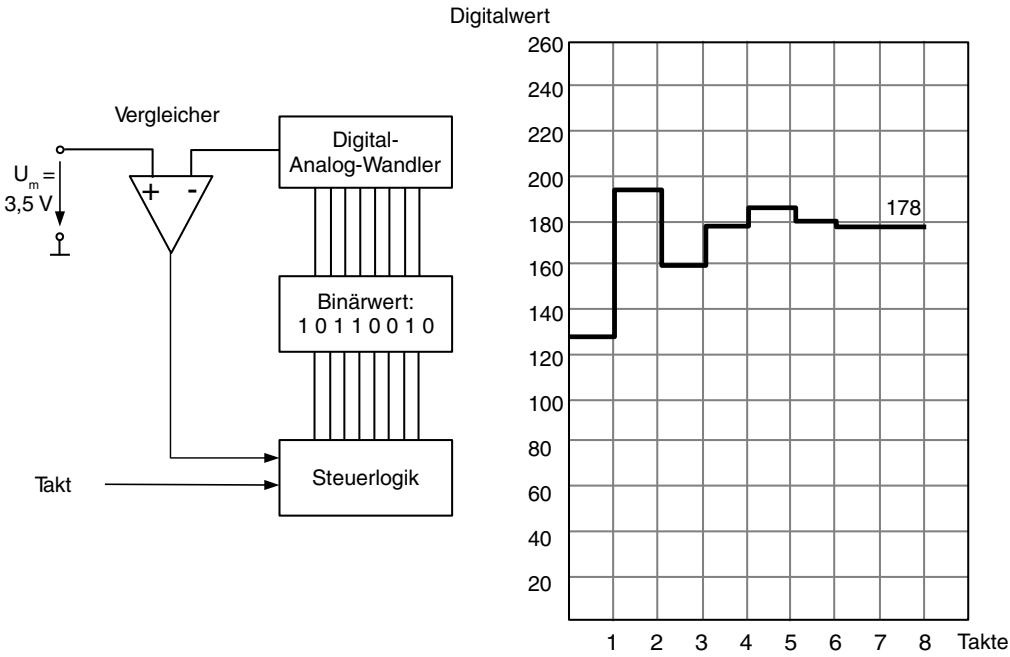
11 Der Analog/Digital-Wandler

Mithilfe eines Analog/Digital-Wandlers werden analoge Spannungen in digitale Werte umgewandelt, die im Mikrocontroller dann weiter verarbeitet werden können. Es lässt sich beispielsweise die Temperatur über einen temperaturabhängigen Widerstand messen. Die unterschiedlichen Spannungen einer dazugehörigen Messschaltung werden im Controller zum Beispiel in °C umgerechnet und auf einem LC-Display ausgeben.

Der Digitalwert wird meistens nach dem Prinzip der sukzessiven Approximation (Wägeverfahren) ermittelt.

Die Funktionsweise des Wägeverhaltens soll anhand eines Analog/Digital-Wandlers gezeigt werden, der Eingangsspannungen von 0 V bis 5 V in eine 8-Bit-Digitalzahl umwandelt. Am Eingang liegt in diesem Beispiel eine Spannung von 3,5 V an.

Funktionsweise des Wägeverfahrens



Das Wägeverfahren benötigt einen Vergleicher, einen Digital/Analog-Wandler und eine Steuerlogik. Zunächst werden über die Steuerlogik alle Bits des zu ermittelnden Binärwertes mit 0 deaktiviert. Probeweise wird dann das höchste Bit zunächst auf 1 gesetzt. Der Binärwert

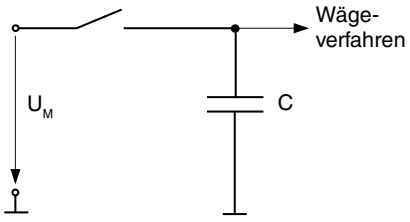
1000 0000 entspricht dem Dezimalwert 128 und der Analogspannung von 2,5 V. Diese mit dem Digital/Analog-Wandler erzeugte Spannung wird mit der Messspannung von 3,5 V verglichen. Da sie größer ist, bleibt die 1 des höchsten Bits gesetzt. Im nächsten Schritt wird probeweise das nächstniedrigere Bit 1 gesetzt. Der Binärwert 1100 0000 entspricht dem Dezimalwert 192 und der Analogspannung 3,75 V. Diese Spannung wird nun mit dem Messwert verglichen. Diesmal ist die Eingangsspannung niedriger, deshalb wird dieses Bit wieder auf Null gesetzt. Anschließend wird wieder das nächstniedrigere Bit probeweise auf 1 gesetzt. Der Binärwert 1010 0000 entspricht dem Dezimalwert 160 und der Analogspannung 3,125 V. Nun ist die Spannung wieder niedriger als die Messspannung, somit bleibt die 1 erhalten. So wird dann schrittweise weiter verfahren. Am Ende entsteht der Binärwert 1011 0010, der dem Dezimalwert 178 und der Messspannung 3,5 V entspricht. Die Wandlung dauert also maximal 8 Takte und hat eine Genauigkeit von 8 Bit.

Die Genauigkeit von 8-Bit sagt aus, dass der analoge Wert in 256 Stufen erfasst werden kann, welches bei einer 5-V-Referenzspannung ungefähr 20 mV pro Stufe bedeutet.

$$\frac{5\,\mathrm{V}}{255} = 0{,}019\,60\,\mathrm{V} \approx 20\,\mathrm{mV}$$

„Sample and Hold“-Schaltung

Vor dem eigentlichen Analog/Digital-Wandler mit dem Wägeverfahren ist eine sogenannte „Sample and Hold“-Schaltung verbaut. In der „Sample and Hold“-Schaltung lädt sich ein Kondensator über die Eingangsspannung auf. Anschließend wird die Eingangsspannung abgetrennt und der Kondensator hält nun die Spannung für den A/D-Wandler. Diese Stufe ist notwendig, um erhebliche Messfehler zu vermeiden, die sich bei einer Veränderung des Analogwertes während der Messung ergeben würden.



Referenzspannung

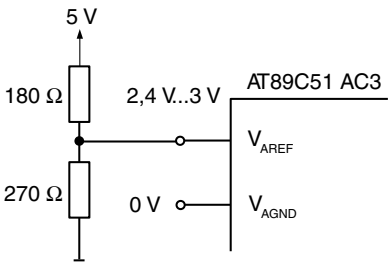
Mit der Referenzspannung wird die maximale interne Vergleichsspannung des Wägeverfahrens festgelegt. Die Genauigkeit des ermittelten digitalen Wertes ist abhängig von der Genauigkeit dieser Spannung. In den meisten Fällen reicht es, die Spannung von der Versorgungsspannung zu entnehmen.

Leider ist beim Controller AT89C51AC3 die maximale Referenzspannung auf 3 V festgelegt, sodass diese entweder über einen Spannungsregler oder über einen Spannungsteiler erzeugt werden muss.

Beim Spannungsteiler sollten relativ kleine Widerstandswerte verwendet werden, da der interne Widerstand zwischen 12 kΩ und 24 kΩ liegt und so die gewünschte Spannung besser erreicht wird.

Liegt die Referenzspannung bei 3 V, so reduziert sich die Schrittweite zwischen den einzelnen Stufen auf ungefähr 11,8 mV.

Bei vielen anderen Controllern liegt die maximale Referenzspannung bei der Versorgungsspannung (hier 5 V), sodass der Spannungsteiler entfällt.



Schrittweite bei 3 V (8-Bit-Wandler):

$$3\,\text{V} : 255 = 11{,}76\,\text{mV}$$

Optimierung der Messgenauigkeit

Um die Messgenauigkeit zu optimieren, gibt es zwei Möglichkeiten. Entweder wird anstelle des 8-Bit-Wandlers ein Wandler mit mehr Binärstellen verwendet, oder es wird der Bereich der Vergleichsspannung mittels einer sogenannten Spannungslupe reduziert.

Erhöhung der Binärstellen

Beim AT89C51AC3-Mikrocontroller kommt immer ein 10-Bit-Wandler zum Einsatz. Zusätzlich lässt sich ein sogenannter „Precision-Mode“ einstellen. Dieser Modus stoppt die CPU, um ein mögliches Rauschen zu reduzieren.

Die Schrittweite bei einem 10-Bit-Wandler mit einer Referenzspannung von 3 V verkleinert sich auf 2,93 mV, sodass wesentlich genauer gemessen werden kann.

10-Bit-Wandler: $2^{10} - 1 = 1023$ Schritte

$$\rightarrow \frac{3\,\text{V}}{1023} = 2{,}93\,\text{mV}$$

Spannungslupe

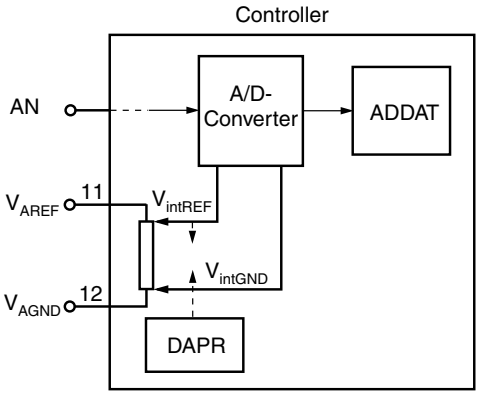
Die Methode der Vergrößerung der Binärstellen beim Wandler wird heute meistens bevorzugt, da sich hierdurch ein genauerer oder gleicher Präzisionsgrad ergibt. Trotzdem soll anhand eines älteren Controllers von Siemens (Typ: SAB 80C535) gezeigt werden, wie das Prinzip der Spannungslupe funktioniert.

Hierbei kann *V*AGND angehoben und *V*AREF abgesenkt werden. Dies geschieht per Programm durch Einstellungen in einem Register DAPR (D/A-Converter Programming).

Mit DAPR stellt sich die interne Referenzspannung ein, aus der sich die Vergleichsspannung ableitet.

Die Genauigkeit wird größer, weil der verringerte Referenz-Spannungsbereich wieder in 255 Stufen geteilt wird.

Soll die interne Verringerung der Referenzspannung nicht genutzt werden, ist der Wert 00h in das Register DAPR zu schreiben. Mit dem Einschreiben des Wertes in das Register DAPR wird die Wandlung gestartet.



Über je vier Bit im Register DAPR lassen sich V_{int} REF und V_{intGND} einstellen.

DAPR nicht bitadressierbar Resetwert 00h

7							0
V_{intREF}				V_{intGND}			

Stufe 0 =	0	0	0	0	0	0	0	0
1 =	0	0	0	1	0	0	0	1
14 =	1	1	1	0	1	1	1	0
15 =	1	1	1	1	1	1	1	1

Spannung pro Stufe: $\frac{5\,\text{V}}{16} = 0{,}3125\,\text{V}$

Es lassen sich jedoch nicht alle Stufen nutzen. Außerdem soll die Spannungsdifferenz zwischen oberer und unterer Grenze mindestens 1 V betragen.

Tabelle der einstellbaren Spannungen:

Stufe	V_{intREF} in V	V_{intGND} in V
0	5,0	0,0
1	–	0,3125
2	–	0,625
3	–	0,9375
4	1,25	1,25
5	1,5625	1,5625
6	1,875	1,875
7	2,1875	2,1875
8	2,5	2,5
9	2,8125	2,8125
10 = Ah	3,125	3,125
11 = Bh	3,4375	3,4375
12 = Ch	3,75	3,75
13 = Dh	4,0625	–
14 = Eh	4,375	–
15 = Fh	4,6875	–

Beispiel 11.1

Ein analoger Geber verändert seine Spannung von 1 V bis 3 V. Um die mögliche Genauigkeit auszunutzen, wird die interne Referenzspannung diesem Bereich angepasst.

Gewählt:

$$V_{\text{intGND}} = 0{,}9375\,\text{V} = \text{Stufe } 10 = \text{A}$$

$$V_{\text{intREF}} = 3{,}125\,\text{V} = \text{Stufe } 3 = 3$$

In Register DAPR ist die Hexzahl A3 zu schreiben.

```
DAPR=0xA3
```

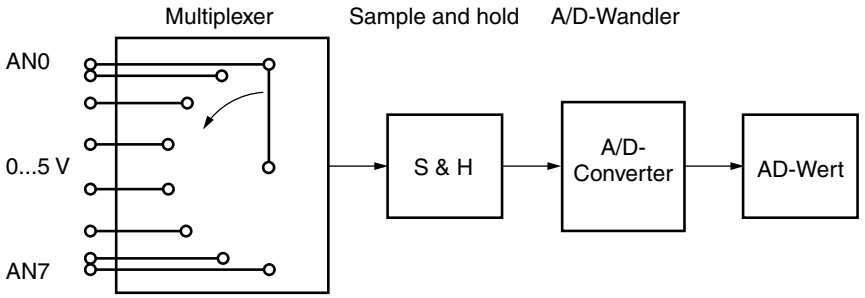
Damit ist der Wandelbereich festgelegt und gleichzeitig wird die Wandlung gestartet.

Jetzt wird der Bereich von 0,9375 bis 3,125 in 255 Stufen aufgeteilt.

$$\frac{3{,}125\,\text{V} - 0{,}9375\,\text{V}}{255} = 8{,}57\,\text{mV}$$

Analogeingänge

In den meisten Mikrocontrollern ist nur eine Schaltung für die Analog/Digital-Wandlung vorhanden, trotzdem kann an mehreren Eingängen (am AT89C51AC3 acht Eingänge) eine analoge Spannung eingelesen werden. Die Analogeingänge werden über einen Multiplexer auf die Wandlerschaltung gegeben.

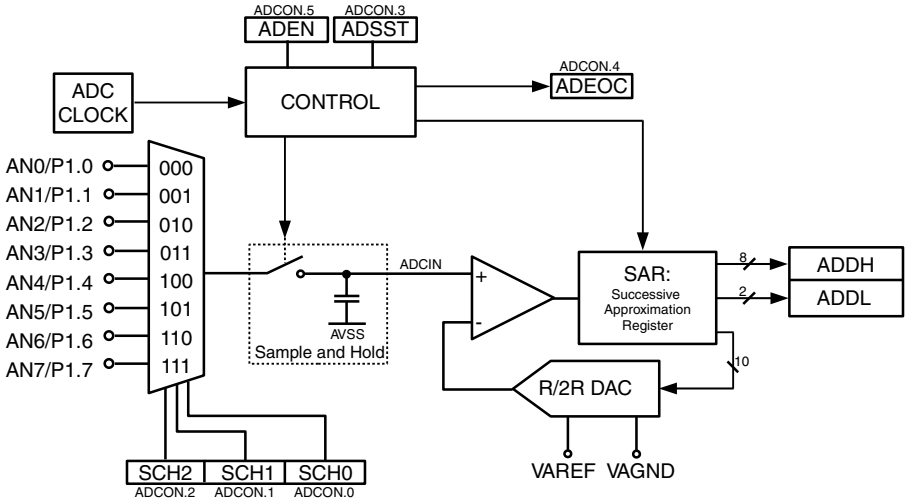


11.1 Analogwandlung mit dem AT89C51AC3 von Atmel

Es soll nun die konkrete Realisierung der A/D-Wandlung mit dem AT89C51AC3-Mikrocontroller gezeigt werden, als auch später die Umsetzung mit dem alten SAB 80C535 Mikrocontroller von Siemens, um einen Vergleich der beiden Architekturen zu haben.

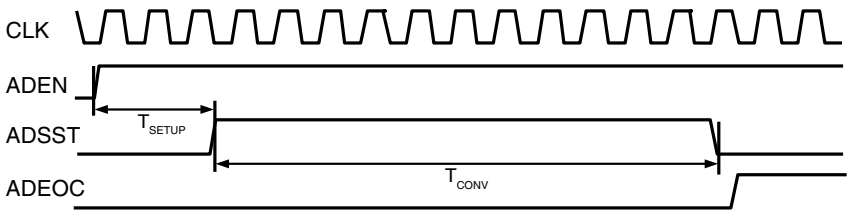
Als Analogspannung dient ein angeschlossenes Potenziometer, welches eine Spannung von 0 V bis 3 V an den Controller weitergibt.

Blockschaltbild des A/D-Wandlers



Links im Blockschaltbild sind die möglichen Eingänge für die A/D-Wandlung dargestellt. Diese können über SCH2, SCH1 und SCH0 gewählt werden. Nachfolgend ist der „Sample and Hold"-Block mit dem gesteuerten Schalter zu erkennen, an dem sich der Block der A/D-Wandlung anschließt (SAR: successive approximation register). Der benötigte Digital/Analog-Wandler (R/2R DAC) wird mit den Eingangsspannungen VAREF und VAGND gespeist. Die gewandelten Binärzahlen werden in den Registern ADDH und ADDL gespeichert. Über diesen Einheiten befindet sich die zeitliche Steuerung der Wandlung mit den notwendigen Ausgaben, die in dem folgenden Timing Diagramm dargestellt sind.

Timing Diagramm



ADEN: Analog/Digital-Wandler – **Enable/Standby Mode** (Hierüber wird der Wandler ein- und ausgeschaltet.)

ADSST: Analog/Digital-Wandler – **Start and Status** (Hierüber wird der Wandler gestartet. Während der Wandlung ist das Signal „High".)

ADEOC: Analog/Digital-Wandler – **End Of Conversion** (Hierüber wird angezeigt, wann die Wandlung fertig ist.)

Diese Bits, die Auswahl der Ports und die Umschaltung auf höchste Genauigkeit sind im Register ADCON gespeichert (nicht bitadressierbar):

ADCON nicht bitadressierbar Reset-Wert: X000 0000

—	PSIDLE	ADEN	ADEOC	ADSST	SCH2	Sch1	SCH0
7							0

Bit	Funktion
PSIDLE	**Pseudo Idle Mode** (höchste Genauigkeit) 1: höchste Genauigkeit bei der Wandlung; CPU wird in Standby gesetzt. 0: Wandlung ohne „Idle Mode“
ADEN	**Enable/Standy Mode** 1: Analog/Digital-Wandler eingeschaltet. 0: Analog/Digital-Wandler in Standby (Leistungsverbrauch 1 μW)
ADEOC	**End Of Conversion** Wird vom Mikrocontroller gesetzt, wenn der Wandler fertig ist. Muss per Software wieder gelöscht werden.
ADSST	**Start and Status** Muss gesetzt werden, um eine Analog/Digital-Wandlung zu starten. Wird nach der Wandlung automatisch wieder auf 0 gesetzt.
SCH2-0	**Selection of Channel to Convert** Auswahl des zu wandelnden Kanals (siehe Blockschaltbild)

Zusätzlich zu diesem Register wird das Register ADCF (Analog/Digital-Wandler Configuration) benötigt. Hier wird entschieden, ob der Eingang ein normaler I/O-Port ist, oder ein A/D-Wandlereingang.

ADCF:

ADCF nicht bitadressierbar Reset-Wert: 0000 0000

CH 7	CH 6	CH 5	CH 4	CH 3	CH 2	CH 1	CH0
7							0

Beispiel 11.2

Eine analoge Spannung am Eingang AN2 (P1.3) ist einzulesen, in eine 8-Bit-Zahl zu wandeln und an Port 2 binär auszugeben. Die Spannung variiert im Bereich von 0V bis 3V.

Vorüberlegungen:

Es soll eine analoge Spannung an Port P1.3 eingelesen werden. Aufgrund dessen wird Kanal 3 (CH3) im Register ADCF für eine Analog-Wandlung gesetzt, die übrigen Kanäle können unberücksichtigt bleiben. Dies geschieht über eine direkte Zuweisung oder sinnvoller über eine Maskierung (siehe Kapitel 7, C-Programmierung), damit die evtl. schon getätigten Zuweisungen zuvor im Programm nicht versehentlich verändern werden.

ADCF=0000 1000 (für Kanal 3)

in HEX:
ADCF=0x08;

mit Maskierung:
ADCF=ADCF|0x08;

Anschließend kann der Wandler am besten über eine komplette Zuweisung von ADCON gestartet werden. Die genauen Eintragungen hierzu sind rechts dargestellt.

Wandler starten:
Registereinträge in ADCON:
Bit 7: –
Bit 6: PSIDLE=0 (normale Wandlung)
Bit 5: ADEN=1 (Wandler einschalten)
Bit 4: ADEOC=0 (Flag löschen)
Bit 3: ADSST=1 (Wandler starten)
Bit 2-0 : 011 für Port 1.3

$\rightarrow$ ADCON=0010 1011 $\widehat{=}$ 0x2B(Hex)

Nun muss gewartet werden, bis die Wandlung fertig ist. Dies wird über Bit 4 von ADCON ausgegeben. Da das Byte nicht bitadressierbar ist, muss es mit einer Maskierung abgefragt werden.

Warten, bis Wandler fertig ist:
while((ADCON&0x10)!=0x10)
{
}

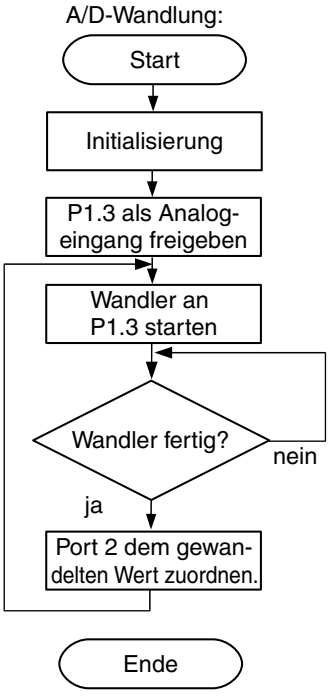
Im Fall der 8-Bit-Wandlung steht der gewandelte Wert dann im Register ADDH.

Zuweisung des 8-Bit-Wertes auf P2:
P2=ADDH

C-Programm:

```
#include <t89c51ac2.h>

void main()
{
     ADCF=ADCF|0x08;  //P1.3 als
                      //Analogeingang
                      //freigeben
     while(1)         //Endlosschleife
     {
          ADCON=0x2B; //Wandler an P1.3
                      // starten
          while((ADCON&0x10)!=0x10);
                      // Warten bis
                      // Wandler fertig
                      // ist.
          P2=ADDH;    // P2 dem
                      // gewandelten
                      // Wert zuordnen.
     }
}
```

PAP:

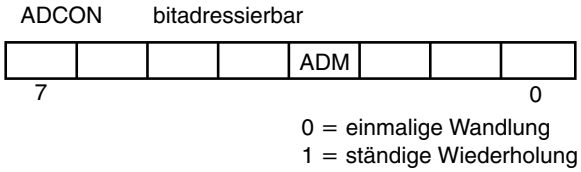


11.2 Analogwandlung mit dem SAB 80C535 von Siemens

Betriebsarten

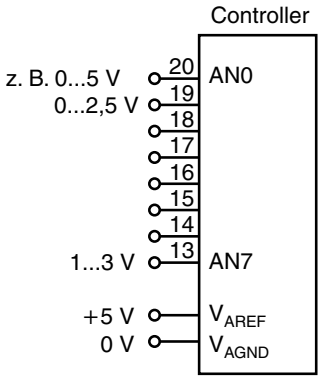
Der Wandler beim SAB 80C535 von Siemens lässt sich in zwei Arten betreiben:

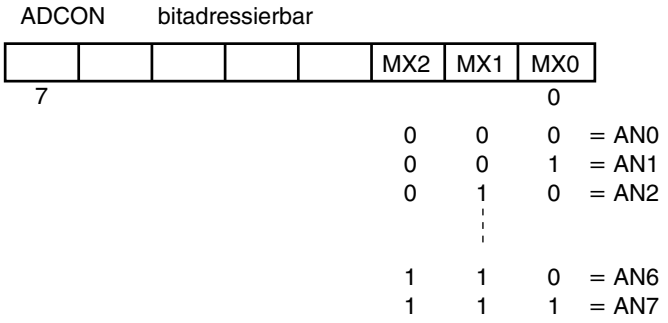
Die Betriebsart lässt sich mit Bit ADM (A/D-Converter-Modus) in Register ADCON einstellen.



Auswahl des Wandlereingangs

Einer der acht Analogeingänge wird über einen Multiplexer auf den A/D-Wandler geschaltet. Welcher Analogeingang gewandelt werden soll, lässt sich über die unteren drei Bits im Register ADCON (A/D-Converter) wählen.





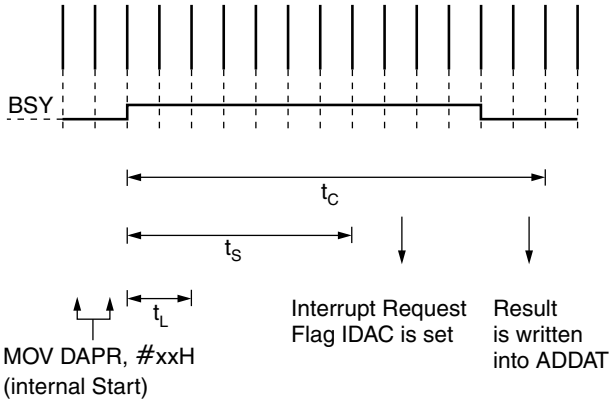
Wandelzeiten

Die Wandelzeiten beziehen sich auf die Zykluszeit des Controllers, die wiederum von der Quarzfrequenz abhängt. Ein Zyklus besteht aus 12 Takten.

Der Spannungswert, der am Ende der Load-Time erreicht ist, wird in den Digitalwert gewandelt. Nach der Conversion-Time steht der Digitalwert in Register ADDAT zur Verfügung.

t_L = Load-time
t_S = Sample-Time
t_C = Conversion-Time

Diagramm der Wandelzeiten:



Die Dauer der Wandlung wird durch das Busy-Bit BSY in Register ADCON angezeigt. Es wird durch die interne Hardware zu Beginn der Wandlung gesetzt und am Ende zurückgesetzt. Dieses Busy-Bit kann vom Programm abgefragt werden. Außer dem Busy-Bit wird nach beendeter Wandlung ein Interrupt-Bit IADC in Register IRCON gesetzt. Dieses Interrupt-Bit muss per Programm zurückgesetzt werden.

Datenblatt-Angaben:

A/D Converter Characteristics

$V_{CC} = 5\,\text{V} \pm 10\,\%$;
$V_{SS} = 0\,\text{V}$;
$V_{AREF} = V_{CC} \pm 5\,\%$;
$V_{AGND} = V_{SS} \pm 0{,}2\,\%$;
$V_{IntAREF} - V_{IntAGND} \geqq 1\,\text{V}$;

Beispiel 11.3

Ähnliches Beispiel wie Beispiel 11.2, nur mit dem Controller SAB 80C535:

Eine analoge Spannung am Eingang AN2 ist einzulesen, in eine 8-Bit-Zahl zu wandeln und an Port 2 binär auszugeben. Die Spannung variiert im Bereich von 0 V bis 5 V.

Vorüberlegungen:

Es soll eine analoge Spannung an AN2 eingelesen werden. Dies geschieht über die letzten 3 Bits des Registers ADCON. Am besten wird dies auch über eine Maskierung eingestellt.

Die letzten drei Bits von ADCON für den Wandlereingang AN2
Bit 2: 0
Bit 1: 1
Bit 0: 0

Maskierung:
ADCON=((ADCON&0xF8)|0x02)

Über das Bit ADM wird eine einmalige Wandlung gewählt.

ADM=0

Der Wandler wird über DAPR gestartet. Als Referenz-Spannungsbereich kann der Wert 0x00 eingeladen werden, da der maximale Spannungsbereich verwendet wird.

Wandler starten:
DAPR=0x00;

Nun muss gewartet werden, bis der Wandler fertig ist.

Warten, bis Wandler fertig ist:
while(BSY==0)
{
}

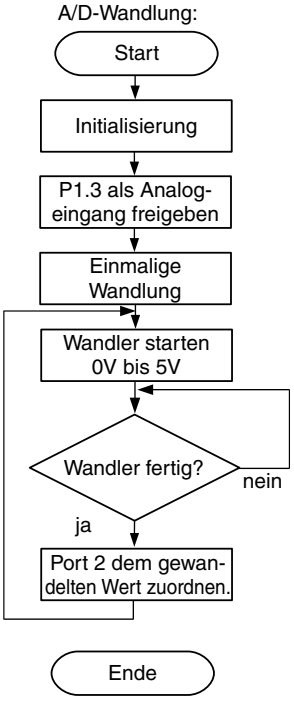
Der gewandelte 8-Bit-Wert steht dann im Register ADDAT.

Zuweisung des 8-Bit-Wertes auf P2:
P2=ADDAT

C-Programm:

```
#include <reg515.h>

void main()
{
     // P1.3 als A/D-Eingang freigeben:

     ADCON=((ADCON&0xF8)|0x02);
     ADM=0; // Einmalige Wandlung
     while(1) //Endlosschleife
     {
         DAPR=0x00;      //Wandler an AN2
                         // starten
         while(BSY==0);  //Warten bis
                         //Wandler fertig
                         // ist.
         P2=ADDAT;       // P2 dem
                         // gewandelten
                         // Wert zuordnen.
     }
}
```

PAP:



Übung 11.1

Ein spezieller Backofen hat eine LED, um anzuzeigen, ob eine nötige Temperatur von 200 °C erreicht ist. Mithilfe eines Temperatursensors mit einer Ausgangsspannung von 0 V bis 3 V soll diese Überwachung durchgeführt werden. Ist die Temperatur unter 200°C, so hat der Sensor eine Spannung unter 1,5 V. Bei einer Spannung größer oder gleich 1,5 V ist die Temperatur erreicht. Der Sensor befindet sich am Port P1.3 und die LED zur Anzeige an Port P2.0.

Realisieren Sie das C-Programm unter Verwendung des Controllers AT89C51AC3!

Übung 11.2

Mit einem Potentiometer an Port P1.3 kann eine Spannung zwischen 0 V und 3 V variabel eingestellt werden. Diese Spannung ist einzulesen und auf einer 7-Segmentanzeige mit Komma anzuzeigen. (8-Bit Genauigkeit reichen aus.)

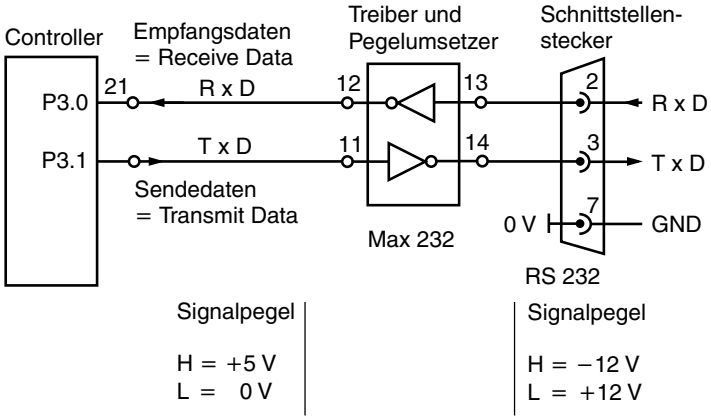
Hinweis:

- Der gewandelte Wert muss in die anzuzeigenden Werte umgerechnet werden.
- Sie können die Funktion für die 7-Segmentanzeige nutzen.

12 Die serielle Schnittstelle

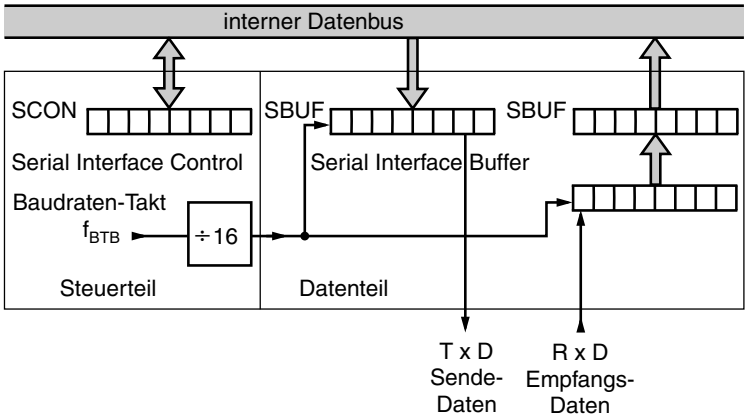
Der Controller beinhaltet eine serielle Schnittstelle, die voll duplex arbeitet. Sendeleitung und Empfangsleitung für die seriell übertragenen Datenbits sind die Portanschlüsse P3.0 und P3.1, die auf ihre Alternativfunktion umgeschaltet werden müssen. Dies geschieht per Software mithilfe der zugeordneten Spezial-Funktions-Register.

Serielle Datenübertragung:



12.1 Prinzipieller Aufbau

Die Schaltung der Schnittstelle lässt sich unterteilen in den Steuerteil und den Datenteil. In beiden Teilen befinden sich Spezial-Funktions-Register, die über den internen Datenbus des Controllers gelesen oder beschrieben werden.

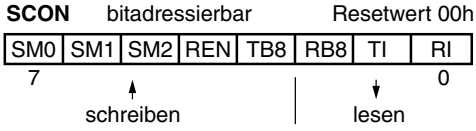


Steuerteil

Die Funktion des Steuerteiles wird mithilfe des Registers SCON (Serial Interface Control) eingestellt. Durch Setzen oder Löschen bestimmter Bits lässt sich die gewünschte Funktion erzielen. Andere Bits dienen als Statusanzeige und geben Auskunft über den momentanen Zustand der Schnittstelle. Sie lassen sich per Programm auslesen.

In diesem Teil wird die standardmäßige Konfiguration bei den 8051er-Controllern gezeigt. Der Atmel AT89C51AC3-Controller verfügt noch über mehr Besonderheiten, wie eine automatische Fehlererkennung, die hier nicht gezeigt werden soll und im Datenblatt nachgelesen werden kann.

Steuerregister SCON:



Bit		Funktion
SM0	SM1	Betriebsarten:
0	0	Modus 0: Schieberegister-Modus Baudrate: FXTAL/12
0	1	Modus 1: 8 Bit UART, flexible Baudrate
1	0	Modus 2: 9 Bit UART, feste Baudrate Baudrate: FXTAL/64 oder FXTAL/32
1	1	Modus 3: 9 Bit UART, flexible Baudrate
SM2		Freigabe bzw. Sperren des Empfängers bei Mehr-Rechner-Datenaustausch. (Nur Betriebsart 2 oder 3)
0		Empfänger freigeben
1		Empfänger empfängt nur Datenbytes, bei denen das 9. Bit (RB8) gesetzt war.
REN		Receiver Enable = Empfänger Freigabe
0		Empfänger gesperrt
1		Empfänger freigegeben
TB8		Transmitter Bit 8 = Sendebit 8
		Dies ist das 9. Datenbit, das in Modus 2 oder 3 ausgesendet wird.
		Es muss durch Software gesetzt bzw. gelöscht werden.

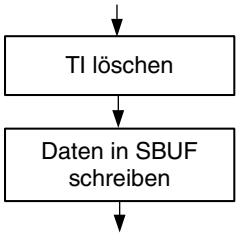
Bit	Funktion
RB8	Receiver Bit 8 = Empfangsbit 8 Dies ist das 9. Datenbit, das im Modus 2 oder 3 empfangen wird.
TI 1	Transmit Interrupt = Sende Interrupt Senderegister SBUF ist leer. Es kann mit einem neuen Datenbyte beschrieben werden. TI wird durch die interne Hardware gesetzt. Nach dem Einschreiben eines neuen Datenbytes in SBUF muss es per Software wieder auf 0 gesetzt werden.
RI 1	Receive Interrupt = Empfangs-Interrupt Empfangsregister SBUF enthält ein neu empfangenes Datenbyte. RI wird durch die interne Hardware gesetzt. Es muss nach dem Auslesen des neuen Datenbytes aus SBUF per Software wieder auf 0 gesetzt werden.

Datenteil

Im Datenteil befinden sich ein Senderegister SBUF und ein Empfangsregister, ebenfalls mit dem Namen SBUF. Beide Register werden unter der gleichen symbolischen Bezeichnung und Adresse als Spezial-Funktions-Register angesprochen. Welches der beiden Register gemeint ist, ergibt sich aus der Daten-Transportrichtung.

Daten senden

Mit dem Einschreiben in SBUF werden die Daten zum Senden freigegeben. Vor dem Schreiben der Daten nach SBUF muss TI im Steuerregister SCON als „Merker“ auf 0 gesetzt werden. Die Daten werden nach dem Einschreiben seriell über die Sendeleitung TxD (Transmit Data) Bit für Bit herausgeschoben. Die Frequenz wird durch die Baudrate bestimmt. Ist das Senderegister leer, wird TI im Steuerregister TCON automatisch wieder auf 1 gesetzt. An TI lässt sich also erkennen, ob das Byte gesendet wurde und ein neues in SBUF geschrieben werden kann.



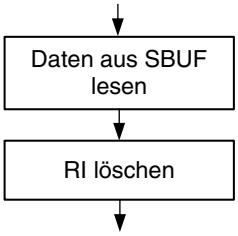
Nach dem Senden:
Funktionseinheit setzt TI=1 (automatisch).

Daten lesen

Die seriell ankommenden Empfangsdaten gelangen über den Eingang RxD (Receive Data) in das Empfangs-Schieberegister der seriellen Schnittstelle. Von dort werden sie in das Empfangsregister SBUF übernommen, wenn RI in TCON auf 0 gesetzt wurde. RI ist auf 0 zu setzen, wenn das empfangene Byte abgeholt wurde.

An RI lässt sich erkennen, ob das empfangene Byte abgeholt wurde. Nur wenn RI auf 0 gesetzt ist, wird ein neu angekommenes Byte vom Empfangs-Schieberegister nach SBUF übernommen, andernfalls geht es verloren.Wird das neue Byte von SBUF übernommen, wird RI automatisch auf 1 gesetzt.

Vor dem Lesen:
Abfragen, ob RI=1 ist.



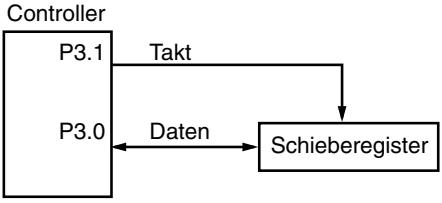
12.2 Betriebsarten

Die serielle Schnittstelle arbeitet in vier Betriebsarten; einem synchronen Modus (Modus 0) und drei asynchronen Modi (Modus 1,2,3). Die Baudrate wird in Modus 0 und 2 vom Systemtakt geliefert. In Modus 1 und 3 wird die Baudrate e von einem Timer erzeugt. In Modus 0 und 1 werden acht Bit, in Modus 2 und 3 neun Bit übertragen.

Modus 0

Schieberegister-Modus, Synchronbetrieb

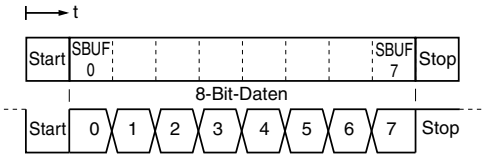
In diesem Modus werden acht Datenbits über P3.0 ausgegeben oder eingelesen. Der Schiebetakt erscheint an P3.1. Die Baudrate ist 1/12 $f_{\text{osz.}}$ Diese Betriebsart eignet sich zur seriellen Ein- und Ausgabe für ein angeschlossenes Schieberegister.



Modus 1

Asynchrone 8-Bit-Datenübertragung, variable Baudrate

In den asynchronen Betriebsarten wird kein Takt ausgegeben. Alle Teilnehmer müssen auf die gleiche Baudrate eingestellt werden. Das Datenbyte wird von einem Start- und einem Stopbit eingerahmt. Im Ruhezustand hat die Datenleitung H-Signal. Die Übertragung beginnt mit der fallenden Flanke des Startbits. Die Übertragung wird durch das Einschreiben eines Datenbytes in SBUF gestartet. Nach dem Startbit kommen die Datenbits, beginnend mit Bit D0. Anschließend geht die Datenleitung für mindestens eine Periode auf H-Signal. Das ist das Stopbit. Nach dem Stopbit könnte eine neue Übertragung, beginnend mit dem Startbit, erfolgen.



Modus 2

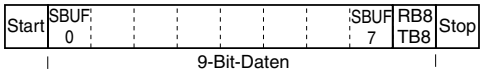
Asynchrone 9-Bit-Datenübertragung, feste Baudrate

Modus 3

Asynchrone 9-Bit-Datenübertragung, variable Baudrate

In diesen Modi werden 9 Datenbits übertragen. Das neunte Bit kommt beim Senden aus TB8. Es muss vor dem Senden in TB8 eingeschrieben werden. Beim Empfang geht das neunte Bit nach RB8. Dort kann es ausgelesen werden.

Das neunte Bit kann z. B. ein Paritybit oder ein zusätzliches Stopbit sein.



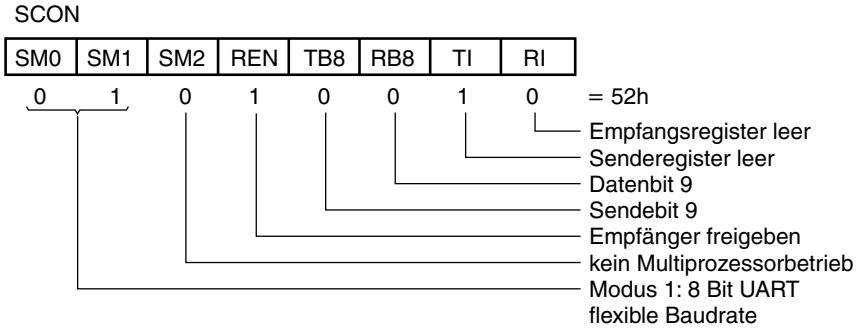
12.3 Programmierung

Vor Beginn der seriellen Datenübertragung ist die Schnittstelle so zu initialisieren, dass die Daten mit einer bestimmten Baudrate und einem bestimmten Übertragungsrahmen gesendet oder empfangen werden. Sender und Empfänger sind auf gleiche Werte einzustellen.

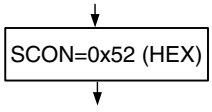
Im Folgenden wird die Programmierung der gebräuchlichen Übertragungsrahmen gezeigt. Auf die Erzeugung der Baudrate wird anschließend eingegangen.

Übertragungsrahmen 1 Startbit, 8 Datenbits, 1 Stopbit

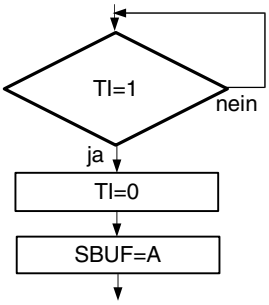
Initialisierung der Schnittstelle über Register SCON:



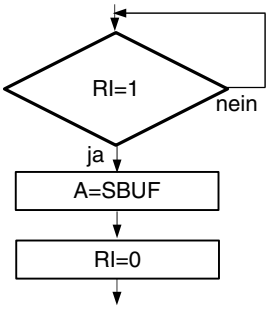
Initialisieren der Schnittstelle:



Datenausgabe:
Das Byte A soll über die serielle Schnittstelle ausgegeben werden.



Datenempfang:
Das seriell empfangene Byte soll in das Byte A geschrieben werden.

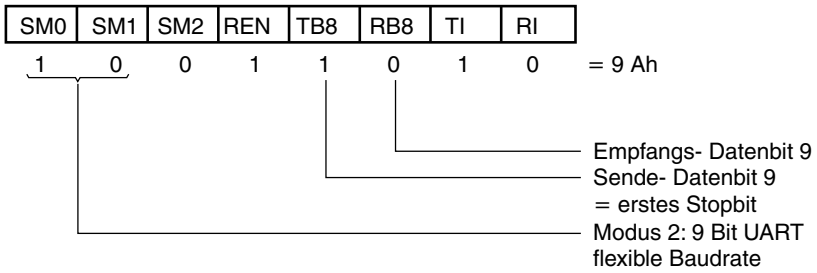


Übertragungsrahmen 1 Startbit, 8 Datenbits, 2 Stopbits

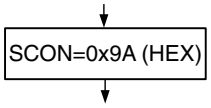
Initialisieren der Schnittstelle:

Die Schnittstelle gibt generell 1 Startbit, dann die Datenbits und anschließend 1 Stopbit aus. Werden zwei Stopbits gewünscht, muss die Schnittstelle auf neun Datenbits initialisiert werden. Die ersten acht Datenbits sind das zu übertragende Byte, das neunte Datenbit ist das erste Stopbit. Es ist als Stopbit immer auf 1 zu setzen.

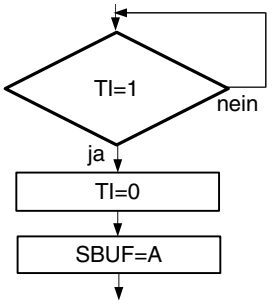
Initialisierung des Registers SCON



Initialisieren der Schnittstelle:



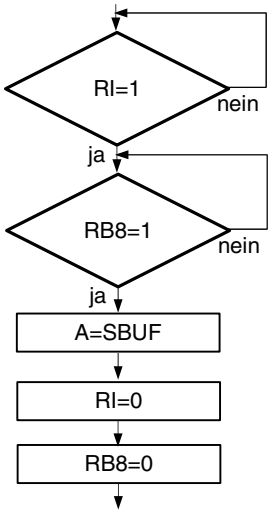
Datenausgabe (wie vorher):
Das neunte ausgegebene Datenbit ist TB8. Es ist auf 1 gesetzt. Vom angeschlossenen Gerät wird es als erstes Stopbit interpretiert.



Datenempfang:
Das seriell empfangene Byte soll in das Byte A geschrieben werden.

Es wird kontrolliert, ob ein Byte empfangen wurde und ob das neunte Bit als Stopbit 1-Signal hatte.

Wird das zusätzliche Stopbit nur gesendet, um einen größeren zeitlichen Abstand zwischen zwei Übertragungen zu erhalten, muss es nicht überprüft werden, und es lässt sich ebenfalls der vorher beschriebene PAP verwenden.

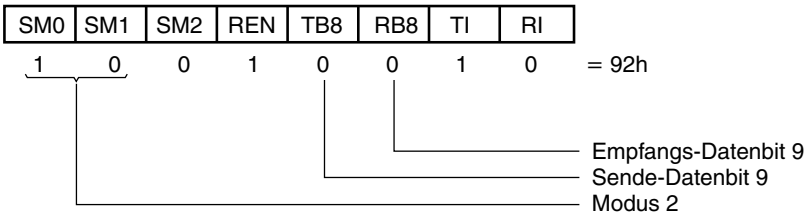


Übertragungsrahmen 1 Startbit, 8 Datenbits, 1 Paritybit, 1 Stopbit

TB8 und RB8 können auch für ein sogenanntes Parity-Bit genutzt werden. Dabei wird über ein Bit die Anzahl der zu sendenden Bits auf eine gerade Anzahl ergänzt. Dieses Bit kann dann in

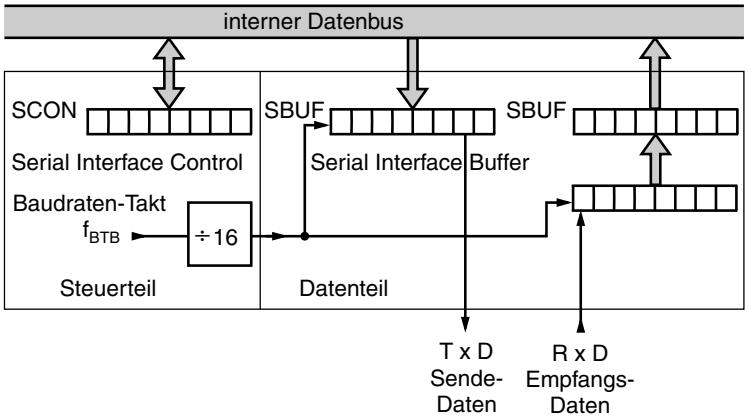
TB8 geschrieben werden. Der Empfänger prüft dann, ob mit der Ergänzung dieses Bits (beim Empfang RB8) eine gerade Anzahl von Bits übertragen wurde. Ist dies nicht der Fall, ist ein Übertragungsfehler passiert.

Initialisierung des Registers SCON



Erzeugung der Baudrate

Der Baudratentakt wird in der seriellen Schnittstelle für das Auslösen der verschiedenen internen Schaltzustände benötigt. Zum Senden oder Empfangen eines Bits sind 16 Takte erforderlich. Deshalb beträgt die Baudrate BD, mit der die Bits übertragen werden, 1/16 des Baudratentaktes f_{BTB}.



Der Baudratentakt wird in der Regel vom internen Systemtakt der CPU abgeleitet, der wiederum von der Oszillatorfrequenz des Quarzes bestimmt wird.

Mithilfe eines Schalters SMOD im Register PCON lässt sich der Systemtakt halbieren.

PCON nicht bitadressierbar

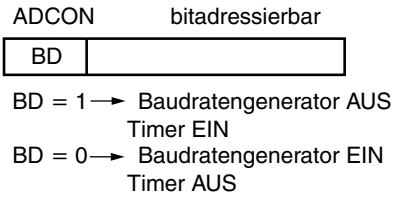
SMOD	

Beim ATMEL AT89C51AC3:
SMOD1 =1 $\rightarrow \quad f = 1/2 \cdot f_{\mathrm{sys}}$
SMOD1 = 0 $\rightarrow \quad f = f_{\mathrm{sys}}$

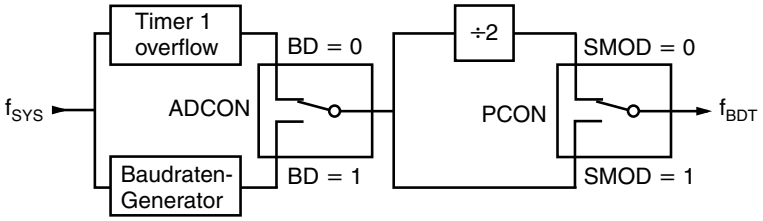
Beim Siemens SAB 80C535:
SMOD=1 $\rightarrow \quad f = f_{\mathrm{sys}}$
SMOD=0 $\rightarrow \quad f = 1/2 \cdot f_{\mathrm{sys}}$

Bei einigen 8051er-Controllern lässt sich die Baudrate über einen Baudratengenerator oder einen Timer einstellen, wie zum Beispiel beim Siemens SAB 80C535. Wenn kein Baudratengenerator vorhanden ist, wie zum Beispiel beim AT89C51AC3, lässt sich die Baudrate nur über einen Timer generieren.

Beim SAB 80C535 lässt sich über das Bit BD im Register ADCON zwischen diesen beiden Möglichkeiten umschalten.



Baudratentakterzeugung beim Siemens SAB 80C535:

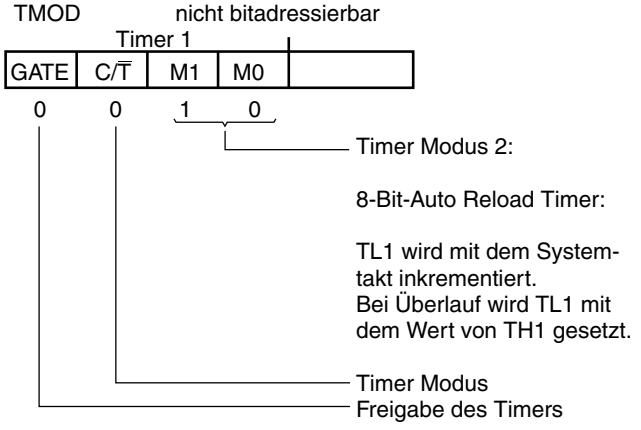


Der Baudratengenerator ist so ausgelegt, dass er bei der üblichen Quarzfrequenz von 12 MHz die gängigen Baudraten von 4800 oder 9600 Bits pro Sekunde liefert.

SMOD = 0 → BD = 4800
SMOD = 1 → BD = 9600

Erzeugung der Baudrate mit dem Timer 1 (AT89C51AC3/SAB80C535 etc.)

Die Überlaufrate von Timer 1 wird zur Erzeugung des Baudratentaktes verwendet. Die Überlaufrate ist die Anzahl der Timerüberläufe pro Sekunde. Der Timer 1 wird am sinnvollsten im 8-Bit-Autoreload-Modus betrieben.



Berechnung der Baudrate (AT89C51AC3) SMOD1=0:

Beim AT89C51AC3 wird der Baudratentakt nicht durch 16, sondern durch 32 geteilt.

$$BD = \frac{1}{12} \cdot f_{\text{osz}} \cdot \frac{1}{256 - THI} \cdot \frac{1}{32}$$

Berechnung der Baudrate (SAB80C535) SMOD=1:

$$BD = \frac{1}{12} \cdot f_{\text{osz}} \cdot \frac{1}{256 - THI} \cdot \frac{1}{16}$$

Beispiel 12.1

Es soll beispielhaft auf einem Computerterminalprogramm ununterbrochen der Text „Hallo Welt!“ dargestellt werden. Die Baudrate für die Übertragung soll 2400 Baud betragen und im Mode 1 geschehen.

Die Oszillator-Frequenz des Controllers beträgt 12 MHz.

Vorüberlegungen:

Das Bit SMOD von PCON ist z. B. auf 0 zu setzen, um eine ungeteilte Frequenz zu erhalten.

Dies geschieht am besten durch eine Maskierung:
PCON =PCON&0x7F;

	XXXX XXXX	PCON
&	01111111	0x7F
	0XXX XXXX	PCON

Dann kann TH1 für den Baudratentakt berechnet werden. Für 2400 Baud ergibt sich für TH1=243 $\rightarrow$ 0xF3 (nach oben angegebener Formel).

Der Timer 1 wird als 8-Bit-Autoreload-Timer konfiguriert und gestartet.

```
TMOD=0x20;
TR1=1;
```

Zum Übertragen des Textes kann die Funktion schreiben vom LC-Display abgeändert werden.

```
void schreiben(char text[])           // Funktionskopf schreiben
{

      int k=0;                        // Variablendeklaration k
      while(text[k]!='\0')            // Solange das zu übertragende Datenfeld
      {                               // nicht beendet ist, wird nacheinander
               SBUF=text[k];          // jedes Zeichen in SBUF geschrieben.
            while(TI==0);             // Es wird gewartet bis TI eins ist.
            TI=0;                     // TI wird wieder 0 gesetzt.
            warten();                 // Warten, bis das Zeichen übertragen ist.
            k++;
      }
}
```

Diese Funktion kann dann wie folgt aufgerufen werden:

```
schreiben ("Hallo Welt!");
```

Das C-Programm für die Übertragung „Hallo Welt!“:

```
#include <t89c51ac2.h>  // Einbinden der Bibliothek

void ausgabe_text(char text[]);  // Funktionsprototypen
void warten(void);

void main(void)
{
      PCON=PCON|0x80;
      SCON=0x52;                // Mode 1 (Startbit, 8 Datenbits, 1 Stoppbit)
      TI=0;                     // Sendeflag löschen

      TMOD=0x20;                // Timer1 im 8 Bit-Reload-Betrieb (Mode 2)
      TL1=0xF3;                 // Start-Wert ist F3
      TH1=0xF3;                 // Reload-Wert ist F3 für 2400 Baud
      TR1=1;                    // Timer1 einschalten
      while(1)
      {
              ausgabe_text("  Hallo Welt!  ");
      }
}
void warten(void)
{
      unsigned int x;
      for(x=0;x<=1000;x++);
}

void ausgabe_text(char text[])  // Funktionskopf schreiben
{
      int k=0;
      while(text[k]!='\0')      // Solange das zu übertragende Datenfeld
      {                         // nicht beendet ist, wird nacheinander
               SBUF=text[k];    // jedes Zeichen in SBUF geschrieben.
            while(TI==0);       // Es wird gewartet bis TI eins ist.
            TI=0;               // TI wird wieder 0 gesetzt.
            warten();           // Warten, bis das Zeichen übertragen
            k++;                // ist.
      }
}
```

Eine weitere Möglichkeit der Übertragung von Zeichen besteht darin, die Standardbibliothek <stdio.h> in C einzubinden. Dann können mithilfe des printf-Befehls die Daten übertragen werden. (z. B. printf („Hallo Welt!“). Der Nachteil an diese Methode ist allerdings, dass sehr viel Speicherplatz für das Programm benötigt wird, da diese Befehle über eine Vielzahl von Parametern verfügen.

Übung 12.1

Realisieren Sie das obige Programm mithilfe des printf-Befehls!

12.4 Terminal Emulation VT52

Mithilfe von passenden Terminalprogrammen auf dem PC, zum Beispiel ein VT52-Terminal, lassen sich Zeichen auf dem Computermonitor anordnen.

Die entsprechenden Befehle werden mit einer sogenannten Escape-Sequenz eingeleitet, der dann die Befehle für die Anordnung der Zeichen folgt. Die Sequenzen sind durch festgelegte Hexadezimalzahlen wie in der unten stehenden Tabelle definiert.

Die „Home-Positon“ des Bildschirms befindet sich oben links. Es gibt normalerweise 24 Zeilen und 80 Spalten.

Tabelle mit den üblichen Escape-Sequenzen:

Escape-Sequenz	Hex-Code	Beschreibung
ESC H	1B 48	Cursor auf die „Home-Position“
ESC J	1B 4A	Nach der Cursor-Position wird der Rest des Bildschirms gelöscht.
ESC K	1B 4B	Nach der Cursor-Position wird der Rest der Zeile gelöscht.
ESC A	1B 41	Eine Zeile nach oben springen.
ESC B	1B 42	Eine Zeile nach unten springen.
ESC C	1B 43	Eine Spalte weiter nach rechts.
ESC D	1B 44	Eine Spalte weiter nach links.
ESC Y	1B 59	Mit diesem Befehl kann der Cursor frei positioniert werden. Nach dem Befehl folgt die Zeile und die Spalte. Diese müssen mit 20h addiert werden.

Beispiel 12.2

„Hallo Welt!“ soll in der zweiten Zeile positioniert werden.

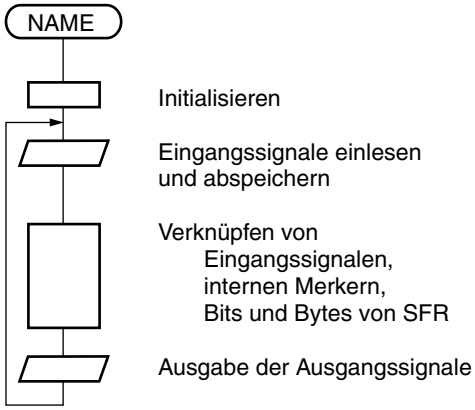
Der Quellcode ist nur auszugsweise dargestellt.

```
...
void main()
{
      .
      .
      .
      printf(“\x1B\x48“);              // Cursor auf Home-Position
      printf(“\x1B\x4a“);              // Bildschirm löschen
      while(1)
      {
            printf(“x1b\x59\x22\x20“);  // Cursor auf die 2.Zeile stellen
            printf(“Hallo Welt“);
            .
            .
            .
      }
}
```

13 Das Interrupt-System

Die Funktion des Controllers wird vorgegeben durch das Programm, welches er bearbeitet. Bei Steuerungen zur Automatisierung von Maschinen oder Prozessen erfolgt die Programmbearbeitung zyklisch.

Zu Programmbeginn werden die Eingangssignale eingelesen und abgespeichert. Während der Programmbearbeitung erfolgt die Verknüpfung der Eingangssignale sowie die Verarbeitung interner Merker und Bits oder Bytes von Spezial-Funktions-Registern. Als Ergebnis wird das Ausgangssignal-Abbild zusammengestellt, interne Merker gesetzt und Bits oder Bytes in Spezial-Funktions-Registern manipuliert. Am Ende des Programmzyklus erfolgt die Ausgabe der Ausgangssignale.



Die Reaktionszeit auf Eingangssignale beträgt also maximal eine Zykluszeit und liegt damit im Millisekundenbereich. Bedingung dafür ist allerdings, dass das Programm keine Warteschleifen und Rückwärtssprünge enthält.

Bei dieser Art der Programmierung werden Eingangssignale oder interne Zustände erst registriert, wenn sie durch das Programm abgefragt werden. Diese Methode wird Polling genannt.

Soll die Reaktionszeit auf ein Eingangssignal oder ein internes Ergebnis verringert werden, muss die Möglichkeit bestehen, den Controller in seinem laufenden Programm zu unterbrechen, um auf das eingetretene wichtige Ereignis sofort zu reagieren. Solch eine Unterbrechungsanforderung nennt man Interrupt. Auf den Interrupt kann dann der Controller mit der Interrupt-Service-Routine reagieren.

Diese Interrupt-Service-Routine ist aufgebaut wie ein Unterprogramm bzw. eine Funktion. Der Aufruf dieses Unterprogramms erfolgt von der internen Interrupt-Steuerung durch einen internen Befehl, der nicht im Programm steht. Am Ende der Interrupt-Service-Routine wird, wie bei einem Unterprogramm, zu dem ursprünglichen Programm zurückgekehrt.

Der Interrupt kann entweder von externen Ereignissen über bestimmte Porteingänge oder von internen Ereignissen über bestimmte Bits in Spezial-Funktions-Registern ausgelöst werden. Diese Bits heißen Interrupt-Request-Flags. Einige Flags setzen sich bei Einsprung in die Service-Routine selbst zurück, andere müssen per Programm gelöscht werden.

Die interne Interrupt-Steuerung (Interrupt-Controller) fragt jedes Request-Flag ab, ob es gesetzt ist. Zu jedem Flag gehört noch ein Freigabe-Bit (Interrupt Enable). Der Interrupt wird nur angenommen, wenn er freigegeben ist. Ist das Flag gesetzt und der Interrupt freigegeben, erfolgt nach Beendigung eines Befehls im laufenden Programm der Sprung in die Interrupt-Service-Routine.

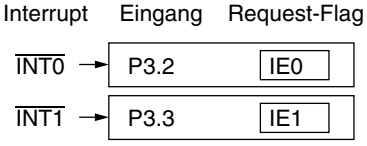
Bei mehreren gesetzten Interrupt-Request-Flags gilt eine programmierte Prioritätenfolge. Jetzt kennen sie den Unterschied zwischen Polling- und Interruptbetrieb. Außerdem haben Sie eine allgemeine Vorstellung von der Interrupt-Steuerung und der Interrupt-Service-Routine.

Im Folgenden wird die Interrupt-Steuerung detailliert beschrieben.

13.1 Interrupt-Quellen und Anforderungs-Flags

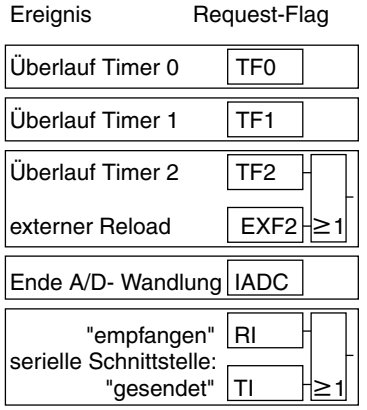
Externe Interrupt-Quellen

Über die Porteingänge P3.2 und P3.3 kann ein externer Interrupt ausgelöst werden. Wenn diese Interrupts konfiguriert sind, und ein Ereignis an den Pins passiert, wird das aktuelle Programm unterbrochen.



Interne Interrupt-Quellen

Interne Interrupts sind Ereignisse, die innerhalb des Controllers auftreten. Zum Beispiel, wenn ein Timer überläuft, oder der A/D-Wandler fertig ist. Dies ist sehr hilfreich, um Warteschleifen im Programm zu vermeiden und damit den Programmzyklus zu optimieren.



Die Interrupt-Quellen setzen die angegebenen Request-Flags. Diese Flags sind Bits in Spezial-Funktions-Registern.

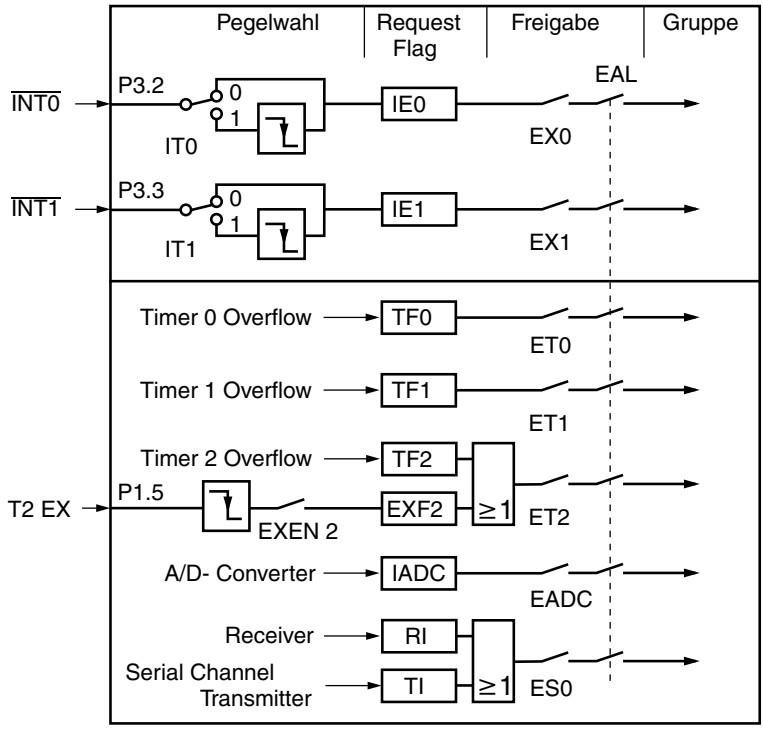
13.2 Pegelwahl und Interrupt-Freigabe

Bei einigen externen Interrupts kann gewählt werden, ob die Signalflanke oder der Signalpegel einen Interrupt auslösen soll. Diese Wahl erfolgt wieder über Bits in Spezial-Funktions-Registern.

Der Interrupt-Controller fragt zyklisch ab, welche Request-Flags gesetzt sind. Dabei werden jedoch nur die freigegebenen Flags berücksichtigt. Jedes Flag hat eine eigene Freigabe (Enable). Außerdem gibt es eine generelle Freigabe für alle Flags. Die Freigabe erfolgt ebenfalls durch Setzen bestimmter Bits in Spezial-Funktions-Registern.

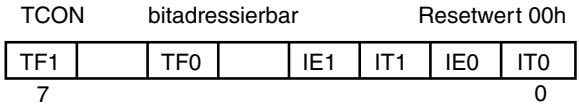
Der Interrupt wird angenommen, wenn das Flag freigegeben und es von der Priorität her an der Reihe ist. Ehe jedoch der Interrupt-Controller die Interrupt-Service-Routine erzeugt, muss der gerade laufende Befehl beendet werden.

Pegelwahl und Interrupt-Freigabe:



Spezial-Funktions-Register für Pegelwahl, Request-Flag und Freigabe

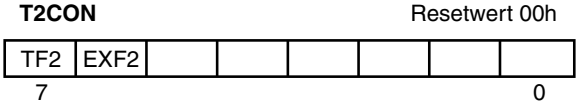
TCON



Bedeutung der einzelnen Bits:

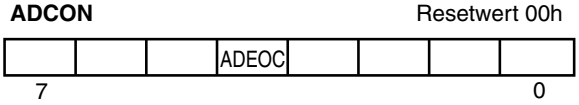
Bit	Funktion
TF1	Interrupt-Request-Flag des Überlaufes von Timer 1. Wird beim Einsprung in die Interrupt-Routine automatisch gelöscht.
TF0	Interrupt-Request-Flag des Überlaufes von Timer 0. Wird beim Einsprung in die Interrupt-Routine automatisch gelöscht.
IE1	Interrupt-Request-Flag des externen Interrupts 1 an P3.3. IT1 = 1: IE1 wird gesetzt bei einer fallenden Flanke an P3.3. IE1 wird bei Einsprung in die Interrupt-Routine automatisch gelöscht. IT1 = 0: IE1 bleibt gesetzt, so lange L-Pegel anliegt. Die Interrupt-Routine muss veranlassen, dass der L-Pegel am Eingang P3.3 verschwindet.
IT1 0 1	Selektionsbit für Interrupt 1 (Interrupt 1 Type Select Bit). Interrupt wird durch L-Pegel ausgelöst. Interrupt wird durch eine fallende Flanke ausgelöst.
IE0	Interrupt-Request-Flag des externen Interrupts 0 an P3.2. IT1 = 1: IE0 wird gesetzt bei einer fallenden Flanke an P3.2. IE0 wird bei Einsprung in die Interrupt-Routine automatisch gelöscht. IT1 = 0: IE0 bleibt gesetzt, so lange L-Pegel anliegt. Die Interrupt-Routine muss veranlassen, dass der L-Pegel am Eingang P3.2 verschwindet.
IT0 0 1	Selektionsbit für Interrupt 0. Interrupt wird durch L-Pegel ausgelöst. Interrupt wird durch fallende Flanke ausgelöst.

T2CON



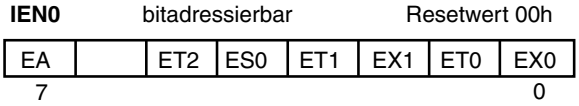
Bit	Funktion
TF2	Interrupt Request Flag des Überlaufes von Timer 2. Bei einem Überlauf wird es automatisch gesetzt. Es muss per Software rückgesetzt werden.
EXF2	Interrupt Request Flag des externen Timer-2-Nachlademodus. Es wird bei einer fallenden Flanke an T2EX gesetzt, wenn der externe Reload-Modus für den Timer 2 gewählt wurde. Ist der Nachlade-Interrupt des Timers freigegeben (EXEN2=1), wird in die Interrupt-Routine gesprungen. EXF2 muss per Software gelöscht werden.

ADCON



Bit	Funktion
ADEOC	Interrupt Request Flag des A/D-Wandlers. Es wird am Ende einer A/D-Wandlung automatisch gesetzt. Es muss per Software glöscht werden.

IEN0 (Interrupt Enable Register 0)



Bedeutung der einzelnen Bits des Interrupt-Freigaberegisters 0 (Interrupt Enable):
Generell gilt: Ein 0-Signal sperrt den Interrupt, ein 1-Signal gibt ihn frei.

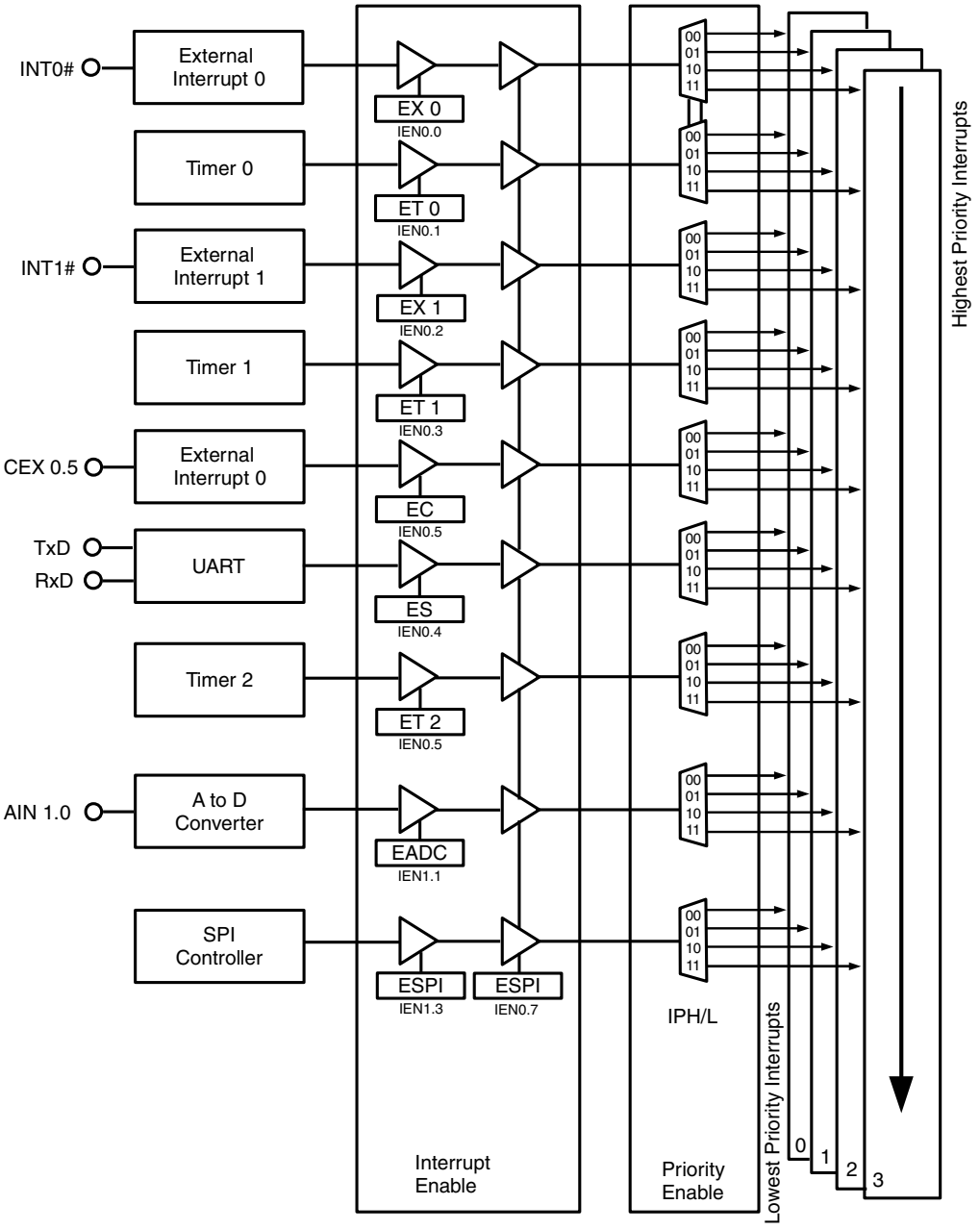
Bit	Funktion
EA 0 1	Generelles Freigabebit für alle Interrupts Es wird kein Interrupt bedient. Es gilt das individuelle Freigabebit des jeweiligen Interrupts.
ET2	Freigabebit des Timer-2-Interrupts
ES0	Freigabebit des Interrupts der seriellen Schnittstelle
ET1	Freigabebit des Timer-1-Interrupts
EX1	Freigabebit des externen Interrupts 1
ET0	Freigabebit des Timer-0-Interrupts
EX0	Freigabebit des externen Interrupts 0

IEN1 (Interrupt Enable Register 1)

Bit	Funktion
ESPI	Freigabe des SPI-Interrupts
EADC	Freigabe des A/D-Wandler-Interrupts

13.3 Interrupt-Prioritäten

Den Interrupts lassen sich vier Prioritätsstufen zuordnen. Bei gleicher Priorität entscheidet die vorgegebene Reihenfolge (siehe Abbildung) der Abfragen, welcher Interrupt zuerst angenommen wird.


INT0#
External Interrupt 0
EX 0
IEN0.0
Timer 0
ET 0
IEN0.1
INT1#
External Interrupt 1
EX 1
IEN0.2
Timer 1
ET 1
IEN0.3
CEX 0.5
External Interrupt 0
EC
IEN0.5
TxD
RxD
UART
ES
IEN0.4
Timer 2
ET 2
IEN0.5
AIN 1.0
A to D Converter
EADC
IEN1.1
SPI Controller
ESPI
IEN1.3
ESPI
IEN0.7
00
01
10
11
IPH/L
Highest Priority Interrupts
Lowest Priority Interrupts
0
1
2
3
Interrupt Enable
Priority Enable

Die Prioritätsstufen werden über die Register IPH0 und IPL0 bzw. IPH1 und IPL1 wie in der unten dargestellten Tabelle zugeordnet.

IPH.x	IPL.x	Interrupt Level Priority
		0 (Lowest)
	1	1
1	0	2
1	1	3 (Highest)

IPH0

IPH0 nicht bitadressierbar

-	PPCH	PT2H	PSH	PT1H	PX1H	PT0H	PX0H
7							0

IPL0

IPL0 bitadressierbar

-	PPC	PT2	PS	PT1	PX1	PT0	PX0
7							0

Bit	Funktion
PPC	PCA Interrupt Priority bit
PT2	Timer 2 Überlauf Interrupt Priority bit
PS	Serial Port Priority bit
PT1	Timer 1 Überlauf Interrupt Priority bit
PX1	External Interrupt 1 Priority bit
PT0	Timer 0 Überlauf Interrupt Priority bit
PX0	External Interrupt 0 Priority bit

IPH1

IPH1 nicht bitadressierbar

-	-	-	-	SPIH	-	PADCH	
7							0

IPL1

IPL1 bitadressierbar

-	-	-	-	SPIL	-	PADCL	
7							0

Bit	Funktion
SPI	SPI Interrupt Priority Bit
PADC	ADC Interrupt Priority Bit

13.4 Interrupt-Vektoren/Interruptnummer

Wird ein Interrupt-Request-Flag angenommen, erzeugt die Interrupt-Steuerung intern einen Sprung an die entsprechende Speicherstelle. Dabei ist jedem Flag eine bestimmte Sprungadresse fest zugeordnet. Diese festen Adressen nennt man Interrupt-Vektoren.

In C wird ein Interrupt durch eine sogenannte Interrupt-Service-Routine (ISR) bearbeitet. Dabei handelt es sich um eine Funktion, die durch einen Interrupt ausgelöst wird. Damit der Compiler weiß, dass es sich um eine ISR handelt, folgt dem Funktionskopf das Schlüsselwort „interrupt“ mit der Interruptnummer.

Zum Beispiel:

```
void Ereignis(void) interrupt 0
{
    Block1;
}
```

Die Interruptnummer lässt sich aus der Vektoradresse berechnen.

Interruptnummer=(Vektoradresse-3)/8

Zuordnungen

Interruptquelle	Vektoradresse	Interruptnummer
Externer Interrupt (INT0)	0003h	0
Timer0 (TF0)	000Bh	1
Externer Interrupt (INT1)	0013h	2
Timer1 (TF1)	001Bh	3
PCA (CF or CCFn)	0033h	6
Serielle Schnittstelle (UART: RI oder TI)	0023h	4
Timer2 (TF2)	002Bh	5
Analog/Digital-Wandler (ADCI)	0043h	8
SPI Interrupt	0053h	10

13.5 Anwendungen

Beispiel 13.1

Motorsteuerung mit Not-Aus an Interrupt-Eingang 0

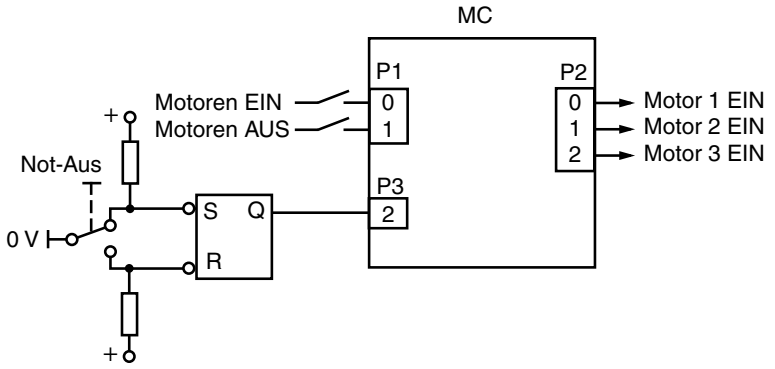
An Port P3.2 ist ein Not-Aus-Taster angeschlossen, der bei Betätigung sofort drei Motoren abschalten soll. Die Motoren sind an Port P2.0, P2.1 und P2.2 angeschlossen.

Der Interrupt 0 wird an Port P3.2 ausgelöst. Er setzt das Request-Flag IE0. Die Vektoradresse für Interrupt 0 ist 0003h, welches in C der Interruptnummer 0 entspricht. An dieser Adresse steht

der Sprung in die Interrupt-Service-Routine NOT_AUS. Der Interrupt soll durch die negative Flanke an P3.2 ausgelöst werden. Das ankommende Not-Aus-Signal ist vor dem Portpin über ein Flipflop entprellt.

Um den Interrupt zu testen, soll ein einfaches Hauptprogramm zur Motorsteuerung laufen. Die Motoren sind über je eine Taste ein- bzw. auszuschalten. Dieses Hauptprogramm wird durch den Interrupt unterbrochen.

Anschlussplan:



C-Programm:

```
#include <t89c51ac2.h>          // Einbinden der Controller-Bibliothek

void main ()                    // Hauptprogramm
{
      IT0 = 1;                  // Interruptauslösung durch fallende Flanke an P3.2
      EX0 = 1;                  // Freigabe des Interrupts INT0
      EA = 1;                   // Freigabe aller Interrupts
      IPH0 = IPH0 | 0x01;       // Höchste Priorität
      IPL0 = IPL0 | 0x01;       // einstellen
      P2 = 0x00;                // Alle Motoren aus
      INT0 = 1;                 // INT0 als Eingang
      while(1)                  // Endlosschleife
      {

         if (P11==0)            // Wenn die Taste Motoren AUS gedrückt wird,
         {
            P2 = 0x00;          // soll der Ausgang der Motoren auf AUS gesetzt werden,
         }
         else                   // ansonsten
         {
            if (P10==0)         // wird der Ausgang der Motoren gesetzt,
             {
              P2 = 0x07;        // falls der Taster Ein gedrückt ist.
            }
         }
      }                         // Ende der Endlosschleife
}                               // Ende von main()
```

```
void ISR_INT0 (void) interrupt 0 // Interrupt-Service-Routine für Interrupt 0
{
P2 = 0;                          // Alle Motoren ausschalten
IE0 = 0;                         // Interrupt-Flag zurücksetzen
}
```

Beispiel 13.2

Blinklicht mit Zeitinterrupt

In diesem Beispiel wird ein Blinklicht mit 0,5 Hz an Port P2.0 erzeugt, ohne dass im Hauptprogramm etwas bearbeitet wird.

Hierzu läuft 20 Mal der Timer 0 über, wenn dieser mit dem Wert TL0=0xB0 und TH0=0x3C vorgeladen ist. (Siehe hierzu auch das Beispiel 10.2.)

```
#include <t89c51ac2.h>            // Einbinden der Bibliothek
sbit P20=P2^0;                    // Anzeige- LED, blinkt im 0,5s Takt
unsigned char an=0,uz=0;          // Hilfsvariablen

void main(void)
{
      // Der Timer 0 wird einmal eingestellt und gestartet:

      TR0=0;                      // Timer stopp
      TMOD=((TMOD&0xF0)|0x01);    // Timer0 einstellen ohne Timer1 zu ändern
      EA=1;                       // Interrupts freischalten
      ET0=1;                      // Timer 0 Interrupt freigeben
      TL0=0xb0;                   // Lowbyte von Timer0 in TL0
      TH0=0x3c;                   // Highbyte von Timer0 in TH0
      TF0=0;                      // Überlaufbit zurücksetzen
      TR0=1;                      // Timer starten
      while(1);                   // Endlosschleife ohne Programmcode
}

// Blinken von LED P2.0 durch Interrupt

void timer_0(void) interrupt 1
{
      uz++;                      // Zähler für Timeraufrufe erhöhen
      TR0=0;                     // Zähler stopp
      TL0=0xB0;                  // Lowbyte von Timer0 in TL0
      TH0=0x3C;                  // Highbyte von Timer0 in TH0
      TR0=1;                     // Timer starten

// die folgende if Abfrage wird nur einmal ausgewertet, wenn der Timer
// 20 mal aufgerufen wurde, sonst wird sie übergangen.

      if(uz==20)                 // 20 Timerdurchläufe für 1s
      {
            P20=~P20;
            uz=0;                // Zähler zurücksetzen
      }
}
```

14 Programmierung in Assembler

In diesem Kapitel soll auf die Programmierung des Mikrocontrollers in Assembler eingegangen werden, da auch heutzutage noch häufig vor allem bei zeitkritischen Problemen in Assembler programmiert wird. Der Vorteil von Assembler liegt klar darin, dass der Programmierer jeden einzelnen Schritt des Controllers programmiert und somit die Hardware besser nutzen und ausschöpfen kann. Anhand des Wissens der Ausführungszeiten der einzelnen Befehle können genaue Zeiten für den ausgeführten Code bestimmt werden . Zudem lässt sich konkret auf das Problem bezogen eine exakte Lösung finden, wodurch meistens der Maschinencode kleiner ausfällt. Zum Nachteil gegenüber C ist sicherlich die Übersichtlichkeit. Der Programmcode zieht sich auch bei kleineren Problemen über mehrere Programmzeilen mit teilweise mehreren Programmsprüngen, sodass schnell der Überblick verloren geht. Aufgrund dessen ist auch hier ein genauer Programmablaufplan bzw. eine genaue Programmstruktur unumgänglich. Außerdem müssen die entsprechenden Befehle des Controllertyps bekannt sein, die im Folgenden vorgestellt werden.

Die Programmentwicklung auf dem PC ist ähnlich zu C. Meistens wird auch mit integrierten Entwicklungsumgebungen programmiert. Anstelle eines C-Codes wird dann das Assemblerprogramm in das entsprechende Projekt eingebunden. Dieses wird dann anstelle eines Compilers mit einem Makroassembler in das Maschinenprogramm umgesetzt. Der erzeugt HEX-File wird dann auf die gleiche Art und Weise in den Controller übertragen.

Häufig ist es in den Entwicklungsumgebungen möglich, C-Code und Assemblercode zu mischen. Dann lässt sich zum Beispiel eine zeitkritischer Teil in Assembler und der Rest des Programms übersichtlich in C lösen.

Eine Programmstruktur in Assembler ist sicherlich notwendig, wird aber nicht wie bei C erzwungen. In C muss zum Beispiel immer mit Funktionen gearbeitet werden. Selbst das Hauptprogramm steht in der Funktion main().

In Assembler reihen sich die Befehle nur hintereinander. Der Programmierer selbst kann aber Unterprogramme mithilfe von Sprungbefehlen erzeugen.

In diesem Buch wird wie in C mit der Entwicklungsumbegung von Keil gearbeitet. Es lässt sich, wie bei den meisten anderen Umgebungen auch, ein include File einbinden. So gibt es auch einen include-File für den Atmel AT89C51AC2, der hier verwendet weden soll. Für den Typ AC3 ist wie in C noch kein include-File vorhanden.

Die Programme beginnen dann wie folgt:

```
/* Programm: Programmstruktur
   datei: einstieg.asm
*/
```

```
$nomod51
$include(t89c51ac2.INC)

nop\index{nop} ; Befehl: No Operation
```

Kommentare können mit einem Semikolon, mit zwei Schrägstrichen für eine Zeile oder einem Schrägstrich mit Stern für einen Block eingeleitet werden.

15 Der Befehlssatz der Controller-Familie 8051

Es wird hier der Befehlssatz vorgestellt, der als Zusammenfassung (Instruction Set Summary) im Handbuch (Users Manual) des Controllers angegeben ist. Die Befehle gelten für alle Controllerbausteine der 8051er-Familie.

Um den Rahmen dieses Buches nicht zu sprengen, werden die Befehle nicht im einzelnen ausführlich erläutert, sondern zu Funktionsgruppen zusammengefasst vorgestellt. Die englische Kurzbeschreibung der einzelnen Befehle ist gut verständlich und beschreibt sie präzise und treffend.

15.1 Befehle zum Datentransfer

Die Transferbefehle für den internen Speicher und die internen Register beginnen mit MOV. Die Transportbefehle zwischen Akku und externem Datenspeicher beginnen mit MOVX, die zwischen Akku und externem Programmspeicher mit MOVC.

Mnemonic		Description	Byte	Cycle
Data Transfer				
MOV	A,Rn	Move register to accumulator	1	1
MOV	A,direct [1)]	Move direct byte to accumulator	2	1
MOV	A,@Ri	Move indirect RAM to accumulator	1	1
MOV	A,#data	Move immediate data to accumulator	2	1
MOV	Rn,A	Move accumulator to register	1	1
MOV	Rn,direct	Move direct byte to register	2	2
MOV	Rn,#data	Move immediate data to register	2	1
MOV	direct,A	Move accumulator to direct byte	2	1
MOV	direct,Rn	Move register to direct byte	2	2
MOV	direct,direct	Move direct byte to direct byte	3	2
MOV	direct,@Ri	Move indirect RAM to direct byte	2	2
MOV	direct,#data	Move immediate data to direct byte	3	2
MOV	@Ri,A	Move accumulator to indirect RAM	1	1
MOV	@Ri,direct	Move direct byte to indirect RAM	2	2

[1)] MOV A,ACC is not a valid instruction

Mnemonic		Description	Byte	Cycle
MOV	@Ri,#data	Move immediate data to indirect RAM	2	1
MOV	DPTR,#data 16	Load data pointer with a 16-bit constant	3	2
MOVC	A,@A+DPTR	Move code byte relative to DPTR to accumulator	1	2
MOVC	A,@A+PC	Move code byte relative to PC to accumulator	1	2
MOVX	A,@Ri	Move external RAM (8-bit addr.) to A	1	2
MOVX	A,@DPTR	Move external RAM (16-bit addr.) to A	1	2
MOVX	@Ri,A	Move A to external RAM (8-bit addr.)	1	2
MOVX	@DPTR,A	Move A to external RAM (16-bit addr.)	1	2
PUSH	direct	Push direct byte onto stack	2	2
POP	direct	Pop direct byte from stack	2	2
XCH	A,Rn	Exchange register with accumulator	1	1
XCH	A,direct	Exchange direct byte with accumulator	2	1
XCH	A,@Ri	Exchange indirect RAM with accumulator	1	1
XCHD	A,@Ri	Exchange low-order nibble indir. RAM with A	1	1

MOV-Befehle

Hinter der Transport-Operation MOV steht zuerst das Ziel und dann, durch ein Komma getrennt, die Quelle.

A = Akkumulator

direkt = Adresse, symbolisch oder absolut, im direkt adressierbaren internen RAM-Speicher.

```
MOV direkt,A
MOV A,direkt
MOV direkt,direkt
```

Beispiele:

```
MOV 20,A
MOV A,P1
MOV 21,P5
```

Rn = Register R0 bis R7 der aktuellen Registerbank

```
MOV Rn,A
MOV A,Rn
MOV Rn, direkt
```

Beispiele:

```
MOV R0,A
MOV A,R7
MOV R3,45
```

Ri = Register R0 oder R1
Diese Register werden als Zeiger für die indirekte Adressierung z. B. im internen RAM-Speicher genutzt. Der Zeiger enthält die 8-Bit-Adresse.

```
MOV A,@Ri
MOV @Ri,A
MOV @Ri,direkt
```

Beispiele:

```
MOV A,@R0
MOV @R1,P1
```

@ = Kennzeichnung für indirekte Adressierung. Es folgt das Register mit der Adresse.

\# = Kennzeichnung für eine folgende Konstante

data = 8-Bit-Konstante
data 16 = 16-Bit-Konstante

```
MOV A,#data
MOV DPTR,#data 16
```

Beispiele:

```
MOV A,#0F
MOV DPTR,#E100
```

MOVX

MOVX kennzeichnet einen Datentransfer zwischen Akku und externem Datenspeicher. Der Transfer erfolgt immer über indirekte Adressierung. Bei einer 16-Bit-Adresse erfolgt die Adressierung über den Datapointer DPTR, bei einer 8-Bit-Adresse über Ri.

```
MOVX A,@DPTR
MOVX @DPTR,A
MOVX A,@R1
```

MOVC

MOVC kennzeichnet einen Datentransfer vom externen Programmspeicher zum Akku. Die Adressierung erfolgt indirekt. Die Adresse wird gebildet aus der Summe der Inhalte von Akku plus Datapointer DPTR oder aus der Summe der Inhalte von Akku plus Programmcounter PC.

```
MOVC A,@A+DPTR
MOVC A,@A+PC
```

Die in diesem Buch vorgestellten Assemblerprogramme werden mit einem Makroassembler übersetzt, der Zahlen aus verschiedenen Zahlensystemen in Maschinensprache übersetzen kann. Deshalb muss das Zahlensystem hinter der Zahl angegeben werden. Ein H hinter der Zahl steht für Hexadezimalzahlen. Weiter erlaubt der Makroassembler auch symbolische Bezeichnungen von Adressen oder Zahlen. Um eine absolute Zahl von einer symbolischen Angabe unterscheiden zu können, ist bei allen Hexzahlen, die mit einem Buchstaben beginnen, zur Kennzeichnung eine Null voranzusetzen.

Alle Buchstaben dürfen groß- oder kleingeschrieben werden.

Beispiele:

```
MOV A,0F5H
mov 0e1h,a
```

15.2 Befehle zu arithmetischen Operationen

Die Befehle zu arithmetischen und logischen Operationen beeinflussen die Flags im Programmstatuswort PSW. PSW ist ein Spezial-Funktions-Register.

Spezial-Funktions-Register PSW

PSW bitadressierbar Resetwert 00h

CY	AC	F0	RS1	RS0	OV	F1	P

Bit	Funktion
CY	Carry Flag, Übertragsbit
AC	Auxilary Carry Flag, Hilfs-Übertragsbit
F 0	Benutzerflag 0, zur freien Verfügung
RS1 RS0	Auswahl der Registerbank
0 0 0 1 1 0 1 1	Bank 0 1 2 3
OV	Overflow Flag, Überlaufbit
F1	Benutzerflag 1, zur freien Verfügung
P	Parityflag, Paritätsbit

Erläuterung der Flags

Carry Flag CY

Wird bei einem Übertrag des Ergebnisses gesetzt. Der Übertrag tritt bei einer Addition auf, wenn das Ergebnis größer 8 Bit ist. Er tritt bei einer Subtraktion auf, wenn das Ergebnis negativ wird. (Borrow)

Auxiliary Flag AC

Wird bei einem Übertrag der niederwertigen vier Bits des Ergebnisses gesetzt. Wird beim Rechnen mit BCD-Zahlen benötigt.

Overflow Flag OV

Eine Kopie von Bit 7 des Ergebnisses. Wird benötigt beim Rechnen mit vorzeichenbehafteten Zahlen (Vorzeichenbit).

Parity Flag

Wird gesetzt, wenn sich eine ungerade Anzahl von 1-Signalen im Akku befindet.

Das B-Register

Das B-Register ist ein Spezial-Funktions-Register, welches bei der Multiplikation und Division benötigt wird.

Bei einer Multiplikation stehen in A und B die beiden 8-Bit-Zahlen. Nach Ausführung des Befehls befindet sich der niederwertige Teil des Ergebnisses in A und der höherwertige Teil in B. Ist der Teil in B > 0 (Ergebnis > 255), wird das Overflow Flag gesetzt.

Bei einer Division wird die 8-Bit-Zahl in A durch die 8-Bit-Zahl in B dividiert. Nach Ausführung des Befehls steht in A das Ergebnis als ganze Zahl. Der Rest steht in B.

Mnemonic		Description	Byte	Cycle
Arithmetic Operations				
ADD	A,Rn	Add register to accumulator	1	1
ADD	A,direct	Add direct byte to accumulator	2	1
ADD	A,@Ri	Add indirect RAM to accumulator	1	1
ADD	A,#data	Add immediate data to accumulator	2	1
ADDC	A,Rn	Add register to accumulator with carry flag	1	1
ADDC	A,direct	Add direct byte to A with carry flag	2	1
ADDC	A,@Ri	Add indirect RAM to A with carry flag	1	1
ADDC	A,#data	Add immediate data to A with carry flag	2	1
SUBB	A,Rn	Subtract register from A with borrow	1	1
SUBB	A,direct	Subtract direct byte from A with borrow	2	1
SUBB	A,@Ri	Subtract indirect RAM from A with borrow	1	1
SUBB	A,#data	Subtract immediate data from A with borrow	2	1
INC	A	Increment accumulator	1	1
INC	Rn	Increment register	1	1
INC	direct	Increment direct byte	2	1
INC	@Ri	Increment indirect RAM	1	1
DEC	A	Decrement accumulator	1	1
DEC	Rn	Decrement register	1	1
DEC	direct	Decrement direct byte	2	1
DEC	@Ri	Decrement indirect RAM	1	1
INC	DPTR	Increment data pointer	1	2
MUL	AB	Multiply A and B	1	4
DIV	AB	Divide A by B	1	4
DA	A	Decimal adjust accumulator	1	1

15.3 Befehle zu logischen Operationen

Mnemonic		Description	Byte	Cycle
Logic Operations				
ANL	A,Rn	AND register to accumulator	1	1
ANL	A,direct	AND direct byte to accumulator	2	1
ANL	A,@Ri	AND indirect RAM to accumulator	1	1
ANL	A,#data	AND immediate data to accumulator	2	1
ANL	direct,A	AND accumulator to direct byte	2	1
ANL	direct,#data	AND immediate data to direct byte	3	2
ORL	A,Rn	OR register to accumulator	1	1
ORL	A,direct	OR direct byte to accumulator	2	1
ORL	A,@Ri	OR indirect RAM to accumulator	1	1
ORL	A,#data	OR immediate data to accumulator	2	1
ORL	direct,A	OR accumulator to direct byte	2	1
ORL	direct,#data	OR immediate data to direct byte	3	2
XRL	A,Rn	Exclusive OR register to accumulator	1	1
XRL	A,direct	Exclusive OR direct byte to accumulator	2	1
XRL	A,@Ri	Exclusive OR indirect RAM to accumulator	1	1
XRL	A,#data	Exclusive OR immediate data to accumulator	2	1
XRL	direct,A	Exclusive OR accumulator to direct byte	2	1
XRL	direct,#data	Exclusive OR immediate data to direct byte	3	2
CLR	A	Clear accumulator	1	1
CPL	A	Complement accumulator	1	1
RL	A	Rotate accumulator left	1	1
RLC	A	Rotate accumulator left through carry	1	1
RR	A	Rotate accumulator right	1	1
RRC	A	Rotate accumulator right through carry	1	1
SWAP	A	Swap nibbles within the accumulator	1	1

15.4 Befehle zur Programm- und Maschinensteuerung

Mnemonic		Description	Byte	Cycle
Program and Machine Control				
ACALL	addr11	Absolute subroutine call	2	2
LCALL	addr16	Long subroutine call	3	2
RET		Return from subroutine	1	2
RETI		Return from interrupt	1	2
AJMP	addr11	Absolute jump	2	2
LJMP	addr16	Long jump	3	2
SJMP	rel	Short jump (relative addr.)	2	2
JMP	@A+DPTR	Jump indirect relative to the DPTR	1	2
JZ	rel	Jump if accumulator is zero	2	2
JNZ	rel	Jump if accumulator is not zero	2	2
JC	rel	Jump if carry flag is set	2	2
JNC	rel	Jump if carry flag is not set	2	2
JB	bit,rel	Jump if direct bit is set	3	2
JNB	bit,rel	Jump if direct bit is not set	3	2
JBC	bit,rel	Jump if direct bit is set and clear bit	3	2
CJNE	A,direct,rel	Compare direct byte to A and jump if not equal	3	2
CJNE	A,#data,rel	Compare immediate to A and jump if not equal	3	2
CJNE	Rn,#data,rel	Compare immed. to reg. and jump if not equal	3	2
CJNE	@Ri,#data,rel	Compare immed. to ind. and jump if not equal	3	2
DJNZ	Rn,rel	Decrement register and jump if not zero	2	2
DJNZ	direct,rel	Decrement direct byte and jump if not zero	3	2
NOP		No operation	1	1

Sprungbefehle

Der Befehl ACALL beinhaltet eine 11-Bit-Adresse. Diese wird rechtsbündig im Programm Counter eingesetzt. Die oberen 5 Bit im PC bleiben erhalten.

Der Befehl LCALL beinhaltet eine 16-bit-Adresse. Das gleiche gilt für AJMP und LJMP.

Der Befehl SJMP enthält eine 8-Bit-Adresse. Damit sind relative Sprünge im Bereich von $+127$ bis -128 möglich.

Da der Programmierer bei Verwendung eines Makroassemblers mit symbolischen Sprungmarken arbeiten kann, berechnet der Makroassembler die absoluten Sprungadressen. Falls die Sprungweite überschritten wird, macht der Makroassembler ihn darauf aufmerksam. Zum Glück entfällt damit das Berechnen von Sprungweiten oder Sprungadressen.

15.5 Befehle zur Bitverarbeitung

Der Controller hat in der unteren Hälfte des internen Datenspeichers einen Bitspeicher mit 128 einzeln adressierbaren Bits. Der Speicher ist also für die Verarbeitung einzelner Bits eingerichtet.

Ein Blick auf den Befehlssatz zeigt, dass es eine ganze Reihe von Befehlen für den Transport und für Boole'sche Verknüpfungen einzelner Bits gibt. Boole'sche Verknüpfungen sind die logischen Verknüpfungen UND, ODER und NICHT, die in der Steuerungstechnik bei der Erstellung von Funktionsplänen verwendet werden.

Boolean Variable Manipulation				
CLR	C	Clear carry flag	1	1
CLR	bit	Clear direct bit	2	1
SETB	C	Set carry flag	1	1
SETB	bit	Set direct bit	2	1
CPL	C	Complement carry flag	1	1
CPL	bit	Complement direct bit	2	1
ANL	C,bit	AND direct bit to carry flag	2	2
ANL	C,/bit	AND complement of direct bit to carry	2	2
ORL	C,bit	OR direct bit to carry flag	2	2
ORL	C,/bit	OR complement of direct bit to carry	2	2
MOV	C,bit	Move direct bit to carry flag	2	1
MOV	bit,C	Move carry flag to direct bit	2	2

Wie aus dem Befehlssatz zu erkennen, wird bei der Verknüpfung einzelner Bits das Carry-Flag C als „Akkumulator" genutzt. Die Boole'schen Verknüpfungen finden alle in C statt. Außer den Befehlen zur Boole'schen Verknüpfung werden bei der Bitverarbeitung auch bedingte Sprungbefehle benötigt, wenn sich die Sprungbedingung auf ein Bit oder auf C bezieht.

Program and Machine Control				
JC	rel	Jump if carry flag is set	2	2
JNC	rel	Jump if carry flag is not set	2	2
JB	bit.rel	Jump if direct bit is set	3	2
JNB	bit.rel	Jump if direct bit is not set	3	2
JBC	bit.rel	Jump if direct bit is set and clear bit	3	2

16 Controller-Grundfunktionen in Assembler

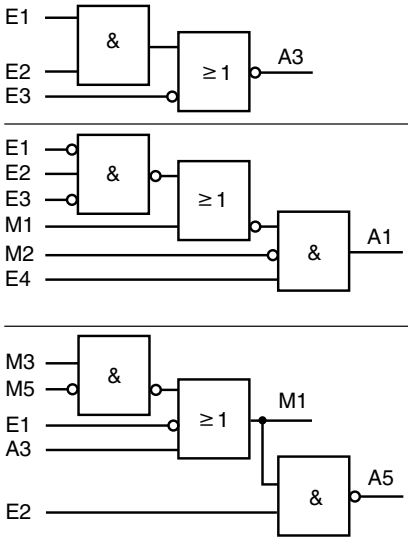
16.1 Programmieren von Verknüpfungssteuerungen

Es sollen, ähnlich zum C-Teil, zunächst Assemblerprogramme für Verknüpfungssteuerungen vorgestellt werden. Als Ein- und Ausgangssignale werden symbolische Namen für die Byte- oder Bitvariablen verwendet. Die Zuordnung der symbolischen Variablennamen zu den absoluten Byte- oder Bitadressen erfolgt über eine Zuweisungsliste zu Anfang des Assemblerprogramms. Das Übersetzerprogramm, der Makroassembler, setzt dann beim Übersetzen in die Maschinensprache für die symbolischen Bezeichnungen die absoluten Adressen ein. Mit den symbolischen Namen für die Variablen werden die Programme leichter lesbar. Außerdem können sie unabhängig von absoluten Adressen geschrieben werden; die Programme werden adressenunabhängig.

Alle hier vorgestellten Programme können auf derselben Hardware wie in der Programmiersprache C getestet werden.

Beispiele für die Programmierung von Verknüpfungssteuerungen

Verknüpfungssteuerung



Assemblerprogramm

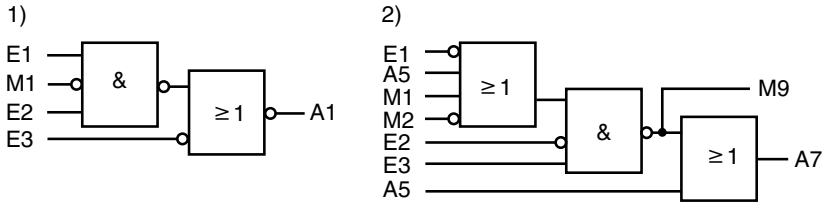
```
MOV   C,E1    ; UND
ANL   C,E2    ;
ORL   C,/E3   ; ODER
CPL   C
MOV   A3,C
```

```
MOV   C,E2    ; UND
ANL   C,/E1
ANL   C,/E3
CPL   C
ORL   C,M1    ; ODER
CPL   C
ANL   C,/M2   ; UND
ANL   C,E4
MOV   A1,C
```

```
MOV   C,M3    ; ODER
ANL   C,/M5
CPL   C
ORL   C,/E1   ; ODER
ORL   C,A3
MOV   M1,C
ANL   C,E2    ; UND
MOV   A5,C
```

Übung 16.1

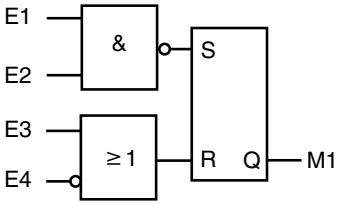
Schreiben Sie für folgende logischen Verknüpfungen die Assemblerprogramme.



Enthält die Schaltung Flipflops zur Speicherung von Signalen, werden die Bits des Flipflop-Speichers entweder gesetzt oder rückgesetzt. Das Setzen oder Rücksetzen erfolgt jedoch nur, wenn die Bedingungen dafür erfüllt sind. Andernfalls wird das Setzen oder Rücksetzen der Bits übersprungen. Dazu lassen sich die bedingten Sprungbefehle verwenden. Im Folgenden wird die Programmierung von rücksetzdominaten Fliplops gezeigt.

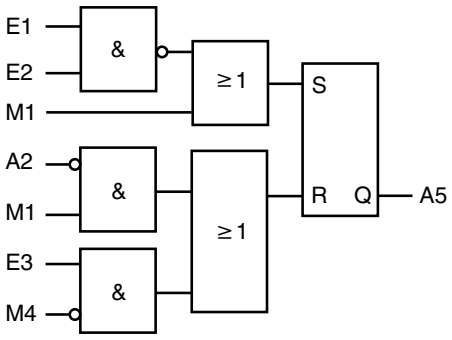
Beispiele für die Programmierung von Verknüpfungssteuerungen mit Flipflops

Verknüpfungssteuerung mit Flipflops



Assemblerprogramm

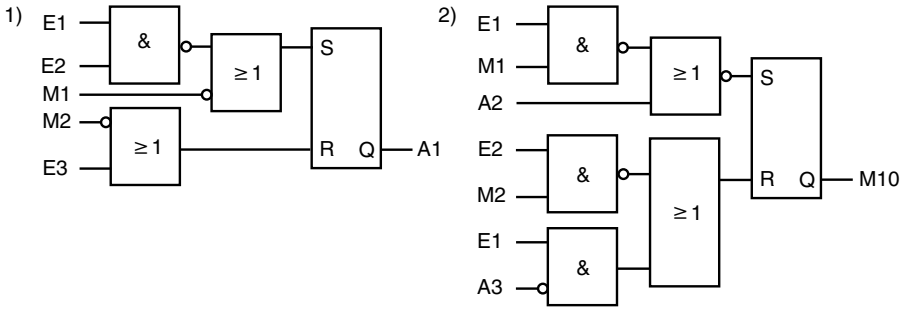
```
    MOV  C,E1   ; M1 setzen?
    ANL  C,E2
    CPL  C
    JNC  S1     ; Nein
    SETB M1     ; Ja
S1: MOV  C,E3   ; M1 rücksetzen?
    ORL  C,/E4
    JNC  S2     ; Nein
    CLR  M1     ; Ja
S2:
```




```
    MOV  C,E1   ; A5 setzen?
    ANL  C,E2
    CPL  C
    ORL  C,M1
    JNC  S1     ; Nein
    SETB A5     ; Ja
S1: MOV  C,M1   ; A5 rücksetzen?
    ANL  C,/A2
    MOV  M5,C   ; Hilfsmerker
    MOV  C,E3
    ANL  C,/M4
    ORL  C,M5
    JNC  S2     ; Nein
    CLR  A5     ; Ja
S2:
```

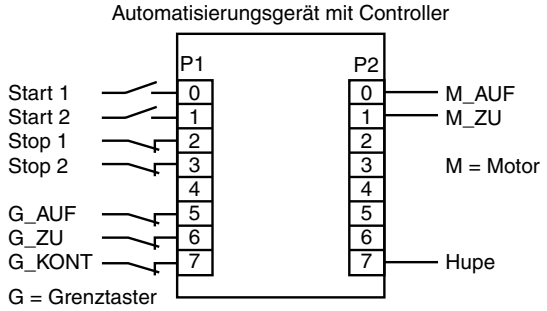
Übung 16.2

Schreiben Sie für folgende logische Verknüpfungen mit Flipflops die Assemblerprogramme.

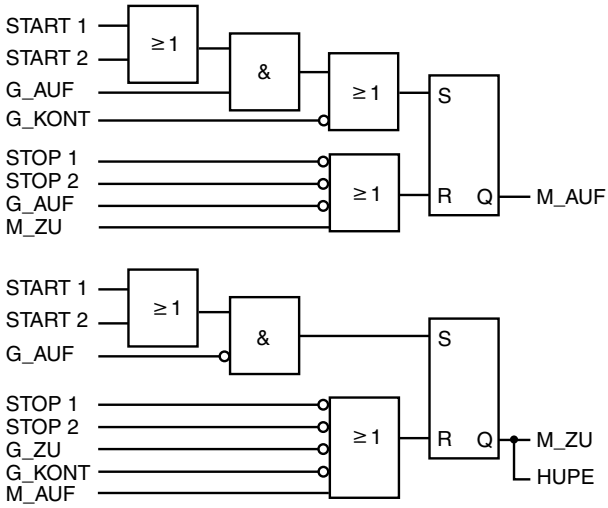


Beispiel 16.1

Setzen Sie das Programm „Steuerung eines Hallentores (Abschnitt 8.2.1) in Assembler um! Hierzu sind nochmals die Zuordnungen der Ein- und Ausgänge als auch die zugehörige Verknüpfungssteuerung dargestellt.



Verknüpfungssteuerung:



Programmstruktur

Auch ein Assemblerprogramm muss zyklisch arbeiten, damit in Echtzeit auf die Eingangssignale reagiert werden kann. Da hier mehrere Verknüfungssteuerungen verarbeitet werden, muss zunächst, wie in C, ein Signalabbild der Ein- und Ausgänge erzeugt werden. Erst anschließend lässt sich zuverlässig die Bitverarbeitung durchführen. Am Ende des Programmzyklus wird dann das Ergebnis in den Port P2 übertragen. So kann es nicht vorkommen, dass das Tor gleichzeitig auf- und zu fährt, welches ohne Erzeugung der Signalabbilder passieren könnte.

Die Bearbeitungszeit eines Programmdurchlaufs liegt im Millisekundenbereich. Dabei darf das Programm keine Rückwärtssprünge und Warteschleifen enthalten. Bei dieser Zykluszeit erfolgt die Reaktion auf wechselnde Eingangssignale auch ohne Interruptbetrieb immer in Echtzeit.

Assemblerprogramm

Das Assemblerprogramm wird von einem Makroassembler in die Maschinensprache übersetzt. Der Makroassembler erlaubt symbolische Sprungmarken und Operanden. Diese müssen zu Beginn des Programms den absoluten Adressen oder Werten zugewiesen werden. Bei den Sprungmarken erfolgt die Zuweisung durch das Setzen der Marke an der entsprechenden Adresse. Für diese und andere Programmierhilfen gibt es spezielle Steueranweisungen für den Makroassembler. Da diese Steueranweisungen den Assemblerbefehlen gleichen, nennt man sie auch „Pseudobefehle“. Die EQU-Anweisung (s. u. Rahmenprogramm, Zuweisungen) bei den Zuweisungen ist solch ein Pseudobefehl.

Beim Übersetzen in die Maschinensprache werden dann für die symbolischen Werte und Adressen die absoluten Zahlen eingesetzt.

Rahmenprogramm

Einbinden des Controllers:

Wie am Anfang des Kapitels vorgestellt, wird zunächst die Bibliothek des zu programmierenden Controllers eingebunden.

```
$nomod51
$include(t89c51ac2.INC)
```

Zuweisungen:

Die Eingangssignale an Port P1 werden in einem Byte im bitadressierbaren internen RAM-Speicher unter der Adresse 20h gespeichert. Dieser Speicher für das Eingangssignal-Abbild erhält die symbolische Adresse EIN1. Das Ausgangssignal-Abbild erhält den Namen AUS2 und wird in der Adresse 21h abgelegt. Bei der Ausgabe wird es nach P2 transportiert. Die Adresse 22h wird für interne Merker reserviert und erhält den symbolischen Namen M1. Die einzelnen Bits dieser Byteadressen lassen sich durch einen Punkt getrennt hinter die symbolische Byteadresse schreiben.

```
EIN1 EQU 20h
                    ; Eingangssignal-Abbild
START1 EQU EIN1.0
START2 EQU EIN1.1
STOP1 EQU EIN1.2
STOP2 EQU EIN1.3
G_AUF EQU EIN1.5
G_ZU EQU EIN1.6
G_KONT EQU EIN1.7
AUS2 EQU 21h
                    ; Ausgangssign.-Abbild
M_AUF EQU AUS2.0
M_ZU EQU AUS2.1
HUPE EQU AUS2.7
M1 EQU 22h          ; interne Merker
```

Initialisierung:

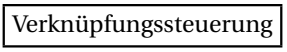
Bei der Initialisierung werden die Register oder Speicher der Steuerung in die Grundstellung gebracht. Das Eingangssignal- und Ausgangssignal-Abbild wird auf null gestellt. Damit über Port P1 eingelesen werden kann, müssen die Port-Flipflops auf 1 gesetzt werden.

```
MOV EIN1,#00h
MOV AUS2,#00h
MOV P1,#0FFh
```

Eingangssignale lesen:

Nun folgt die eigentliche Verknüpfungssteuerung, die im Folgenden beschrieben wird.

```
ANF: MOV EIN1,P1
```



Ausgangssignale ausgeben:

```
MOV P5,AUS5
```

Rücksprung zum Anfang:

```
LJMP ANF
```

Die eigentliche Verknüpfungsprogrammierung wird im folgenden gezeigt:

Assemblerprogramm der Verknüpfungssteuerung:

```
$nomod51
$include(t89c51ac2.INC)
     MOV  C,START1    ; M_AUF setzen?
     ORL  C,START2
     ANL  C,G_AUF
     ORL  C,/G_KONT
     JNC  S1          ; Nein
     SETB M_AUF       ; Ja
S1:  MOV  C,M_ZU      ; M_AUF rücksetzen?
     ORL  C,/STOP1
     ORL  C,/STOP2
     ORL  C,/G_AUF
     JNC  S2          ; Nein
     CLR  M_AUF       ; Ja
S2:  MOV  C,START1    ; M_ZU setzen?
     ORL  C,START2
     ANL  C,/G_AUF
     JNC  S3          ; Nein
     SETB M_ZU        ; Ja
S3:  MOV  C,M_AUF     ; M_ZU rücksetzen?
     ORL  C,/STOP1
     ORL  C,/STOP2
     ORL  C,/G_Zu
     ORL  C,/G_KONT
     JNC  S4          ; Nein
```

```
     CLR  M_Zu         ; Ja
S4:  MOV  C,M_ZU       ; HUPE = M_ZU
     MOV  HUPE,C
END
```

Übung 16.3

Schreiben Sie das Programm für die Transportsteuerung von Übung 8.4 in Assembler einschließlich der Zuweisungen und der Initialisierung.

16.2 Blink- und Lauflichtprogramme in Assembler

Blinklicht 1

Alle Bits von Port 2 sollen mit einer sichtbaren Frequenz blinken.

Assemblerprogramm:

```
;********************* Blinklicht 1, Blink1.asm **********************
;
;synchrones Blinken aller Ausgänge Port 2 mit sichtbarer Frequenz
;
;*********************************************************************
$nomod51
$include(t89c51ac2.INC)
    mov p2,#0ffh          ;alle Ausgänge P2 ein
b11:mov r2,#0ffh          ;Wartezeit: 2 Register als 16-Bit-Zähler
t2: mov r1,#0ffh          ;herunterzählen
t1: djnz r1,t1
    djnz r2,t2
    mov a,p2              ;P2 invertieren
    cpl a
    mov p2,a
    ajmp b11
END
```

Blinklicht 2

Die blinkenden Bits an Port 2 sollen mit einem Schalter an Port 1.0 angehalten werden.

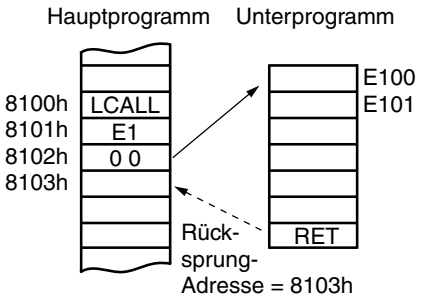
Assemblerprogramm:

```
;********************* Blinklicht 2, Blink2.asm **********************
;
;Synchrones Blinken aller Ausgänge an Port 2 mit sichtbarer Frequenz,
;wenn Einschalter P1.0 = 1.
```

```
;
;***********************************************************************
$nomod51
$include(t89c51ac2.INC)
    mov p2,#0ffh          ;alle Ausgänge P2 ein
b11:jnb p1.0,b11          ;Einschalter = 0? Ja: Warteschleife
    mov r2,#0ffh          ;Wartezeit: 2 Register als 16-Bit-Zähler
t2: mov r1,#0ffh          ;herunterzählen
t1: djnz r1,t1
    djnz r2,t2
    mov a,p2              ;P2 invertieren
    cpl a
    mov p2,a
    ajmp b11
END
```

16.3 Unterprogramme

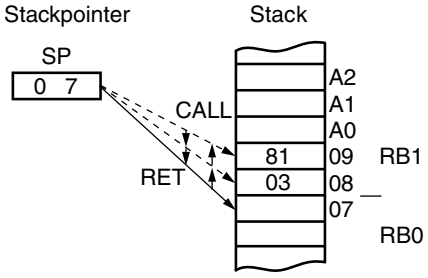
Unterprogramme werden vom Hauptprogramm mit dem Befehl ACALL oder LCALL aufgerufen. Bei der Ausführung des Befehls wird erst die Rücksprungadresse auf dem Stack gerettet. Dann gibt der Controller die Anfangsadresse des Unterprogramms aus, um dieses zu bearbeiten.



Das Unterprogramm endet mit dem Befehl RET. Dieser bewirkt ein Rückholen der Adresse vom Stack. Diese Adresse wird als nächstes ausgegeben und damit das Hauptprogramm fortgesetzt.

Retten der Rücksprungsadresse auf dem Stack

Der Stack befindet sich im internen RAM-Speicher. Er wird durch den Stackpointer verwaltet.



Bei einem CALL-Befehl wandert der Stackpointer zwei Adressen aufwärts. bei einem RET-Befehl zwei Adressen abwärts. Die zu rettende Rücksprungadresse ins Hauptprogramm wird zu zweimal 8 Bit auf die Adressen im Stack gelegt, auf die der Stackpointer zeigt. Am Ende des Unterprogramms wird die Adresse durch RET wieder in den Programm-Counter zurückgeholt.

Nach einem Reset steht im Stackpointer die Adresse 07. Der Stack beginnt dann ab Adresse 08.

Der Stackpointer lässt sich durch Laden einer Konstanten auf jede Adresse innerhalb des internen RAMs legen.

Retten von Registerinhalten auf dem Stack

Genau wie die Rücksprungadresse lassen sich auch andere Speicherinhalte des direkt adressierbaren Speicherbereiches auf dem Stack retten und nach der Bearbeitung des Unterprogramms wieder zurückholen. Mit dem PUSH-Befehl werden Inhalte auf dem Stack gerettet, mit dem POP-Befehl wieder zurückgeholt.

Retten von Speicherinhalten:
PUSH direkt
Funktion wie bei Befehl CALL.

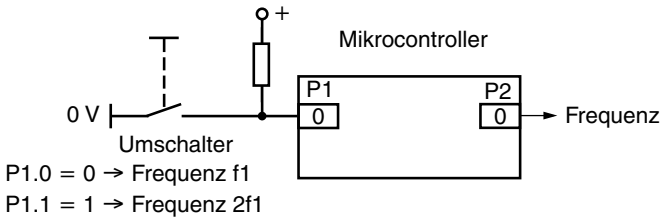
Rückholen der Speicherinhalte:
POP direkt
Funktion wie bei Befehl RET.

Lauflicht 1 mit Unterprogramm für Zeitwerk

```
;****************** Lauflicht 1, LAUF1.ASM *************************
;
;Lauflicht an Port 2
;
;***************************************************************************
$nomod51
$include(t89c51ac2.INC)
       setb p2.1         ;Lauflicht Ausgangsmuster
anf:   lcall zeit2       ;Wartezeit
       mov a,p2          ;Lauflicht 1 Stelle links schieben
       rl a
       mov p2,a
       ljmp anf
;-------------------- Unterprogramm Zeit2 ----------------------------
zeit2: mov r2,0ffh
   t2: mov r1,0ffh
   t1: djnz r1,t1
       djnz r2,t2
       ret
;***************************************************************************
```

Übung 16.4

Schreiben Sie ein Programm für einen umschaltbaren Frequenzgenerator. Die Frequenzen sollen im sichtbaren Bereich liegen und im Verhältnis 2 : 1 umschaltbar sein.



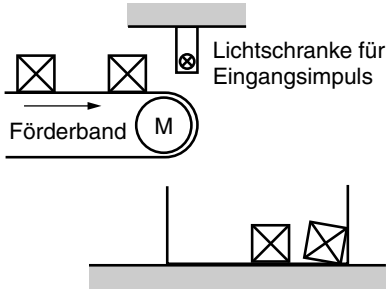
16.4 Zählersteuerung

In einer Verpackungsanlage soll ein Warenkorb mit einer bestimmten Anzahl Päckchen gefüllt werden. Die Anzahl der Päckchen ist auf einem Bedienpult einzustellen.

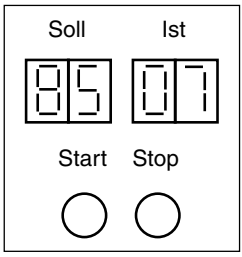
Die Päckchen kommen auf einem Transportband an und fallen in den Korb. Dabei müssen sie gezählt werden. Dazu werden sie von einer Lichtschranke erfasst. Ist die eingestellte Anzahl erreicht, hält das Transportband an.

16.4.1 Steuerungsbeschreibung

Technologieschema



Bedientableau



Auf dem Bedientableau lässt sich die Anzahl Päckchen von 00 bis 99 einstellen (= Sollwert). Die Anzahl der durchgelaufenen Päckchen (= Istwert) wird auf einer zweistelligen Siebensegment-Anzeige dargestellt.

Funktion

Der Istwert ist gleich dem Sollwert, das Transportband steht.

Der neue Sollwert wird eingestellt und die Start-Taste gedrückt. Daraufhin wird der eingestellte Sollwert in die Steuerung übernommen und der angezeigte Istwert auf null gestellt.

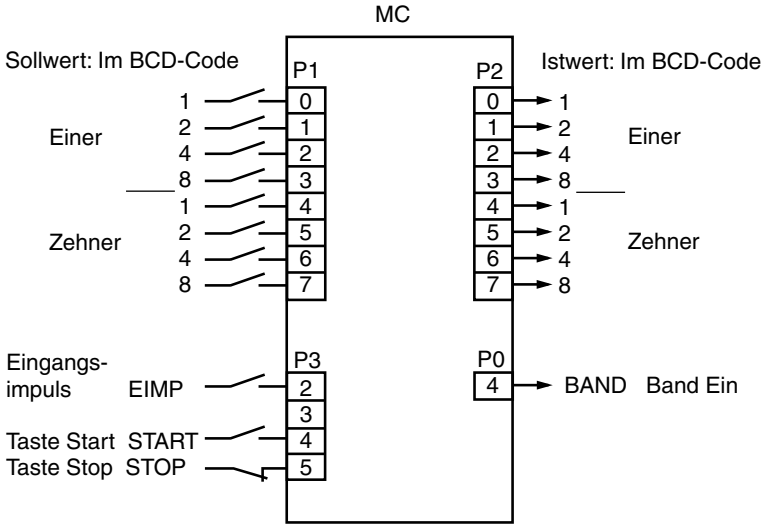
Das Band läuft an. Die Pakete laufen durch die Lichtschranke und werden gezählt. Der Zählerstand wird als Istwert angezeigt.

Stimmen Ist- und Sollwert überein, hält das Band an.

Eingangs- und Ausgangssignale der Steuerung

Die Eingangs- und Ausgangssignale der Steuerung werden über Verstärker mit Pegelumsetzung, sowie Optokopplern zur galvanischen Trennung an die Ports des Controllers angeschlossen.

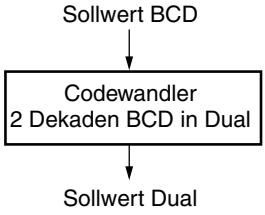
Anschlussplan



16.4.2 Programmentwicklung

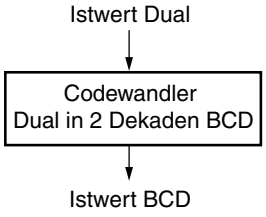
Strukturierung der Aufgabe

Der Sollwert wird als Dezimalzahl im BCD-Code eingegeben. Die interne Verarbeitung erfolgt jedoch im Dualcode. Deshalb muss eine Codewandlung zwei Dekaden BCD in Dual durchgeführt werden.



Istwertanzeige

Der Istwert wird intern über einen Zähler im Dualcode gebildet. Die Istwertanzeige erfolgt dezimal. Daher ist der Istwert im BCD-Code auszugeben. Es ist also erforderlich, zur Ausgabe den internen Dualcode in der BCD-Code umzuwandeln.

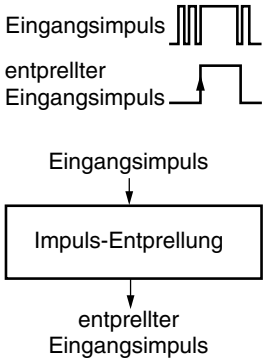


Istwertzähler

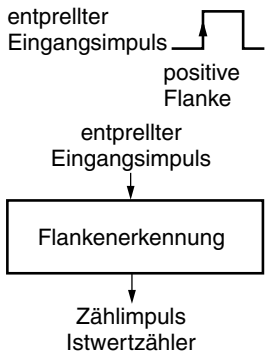
Zur Bildung des Istwertes müssen die Päckchen gezählt werden.

Dabei treten zwei Probleme auf:

1. Die Eingangsimpulse prellen.
   Zu Anfang und Ende des Zählimpulses treten kurze Fehlimpulse auf, die ausgeblendet werden müssen.

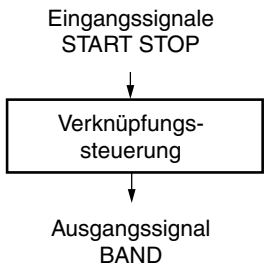


2. Pro Eingangsimpuls nur ein Zählimpuls.
   Da das Steuerungsprogramm zyklisch arbeitet, darf nicht ständig gezählt werden, so lange der Eingangsimpuls auf high liegt. Es darf daher nur einmal beim Signalwechsel gezählt werden. Es ist also eine Flankenerkennung notwendig; pro Flanke ein Zählimpuls.



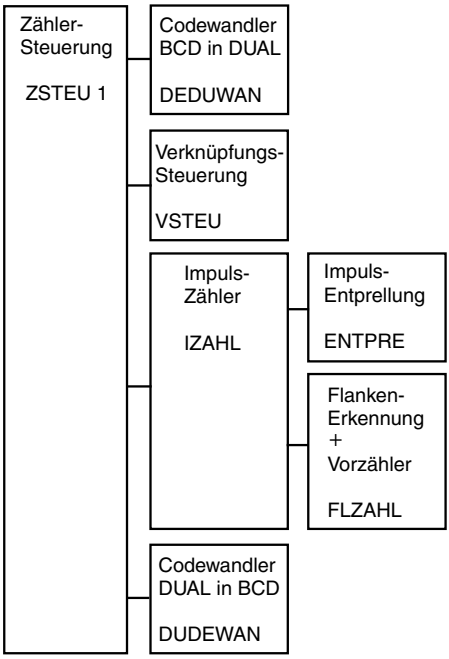
Steuerungsverknüpfungen

Um die geforderte Funktion zu erreichen, ist auf Start- und Stop-Taste zu reagieren, Ist- und Sollwert sind zu setzen, und das Band ist ein- und auszuschalten. Diese Forderungen sind mit einer Verknüpfungssteuerung zu realisieren.



Damit ergibt sich folgende Programmstruktur:

Programmstruktur:



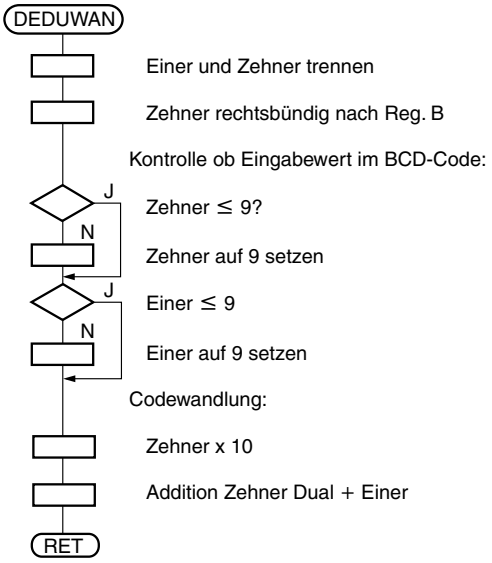
Im Folgenden werden die einzelnen Programmbausteine entwickelt.

Programmbaustein DEDUWAN, Codewandler BCD in DUAL

Der Programmbaustein wandelt eine zweidekadige gepackte BCD-Zahl in eine 8-Bit-Dualzahl.

Die BCD-Zahl steht vor der Wandlung im Akku. Nach der Wandlung steht die Dualzahl im Akku.

Lösungsstrategie:



Assemblerprogramm:

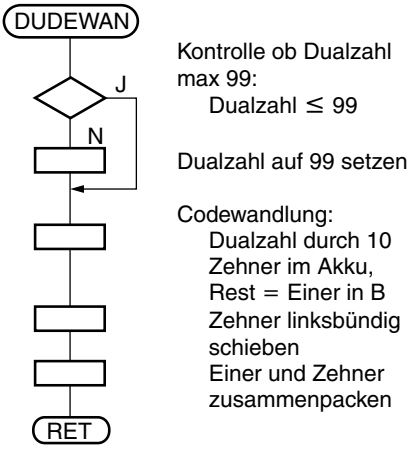
```
;*********************************************************************
;Wandler 2 Stellen BCD gepackt in Dual              Datei DEDUWAN.ASM
;*********************************************************************
;BCD-Zahl vor Wandlung im Akku,
;Dualzahl nach Wandlung im Akku,
;Hilfsregister R0
;---------------------------------------------------------------------
;Title "Dezimal/Dual-Wandler, Datei DEDUWAN"

deduwan:   mov r0,a              ;BCD-Zahl retten
           swap a                ;Zehner rechtsbündig in A
           anl a,#0fh
           mov b,a               ;Zehner nach b
           subb a,#10            ;Zehner begrenzen auf 9:
           jc bc1                ;Zehner =< 9
           mov b,#9              ;Zehner auf 9 setzen
bc1:       mov a,r0              ;Einer nach R0:
           anl a,#0fh
           mov r0,a
           subb a,#10            ;Einer begrenzen auf 9:
           jc bc2                ;Einer =< 9
           mov r0,#9             ;Einer auf 9 setzen
;---------------------- Codewandlung -----------------------------
bc2:       mov a,#10             ;Zehner x 10
           mul ab
           add a,r0              ;Zehner + Einer
           ret
;*********************************************************************
```

Programmbaustein DUDEWAN, Codewandler DUAL in BCD

Der Programmbaustein wandelt eine Dualzahl von maximal 99 in eine zweidekadige gepackte BCD-Zahl.

Vor der Wandlung steht die Dualzahl im Akku, nach der Wandlung die BCD-Zahl.

Lösungsstrategie:



Assemblerprogramm:

```
;*********************************************************************
;Wandler Dual max 99 in 2 Stellen BCD gepackt       Datei DUDEWAN.ASM
;*********************************************************************
;Dualzahl vor Wandlung im Akku, Zahlen über 99 werden auf 99 begrenzt,
;BCD-Zahl nach Wandlung im Akku,
;Hilfsregister R0
;---------------------------------------------------------------------
;Title "Dual/Dezimal-Wandler, Datei DUDEWAN"

dudewan:   mov r0,a                    ;Dualzahl retten
           clr c                       ;Dualzahl auf 99 begrenzen
           subb a,#100
           jc du1                      ;Dualzahl =<99
           mov r0,#99                  ;Dualzahl auf 99 setzen
;-------------------- Codewandlung ---------------------------------
du1:       mov a,r0                    ;Dualzahl max 99
           mov b,#10                   ;Dualzahl : 10
           div ab                      ;Einer in B, Zehner in A
                                       ;Einer und Zehner packen:
           swap a                      ;Zehner linksbündig in A
           orl a,b                     ;Einer und Zehner gepackt in A
           ret
;*********************************************************************
```

Programmbaustein Impulszähler IZAHL

Der Impulszähler ruft die beiden Programmbausteine „Impulsentprellung“ und „Flankenerkennung mit Vorzähler“ auf.

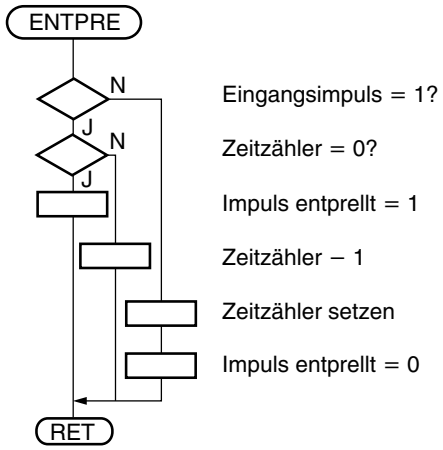
Programmbaustein Impulsentprellung ENTPRE

Die Entprellung wird über eine Zeitverzögerung realisiert. Erst wenn der Eingangsimpuls über eine bestimmte Zeit ansteht, ist er „echt“ und wird ausgewertet. Kürzere Impulse werden nicht registriert.

Die Zeitverzögerung geschieht über einen Zähler, der auf eine bestimmte Zahl gesetzt und bei jedem Programmzyklus um eins herabgezählt wird, wenn der Eingangsimpuls ansteht. Hat der Zähler null erreicht, ist die Zeit abgelaufen, und der Eingangsimpuls wird als „Eingangsimpuls entprellt“ abgespeichert.

Da Programmzyklen gezählt werden, ist die Zeitverzögerung abhängig von der Programmlänge. Von Vorteil ist, dass das Programm zyklisch bearbeitet wird und nicht in einer Warteschleife verweilt.

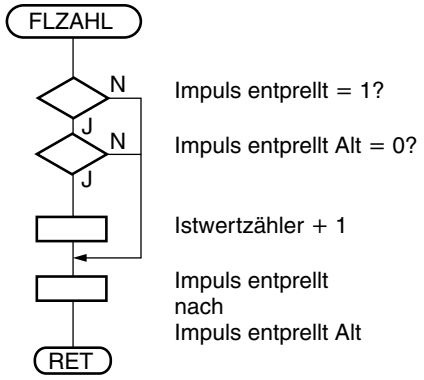
Lösungsstrategie:



Programmbaustein Flankenerkennung mit Vorzähler FLZAHL

Um Signalübergänge oder Flanken erkennen zu können, muss ein momentaner Zustand mit einem vorhergehenden verglichen werden. Im folgenden Programm wird der im momentanen Zyklus vorliegende entprellte Eingangsimpuls mit dem im vorhergehenden Zyklus gespeicherten verglichen. Ist der momentane Zustand des entprellten Impulses high und der im vorhergehenden Zyklus gespeicherte low, liegt eine positive Flanke vor, die gezählt wird.

Lösungsstrategie:



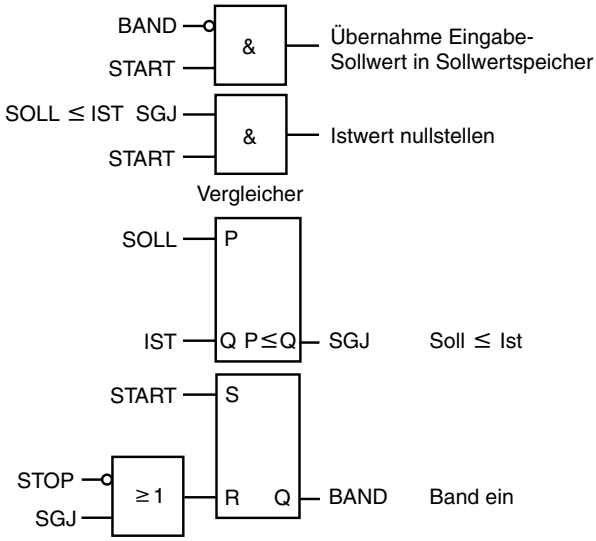
Assemblerprogramm Impulszähler:

```
;***********************************************************************
;Impulszähler                                          Datei IZAHL.ASM
;Entprellt den Eingangsimpuls, erkennt positive Flanke und zählt Vor
;***********************************************************************
;Variable: EIMP    = Eingangsimpuls
;          EIMPE   = Eingangsimpuls entprellt
;          ZEITZ1  = 2x8 Bit Zeitzähler für Entprellzeit
;         +ZEITZ2
;          EIMPE_A = Eingangsimpuls entprellt Alt
;                    (vom vorhergehenden Zyklus)
;          IST     = Istwertzähler
;-----------------------------------------------------------------------
;Title "Impulszähler, Datei IZAHL"

izahl:     lcall entpre         ;Eingangsimpuls entprellen
           lcall flzahl         ;Flanke zählen
           ret
;------------------ entprellen des Eingangsimpulses ------------------
;
entpre:    jnb eimp,en1         ;Eingangsimpuls = 1? Nein:
           djnz zeitz1,en2      ;Ja: Zeitzähler - 1
           djnz zeitz2,en2
           setb eimpe           ;Zeitzähler = 0: Eimp entprellt = 1
           ljmp en2
en1:       mov zeitz1,#0ffh     ;Zeitzähler setzen
           mov zeitz2,#10h
           clr eimpe            ;Eingangsimpuls entprellt = 0
en2:       ret
;
;----- Erkennung positive Flanke und Impulszähler inkrementieren -----
;
flzahl:   jnb eimpe,fl1         ;Eingangsimpuls entprellt = 1? Nein:
          jb eimpe_a,fl1        ;Ja: Impuls entprellt Alt = 0? Nein:
```

```
          inc ist               ;Ja: Istwertzähler inkrementieren
fl1:      mov c,eimpe           ;Eimp entprellt = Eimpe Alt
          mov eimpe_a,c
          ret
;***************************************************************************
```

Programmbaustein Verknüpfungssteuerung VSTEU

Lösungsstrategie als Funktionsplan:



Assemblerprogramm:

```
;***************************************************************************
;Verknüpfungssteuerung                                    Datei VSTEU.ASM
;***************************************************************************
;Variable:    START
;             STOP         Low-aktiv, Öffner
;             BAND       = Bandmotor Ein
;             Soll       = Sollwert
;             IST        = Istwert
;             ESOLL      = Eingabe-Sollwert als Dualzahl
;             SGI        = Soll =< Ist
;---------------------------------------------------------------------------
;Title "Verknüpfungssteuerung, Datei VSTEU"

vsteu:  mov c,start               ;Sollwert speichern?
        anl c,/band
        jnc vs1                   ;Nein
        mov soll,esoll            ;Ja
vs1:    mov c,sgi                 ;Istwert nullstellen?
        anl c,start
        jnc vs2                   ;Nein
        mov ist,#00h              ;Ja
vs2:    clr c                     ;Soll =< Ist?
```

```
        mov a,ist                   ;(Ist - Soll: wenn Soll=<Ist, C=0)
        subb a,soll
        cpl c                       ;Ja: SGI=1        Nein: SGI=0
        mov sgi,c
        jnb start,vs3               ;Band Ein setzen? Nein:
        setb band                   ;Ja
vs3:    mov c,stop                  ;Band Ein rücksetzen?
        cpl c
        orl c,sgi
        jnc vs4                     ;Nein
        clr band                    ;Ja
vs4:    ret
;*********************************************************************
```

Hauptprogramm Zählersteuerung ZSTEU1

Assemblerprogramm:

```
;*********************************************************************
;Zählersteuerung 1                                    Datei ZSTEU1.ASM
;
;Eingangssignale: Sollwert BCD, 2 Dekaden       Port P1
;            Eingangsimpuls     Port P3.2
;            Start              Port P3.4
;            Stop/N             Port P3.5
;Ausgangssignale: Istwert BCD, 2 Dekaden        Port P5
;            Band Ein           Port 4.4
;*********************************************************************
Title "Zählersteuerung 1, Datei ZSTEU1.ASM"
$nomod51
$include(t89c51ac2.INC)
;
;------- Zuordnung: symbolische Variable zu absoluten Adressen ------
;
ein1      equ     20h       ;Eingangssignalabbild Port 1
ein3      equ     21h       ;Eingangssignalabbild Port 3
aus0      equ     22h       ;Ausgangssignalabbild Port 0
aus2      equ     23h       ;Ausgangssignalabbild Port 2
im        equ     24h       ;interne Merker
esoll     equ     25h       ;Eingabe-Sollwert Dual
soll      equ     26h       ;Sollwert
ist       equ     27h       ;Istwert (Impulszähler)
zeitz1    equ     28h       ;Zeitzähler 1
zeitz2    equ     29h       ;Zeitzähler 2
eimpe     equ     im.0      ;Eingangsimpuls entprellt
eimpe_a   equ     im.1      ;Eingangsimpuls entprellt Alt
sgi       equ     im.2      ;Soll =< Ist
eimp      equ     ein3.2    ;Eingangsimpuls
start     equ     ein3.4    ;Start
stop      equ     ein3.5    ;Stop/N
band      equ     aus0.4    ;Band Ein
;
```

```
;------------------------- Initialisierung --------------------------
;
mov p1,#0ffh
mov p3,#0ffh
mov ein1,#00h
mov ein3,#00h
mov aus0,#00h
mov aus2,#00h
mov im,#00h
mov zeitz1,#0ffh
mov zeitz2,#10h
mov soll,#00h
mov esoll,#00h
mov ist,#00h
;
;-------------------------- Hauptprogramm ---------------------------
;
zs1: mov ein1,p1             ;Eingangssignale einlesen
     mov ein3,p3
     mov a,ein1              ;Sollwert BCD für Wandler
     lcall deduwan           ;Wandler BCD in Dual
     mov esoll,a             ;Eingabe-Sollwert Dual
     lcall vsteu             ;Verknüpfungssteuerung
     lcall izahl             ;Impulszähler
     mov a,ist               ;Istwert Impulszähler für Wandler
     lcall dudewan           ;Wandler Dual in BCD
     mov aus2,a              ;Istwert BCD für Anzeige
     mov p0,aus0             ;Ausgangssignale ausgeben
     mov p2,aus2
     ljmp zs1                ;zyklische Bearbeitung
;
;----------------- Bereitstellen der Unterprogramme -----------------
;
$include (deduwan.asm)
$include (vsteu.asm)
$include (izahl.asm)
$include (dudewan.asm)
;
;*********************************************************************
END
```

Übung 16.5

Ändern Sie die Zählersteuerung wie folgt:

1. Die Ein- und Ausgabe erfolgt im Dualcode.
2. Die Stop-Taste entfällt; es muss immer die eingestellte Stückzahl abgepackt werden.
3. Auf der Ist-Anzeige wird der „Restwert“, also die noch fehlenden Päckchen, angezeigt.

17 Programmierung von Controller-Erweiterungen in Assembler

In diesem Kapitel wird die Ansteuerung der integrierten Funktionseinheiten in Assembler beschrieben. Die jeweilige genauere Funktion kann aus dem Kapitel zur C-Programmierung entnommen werden.

17.1 Der Zähler/Zeitgeber Timer 0 und 1

Initialisieren Sie den Timer 1 als Zeitgeber. Es soll eine Zeit für ein Blinklicht erzeugt werden. Timer 0 ist nicht zu beeinflussen.

Timer-Modus einstellen:

TMOD nicht bitadressierbar

GATE	C/$\overline{T}$	M1	M0				
0	0	0	1	x	x	x	x

Assemblerprogramm:

```
INIT_T1:    ANL TMOD,#0Fh; 16-Bit
            ORL TMOD,#10h
            RET
```

Der Timer soll vom laufenden Programm auf einen Zeitwert gesetzt und auf Überlauf abgefragt werden. Nach jedem Zeitabschnitt wird das Blinklicht an P2.0 umgeschaltet.

Assemblerprogramm:

```
BLINK:
    JNB TF1,BL1;Timer Überlauf?
    CLR TR1; Ja: Timer stoppen
    MOV TL1,#ZL1; Timer setzen
    MOV TH1,#ZH1
    CLR TF1
      ; Überlauf rücksetzen
    CPL P5.0
      ; Blinklicht umschalten
    SETB TR1; Timer starten
BL1: RET
```

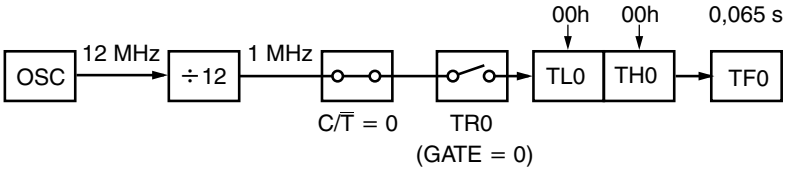
Übung 17.1

Initialisieren Sie Timer 0 als 16-Bit-Zähler. Es sollen externe Impulse gezählt werden. Die Eingangsimpulse vom Geber sind über eine monostabile Kippstufe entprellt. Über P3.2 soll sich der Zähler auf null stellen lassen. Zeigen Sie den Zählerstand auf Port 1 und 2 an.

17.1.1 Anwendung als Zeitgeber

Timer 0 soll als Zeitgeber für ein Lauflicht eingesetzt werden. Die Zeiten sollen einstellbar sein und in einem Bereich liegen, der mit den Augen erkennbar ist.

Die Basiszeit liefert der 16-Bit-Timer 0.



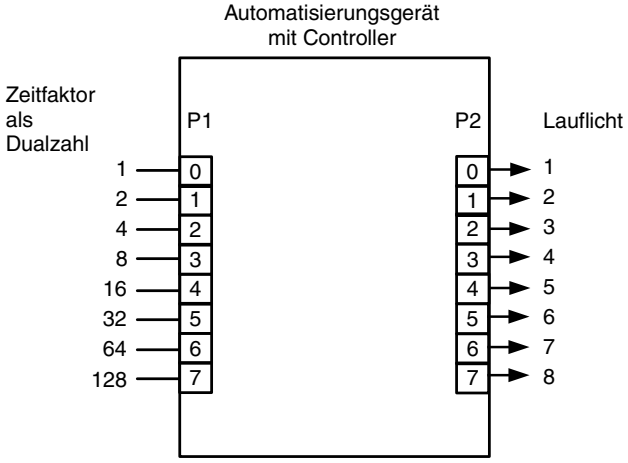
Bei Laden des Timers mit der Zahl 0000h ergibt sich bis zum Überlauf eine Zeit von 0,065 53 s, wenn der Controller mit einem 12-MHz-Quarz arbeitet.

Dieser Basiswert wird mit einem einstellbaren Zeitfaktor multipliziert.

$$\text{Zeit} = \text{Zeitfaktor} \times \text{Basiswert}$$

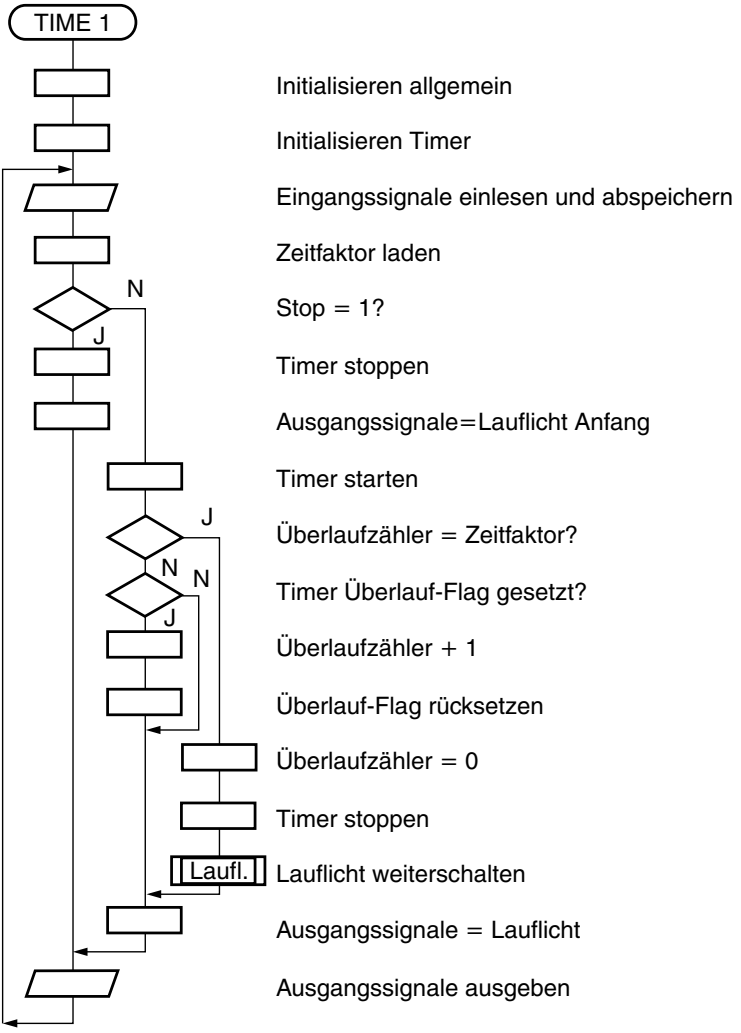
Dazu müssen die Timer-Überläufe gezählt werden. Stimmt der Zählerstand des Überlaufzählers mit dem eingestellten Zeitfaktor überein, ist die Zeit erreicht, und das Lauflicht schaltet um eine Stelle weiter.

Anschlussplan:



Ist das Stop-Signal = 1, soll die laufende Zeit unterbrochen werden und das Lauflicht in die Grundstellung gehen. Bei Stop = 0 setzt das Lauflicht die unterbrochene Zeit und Bewegung fort.

Programmablaufplan:



Assemblerprogramm:

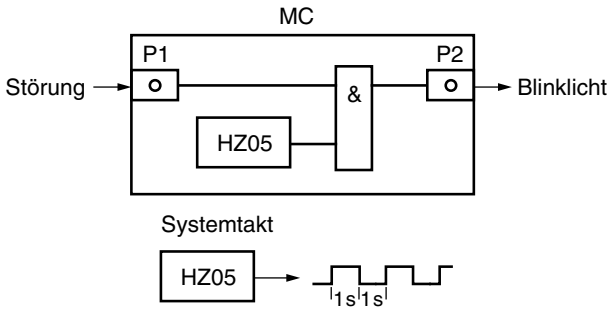
```
;********************** Timer als Zeitgeber ************************
;
;Timer 0 als Zeitgeber mit einstellbarer Zeit.
;Der Zeitfaktor wird auf Port 1 eingestellt.
;Nach abgelaufener Zeit soll ein Lauflicht an Port 2 um eine
```

```
;Stelle weiter schalten.
;Mit einem Stop-Signal an Port 3.2 ist das Lauflicht anzuhalten.
;
;Eingabe: Port 1
;         Bit 0 bis Bit 7: einstellbarer Zeitfaktor als Dualzahl
;         Port3
;         Bit 2: Stop
;                  Bei 1-Signal soll der Zeitgeber angehalten werden
;                  und das Lauflicht in die Ausgangsstellung gehen.
;                  Bei anschließendem 0-Signal soll das Lauflicht bei
;                  der unterbrochenen Stellung weiterarbeiten.
;Ausgabe: Port 2
;         Bit 0 bis Bit 7: Lauflicht
;
;*********************************************************************
;Title "Timer als Zeitgeber, Datei TIME1.ASM"
$nomod51
$include(t89c51ac2.INC)
;
;--------------------------- Zuordnungen ----------------------------
;
ein1 equ 20h                    ;Eingangssignalabbild Port 1
ein3 equ 21h                    ;Eingangssignalabbild Port 3
aus2 equ 22h                    ;Ausgangssignalabbild Port 2
zf equ 23h                      ;Zeitfaktor
uz equ 24h                      ;Überlaufzähler
lauf equ 25h                    ;Muster Lauflicht
stop equ ein3.2                 ;Schalter "Stop"
;
;--------------------- Initialisieren allgemein ---------------------
;
mov p1,0ffh
mov p3,0ffh
mov uz,#00h
mov lauf,#01h                   ;Lauflicht Anfang
;
;---------------------- Initialisieren Timer0 -----------------------
;
anl tmod,#0f0h                  ;16 Bit Timer:
orl tmod,#01h                   ;Gate=0, C/T=0, M1=0, M0=1
mov tl0,#00h                    ;Timer nullstellen
mov th0,00h
;
;------------------------- Hauptprogramm ----------------------------
;
ti5:   mov ein1,p1              ;Eingangssignale einlesen
       mov ein3,p3
       mov zf,ein1              ;Zeitfaktor nach ZF
       jnb stop,ti1             ;Stop = 1? Nein: nach ti1
       clr tr0                  ;Ja: Timer stoppen
       mov aus2,#01h            ;Ausgangssignale = Lauflicht Anfang
       ljmp ti2
```

```
ti1:   setb tr0                 ;Timer starten
       mov a,zf                 ;Überlaufzähler = Zeitfaktor?
       cjne a,uz,ti3            ;Nein: nach ti3
       mov uz,#00h              ;Ja: Überlaufzähler nullstellen
       clr tr0                  ;Timer stoppen
       lcall laufli             ;Lauflicht weiterschalten
       ljmp ti4
ti3:   jnb tf0,ti2              ;Timer Überlauf? Nein: nach ti2
       inc uz                   ;Ja: Überlaufzähler + 1
       clr tf0                  ;Überlauf-Flag nullstellen
ti4:   mov aus2,lauf            ;Ausgangssignale = Lauflicht
ti2:   mov p2,aus2              ;Ausgangssignale ausgeben
       ljmp ti5                 ;zyklische Programmbearbeitung
;
;-------------------- Unterprogramm Lauflicht -----------------------
;
laufli: mov a,lauf
        rl a                    ;Lauflicht 1 Stelle weiter
        mov lauf,a
        ret
;
;************************************************************************
END
```

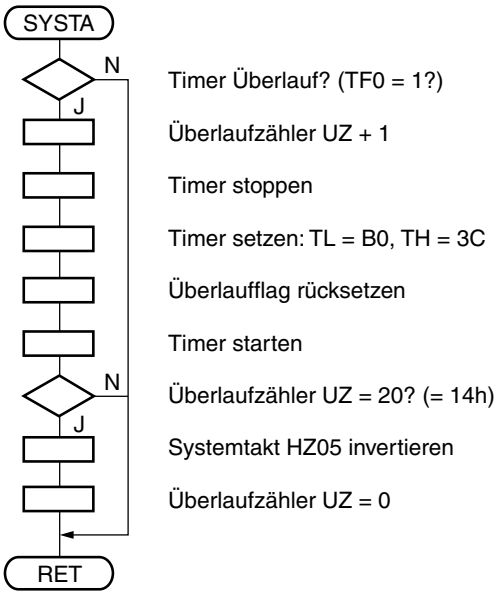
Beispiel 17.1

Als Anwendung soll, wie im C-Teil, ein Blinklicht von 0,5 Hz an Port P2.0 bei einer eintreffenden Störung am Port P1.0 erscheinen.



Um eine Zeit von 50ms zu erhalten, muss der Timer auf die Dezimalzahl 15 536 dezimal oder 3C B0 hex gesetzt werden. Die 50 ms sind mit 20 zu multiplizieren, um auf eine Sekunde zu kommen. Dazu wird der Überlaufzähler mit dem symbolischen Namen UZ vom Überlaufflag TF0 von 0 bis 20 hochgezählt. Ist die 20 erreicht, wird das Bit HZ05 invertiert.

Programmablaufplan:



Assemblerprogramm:

```
;***********************************************************************
;Unterprogramm SYSTA.ASM zur Erzeugung eines Systemtaktes von 0,5 Hz
;an Bit HZ05.
;Das Unterprogramm benötigt   1 Merkerbit, Name HZ05
;                             1 Merkerbyte, Name UZ
;Initialisierung des Timers 0 als 16-Bit-Timer
;***********************************************************************
systa: jnb tf0,sy1
       inc uz
       clr tr0
       mov tl0,#0b0h
       mov th0,#3dh
       clr tf0
       setb tr0
       mov a,uz
       cjne a,#14h,sy1
       cpl hz05
       mov uz,#00h
sy1:   ret
```

Übung 17.2

Schreiben Sie das Hauptprogramm für die in Beispiel 17.1 aufgeführte Anwendung des Systemtaktes HZ05 zur Ansteuerung eines Blinklichtes an Ausgabeport P2.0, wenn am Eingabeport P1.0 ein Signal „Störung“ anliegt.

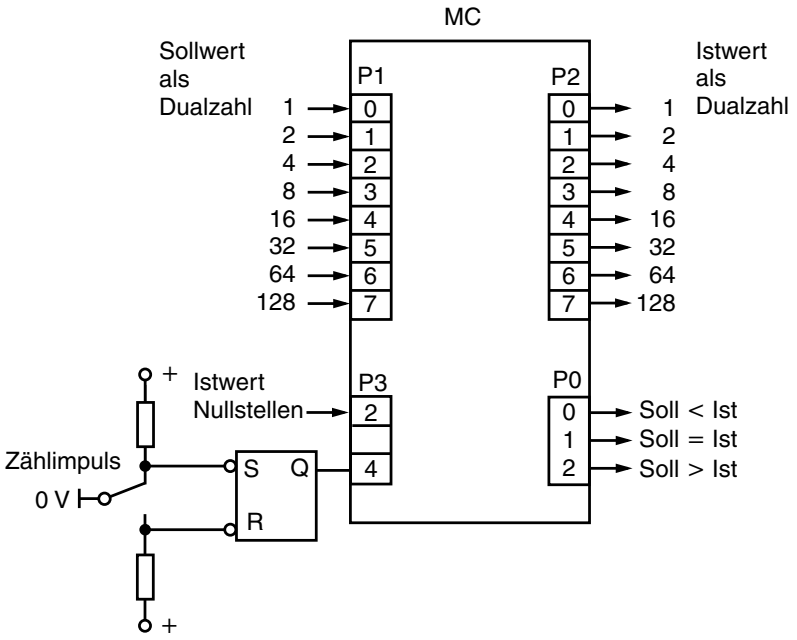
Denken Sie an die Zuweisungen und die Initialisierung für das benötigte Unterprogramm SYSTA.

17.1.2 Anwendung als Ereigniszähler

Bei einer digitalen Regelung wird ein eingestellter Sollwert mit einem Istwert verglichen. Der Istwert wird digital gebildet, indem eintreffende Zählimpulse von einem Geber einen Vorwärtszähler hochzählen.

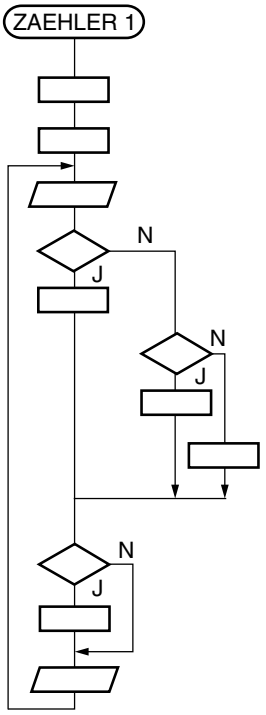
Die digitale Regelung könnte z. B. Teil eine Wegregelung sein, bei der pro Haltepunkt ein Zählimpuls erzeugt wird.

Anschlussplan:



Als Istwertzähler wird Timer 0 als 8-Bit-Reloadzähler eingesetzt. Der Reloadwert ist die Zahl 00h. Der Timer zählt die an P3.4 eintreffenden Impulse unabhängig vom laufenden Programm. Der Istwert des Timers wird vom laufenden Programm gelesen und bearbeitet.

Programmablaufplan:



Initialisieren allgemein

Initialisieren Timer 0

Eingangssignale einlesen und abspeichern

Soll = Ist?

Soll = Ist setzen

Soll < Ist?

Soll < Ist setzen

Soll > Ist setzen

Istwert Nullstellen?

Istwert = 0

Ausgabe Istwert und Ausgangssignale

Assemblerprogramm:

```
;******************** Timer0 als Ereigniszähler *********************
;
;Title "Ereigniszähler, Datei ZAEHLER1.ASM"
$nomod51
$include(t89c51ac2.INC)
ljmp za_anf
;--------------------------- Zuordnungen ----------------------------
ein0 equ 20h                        ;Eingangssignalabbild 0
ein1 equ 21h                        ;Eingangssignalabbild 1
aus0 equ 22h                        ;Ausgangssignalabbild 0
;
;------------------- Initialisierung des Timer0 ---------------------
;
;Der Timer0 wird als 8-Bit-Reload-Counter eingesetzt.
;Der Reloadwert ist 00h.
;
init_t0: anl tmod,#0f6h             ;G=0, M0=0
         orl tmod,#06h              ;C/T=1, M1=1
         setb tr0                   ;Zähler Ein
         setb p3.4                  ;Vorbereitung p3.4 als Eingang
         mov th0,#00h               ;Reloadwert = 00h
         ret
```

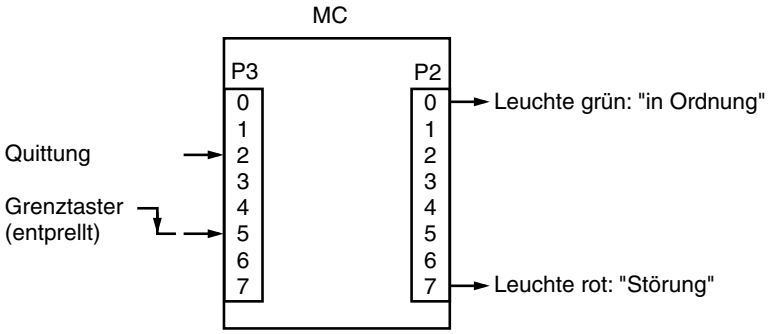
```
;------------------ Initialisierung der Steuerung ------------------
;
za_anf:  mov ein0,#00h              ;Eingangssignalabbild 0 = 0
         mov ein1,#00h              ;Eingangssignalabbild 1 = 0
         mov aus0,#00h              ;Ausgangssignalabbild 0 = 0
         mov p1,#0ffh               ;Eingabeports vorbereiten
         setb p3.2
         lcall init_t0              ;Timer0 initialisieren
;------------------------ Programm --------------------------------
za_5:    mov ein0,p1                ;Eingangssignale abspeichern
         mov ein1,p3
         mov a,ein0                 ;Soll - Ist
         subb a,tl0
         jz za_1                    ;Soll = Ist
         jc za_2                    ;Soll < Ist
         setb aus0.2                ;Ausgabebit S>I auf 1
         ljmp za_3
za_1:    setb aus0.1                ;Ausgabebit S=I auf 1
         ljmp za_3
za_2:    setb aus0.0                ;Ausgabebit S<I auf 1
za_3:    mov c,ein1.2               ;Istwert 0-stellen?
         jnc za_4                   ;Nein
         mov tl0,#00h               ;Istwert auf 00h stellen
za_4:    mov p2,tl0                 ;Ausgabe Istwert auf Port 2
         mov p0,aus0                ;Ausgabe Steuersignale auf Port 0
         ljmp za_5                  ;zyklische Programmbearbeitung
END
```

Übung 17.3

Schreiben Sie ein Assemblerprogramm für folgende Aufgabenstellung:

Ein Grenztaster liefert Eingangsimpulse. Wird er innerhalb von fünf Sekunden mehr als dreimal betätigt, liegt eine Störung vor. Im Normalbetrieb leuchtet eine grüne Lampe, im Störfall eine rote. Die Störung muss mit einer Taste quittiert werden.

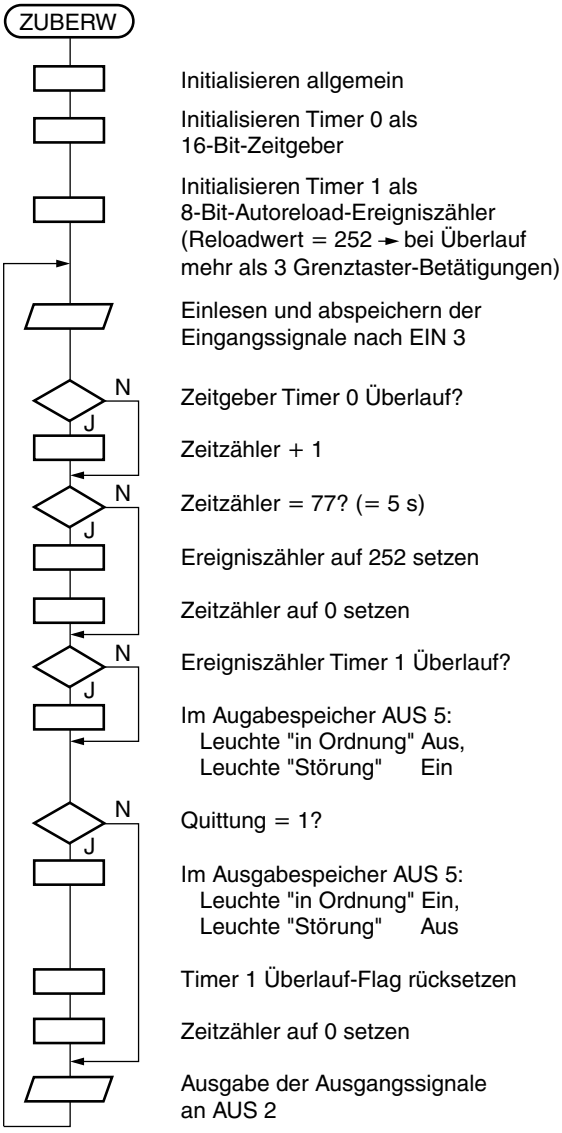
Anschlussplan:



Als Zeitgeber wird Timer 0 verwendet. Die Basiszeit bis zum Überlauf beträgt 0,065 s. Die Überläufe sind zu zählen. Hat der Überlaufzähler die Zahl 77 erreicht, sind ca. 5 s vergangen.

Als Ereigniszähler wird Timer 1 eingesetzt. Hat der Ereigniszähler innerhalb 5 s mehr als drei Grenztaster-Betätigungen gezählt, liegt eine Störung vor, die angezeigt wird.

Programmablaufplan:



17.2 Der Analog/Digital-Wandler

Hier soll gezeigt werden, wie am Controller AT89C51AC3 eine analoge Spannung binär auf Port 2 dargestellt werden kann.

Dieses Beispiel ist aus dem Kapitel 9 entnommen und wird hier in Assemblercode realisiert.

Beispiel 17.2

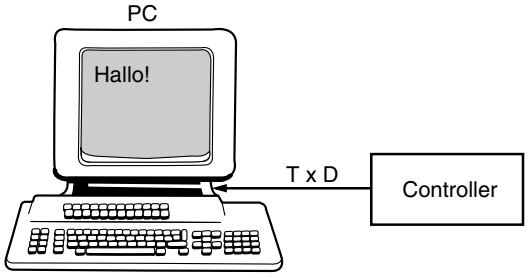
Eine analoge Spannung am Eingang AN2 (P1.3) ist einzulesen, in eine 8-Bit-Zahl zu wandeln und an Port 2 binär auszugeben. Die Spannung variiert im Bereich von 0V bis 3V.

```
;*********************************************************************
;A/D Wandlung. Datei a_d.ASM
;*********************************************************************
;Es wird der Analogwert    an P1.3 ermittelt und
;auf Port 2 dargestellt
;---------------------------------------------------------------------
$nomod51
$include(t89c51ac2.INC)
           mov ADCF,#08h        ;Port P1.3 für die Wandlung freigeben
anf:       mov ADCON,#02Bh      ;Wandler an Port P1.3 starten
warten:    mov A,ADCON          ;Prüfen, ob Wandler fertig ist
           anl A,#010h
           CJNE A,#010h,warten
           mov P2,ADDH          ;Übertragen des gewandelten Wertes an P2
           ajmp anf             ;Endlosschleife
END
```

17.3 Die serielle Schnittstelle

Der Controller soll auf dem Bildschirm eines PCs einen Text ausgeben. Dazu muss der PC über ein Programm als Terminal geschaltet werden.

Im Terminal-Programm des PCs wurde eine Baudrate von 2400 und folgender Übertragungsrahmen eingestellt: 1 Startbit, 8 Datenbits, 1 Stopbit, keine Paritätskontrolle.



Die folgenden Programme geben den Text „Hallo“ auf dem Bildschirm aus.

Die serielle Schnittstelle des Controllers ist durch ein Initialisierungprogramm auf die gleiche Baudrate und den gleichen Übertragungsrahmen wie das Terminal einzustellen.

Es wird der Timer 1 als Baudratengenerator programmiert.

```
Title "Serielle Ausgabe auf dem Bildschirm, Datei ser2.asm"
;***************************************************************************
$nomod51
$include(t89c51ac2.INC)
;--------------------------- Zuweisungen  ----------------------------
end equ 00h                             ;Zeichen "Ende" der Zeichenkette
;------------------------- Initialisieren ----------------------------
;serielle Schnittstelle:
mov scon,#52h                           ;1 Start, 8 Dat, 1 Stop
;Timer 1 als Baudratengenerator 2400Bd:
anl pcon,#7Fh                           ;keine Frequenzteilung (SMOD=0)
clr tr1                                 ;Timer1 stoppen
clr bd                                  ;Timer 1 als Baudratengenerator
anl tmod,#0fh                           ;Timer in Timer-Mod2 = 8Bit
orl tmod,#20h                           ;Auto Reload
mov th1,#0f3h                           ;Reloadwert für Bd2400
mov tl1,#0f3h
setb tr1                                ;Timer 1 freigeben
;----- Programm: Ausgabe einer Zeichenkette auf dem Bildschirm -------
             mov r2,#00h                ;Zeichenzeiger 0-Stellen
             mov dptr,#zk0              ;Basisadresse Zeichenkette
bi3:         mov a,r2                   ;Zeichen laden
             movc a,@a+dptr
             cjne a,#end,bi1            ;Zeichen "Ende" ?   Nein:
             ljmp bi2                   ;Ja:
bi1:         lcall seraus1              ;Zeichen seriell ausgeben
             inc r2                     ;nächstes Zeichen
             ljmp bi3
bi2:         ljmp bi2                   ;Endlosschleife
;------------------ Unterprogramm serielle Ausgabe -------------------
seraus1:     jnb ti,seraus1             ;warten, bis Senderegister frei
             clr ti                     ;Flag "Sendereg. beschrieben"
                                        ;setzen
             mov sbuf,a                 ;Byte nach Sendereg. und Senden
                                        ;starten
             ret
;---------------- auszugebende Zeichenkette: "Hallo!" ----------------
zk0:         db 0ch                     ;Bildschirm löschen
             text "Hallo!"              ;Text
             db 00h                     ;Zeichen "Ende"
;***************************************************************************
END
```

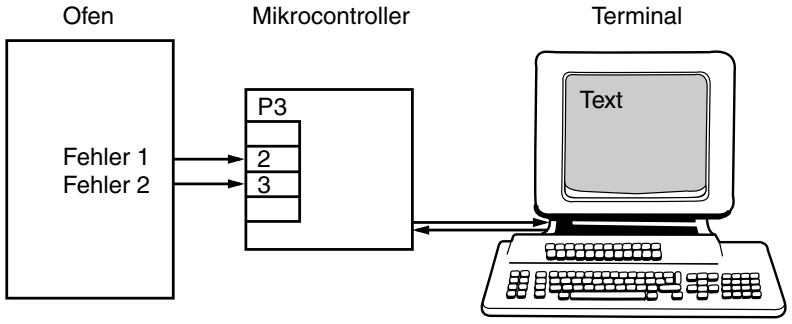
Übung 17.4

Erweitern Sie das Programm SER2.ASM zur Ausgabe einer Meldung auf dem Bildschirm wie folgt: Die Meldung soll immer dann ausgegeben werden, wenn auf der Tastatur des PCs der Buchstabe „S“ oder „s“ gedrückt wird. (S = 01010011; s = 01110011)

Übung 17.5

Ofenüberwachung mit Mikrocontroller

Bei einem Brennofen werden verschiedene Kriterien überwacht. Tritt ein Fehler auf, soll dieser in einer Steuerwarte auf einem Bildschirm angezeigt werden.



Das Terminal ist über eine serielle Schnittstelle mit dem Mikrocontroller verbunden.

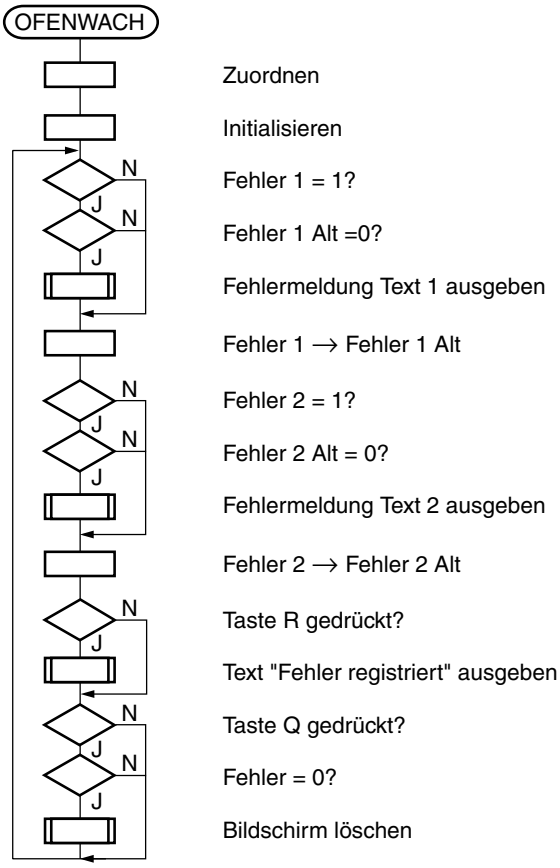
Damit die Meldung nur einmal auf dem Bildschirm erscheint, wird sie bei der positiven Flanke des Fehlersignals am Eingang des Controllers ausgegeben.

Meldung 1: „Brennstoffzufuhr gestört“
Meldung 2: „Ofentemperatur zu hoch“

Durch Drücken der Taste „R“ auf der Tastatur des Terminals soll die Meldung „Fehler registriert“ auf dem Bildschirm erscheinen.

Ist der Fehler beseitigt und steht kein Fehlersignal mehr an (Fehlersignal = 0), soll die Fehlermeldung auf dem Bildschirm durch Drücken der Taste „Q“ auf dem Terminal gelöscht werden.

Schreiben Sie das Programm in Anlehnung an das folgende Ablaufdiagramm.



17.4 Das Interrupt-System

17.4.1 Anwendung mit Ereignis-Interrupt

Motorsteuerung mit Not-Aus an Interrupt-Eingang 0

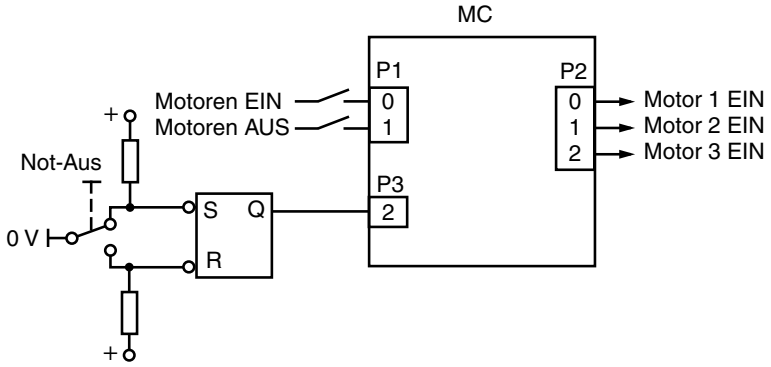
An Port P3.2 ist ein Not-Aus-Taster angeschlossen, der bei Betätigung sofort drei Motoren abschalten soll. Die Motoren sind an Port P2.0, P2.1 und P2.2 angeschlossen. Das Ausgangssignal-Abbild für Port 2 liegt in der symbolischen Adresse AUS2. Im Ausgangssignal-Abbild sollen die Bits für die Motoren ebenfalls auf null gesetzt werden.

Der Interrupt 0 wird an Port P3.2 ausgelöst. Er setzt das Request-Flag IE0. Die Vektoradresse für Interrupt 0 ist 0003h. An dieser Adresse steht der Sprung in die Interrupt-Service-Routine

NOT_AUS. Der Interrupt soll durch die negative Flanke an P3.2 ausgelöst werden. Das ankommende Not-Aus-Signal ist vor dem Portpin über ein Flipflop entprellt.

Um den Interrupt zu testen soll ein einfaches Hauptprogramm zur Motorsteuerung laufen. Die Motoren sind über je eine Taste ein- bzw. auszuschalten. Dieses Hauptprogramm wird durch den Interrupt unterbrochen.

Anschlussplan:



Assemblerprogramm:

```
;***********************************************************************
;Motorsteuerung mit Not-Aus an Interrupt 0     Datei INT0.ASM
;***********************************************************************
;
;---------------- Hauptprogramm Motorsteuerung ----------------------
;
;------------------------ Zuordnungen -------------------------------
$nomod51
$include(t89c51ac2.INC)                          ;Adresse Programm Anfang
ein1 equ 20h                        ;Eingangssignalabbild
m_ein equ ein1.0                    ;Motoren Ein
m_aus equ ein1.1                    ;Motoren Aus
aus2 equ 21h                        ;Ausgangssignalabbild
;---------------------- Initialisierung -----------------------------
;Hauptprogramm:
mov p1,#0ffh
mov ein1,#00h
mov aus2,#00h
;Interrupt 0:
lcall not_aus_init
;------------------------ Programm ---------------------------------
;
anf:mov ein1,p1                     ;einlesen Eingangssignalabbild
    jnb m_ein,ms1                   ;Motoren Ein?    Nein:
    orl aus2,#07h                   ;Ja:
ms1:jnb m_aus,ms2                   ;Motoren Aus?    Nein:
    anl aus2,#0f8h                  ;Ja:
ms2:mov p2,aus2                     ;Ausgabe Ausgangssignalabbild
```

```
    ljmp anf                        ;zyklische Bearbeitung
;*********************************************************************
;
;-------------- Interrupt-Steuerung für Not-Aus ---------------------
;
;Initialisieren Interrupt 0:
;
not_aus_init:setb it0               ;Auslösung durch fallende Flanke
             setb ex0               ;Freigabe INT0
             setb ea                ;Freigabe aller Interrupts
             orl iph0,#01h          ;höchste Priorität
             orl ipl0,#01h
             ret
;
;Interrupt-Service-Routine: NOT_AUS
;an Vektoradresse 0003h steht der Befehl ljmp 8003h
org 8003h                           ;Anfangsadresse Service-Routine
;
             anl aus2,#0f8h         ;Motoren Aus
             mov p2,aus2            ;Ausgabe Ausgangssignalabbild
             clr ie0                ;Interrupt-Flag rücksetzen
             reti
;*********************************************************************
END
```

17.4.2 Anwendung mit Zeit-Interrupt

Ein laufendes Programm soll durch einen Interrupt in einem festen Zeitrhythmus unterbrochen werden, um eine bestimmte Reaktion auszulösen.

Dieses Zeit-Interrupt-Programm ließe sich z. B. für einen Datenlogger verwenden. Der Datenlogger speichert z. B. im Minutenraster alle Temperaturwerte, die bei einem Brennofen während eines Brennvorganges anfallen. Die gespeicherten Werte könnten dann über die serielle Schnittstelle in einen PC übertragen und ausgewertet werden.

Die Zeit ist dabei unabhängig vom laufenden Automatisierungsprogramm und dessen Struktur.

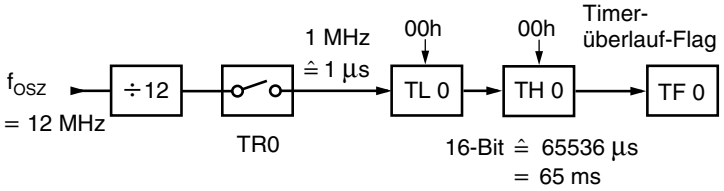
Um das zu zeigen, soll in folgendem Beispiel das normale Programm ein Lauflicht im Sekundenbereich weiterschalten. Die Zeit für das Weiterschalten wird durch eine 3-Register-Zeitschleife realisiert. Diese Zeitschleife lässt sich normalerweise nicht unterbrechen. Die einzige Möglichkeit zur Unterbrechung ist ein Interrupt.

Der Zeitinterrupt soll das Lauflicht alle 2 s in die Grundstellung bringen.

Programm

Programm für Zeitinterrupt:

Zur Erzeugung des Zeitinterrupts soll Timer 0 verwendet werden.



Der Timer 0 wird als 16-Bit-Timer initialisiert und mit dem Wert 0000h geladen.

Ein zusätzlicher 8-Bit-Zähler muss die erzeugten Interrupts, die im Abstand von 65 ms kommen, zählen. 32 Interrupts ergeben ca. 2 s.

Gesamtprogramm:

```
;***************************************************************************
;Lauflicht mit Zeitinterrupt                              Datei INTT0.ASM
;***************************************************************************
;Gesteuert über den Timer0 wird alle 2s ein Interrupt ausgelöst.
;Der Interrupt bringt das Lauflicht in die Grundstellung.
;---------------------------------------------------------------------------
;
;------------------- Hauptprogramm für Lauflicht ----------------------
;
;-------------------------- Zuordnungen -------------------------------
$nomod51
$include(t89c51ac2.INC)
lauf equ 30h                    ;Muster Lauflicht
intz equ 31h                    ;Interruptzähler
;---------------------- Initialisierung ----------------------------
mov lauf,#01h                   ;Grundstellung Lauflicht
lcall zeit_int_init             ;Initialisierung Zeitinterrupt
;
;-------------------------- Programm ----------------------------------
;
la1:  mov p2,lauf               ;Ausgabe Muster Lauflicht an Port 2
      lcall zeit3               ;Wartezeit
      mov a,lauf                ;Lauflicht eine Stelle weiter
      rl a
      mov lauf,a
      ljmp la1                  ;zyklische Bearbeitung
;------------------- Unterprogramm Wartezeit ------------------------
zeit3:mov r7,#5                 ;Wartezeit
ze3:  mov r6,#0ffh
ze2:  mov r5,#0ffh
ze1:  djnz r5,ze1
      djnz r6,ze2
      djnz r7,ze3
```

```
      ret
;************************************************************************
;
;------------------------ Zeitinterrupt -----------------------------
;
;-------------------- Initialisierung gesamt -------------------------
zeit_int_init:lcall init_t0      ;Initialisieren Timer 0
              lcall init_intr_t0 ;Initialisieren Interrupt-Steuerung
              mov intz,#00h      ;Interruptzähler auf 0
;------------------- Initialisierung Timer 0 -------------------------
init_t0:anl tmod,#11111011b      ;Timer Modus
        anl tmod,#11111100b      ;16-Bit-Timer
        orl tmod,#00000001b
        mov tl0,#00h             ;Timer laden
        mov th0,#00h
        setb tr0                 ;Timer starten
        ret
;-------------- Initialisierung Interrupt-Steuerung -----------------
init_intr_t0:setb et0            ;Interrupt Timer0 freigeben
             setb ea             ;alle Interrupts freigeben
             orl iph0,#02h       ;höchste Priorität
             orl ipl0,#02h
             ret
;---------------------Interrupt-Service_Routine----------------------
;in Vektoradresse 000bh steht der Befehl ljmp 800bh
org  800bh                       ;Anfangsadresse Service-Routine
     inc intz                    ;Interruptzähler + 1
     mov a,intz
     cjne a,#32,is1              ;Zeit = 2s? Nein:
     lcall aktion                ;Ja: Aktion veranlassen
     mov intz,#00h               ;Interruptzähler = 0
is1: reti
;
;------------------- Unterprogramm "Aktion" -------------------------
aktion:mov p2,#01h                ;Lauflicht in Grundstellung
       ret
;************************************************************************
END
```

18 Lösungen zu den Übungsaufgaben

Zu Übung 1.1

1. Adressbus z. B. A0 bis A15
   Datenbus z. B. D0 bis D7
   Steuerbus z. B. RD, WR, PSEN (alle Low-aktiv)
2. Im Programm-Counter steht die als nächstes auszugebende Adresse. Der PC enthält das High-Byte PCH und das Low-Byte PCL.
3. Adressbereich 64 K, Adressen von 0000h bis FFFFh.
4. 1 Sendeleitung TxD
   1 Empfangsleitung RxD
   1 Bezugspotenzial 0 V
   = 3 Leitungen für Duplexbetrieb
5. Die Baudrate gibt an, mit wie viel Bits pro Sekunde Daten seriell übertragen werden.
6. z. B. `MOV 20,30 = 1000 0101 0001 0100 0001 1110 = 3 Byte`
   `MOV A,R1 = 1110 1001 = 1 Byte`
7. Das erste Byte enthält den Operationscode.
8. Das Programm ab Adresse 0000h ist in einem EPROM gespeichert. Das Programm bleibt beim Ausschalten der Stromversorgung erhalten.
9. Variable Daten werden im RAM gespeichert. Die Daten im RAM lassen sich einschreiben und auslesen.

Zu Übung 1.2

1. Assemblerbefehle:

```
MOV R0,E8     2 Byte, 2 Zyklen
MOV A, 90     2 Byte, 1 Zyklus
ADD A, R0     1 Byte, 1 Zyklus
ADD A, #10    2 Byte, 1 Zyklus
MOV F0, A     2 Byte, 1 Zyklus
```

2.

Adresse	Inhalt	Befehl
0000	A8	MOV R0,
0001	E8	E8
0002	E5	MOV A,
0003	90	90
0004	28	ADD A, R0
0005	24	ADD A,#
0006	10	10
0007	F5	MOV dadr,A
0008	F0	F0

Zu Übung 2.1

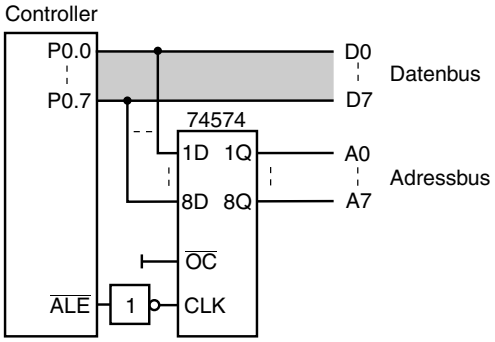
1. Port 0.0 bis 0.7: 1. Adressbus A0 bis A7
   2. Datenbus D0 bis D7
   Es erfolgt eine interne Umschaltung zwischen der Ausgabe Adressbus und Datenbus
   Port 2.0 bis 2.7: Adressbus A8 bis A15
2. Portpin 4.0 und Portpin 4.1
3. als Alternativfunktion: Port 3.0 = serieller Datenempfang RxD
   Port 3.1 = serielle Datenausgabe TxD
4. Port 3.5 = Timer 1 Eingangsimpuls T1
5. 8 Eingänge (P0.0 bis P0.7); maximal 3 V Referenzspannung

Zu Übung 2.2

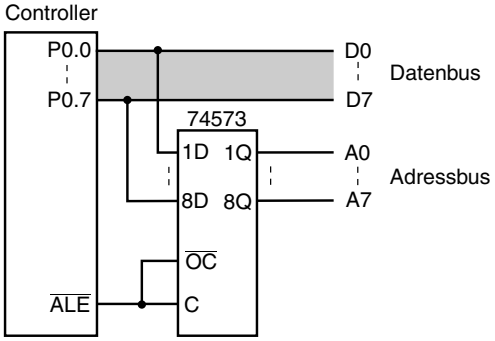
1. Bei dem Gehäuse mit mehr Pins wird zusätzlich noch der Portpin P4.2, P4.3 und P4.4 nach außen geführt. In diesen Gehäusen steht dann das SPI-Interface zusätzlich zur Verfügung.
2. Timer 0, Timer 1, Timer 2, Watchdog, PCA
3. Im 64 KByte Flash-Speicher.
4. Im 2 KByte ERAM-Speicher.

Zu Übung 3.1

Schaltung mit positiv flankengesteuertem 8-D-Flipflop 74574:

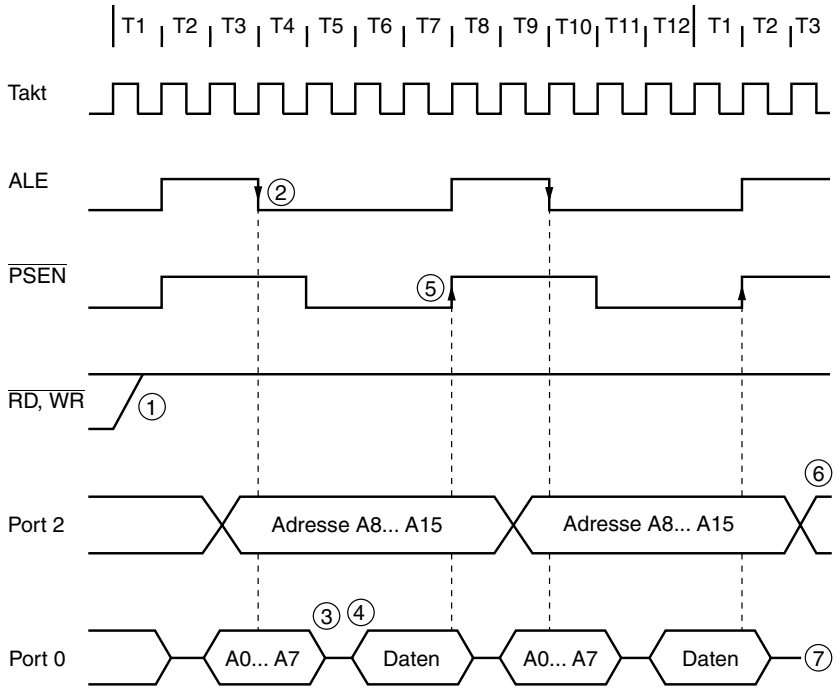


Schaltung mit zustandsgesteuertem 8-D-Flipflop 74573:



Zu Übung 3.2

Speicherzugriff auf externen Programmspeicher:

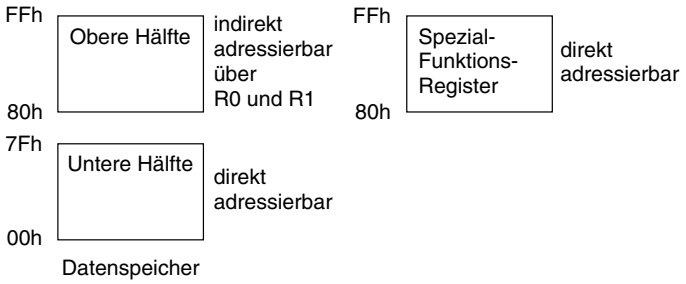


1. $\overline{\text{RD}}$ und $\overline{\text{WR}}$ werden oder bleiben inaktiv.
2. Die fallende Flanke von ALE wird benutzt zur Abspeicherung von Adressen A0 . . . A7 die nun an Port 0 anliegen. Die Adressen A8 . . . A15 stehen an Port 2 zur Verfügung.
3. Mit Beginn von $\overline{\text{PSEN}}$ wird der Port 0 hochohmig.
4. Der Speicher bringt nun mit dem Lesesignal $\overline{\text{PSEN}}$ seine Daten auf den Bus.
5. Mit der steigenden Flanke von $\overline{\text{PSEN}}$ wird der Bus wieder hochohmig.
6. Es erfolgt der nächste Zugriff auf den Programmspeicher.
7. Nach Ende eines Zugriffes enthält das Ausgaberegister von Port 0 FFH, der Bus ist hochohmig.

Der Takt entspricht direkt der Quarz-Frequenz. Innerhalb von 6 Taktzyklen greift der Prozessor einmal auf den Programmspeicher zu.

Zu Übung 4.1

1.



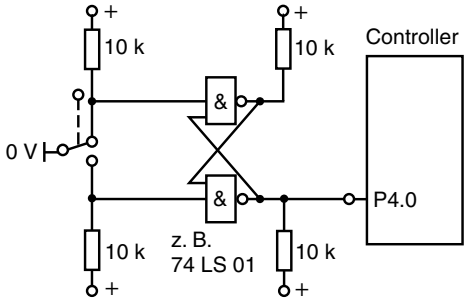
2. Es gibt vier Registerbänke zu je 8 Registern. Die Register heißen bei allen Registerbänken R0 bis R7.

   Adressen: RB0 von 0 bis 7
   RB1 von 8 bis 15
   RB2 von 16 bis 23
   RB3 von 24 bis 32

   RB0 und RB1 werden als Zeiger zur indirekten Adressierung verwendet. Alle Register lassen sich als Speicher einsetzen.
   Die aktuelle Registerbank lässt sich über Bit 3 und 4 im Programmstatuswort einschalten. Bei der Umschaltung der Registerbänke bleibt deren Inhalt erhalten.
   Beim Sprung in ein Unterprogramm können diesem durch Umschalten der Bank eigene Register zugeordnet werden.
3. Es lassen sich 16×8 Bit speichern. Diese 128 Bit haben die Byte-Adressen 00h bis 7Fh. Der Bitspeicher geht von Byte-Adresse 20h bis 2Fh.
4. Der direkt adressierbare Speicher geht von Adresse 00h bis 7Fh. Zieht man davon die Registerbänke und den Bitspeicher ab, bleiben die Adressen 30h 7Fh als Bytespeicher übrig. Das sind 80 Byte.
5. Die Spezial-Funktions-Register gehen von Adresse 80h bis FFh. Sie sind direkt adressierbar.
6. Die obere Hälfte des internen RAM-Speichers geht von 80h bis FFh. Diese 128 Bytes sind indirekt adressierbar mithilfe der Register R0 oder R1 als Zeiger.

Zu Übung 5.1

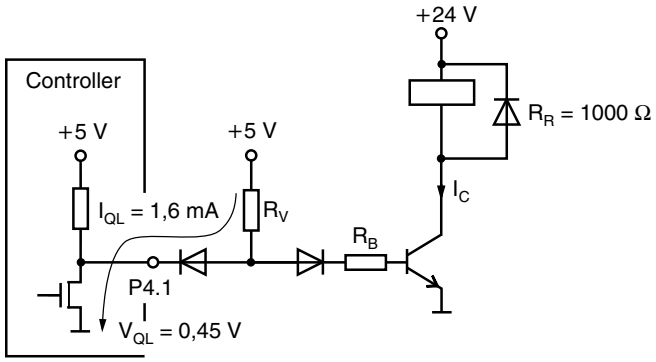
Der Kollektorwiderstand des ansteuernden Bausteins sollte 10 kΩ betragen. Da die internen Kollektorwiderstände der TTL-Bausteine zu niederohmig sind, sind Bausteine mit Open-Collector einzusetzen und ein externer Kollektorwiderstand anzuschließen.



Das Port-Latch ist auf 1-Signal zu setzen. Um sicher zu gehen werden deshalb alle Port-Latches bei einem Reset automatisch auf 1 gesetzt.

Zu Übung 5.2

Aufgrund der Außenbeschaltung kann am Controller-Pin nur ein Strom fließen, wenn er ein 0-Signal ausgibt. Bei 1-Signal wird der Strom von der externen Diode abgeblockt.



$$R_V = \frac{+5\,V - U_D - V_Q}{I_{QL}} = \frac{5\,V - 0{,}7\,V - 0{,}45\,V}{1{,}6\,mA}$$

$$\underline{\underline{R_V = 2{,}4\,k\Omega}}$$

$$I_C = \frac{24\,V - U_{CE}}{R_R} = \frac{24\,V - 0{,}2\,V}{1000\,\Omega} = 23{,}8\,mA$$

$$I_B = \frac{I_C \cdot 2}{B} = \frac{23{,}8\,mA \cdot 2}{200} = 0{,}238\,mA$$

$$R_B = \frac{5\,V - U_D - U_{BE}}{I_B} - R_V = \frac{5\,V - 0{,}7\,V - 0{,}5\,V}{0{,}238\,mA} - 2{,}4\,k\Omega$$

$$\underline{\underline{R_B = 13{,}56\,k\Omega}}$$

Zu Übung 8.1

Am Ausgang des ULN-Treibers: $I_{L,Ausgang} = 300\,mA$

$\Rightarrow$ Am Eingang des ULN-Treibers wird ein Strom von $I_{L,Eingang} = 450\,\mu A$ benötigt.
(siehe Kennlinienfeld Eingangsstrom I_i zu I_C)

So kann nun der Widerstand berechnet werden:

$$R_{pull\text{-}up} = \frac{5\,V - V_{OH}}{I_L - I_{OH}} = \frac{5\,V - 2{,}5\,V}{450\,mA} = 5{,}56\,k\Omega$$

Es muss jetzt geprüft werden, ob der Strom beim Schalten nicht den maximalen Strom pro Portpin überschreitet.

Laut Datenblatt gelten folgende Einschränkungen (siehe Kapitel 5):
Maximaler I_{OL} pro Portpin 10 mA,
Port 0: insgesamt maximal 26 mA,

Port 1, 2, 3, 4: pro Port max. 15 mA
zusammen I_{OL} max. 71 mA.

Überprüfung:

$$I_{OL} = \frac{5\,V}{5{,}56\,k\Omega} = 899\,\mu A$$

⇒ Es fließt also ein unkritischer Strom.

Zu Übung 8.2

```
#include<t89c51ac2.h>       // Einbinden der Controller-Bibliothek

// Den Portpins symbolische Namen zuordnen:

sbit E1=P1^0;               // E1 wird Port P1.0 zugeordnet
sbit E2=P1^1;               // E2 wird Port P1.1 zugeordnet
sbit E3=P1^2;               // E3 wird Port P1.2 zugeordnet
sbit M1=P1^3;               // M1 wird Port P1.3 zugeordnet
sbit A1=P2^0;               // A1 wird Port P2.0 zugeordnet
sbit A2=P2^7;               // A2 wird Port P2.7 zugeordnet

void main (void)            // Hauptprogramm
{
      while(1)              // Endlosschleife für zyklische Programmbearbeitung
      {
            A2=((!E1&&!E2&&M1);    // Logische Funktion für A2
            A1=!(A2||E3);          // Logische Funktion mit Invertierung der Eingänge
                                   // berücksichtigt (siehe Hardwarebeschaltung)
         }
}
```

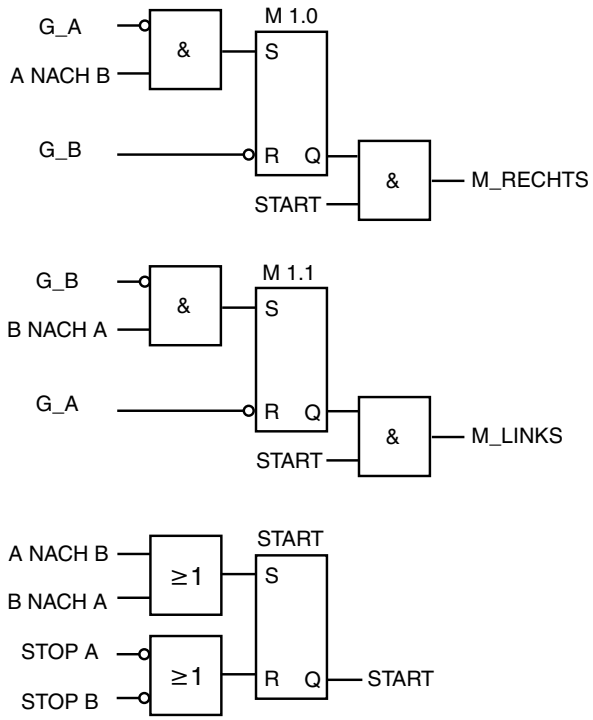
Zu Übung 8.3

```
Set- und Reset-Abfrage werden in der Reihenfolge vertauscht.

#include<t89c51ac2.h>       // Einbinden der Controller-Bibliothek

// Den Symbolische Namen die Adressen der Portpins
// zuordnen:
sbit E1=P1^0;               // SFR-Adresse von Port P1.0
sbit E2=P1^1;               // SFR-Adresse von Port P1.1
sbit E3=P1^2;               // SFR-Adresse von Port P1.2
sbit E4=P1^3;               // SFR-Adresse von Port P1.3
sbit A1=P2^0;               // SFR-Adresse von Port P2.0

bit  M0;                    // Merkerbit M0 deklarieren


void main (void)            // Hauptprogramm
{
      while(1)              // Endlosschleife für die zyklische Programmbearbeitung
      {
```

```
            if ((!E3|E4)==1)   // Abfrage Reseteingang (Eingänge wegen der
                               // Hardwareverschaltung invertiert)
            {
                  M0=1;
            }

            if ((!E1&&!E2)==1) // Abfrage Seteingang (Eingänge wegen der
                               // Hardwareverschaltung invertiert)
            {
                  M0=0;
            }
            A1=M0;
      }
}
```

Zu Übung 8.4

Funktionsplan:



C-Programm:

```
#include<t89c51ac2.h>          // Einbinden der Controller-Bibliothek

// Signalabbilder und symbolische Namen zuordnen:

// Eingaenge (Signalabbild P1):
char bdata EB1=0x00;
```

```
sbit AnachB =EB1^1;              // Abbild von Port P1.1
sbit StopA  =EB1^2;              // Abbild von Port P1.2
sbit BnachA =EB1^3;              // Abbild von Port P1.3
sbit StopB  =EB1^4;              // Abbild von Port P1.4
sbit G_A    =EB1^6;              // Abbild von Port P1.6
sbit G_B    =EB1^7;              // Abbild von Port P1.7

// Ausgaenge (Signalabbild P2):

char bdata AB2=0x00;             // Deklarieren des Signalabbildes
                                 // von P2
sbit MRechts=AB2^1;              // Abbild von Port P2.1
sbit MLinks =AB2^2;              // Abbild von Port P2.2

// Merker definieren:
bit M0;                          // Merkerbit M0 deklarieren
bit M1;                          // Merkerbit M1 deklarieren
bit Start;                       // Merkerbit Start deklarieren

void main (void)                 // Hauptprogramm
{
    P1=0xFF;                     // Vorbereitung, um über P1 Daten einzulesen

    while(1)                     // Endlosschleife für zyklische
                                 // Programmbearbeitung
    {
        EB1=~P1;                 // Eingangssignale lesen, wegen Hardwarebeschaltung
                                 // mit bitweiser Invertierung

            // Flipflop für M_Rechts programmieren:

            if ((!G_A&&AnachB)==1)   // Abfrage Seteingang
            {
                  M0=1;
            }
            if (!G_B==1)             // Abfrage Reseteingang
            {
                  M0=0;
            }
            MRechts=M0&&Start;       // Zuweisung von M_Rechts

            // Flipflop M_Links programmieren:

            if ((!G_B&&BnachA)==1)   // Abfrage Seteingang
            {
                  M1=1;
            }
            if (!G_A==1)             // Abfrage Reseteingang
            {
                  M1=0;
            }
```

```
            MLinks=M1&&Start;        // Zuweisung von M_Links

            // Flipflop Start programmieren:

            if ((AnachB||BnachA)==1) // Abfrage Seteingang
            {
                  Start=1;
            }
            if ((!StopA||!StopB)==1) // Abfrage Reseteingang
            {
                  Start=0;
            }
            // Zuweisungen auf Port 2 übertragen:
            P2=AB2;
      }
}
```

Zu Übung 8.5

```
#include<t89C51ac2.h>

void warten();   // Funktionsprototyp

void main()      // Funktionskopf main
{
      int i;

      char  Tabelle[8] = {0x80,0x40,0x20,0x10,0x08,0x04,0x02,0x01};

      while (1)  // Endlosschleife
      {
            for (i=0;i<8;i++)
            {
                  P2=Tabelle[i];
                  warten();
            }
            for (i=7;i>=0;i--)
            {
                  P2=Tabelle[i];
                  warten();
            }
      }           // Ende Endlosschleife
}                 // Ende main
```

Das Problem des äußeren Stehenbleibens liegt darin, dass die äußeren Einträge der Tabelle zweimal aufgerufen werden. Das Problem kann gelöst werden, indem entweder die obere For-Schleife bei $i = 1$ startet und bei $i < 7$ endet, oder die untere Schleife entsprechend verändert wird.

Zu Übung 8.6

```
#include<t89C51ac2.h>

void warten();   // Funktionsprototyp
```

```
void main()      // Funktionskopf main
{
      int i;

      char  Tabelle[7] = {0x81,0xC3,0xE7,0xFF,0xE7,0xC3,0x81};

      while (1)  // Endlosschleife
      {
            for (i=0;i<7;i++)
            {
                  P2=Tabelle[i];
                  warten();
            }

      }         // Ende Endlosschleife
}               // Ende main
void warten()
{
      unsigned int i;
      for (i=0;i<30000;i++);
}
```

Zu Übung 8.7

```
#include<t89C51ac2.h>

void warten();               // Funktionsprototyp
void wischer();              // Funktionsprototyp
void lauflicht();            // funktionsprototyp

void main()                  // Funktionskopf main
{

    while (1) // Endlosschleife
      {
            wischer();       // Aufruf der Funktion wischer
            lauflicht();// Aufruf der Funktion lauflicht
            wischer();       // etc.
            wischer();
            lauflicht();
            lauflicht();
            lauflicht();

      }      // Ende Endlosschleife
}
            // Ende main

void lauflicht()             // Funktionskopf Lauflicht
{
      int i;                 // interne Variablendeklaration

      char  Tabelle[8] = {0x80,0x40,0x20,0x10,0x08,0x04,0x02,0x01};
```

```
    for (i=0;i<8;i++)        // Lauflicht von links nach rechts.
      {
            P2=Tabelle[i];
            warten();
      }
      for (i=7;i>=0;i--)     // Lauflicht von rechts nach links.
      {
            P2=Tabelle[i];
            warten();
      }
}

void wischer()               // Funktionskopf wischer
{
      int i;                 // interne Variablendeklaration

      char  Tabelle[7] = {0x81,0xC3,0xE7,0xFF,0xE7,0xC3,0x81};

      for (i=0;i<7;i++)      // Durchlauf des Tabellenfeldes
      {
            P2=Tabelle[i];
            warten();
      }
}
void warten()                // Funktionskopf warten: Funktion zum Warten
{
      unsigned int i;
      for (i=0;i<30000;i++);
}
```

Zu Übung 8.8

```
#include <t89c51ac2.h>

sbit CLK=P3^7;               // Takteingang
sbit P34=P3^4;               // Uhr anhalten zum Stellen
sbit P35=P3^5;               // Uhr weiterschalten
bit Puls;                    // globaler Merker Puls

char sec=0,min=0,h=0;        // Variablendeklaration von
                             // sec(Sekunden), min(Minuten) und h (Stunden)

bit flanke ();               // Funktionsprototyp flanke
void stellen();              // Funktionsprototyp stellen

void main()                  // Hauptprogramm
{
      while(1)               // Endlosschleife
      {
            if (flanke())    // Sobald eine Flanke auftritt,
            {
                  sec++;     // werden die Sekunden um eins erhöht.
```

```
            }
            if (sec==60)      // Wenn die Sekunden den Wert 60 erreicht
                              // haben, werden diese wieder zurückgesetzt,
            {                 // und die Minuten um 1 erhöht.
                  sec=0;
                  min++;
            }
            if (min==60)      // Wenn die Minuten 60 erreicht haben,
            {                 // werden die Minuten zurückgesetzt, und
                  min=0;      // die Stunden um 1 erhöht.
                  h++;
            }
            if (h==24)        // Wenn die Stunden 24 erreicht haben,
            {                 // werden diese zurückgesetzt.
                  h=0;
            }
            while (P35==0)    // Solange P35 betätigt wird,
            stellen();        // kann die Uhrzeit verstellt werden-
            P0=h;             // Die Stunden werden auf Port P0 dargestellt.
            P1=min;           // Die Minuten werden auf Port 1 dargestellt.
            P2=sec;           // Sie Sekunden werden auf Port 2 dargestellt.
      }
}

bit flanke ()                 // Funktionskopf Flanke
{
      if(!CLK)                // Abfragen, ob Eingangssignal=0
      {
            Puls=1;           // Puls =1 setzen
      }
      if ((CLK==1)&Puls)      // Positive Flanke erkennen
      {
            Puls=0;           // Puls=0 setzen
            return 1;         // Die Funktion meldet, dass eine Flanke
                              // auftrat
      }
      else
      {
            return 0;         // Es ist keine Flanke aufgetreten.
      }
}

void stellen ()               // Funktionskopf stellen
{
      int i;                  // Lokale Variablendeklaration
      {
            if (P34==0)       // Wenn P34 betätigt wird, so werden die
            {                 // Minuten hochgezählt.
                  min++;
            }
            if (min>60)       // Wenn die Minuten größer 60 sind,
            {                 // werden die Minuten zurückgesetzt,
```

```
                min=0;
                h++;        // und die Stunden um 1 erhöht.
          }
           if (h>23)        // wenn die Stunden größer 23 sind,
          {                 // werden diese zurückgetzt.
                h=0;
          }
          P1=min;           // Ausgabe der Minuten auf Port 1
          P0=h;             // Ausgabe der Stunden auf Port 0
          for(i=0;i<4000;i++); // Wartezeit für den schnellen
                            // Zeitdurchlauf (Uhr stellen)
           sec=0;           // Am Ende der Funktion werden die Sekunden
                            // 0 gesetz, damit die Zeit sekundengenau
                            // eingestellt werden kann.
     }
}
```

Zu Übung 8.9

Die Initialisierungsfunktion muss verändert werden. Dabei wird in der Initialisierung der Befehl „Display ON/OFF“ von 0x0F (Display an: D=1; Cursor an C=1; blinken an B=1) in den HEX-Wert 0x0C (Display an: D=1; Cursor aus C=0 und blinken aus B=0) an P0 geändert.

```
void init2x16()             // Funktionskopf der Initialisierungsfunktion
{
     char k;                // interne Zählvariable
                            // Befehlsliste:
     unsigned char befehl[7]={0x30,0x30,0x30,0x38,0x0C,0x01,0x06};

     warten();              // mehr als 15 ms warten

     for (k=0;k<7;k++)      // alle Befehle hintereinander auf P0
     {                      // übertragen
          RS=0; RW=0; EN=1;
          warten();
          P0=befehl[k];
          EN=0;
     }
     warten();
}
```

Zu Übung 8.10

```
#include <t89c51ac2.h>            // Einbindern der Controllerbibliothek

sbit RS=P3^6;                     // Definition von RS,RW und EN
sbit RW=P3^3;
sbit EN=P3^2;
sbit BSY=P0^7;
                                  // Funktionsprototypen
void init2x16();
void schreiben(char text[]);
void warten();
void fertig();
```

```
void main()                          // Hauptprogramm
{
      init2x16();                    // Aufruf der Initialisierungsfunktion
      schreiben("Hallo Welt!");      // Aufruf der Schreibfunktion
      while(1);                      // Endlosschleife
}

void init2x16()                      // Funktionskopf der Initialisierungsfunktion
{
      char k;                        // interne Zählvariable
                                     // Befehlsliste:
      unsigned char befehl[7]={0x30,0x30,0x30,0x38,0x0C,0x01,0x06};

      warten();                      // mehr als 15 ms warten

      for (k=0;k<7;k++)              // alle Befehle hintereinander auf P0
      {                              // übertragen
            RS=0; RW=0; EN=1;
            warten();
            P0=befehl[k];
            EN=0;
      }
      warten();
}
void schreiben(char text[])
{

      int k=0;
      while(text[k]!='\0')
      {
               RS=1; RW=0;EN=1;
               P0=text[k];
            EN=0;
            fertig();
               k++;
      }
}

void warten()                        // Funktion für die Verzögerungszeit
{
      int i;
      for(i=0;i<100;i++);
}

void fertig()
{
       RS=0;RW=1;EN=1;
      while(BSY);
}
```

Zu Übung 8.11

```
#include <t89c51ac2.h>           // Einbindern der Controllerbibliothek

sbit RS=P3^6;                    // Definition von RS,RW und EN
sbit RW=P3^3;
sbit EN=P3^2;
sbit BSY=P0^7;
                                 // Funktionsprototypen
void init2x16();
void ausgabe_zahl_2Stellen(char zahl);
void warten();
void fertig();

void main()                      // Hauptprogramm
{
      int i;                     // Initialisierung der Zahlvariablen
      unsigned int j;

      init2x16();                // Aufruf der Initialisierungsfunktion

      while(1)                   // Endlosschleife
      {
            for (i=0;i<100;i++)      // Von 0 bis 100 zählen: Die Zählwerte
                                     // sollen auf dem Display
                                     // dargestellt werden.
            {
                  init2x16();              // Aufruf der Initialisierungsfunktion
                  ausgabe_zahl_2Stellen(i); // Zahlen auf das Display
                                           // schreiben
                  for (j=0;j<40000;j++);   // Zeitverzögerung
             }
      }      // Ende Endlosschleife
}            // Ende main

void init2x16()                  // Funktionskopf der Initialisierungsfunktion
{
      char k;                    // interne Zählvariable
                                 // Befehlsliste:
      unsigned char befehl[7]={0x30,0x30,0x30,0x38,0x0C,0x01,0x06};

      warten();                  // mehr als 15 ms warten

      for (k=0;k<7;k++)          // alle Befehle hintereinander auf P0
      {                          // übertragen
            RS=0; RW=0; EN=1;
            warten();
            P0=befehl[k];
            EN=0;
      }
      warten();
}
```

```
void ausgabe_zahl_2Stellen(char zahl)      // Funktionskopf
{
      char zehner,einer;                   // Deklaration der Variablen
      fertig();                            // Warten bis LCD fertig ist.

      zehner=zahl/10;                      // Umwandeln in einzelne Ziffern
      einer=zahl-(10*zehner);

      RS=1; RW=0; EN=1;                    // LCD für den Datentransfer vorbereiten

      P0=zehner+0x30;                      // Zehner übertragen
      EN=0;

      warten();

      RS=1; RW=0; EN=1;                    // LCD für den Datentransfer vorbereiten

      P0=einer+0x30;                       // Einer übertragen
      EN=0;

      fertig();                            // Warten bis LCD fertig ist.
}

void warten()                              // Funktion für die Verzögerungszeit
{
      int i;
      for(i=0;i<100;i++);
}

void fertig()                             // Funktion, um zu warten, bis das Display
{                                         // fertig ist.
      RS=0;RW=1;EN=1;
      while(BSY);
}
```

Zu Übung 8.12

```
#include <t89c51ac2.h>           // Einbindern der Controllerbibliothek

sbit RS=P3^6;                    // Definition von RS,RW und EN
sbit RW=P3^3;
sbit EN=P3^2;
sbit BSY=P0^7;
                                 // Funktionsprototypen
void init2x16();
void pos(char zeile, char stelle);
void schreiben(char text[]);
void sonderzeichen(char zeichen);
void ausgabe_zahl_2Stellen(char zahl);
void warten();
void fertig();

void main()                      // Hauptprogramm
```

```
{
      int i;
      unsigned int j;
      init2x16();                // Aufruf der Initialisierungsfunktion

      while(1)                   // Endlosschleife
      {
            pos(1,1);                    // Position für "Zählen bis 99"
            schreiben("Z hlen bis 99");  // Ausgabe ohne Sonderzeichen
            pos(1,2);                    // Position des Sonderzeichens
            sonderzeichen(0xE1);         // Sonderzeichen ausgeben (siehe ASCII-Code)
            pos(2,0);                    // Position für "Zählerstand"
            schreiben("Z hlerstand: ");  // Ausgabe ohne Sonderzeichen
            pos (2,1);                   // Position des Sonderzeichens
            sonderzeichen(0xE1);// Sonderzeichen ausgeben (siehe ASCII-Code)

            for (i=0;i<100;i++)          // Von 0 bis 100 zählen: Die Zählwerte
                                         // sollen auf den 7-Seg.-Anzeigen
                                         // dargestellt werden.
            {
                  pos(2,14);             // Position der Zahlen
                  ausgabe_zahl_2Stellen(i);
                  for (j=0;j<40000;j++); // Zeitverzögerung
              }
      }                                  // Ende Endlosschleife
                                         // Ende main
}

void init2x16()                  // Funktionskopf der Initialisierungsfunktion
{
      char k;                    // interne Zählvariable
                                 // Befehlsliste:
      unsigned char befehl[7]={0x30,0x30,0x30,0x38,0x0C,0x01,0x06};

      warten();                  // mehr als 15 ms warten

      for (k=0;k<7;k++)          // alle Befehle hintereinander auf P0
      {                          // übertragen
            RS=0; RW=0; EN=1;
            warten();
            P0=befehl[k];
            EN=0;
      }
      warten();
}

void pos(char zeile, char stelle)
{
      if (zeile==2)              // Wenn die Zeile = 2 sein soll,
      {
            zeile=0x40;          // wird die Variable zeile mit der
      }                          // Adresse 0x40 vorbelegt.
```

```
      else
      {
            zeile=0x00;          // ansonsten mit 0x00;
      }
      RS=0; RW=0; EN=1;          // Vorbereitung, um einen Befehl zu senden.

      P0=0x80+zeile+stelle;      // Befehl berechnen und übertragen.

      EN=0;
      warten();                  // Warten, bis Befehl übertragen ist.
}

void sonderzeichen(char zeichen) // Funktionskopf
{
      RS=1; RW=0;EN=1;
      P0=zeichen;        // auf P0 wird das Sonderzeichen als HEX-Code übertragen
      EN=0;
      fertig();
}

void schreiben(char text[])
{
      int k=0;
      while(text[k]!='\0')
      {
            RS=1; RW=0;EN=1;
            P0=text[k];
            EN=0;
            fertig();
            k++;
      }
}

void ausgabe_zahl_2Stellen(char zahl) // Funktionskopf
{
      char zehner,einer;             // Deklaration der Variablen
      fertig();                      // Warten bis LCD fertig ist.

      zehner=zahl/10;                // Umwandeln in einzelne Ziffern
      einer=zahl-(10*zehner);

      RS=1; RW=0; EN=1;              // LCD für den Datentransfer vorbereiten

      P0=zehner+0x30;                // Zehner übertragen
      EN=0;

      warten();

      RS=1; RW=0; EN=1;              // LCD für den Datentransfer vorbereiten

      P0=einer+0x30;                 // Einer übertragen
      EN=0;
```

```
      fertig();                        // Warten bis LCD fertig ist.
}

void warten()                          // Funktion für die Verzögerungszeit
{
      int i;
      for(i=0;i<100;i++);
}

void fertig()
{
      RS=0;RW=1;EN=1;
      while(BSY);
}
```

Zu Übung 8.13

```
#include <t89c51ac2.h>

// Definition der Bits für die Uhr:
sbit CLK=P3^7;                          // Takteingang
sbit P34=P3^4;                          // Uhr anhalten zum Stellen
sbit P35=P3^5;                          // Uhr weiterschalten
bit Puls;                               // globaler Merker Puls

//Definition der Bits für das Display:

sbit RS=P3^6;                           // Definition von RS,RW und EN
sbit RW=P3^3;
sbit EN=P3^2;
sbit BSY=P0^7;

char sec=0,min=0,h=0;                   // Variablendeklaration von
                                        // sec(Sekunden), min(Minuten) und h (Stunden)

//Funktionsprototypen für die Uhr:

bit flanke ();                          // Funktionsprototyp flanke
void stellen();                         // Funktionsprototyp stellen

//Funktionsprototypen für das Display:

void init2x16();                        // Funktionsprototyp Initialisierung
void pos(char zeile, char stelle);      // Funktionsprototyp Positionierung
void schreiben(char text[]);            // Funktionsprototyp Textausgabe
void ausgabe_zahl_2Stellen(char zahl); // Funktionsprototyp Zahlenausgabe
void warten();                          // Funktionsprototyp warten
void fertig();                          // Funktionsprototyp: Abwarten, bis das
                                        // Display den vorherigen Befehl
                                        // bearbeitet hat.

void main()                             // Hauptprogramm
```

```
{
      init2x16();                       // Displayinitialisierung
      pos(1,4);                         // Position für den Text
      schreiben("Uhrzeit");             // Text ausgeben
      while(1)                          // Endlosschleife
      {
            if (flanke())               // Sobald eine Flanke auftritt,
            {
                  sec++;                // werden die Sekunden um eins erhöht.
            }
            if (sec==60)                // Wenn die Sekunden den Wert 60 erreicht
                                        // haben, werden diese wieder zurückgesetzt,
            {                           // und die Minuten um 1 erhöht.
                  sec=0;
                  min++;
            }
            if (min==60)                // Wenn die Minuten 60 erreicht haben,
            {                           // werden die Minuten zurückgesetzt, und
                  min=0;                // die Stunden um 1 erhöht.
                  h++;
            }
            if (h==24)                  // Wenn die Stunden 24 erreicht haben,
            {                           // werden diese zurückgesetzt.
                  h=0;
            }
            while (P35==0)              // Solange P35 betätigt wird,
            stellen();                  // kann die Uhrzeit verstellt werden.
            pos(2,1);                   // Position für die Uhrzeit
            ausgabe_zahl_2Stellen(h);     // Stunden ausgeben
            schreiben(":");               // Doppelpunkt ausgeben
            ausgabe_zahl_2Stellen(min);   // Minuten ausgeben
            schreiben(" sec: ");          // Text "sec:" ausgeben
            ausgabe_zahl_2Stellen(sec);   // Sekuden ausgeben
      }
}

bit flanke ()                           // Funktionskopf Flanke
{
      if(!CLK)                          // Abfragen, ob Eingangssignal=0
      {
            Puls=1;                     // Puls=1 setzen
      }
      if ((CLK==1)&Puls)                // Positive Flanke erkennen
      {
            Puls=0;                     // Puls=0 setzen
            return 1;                   // Die Funktion meldet, dass eine Flanke
                                        // auftrat
      }
      else
      {
            return 0;                   // Es ist keine Flanke aufgetreten.
      }
```

```
}

void stellen ()                         // Funktionskopf stellen
{
      int i;                            // Lokale Variablendeklaration
      {
            if (P34==0)                 // Wenn P34 betätigt wird, so werden die
            {                           // Minuten hochgezählt.
                  min++;
            }
            if (min>60)                 // Wenn die Minuten größer 60 sind,
            {                           // werden die Minuten zurückgesetzt,
                  min=0;
                  h++;                  // und die Stunden um 1 erhöht.
            }
             if (h>23)                  // wenn die Stunden größer 23 sind,
            {                           // werden diese zurückgetzt.
                  h=0;
            }
            pos(2,1);                   // Position für die Uhrzeit
            ausgabe_zahl_2Stellen(h);      // Stunden ausgeben
            schreiben(":");                // Doppelpunkt ausgeben
            ausgabe_zahl_2Stellen(min);    // Minuten ausgeben
            schreiben(" sec: ");           // Text "sec:" ausgeben
            ausgabe_zahl_2Stellen(0);      // Sekuden ausgeben, beim Stellen immer 0
            for(i=0;i<4000;i++);           // Wartezeit für den schnellen
                                        // Zeitdurchlauf (Uhr stellen)
             sec=0;                     // Am Ende der Funktion werden die
                                        // Sekunden 0 gesetz, damit die Zeit
                                        // sekundengenau
                                        // eingestellt werden kann.
      }
}

void init2x16()                         // Funktionskopf der Initialisierungsfunktion
{
      char k;                           // interne Zählvariable
                                        // Befehlsliste:
      unsigned char befehl[7]={0x30,0x30,0x30,0x38,0x0C,0x01,0x06};

      warten();                         // mehr als 15 ms warten

      for (k=0;k<7;k++)                 // alle Befehle hintereinander auf P0
      {                                 // übertragen
            RS=0; RW=0; EN=1;
            warten();
            P0=befehl[k];
            EN=0;
      }
      warten();
}
```

```
void pos(char zeile, char stelle)
{
      if (zeile==2)                    // Wenn die Zeile = 2 sein soll,
      {
            zeile=0x40;                // wird die Variable zeile mit der
      }                                // Adresse 0x40 vorbelegt.
      else
      {
            zeile=0x00;                // ansonsten mit 0x00;
      }
      RS=0; RW=0; EN=1;                // Vorbereitung, um einen Befehl zu senden.

      P0=0x80+zeile+stelle;            // Befehl berechnen und übertragen.

      EN=0;
      warten();                        // Warten, bis Befehl übertragen ist.
}

void schreiben(char text[])
{

      int k=0;
      while(text[k]!='\0')
      {
            RS=1; RW=0;EN=1;
            P0=text[k];
            EN=0;
            fertig();
            k++;
      }
}

void ausgabe_zahl_2Stellen(char zahl)  // Funktionskopf
{
      char zehner,einer;               // Deklaration der Variablen
      fertig();                        // Warten bis LCD fertig ist.

      zehner=zahl/10;                  // Umwandeln in einzelne Ziffern
      einer=zahl-(10*zehner);

      RS=1; RW=0; EN=1;                // LCD für den Datentransfer vorbereiten

      P0=zehner+0x30;                  // Zehner übertragen
      EN=0;

      warten();

      RS=1; RW=0; EN=1;                // LCD für den Datentransfer vorbereiten

      P0=einer+0x30;                   // Einer übertragen
      EN=0;
```

```
      fertig();                          // Warten bis LCD fertig ist.
}

void warten()                            // Funktion für die Verzögerungszeit
{
      int i;
      for(i=0;i<200;i++);
}

void fertig()
{
      RS=0;RW=1;EN=1;
      while(BSY);
}
```

Zu Übung 10.1

Der Timer 0 wird über ein externes Signal gespeist, sodass C/T=1 gesetzt werden muss. Das Gate bleibt 0 für ein softwaremäßigen Start. Durch M1 und M0 wird der Zähler als 16-Bit Zähler konfiguriert.

TMOD nicht bitadressierbar

GATE	C/T	M1	M0	GATE	C/T	M1	M0
x	x	x	x	0	1	0	1
Timer 1				Timer 0			

Es werden zunächst die unteren vier Bit komplett auf Null gesetzt. Dies geschieht über eine bitweise UND-Verknüpfung mit 0xF0 (Hex). Dabei bleiben die oberen 4 Bit erhalten. (x kennzeichnet beliebige Bitkombinationen.)

```
  xxxx xxxx
& 1111 0000 (UND-Verknüpfung)
  xxxx 0000 (Ergebnis)
```

Anschließend wird dem Byte durch eine bitweise ODER-Verknüfung mit 0x05 (Hex) die endgültige Bitkombination zugeordnet.

```
  0000 xxxx
>= 0000 0101 (ODER-Verknüpfung)
  xxxx 0101 (Ergebnis)
```

In C sieht das wie folgt aus:

TMOD=((TMOD&0xF0)|0x05)

Zu Übung 10.2

```
/*
      Systemtakt
*/

#include<t89C51ac2.h>              // Einbinden der Bibliothek des Controllers.

sbit P20=P2^0;
sbit P10=P1^0;
```

```
bit systemtakt;                       // Definition der Variablen Systemtakt

char UZaehler=0x00;                   // Deklaration der Variablen Uberlaufzähler
void Systemtakt_erz();                // Funktionsprototyp zum Weiterschalten des
                                      // Lauflichts

void main()                           // Der Funktionskopf des Hauptprogramms
{
      unsigned char UZaehler=0x00;    // Variablendeklaration des
                                      // Überlaufzählers

      P1=0xFF;                        // Port 1 zum Einlesen vorbereiten
      TMOD=((TMOD&0xF0)|0x01);        // Timer 0 als 16-Bit Timer initialisieren
      UZaehler=0;
      TF0=1;                          // TF0=1, damit sofort der Timer beim Aufruf
                                      // der Funktion Systemtakt_erz richtig geladen
                                      // wird.
      while (1)                       // Endlosschleife für die zyklische
                                      // Programmbearbeitung
      {
            Systemtakt_erz();         // Aufruf der Funktion Systemtakt_erz
            P20=!P10&&systemtakt;     // 0,5 Hz-Takt an P2.0, wenn P10 betätigt
                                      // wird.
      }
}

void Systemtakt_erz()                 // Funktionskopf
{
      if (TF0==1)                     // Wenn der Timer0 übergelaufen ist,
      {
            UZaehler++;               // wird der Überlaufzähler um 1 erhöht,
            TR0=0;                    // der Timer gestoppt und mit
            TL0=0xB0;                 // den berechneten Werten gefüllt.
            TH0=0x3C;
            TF0=0;                    // Zudem wird das Flag gelöscht, und
            TR0=1;                    // der Timer wieder gestartet.
      }
      if (UZaehler==20)               // Wenn der Überlaufzähler den Wert 20
                                      // erreicht hat,
      {
            systemtakt=~systemtakt;   // wird die globale Variable Systemtakt
                                      // invertiert,
            UZaehler=0;               // und der Überlaufzähler wieder
                                      // zurückgesetzt.
      }
}
```

Zu Übung 11.1

```
#include <t89c51ac2.h>               // Einbinden der Controller-Bibliothek
sbit P20=P2^0;

void main()
```

```
{
       ADCF=ADCF|0x08;              // P1.3 als Analogeingang freigeben

       while(1)                     // Endlosschleife
       {
             ADCON=0x2B; //Wandler an P1.3 starten
            while((ADCON&0x10)!=0x10); // Warten bis Wandler fertig ist
            if (ADDH>=127)          // Wenn der Analogwert 1,5V erreicht
                                    // oder überschreitet,
            {
                  P20=1;            // wird P2.0 aktiviert.
            }
            else
            {
                  P20=0;            // ansonsten wird P2.0 deaktiviert.
            }
      }
}
```

Zu Übung 11.2

```
#include <t89c51ac2.h>              // Einbinden der Mikrocontroller-Bibliothek

sbit P40=P4^0;                      // Port Zehner
sbit P41=P4^1;                      // Port Einer

void seg(unsigned char zahl);       // Funktionprototyp 7-Segmentanzeige
void warten(void);                  // Funktionsprototyp warten

void main(void)                     // Hauptprogramm
{
      int j;                        // Variablendefinition für Zählvariablen
      unsigned char wert;           // Variablendefinition für den berechneten A/D-Wert

      ADCF=ADCF|0x08;               // P1.3 als Analogeingang freigeben

      while (1)                     // Endlosschleife
      {

            ADCON=0x2B;             // Wandler an P1.3 starten
            while((ADCON&0x10)!=0x10); // Warten bis Wandler fertig ist
            wert= ADDH/255.0*30;       // Wert zwischen 0.0-3.0 Volt
                                    // berechnen (ohne Komma)
            seg(wert);              // berechneten Wert darstellen
      }                             // Ende Endlosschleife
}                                   // Ende main


void seg(unsigned char zahl)        // 7-Segment-Ansteuerung
{
      unsigned char z=0;            // Variable z für Zehner
      unsigned char e=0;            // Variable e für Einer
```

```
                                    // Tabelle der Anzeigewerte für die 7 Segmente:
      unsigned char ziffer[10]={0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F};

      e=zahl%10;                    // Einer ermitteln mittels Modulo-Operator
      z=(zahl-e)/10;                // Zehner ermitteln
      P40=1;                        // Display Einer wählen
      P41=0;
      P2=ziffer[e];                 // Einer ausgeben
      warten();                     // Funktion warte zur Stabilisierung der Anzeige

      P40=0;                        // Display Zehner wählen
      P41=1;
      P2=ziffer[z]+0x80;            // Zehner ausgeben mit Punkt (+0x80)
      warten();                     // Funktion warte zur Stabilisierung der Anzeige
}

void warten(void)                   // Funktion warten
{
      unsigned int j;               // Eine Zählvariable wird von
                                    // 0 bis 300 hochgezählt.
      for(j=0;j<300;j++);
}
```

Zu Übung 12.1

```
#include <t89c51ac2.h>              // Einbinden der Bibliothek
#include <stdio.h>                  // Einbinden der Standardbibliothek

void main(void)
{
      PCON=PCON|0x80;
      SCON=0x52;                    // Mode 1 (Startbit, 8 Datenbits, 1 Stoppbit)
      TI=0;                         // Sendeflag löschen

      TMOD=0x20;                    // Timer1 im 8 Bit-Reload-Betrieb (Mode 2)
      TL1=0xF3;                     // Start-Wert ist F3
      TH1=0xF3;                     // Reload-Wert ist F3 für 2400 Baud
      TR1=1;                        // Timer1 einschalten
      while(1)
      {
              printf("  Hallo Welt!  ");      // Ausgabe von Hallo Welt mit printf
      }
}
```

Zu Übung 16.1

1.
```
MOV C,E1        ;UND
ANL C,/M1
ANL C,E2
CPL C           ;Ausgangsinvertierung
ORL C,/E3       ;ODER
CPL C           ;Ausgangsinvertierung
MOV A1,C        ;= A1
```

```
2. MOV C,A5         ;ODER
   ORL C,/E1
   ORL C,M1
   ORL C,/M2
   ANL C,/E2        ;UND
   ANL C,E3
   CPL C            ;Ausgangsinvertierung
   MOV M9,C         ;= M9
   ORL C,A5         ;ODER
   MOV A7,C         ;= A7
```

Zu Übung 16.2

```
1.         MOV C,E1         ;UND
           ANL C,E2
           CPL C            ;Ausgangsinvertierung
           ORL C,/M1        ;ODER
           JNC S1           ;Setzbedingung A1 erfüllt?   Nein:
           SETB A1          ;Ja: setzen A1
   S1:     MOV C,E3         ;ODER
           ORL C,/M2
           JNC S2           ;Rücksetzbedingung A1 erfüllt?  Nein:
           CLR A1           ;Ja: rücksetzen A1
   S2:
2.         MOV C,E1         ;UND
           ANL C,M1
           CPL C            ;Ausgangsinvertierung
           ORL C,A2         ;ODER
           CPL C
           JNC S3           ;Setzbedingung M10 erfüllt?  Nein:
           SETB M10         ;Ja: M10 setzen
   S3:     MOV C,E2         ;UND
           ANL C,M2
           CPL C            ;Ausgangsinvertierung
           MOV M20,C        ;Hilfsmerker
           MOV C,E1         ;UND
           ANL C,/A3
           ORL C,M20        ;ODER
           JNC S4           ;Rücksetzbedingung M10 erfüllt?  Nein:
           CLR M10          ;Ja: M10 rücksetzen
   S4:
```

Zu Übung 16.3

```
;************************ Transportsteuerung ************************
;
;---------------------------- Zuweisungen -----------------------------
;
;Title "Transportsteuerung Datei TRANS.ASM"
$nomod51
$include(t89c51ac2.INC)
```

```
ein1 equ 20h                    ;Eingangssignalabbild
aus2 equ 21h                    ;Ausgangssignalabbild
m1 equ 22h                      ;interne Merker
anachb equ ein1.1
stopa equ ein1.2
bnacha equ ein1.3
stopb equ ein1.4
g_a equ ein1.6
g_b equ ein1.7
m_rechts equ aus2.1
m_links equ aus2.2
start equ m1.3
;
;------------------------ Initialisierung ----------------------------
;
mov ein1,#00h
mov aus2,#00h
mov p2,#0ffh
;
;----------------- Programm gemäß Funktionsplan ---------------------
;
tr1:    mov ein1,p1             ;Eingangssignalabbild einlesen
        mov c,anachb            ;m1.0 setzen?
        anl c,/g_a
        jnc tr2                 ;Nein
        setb m1.0               ;Ja
tr2:    mov c,g_b               ;m1.0 rücksetzen?
        cpl c
        jnc tr3                 ;Nein
        clr m1.0                ;Ja
tr3:    mov c,m1.0              ;Ausgang Motor Rechts
        anl c,start
        mov m_rechts,c
        mov c,bnacha            ;m1.1 setzen?
        anl c,/g_b
        jnc tr4                 ;Nein
        setb m1.1               ;Ja
tr4:    mov c,g_a               ;m1.1 rücksetzen?
        cpl c
        jnc tr5                 ;Nein
        clr m1.1                ;Ja
tr5:    mov c,m1.1              ;Ausgang Motor Links
        anl c,start
        mov m_links,c
        mov c,anachb            ;Start setzen?
        orl c,bnacha
        jnc tr6                 ;Nein
        setb start              ;Ja
tr6:    mov c,stopa             ;Start rücksetzen?
        orl c,stopb
        jnc tr7                 ;Nein
        clr start               ;Ja
```

```
tr7:    mov p2,aus2             ;Ausgangssignalabbild ausgeben
        ljmp tr1                ;zyklische Bearbeitung
;************************************************************************
END
```

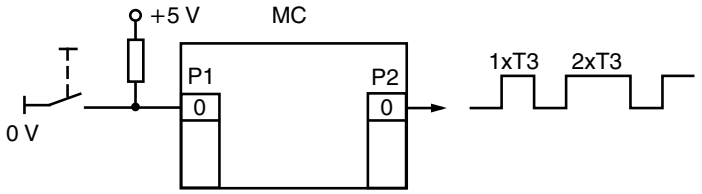
Zu Übung 16.4

Das Unterprogramm ZEIT wird aus einer 3-Register-Zeitschleife aufgebaut. Register R3 enthält den Zeitfaktor für R1 und R2. Register R3 wird vor dem Aufrug des Unterprogramms ZEIT3 mit dem Faktor 1 oder 2 geladen.

Die Umschaltung der Frequenz soll über das Eingangssignal an P1.0 erfolgen.

P1.0 = 0 ⇒ langsame Frequenz ⇒ Zeitfaktor R3 = 2
P1.0 = 1 ⇒ schnelle Frequenz ⇒ Zeitfaktor R3 = 1

Die Frequenz wird an P2.0 ausgegeben.



Assemblerprogramm GEN2.ASM:

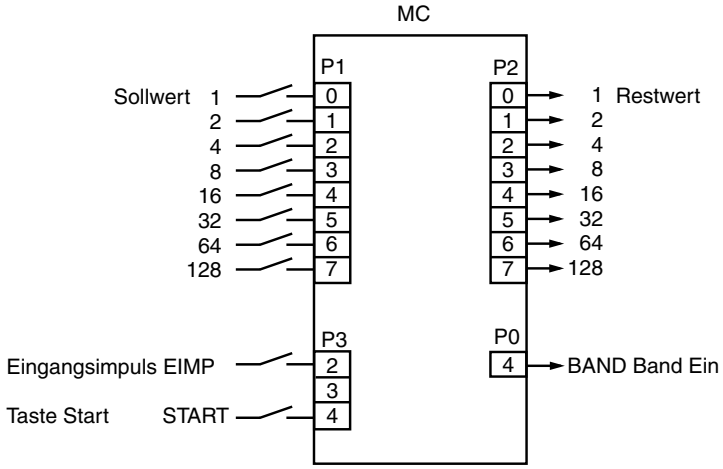
```
;****************** Umschaltbarer Frequenzgenerator *****************
;
;Title "Umschaltbarer Frequenzgenerator; Datei GEN2.ASM"
$nomod51
$include(t89c51ac2.INC)
clr p2.0                        ;Frequenzausgang = 0
uf0:    jb p1.0,uf1             ;Frequenzwahl = 0?   Nein:
        mov r3,#02h             ;Ja: Zeit = 2xT3
        ljmp uf2
uf1:    mov r3,#01h             ;Zeit = 1xT3
uf2:    lcall zeit3             ;Zeitverzögerung T3
        cpl p2.0                ;Frequenzausgang invertieren
        ljmp uf0                ;zyklische Bearbeitung
;
;------------------- Unterprogramm ZEIT3 ----------------------------
zeit3:  mov r2,#0ffh
ze2:    mov r1,0ffh
ze1:    djnz r1,ze1
        djnz r2,ze2
        djnz r3,zeit3
        ret
;************************************************************************
END
```

Zu Übung 16.5

Das Programm „Zählersteuerung 1“ soll weitgehend übernommen und nur an den erforderlichen Stellen abgeändert werden.

Beim Hauptprogramm entfallen die Codewandler bei der Sollwerteingabe und der Istwertausgabe. Da der Restwert angezeigt werden soll, muss vor der Ausgabe der Ist- vom Sollwert abgezogen werden. Bei der Verknüpfungssteuerung fällt das Stop-Signal weg. Die übrige Logik kann übernommen werden. Sie bekommt den Dateinamen VSTEU1.ASM.

Anschlussplan:



Hauptprogramm ZSTEU2.ASM:

```
;*********************************************************************
;Zählersteuerung 2                    Datei ZSTEU2.ASM
;
;Eingangssignale: Sollwert Dual, 8 Bit     Port P1
;                 Eingangsimpuls           Port P3.2
;                 Start                    Port P3.4
;Ausgangssignale: Restwert Dual, 8 Bit     Port P2
;                 Band Ein                 Port 0.4
;*********************************************************************
;Title "Zählersteuerung 2, Datei ZSTEU2.ASM"
$nomod51
$include(t89c51ac2.INC)
;
;------ Zuordnung: symbolische Variable zu absoluten Adressen -------
;
ein1       equ        20h             ;Eingangssignalabbild Port 1
ein3       equ        21h             ;Eingangssignalabbild Port 3
aus0       equ        22h             ;Ausgangssignalabbild Port 0
aus2       equ        23h             ;Ausgangssignalabbild Port 2
im         equ        24h             ;interne Merker
esoll      equ        25h             ;Eingabe-Sollwert Dual
soll       equ        26h             ;Sollwert
ist        equ        27h             ;Istwert (Impulszähler)
```

```
zeitz1     equ        28h             ;Zeitzähler 1
zeitz2     equ        29h             ;Zeitzähler 2
eimpe      equ        im.0            ;Eingangsimpuls entprellt
eimpe_a    equ        im.1            ;Eingangsimpuls entprellt Alt
sgi        equ        im.2            ;Soll =< Ist
eimp       equ        ein3.2          ;Eingangsimpuls
start      equ        ein3.4          ;Start
band       equ        aus0.4          ;Band Ein
;
;------------------------- Initialisierung --------------------------
;
mov p1,#0ffh
mov p3,#0ffh
mov ein1,#00h
mov ein3,#00h
mov aus0,#00h
mov aus2,#00h
mov im,#00h
mov zeitz1,#0ffh
mov zeitz2,#10h
mov soll,#00h
mov esoll,#00h
mov ist,#00h
;
;--------------------------- Hauptprogramm --------------------------
;
zs1:   mov ein1,p1              ;Eingangssignale einlesen
       mov ein3,p3
       mov esoll,ein1           ;Eingabe-Sollwert Dual
       lcall vsteu              ;Verknüpfungssteuerung
       lcall izahl              ;Impulszähler
       clr c                    ;Restwert für Anzeige ermitteln
       mov a,soll
       subb a,ist
       mov aus2,a               ;Restwert für Anzeige
       mov p0,aus0              ;Ausgangssignale ausgeben
       mov p2,aus2
       ljmp zs1                 ;zyklische Bearbeitung
;
;----------------- Bereitstellen der Unterprogramme -----------------
;
include "vsteu1.asm"
include "izahl.asm"
;
;*********************************************************************
```

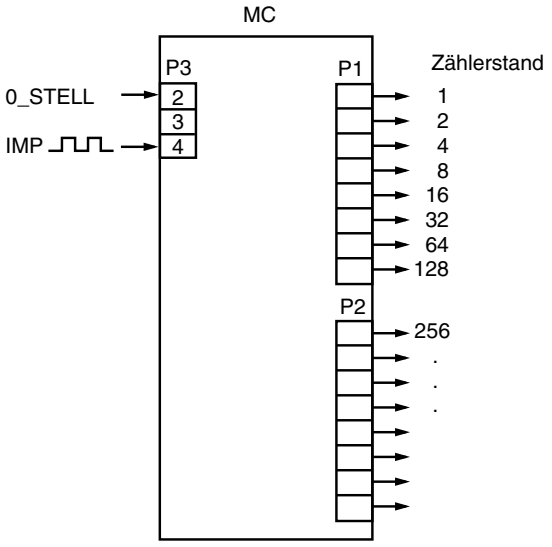
Programm der Verknüpfungssteuerung VSTEU1.ASM:

```
;*********************************************************************
;Verknüpfungssteuerung (ohne Stoptaste)       Datei VSTEU1.ASM
;*********************************************************************
;Variable: START
;          BAND       = Bandmotor Ein
```

```
;          Soll       = Sollwert
;          IST        = Istwert (Impulszähler)
;          ESOLL      = Eingabe-Sollwert als Dualzahl
;          SGI        = Soll =< Ist
;---------------------------------------------------------------------
;Title "Verknüpfungssteuerung, Datei VSTEU1.ASM"
;Controller 80535
vsteu: mov c,start               ;Sollwert speichern?
       anl c,/band
       jnc vs1                   ;Nein
       mov soll,esoll            ;Ja
vs1:   mov c,sgi                 ;Istwert nullstellen?
       anl c,start
       jnc vs2                   ;Nein
       mov ist,#00h              ;Ja
vs2:   clr c                     ;Soll =< Ist?
       mov a,ist                 ;(Ist - Soll: wenn Soll=<Ist, C=0)
       subb a,soll
       cpl c                     ;Ja: SGI=1      Nein: SGI=0
       mov sgi,c
       jnb start,vs3             ;Band Ein setzen? Nein:
       setb band                 ;Ja
vs3:   mov c,sgi                 ;Band Ein rücksetzen?
       jnc vs4                   ;Nein
       clr band                  ;Ja
vs4:   ret
;*********************************************************************
END
```

Zu Übung 17.1

16-Bit-Zähler:



Assemblerprogramm ZAEHL16.ASM:

```
;*********************************************************************
;                    16-Bit-Ereigniszähler
;*********************************************************************
;title "16-Bit-Ereigniszähler, Datei ZAEHL16.ASM"
$nomod51
$include(t89c51ac2.INC)
;---------------------- Initialisierung Timer0 ----------------------
clr tr0                         ;Timer stoppen
mov th0,#00h                    ;Timer 0-stellen
mov tl0,#00h
anl tmod,#0f0h                  ;C/T=0, GATE=0, Modus=0
orl tmod,#05h
clr tf0                         ;Überlaufflag=0
setb tr0                        ;Timer0 starten
;------------------------ Programm ----------------------------------
anf:   jnb p3.2,er1             ;Zähler 0-stellen?   Nein:
       clr tr0                  ;Ja:
       mov th0,#00h
       mov tl0,#00h
       setb tr0
er1:   mov a,th0                ;Zählerstand lesen:
       mov r2,tl0
       cjne a,th0,er1           ;TH0 während Auslesen konstant? Nein:
       mov p2,a                 ;Ja: Zählerstand ausgeben
       mov p1,r2
       ljmp anf                 ;zyklische Programmbearbeitung
;*********************************************************************
END
```

Zu Übung 17.2

Hauptprogramm für Störmeldung:

```
;*********************************************************************
;Hauptprogramm für Störmeldung STOEBLI.ASM mit Systemtakt HZ05
;*********************************************************************
$nomod51
$include(t89c51ac2.INC)
;----------------------------Zuweisungen---------------------------
stoer equ p1.0           ;Signal "Störung"
blink equ p2.0           ;Ausgang Blinklicht
merk equ 20h
uz equ 21h               ;Überlaufzähler
hz05 equ merk.0          ;Systemtakt 0,5Hz
;---------------------------Initialisierung-------------------------
mov merk,#00h
mov uz,#00h
anl tmod,#0f0h           ;Timer 0 als 16-Bit-Timer
orl tmod,#01h
setb tr0
;---------------------------Hauptprogramm---------------------------
st0:lcall systa          ;Unterprogramm Systemtakt aufrufen
```

```
    mov c,stoer
    anl c,hz05
    mov blink,c
    ljmp st0             ;zyklische Bearbeitung
$include (systa.asm)     ;Unterprogramm Systemtakt
END
```

Zu Übung 17.3

Assemblerprogramm ZUBERW.ASM:

```
;*********************************************************************
;              Zeitüberwachung mit Störmeldung
;*********************************************************************
;title "Zeitüberwachung mit Störmeldung, Datei ZUBERW.ASM"
$nomod51
$include(t89c51ac2.INC)
;------------------------ Zuweisungen -------------------------------
ein3 equ 20h                  ;Eingangssignalabbild
aus2 equ 21h                  ;Ausgangssignalabbild
quit equ ein3.2               ;Qittungssignal
zeitz equ 22h                 ;Zeitzähler
;----------------------- Initialisierung ----------------------------
;Allgemein:
mov p3,#0ffh
mov ein3,#00h
mov aus2,#00000001b
mov zeitz,#00h
;
;Timer 0 als 16-Bit-Zeitzähler:
clr tr0                       ;Timer stoppen
clr tf0
mov tl0,#00h                  ;0-stellen
mov th0,#00h
anl tmod,#0f0h                ;Modus 16-Bit-Zeitgeber
orl tmod,#00000001b
setb tr0                      ;Timer starten
;
;Timer 1 als 8-Bit-Reload-Ereigniszähler:
clr tr1                       ;Timer stoppen
clr tf1
mov tl1,#252d                 ;auf 252 stellen
mov th1,#252d
anl tmod,#0fh                 ;Modus 8-Bit-Reload-Ereigniszähler
orl tmod,#01100000b
setb tr1                      ;Timer starten
;-------------------------- Programm --------------------------------
zu1:  mov ein3,p3             ;Eingangssignale einlesen
      jnb tf0,zu2             ;Überlauf Timer 0?   Nein:
      inc zeitz               ;Ja: Zeitzähler +1
zu2:  mov a,zeitz             ;Zeitzähler = 77 (=5s)?
      cjne a,#77d,zu3         ;Nein:
      clr tr1                 ;Ja: Ereigniszähler auf 252
```

```
      mov tl1,#252d
      setb tr1
      mov zeitz,#00h           ;    Zeitzähler auf 0
zu3:  jnb tf1,zu4              ;Überlauf Ereigniszähler?  Nein:
      mov aus2,#10000000b      ;Ja: Störung Ein
zu4:  jnb quit,zu5             ;Quittung = 1?   Nein:
      mov aus2,#00000001b      ;Ja: Störung Aus
      clr tf1                  ;TF1 rücksetzen
      mov zeitz,#00h           ;    Zeitzähler = 0
zu5:  mov p2,aus2              ;Ausgangssignale ausgeben
      ljmp zu1                 ;zyklische Bearbeitung
;*********************************************************************
END
```

Zu Übung 17.4

Assemblerprogramm SER3.ASM:

```
;*********************************************************************
Title "Serielle Eingabe über Tastatur und Ausgabe auf Bildschirm,
Datei ser3.asm"
;*********************************************************************
$nomod51
$include(t89c51ac2.INC)
;--------------------------- Zuweisungen -----------------------------
end equ 00h                    ;Zeichen "Ende" der Zeichenkette
;------------ Initialisieren serielle Schnittstelle -----------------
Mov scon,#52h                  ;1Start,8Dat,1Stop
orl pcon,#80h                  ;Baudrate 9600 (smod = 1)
setb bd                        ;Baudratengenerator ein
;----------- Programm: serielle Eingabe über PC-Tastatur ------------
eing:    lcall serin           ;warten auf Zeichen S oder s
         anl a,#11011111b      ;Bit 5 ausblenden
         cjne a,#53h,eing
;- Programm: serielle Ausgabe einer Zeichenkette über PC-Bildschirm -
         mov r2,#00h           ;Zeichenzeiger 0-Stellen
         mov dptr,#zk0         ;Basisadresse Zeichenkette
bi3:     mov a,r2              ;Zeichen laden
         movc a,@a+dptr
         cjne a,#end,bi1       ;Zeichen "Ende" ?   Nein:
         ljmp bi2              ;Ja:
bi1:     lcall seraus1         ;Zeichen seriell ausgeben
         inc r2                ;nächstes Zeichen
         ljmp bi3
bi2:     ljmp eing             ;zyklische Bearbeitung
;
;---------------------------------------------------------------------
;
;--------------- Unterprogramm serielle Eingabe ---------------------
serin:    jnb ri,serin        ;warten, bis neues Zeichen eingetroffen
          mov a,sbuf
          clr ri              ;Flag "Empfangsreg. leer" setzen
          ret
```

```
;--------------- Unterprogramm serielle Ausgabe ---------------------
seraus1:  jnb ti,seraus1      ;warten, bis Senderegister frei
          clr ti              ;Flag "Sendereg. beschrieben" setzen
          mov sbuf,a          ;Byte nach Sendereg. und Senden starten
          ret
;------------ auszugebende Zeichenkette: "Hallo!" -------------------
zk0:      db 0ch              ;Bildschirm löschen
          text "Hallo!"       ;Text
          db 00h              ;Zeichen "Ende"
;***********************************************************************
END
```

Zu Übung 17.5

Assemblerprogramm OFENWACH.ASM:

```
;***********************************************************************
;Programm zur Ofenüberwachung OFENWACH.ASM
;
;Fehlermeldungen eines Brennofens werden in einer Steuerwarte
;angezeigt und quittiert.
;***********************************************************************
$nomod51
$include(t89c51ac2.INC)
;-------------------------------Zuweisungen---------------------------
ein3 equ 20h                      ;Eingangsignalabbild Port 3
aus4 equ 21h                      ;Ausgangssignalabbild Port 4
m1 equ 22h                        ;interne Merker
f1 equ ein3.2                     ;Fehler 1
f1a equ m1.0                      ;Fehler 1 Alt
f2 equ ein3.3                     ;Fehler 2
f2a equ m1.1                      ;Fehler 2 Alt
taste equ 23h                     ;eingelesene Taste
;-----------------------------Initialisieren------------------------
mov m1,#00h
mov taste,#00h
mov ein3,#00h
mov aus4,#00h
mov scon,#52h                     ;serielle Schnittstelle
orl pcon,#80h
setb bd
;-----------------------------Hauptprogramm-------------------------
of0:mov ein3,p3                   ;Eingangssignale einlesen
  jnb f1,of1                      ;positive Flanke an F1?
  jb f1a,of1
  mov dptr,#text1                 ;Text 1 ausgeben
  lcall textaus
of1:mov c,f1                      ;f1 nach f1a
  mov f1a,c

  jnb f2,of2                      ;positive Flanke an F2?
  jb f2a,of2
  mov dptr,#text2                 ;Text 2 ausgeben
```

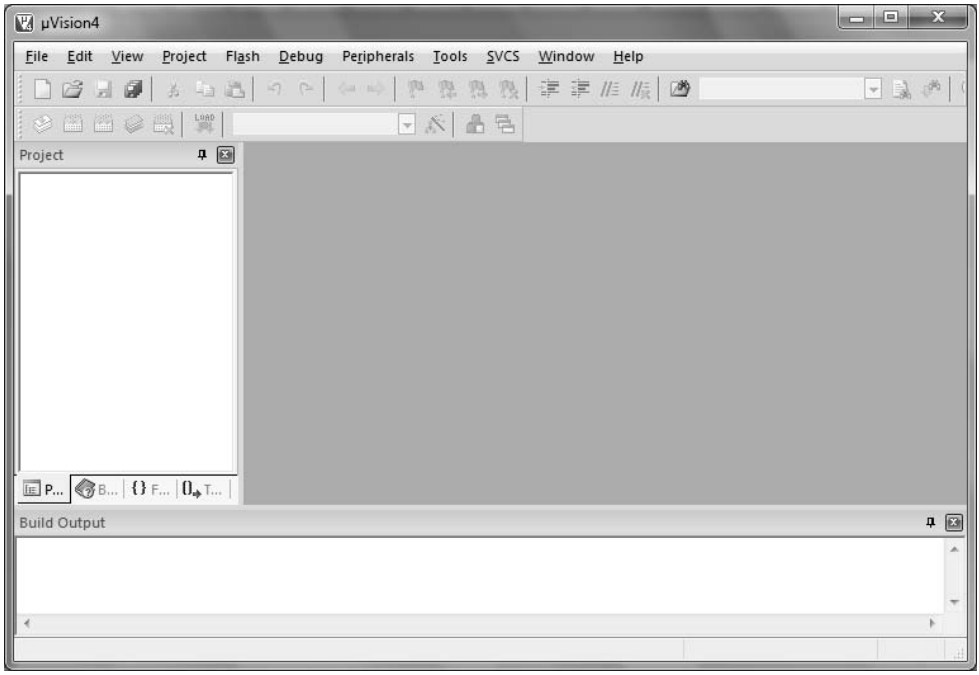
```
  lcall textaus
of2:mov c,f2                    ;f2 nach f2a
  mov f2a,c

  jnb ri,of0                    ;warten auf Taste
  mov taste,sbuf                ;Taste R ?
  cjne a,#"R",of3
  mov c,f1                      ;und Fehler 1 oder 2?
  orl c,f2
  jnc of3
  mov dptr,#text3               ;Text 3 ausgeben
  lcall textaus
of3:mov a,taste                 ;Taste Q ?
  cjne a,#"Q",of4
  jb f1,of4                     ;und keine Fehlermeldung
  jb f2,of4
  mov a,#0ch                    ;Bildschirm löschen
  lcall seraus
of4:ljmp of0                    ;zyklische Bearbeitung


;------------------------------Unterprogramme-------------------------
seraus:jnb ti,seraus            ;serielle Ausgabe
    mov sbuf,a
    clr ti
    ret

textaus:mov r2,#00h             ;Textausgabe
aus3:   mov a,r2
        movc a,@a+dptr
        cjne a,#00h,aus1
        ljmp aus2
aus1:   lcall seraus
        inc r2
        ljmp aus3
aus2:   ret
;------------------------------Texte----------------------------------
text1:db 0ch                    ;Bildschirm löschen
      text "Brennstoffzufuhr gestört"
      db 00h                    ;Text Ende
text2:db 0ch
      text "Ofentemperatur zu hoch"
      db 00h
text3:db 0ch
      text "Fehler registriert"
      db 00h
;*********************************************************************
END
```

19 Anhang

19.1 Erstellen eines Projektes mit Keil µVision 4

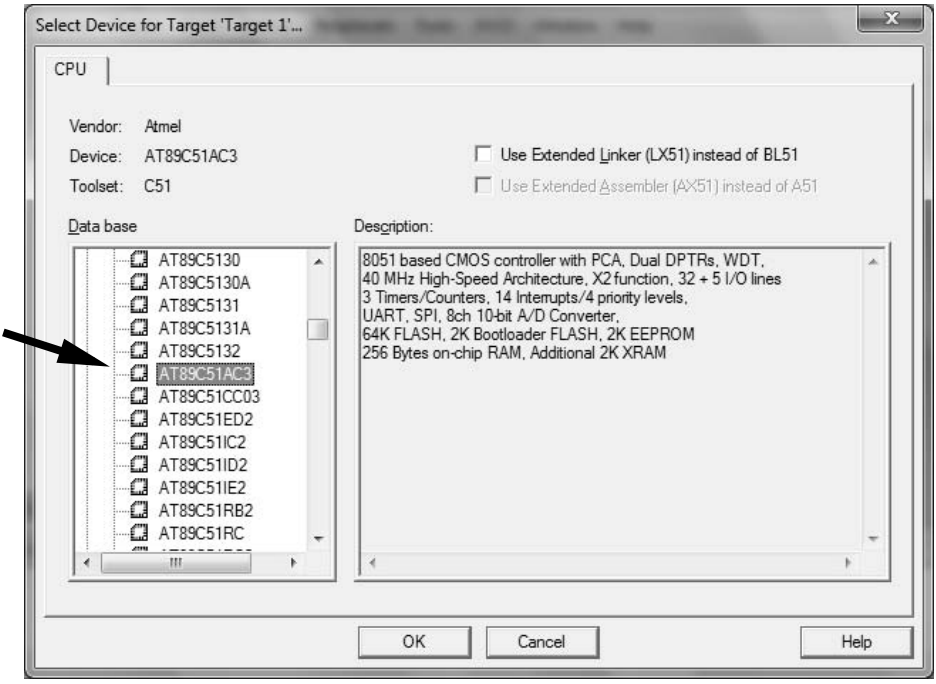
Hier wird gezeigt, wie Projekte mit der Entwicklungsumgebung µVision 4 von Keil für den Mikrocontroller Atmel AT89C51AC3 erstellt werden. Die Entwicklungsumgebung lässt sich kostenlos von Keil beziehen, mit der Einschränkung, dass nur Programme mit einer Größe von maximal 2 KByte erzeugt werden können.

a) Aufruf des Programms. Nachdem das Programm gestartet wurde, erscheint folgender Bildschirm:



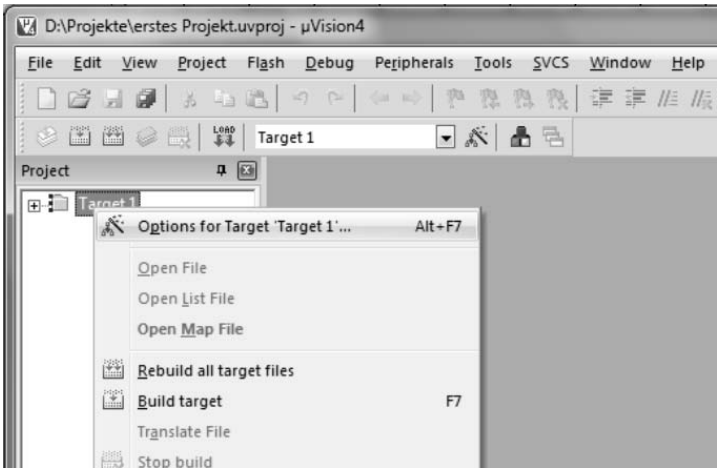
b) Unter dem Register „Project“ den Menüpunkt „new µVision Project …“ anklicken, und einen Namen für das zu erstellende Projekt erstellen. Anschließend erscheint ein Fenster, in dem der zu programmierende Mikrocontroller ausgesucht werden muss.

Controller „ATMEL“ → AT89C51AC3 auswählen.

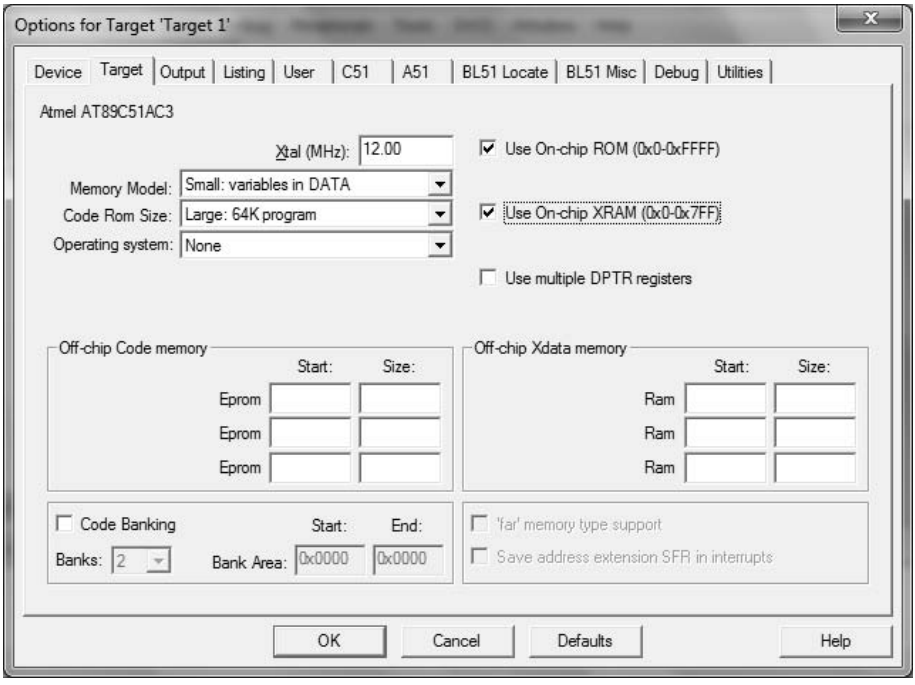


c) Standard 8051 Startup Code zum Projekt hinzufügen, indem die nachfolgende Abfrage mit „Ja“ bestätigt wird.

d) Es wird nun im linken Projektfenster der Punkt „Target 1“ erzeugt. Mithilfe der rechten Maustaste lassen sich die Optionen einstellen. (Auswahl „Options for Target 1“)

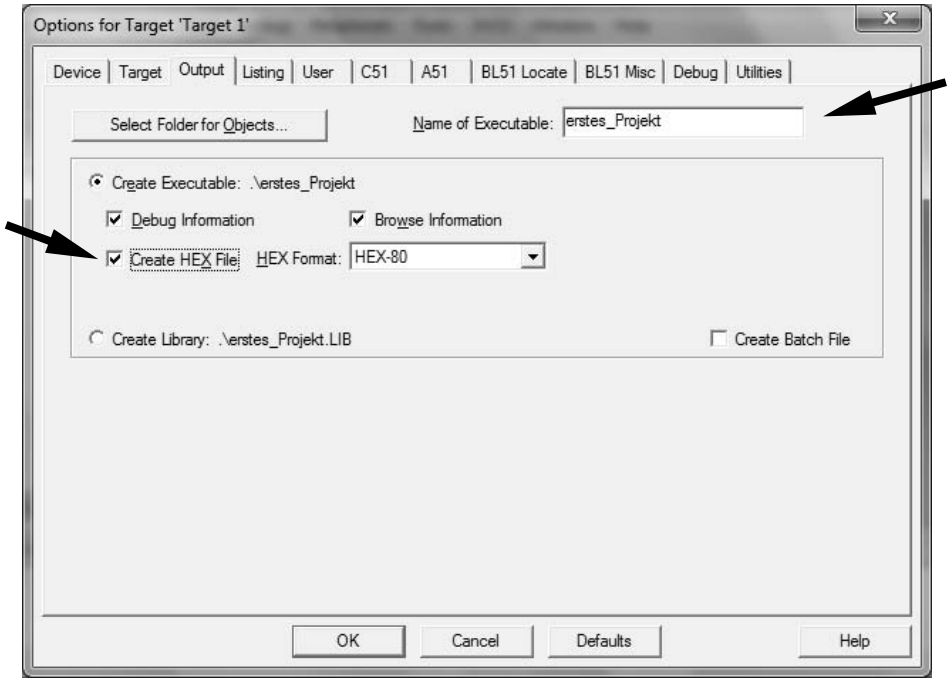


e) Es erscheint folgende Auswahl:



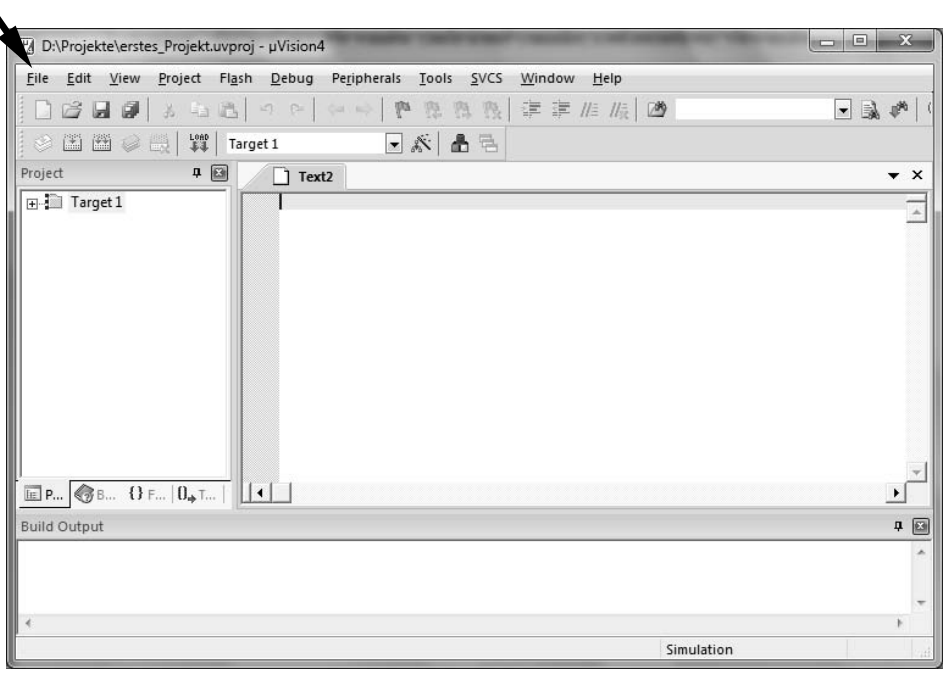
Unter Target können weitere Optionen eingestellt werden, zum Beispiel die Taktfrequenz des verwendeten Oszillators, die Nutzung des ON-Chip-ROMs und des ON-Chip XRAMs und weitere Optionen. Hier sollen die oben gewählten Optionen und Einstellungen vorgenommen werden.

f) Unter dem Register „Output“ muss die Option „Create Hex_File“ aktiviert werden, damit ein Hex-File von dem Projekt erzeugt wird. Der Name des erzeugten „Hex-Files“ steht hinter „Name of Executable“ und lässt sich frei wählen. (Bitte keine Umlaute, Leerzeichen etc. verwenden.)

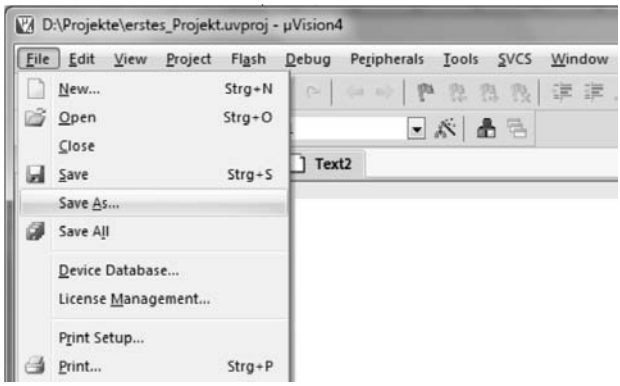


Die übrigen Register müssen nicht verändert werden, da diese schon richtig konfiguriert sind. Mit „OK“ alle Einstellungen bestätigen.

g) Unter File → NEW eine neue Textdatei erzeugen.

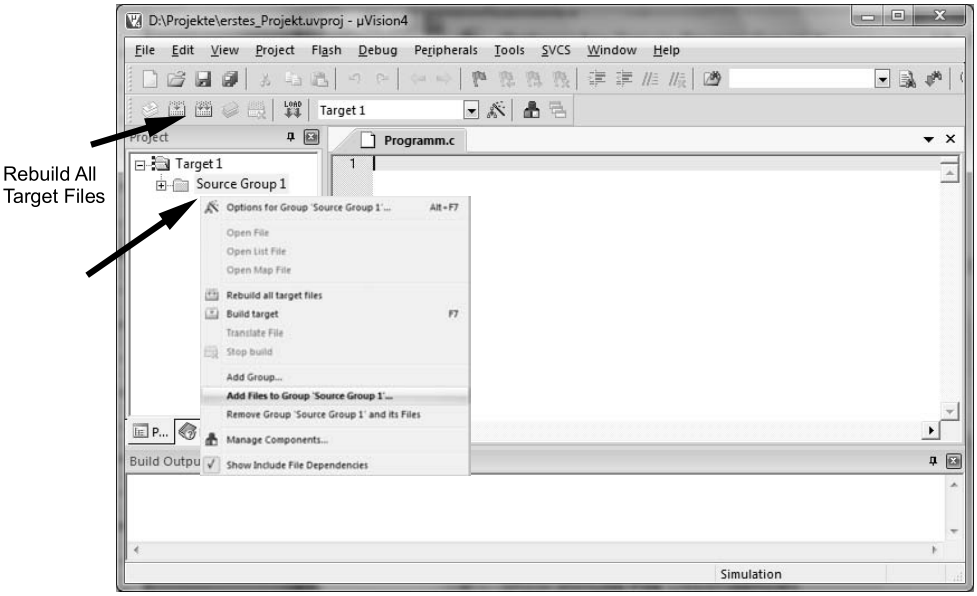


h) Diese Datei unter „Save As …“ mit der Endung .c (Für C-Code) oder .asm (für Assembler-Code) abspeichern.



i) Mit der rechten Maustaste unter SourceGroup1 lässt sich mit „Add Files to Group..“ der erstellte Programmdatei in das Projekt einbinden.

Anschließend erscheint diese im Projektfenster.



j) Jetzt kann in der Textdatei das zu erstellende Programm geschrieben werden.

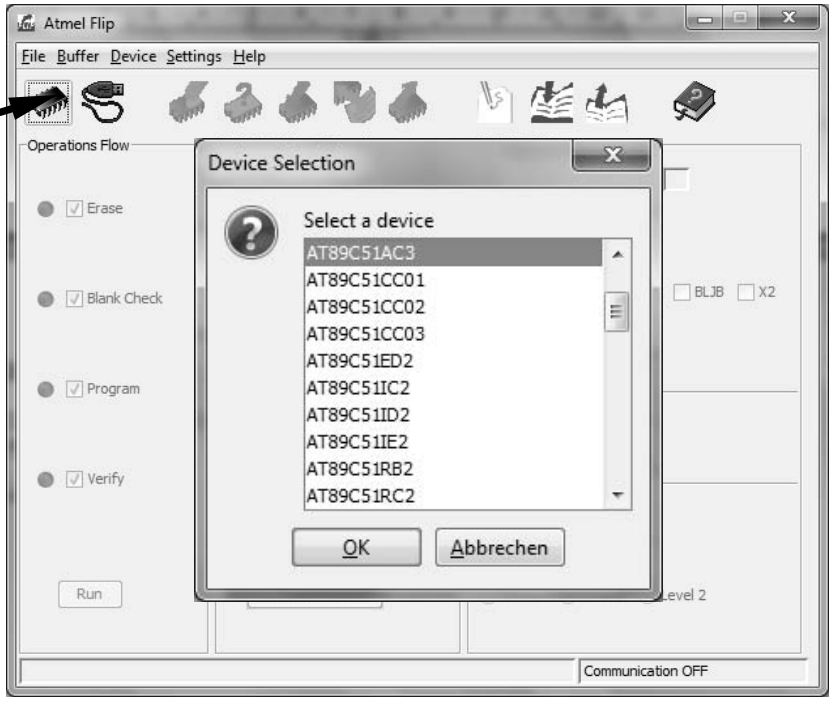
k) Nachdem dieses erstellt wurde, lässt sich mit dem Button „Rebuild All Target Files“ das Projekt compilieren. Der Hex-File wird nun automatisch erzeugt. Programmfehler werden im unteren Fenster aufgezeigt, und müssen vor der Erzeugung des HEX-Files behoben werden. Nun steht der HEX-File bereit, um diesen in den Mikrocontroller zu übertragen.

l) Falls eine Simulation gewünscht wird, so lässt sich diese unter „Options for Taget“ im Kontext „Debug einstellen“ (siehe Punkt e/f). Mithilfe der Lupe „Debug“ (siehe Unten) wird der Simulator gestartet. Weitere Hinweise können der Onlinehilfe entnommen werden.

19.2 Übertragen des HEX-Files auf den AT89C51 AC3 mittels Atmel Flip

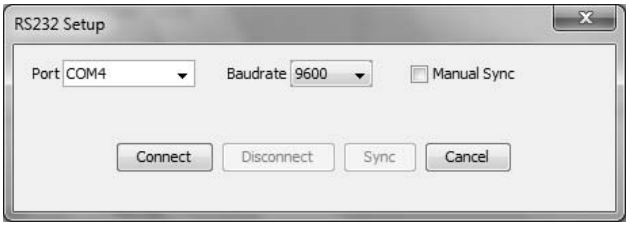
Der erzeugte Hex-File wird z. B. mit dem Programm Atmel Flip, welches kostenlos von Atmel zur Verfügung gestellt wird, auf den Controller übertragen.

a) Programm starten und den passenden Controller aussuchen.



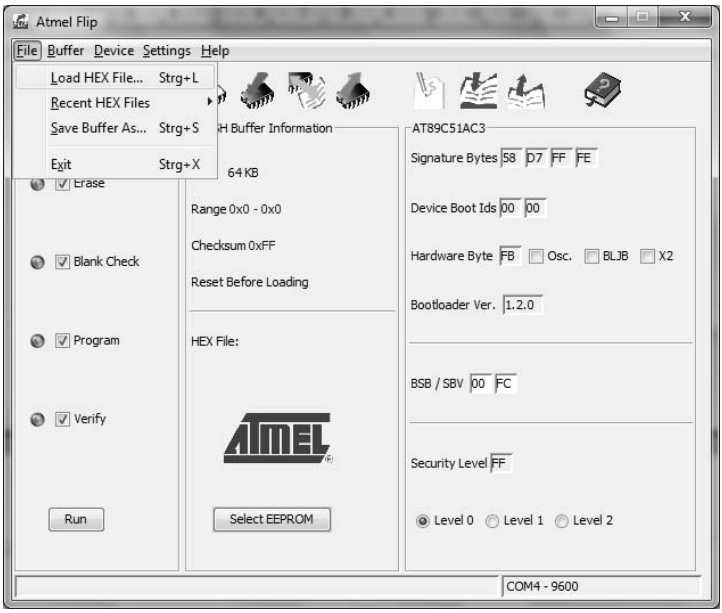
b) Verbindung zum Controller über die serielle Schnittstelle herstellen:

Hinweis: Es kann auch ein USB-Seriell-Adapter verwendet werden.



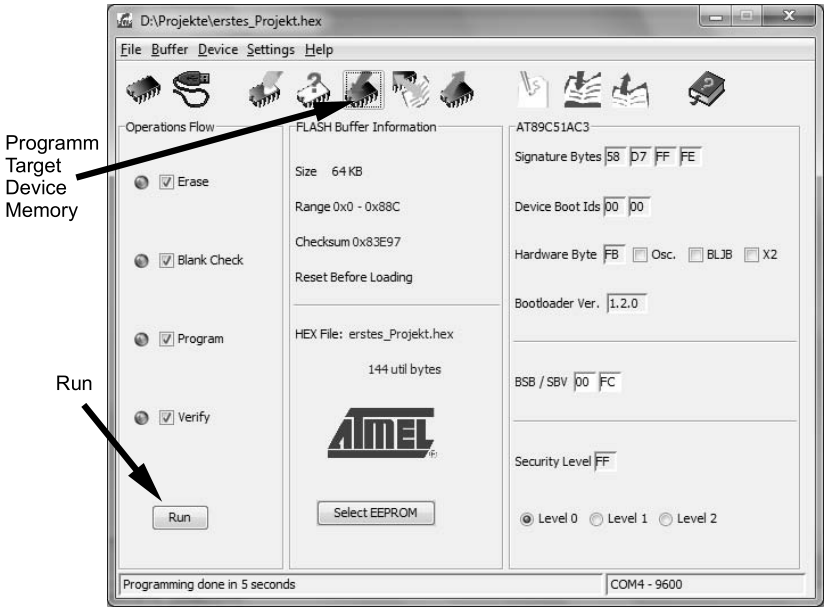
Hierzu den Controller in den Programmiermodus stellen und ein „Reset“ am Controller ausführen.

c) Anschließend unter „File“ → „Load HEX File“ den erzeugten HEX-File laden.



d) Einstellungen des Hardwarebytes kontrollieren (rechte Programmseite).

e) Über „Run“ oder „Programm Target Device Memory“ den Hex-File übertragen.



f) Den Controller in den Run-Modus stellen, und ein Reset ausführen.

Literatur- und Quellenverzeichnis

Bücher

Roth, Andreas
Das Mikrocontroller Kochbuch
IWT-Verlag, Vaterstetten b. München

Müller, Helmut/Walz, Lothar
Mikroprozessortechnik
Vogel Buchverlag, Würzburg

vom Berg, Bernd/Groppe, Peter/Klein, Joachim
C-Programmierung für 8051er
1: Der Einstieg
2: Die ON-Chip-Peripherie
3: Externe Peripherie
Elektor-Verlag, Aachen

Dansmann, Manfred/Goll, Joachim/Bröckl, Ulrich
C als erste Programmiersprache
Teubner Verlag, Stuttgart

Link zum Wägeverfahren bei der A/D-Wandlung:
http://groups.uni-paderborn.de/cc/arbeitsgebiete/messtech/simulationen/ad/waege/waege.html

Datenblätter

Siemens/Infineon
Microcomputer Components:
8-Bit CMOS Single-Chip Microcontroller SAB 80C517/80C537
Data Sheet 04.95
http://www.infineon.de

Atmel
Enhanced 8-bit Microcontroller with 64KB Flash Memory AT89C51AC3
http://www.atmel.com

Hitachi semiconductor
Datasheet HD44780U (LCD-II)
http://www.renesas.com/eng/

ST- SGS-Thomson Microelectronics
Datasheet ULN2803A
September 1997
http://www.st.com

Motorola SEMICONDUCTOR
Datasheet MC 14520B
http://www.motorola.com

Maxim
Datasheet RS232
19-4323; Rev 16; 7/10
http://www.maxim-ic.com

Software

Keil µVision4
http://www.keil.com

Atmel Flip
http://www.atmel.com

Weitere Compiler/Entwicklungsumgebungen

http://www.engelmann-schrader.de

http://www.wickenhaeuser.de

http://www.phytec.de

Passender Bausatz zum Buch

http://www.boeckerboards.de

Weitere Hardware/Bausätze

http://www.rakers.de/

http://www.elektor.de

http://www.palmtec.de

http://www.engelmann-schrader.de

Sachwortverzeichnis

Schaaf · Böcker

Mikrocomputertechnik

Dieses Lernbuch der Technik vermittelt ein breites Basiswissen über Mikrocontroller sowie über deren Beschaltung zum Mikrocomputer. Nach Durcharbeiten des Buches können Sie Controller-Boards für die Automatisierungstechnik und auch Entwicklungssysteme zur Programmentwicklung konstruieren.

Hardware und Programmentwicklung sind eng miteinander verzahnt. Sie lernen zu jeder Funktionseinheit des Controllers die Befehle zur Programmierung kennen und schreiben praxisbezogene Anwenderprogramme.

Das Buch wendet sich an Schüler von Fachschulen und Studierende an Fachhochschulen, aber auch an interessierte Elektroniker, die sich auf dem Gebiet der Mikrocontrollertechnik weiterbilden wollen.

Das komplexe Wissen über Mikrocontroller wird in diesem Lernbuch in kleine, aufeinander aufbauende Einheiten gegliedert. Beispiele und Übungen festigen die gelernten Inhalte und erlauben eine Selbstkontrolle. Die didaktische Aufbereitung ist das Ergebnis langjähriger Unterrichtspraxis. Konzeptionelles Ziel des Buches ist leichtes Lernen und Freude am Lernerfolg.

Die neue 6. Auflage erläutert die 8051-Architektur jetzt an einem Atmel Controller und erläutert die einzelnen Funktionen und Programmierungen nun in der Programmiersprache C. Zum besseren Verstehen von Controlleraufbau und -programmierung sind Einzelheiten zur Assemblerprogrammierung nach wie vor enthalten. Ein neues Kapitel zeigt, wie Programmabläufe umgesetzt und technische Probleme strukturiert gelöst werden können.

Die Autoren:
OStR Dipl.-Ing. Bernd-Dieter Schaaf
StR Dipl.-Ing. Stephan Böcker
Heinz-Nixdorf-Berufskolleg Essen

Aus dem Inhalt:

- Mikrocomputer und Mikrocontroller
- Externe und interne Speicherorganisation
- Konstruktion eines Controllerboards
- Programmentwicklung und C-Programmierung
- C-Programme für Controller-Grundfunktionen
- Controller Erweiterungen
- Der Zähler/Zeitgeber Timer 0 und Timer 1
- Der Analog/Digital-Wandler
- Die serielle Schnittstelle
- Das Interrupt-System
- Assemblerprogrammierung
- Der Befehlssatz der Controller-Familie 8051
- Controller-Grundfunktionen und Programmierung von Controller-Erweiterungen in Assembler

HANSER

www.hanser-fachbuch.de

€ 29,90 [D] | € 30,80 [A]

ISBN 978-3-446-43078-5