Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Fakultät Elektro- und Informationstechnik

Skript zur Vorlesung

Mikrocontrollersysteme

Prof. Dr.-Ing. G. Schäfer

Stand 01.11.2012

Inhalt

1 Einführung 5

1.1 Geschichtliche Entwicklung 7
1.2 Einsatzgebiete 13

2 Systemkonzept des Mikrocontrollers 22

2.1 Funktionseinheiten 23
2.1.1 Rechenwerk 25
2.1.2 Steuerwerk 26
2.1.3 Speicher 27
2.1.4 Ein- /Ausgabe 29
2.2 BUS-Struktur 30
2.2.1 Datenbus 32
2.2.2 Adressbus 32
2.2.3 Bustreiber 34
2.3 Organisation und Arbeitsweise der Zentraleinheit 38
2.4 Steuerung der Speicher- und Ein-/Ausgabebausteine 55
2.4.1 Multiplex-Bussysteme 55
2.4.2 Schreib/Lese-Steuerung 57
2.4.3 Memory Mapped I/O 60

3 Der Mikrocontroller 8051 61

3.1 Modell des 8051 61
3.1.1 Register 66
3.1.2 Programmspeicher 69
3.1.3 Datenspeicher 70
3.2 Befehlssatz 77
3.2.1 Struktur der Befehle 77
3.2.2 Adressierungsarten 78
3.2.2.1 Implizite Adressierung (Implied Addressing) 78
3.2.2.2 Direkte oder absolute Adressierung (Direct Addressing) 79
3.2.2.3 Unmittelbare Adressierung (Immediate Addressing) 79
3.2.2.4 Indirekte (Indirect) Adressierung 80
3.2.2.5 Indizierte Adressierung (Indexed Addressing) 82
3.2.2.6 Relative Adressierung 82
3.2.3 Beschreibung der Befehle 83
3.2.3.1 Arithmetische Befehle 84
3.2.3.2 Logische Befehle 85
3.2.3.3 Verschiebebefehle 86
3.2.3.4 Datentransfer-Befehle 87
3.2.3.5 Befehle zur Bitverarbeitung (Boole'sche Operationen) 89
3.2.3.6 Verzweigungs- und Sprungbefehle 90

4 Programmiergrundlagen 96

4.1 Randbedingungen 96
4.2 Erstellen und Testen von Quellcodes 97
4.3 Darstellung von Struktogrammen 99
4.4 Ablauffolgediagramme 100
4.5 ASSEMBLER-Programmierung 102
4.5.1 Assembleroperatoren 103
4.5.2 Assembler Direktiven 104
4.5.3 Programmbeispiele 107
4.5.4 Unterprogrammtechnik 119
4.5.4.1 Datenverwaltung 120
4.5.4.2 Unterprogrammablauf 121
4.5.5 Verarbeitung externer Ereignisse 123
4.5.6 Bearbeitung mehrerer Interruptquellen 134
4.5.7 Zeitgeber 137
4.5.7.1 Funktion der Timer 138
4.5.7.2 Timeranwendungen 145
4.6 Allgemeine Programmiergrundlagen in C 150
4.6.1 Datentypen 150
4.6.1.1 Skalare Datentypen 150
4.6.1.2 Feldtypen 151
4.6.1.3 Verweise 151
4.6.2 Typische Abläufe in Mikrocontrollerprogrammen 153
4.6.2.1 Bitoperationen 153
4.6.2.2 Arbeiten mit Feldern 155
4.6.2.3 Schleifen 156
4.6.3 Aufgaben zur Selbstkontrolle 158
4.7 Programme in C für den Mikrocontroller 159
4.7.1 Anforderungen 159
4.7.2 Verwendung der Datentypen in C 160
4.7.3 Angabe von Speicherbereichen 161
4.7.4 Definition von Interrupt Service Routinen 162
4.7.5 Anwendungsbeispiel 162

5 Parallele Ein-/Ausgabe 165

5.1 Aufbau der Ports 165
5.2 Betrieb der Ports des C8051F340 170
5.3 Matrixtastatur 176
5.4 Entprellung von Tasten 181

6 Serielle Datenübertragung 184

6.1 Grundlagen 184
6.2 Serielle Schnittstellen des Prozessors C8051F340 191

7 I²C Bus 202

7.1 Grundstruktur 202
7.1.1 Physikalische Signale 203
7.1.2 Ablauffolgen 205
7.1.3 Masterrealisierung mittels Software 207
7.2 Verwendung des I²C Bus mit dem C8051F340 212

8 Vergleich C-Code –Assembler 224

8.1 Arithmetische Operationen 224
8.1.1 Byte Verarbeitung 224
8.1.2 Word Verarbeitung 229
8.1.3 Integer Verarbeitung 234
8.1.4 FLOAT Verarbeitung 235
8.2 Einsatz von Pointern 238
8.2.1 Pointer auf Byte 238
8.2.2 Verwendung anderer Datentypen 244
8.3 STRING Verarbeitung 245
8.4 Schleifenkonstruktionen 248
8.4.1 While Schleifen 248
8.4.2 Do…While Schleifen 250
8.4.3 For Schleifen 252
8.4.4 Unterprogramme 255
8.5 Zugriff auf feste Adressen 270

9 Echtzeitsysteme 271

9.1 Anforderungen 271
9.2 Konzepte 271
9.2.1 Foreground/Backgroundsysteme 271
9.2.2 Betriebssysteme 277
9.2.3 Round Robin Verfahren (Time Slicing) 280
9.3 Verwendung eines einfachen Echtzeitsystems (HKRO) 281
9.3.1 Einführungsbeispiel 282
9.3.2 Prioritäten 286
9.3.3 Wartebedingungen 287
9.3.3.1 Messages 288
9.3.3.2 Delays 289
9.3.3.3 Semaphore 293
9.3.3.4 Interrupts 298
9.3.4 Unterprogrammaufrufe 303
9.3.4.1 Eingeschränkte Unterprogrammverwendung 303
9.3.4.2 Reentrant Funktionen 303
9.3.4.3 Prioritätsinversion 305
9.3.5 HKRO Funktionen 307
9.3.5.1 Tabelle der HKRO Funktionen 307
9.3.5.2 Beschreibung der HKRO Funktionen 308

10 Literatur 313

1 Einführung

Die Verarbeitung von Informationen spielt bei der Entwicklung von Systemen in allen Bereichen eine besondere Rolle. Die dazu notwendige Basis stellt in der Regel ein digitales Rechnersystem dar. Welche Leistung das Verarbeitungssystem haben muss und welche Datenströme verarbeitet werden sollen, ist stark von der Anwendung abhängig. Einen bedeutenden Teil der Anwendungen können im weitesten Sinne dabei in Bereiche angesiedelt werden, bei denen die Anforderungen an die Verarbeitungsgeschwindigkeit und die Anzahl der benötigten Datenkanäle keine zu hohen Anforderung erfüllen müssen (z.B. in Bereichen der Automatisierungstechnik, Haushaltsgeräte …). Mikrocontroller stellen in diesem Bereich eine flexible und kostengünstige Lösung dar. Die angebotenen Prozessoren decken dabei, angefangen mit einfachsten Prozessorarchitekturen bis hin zu komplexen Rechnerstrukturen, einen großen Bereich ab.
Im Vordergrund stehen dabei auch Stromverbrauch und Baugröße, die Anzahl und Art der Ein- und Ausgänge sowie Schnittstelleneinheiten, die oft verwendete Datenübertragungsprotokolle schon direkt verarbeiten können und damit den Prozessor entlasten.
Mikrocontroller werden in einer Vielzahl von Systemen eingesetzt, in der Rechenleistung benötigt wird um bestimmte Funktionen zu realisieren, von denen der Benutzer aber zunächst keine Kenntnis hat, dass daran ein Mikrocontroller beteiligt ist. Diese Systeme werden als eingebettete Systeme oder „Embedded Systems“ bezeichnet. Sie stellen den überwiegenden Teil der Mikrocontrolleranwendungen dar.
Wegen der von der Industrie gestellten Forderungen werden ständig neue Prozessoren entwickelt, die zusätzliche Funktionen bereitstellen oder die Leistungsfähigkeit verbessern und dabei kostengünstiger sind. Als Grundlage dienen hierzu oft bereits vorhandene Systeme, die auf der Grundlage eines vorhandenen Prozessorkerns erweitert werden. Komplette Neuentwicklungen sind eher selten.

Eine weitere Entwicklung von prozessorbasierenden Systemen findet im Bereich der programmierbaren Digitalbausteine statt (FPGA). FPGAs werden mit Hilfe von Hochebenenbeschreibungen z.B. VHDL programmiert. Sollen Prozessorsysteme mit variabler Peripherie entwickelt werden, so können sowohl vom Prozessor als auch von der Peripherie VHDL Beschreibungen erstellt werden und als Hardwarekonfiguration des FPGAs synthetisiert werden. VHDL- Beschreibungen von gängigen Prozessoren können entweder als geschützter Code käuflich erworben oder unter der Beachtung der Lizenzbedingungen frei zugängliche Versionen verwendet werden. Eine solche Vorgehensweise wird im Bereich der schnellen Prototypentwicklung (Rapid Prototyping) verwendet um in kurzer Zeit Systeme zum Testen entwickeln zu können.

Im vorliegenden Skript soll auf ein fest vorliegendes System der Firma Silicon Labs mit dem Prozessor F340 zurückgegriffen werden. Der Prozessor basiert auf der Architektur des Prozessors 8051, der ursprünglich von der Firma Intel entwickelt wurde.
Dieser Prozessortyp ist neben ARM-Architekturen und PIC-Prozessoren einer der am meisten eingesetzten Prozessortypen, besonders im Bereich der eingebetteten Systeme.
Es wird zunächst die grundsätzliche Entwicklung von Prozessorsystemen vorgestellt und auf die Hardwarestruktur eines solchen Systems eingegangen. Darauf aufbauend erfolgt die Beschreibung der Hardwarearchitektur und des Befehlssatzes des bereits vorgestellten Prozessors F430. Die Entwicklung, der für eine Applikation benötigten Programme wird auf der Grundlage der Assembler und der C-Programmierung durchgeführt. Es werden dabei typische Programmstücke zur Bearbeitung von Feldern, die Handhabung von Tasten, die

Programmierung von Zeitanforderung mit Hilfe von Timern und die Verwendung der seriellen Schnittstelle angesprochen. Zusätzlich werden die Auswirkungen der C-Programmierung auf die Güte des erstellten Codes durchleuchtet, die von unbedarften Programmierern gerne unterschätzt wird. Den Abschluss bildet ein Echtzeitbetriebssystem, das speziell für die Anwendung in der Lehre entwickelt wurde und die prinzipielle Vorgehensweise bei der Verwendung eines solchen Systems zeigen soll (Taskbeschreibung, Semaphore, Messagesystem, ...).
Ergänzt wird die Vorlesung durch ein Labor, das die gewonnen Kenntnisse in der Praxis festigt.

Grundsatzaussagen:

- Mikrocomputer und –controller sind in ihrer Grundstruktur Rechner im üblichen Sinne.

- Als Haupteinsatzgebiet ist jedoch die Anwendung als Steuerungseinheit z.B. in der Automatisierung zu sehen.

- Der Einsatz eines Mikrocontrollers in einem Gerät ist meist nicht direkt erkennbar (versteckte Betriebsweise). Die Verwendung von Rechenleistung ist indirekt nur an der Intelligenz und dem Umfang der Funktionen sichtbar.

Obwohl das Verständnis über die Art der Basiseinheiten in einem Mikrocontroller unterschiedlich ist, so kann doch die folgende Definition eines Mikrocontrollers als allgemein angesehen werden:

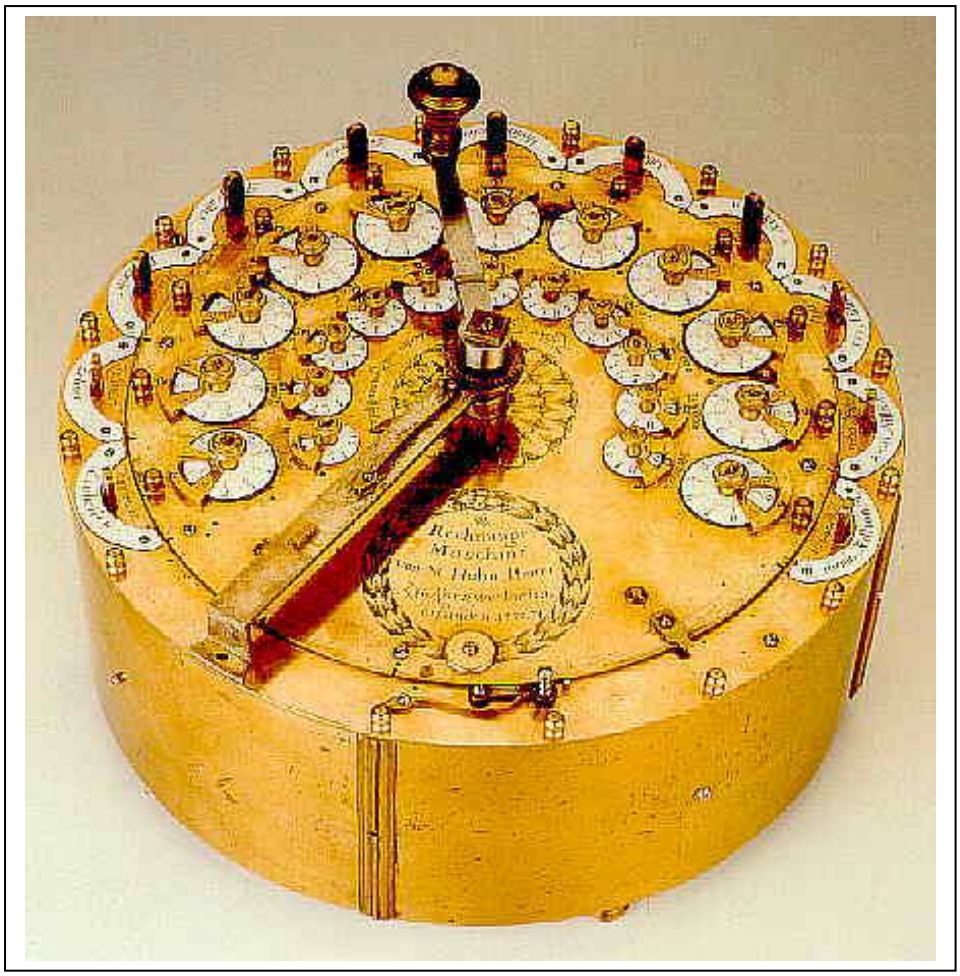
Mikrocontroller =

Einheiten eines Digitalrechners
+
Peripherieeinrichtungen

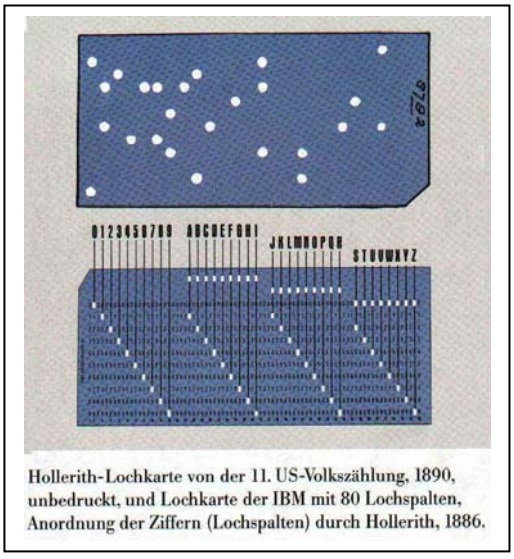
auf einem Chip

1.1 Geschichtliche Entwicklung

1770 1. Brauchbare Rechenmaschine von Hahn [GE1]
(Addition, Subtraktion, Multiplikation und Division)

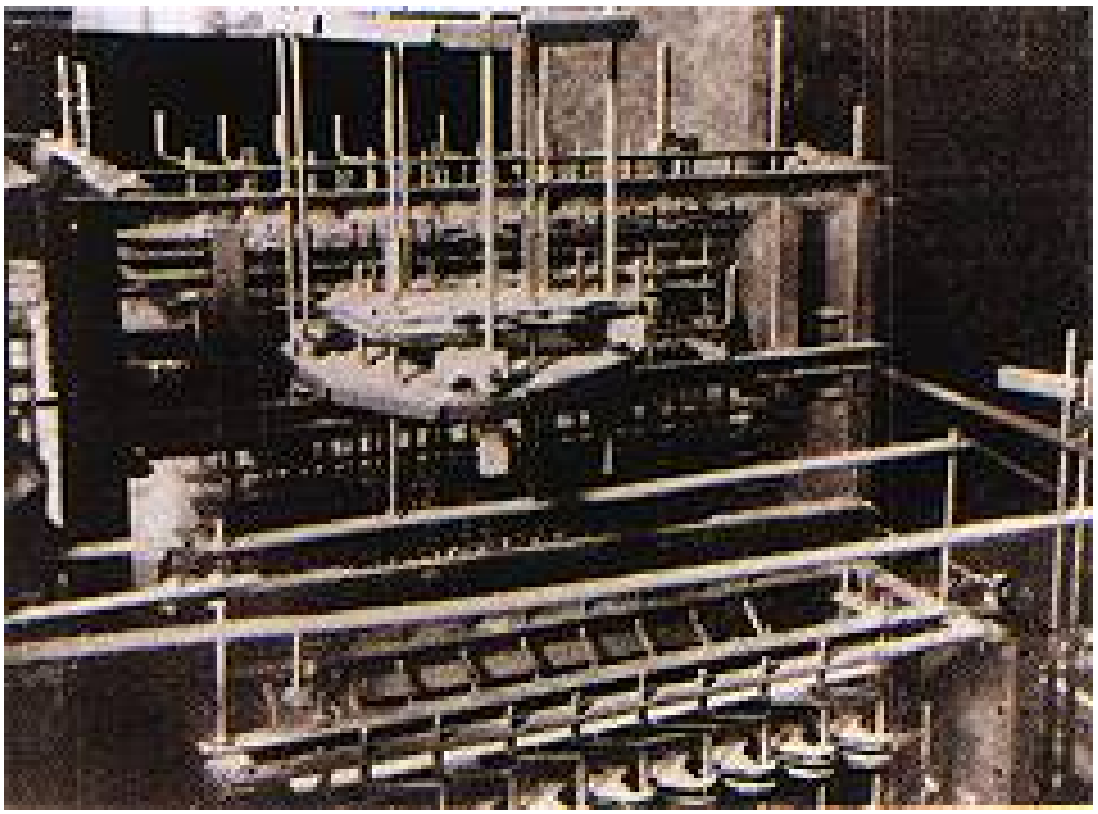


1882 Lochkartenmaschine von Hollerith [GE2]
Speicherung von Daten auf Lochkarten

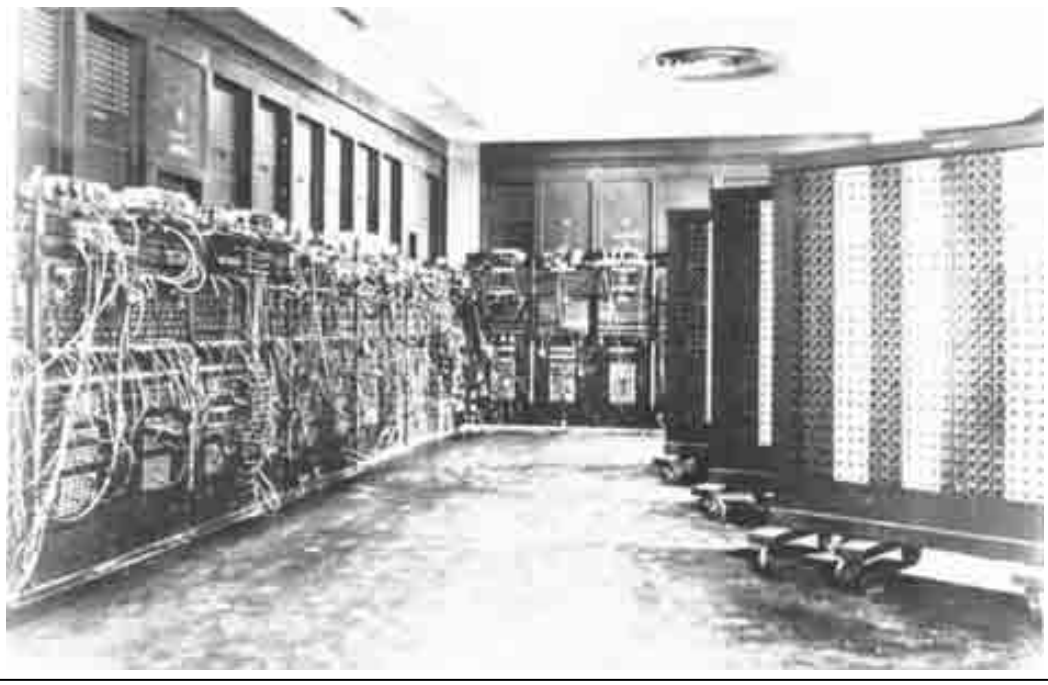


Hollerith-Lochkarte von der 11. US-Volkszählung, 1890, unbedruckt, und Lochkarte der IBM mit 80 Lochspalten, Anordnung der Ziffern (Lochspalten) durch Hollerith, 1886.

1941 1. Funktionsfähige programmgesteuerte Rechenanlage in Relaistechnik von Zuse Z1 [GE3]
20 Additionen pro Sec.,Speicherkapazität: 1408 Bit, 2600 Relais,



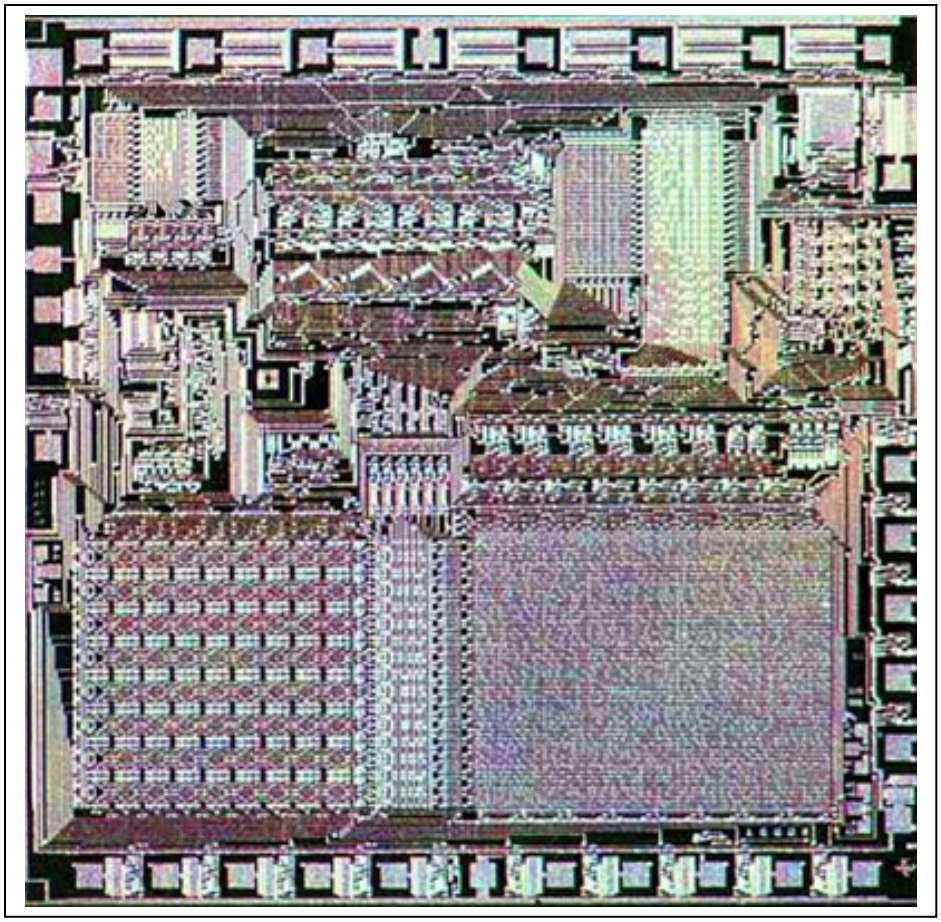
1946 1. Röhrenmaschine ENIAC [GE4]
(Die 1. Generation hatte viele Ausfälle und war sehr temperaturempfindlich) 18000 Röhren**,** 1000 Additionen pro Sec.



1958 Transistorenrechner von SIEMENS (2002) und TELEFUNKEN TR4(2. Generation)[GE5]

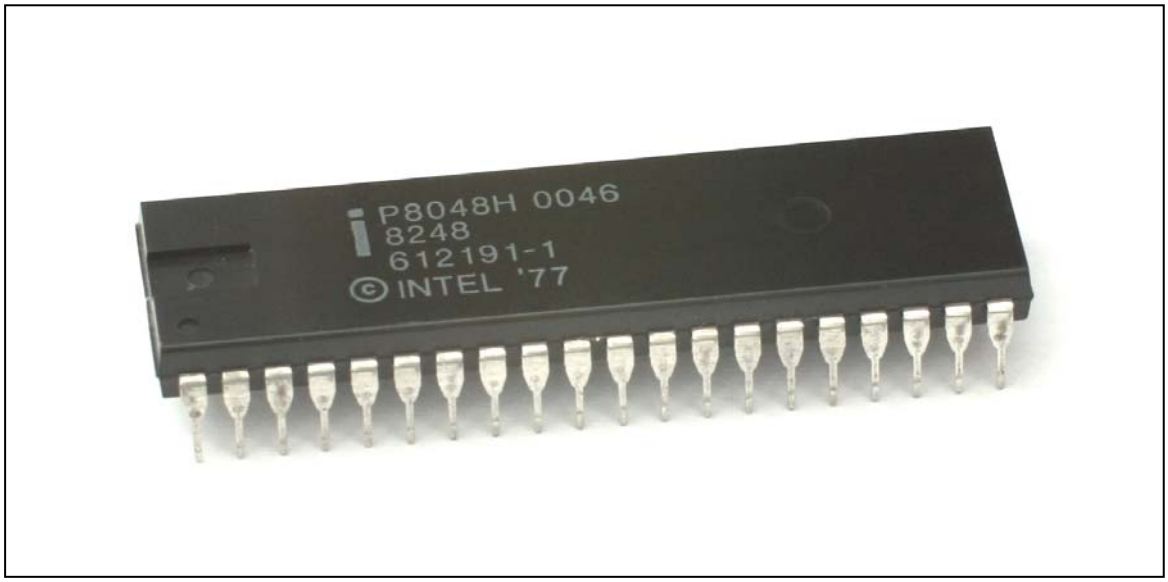
1966 Monolithische Integration [GE6]
(3. Generation) IBM 360, Siemens 4004

1971 4-Bit Mikroprozessor TMS1000 von Texas Instruments [GE7]



1974 INTEL stellt den 1. Mikrocontroller her (Labormuster)
(2. industrielle Revolution) [GE8]

1976 Verkauf des 8-Bit Mikroprozessors 8048



1980 Intel 8-Bit Prozessor 8051 [GE9]

1985 16-Bit Mikroprozessor 8096 von INTEL [GE10]

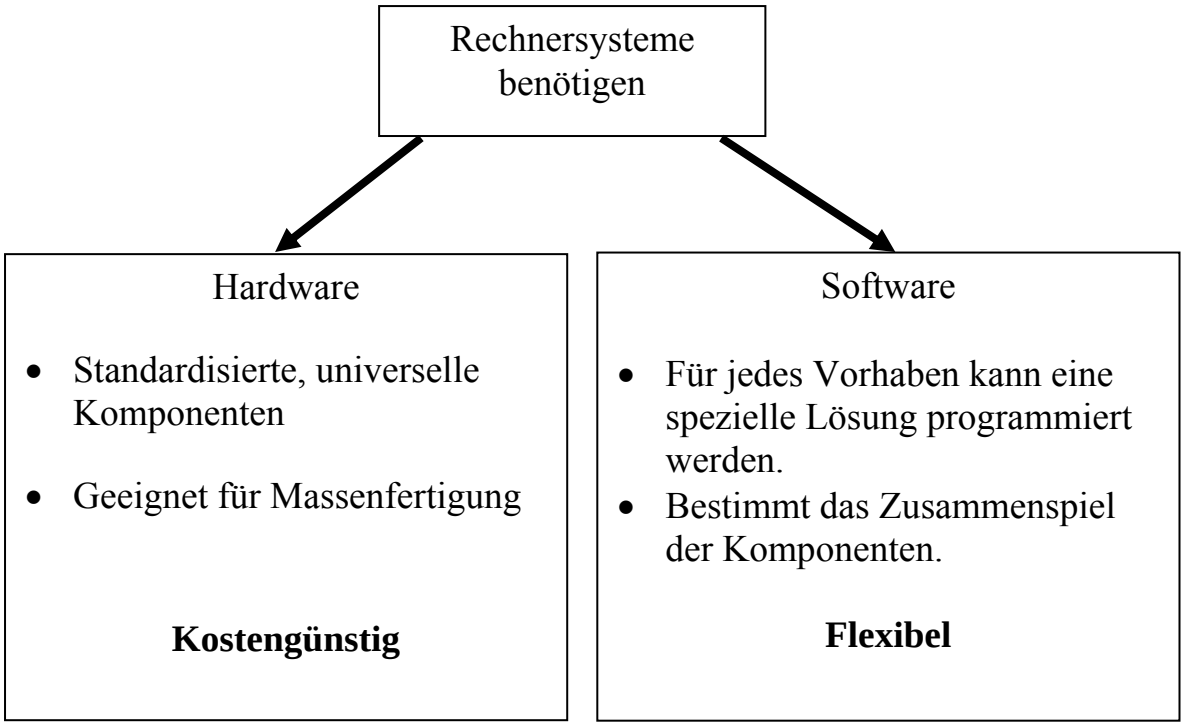
....

Die Weiterentwicklungen änderten nichts mehr an der prinzipiellen Technologie bei der Herstellung von Prozessoren. Die Leistungsfähigkeit, das Speichervermögen und die zur Verfügung stehenden Funktion übersteigen bei Weitem die bisher vorgestellten Prozessoren.
Ein gutes Indiz für die Fähigkeiten der Prozessortechnik im Hinblick auf eine breite Anwendung sind die Angaben in den Prospekten der PC-Discounter. Dieser Leistungsbereich liegt jedoch weit oberhalb der Anwendungsbereiche für ein Rechnersystem, für das Mikrocontroller eingesetzt werden.

[GE1]	http://www.rechenhilfsmittel.de/rmhahn.jpg,	31.08.11
[GE2]	http://privat.swol.de/svenbandel/Hollertith2.jpg,	31.08.11
[GE3]	http://www.weller.to/his/img/zuse_z1.jpg,	31.08.11
[GE4]	http://www.at-mix.de/images/glossar/eniac1.jpg,	31.08.11
[GE5]	http://upload.wikimedia.org/wikipedia/commons/thumb/5/53/ Telefunken-tr4.jpg/220px-Telefunken-tr4.jpg,	31.08.11
[GE6]	http://static.nol.hu/media/picture/ 92/27/00/000002792-3500-330.jpg,	31.08.11
[GE7]	http://www.zdnet.co.uk/i/z5/illo/nw/story_graphics/10dec/ intels-victims/intelvics-tms-1000-texasinstruments.jpg,	31.08.11
[GE8]	http://upload.wikimedia.org/wikipedia/commons/2/2c/ KL_Intel_P8048H.jpg,	31.08.11
[GE9]	http://www.rcs.hu/roboshop/Microrobot/js8051a1cpu.htm,	31.08.11
[GE10]	http://cpucollection.ca/IntelN8096BH.jpg,	31.08.11
[GE11]	http://www.technikimbuero.at/Museum/Computer.htm,	31.08.11
[GE12]	http://www.efton.sk/t0t1/history8051.pdf	31.08.11

1.2 Einsatzgebiete

Die universelle Anwendungsmöglichkeit von Mikrocontrollersystemen erklärt sich aus der Kombination einer standardisierten Hardware und einer problemspezifischen Software.



Die Abdeckung von Problemlösungen in extremen Bereichen ist nur bedingt möglich z.B.:

- Hochfrequenzanwendungen (Front-Ends)
- Direkte Verarbeitung von schnellen analogen Signalen
- Schnelle Regelungen

Anwendungsgebiete:

Die im Folgenden aufgeführten Einsatzgebiete sind nur als Stellvertreter des breiten Anwendungsbereiches von Mikrocontrollern zu sehen [BLE94]. Die Grenzen der Anwendungen beispielweise zu Signalprozessoranwendung sind dabei fließend. Über die Komplexität des Codes oder der benötigten Datenspeicherkapazität werden keine Angaben gemacht.

1. Kfz Elektronik
   Motormanagement, Airbagsystem, ABS, ESP, Diagnose Computer, Bremsassistent usw.

2. Medizin
   Patientenüberwachung, Tomographie, Herzschrittmacher

3. Industrie
   Prozesskontrolle, Antriebsregelung, NC-Maschinen, Messwerterfassung, Robotertechnik, Zeiterfassung

4. Netzanbindung
   CAN-Bus, I2C, RS232, Ethernet, Intelligente Sensoren

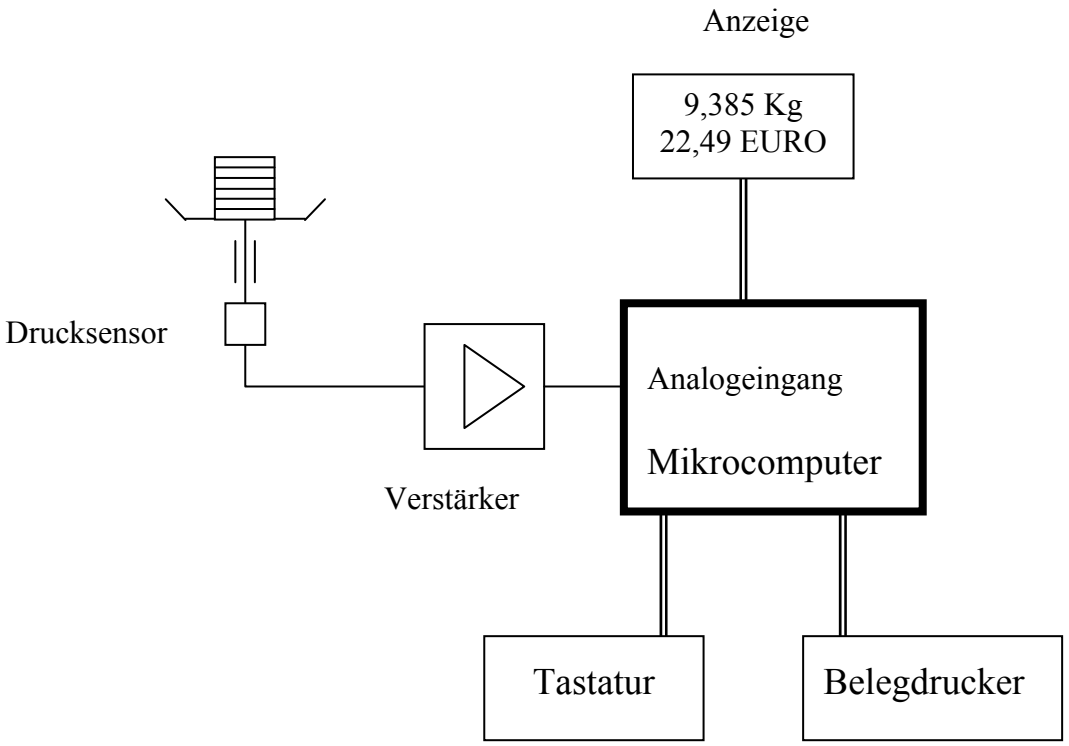
5. Verkehr
   Autopilot, Navigationssysteme, Positionsbestimmung (GPS) Verkehrserfassung.

6. Haushalt
   Waschmaschine, Fernsehgerät, Videorecorder, Kameras, Heizungsregelung, Spielecomputer, Waagen.

Beispiel eines Mikrocontrollersystems

Waage - Kasse - Drucker Verbundsystem eines Metzgerladens:

In einem typischen mikrocontrollerbasierenden System wird die Verarbeitungseinheit mit ihrer Fähigkeit komplexe Verknüpfungen herzustellen, mit Komponenten verbunden, die Daten aus der Umwelt aufnehmen und ebenso Daten für die Umwelt zur Verfügung stellen. Informationsquellen und/oder -senken stellen dabei nicht nur Menschen, sondern auch andere Maschinen mit ihren Sensoren und/oder Aktoren dar [BLE94].



Aufgaben des Systems:

- Berechnung des Verkaufspreises (Gewicht × Einheitspreis)
- Drucken des Beleges
- Registrierung des Kassenbestandes
- Schnelles Erfassen des Endgewichtes trotz Nachschwingen der Waagschale
- Justierung des Nullpunktes
- Test auf Fehlerfunktion

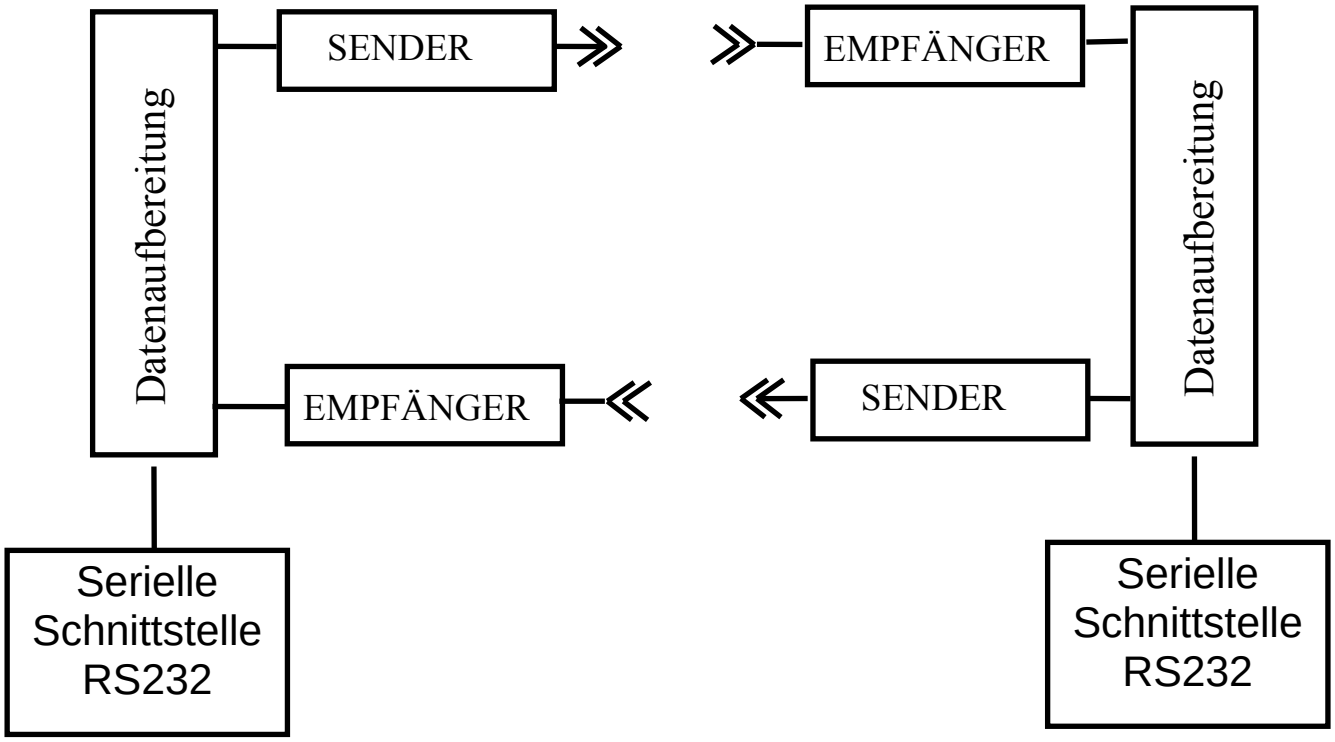
HF-Übertragungsstrecke

Am Beispiel einer HF Übertragungsstrecke soll an einem anschaulichen Anforderungsprofil der vereinfachte Entwicklungsprozess zur Definition einer Systemkonfiguration gezeigt werden.

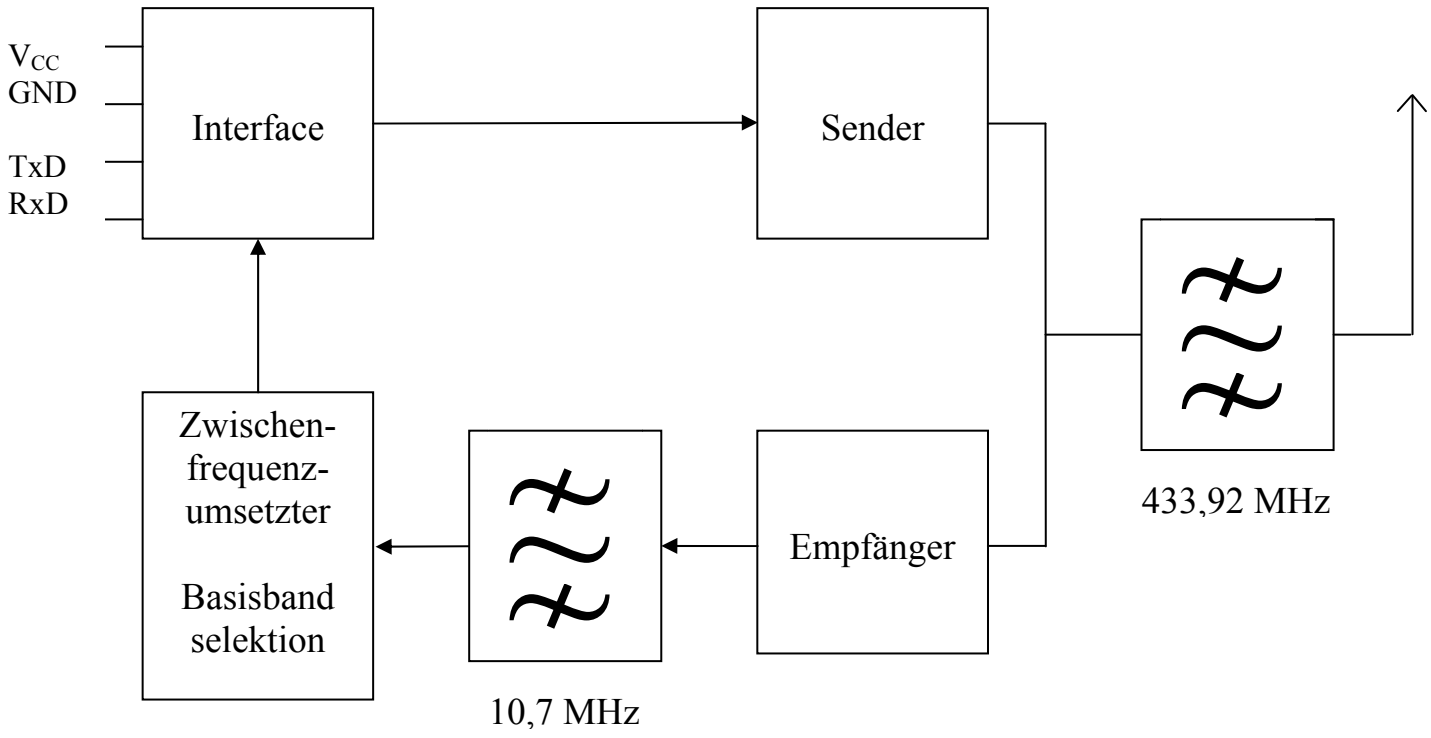
Gestellte Anforderungen:

- 100 Kbits/s Datenrate
- Serielle Datenübertragung (RS232)
- Datenübertragung in beide Richtungen soll möglich sein
- Übertragungsverhalten wie bei einem Kabel
- Sender-/ Empfängeridentifizierung zum Aufbau unterschiedlicher Verbindungen.
- Es soll eine Low-Cost-Lösung angestrebt werden.

Neben den eigentlichen Anforderungen existieren auch Randbedingungen, die vom Entwicklungsumfeld abhängig sind: z.B. Entwicklungssysteme oder, wie in diesem Fall angenommen, ein vorhandenes Übertragungsmodul. Eine erste Systemstruktur ergibt sich meist intuitiv aus dem Aneinandersetzen essentieller Verarbeitungsschritte, die auf Erfahrungswerten beruhen. Die Auswahl einer optimalen Lösung hinsichtlich Kosten und Funktion bei mehreren Alternativen ist an dieser Stelle entscheidend für den späteren Erfolg des Systems als Produkt.



Blockschaltbild des Transceiverbaustein geeignet für den Halbduplexbetrieb mit Datenraten bis ca. 38,4 Kbits/s.



Die geforderten Leistungsdaten sind offensichtlich mit dem vorhandenen Transceiverbaustein alleine nicht zu erfüllen. Die Alternative, einen anderen Baustein zu verwenden, soll an dieser Stelle aus Kostengründen und wegen Entscheidungen im Management nicht zur Verfügung stehen. Hieraus ergibt sich die Frage: Sind die gestellten Forderungen mit zusätzlicher Rechenleistung zu erfüllen und sind sie möglichst günstig realisierbar?
Ausgefeilte Methoden zur Abschätzung der Möglichkeiten stellt das Verfahren des Quality Function Deployment dar, das hier den Rahmen der Vorlesung sprengen würde. Eine einfache Machbarkeitsanalyse soll jedoch das prinzipielle Vorgehen darstellen.

Machbarkeitsanalyse
Aufteilung der Funktionalität in Hardware und Software

Systemanforderungen			
Mindestens Halbdublexbetrieb	++	++	++
Datenübertragungsrate 100kBits/s	--	--	+
Anschluss von Peripheriegeräten	+	++	++
Unterstützung einfacher Netzwerkprotokolle	0	+	+
Einstellbare Verbindungspartner	--	+	+
Punkt zu Punkt Verbindung über etwa 20m	+	++	++
Nachbildung eines seriellen Anschlusses RS232	+	+	++
Keine Zusatzsoftware beim Sender und Empfänger	0	+	+
Störeinflüsse wie beim Kabelbetrieb	--	+	++

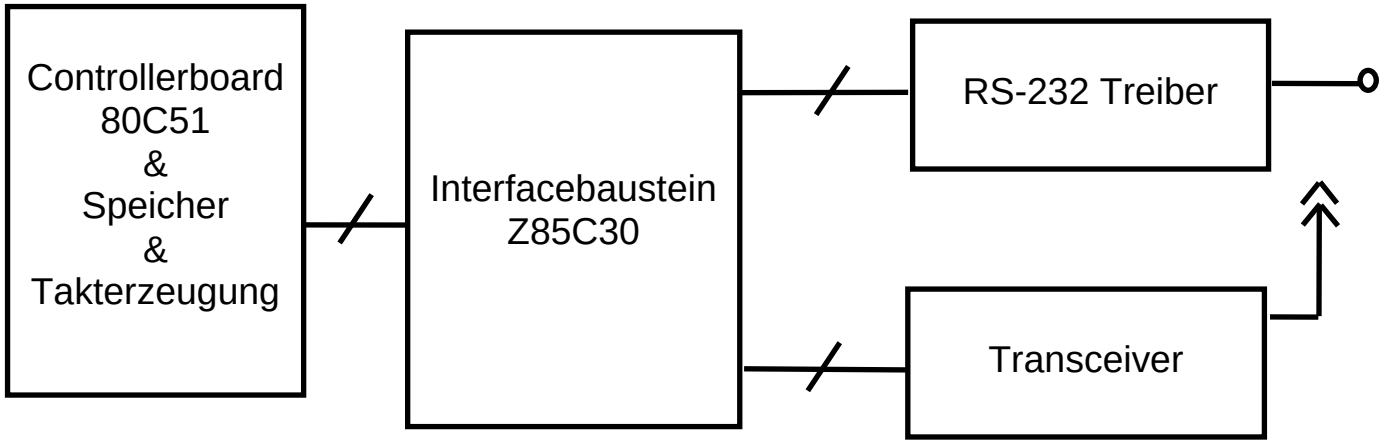
Technische Realisationsalternativen
Transceiverbaustein mit 38. 4kbits/s
Transceiverbaustein & Prozessor
Transceiverbaustein & Prozessor & Schnittstellenbaustein

Die Gegenüberstellung der Alternativen in Form einer Matrix ist zum einen ein Hilfsmittel die Güte der Lösung zu bewerten, zum anderen aber auch eine Strukturierungshilfe damit alle Kriterien gleichmäßig betrachtet werden. Nach der Auswahl der gewünschten Hardwarekonfiguration müssen im nächsten Schritt die Aufgaben auf die vorhandenen Hardwareblöcke verteilt werden. Hierbei gilt es auch abzuwägen, welche Problemlösungen besser mittels eines Programms oder unter Verwendung direkter Hardwarefunktionen erstellt werden. Dargestellt sind im Folgenden:

- Das Datenflussdiagramm (nur Darstellung der Datenverarbeitung) und
- das Strukturdiagramm (Datenfluss und Hardware)

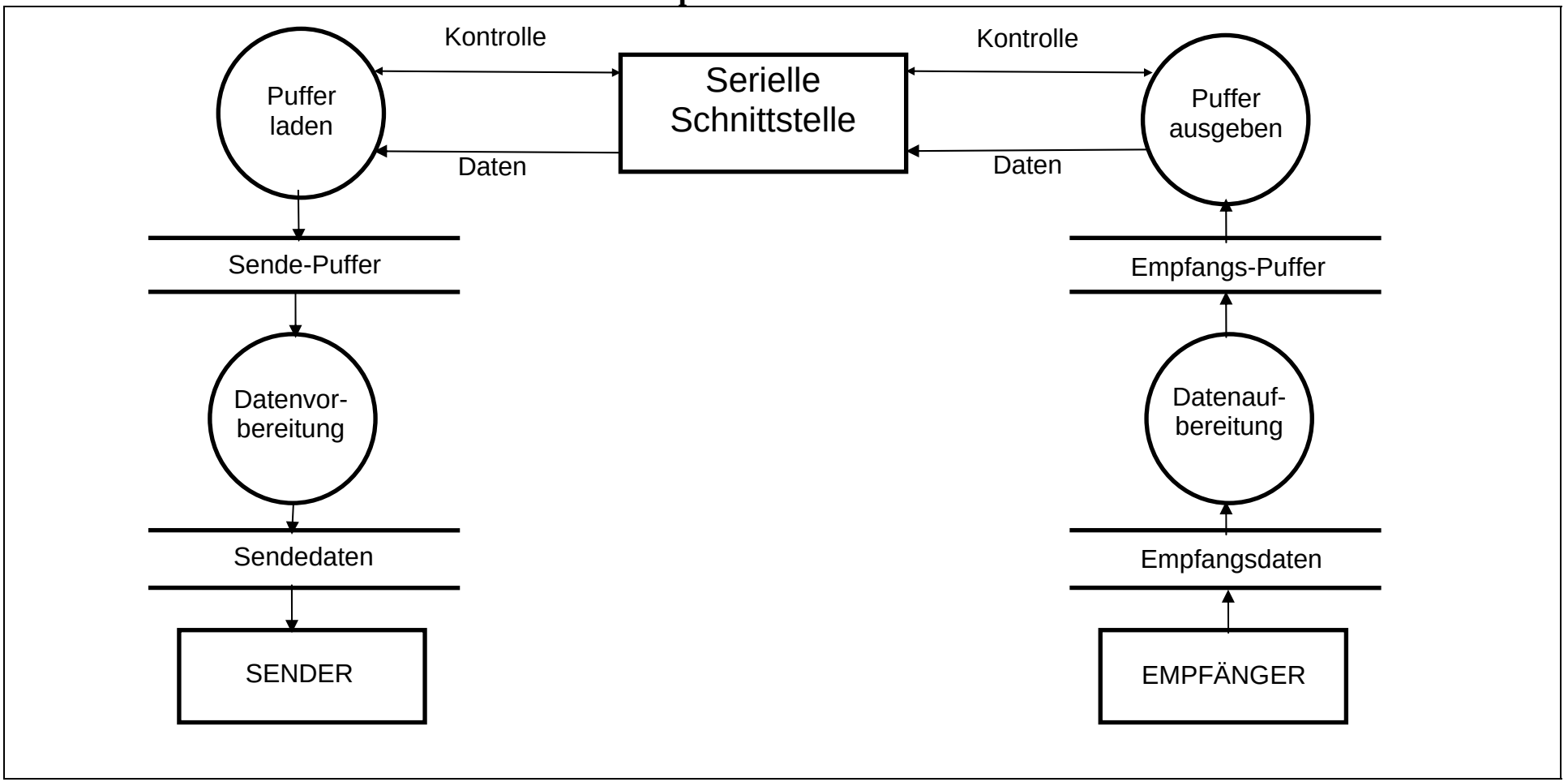
Die gezeigten Darstellungen sind aus dem Bereich Software Engineering entlehnt. Weitere Erklärungen befinden sich in Kapitel 4.

Ausgewählte Hardwarekonfiguration

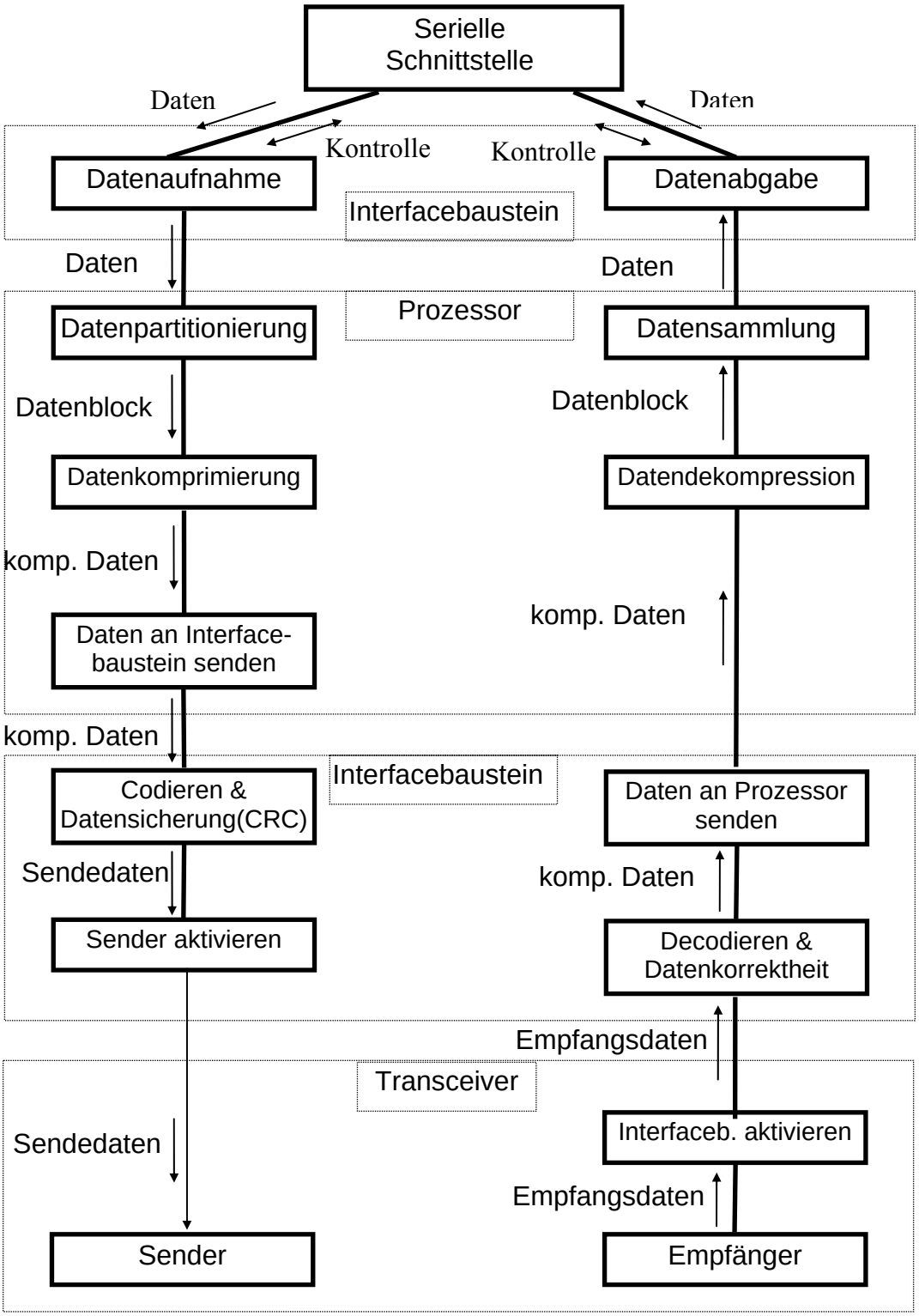


Datenflussdiagramm

Prinzipielle Funktionsweise:



Strukturdiagramm
Aufteilung der Funktionen auf die Hardwarebasis



2 Systemkonzept des Mikrocontrollers

Die Kennzeichen für die Leistungsfähigkeit eines Mikrocontrollers haben den Charakter von **Schlagworten:**

- 4-Bit, 8-Bit, 16-Bit, 32-Bit Mikrocomputer
- Anzahl der Ein-/Ausgänge
- A/D-Wandler, Auflösung
- Speichergröße, RAM, ROM
- Anzahl und Art von Timern
- Serielle Schnittstelle
- BUS-Anschlüsse/Protokolle
- PWM (Puls-Weiten-Modulation)
- Arithmetik (Multiplikation, Division, Addition, Subtraktion)
- Taktrate
- Architektur (von Neumann, Havard)
- Interruptstruktur
- Programmierung (Assembler, C)

2.1 Funktionseinheiten

Der Aufbau eines klassischen Digitalrechners ist zunächst grob mit folgenden Funktionseinheiten beschrieben:

Steuerwerk Rechenwerk	Aufbau eines klassischen Digitalrechners
Speicher Ein-/Ausgabeeinheit	

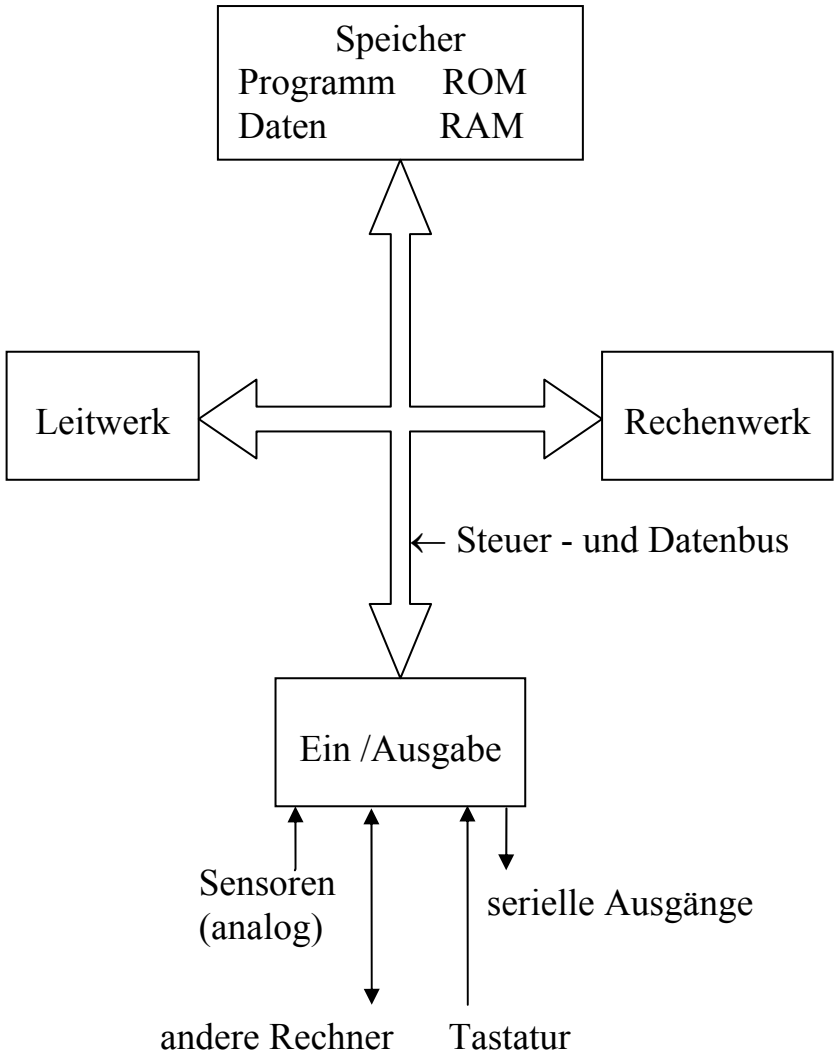
Alles auf einem Chip → **Mikrocontroller**

Je nach Aufteilung und Verwendungsmöglichkeiten der Speicher werden meist zwei Architekturvarianten unterschieden:

Gemeinsame Adressbereiche für Daten und Programm → **von Neumann Struktur**
Getrennter Bereich für Daten und Programm (meist bei Signalprozessoren) → **Havard Struktur**

Beiden Strukturen gemeinsam ist der Einsatz von BUS-Konzepten zum Austausch von Informationen zwischen den Funktionseinheiten. BUS-Strukturen erlauben einen universellen Verbindungsaufbau zwischen unterschiedlichen Einheiten. Bedingung ist jedoch, dass nur jeweils eine Einheit sendet und sich die anderen Einheiten in einem Empfangsmodus befinden. Die Organisation, wann welche Einheit senden darf, wird üblicherweise vom Leitwerk vorgenommen.

Zusammenspiel der Funktionseinheiten eines Mikrocontrollers über ein BUS-System.

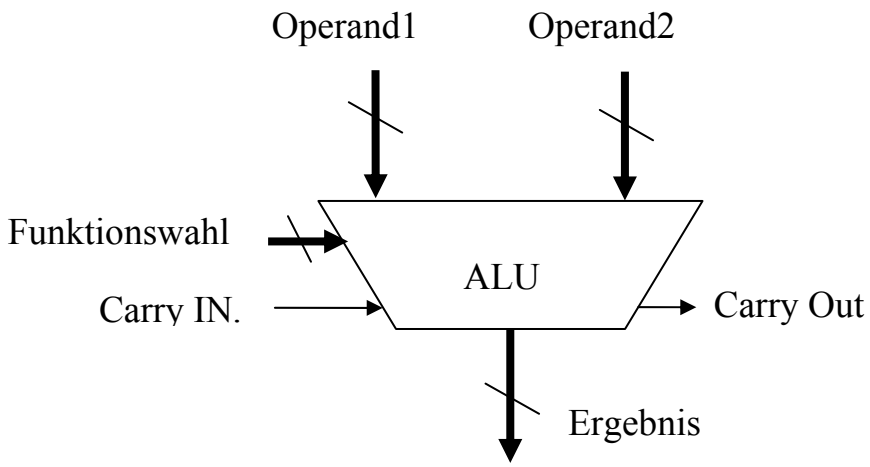


2.1.1 Rechenwerk

Unter dem Rechenwerk versteht man die Zusammenfassung von arithmetischen und logischen Funktionen

z.B. Addieren, Subtrahieren.....
Logisches ODER, UND.....

Ein Rechenwerk übernimmt meist ein oder zwei Operanden und führt die geforderte Funktion aus.
Neben dem eigentlichen Ergebnis können auch noch Zusatzinformation bereitgestellt oder zusätzlich mit einbezogen werden.
Bei einer Addition wird z.B. der Übertrag mit berechnet. Im binären Zahlensystem wird hierzu nur ein Bit benötigt. Ebenso kann hier bei der ersten Stufe ein weiteres Bit in die Rechnung mit einbezogen werden.



Funktionsauswahl						Resultat	
S_1	S_2	S_3	S_4	S_5	S_6	für $c_0 = 0$	Für $c_0 = 1$
0	0	0	0	0	0	0	1
0	0	1	0	0	0	B	B + 1
0	0	1	1	0	0	B	- B
0	1	1	0	0	0	B - 1	B
1	0	1	0	0	0	A + B	A + B + 1
1	0	0	0	0	0	A	A + 1
1	0	0	1	0	0	A - 1	A
1	0	1	1	0	0	A - B - 1	A - B
1	1	0	0	0	0	A’	- A
1	1	1	0	0	0	B - A - 1	B - A
0	0	0	0	1	0	A $\forall$ B	
0	0	0	0	0	1	A $\wedge$ B	
0	0	0	0	1	1	A $\vee$ B	

2.1.2 Steuerwerk

Basis für den Ablauf der Aktionen in einem Mikrorechner ist ein Befehl. Der Befehl enthält die Information welche Rechenoperation durchgeführt werden soll und gibt den Zielort des Ergebnisses bzw. die Herkunft, der zu verwendenden Daten an. Üblicherweise existieren eine Vielzahl von Befehlen mit unterschiedlichen Aktionen. Aufgabe des Steuerwerkes ist es nun, die vorhandenen Funktionseinheiten so zu steuern, dass die gewünschte Aktion folgerichtig durchgeführt werden kann. Zu den wichtigsten Koordinierungsaufgaben gehören folgende Aktivitäten der Baugruppen:

- Lese- / Schreibselektion einer Funktionsgruppe z.B. RAM
- Funktionsauswahl des Rechenwerkes
- Programmablauf durch Holen des nächsten Befehl über
  - Die Erhöhung des Programmzählers (+1 → Sequenz)
  - Das Setzen des Programmzählers (neue Adresse → Sprung)

Die Änderung des Programmablaufs ist als Aufgabe ebenfalls dem Steuerwerk zugeordnet. Verzweigungen werden datenabhängig durchgeführt. Als Quelle dieser Daten dient folgerichtig das Rechenwerk z.B Carry Out.

2.1.3 Speicher

Speicher dienen zur Aufnahme von Daten und Programmen.
Die Selektion eines einzelnen Datums oder Programmbefehls wird durch eine Adresse vorgenommen.

1. Das Programm wird in einem nichtflüchtigen Speicher abgelegt.
   z. B. ROM, EPROM, Flash Speicher

   Maskenprogrammierbare Mikrocontroller.
   - ♦ Bei der Fertigung wird bereits das Programm mit festgelegt. Rentabel bei hoher Stückzahl.
   - ♦ Externer Programmspeicher sind meist EPROM oder ROM-Bausteine

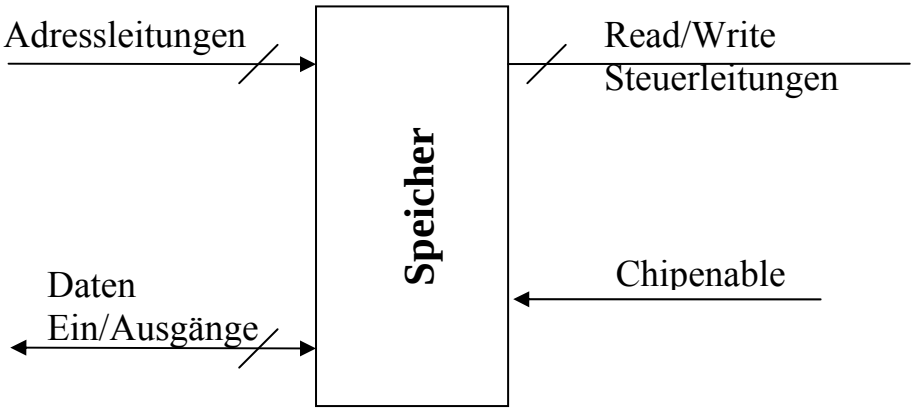
2. Der Datenspeicher enthält Daten, die während der Arbeit des Mikrocontrollers entstehen. Es werden z.B. flüchtige Schreib-Lesespeicher (RAMs)verwendet:

   - ♦ interne Speicher z.B. Akkumulator, Stack, Stackpointer
   - ♦ externe Speicher z.B. größere Datensätze

3. Flash RAMS
   Werden zum Ablegen von Daten und Programmen verwendet.
   Sie sind nicht flüchtig.

Speicher in einem Mikrocontrollersystem werden meist modular aufgebaut und den Bausteinen werden Adressbereiche zugewiesen. Der Speicherbaustein überstreicht dabei meist nicht den gesamten Adressbereich. Ein weiterer Grund ist das Ansprechen von Ein/Ausgabebausteinen ebenfalls über Adressbereiche, was zu einer Reduzierung des verfügbaren Adressraumes führt. Ein zusätzlicher Decoderbaustein sorgt dann zur Bestimmung eines Adressbereiches durch das Setzen einer dem Adressbereich zugeordneten Leitung. Die meisten Speicher besitzen zu diesem Zweck einen eignen Aktivierungseingang (Chip Enable). Der Speicher oder der Ein/Ausgabebereich wird an seinen Adressleitungen dann nur mit einer Auswahl der vorhandenen Adressleitungen versorgt.

Bei Schreib/Lesespeicher muss weiter zwischen einem Schreib- und einem Lesevorgang unterschieden werden. Durch Verwendung einer zusätzlichen Steuerleitung werden die Vorgänge gekennzeichnet.



Speicheranschlussbelegung zur Verwendung in einem Mikrocontrollersystem

2.1.4 Ein- /Ausgabe

Die Kommunikation mit der Außenwelt wird über besondere Leitungen oder Leitungsbündel durchgeführt. Es stehen daher bei einem Mikrocontroller eine Anzahl von Verbindungen mit unterschiedlichen Ein/Ausgabeaufgaben zur Verfügung:

- Serielle Ein-/Ausgabe RS232, I^2C, SPI
- Ausgabe der vom Rechen- oder Steuerwerk gelieferten Daten als Dauersignal oder in Form von Protokollen z.B. CAN-BUS
- Eingabe von Daten in das System
  - Durch Abtastung externer Signale (z.B. von Schaltern)
  - Umwandlung analoger Signale in ein vom Rechner verarbeitbares binäres Wort.
- Verarbeitung von Zählereignissen und zeitgebundenen Signalen (TIMER)
- Ansteuerung von Anzeigeeinheiten z.B. LCD Display

2.2 BUS-Struktur

Bisher wurde die Grobstruktur eines Mikrocontrollersystems vorgestellt. Die Verwendung von BUS-Strukturen zur internen und externen Datenkommunikation wurde bereits als ein Merkmal eines Mikrocontrollersystems genannt. Darunter wird die Verbindung fertiger komplexer integrierter Schaltkreise über definierte Leitungsbündel verstanden ($\rightarrow$ BUS (Sammelschiene)).

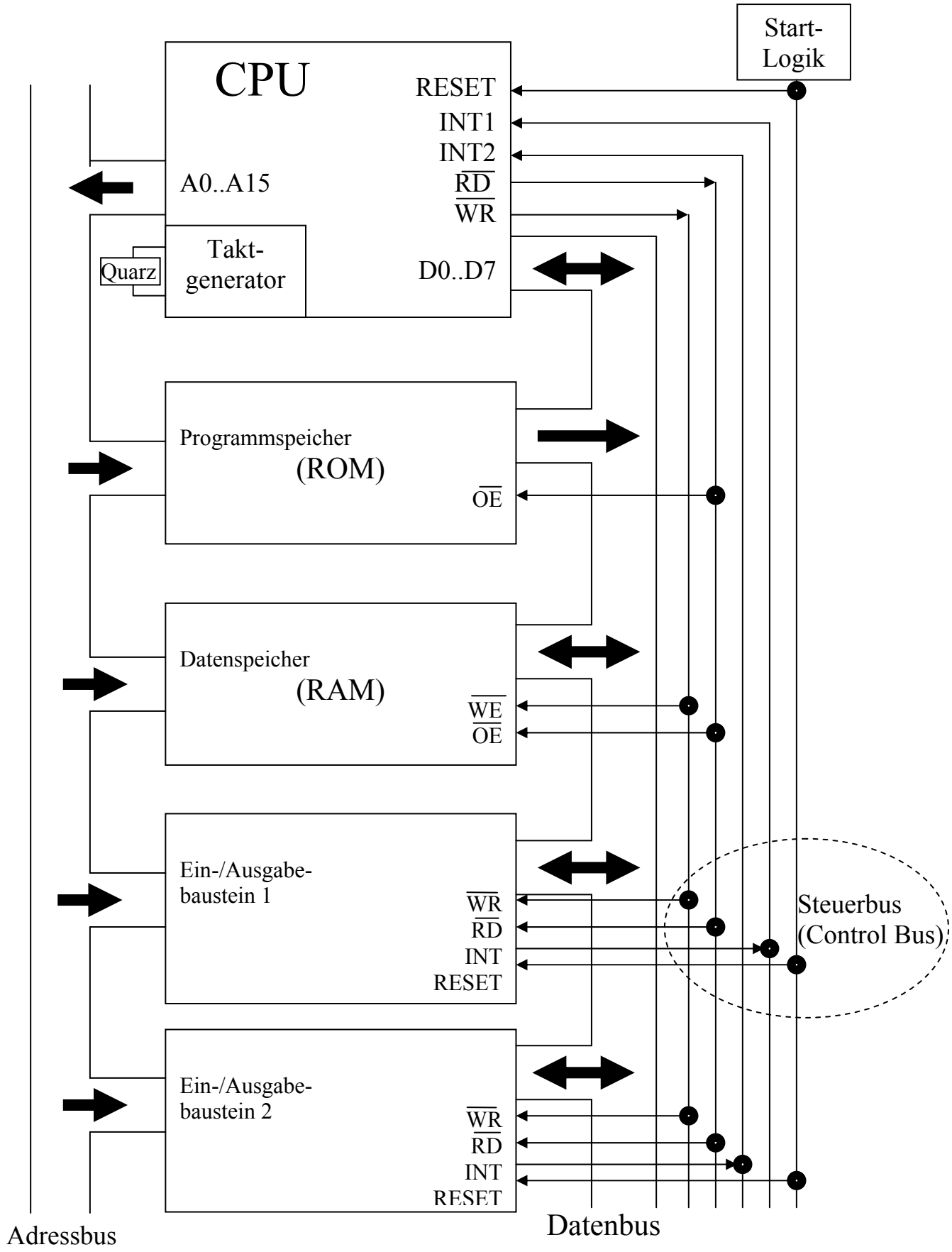
Zum Verständnis der Arbeitsweise eines Prozessors und der effektiven Nutzung der vorhandenen Ressourcen ist es notwendig, folgende Gebiete zu betrachten:

- Art und Form des Datenaustausches
- Schaltungstechnik
- Organisation

Typisch : Drei – Bus – System

$\Rightarrow$ Datenbus
$\Rightarrow$ Adressbus
$\Rightarrow$ Steuerbus
$\Rightarrow$ + Betriebsspannungszuführung

Typisches Drei-Bus-System [BLE94]



2.2.1 Datenbus

Der Datenbus dient zur Übertragung der zu verarbeitenden Informationen oder der Ergebnisse aus der Ausführung eines Programmschrittes.
Die Anzahl der Leitungen entspricht der Wortbreite des Mikrocontrollers. Hierbei sind 4, 8, 16 oder 32 Bit üblich.

2.2.2 Adressbus

Der Adressbus wird zum Anwählen eines Datenwortes im Speicher verwendet.
Bei 8 Bit Mikrocontroller wird meist ein Bündel von 16 Leitungen benutzt.

Das Steuerwerk gibt über diesen BUS die Adresse eines Speicherplatzes oder eines Ein- / Ausgabe-Registers an, aus dem es entweder Daten- oder Programminformationen lesen will oder in das es Daten schreiben möchte.

Die Adressleitungen sind meist aufgeteilt in Leitungen zur Blockauswahl (Chipselect) und Leitungen zur Auswahl eines Wortes im Block.

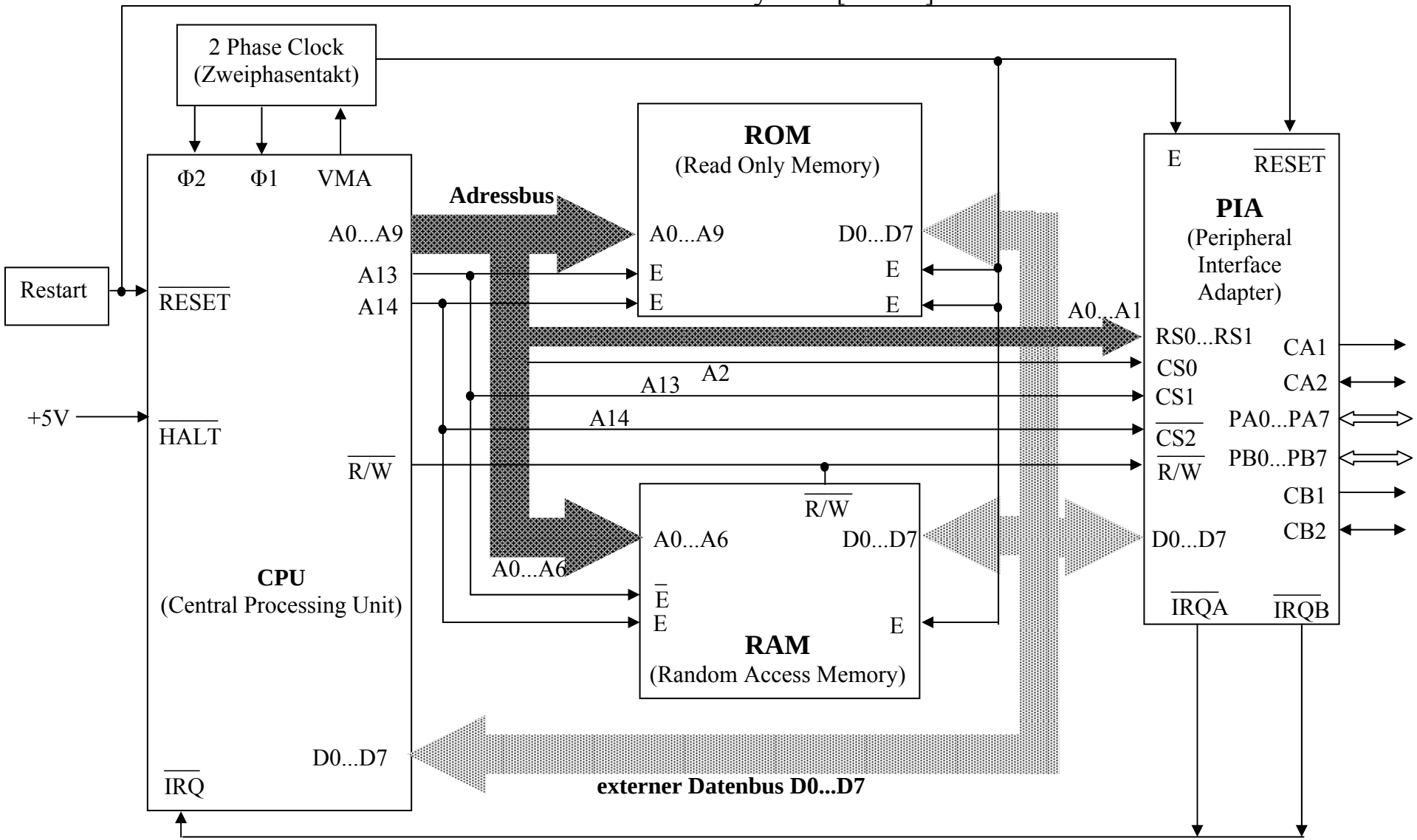
Mögliche Unterteilung :

Chipselect: höherwertige Adressleitungen
Wortauswahl: niederwertige Adressleitungen

Der Inhalt des durch die Adressleitungen ausgewählten Wortes erscheint dann nach einer spezifischen Zugriffszeit etwa gleichzeitig auf dem Datenbus.

Alle übrigen Leitungen, dienen der Steuerung des Systems. Die Anzahl ist variabel.

Ein Minimalsystem [BLE94]



A = Adressleitung, D = Datenleitung, E, E, CS = Freigabeanschluss, Φ = Takt, IRQ = Unterbrechungsleitungen (aktiv L), VMA = Valid Memory Address)

2.2.3 Bustreiber

Konsequenzen einer Busstruktur ->

Am BUS liegen alle Bausteine parallel

- Es entsteht ein Schreibkonflikt, wenn 2 oder mehr Bausteine Daten ausgeben möchten.

- Der Datenbus und einige Kontrollleitungen werden bidirektional verwendet.

- Soll einer der angekoppelten Bausteine gerade nicht aktiv sein, also weder Schreiben noch Lesen, so ist neben der Unterscheidung der Operationen durch Kontrollleitungen (*Read/Write)* daher ein Zustand „Nichtschreiben“ notwendig.

Lösung:

Neben dem Zustand 0 oder 1 wird ein Zustand „hochohmig“ eingeführt ➔ Tristate Anschluss. Das folgende Bild zeigt eine solche Konfiguration.

Funktion der Schaltung:

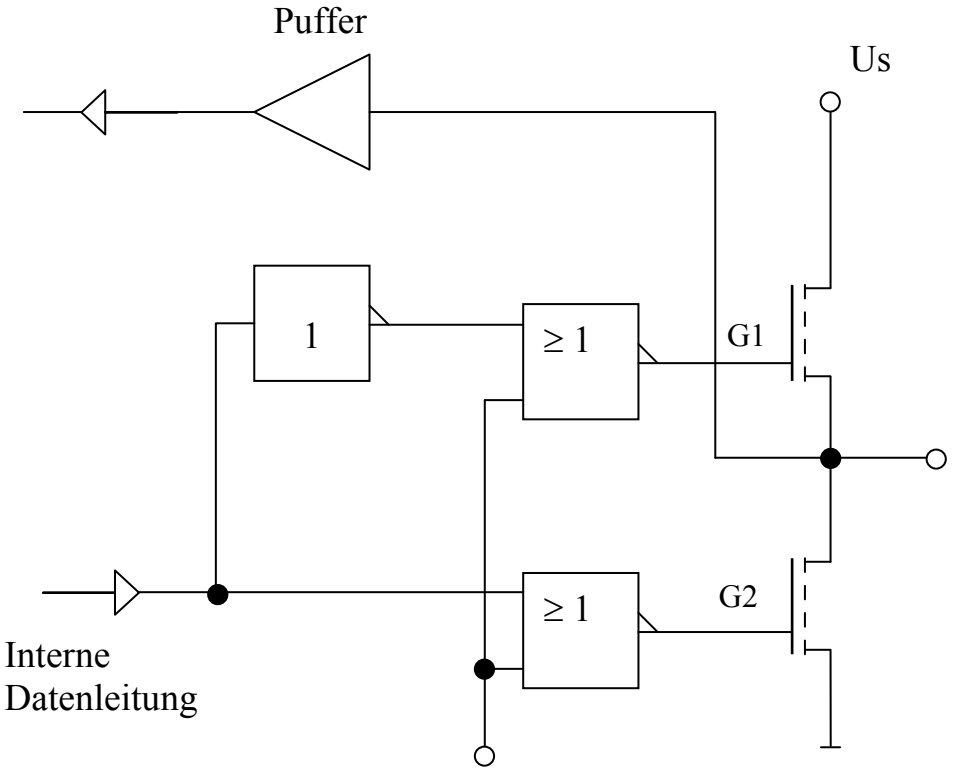
- Beide Ausgangstransistoren können gesperrt werden.

- Damit besteht weder ein Pfad zur Betriebsspannung, noch zur Masse.

- Schreiben kann auf den BUS nur ein Baustein.

- Lesen können mehrere Bausteine.

Realisierung eines Tristate Anschlusses

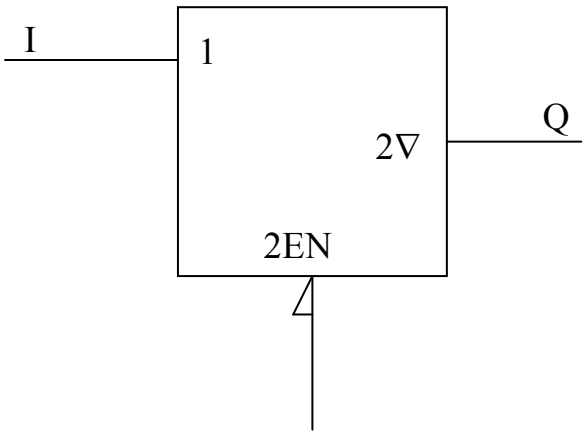


$\overline{E} = \overline{ENABLE}$ (Low Aktiv)

I	$\overline{E}$	G1	G2	
0	0	L	H	0 (Eingang I)
0	1	L	L	Hochohmig
1	0	H	L	1 (Eingang I)
1	1	L	L	Hochohmig

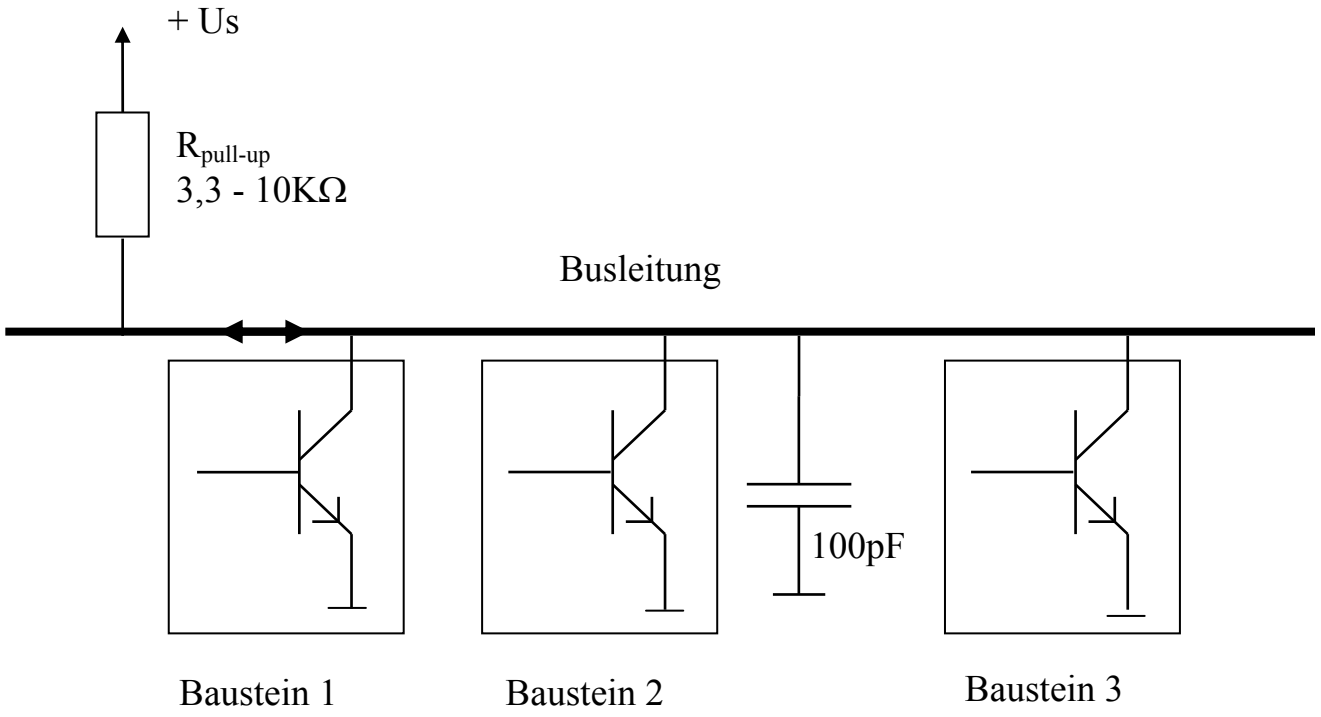
H schaltet den Transistor durch

Schaltzeichen:

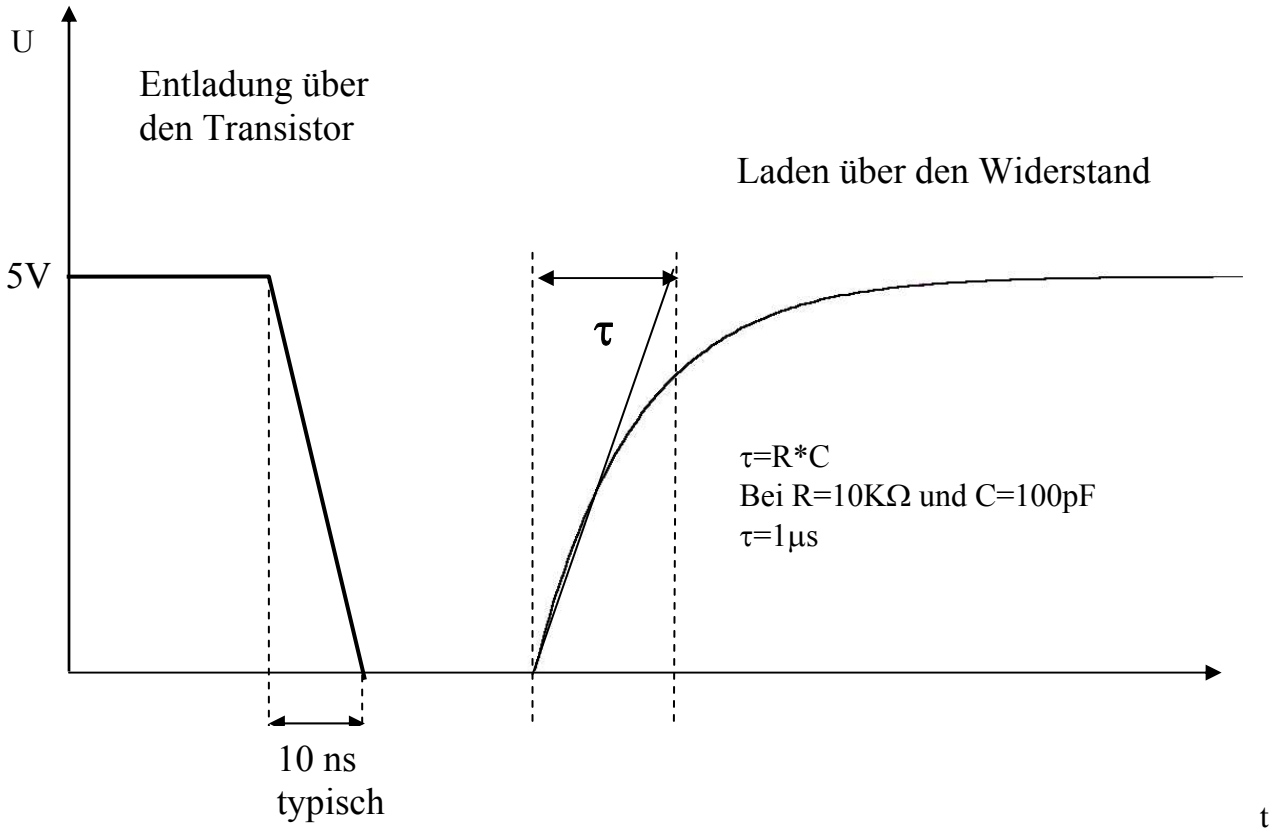


Open Collector Anschaltung [BLE94]

Bei nur einigen Bausteinen werden die Ausgangsstufen häufig nur durch einen Transistor realisiert. Der Kollektor- oder Drainanschluss führt dann direkt auf den Anschlussstift des Bausteines.



Signalverlauf auf einer Busleitung [BLE94]



2.3 Organisation und Arbeitsweise der Zentraleinheit

Zur Darstellung des Zusammenspiels der wichtigsten Funktionsblöcke eines Mikrocontrollers soll ein vereinfachtes Modell einer Recheneinheit (CPU) betrachtet werden. Die angegebene Struktur wurde dem ähnlich dem Aufbau eines häufig verwendeten Mikrocontrollers (8051) gestaltet.

Arithmetisch /logische Einheit (ALU)

Die Aufgaben einer arithmetischen Einheit sind bereits in den vorangehenden Kapiteln beschrieben worden. Die Baugruppe befähigt den Rechner beispielsweise arithmetische oder logische Operationen durchzuführen.

Register

Zusätzlich sind Register vorhanden, die zur Aufnahme von Daten und dem Zugriff auf Speicherbausteine dienen:

- Befehlszähler (Programm Counter ***PC***)
  enthält die (nächst folgende) Programmadresse.

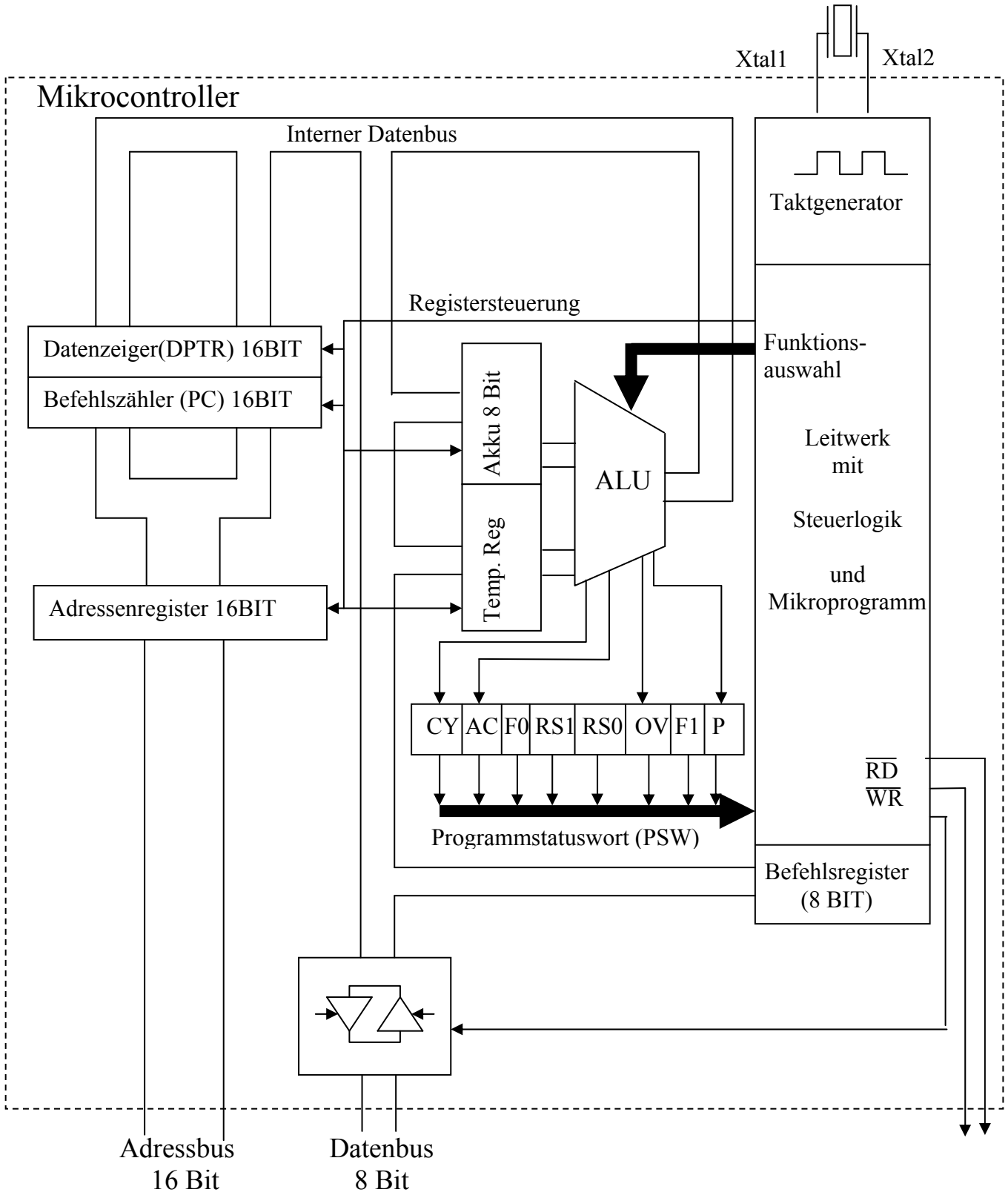
- Datenadresse (Data Pointer ***DPTR***)
  Quelle der Adresse für den Zugriff auf den externen Speicher ist der Datenzeiger.

- Adressenpuffer (Nicht für den Benutzer zugänglich.)
  Sorgt für die statische Ausgabe der aktuellen Adresse.

Leitwerk

- Steuerung der ALU über den Operationscode, abgelegt im Befehlsregister (Instructionregister)
- Auflösen eines Befehles in eine Reihe von Operationen
  durch ein Mikroprogramm (internes ROM-Programm)

Vereinfachtes Modell einer CPU [BLE94]
Angelehnt an die Struktur des Mikrocontrollers 8051



Der Datenfluss innerhalb der Beispiel-CPU soll im Folgenden an einem kleinen Programmbeispiel erläutert werden. Das angegebene Programm dient zur Addition zweier BCD-Zahlen. Eine BCD-Zahl (Binary Coded Decimal) kodiert die Symbole des dekadischen Zahlensystems (0…9) als binäre Zahl (0000…1001). Da für eine BCD-Zahl 4-Bit benötigt werden, können in einem 8 Bit-Datenwort also zwei BCD-Zahlen untergebracht werden. Um solche gepackte BCD-Zahlen effektiv verarbeiten zu können, stehen Befehle bereit, die nach einer Addition, die speziellen notwendigen Korrekturen zu Errechnung eines fehlerfreien Ergebnisses im BCD-Format mit Berücksichtigung eines Übertrages vornehmen können (DA Decimal Adjust).

Beispielprogramm: Einfache BCD-Addition:

Inhalt der Speicherzelle 3f05 ist 28h
Interpretation als BCD-Zahl → Rechnung 28 + 59 = 87

```
MOV     DPTR, #3F05h        ; Lade Datenadresse
MOVX    A, @DPTR            ; Hole den Speicherinhalt
                            ; in den Akkumulator
ADD     A, #59h             ; Addiere 59 hinzu (BCDZahl)
DA      A                   ; korrigiere das Ergebnis im
                            ; Akku dezimal
MOVX    @DPTR, A            ; Ergebnis am selben
                            ; Speicherplatz ablegen
```

Programmerläuterungen:

Verwendung eines Mnemonischen Codes z.B. MOV
(Merkbare Beschreibung eines Befehls, Hexadezimale Zahlen sind schwer zu behalten)

z.B. MOV DPTR, #data16 3 Byte Befehl
(es gibt auch 1,2 Byte Befehle)

bezeichnet einen Befehl um eine Adresse (16 Bit) in den Datenzeiger zu schreiben.

OPCODE	90_h	10010000_b
Daten 1	$3F_h$	00111111_b
Daten 2	05_h	00000101_b

Bedeutung der Zusatzangaben bei den Datenwerten:

direkte Angabe eines Zahlenwertes. Die Folge **muss** mit einer Zahl beginnen. Also # $0FF_h$

Kennzeichnung der Basiszahl:

h	=	Hexadezimal
b	=	Binärzahl
Nichts	=	Dezimalzahl

#10 → $\#0A_h$ → $\#00001010_b$

Bedeutung der verwendeten Befehle:

MOV zum Zugriff auf „interne Daten“
MOVX zum Zugriff aus „externe Daten“
ADD Addition des Inhaltes des Akkumulators, in diesem Fall zu einer Konstanten (A) = (A) + #Daten

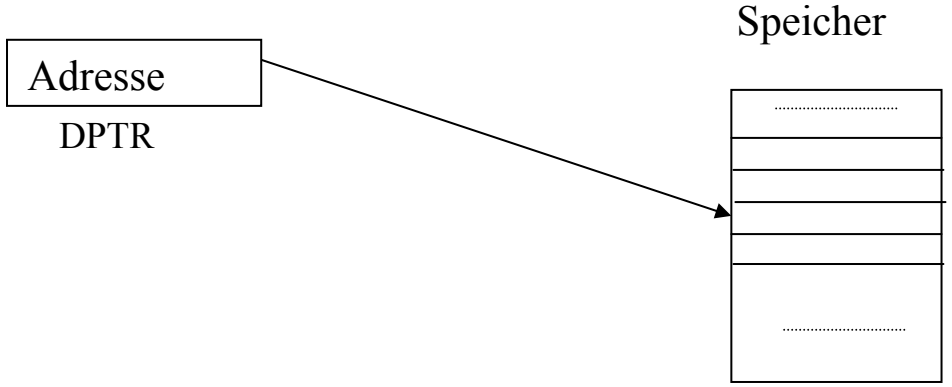
DA Decimal Adjust
Es erfolgt eine Aufteilung des Akkumulators in 2 Hälften (Halbbytes).

4 höherwertige Bit (MSB)
4 niederwertige Bit (LSB)

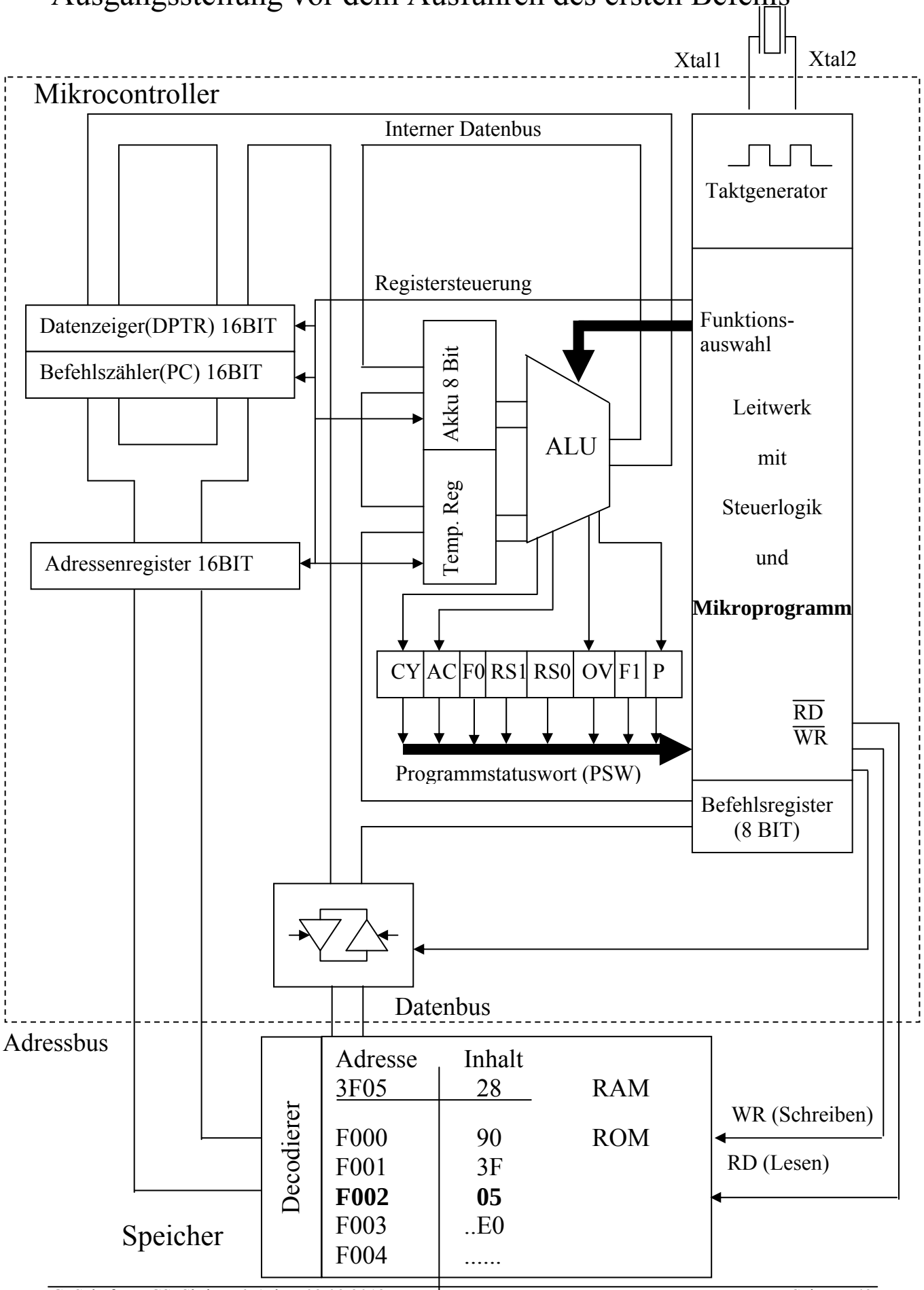
Ist der Inhalt des jeweiligen Halbbytes>9 wird 6 addiert. Wegen des Überlaufes (Carry) wird mit den niederwertigen Bits begonnen.

Vorsicht!!! Keine Umwandlung in BCD Zahlen
Keine dezimale Subtraktion

@ Kann als Auflösung des Verweises interpretiert werden. Vergleichbar mit * in C



Ausgangsstellung vor dem Ausführen des ersten Befehls



Ausführen des 1. Befehls:

Bei einem Befehl wird zwischen der Holphase und der Ausführungsphase unterschieden. In der Holphase werden in einer rechnerspezifischen Weise die Daten für die nächste Aktion in die entsprechenden Register geladen. In der Ausführungsphase wird dann die vom Programmierer gewünschte Aktion ausgeführt.

Holphase:

1. Der Befehlszähler wird in das Adressenregister geladen und sofort um 1 erhöht. Damit kann ein später folgender Speicherzugriff schon vorbereitet werden.
2. Das Lesesignal $\overline{RD}$ wird auf L gesetzt. (Beim 8051 auch $\overline{PSEN}$ genannt)
   2 Takte nach Ausgabe der Adresse erscheint die Information am Ausgang: 90h=10010000b
3. Der Prozessor liest das Datum in den Datenpuffer ein. Mit der steigenden Flanke des Lesesignals wird die Information in die gewünschte Datensenke (Register) weitergeleitet.

Ausführungsphase

Nach der Holphase beginnt die Ausführungsphase des im Befehlsregister stehenden Operationscodes MOV DPTR, #3705h

Interpretation:

1. Ein Operand wird benötigt, der dem Operationscode unmittelbar folgt.

2. Der Operand ist eine 2 Byte Zahl.

3. Der Operand ist in den Datenzeiger zu bringen.
   Dazu werden folgende Aktionen durchgeführt:
   - (ADR) ← (PC) Inhalt von PC ergibt den Inhalt vom ADR
   - (PC) ← (PC) + 1 Programmzähler erhöhen
   - (ADR) = F001h Ausgabe an die Adressleitungen
   - Lesevorgang einleiten $\overline{RD} \rightarrow$ L
   - Speicher gibt die Information 3Fh auf den Bus aus

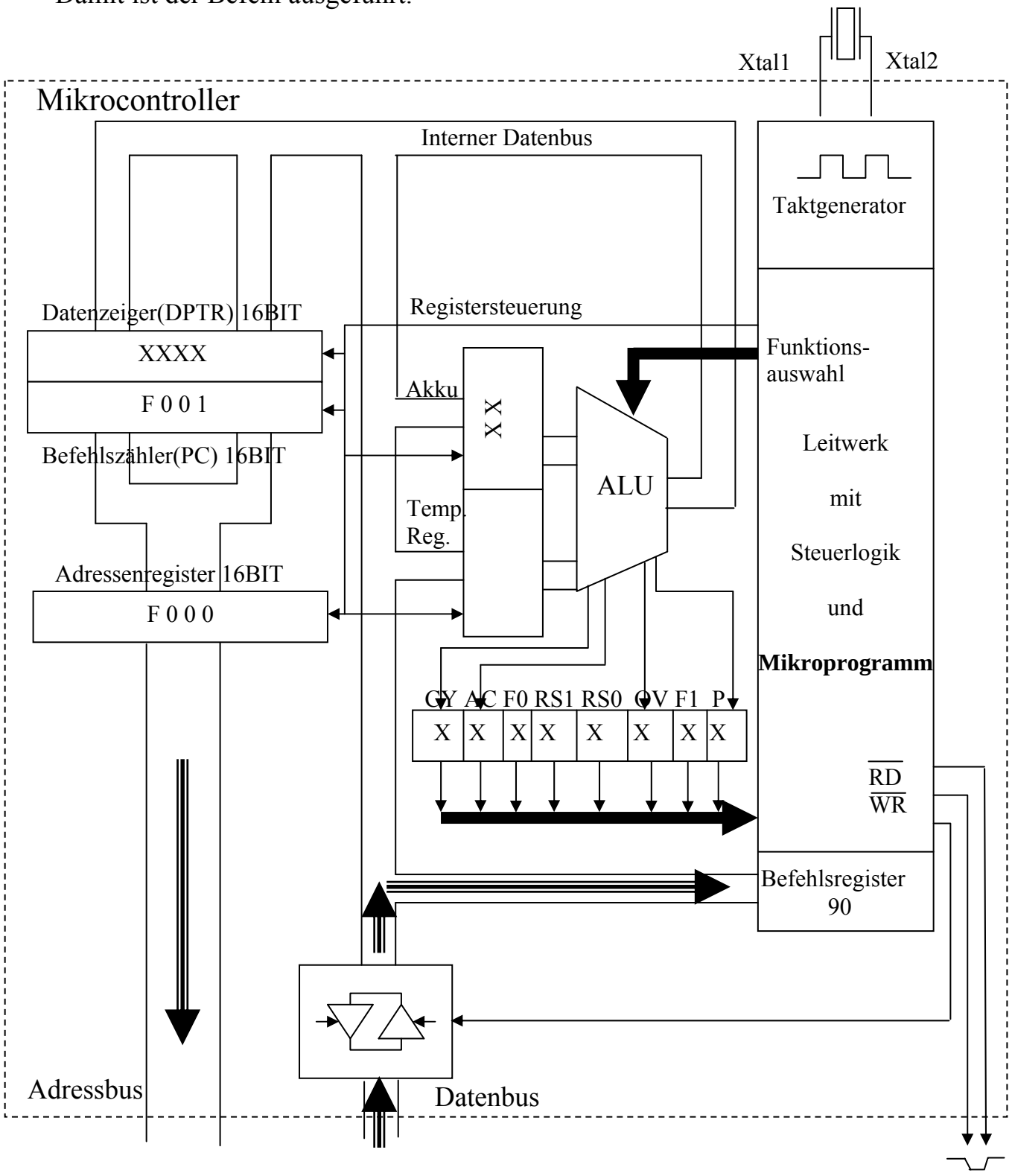
   Bei der steigenden Flanke von $\overline{RD}$ Übernahme in das MSB Byte des Datenzeigers.

   Der Vorgang muss für das LSB wiederholt werden:

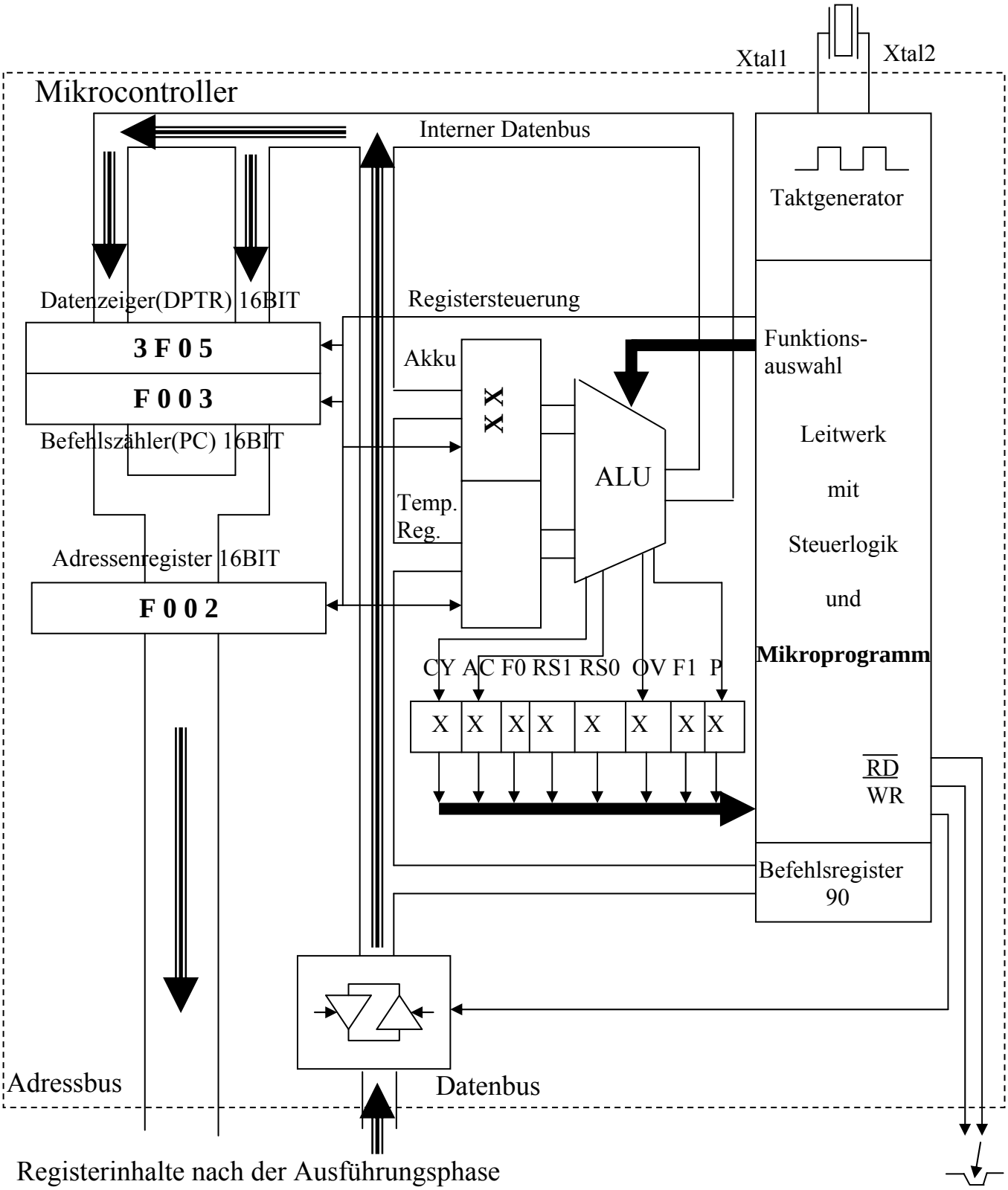
Zu 3. Fortsetzung
Holen des LSB der Adresse

- (ADR) ← (PC); (PC) ← (PC) + 1
  (ADR) = F002h
- Leseleitung → L →05h wird in das LSB geladen

Damit ist der Befehl ausgeführt.



Registerinhalte nach der Holphase


Xtal1
Xtal2
Mikrocontroller
Interner Datenbus
Taktgenerator
Datenzeiger(DPTR) 16BIT
Registersteuerung
3 F 0 5
F 0 0 3
Akku
XX
Funktions-
auswahl
Befehlszähler(PC) 16BIT
ALU
Leitwerk
mit
Steuerlogik
und
Mikroprogramm
Temp.
Reg.
Adressenregister 16BIT
F 0 0 2
CY AC F0 RS1 RS0 OV F1 P
X X X X X X X X
RD
WR
Befehlsregister
90
Adressbus
Datenbus
Registerinhalte nach der Ausführungsphase

Bearbeitung von MOVX A, @DPTR:

Holphase:

- (ADR) ← (PC) Inhalt von PC ergibt den Inhalt vom ADR
  (PC) ← (PC) + 1Programmzähler erhöhen
  (ADR)= $F003_h$

- Lesesignal generieren
  (Befehlsregister) = E0

Ausführungsphase:

- (ADR) ← (DPTR)

- Lesesignal ausgeben $\overline{RD} \rightarrow$ L

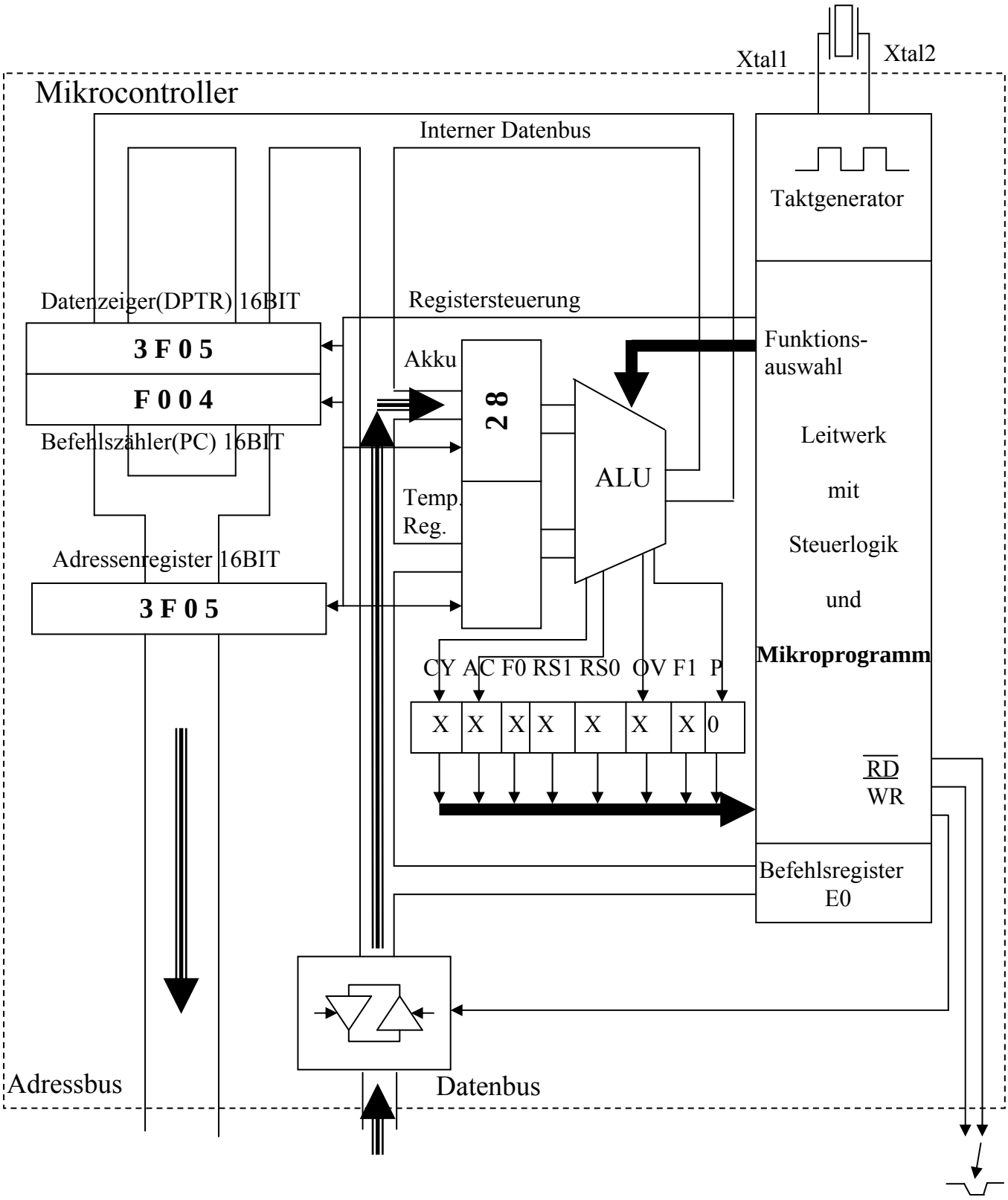
- Speicherinhalt liegt am Datenbus

- Mit steigender Flanke von $\overline{RD} \rightarrow$ Übernahme in den AKKU
  (A)= 28_h

- Aktualisierung der Kennzeichenbits

Paritybit 28_h = 0010 1000_b →gerade Anzahl von Einsen → P=0

Welche Kennzeichenbit betroffen sind, ist jeweils der Befehlsliste zu entnehmen!



Registerinhalte nach Ausführung des Befehls MOVX A,@DPTR

Bearbeitung von ADD A, #59h:

Holphase:

- (ADR) ← (PC) Inhalt von PC ergibt den Inhalt vom ADR
  (PC) ← (PC) + 1 Programmzähler erhöhen
  (ADR) = $F004_h$ Ausgabe an die Adressleitungen
- Lesesignal
  (Befehlsregister) = 24_h OP Code geholt

Ausführungsphase:

- (ADR) ← (PC) Operand holen
- Lesesignal
  (temp. Reg.) = 59_h
- Addition ausführen:

	$0010\ 1000_b$ oder	28_h	
+	$0101\ 1001_b$	59_h	
	$1000\ 0001_b$	81_h	Dualzahl keine BCD-Zahl

- Übernahme des Ergebnisses in den Akkumulator
  (A) = 81_h

Setzen des Programmstatuswortes:

CY = 0 kein Übertrag vorgenommen

AC = 1 Auxiliary Carry. Die unteren (Halbbytes) haben einen Übertrag produziert.

OV = 1 Die Addition der beiden positiven Dualzahlen ergab eine Zahl, die als negative Zahl aufgefasst werden kann. (Vorzeichenbehaftete Dualzahl)

P = 0 Anzahl der Einsen ist gerade

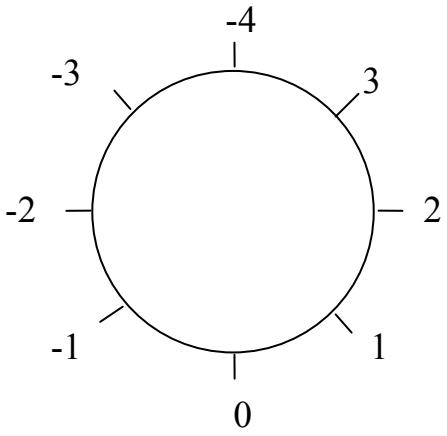
Overflowbearbeitung:

Tritt bei einer Addition oder Subtraktion eine Überschreitung des Zahlenbereiches auf, so wird das durch das Overflowflag angezeigt, wenn die verwendeten Zahlen in K2-Format (K2-Komplement) vorliegen. Die Wirkungsweise soll für einen Zahlenbereich von 3 Bit gezeigt werden. Für höhere Bitzahlen gelten die Überlegungen entsprechend.

Für 3 Bit gelten folgende Zuordnungen:

Nr.	Z2	Z1	Z0	Bedeutung
0	0	0	0	0
1	0	0	1	1
2	0	1	0	2
3	0	1	1	3
4	1	0	0	-4
5	1	0	1	-3
6	1	1	0	-2
7	1	1	1	-1

Darstellung im Zahlenkreis



Beispiele
Ohne Überlauf

Addition:			Subtraktion		
	(-1)	111		(-1)	111
+	(-2)	110	-	(-2)	010
	(-3)	1 101		1	0 001

Mit Überlauf

Addition:			Subtraktion		
	(-1)	111		(1)	001
+	(-4)	100	-	(-3)	011
	(-5?)	1 011		(+4?)	0 100

Bei der Subtraktion wird die zu subtrahierende Zahl in ihr Komplement gewandelt und eine Addition durchgeführt.

Überlaufregeln

- Wenn die Addition mit zwei negativen Zahlen durchgeführt wird und das Ergebnis ist positiv, so ist ein Überlauf aufgetreten.

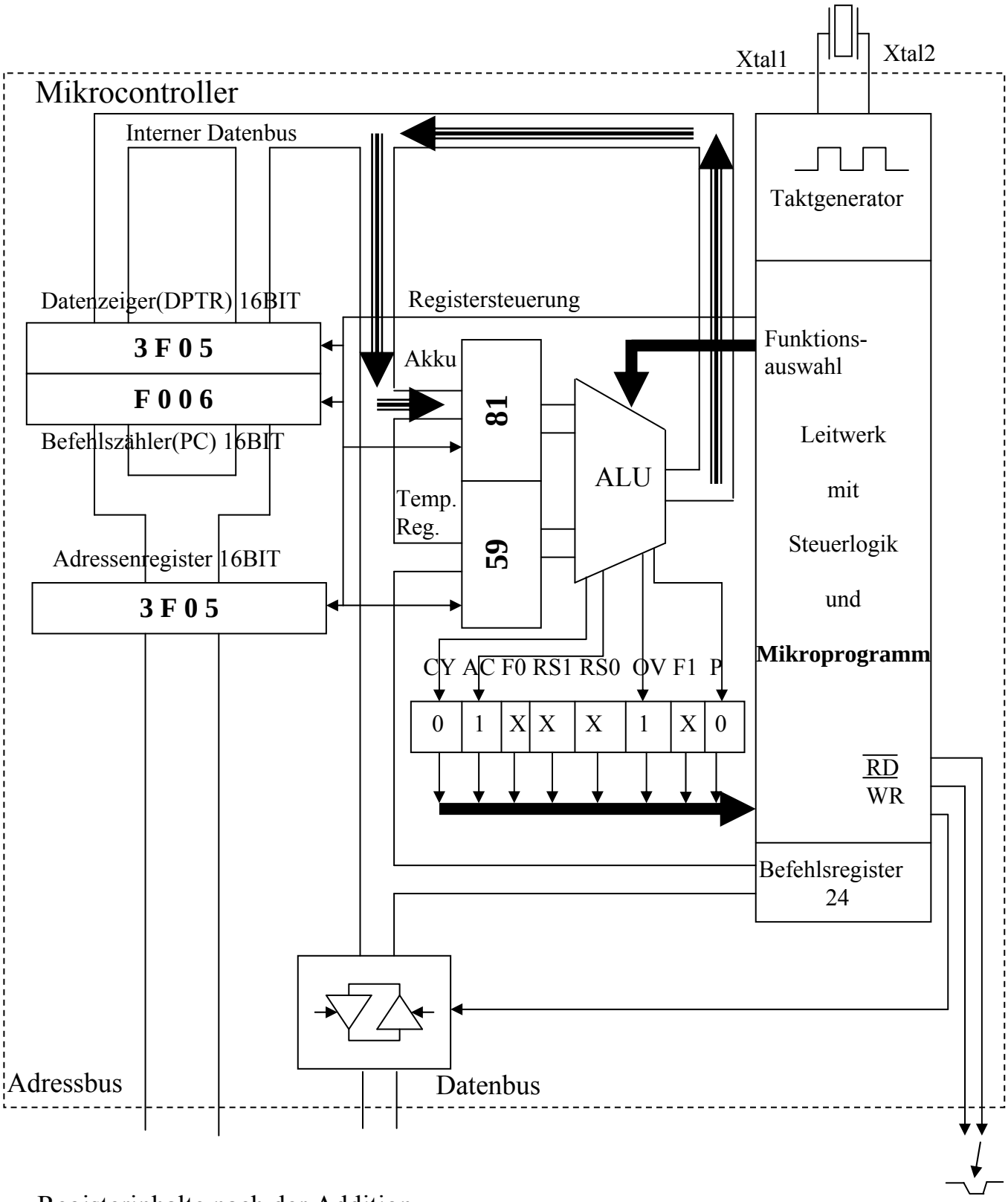
- Wenn die Addition mit zwei positiven Zahlen durchgeführt wird und das Ergebnis ist negativ, so ist ein Überlauf aufgetreten.

- Bei Subtraktionen gelten die Regeln nach der Umwandlung genauso.

Überlaufermittlung:

Nr.	VZ2	VZ1	VErg	OV
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	0
4	1	0	0	0
5	1	0	1	0
6	1	1	0	1
7	1	1	1	0

$OV=(VZ2' \wedge VZ1' \wedge VERG) \vee (VZ2 \wedge VZ1 \wedge VERG')$



Registerinhalte nach der Addition

Bearbeitung von DA A:

Holphase:

- (ADR) ← (PC) Inhalt von PC ergibt den Inhalt vom ADR
  (PC) ← (PC) + 1 Programmzähler erhöhen

- Lesesignal
  (Befehlsregister) = $D4_h$

Ausführungsphase:

- Ausführung ← Addition von 6 bei den Halbbytes, wenn nötig.

Die Addition von 6 ist notwendig, wenn bei den niederwertigen Bits der Wert > 9 ist (AC==1). Dies gilt entsprechend für die höherwertigen Bits bei (CY==1);
In diesem Falle AC == 1 →+6
81(Dualzahl) + 6(Dualzahl) = 87(BCD-Zahl)

- Ergebnis abspeichern im AKKU

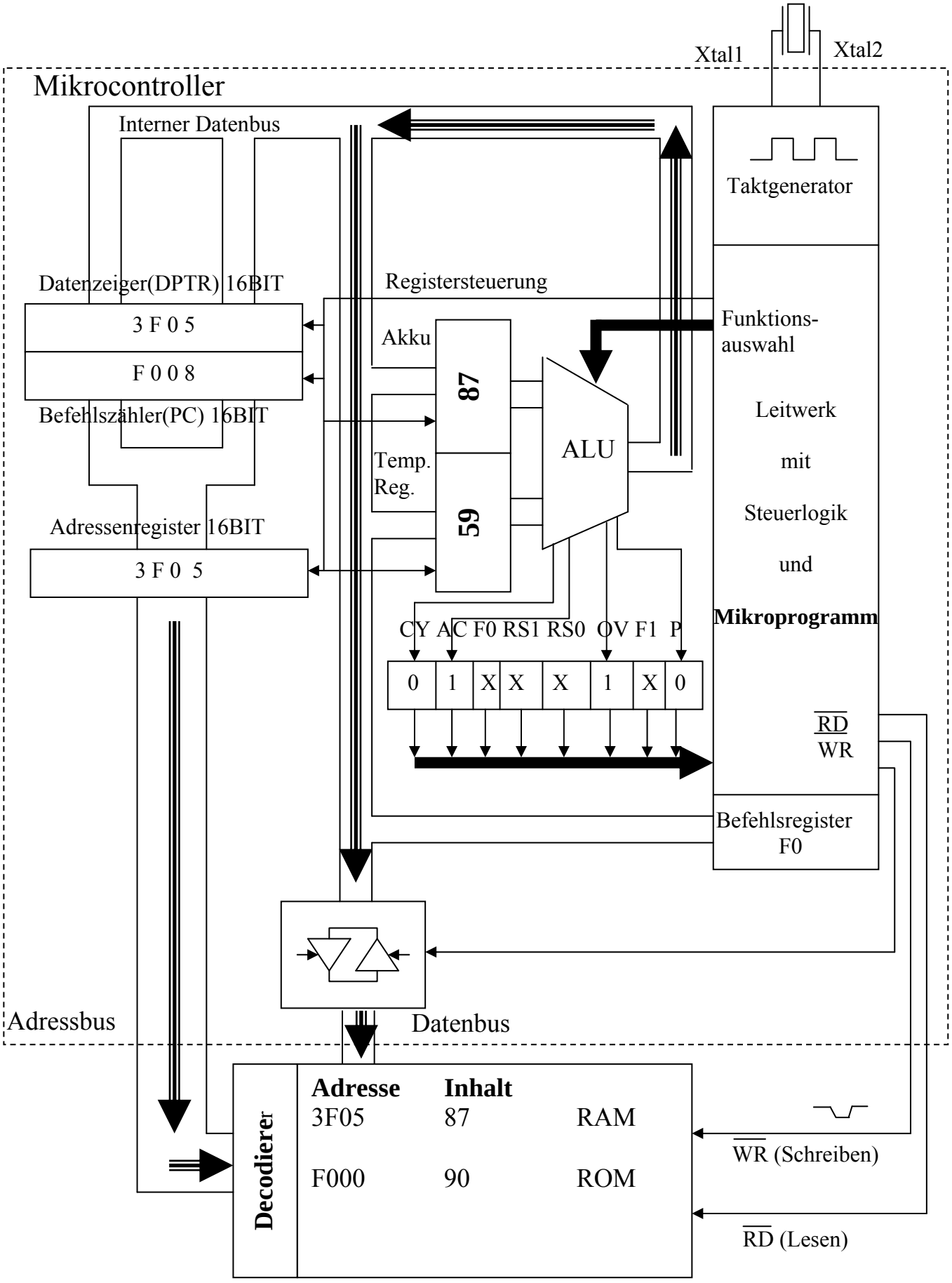
- CY und P werden aktualisiert
  Ein bereits gesetztes CY bleibt gesetzt.

Bearbeitung von MOVX @DPTR, A:

- (ADR) ← (PC)
  (PC) ← (PC) + 1
  (Befehlsregister) = $F0_h$

- (ADR) ← (DPTR) = $3F05_h$

- Schreibleitung $\overline{WR}$ = L
  Gleichzeitig: Datenbus ← (AKKU)
  Übernahme in den Speicher, spätestens mit der steigenden Flanke des Schreibsignals



Speichern des Ergebnisses

Zeitbetrachtungen für das Programmbeispiel:

Für den C8051F340 können die benötigten Takte aus dem Datenblatt ermittelt werden:

Befehl	Taktzyklen
MOV DPTR, #3F05h	3
MOVX A, @DPTR	3
ADD A, #59h	2
DA A	1
MOVX @DPTR, A	3
Summe	12

Benötigt werden: 12 Takte
Bei einer Taktfrequenz von 48 MHz

$\Rightarrow$ 250 ns Programmdauer

Bei einer Taktfrequenz von 12 MHz

$\Rightarrow$ 1 µs Programmdauer

Bemerkungen zum Datentransport:

Bei allen Datentransporten bleiben die Daten der Quelle unverändert. z.B. Befehlszähler, Speicher, Akkumulator

$\rightarrow$ wiederholte Anweisungen mit den gleichen Quellendaten sind möglich.

Der Datentransport ist ein Kopiervorgang, der die beim Bestimmungsort vorhandenen Daten überschreibt.

2.4 Steuerung der Speicher- und Ein-/Ausgabebausteine

2.4.1 Multiplex-Bussysteme

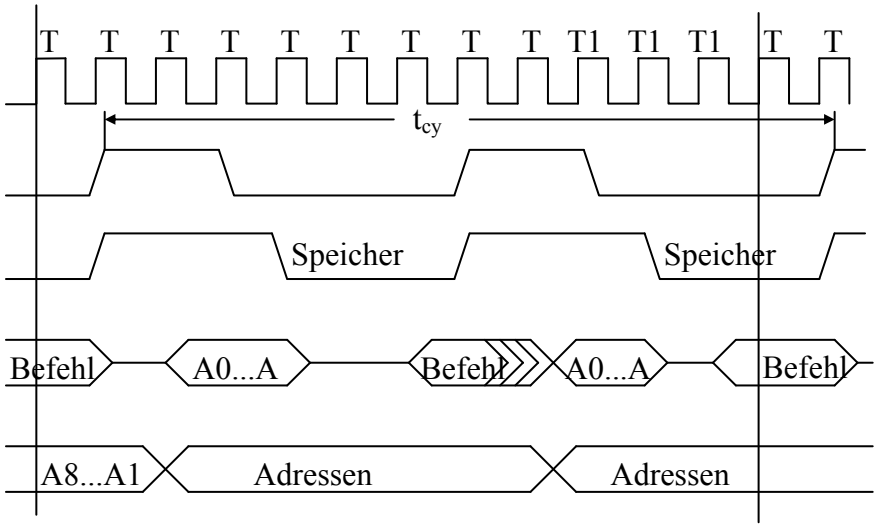
Die Vielseitigkeit eines Mikrocontrollers ist auch durch die Anzahl der verfügbaren Pins bestimmt. Zur Erweiterung der Ausgabekapazität werden u.a. Anschlüsse durch zeitliches Multiplexen mehrfach ausgenutzt. Die interne Organisation eines Prozessors liefert hier gegebenenfalls Möglichkeiten, durch einfaches Puffern von Informationen eine Mehrfachbelegung von Ein-/Ausgabeanschlüssen vorzunehmen. Angewendet wird ein solches Verfahren bei den Prozessoren der ursprünglichen 8051 Familie für die Adress- und Dateninformationen. Zuerst wird hier die Speicheradresse ausgegeben, zum Teil gepuffert und dann der Datentransport ausgeführt.

Die schaltungstechnische Realisierung ist im folgenden Bild der Multiplexereinsatz zur Adresscodierung dargestellt. Der Port 0 beispielsweise liefert oder übernimmt zum einen Datenwerte und gibt zum anderen die niederwertigen Adressenbits aus. Die höherwertigen Adressbits werden an einem anderen Port (z.B. Port 2) statisch ausgegeben. Es wird deshalb hier keine zusätzliche Maßnahme notwendig. Zur Steuerung des Auffangregisters wird ein Signal zur Verfügung gestellt, das die Übernahme in den Puffer anstößt.

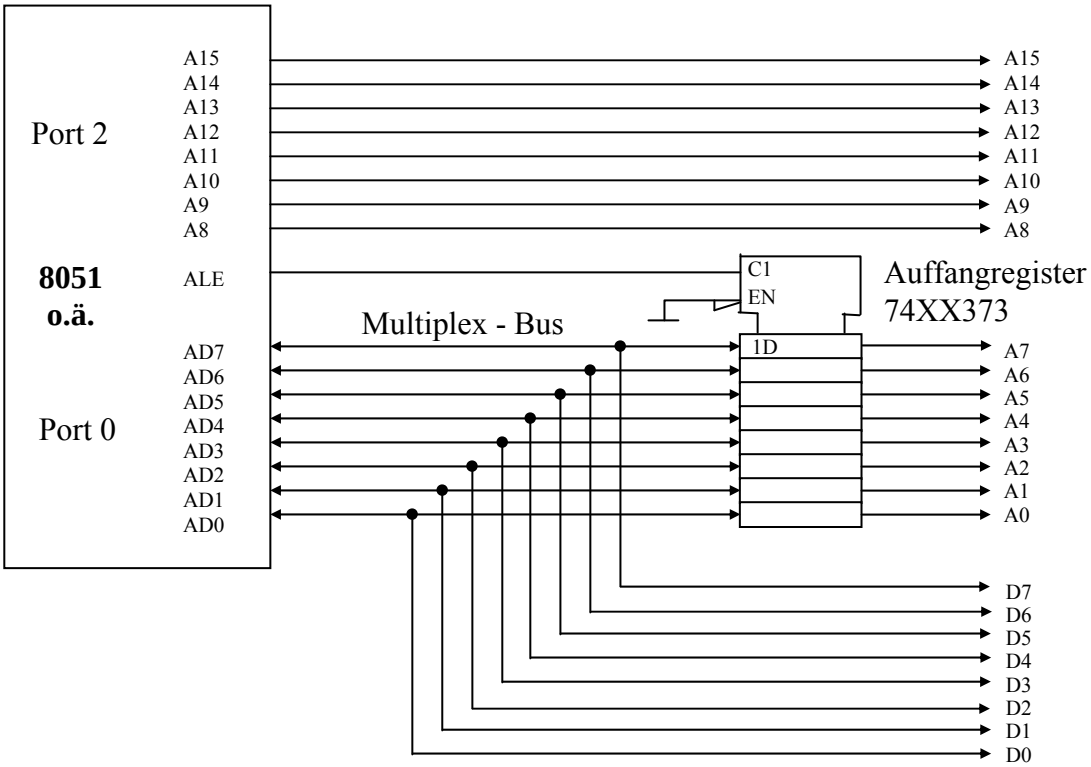
ALE (Adress Latch Enable)

Der Verlauf der Daten und der Steuersignale ist im Bild dargestellt. Das Signal PSEN dient zur Steuerung der Speicherbausteine. Zur Erklärung der Funktion wird auf den nächsten Abschnitt verwiesen.

Der C8051F340 verfolgt ein Konzept, das mit EMIF (External Memory Interface) bezeichnet wird. Hier werden tatsächlich zwei Ports für die Adresse und ein Port für die Daten verwendet. Zusätzliche werden die Steuerleitungen R,W, ALE zur Verfügung gestellt. Zur Bereitstellung von Daten für die acht höherwertigen Bit bei der Verwendung von 8 Bit Adresszugriffen muss eine spezielles Register (EMI0CN**)** geladen werden. Das Interface kann auch in seiner Geschwindigkeit konfiguriert werden. Die Implementierung der externen Daten- und Programmspeicher zum großen Teil mit auf dem Chip des Prozessors lassen eine solche Vorgehensweise, mit dem Nachteil eines hohen Pinverbrauchs für diese Schnittstelle, für besondere Anwendungen zu.



Bus-Signale des Prozessors 8051 beim Zugriff auf den Programmspeicher



Adressgenerierung eines 8051-Systems [BLE94]

2.4.2 Schreib/Lese-Steuerung

Im vorangehenden Abschnitt wurde bereits die Steuerung der Speicherbausteine über Steuerleitungen angesprochen. Insgesamt werden in einen 8051 System folgende Steuerleitungen zur Verfügung gestellt:

$\overline{PSEN}$ (Programm Storage Enable)

Das Signal dient zur Freigabe der Ausgänge des Programmspeichers (ROM)

$\overline{RD}$ Low aktiv, der Speicher wird zum Lesen aufgefordert.
$\overline{WD}$ Low aktiv, der Speicher wird zum Schreiben aufgefordert.

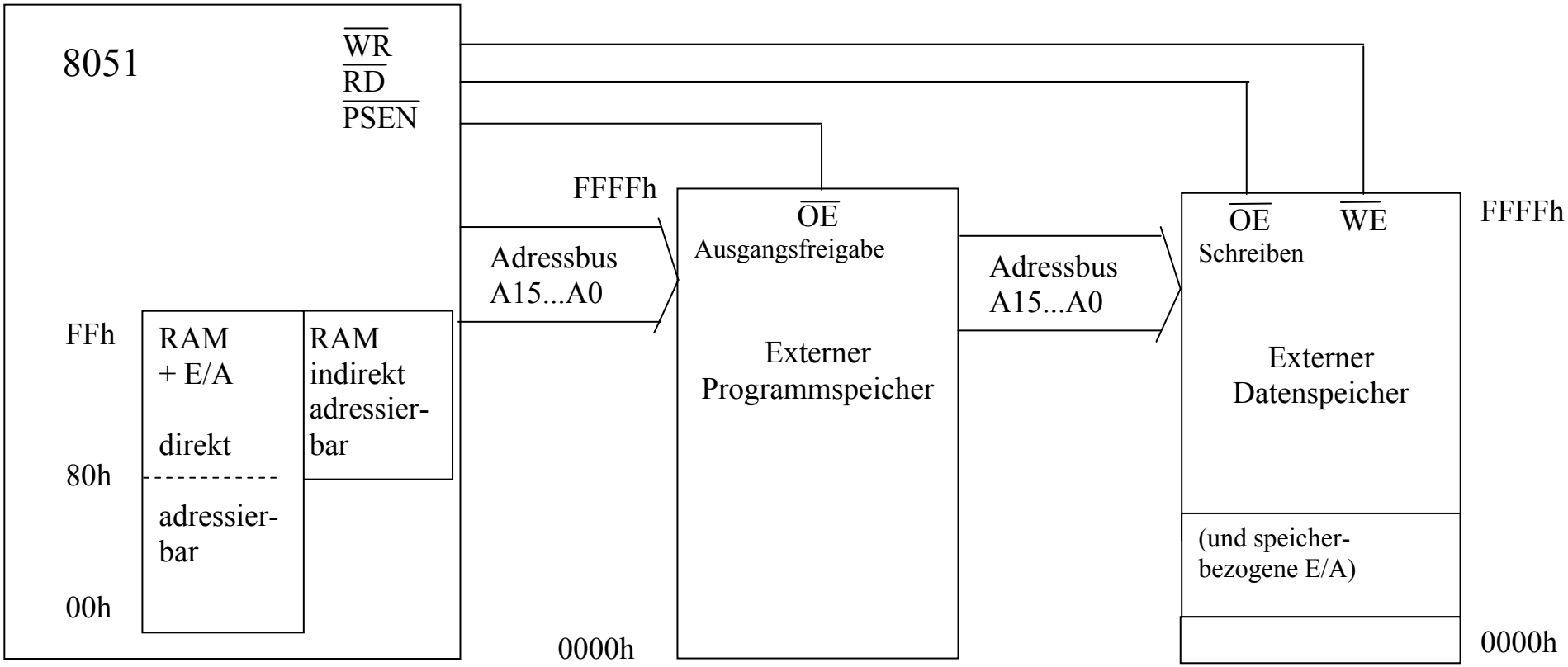
Die Schreib/Lesesteuerung wird sowohl bei Speicherbausteinen als auch bei Ein-/Ausgabebausteinen verwendet.

Einteilung der Speicherbereiche:

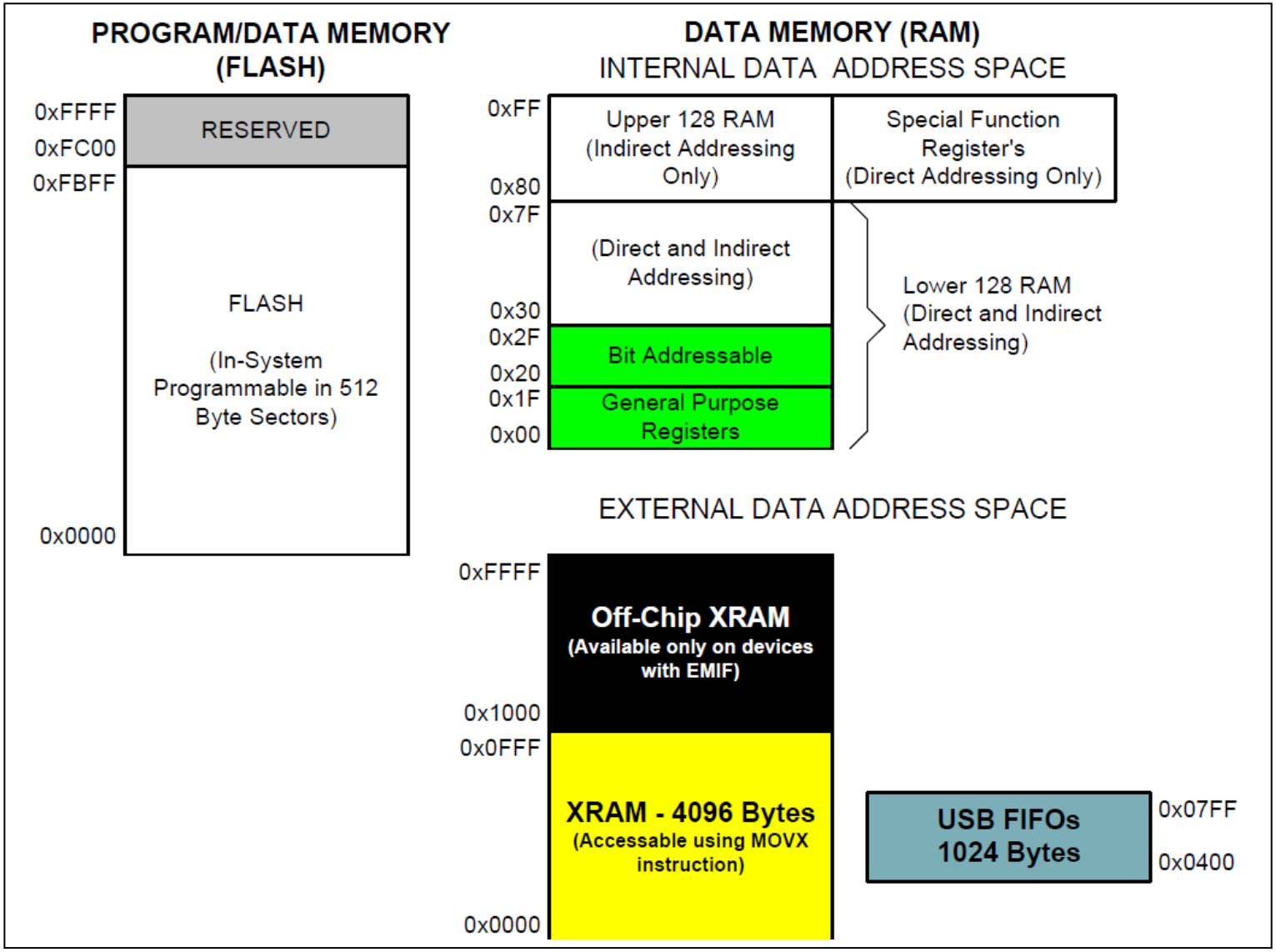
Internes ROM
Internes RAM
Externer Programmspeicher (ROM, EPROM)
Externer Datenspeicher
    RAM
    Ein- /Ausgabebaustein (Speicherbezogen)

Es können extern sowohl 64KBit ROM als auch 64KBit RAM angesprochen werden.
→ max. Speichergröße für Programme und Daten

RAM intern
00 – FF direkt adressierbar
    256 Speicherplätze
80 – FF indirekt adressierbar
    Vorher muss ein Adressregister geladen werden
    128 Speicherplätze



Maximal mögliche Speicherbereiche des Prozessors 8051



Speicherkonfiguration des Prozessors C8051F340 [DB1]

2.4.3 Memory Mapped I/O

Der Prozessor 8051 realisiert Ein-/Ausgabe über den direkt adressierbaren internen Speicherbereich. Alle Kanäle (Port) sind wie Speicher zugänglich.

➔ Speicherbezogene Ein/Ausgabe (Memory Mapped I/O)

Spezielle Ein- / Ausgabebausteine werden über den externen RAM-Speicherbereich adressiert.
Im Programm verhalten sich diese Bausteine wie beim Schreiben / Lesen auf eine Speicheradresse.

Damit jeder Baustein weiß, wann er Daten liefern soll, muss das jeweilige Chipenable über einen Kodierbaustein erzeugt werden. Man ordnet jedem Baustein hierbei einen Adressbereich zu und stellt dann fest, ob der Adressbereich gerade aktuell für einen Baustein gültig ist und das Chipenable des Bausteins wird über eine spezielle Leitung aktiviert.

3 Der Mikrocontroller 8051

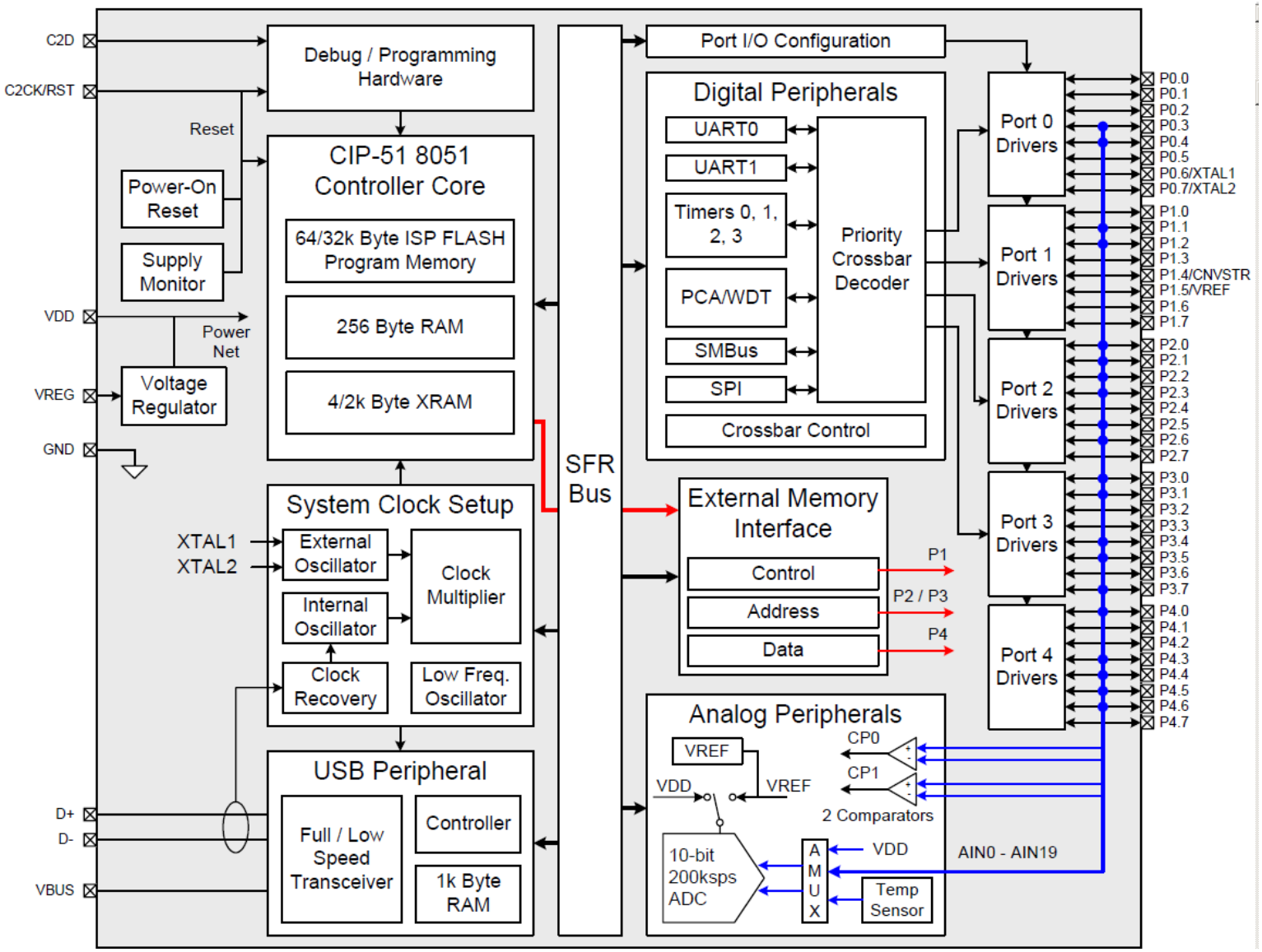
3.1 Modell des 8051

Zur Erarbeitung von Problemlösungen werden C- oder Assemblerprogramme erstellt, die auf vorhandene Daten in den verschiedenen Funktionseinheiten zugreifen. Hierbei sind nicht nur Speicher im herkömmlichen Sinne gemeint sondern auch Register zum Ablegen von Steuerinformationen z.B. von Timern oder Schnittstellenbausteinen. Stellt man die vorhandenen Speicherbereiche, Steuerregister, Ein-/und Ausgabeleitungen in einem Schaubild dar, erhält man ein abstraktes Bild des Prozessors. Dieses Modell ist kein Schaltplan, sondern hilft bei der Programmierung zum Auffinden von Adressen und Speicherbelegungen.

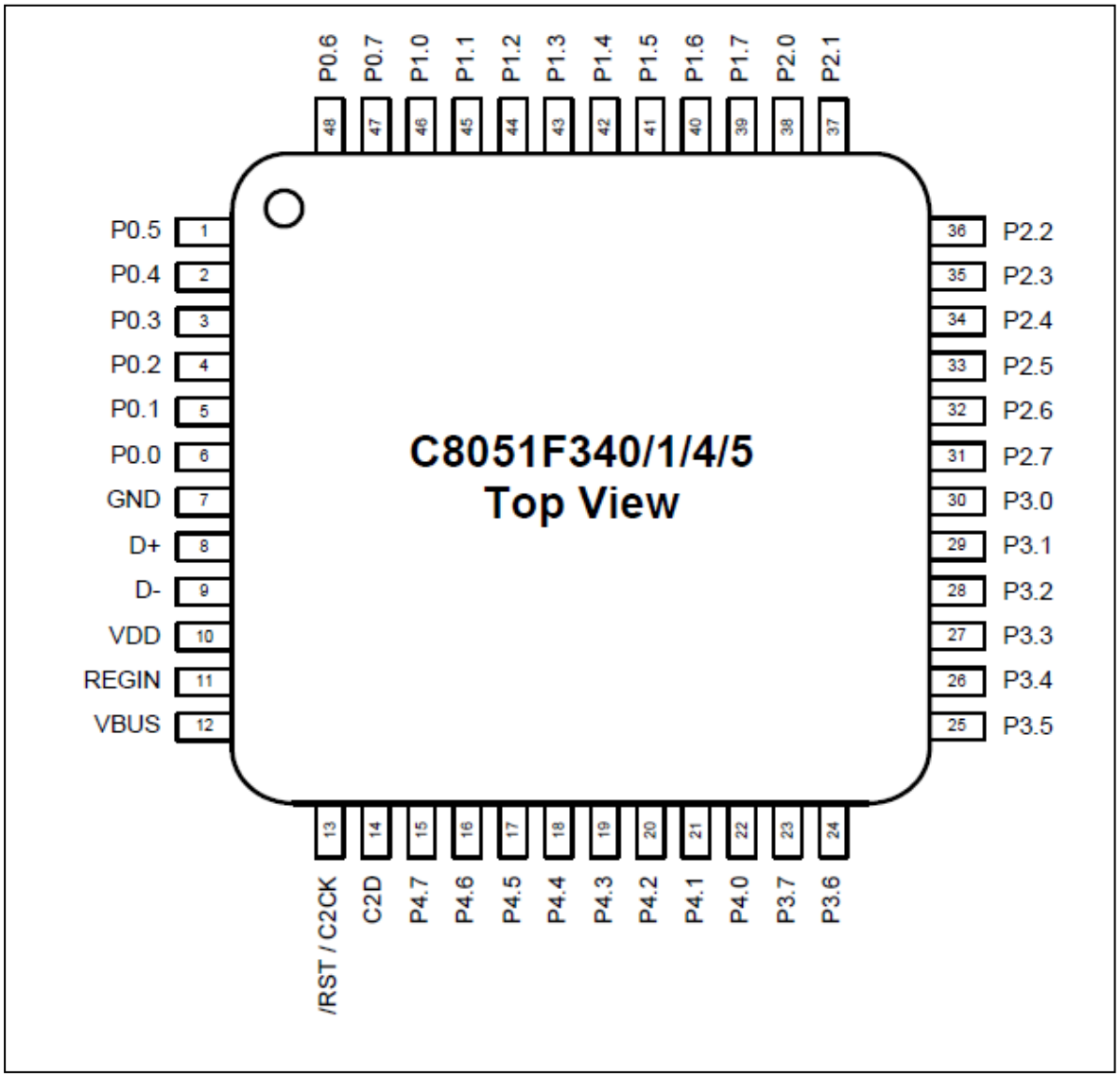
Die für die Programmierung häufig verwendeten Speicherplätze z.B. den Akkumulator werden neben ihrer Speicheradresse (im internen RAM) auch mit symbolischen Namen angegeben. Diese Namen können dann im Programmtext verwendet werden und erhöhen wesentlich die Lesbarkeit.

Würde versucht werden alle Informationen, die zu einer Problemlösung notwendig sind, in einem solchen Modell unterzubringen, würde sehr schnell eine unübersichtliche Grafik entstehen. Das Lesen des Datenblattes bleibt dem Programmierer trotz Vorhandenseins eines Modells nicht erspart. In den folgenden Kapiteln werden zunächst die dargestellten Register und Speicherbereiche erläutert.

Struktur des Mikrocontrollers C8051F340 [DB1]



C8051F340 Gehäuse TQFP-48 Anschlussbelegung[DB1]



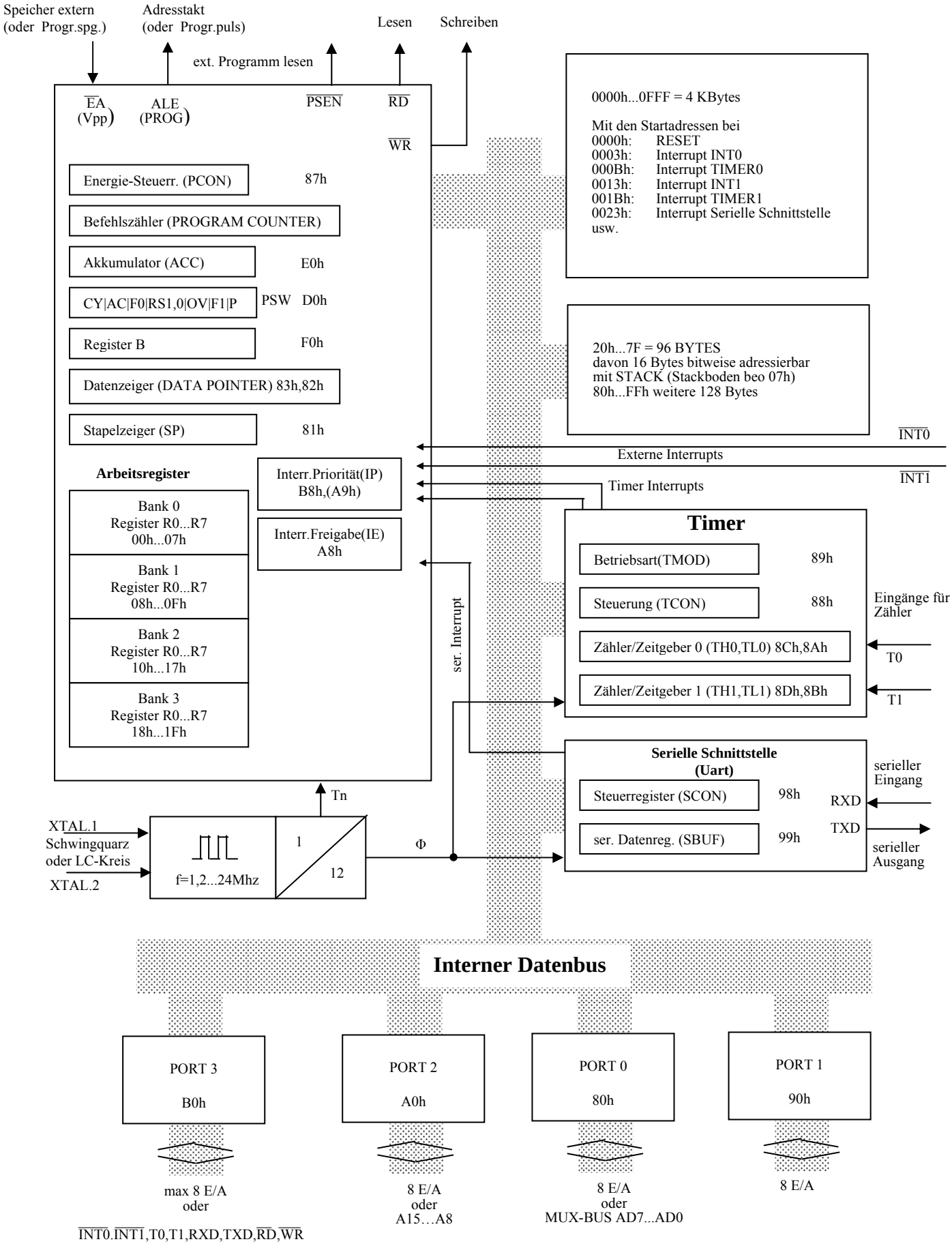
Pinbelegung des C8051F340 für die Vorlesung und das Labor

Pin	Port	Verwendung im Labor
6	P0.0	Int0 – Taster 9
5	P0.1	Int1 – Taster 10
4	P0.2	I²C – SDA
3	P0.3	I²C – SCL
2	P0.4	UART – TX
1	P0.5	UART – RX
48	P0.6	NC
47	P0.7	XTAL – externer Oszillator
46	P1.0	GPIO
45	P1.1	ADC – PB4RT_1 oder POTI (über JP1 auswählbar)
44	P1.2	ADC – PB4RT_2
43	P1.3	GPIO
42	P1.4	GPIO
41	P1.5	VREF
40	P1.6	GPIO
39	P1.7	GPIO
38	P2.0	Enable 7-Segment #6
37	P2.1	Enable 7-Segment #5
36	P2.2	Enable 7-Segment #4
35	P2.3	Enable 7-Segment #3
34	P2.4	Enable 7-Segment #2
33	P2.5	Enable 7-Segment #1
32	P2.6	UART – RTS
31	P2.7	UART – CTS
30	P3.0	7-Segment – A
29	P3.1	7-Segment – B
28	P3.2	7-Segment – C
27	P3.3	7-Segment – D
26	P3.4	7-Segment – E
25	P3.5	7-Segment – F
24	P3.6	7-Segment – G
23	P3.7	7-Segment – DP

Pin	Port	Verwendung im Labor
22	P4.0	Taster 4
21	P4.1	Taster 3
20	P4.2	Taster 2
19	P4.3	Taster 1
18	P4.4	LED 4
17	P4.5	LED 3
16	P4.6	LED 2
15	P4.7	LED 1
7		GND
8		USB – D+
9		USB – D-
10		VDD (3,3V)
11		REGIN (On-Chip-Voltage-Regulator)
12		USB – VSENSE
13		Reset
14		Debug Interface

```
void Port_IO_Init(){
  //Activate Analog Inputs on P1.1 and P1.2
  P1MDIN= 0xF9;     // 1111 1001b
  //Activate PushPull for P2 and P3
  P2MDOUT= 0xFF; // 1111 1111b
  P3MDOUT= 0xFF; // 1111 1111b
  Activate PushPull for P4.7-4
  P4MDOUT   = 0xF0;// 1111 0000b
  //Bit0+1: Skipped Pins for INT0/INT1
  //Bit6+7: Skipped Pins for Quartz
  P0SKIP= 0xC3;// 1100 0011b
  //Bit1+2: Skipped Pins for Anal. Inputs
  P1SKIP= 0x06;// 0000 0110b
  // Bit0: Enable UART0 on Crossbar
  // Bit2: Enable SMBus on Crossbar
  XBR0= 0x05; //0000 0101b
  // 0100 0000b Enable Crossbar for Pi
  XBR1= 0x40;
}
```

Schema des Mikrocontrollers 8051 [BLE94]



3.1.1 Register

Register sind ausgezeichnete Speicherplätze im Prozessor mit einem ausgezeichneten Zweck. Der Prozessor 8051 ist so organisiert, dass die Register auch als Teil des internen RAMs angesprochen werden können, d.h. über eine hexadezimale Adresse. Die Lage der Register im RAM-Bereich erfordert jedoch eine gewisse Sorgfalt bei der Programmierung. Werden z.B. Feldadressierungsarten verwendet, so muss sichergestellt sein, dass benötigte Registerinhalte nicht überschrieben werden.

Erklärung der Register:

- **Befehlszähler (PC)** zeigt auf die nächste aktuelle Programmzeile
- **Akkumulator (ACC)** Alle Operationen der ALU werden mit diesem Rechenregister durchgeführt
- **Registerbänke 0 – 3**
  Jede Registerbank besitzt 8 Register R0 – R7
  Die aktuelle Registerbank wird über RS1 und RS0 im Programmzustandsregister (PSW) selektiert.
  (RS = Register Select)

  Zweck:
  - Operanden in der CPU halten
  - Schneller Transport von Daten zur ALU
  - Einfache Operationen können direkt mit den Registern durchgeführt werden:

    Datentransport
    Zählen
    Vergleichen

  R0 und R1 können bei der indirekten Adressierung einer Operandenadresse eingesetzt werden.

Datenzeiger DPTR

- Der Datenzeiger wird bei der indirekten Adressierung zur Erstellung einer 16 Bit Adresse verwendet.
- Das Register kann inkrementiert, aber nicht dekrementiert werden.
- Verwendbar ggf. als Zähler.
  Jedoch ist der Test auf Überlauf sehr mühsam.
- Der Datenzeiger DPTR kann als die Zusammenfassung zweier unterschiedlicher Register betrachtet werden. Aufteilung in DPH und DPL mit jeweils 8 Bit.

Speicherplatz B

Der Speicherplatz B ist als Eingangsspeicher speziell für Multiplikations- und Divisionsoperationen gedacht. Es stellt hierbei zunächst einen Operanden zur Verfügung und dient dann zur Aufnahme eines Teils des Ergebnisses.

Multiplikation	MUL AB	
1. Operand	→	Akku
2. Operand	→	B
Ergebnis LSB	→	AKKU
Ergebnis MSB	→	B
Division AW/BW	DIV AB	
AW	→	AKKU
BW	→	B
Ergebnis Quotient	→	AKKU
Ergebnis Rest	→	B

Stapelzeiger (SP = Stack Pointer)

Nach dem Einschalten des Prozessors oder nach einem Reset enthält der Stackpointer Adresse 07h des internen RAM´s.

07h Beginn des Stacks

Verwendung bei:

- Unterprogrammen
- Programmunterbrechungen (Interrupts)

zum Ablegen der Rücksprungadresse oder anderer Daten.

Schreiben von Daten (mit CALL oder PUSH)
Aktion:

(SP) = (SP) + 1
((SP)) ← Datum
(SP) = 07h → 08h (Vorsicht! Das ist Bereich der Registerbank 1)

Holen von Daten (mit RET I; RET; POP)
Aktion:

Ziel ← ((SP))
(SP) ← (SP) - 1

Der Stack kann an einen anderen Speicherbereich verlegt werden, wenn der Stackpointer mit einer anderen Anfangsadresse geladen wird.

(SP) <- Anfangsadresse – 1 (da zuerst inkrementiert und dann geschrieben wird)

Interrupt Register

An dieser Stelle sollen diese Register nur genannt werden. Sie sind Gegenstand weiterer Erklärungen bei der Behandlung von ereignisgesteuerten Unterbrechungen des laufenden Programms in späteren Kapiteln.

Interruptfreigaberegister (Interrupt Enable Register)
Interruptprioritätsregister (Interrupt Priority Register)
Interrruptquellen:

$\overline{INTO}$ und $\overline{INT1}$ extern
Zähler
Serielle Schnittstelle
(Analog Digital Wandler)
(Capture Compare Unit)

Energiespar Register

Das Energiesparregister dient zur Steuerung des Prozessorverhaltens, wenn kaum oder keine Aktivitäten vom Prozessor erwartet werden. Wichtig sind solche Betriebszustände zur Schonung von Batterien oder Akkumulatoren, um längere Betriebszeiten zu gewährleisten.

Power Down Mode:

- Oszillator ausgeschaltet
- RAM wird mit geringem Strom versorgt
- Beenden mit Hardware RESET

Idle Mode:

- Die CPU wird vom Oszillator getrennt, aber andere periphere Einheiten können aktiv sein.
- Beenden durch Interrupts oder Hardware RESET

3.1.2 Programmspeicher

Je nach Version des Prozessors stehen unterschiedliche interne Programmspeichergrößen zur Verfügung. Für kleinere Anwendungen können beispielweise 4 KByte ausreichend sein. Durch die Steuerung mittels des Anschlusses EA kann zwischen dem internen und dem externen Programmspeicher umgeschaltet werden. Die Verwendung des ausschließlich inneren Programmspeichers ist von Vorteil wenn viele Ein-/und Ausgänge benötigt werden. Die Steuerung von externen Speichermedien sowohl für Daten als auch für Programme erfordert die Bereitstellung von 16 oder gar 24 definierten Aus-/Eingabeleitungen, die nur schwer für andere Zwecke verwendet werden können.

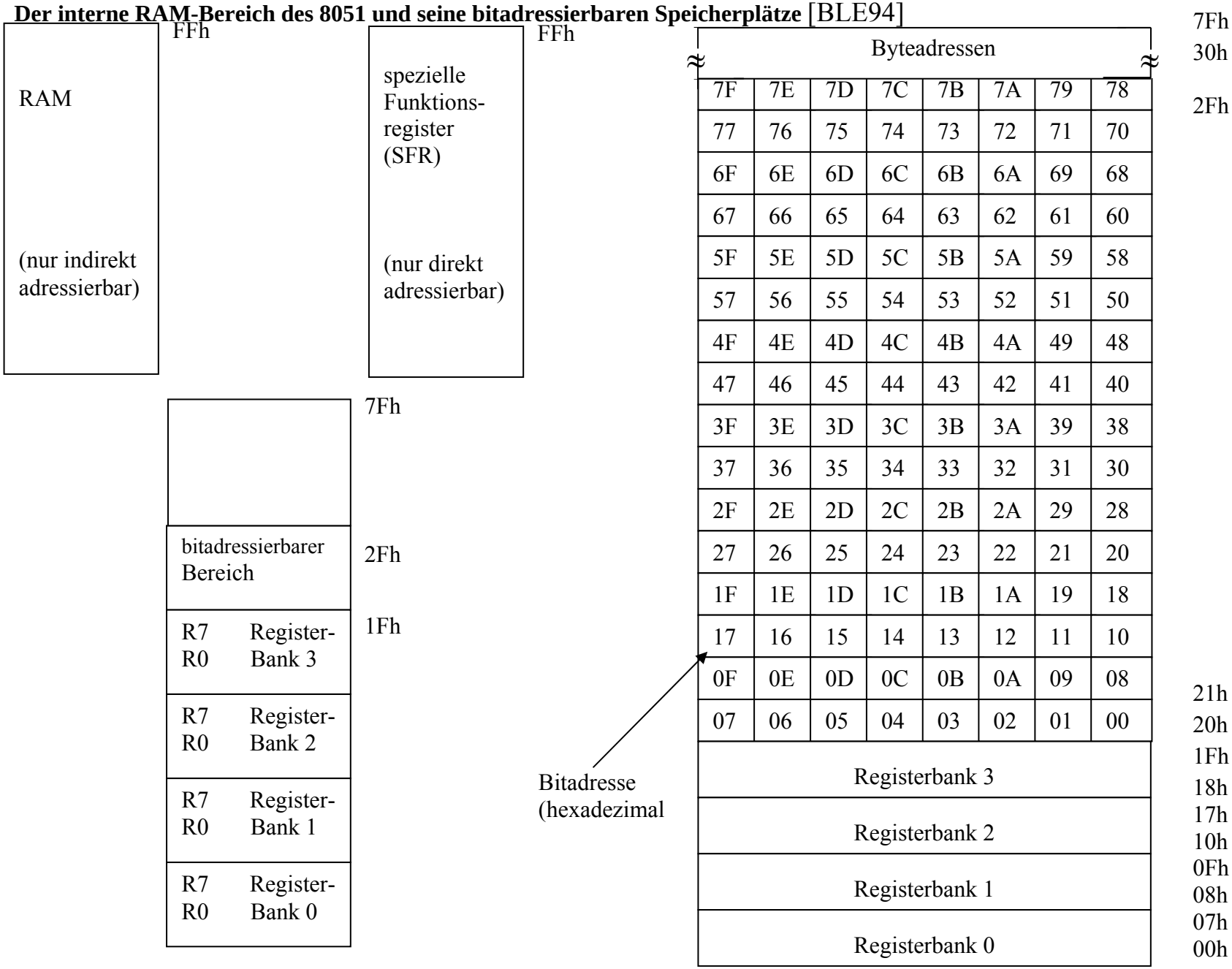
Zur Organisation des Programmspeichers ist es notwendig einige grundsätzliche Prozessoreigenschaften zu kennen:

Nach dem Starten des Prozessors entweder durch Reset oder durch Anschalten der Versorgungsspannung wird der Programmzähler auf 0000h gesetzt. D.h. der Programmablauf beginnt mit dem Befehl an dieser Adresse.
Zum Behandeln von Unterbrechungen (Interrupts) sind Einsprungadressen der Interrupt Service Routinen (das sind spezielle Unterprogramme) auf niedrige Adressen gelegt (z.B. 0003h).
Da hiermit die Programmlänge des startenden Programms stark eingeschränkt ist. Steht an der Adresse 0000h meist nur ein Sprungbefehl zum eigentlichen Programm.

3.1.3 Datenspeicher

Der interne Datenspeicher ist in seinem Adressraum zunächst auf 256 Byte beschränkt. Die Speicherplätze, die direkt über eine Adresse angesprochen werden können, beinhalten ebenso spezielle Register (z.B. Akku), die nur bedingt für allgemeine Speicheranwendungen zur Verfügung stehen. Bei der Programmierung ist daher darauf zu achten, dass diese sogenannten Special Function Register nicht ungewollt überschrieben werden. Neben dem Ansprechen eines Bytes durch eine Adresse wurde für einen bestimmten Bereich die Möglichkeit geschaffen, auf einzelne Bits zu zugreifen. Das ermöglicht die einfachere Handhabung von Kennzeichenflags, die tatsächlich nur zwei Zustände zu beschreiben haben z.B. LED-Anzeige An/Aus.
Durch die Verwendung des vorhandenen Speicherraums auch für spezielle Funktionen wird der vorhanden Speicher doch erheblich reduziert und steht auch nicht unbedingt in einem durchgehenden Stück zur Verfügung.
Wird beispielsweise ein Stack verwendet, ist diese Forderung unerlässlich.
Durch Einfügung eines weiteren Speicherbereichs mit 128 Byte wird diese Situation entspannt. Da jedoch nur ein Adressenbereich von 256 Byte zur Verarbeitung angesprochen werden kann, wurde durch die Verwendung unterschiedlicher Adressierungsarten der Zugriffskonflikt gelöst.
Der Bereich des internen RAMs mit den Adressen 0..7Fh kann sowohl direkt als auch indirekt adressiert werden. Bei der indirekten Adressierung wird die Adresse vorher in einem Register abgelegt. (siehe folgende Kapitel).
Für den Bereich von 80h bis FFh wird jetzt nach der Adressierungsart unterschieden. Bei der direkten Adressierung wird dann auf einen anderen Bereich zugegriffen als bei der indirekten Adressierung. Im direkten Bereich liegen weiterhin die Special Function Register mit der Möglichkeit eines schnellen beliebig steuerbaren Zugriffs. Der indirekte Bereich ist dafür geeignet beispielweise den Stack aufzunehmen oder zum Aufbau von Datenfeldern auf die mittels eines Indexes zugegriffen wird.
Das Ein- und Auslesen von Daten wird ebenfalls durch Ansprechen des Speichers vorgenommen. In der Regel werden 8 Bit zusammen bearbeitet. Diese Zusammenfassung wird dann als Port bezeichnet. Die Ports haben ebenso den Charakter von Special Function Registern und können über Namen angesprochen werden (z.B. P3 spricht den gesamten Port an, P3.1 spricht die zweite Leitung des Ports P3 an, Beginn der Zählung bei 0).
Die Aufteilung des RAM-Bereiches in seine angesprochenen Teile, sowie die Lage und die Bedeutung der Special Function Register ist in den nachfolgenden Bildern dargestellt.

Der interne RAM-Bereich des 8051 und seine bitadressierbaren Speicherplätze [BLE94]



F8	SPI0CN	PCA0L	PCA0H	PCA0CPL0	PCA0CPH0	PCA0CPL4	PCA0CPH4	VDM0CN
F0	B	P0MDIN	P1MDIN	P2MDIN	P3MDIN	P4MDIN	EIP1	EIP2
E8	ADC0CN	PCA0CPL1	PCA0CPH1	PCA0CPL2	PCA0CPH2	PCA0CPL3	PCA0CPH3	RSTSRC
E0	ACC	XBR0	XBR1	XBR2	IT01CF	SMOD1	EIE1	EIE2
D8	PCA0CN	PCA0MD	PCA0CPM0	PCA0CPM1	PCA0CPM2	PCA0CPM3	PCA0CPM4	P3SKIP
D0	PSW	REF0CN	SCON1	SBUF1	P0SKIP	P1SKIP	P2SKIP	USB0XCN
C8	TMR2CN	REG0CN	TMR2RLL	TMR2RLH	TMR2L	TMR2H	-	-
C0	SMB0CN	SMB0CF	SMB0DAT	ADC0GTL	ADC0GTH	ADC0LTL	ADC0LTH	P4
B8	IP	CLKMUL	AMX0N	AMX0P	ADC0CF	ADC0L	ADC0H	-
B0	P3	OSCXCN	OSCICN	OSCICL	SBRLL1	SBRLH1	FLSCL	FLKEY
A8	IE	CLKSEL	EMI0CN	-	SBCON1	-	P4MDOUT	PFE0CN
A0	P2	SPI0CFG	SPI0CKR	SPI0DAT	P0MDOUT	P1MDOUT	P2MDOUT	P3MDOUT
98	SCON0	SBUF0	CPT1CN	CPT0CN	CPT1MD	CPT0MD	CPT1MX	CPT0MX
90	P1	TMR3CN	TMR3RLL	TMR3RLH	TMR3L	TMR3H	USB0ADR	USB0DAT
88	TCON	TMOD	TL0	TL1	TH0	TH1	CKCON	PSCTL
80	P0	SP	DPL	DPH	EMI0TC	EMI0CF	OSCLCN	PCON
	0(8)	1(9)	2(A)	3(B)	4(C)	5(D)	6(E)	7(F)
	bitadressierbar							

Special Function Register beim C8051F340 [DB1]

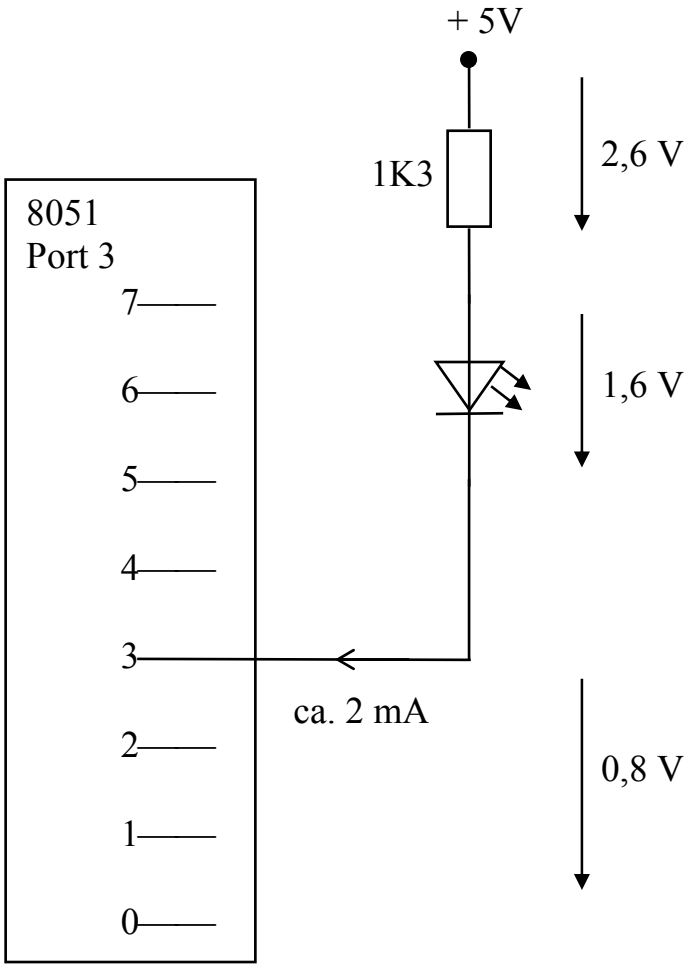
Alphabetische Auflistung der Special Function Register

Register	Address	Description
ACC	0xE0	Accumulator
ADC0CF	0xBC	ADC0 Configuration
ADC0CN	0xE8	ADC0 Control
ADC0GTH	0xC4	ADC0 Greater-Than Compare High
ADC0GTL	0xC3	ADC0 Greater-Than Compare Low
ADC0H	0xBE	ADC0 High
ADC0L	0xBD	ADC0 Low
ADC0LTH	0xC6	ADC0 Less-Than Comp. Word High
ADC0LTL	0xC5	ADC0 Less-Than Comp. Word Low
AMX0N	0xBA	AMUX0 Negative Channel Select
AMX0P	0xBB	AMUX0 Positive Channel Select
B	0xF0	B Register
CKCON	0x8E	Clock Control
CLKMUL	0xB9	Clock Multiplier
CLKSEL	0xA9	Clock Select
CPT0CN	0x9B	Comparator0 Control
CPT0MD	0x9D	Comparator0 Mode Selection
CPT0MX	0x9F	Comparator0 MUX Selection
CPT1CN	0x9A	Comparator1 Control
CPT1MD	0x9C	Comparator1 Mode Selection
CPT1MX	0x9E	Comparator1 MUX Selection
DPH	0x83	Data Pointer High
DPL	0x82	Data Pointer Low
EIE1	0xE6	Extended Interrupt Enable 1
EIE2	0xE7	Extended Interrupt Enable 2
EIP1	0xF6	Extended Interrupt Priority 1
EIP2	0xF7	Extended Interrupt Priority 2
EMI0CN	0xAA	External Memory Interface Control
EMI0CF	0x85	External Memory Interface Config.
EMI0TC	0x84	External Memory Interface Timing
FLKEY	0xB7	Flash Lock and Key
FLSCL	0xB6	Flash Scale
IE	0xA8	Interrupt Enable
IP	0xB8	Interrupt Priority
IT01CF	0xE4	INT0/INT1 Configuration
OSCICL	0xB3	Internal Oscillator Calibration
OSCICN	0xB2	Internal Oscillator Control
OSCLCN	0x86	Internal Low-Frequency Osc.Control
OSCXCN	0xB1	External Oscillator Control
P0	0x80	Port 0 Latch
P0MDIN	0xF1	Port 0 Input Mode Configuration
P0MDOUT	0xA4	Port 0 Output Mode Configuration
P0SKIP	0xD4	Port 0 Skip
P1	0x90	Port 1 Latch
P1MDIN	0xF2	Port 1 Input Mode Configuration
P1MDOUT	0xA5	Port 1 Output Mode Configuration
P1SKIP	0xD5	Port 1 Skip
P2	0xA0	Port 2 Latch
P2MDIN	0xF3	Port 2 Input Mode Configuration
P2MDOUT	0xA6	Port 2 Output Mode Configuration
P2SKIP	0xD6	Port 2 Skip
P3	0xB0	Port 3 Latch
P3MDIN	0xF4	Port 3 Input Mode Configuration
P3MDOUT	0xA7	Port 3 Output Mode Configuration
P3SKIP	0xDF	Port 3Skip
P4	0xC7	Port 4 Latch
P4MDIN	0xF5	Port 4 Input Mode Configuration
P4MDOUT	0xAE	Port 4 Output Mode Configuration
PCA0CN	0xD8	PCA Control
PCA0CPH0	0xFC	PCA Capture 0 High
PCA0CPH1	0xEA	PCA Capture 1 High
PCA0CPH2	0xEC	PCA Capture 2 High
PCA0CPH3	0xEE	PCA Capture 3High
PCA0CPH4	0xFE	PCA Capture 4 High

Register	Address	Description
PCA0CPL0	0xFB	PCA Capture 0 Low
PCA0CPL1	0xE9	PCA Capture 1 Low
PCA0CPL2	0xEB	PCA Capture 2 Low
PCA0CPL3	0xED	PCA Capture 3 Low
PCA0CPL4	0xFD	PCA Capture 4 Low
PCA0CPM0	0xDA	PCA Module 0 Mode Register
PCA0CPM1	0xDB	PCA Module 1 Mode Register
PCA0CPM2	0xDC	PCA Module 2 Mode Register
PCA0CPM3	0xDD	PCA Module 3 Mode Register
PCA0CPM4	0xDE	PCA Module 4 Mode Register
PCA0H	0xFA	PCA Counter High
PCA0L	0xF9	PCA Counter Low
PCA0MD	0xD9	PCA Mode
PCON	0x87	Power Control
PFE0CN	0xAF	Prefetch Engine Control
PSCTL	0x8F	Program Store R/W Control
PSW	0xD0	Program Status Word
REF0CN	0xD1	Voltage Reference Control
REG0CN	0xC9	Voltage Regulator Control
RSTSRC	0xEF	Reset Source Configuration/Status
SBCON1	0xAC	UART1 Baud Rate Generator Control
SBRLH1	0xB5	UART1 Baud Rate Generator High
SBRLL1	0xB4	UART1 Baud Rate Generator Low
SBUF1	0xD3	UART1 Data Buffer
SCON1	0xD2	UART1 Control
SCON0	0x98	UART0 Control
SBUF0	0x99	UART0 Data Buffer
SMB0CF	0xC1	SMBus Configuration
SMB0CN	0xC0	SMBus Control
SMB0DAT	0xC2	SMBus Data
SMOD1	0xE5	UART1 Mode
SP	0x81	Stack Pointer
SPI0CFG	0xA1	SPI Configuration
SPI0CKR	0xA2	SPI Clock Rate Control
SPI0CN	0xF8	SPI Control
SPI0DAT	0xA3	SPI Data
TCON	0x88	Timer/Counter Control
TH0	0x8C	Timer/Counter 0 High
TH1	0x8D	Timer/Counter 1 High
TL0	0x8A	Timer/Counter 0 Low
TL1	0x8B	Timer/Counter 1 Low
TMOD	0x89	Timer/Counter Mode
TMR2CN	0xC8	Timer/Counter 2 Control
TMR2H	0xCD	Timer/Counter 2 High
TMR2L	0xCC	Timer/Counter 2 Low
TMR2RLH	0xCB	Timer/Counter 2 Reload High
TMR2RLL	0xCA	Timer/Counter 2 Reload Low
TMR3CN	0x91	Timer/Counter 3Control
TMR3H	0x95	Timer/Counter 3 High
TMR3L	0x94	Timer/Counter 3Low
TMR3RLH	0x93	Timer/Counter 3 Reload High
TMR3RLL	0x92	Timer/Counter 3 Reload Low
VDM0CN	0xFF	VDD Monitor Control
USB0ADR	0x96	USB0 Indirect Address Register
USB0DAT	0x97	USB0 Data Register
USB0XCN	0xD7	USB0 Transceiver Control
XBR0	0xE1	Port I/O Crossbar Control 0
XBR1	0xE2	Port I/O Crossbar Control 1
XBR2	0xE3	Port I/O Crossbar Control 2

Programmbeispiel:

An einem Beispiel soll demonstriert werden, wie die Funktionseinheiten im Modell angesprochen und zu den gewünschten Aktionen veranlasst werden. Hierzu soll eine einfache Ausgabeprozedur mit folgendem Verhalten betrachtet werden:
An Port 3 sollen angeschlossene LEDs nacheinander aufleuchten (Lauflicht)
Die Anschlüsse des Ports sind alle, wie im Bild gezeigt, mit einer LED beschaltet.
Man beachte, dass die LED an ist, wenn eine „0“ am Ausgang anliegt.



Die Aufgabenstellung soll jetzt schrittweise in ein fertiges Programm umgesetzt werden. Dazu werden nacheinander folgende Beschreibungen erstellt.

- Verbale Beschreibung der Rechenvorschrift (Algorithmus)
- Struktogramm
- C-Programm
- Assembler Programm

Verbale Beschreibung:

Aktionen:

Initialisierung:

Lade Akku mit 1111 1110

Ständig

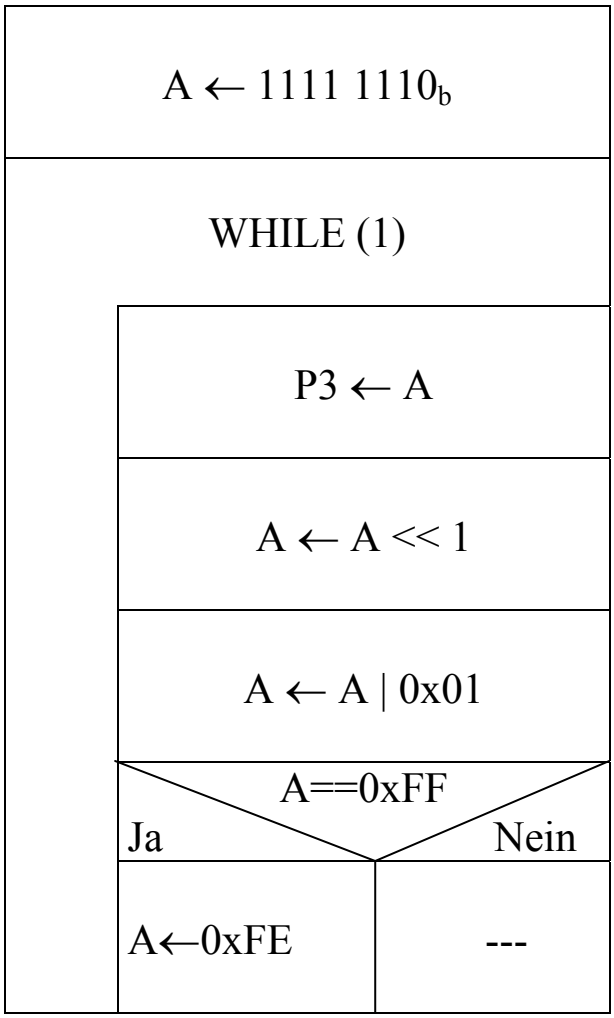
Akkuinhalt zum Port bringen
Verschiebe Akku nach links und ziehe eine 1 nach

T1 1111 1110
T2 1111 1101

Wenn A=1111 1111 dann A=1111 1110

END Ständig

Struktogramm:



Linksverschiebung

Bitweises Oder

C-Code:

```
void main (void)
{
        A = 0xFE;                       /* Initialisierung      */
        While (1){                      /* Endlosschleife       */
                P3 = A;                 /* Portausgabe          */
                A = A << 1;             /* Linksverschiebung    */
                A = A | 0x01;           /* Eins nachziehen      */
                If (A==0xFF) A = 0xFE;  /* Neu initialisieren   */
        }
}
```

Assembler:

```
        MOV A, # 1111 1110b   ; Akku initialisieren
                              ; Bit 0 an Port 3 bringt
                              ; LED 0 zum Leuchten
LOOP:   MOV P3, A             ; Daten aus dem AKKU nach Port 3 laden
        RL A                  ; Akku nach links verschieben A.7 → A.0
                              ; Ringshift !
        JMP LOOP              ; Sprungbefehl nach Loop
                              ; Endlosschleife
```

Die programmtechnischen Realisierungen unterscheiden sich nicht nur in der Sprache, sondern auch in der Anzahl der Befehle. Dies hat seine Ursache in den unterschiedlichen Befehlsausführungen, die sich aus der Definition der C-Konstrukte und der Assemblerbefehle für die Schiebeoperation ergeben.
In C wird beim Schieben nach links nur eine Null nachgezogen. Das hat zur Folge, dass die im Programmbeispiel benötigte Eins zum Löschen der LED zusätzlich erzeugt werden muss.
Der Schiebebefehl RL für den 8051 rotiert die Bits im Akkumulator, so dass das höchstwertigste Bit wieder als niederwertigstes Bit erscheint. Es ergibt sich so direkt das erwünschte Muster zur Realisierung des Lauflichtes.
Ebenso ersichtlich ist der Aufbau der Schleife mit einem Sprungbefehl in der Assemblerprogrammierung. Die Forderungen der strukturierten Programmierung werden hier nicht mehr von der Programmiersprache unterstützt, sondern liegen in der Verantwortung des Programmierers.

3.2 Befehlssatz

3.2.1 Struktur der Befehle

Die zur Verfügung stehenden Befehle für den Prozessor 8051 lassen sich in 1,2 und 3 Byte-Befehle aufteilen [BLE94]. Das erste Byte enthält dabei auf jeden Fall den Operationscode also die Information was getan werden soll. Die nächsten Bytes dienen zur Angabe von Quell- und Zieldaten. Wenn also ein Byte für den Operationscode verwendet werden kann, so sind maximal 256 Befehle kodierbar. Meist werden jedoch nicht alle Bitkombinationen ausgeschöpft.

Verwendet werden etwa 50 - 90 Befehle + Adressierungsarten.
Zu jedem Bitmuster wird ein mnemonischer Code eingeführt.

Programme werden erstellt:

- Im mnemonischen Code (Assembler)
- In höheren Programmiersprachen z.B. C

Befehlsaufbau:
Ein Befehl enthält die Angaben was womit durchgeführt werden soll. Es muss daher die Funktion kodiert sein und der Zugriff auf eventuelle Operanden definiert werden.
Den Zugriff auf den/die Operand(en) bestimmt die Adressierungsart. Unter einer Adressierungsart versteht man das Verfahren zur Ortsbestimmung des Operanden im Speicherbereich. Die einfachste Art ist die direkte Angabe der Adresse. Zur Realisierung effektiver Lösungsalgorithmen sind jedoch auch andere Adressierungsarten interessant. Eine weitere standardisierte Zugriffsart findet man bei der Verwendung von Feldern. Hier wird beginnend von einer Basisadresse die eigentliche Adresse durch die Addition eines Indexes ermittelt. In den folgenden Abschnitten werden die regelmäßig verwendeten Adressierungsarten behandelt und ihre Notation in Assemblerschreibweise dargelegt.

3.2.2 Adressierungsarten

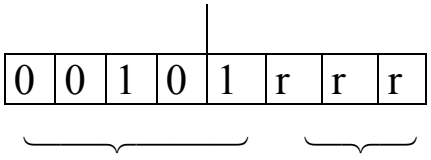
3.2.2.1 Implizite Adressierung (Implied Addressing)

Manche Instruktionen benötigen zur Durchführung keine Daten aus dem Speicher, sondern aus den CPU internen Registern.
Da hier jedoch meist keine großen Adressräume angesprochen werden müssen, genügt die Adressierung über wenige Bits.
Zum Einsatz kommen im 8051 Registerbänke mit jeweils 8 Registern die mit R_r angesprochen werden (r=0..7)

z.B. ADD A, R_r
↑ ↑
Ziel Quelle

Funktion: Addiere Registerinhalt R_r zum AKKU.

Aufgabe des Befehlsbytes:



eigentlicher OpCode

Adressteil

Bedeutung des Adressteils:
000 = R0
001 = R1
:
111 = R7

Je nach adressiertem Register lautet der Additionsbefehl daher:

28_h, 29_h, $2A_h$, $2B_h$, $2C_h$, $2D_h$, $2E_h$, oder $2F_h$

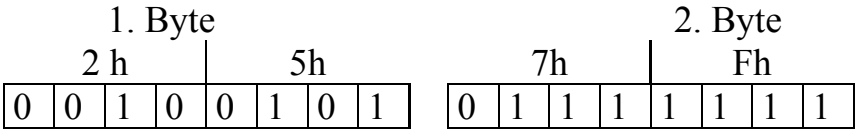
3.2.2.2 Direkte oder absolute Adressierung (Direct Addressing)

Die interne Speicheradresse wird direkt angegeben. Ein Zugriff auf den externen Speicher ist so nicht möglich (siehe MOVX).

z.B.: ADD A, $07F_h$; Addiere zum Akku den Inhalt des Speichers $7F_h$

Der Befehlscode besteht aus zwei Byte

Operationscode ADD A →25h, direkte Operandenadresse 7Fh



3.2.2.3 Unmittelbare Adressierung (Immediate Addressing)

Bei dieser Form Operanden anzugeben, handelt es sich nicht direkt um eine Adressenauswertung, vielmehr wird das zu verarbeitende Bitmuster direkt angegeben.

Operanden sind:

- Konstante
- Bitmuster
- Festgelegte Zeichen

Sie folgen direkt dem Operationscode

z.B. MOV A, #100 ; laden des AKKUs mit der Dezimalzahl 100
; also 0110 0110b

Beim Anwenden der unmittelbaren Adressierung auf 8-Bit-Register folgt dem Operationscode 1 Byte.
Beim Anwenden auf 2-Byte-Register z.B. DPTR folgen 2 Byte.
(DPTR in das Register, das die Adresse zum Zugriff auf den externen Speicher enthält.)

3.2.2.4 Indirekte (Indirect) Adressierung

Bei dieser Adressierungsart wird in einem Register die Operationsadresse abgelegt, z.B. DPTR.
Der Operationscode enthält:

- Die Kennung der indirekten Adressierung
- Welches Register die Adresse enthält
  → kurzer Operationscode

Zweck:
Eine Adressberechnung kann durchgeführt werden. Z.B. der Zugriff auf eine Tabelle mittels einer fortlaufenden Adresse.

Die aktuelle Adresse ergibt sich dann z.B.:

Aktuelle_Adresse = Vorherige_Adresse + 1

Die indirekte Adressierung wird beim 8051 mit Hilfe von speziellen Registern durchgeführt.

Zum Zugriff auf den internen RAM:

@R1, @R0, SP → PUSH, POP – Operationen

Zum Zugriff auf den Externen RAM:

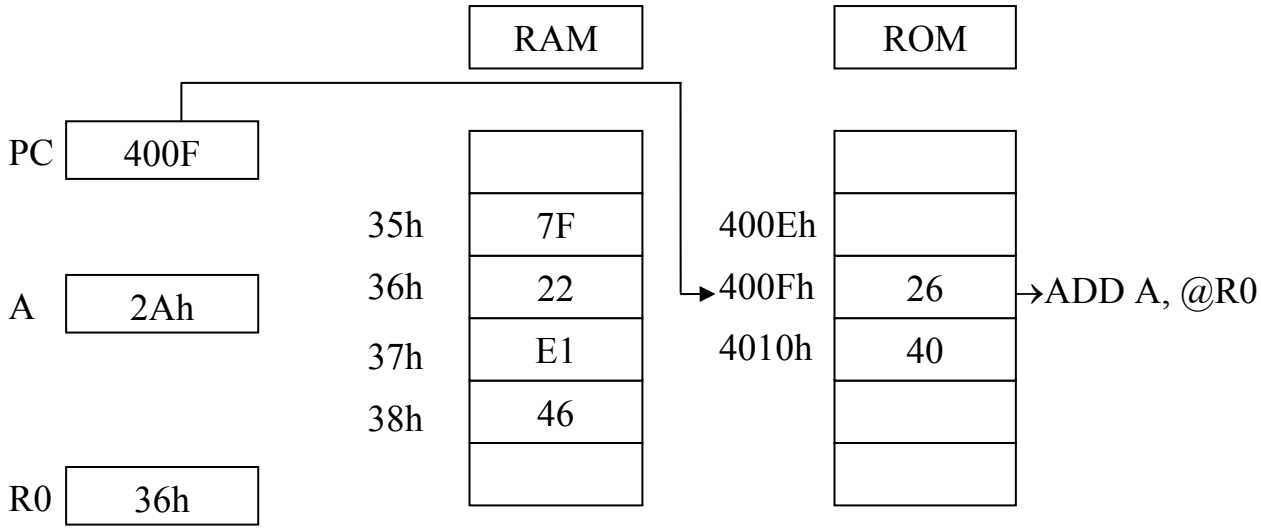
@R1, @R0, @DPTR

Da die Register R1, R0 nur 256 Speicherplätze auswählen können, müssen vorher die Informationen über den Speicherbereich am Port 2 gesetzt werden.
Für den C8051F340 muss das EMI0CN Register geladen werden.

Beispiel für den indirekten Zugriff auf den internen RAM:

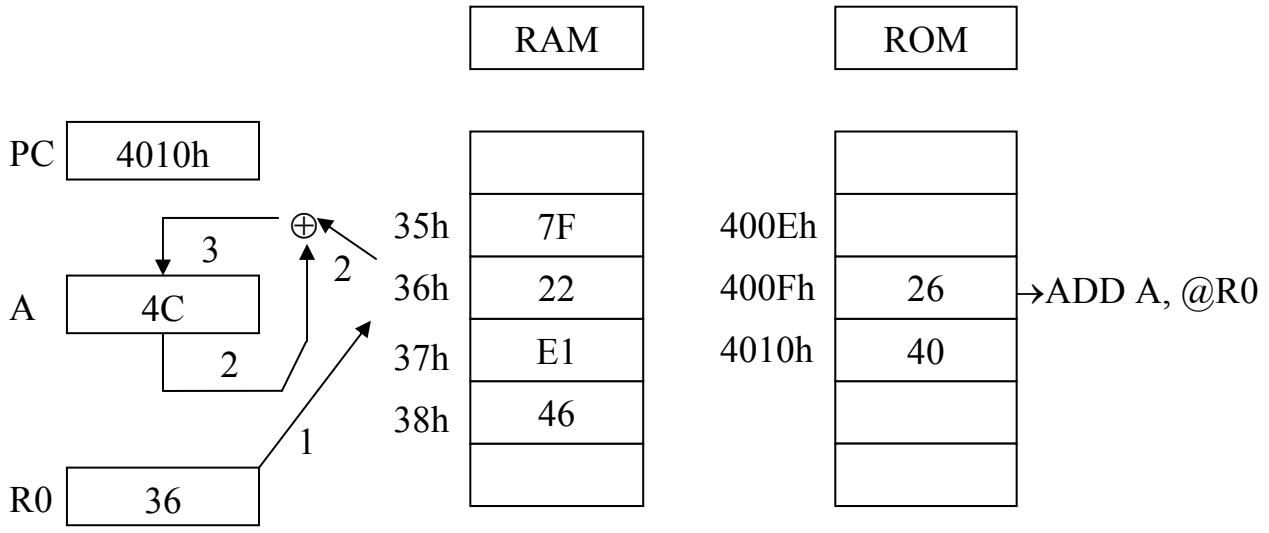
```
ADD A, @R0          ; addiere zum AKKU Inhalt das Datum, dessen
                    ; Adresse in R0 steht.
```

Ausgangsstellung



Nach dem Holen des Operationscodes vom Speicherplatz 400F wird die Adresse 36h aus R0 an das (interne) RAM ausgegeben. Der dort unter Adresse 36 gespeicherte Wert (22h) wird nun zum Akkumulator addiert und die Flags neu eingestellt.

Registerinhalte nach Ausführung des Befehls:



Die Erweiterung der indirekten Adressierung um einen zusätzlichen Index führt zur indizierten Adressierung

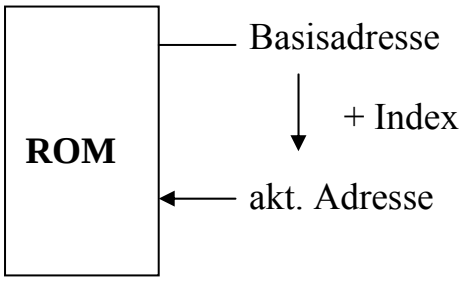
3.2.2.5 Indizierte Adressierung (Indexed Addressing)

Bei der indizierten Adressierung wird die aktuelle Adresse aus einer Basisadresse und einem Index berechnet. Diese Zugriffsart ist beim Prozessor 8051 nur beim Zugriff auf den Programmspeicher beispielsweise zum Lesen von Konstanten möglich. Die Basisadresse wird im Datenpointer (DPTR) oder Programcounter (PC) Register abgelegt. Der Index muss im Akkumulator gespeichert werden. Das Basisregister und der Index werden addiert, auf den Speicher zugegriffen und das Ergebnis im Akkumulator abgelegt, d.h. der ursprüngliche Index geht verloren.

Berechnung der effektiven Adresse

Basisadresse → DPTR
(alternativ) ↘ PC
+ Indexvorgabe →AKKU

akt. Adresse



Bsp.: MOVC A, @A + DPTR
MOVC A, @A + PC

3.2.2.6 Relative Adressierung

Diese Adressierungsart wird zum Zugriff auf Daten für Prozessoren des Typs 8051 nicht verwendet.
Sie hat jedoch ihre Bedeutung bei Sprungbefehlen:
Relativ zur gegenwärtigen Position. (Inhalt des PC); wird zu einem anderen Punkt in der Nähe verzweigt. Es sind Sprünge vorwärts und rückwärts möglich.
Die Relativadresse ist eine vorzeichenbehaftete Zahl im Bereich

+127 ≥ Adr. ≥ -128.

Beispiel: JC #8h

Programmspeicher		Relativ-adressen	
24FFh			
2500h	JC	-2	
2501h	+8	-1	
2502h		0	← Bezugsadresse
2503h		1	
....		...	
250Ah		8	← Ziel

3.2.3 Beschreibung der Befehle

Der gesamte Befehlssatz lässt sich in verschiedene Weise unterteilen. Klassifiziert man nach der Funktion [BLE94], dann gibt es

- arithmetische Befehle
- logische Befehle
- Verschiebebefehle
- Datentransportbefehle
- Befehle zur Bitverarbeitung (Boole'sche Operationen)
- Verzweigungs- und Sprungbefehle
- Befehle zum Einsatz von Unterprogrammen

Zur Darstellung der Operationen noch einige Bemerkungen:

Adr bezeichnet eine Programmspeicheradresse (normal hat sie 16 Bit, falls eine verkürzte Variante gemeint ist, wird die Bitzahl angegeben, z.B. Adr 11 = (A10,A9..A0)

Dadr ist eine Adresse im internen RAM oder die eines Funktionsregisters (SFR), also eine 1-Byte-Adresse

Badr ist eine Bitadresse im internen RAM (00h bis 7Fh) oder in einem SFR (80h bis FFh)

Daten	bedeutet eine 8-Bit-Konstante
Datenl6	ist eine 16-Bit-Konstante
←	gibt an, wohin der Operand transportiert wird
⇔	kennzeichnet einen Austausch der Inhalte
A	Akkumulator
B	Register B (spezielles Funktionsregister auf Adresse F0h)
C	andere Bezeichnung für das Carry-Bit bei boole'schen Operationen
DPTR	Datenzeiger (Data Pointer)
Rr	eines der Arbeitsregister RO bis R7 der aktuellen Bank
Ri	Register RO oder R1 der aktuellen Bank (zur indirekten Adressierung benutzt)
SP	Stapelzeiger (Stack Pointer)
(X)	bezeichnet den Inhalt des Speicherplatzes oder des Registers X
((X))	bezeichnet den Inhalt des Speicherplatzes, auf den der Inhalt des Registers X zeigt
@	kennzeichnet die indirekte Adressierung
#	ist das Kennzeichen für unmittelbare Adressierung
rel	steht anstelle einer Relativadresse
/	bedeutet die Negation eines Bits
∧	logisches UND
v	logisches ODER
exor	Exklusiv-ODER-Verknüpfung

Alle Befehle sind in der allgemein gebräuchlichen anglo-amerikanischen Bezeichnung wiedergegeben. Die Operationscodes für die Befehle sind hier nicht angeführt, da sie sich entsprechend der Adressierungsart ändern!

3.2.3.1 Arithmetische Befehle

Mnemonischer Code	a) anglo-amerikanische Bezeichnung b) deutsche Beschreibung c) Operation	veränderte Zustandsbits (Flags)

ADD A, <Quellbyte> **ADD TO ACCUMULATOR** [CY,OV,AC,P]
Addiere ein Byte zum Akkumulator
A ← (A) + <Quellbyte>
Anmerkung: Der Operand <Quellbyte> kann hier und im folgenden durch eine der angegebenen Adressierungsarten festgelegt werden, das heisst durch:
ADD A, (7Fh) (direkte Adressierung)
ADD A, R7 (Registeradressierung)
ADD A,#127 (unmittelbare Adressierung)
ADD A, @R1 (indirekte Adressierung)

ADDC A, <Quellbyte> **ADD TO ACCUMULATOR WITH CARRY** [CY,OV,AC,P]
Addiere ein Byte und den Übertrag zum Akkumulator
A ← (A) + <Quellbyte> + C

SUBB A, <Quellbyte> **SUBTRACT FROM ACCUMULATOR WITH BORROW**
Subtrahiere vom Akku ein Byte und den Übertrag [CY,OV,AC,P]
A ←(A) - <Quellbyte> -C

INC A **INCREMENT ACCUMULATOR** [P]
Erhöhe Akkumulatorinhalt um 1
A ← (A) + 1

INC <Zielbyte> **INCREMENT BYTE**
Erhöhe das Zielbyte um 1
<Zielbyte> ← <Zielbyte> + 1

INC DPTR **INCREMENT DATA POINTER**
Erhöhe Datenzeiger um 1
DPTR ←DPTR + 1

Beachte: Es gibt keine DEC DPTR-Funktion! Der Datenzeiger kann nur dadurch vermindert werden, dass man sein unteres Byte (auf der direkten SFR-Adresse DPL) vermindert und, falls es vom Wert 00h auf FFh übergeht, zusätzlich das höhere Byte (DPH) dekrementiert.

DEC A **DECREMENT ACCUMULATOR** [P]
Vermindere Akkumulatorinhalt um 1
A ←(A) -1

DEC <Zielbyte> **DECREMENT BYTE**
Vermindere das Zielbyte um 1
<Zielbyte> ←<Zielbyte> - 1

MUL AB **MULTIPLY ACCUMULATOR AND B** [CY, 0V, P]
Multipliziere Akkumulatorinhalt und Inhalt Register B
(B,A) ← (A)*(B)

DIV AB **DIVIDE ACCUMULATOR THROUGH B** [CY, 0V, P]
Dividiere Akkumulator durch Registerinhalt B
A ← Int[A/B] (= Quotient), B ← Mod[A/B] (= Rest)
In A steht also das (ganzzahlige) Divisionsergebnis, in B der Rest

DA A **DECIMAL ADJUST ACCUMULATOR** [CY, P]
Dezimalkorrektur des Akkumulatorinhalts
Dies ist keine Dual - BCD - Wandlung!
Der Befehl ist nicht nach (BCD-)Subtraktionen anwendbar!

3.2.3.2 Logische Befehle

Mnemonischer Code	a) anglo-amerikanische Bezeichnung b) deutsche Beschreibung c) Operation	veränderte Zustandsbits (F1ags)
ANL A, <Quellbyte>	**LOGICAL AND TO ACCUMULATOR** UND-Verknüpfung des Akkumulatorinhalts mit einem Byte A←(A) ∧ Quellbyte> Anmerkung: hier wie auch bei dem entsprechenden ORL- und XRL-Befehl kann der Operand <Quellbyte> mit direkter, indirekter, Register-oder unmittelbarer Adressierung angegeben werden.	[P]
ANL Dadr, A	**LOGICAL AND WITH ACCUMULATOR** UND-Verknüpfung eines Datenspeicherinhalts mit dem Akkumulator Dadr ← (Dadr) ∧ (A)	
ANL Dadr, #Daten	**LOGICAL AND WITH DATA** UND-Verknüpfung eines Datenspeicherinhalts mit einem 8-Bit-Datum Dadr ←(Dadr) ∧ Daten	
ORL A, <Quellbyte>	**LOGICAL OR TO ACCUMULATOR** ODER-Verknüpfung des Akkumulatorinhalts mit einem Byte A ← (A) v <Quellbyte>	[P]
ORL Dadr, A	**LOGICAL OR WITH ACCUMULATOR** ODER-Verknüpfung eines Datenspeicherinhalts mit dem Akkumulator Dadr ← (Dadr) v (A)	
ORL Dadr, #Daten	**LOGICAL OR WITH DATA** ODER-Verknüpfung eines Datenspeicherinhalts mit einem 8-Bit-Datum Dadr ← (Dadr) v Daten	
XRL A, <Quellbyte>	**LOGICAL EXCLUSIVE OR TO ACCUMULATOR** Exclusiv-ODER-Verknüpfung des Akkumulatorinhalts mit einem Byte. A ← (A) exor <Quellbyte>	[P]
XRL Dadr, A	**LOGICAL EXCLUSIVE OR WITH ACCUMULATOR** Exklusiv-ODER-Verknüpfung eines Datenspeicherinhalts mit dem Akkumulator. Dadr←(Dadr) exor (A)	
XRL Dadr, #Daten	**LOGICAL EXCLUSIVE OR WITH DATA** Exklusiv-ODER-Verknüpfung eines Datenspeicherinhalts mit einem 8-Bit-Datum. Dadr ←(Dadr) exor Daten	
CLR A	**CLEAR ACCUMULATOR** Lösche Akkumulatorinhalt A← 00	[P]
CPL A	**COMPLEMENT ACCUMULATOR** Invertiere den Akkumulatorinhalt A←NICHT(A)	[P]
NOP	**NO OPERATION** Leerbefehl, der für kurze zeitliche Verzögerungen im Programm oder als Platzhalter für spätere Modifikationen und Ergänzungen verwendet wird.	

3.2.3.3 Verschiebebefehle

Mnemonischer Code	a) anglo-amerikanische Bezeichnung b) deutsche Beschreibung c) Operation.	veränderte Zustandsbits (Flags)

RL A **ROTATE ACCUMULATOR LEFT**
Rotiere Akkumulatorinhalt nach links um 1 Bit
$A_{n+1} \leftarrow (A_n)$ und $A_0 \leftarrow (A_7)$

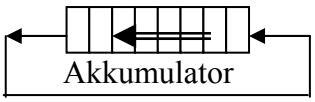
RLC A **ROTATE ACCUMULATOR LEFT THROUGH Carry** [CY, P]
Rotiere Akkumulatorinhalt durch das Carry nach links um 1 Bit
$A_{n+1} \leftarrow (A_n)$ und $A_0 \leftarrow$ (CY) und CY $\leftarrow (A_7)$

RR A **ROTATE ACCUMULATOR RIGHT**
Rotiere Akkumulatorinhalt nach rechts um 1 Bit
$A_n \leftarrow (A_{n+1})$ und $A_0 \leftarrow (A_0)$An ~ (An+i)

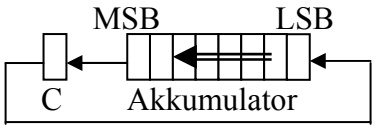
RRC A **ROTATE ACCUMULATOR RIGHT THROUGH Carry** [CY, P]
Rotiere Akkumulatorinhalt durch das Ca£ry nach rechts um 1 Bit
$A_n \leftarrow (A_{n+1})$ und $A_7 \leftarrow$ (CY) und CY $\leftarrow A_0$

Die Rotationsbefehle können grafisch veranschaulicht werden:

RL A

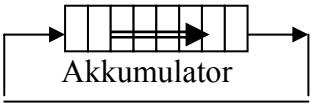


RLC A

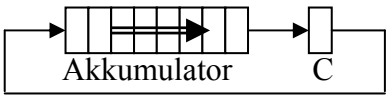


Diese Operation eignet sich besonders für serielle Aus- oder Eingabe mit dem MSB zuerst!
Auch duale Multiplikation mit 2 wird erreicht, wenn das CARRY anfangs auf 0 gebracht wurde.

RR A



RRC A



RRC dient sinngemäß zur Ein- und Ausgabe mit dem LSB zuerst, ebenso zur dualen Division durch 2.

SWAP A **SWAP higher and lower Bits in ACCUMULATOR**
Tausche die Akkumulator-Halbbytes
$(A)_{0-3}$ $(A)_{4-7}$

Dieser Befehl dient vorzugsweise zur Bearbeitung von BCD-Zahlen. Beispielsweise kann eine Dualzahl, die kleiner als 100_{10} ist, leicht mit folgenden Befehlen in eine BCD-Zahl konvertiert werden:

```
MOV  B, #10 ;Teiler 10
DIV  AB     ;ergibt Zehner in A, Einerrest in B, jeweils rechtsbündig
SWAP A      ;Zehner stehen linksbündig in A, das untere Halbbyte ist 0
ADD  A, B   ;Einer werden zu den Zehnern addiert → BCD-Zahl im Akkumulator!
```

3.2.3.4 Datentransfer-Befehle

Mnemonischer Code.	a) anglo-amerikanische Bezeichnung b) deutsche Beschreibung c) Operation	veränderte Zustandsbits (F1ags)

MOV A, <Quelle> **MOVE DATA TO ACCUMULATOR** [P]

Hole Datum aus angegebener Quelle in den Akkumulator
A ←<Quellbyte>
Anmerkung: Der Operand <Quellbyte> kann durch eine der angegebenen Adressierungsarten festgelegt werden, das heisst durch

```
MOV A, (80h)     (direkte Adressierung)
MOV A, R1        (Registeradressierung)
MOV A, #80h      (unmittelbare Adressierung)
MOV A, @Rl       (indirekte Adressierung)
```

MOV <Ziel>, A **MOVE DATA FROM ACCUMULATOR TO DESTINATION**

Speichere Akkumulatorinhalt im angegebenen Ziel
<Zielbyte> ← A
Alle Adressierungsarten außer der unmittelbaren sind möglich

MOV <Ziel>,<Quelle> **MOVE DATA FROM SOURCE TO DESTINATION**

Speichere Akkumulatorinhalt im angegebenen Ziel
<Zielbyte> ← <Quellbyte>
Alle Adressierungsarten sind möglich. Das Quellbyte kann durch seine absolute Adresse, seinen unmittelbaren Wert, die Angabe eines Registers oder indirekt über Register 0 oder 1 angegeben werden. Dies gilt auch für das Ziel, wobei aber die unmittelbare Adressierung ausscheidet. Mögliche Kombinationen:

		<Ziel>		
		Dadr	Rr	@R0 oder @R_1
<Quelle>	#Daten	*	*	*
	Dadr	*	*	*
	Rr	*		
	@R0,@R1	*		

MOV DPTR, #l6Daten **MOVE 16 BIT DATA TO DATA POINTER**

DPTR ←(16 Bit-Datenwort)
Der DPTR wird mit einem 16-Bit-Wort geladen. Üblicherweise wird er so auf eine externe Speicheradresse (16 Bit!) eingestellt.

MOVX A, @DPTR **MOVE FROM EXTERNAL RAM TO ACCUMULATOR** [P]

A←((DPTR))
Aus dem externen RAM (siehe Bild 3.4.1) wird ein Byte in den Akkumulator geholt. Hierzu gibt es eine verkürzte Variante:

MOVX A, @Ri **MOVE FROM EXTERNAL RAM TO ACCUMULATOR** [P]

A ← ((R0 oder R1))
Aus dem externen RAM wird ein Byte in den Akkumulator geholt. Da RO bzw. RI aber nur 8 Bit Adressen (als A_0 bis A_7) ausgeben können, müssen die höheren Adressleitungen des externen RAM z.B. über einzelne Anschlüsse des Port 2 gesteuert werden. Damit wird gewissermaßen die Seite im RAM eingestellt.

MOVX @DPTR, A **MOVE ACCUMULATOR DATA TO EXTERNAL RAM**

((DPTR))←A
Speichere Akkumulatorinhalt in das externe RAM

Datentransfer-Befehle (Fortsetzung)

Mnemonischer Code	a) anglo-amerikanische Bezeichnung b) deutsche Beschreibung c) Operation	veränderte Zustandsbits (FIags)

MOVX @Ri, A **MOVE ACCUMULATOR DATA TO EXTERNAL RAM**
((Ri)) ←A
Speichere Akkuinhalt in den externen RAM, wobei die höheren Adressleitungen des RAM vorab z.B. über Port 2 eingestellt werden müssen.

Beachte: Bei allen Zugriffen auf den externen RAM ist der Akkumulator entweder die Quelle oder das Ziel des bewegten Datenbytes. Nur im Zusammenhang mit den MOVX-Befehlen werden die RD- und WR- Anschlüsse des 8051 gesteuert.

MOVC A, @A+DPTR MOVE CONSTANT FROM ROM-ADDRESS TO ACCU [P]
Hole die Programminformation aus der ROM-Adresse (A+DPTR)
A ← ((A+DPTR))
Dieser Befehl wird vor allem für das Lesen von maximal 256 Byte langen Festwerttabellen im ROM gebraucht, wobei der DPTR auf den Tabellenanfang zeigt.
Dieser Befehl und der folgende steuert die PSEN-Leitung beim Zugriff. Ein entsprechender Schreibbefehl existiert natürlich nicht.

MOVC A, @A+PC **MOVE CONSTANT FROM ROM-ADDRESS TO ACCUMULATOR** [P]
Hole die Programminformation aus ROM-Adresse (A+PC), wobei PC die Adresse des nächstfolgenden Programmbefehls ist.
A ← ((A+PC))

XCH A, <Byte> **EXCHANGE ACCUMULATOR WITH BYTE** [P]
Vertausche den Inhalt des Akkumulators mit einem Datenspeicher.
A ⇔ (Dadr) oder A ⇔((Ri)) oder A ⇔ (Rr)
Der Tauschpartner kann direkt, indirekt oder als Registerinhalt adressiert werden.

XCHD A. @Ri **EXCHANGE DIGIT** [P]
Vertausche die unteren 4 Bit des Akkumulators mit einem Datenspeicher
$(A)_{3\text{-}0}$ $((Ri))_{3\text{-}0}$

PUSH Dadr **PUSH DATA ONTO STACK**
Der Stapelzeiger wird um 1 erhöht, dann der Inhalt des Datenspeichers auf den STACK gebracht.
SP←(SP) + 1, dann (SP) ←(Dadr)

POP Dadr **POP DATA FROM STACK**
Ein Byte wird vom STACK geholt und in die Datenspeicher gebracht. Anschließend wird der Stapelzeiger um 1 erniedrigt. PUSH und POP sind komplementär zueinander.
(Dadr) ←((SP)), dann SP←(SP) - 1

3.2.3.5 Befehle zur Bitverarbeitung (Boole'sche Operationen)

Mnemonischer Code	a) anglo-amerikanische Bezeichnung b) deutsche Beschreibung c) Operation	veränderte Zustandsbits (F1ags)

Im 8051-Prozessor kann das Carry-FIag wie ein 1-Bit-Akkumulator aufgefasst werden. Mit ihm und mit einem weiteren Bit sind eine Reihe einfacher Operationen durchführbar, die ähnlich der byteweisen Datenverarbeitung aus Transportbefehlen, Setzen, Löschen, Invertieren, logischen Befehlen und bedingten Verzweigungen bestehen. Der zweite Operand kann einer der bitweise adressierbaren Speicherplätze im internen RAM oder in einem speziellen Funktionsregister (SFR) sein, beispielsweise eine der Ein-/Ausgabeleitungen.
Alle Bitadressen sind absolute (direkte) Adressen. 00h. ...7Fh sind im unteren Teil des RAMs, Adressen 80h...FFh gehören zu Bits in den Funktionsregistern.

ANL C, Badr	**LOGICAL AND BIT WITH CARRY** UND-Verknüpfung des Carry-Bits mit einem Bit C ←(C) ∧ (Badr)		[CY]
ANL C, /Badr	**LOGICAL AND NOT BIT WITH CARRY** UND-Verknüpfung des Carry-Bits mit einem negierten Bit C ← (C) ∧ Nicht(Badr)		[CY]
ORL C, Badr	**LOGICAL OR BIT WITH CARRY** ODER-Verknüpfung des Carry-Bits mit einem Bit C ← (C) v (Badr)		[CY]
ORL C, /Badr	**LOGICAL OR NOT BIT WITH CARRY** ODER-Verknüpfung des Carry-Bits mit einem negierten Bit C ← (C) v Nicht(Badr)		[CY]
MOV C, Badr	**MOVE BIT TO CARRY** Bringe Bit ins Carry-Bit	C ← (Badr)	[CY]
MOV Badr, C	**MOVE CARRY TO BIT** Bringe das Carry-Bit in ein Bit	(Badr) ← C	
SETB C	**SET CARRY** Setze Carry-Bit	C← 1	[CY = 1]
SETB Badr	**SET BIT** Setze Bit	(Badr) ← 1	
CLR C	**CLEAR CARRY** Lösche Carry-Bit	C←0	[CY =0]
CLR Badr	**CLEAR BIT** Lösche Bit	(Badr) ← 0	
CPL C	**COMPLEMENT CARRY** Invertiere Carry-Bit	C ← Nicht(C)	[CY]
CPL Badr	**COMPLEMENT BIT** Invertiere Bit	(Badr) ← (Badr)	

3.2.3.6 Verzweigungs- und Sprungbefehle

Mnemonischer Code	a) anglo-amerikanische Bezeichnung b) deutsche Beschreibung c) Operation	veränderte Zustandsbits (F1ags)

JC rel **JUMP IF CARRY IS SET**
Verzweige, falls Carry = 1, Sonst keine Operation
PC←(PC)+rel, faIls CY=1

JNC rel **JUMP IF CARRY IS NOT SET**
Verzweige, falls Carry 0, sonst keine Operation
PC←(PC)+rel, falls CY=0

JB Badr, rel **JUMP IF BIT IS SET**
Verzweige, wenn Bit = 1, sonst keine Operation
PC←(PC)+rel, wenn (Badr)=1

JNB Badr, rel **JUMP IF BIT IS NOT SET**
Verzweige, wenn Bit = 0, sonst keine Operation
PC ←(PC) + rel, wenn (Badr) =0

JBC Badr, rel **JUMP IF BIT IS SET AND CLEAR BIT**
Verzweige, wenn Bit = 1 und lösche es dann, sonst keine Operation
PC ←(PC) + rel und (Badr) ← 0, wenn (Badr) = 1

Ein internes Flag kann leicht über einen Portanschluss ausgegeben werden:

```
MOV  C,Flag
MOV  P1.0,C
```

In diesem Beispiel ist Flag irgendein adressierbares Bit im RAM oder einem SFR. Es wird hier über eine Ein-/Ausgabeleitung (das LSB des Port 1) ausgegeben. Je nach Zustand des (internen) Flags zeigt sich nun an P1.0 eine 1 oder eine 0.

Es fällt auf, dass unter den logischen Operationen das Exklusive ODER fehlt. Diese Operation kann aber leicht per Software nachgebildet werden. Beispielsweise kann man
C = Bitl exor Bit2 wie folgt realisieren:

```
      MOV  C, Bit1          ;Hole Bitl
      JNB  Bit2, BLEIBT     ;Ist Bit2 = 0, ergibt Bit1 exor 0 = Bitl
      CPL  C                ;ansonsten ergibt Bitl exor 1 = Bit1.
BLEIBT:  (weiteres Programm)
```

Beachte: Da auch das Zustandsregister bitadressierbar ist, können die einzelnen Flags des PSW getestet und zu Sprungentscheidungen herangezogen werden!

Verzweigungs- und Sprungbefehle (Fortsetzung)

Mnemonischer Code	a) anglo-amerikanische Bezeichnung b) deutsche Beschreibung c) Operation	veränderte Zustandsbits (F1ags)

JMP Adr JUMP TO ADDRESS
Springe zur angegebenen Adresse im Programmspeicher
Diesen Befehl gibt es zwar im Befehlssatz nicht, doch kann man ihn verwenden und dem Assembler die Umsetzung in einen der drei folgenden echten Befehle überlassen. Die (Relativ-)Adressen werden in der Regel in Form einer Marke (LABEL) angegeben, seltener als Hexadezimalzahl. Die tatsächlichen Sprungbefehle lauten SJMP, LJMP und AJMP

SJMP rel SHORT JUMP TO RELATIVE ADDRESS
Verzweige innerhalb von +127...-128 Adressen relativ zur gegenwärtigen Programmadresse.
PC ←(PC) + rel

LJMP Adrl6 LONG JUMP TO ADDRESS
Springe zur 16-Bit-Absolutadresse (= innerhalb des gesamten 64K-Raumes)
PC← AdrI6

AJMP Adr11 ABSOLUTE JUMP TO ADDRESS
Springe zur 11-Bit-Absolutadresse innerhalb des gegenwärtig benutzten 2K-Raumes im Programmspeicher.
$PC_{10\text{-}0}$ ←Adr11, $PC_{15.11}$ bleiben unberührt!

JMP @A+DPTR JUMP INDIRECT
PC←(A) + (DPTR)

Dieser Befehl erlaubt Vielfachverzweigungen bzw. ein Berechnen des Sprungziels. Die Zieladresse ist die Summe aus dem Wert im DPTR und der Variablen im Akkumulator. Typisch setzt man den DPTR auf die Anfangsadresse einer Sprungtabelle, während der Akkumulator den Index (= Nummer des Sprungs) enthält. Bei 5 Sprungzielen beispielsweise kann der Akkumulator die Zahlen 0 bis 4 enthalten. Die Programmbefehle lauten dann:

```
        MOV DPTR, #JMPTAB       ;DPTR auf Tabellenanfangsadresse
        MOV A, INDEX            ;Sprungnummer holen
        RL A            ;mal 2 (da ein Sprungbefehl in der Tabelle 2 Byte lang!)
        JMP @A+DPTR     ;Springe zum Sprungbefehl in der Tabelle!
JNPTAB: AJMP FALL0
        AJMP FALL1      ;Jeder dieser Sprungbefehle ist 2 Bytes lang und führt
        AJMP FALL2      ;zum eigentlichen Sprungziel FALL0 bis FALL4
        AJMP FALL3
        AJMP FALL4
```

JZ rel JUMP IF ZERO
Verzweige, wenn Akkumulatorinhalt = 0
PC ← (PC) + rel

JNZ rel JUMP IF NOT ZERO
Verzweige, wenn Akkumulatorinhalt != 0
PC ← (PC) + rel

DJNZ <Byte>, rel DECREMENT AND JUMP IF NOT ZERO [CY]
Vermindere das Datenbyte (im Register oder im Datenspeicher) um 1 und verzweige, wenn != 0
Rr ← (Rr) - 1; wenn (Rr) != 0. dann PC ← (PC) + rel bzw.
Dadr ← (Dadr) - 1; wenn (Dadr) != 0, dann PC ← (PC) + rel

Verzweigungs- und Sprungbefehle (Fortsetzung)

Mnemonischer Code	a) anglo-amerikanische Bezeichnung b) deutsche Beschreibung c) Operation	veränderte Zustandsbits (F1ags)

CJNE <Zielbyte>, <Quellbyte>, rel COMPARE AND JUMP IF NOT EQUAL [CY]

CJNE vergleicht die Größe zweier Operanden. Sind beide Operanden ungleich, so wird zur Adresse: aktuelle Adresse + rel verzweigt. Weiter wird das Carry Flag CY gesetzt, wenn das Zielbyte kleiner ist als das Quellbyte. Damit lassen sich alle mögliche Vergleiche zwischen zwei Operanden durchführen.

```
            CJNE  R7,#60h, NOT_EQ
            … ….                        ;R7==60h
NOT_EQ:     JC    REQ_LOW               ; Springe nach REQ_LOW wenn R7< 60h
            … ….                        ;R7>60h

REQ_LOW:    … ….                        ;R7 < 60h
```

Folgende Adressarten sind erlaubt:

```
CJNE  A,    direct_adress,rel
CJNE  A,    #data   ,rel
CJNE  Rn,   #data   ,rel
CJNE  @Ri,  #data   ,rel
```

Beispiel: Es soll überprüft werden, ob der Wert im Akkumulator der folgenden Bedingung genügt: 25<A<30. Das Ergebnis soll in F1 im PSW abgespeichert werden.

```
            CLR   F1
            CJNE  A,#25, NOT_EQ1
            JMP   L1                ;A==25
NOT_EQ1:    JC    LOW1
            JMP   W1                ;A>25
LOW1:       JMP   L1                ;A<25
W1:         CJNE  A,#30, NOT_EQ2
            JMP   L1                ;A==30
NOT_EQ2:    JC    LOW2
            JMP   L1                ;A>30
LOW2:       SETB  F1                ;A<30
L1:         …
```

Optimiert:

```
            CLR   F1
            CJNE  A,#25, NOT_EQ1
            JMP   L1                ;A==25
NOT_EQ1:    JC    L1                ;A<25
            CJNE  A,#30, NOT_EQ2    ;A>25
            JMP   L1                ;A==30
NOT_EQ2:    JNC   L1                ;A>30
            SETB  F1                ;A<30
L1:         …
```

Befehlsliste mit der Angabe der verwendete Taktzyklen [DA1]

Mnemonic	Description	Bytes	Clock Cycles
Arithmetic Operations			
ADD A, Rn	Add register to A	1	1
ADD A, direct	Add direct byte to A	2	2
ADD A, @Ri	Add indirect RAM to A	1	2
ADD A, #data	Add immediate to A	2	2
ADDC A, Rn	Add register to A with carry	1	1
ADDC A, direct	Add direct byte to A with carry		2
ADDC A, @Ri	Add indirect RAM to A with carry	1	2
ADDC A, #data	Add immediate to A with carry	2	2
SUBB A, Rn	Subtract register from A with borrow	1	1
SUBB A, direct	Subtract direct byte from A with borrow	2	2
SUBB A, @Ri	Subtract indirect RAM from A with borrow	1	2
SUBB A, #data	Subtract immediate from A with borrow	2	2
INC A	Increment A	1	1
INC Rn	Increment register	1	1
INC direct	Increment direct byte	2	2
INC @Ri	Increment indirect RAM	1	2
DEC A	Decrement A	1	1
DEC Rn	Decrement register	1	1
DEC direct	Decrement direct byte	2	2
DEC @Ri	Decrement indirect RAM	1	2
INC DPTR	Increment Data Pointer	1	1
MUL AB	Multiply A and B	1	4
DIV AB	Divide A by B	1	8
DA A	Decimal adjust A	1	1
Logical Operations			
ANL A, Rn	AND Register to A	1	1
ANL A, direct	AND direct byte to A	2	2
ANL A, @Ri	AND indirect RAM to A	1	2
ANL A, #data	AND immediate to A	2	2
ANL direct, A	AND A to direct byte	2	2
ANL direct, #data	AND immediate to direct byte	3	3
ORL A, Rn	OR Register to A	1	1
ORL A, direct	OR direct byte to A	2	2
ORL A, @Ri	OR indirect RAM to A	1	2
ORL A, #data	OR immediate to A	2	2
ORL direct, A	OR A to direct byte	2	2
ORL direct, #data	OR immediate to direct byte	3	3
XRL A, Rn	Exclusive-OR Register to A	1	1
XRL A, direct	Exclusive-OR direct byte to A	2	2
XRL A, @Ri	Exclusive-OR indirect RAM to A	1	2
XRL A, #data	Exclusive-OR immediate to A	2	2
XRL direct, A	Exclusive-OR A to direct byte	2	2
XRL direct, #data	Exclusive-OR immediate to direct byte	3	3
CLR A	Clear A	1	1
CPL A	Complement A	1	1
RL A	Rotate A left	1	1
RLC A	Rotate A left through Carry	1	1
RR A	Rotate A right	1	1
RRC A	Rotate A right through Carry	1	1
SWAP A	Swap nibbles of A	1	1

Befehlsliste mit der Angabe der verwendete Taktzyklen [DA1] (Fortsetzung)

Mnemonic	Description	Bytes	Clock Cycles
Data Transfer			
MOV A, Rn	Move Register to A	1	1
MOV A, direct	Move direct byte to A	2	2
MOV A, @Ri	Move indirect RAM to A	1	2
MOV A, #data	Move immediate to A	2	2
MOV Rn, A	Move A to Register	1	1
MOV Rn, direct	Move direct byte to Register	2	2
MOV Rn, #data	Move immediate to Register	2	2
MOV direct, A	Move A to direct byte	2	2
MOV direct, Rn	Move Register to direct byte	2	2
MOV direct, direct	Move direct byte to direct byte	3	3
MOV direct, @Ri	Move indirect RAM to direct byte	2	2
MOV direct, #data	Move immediate to direct byte	3	3
MOV @Ri, A	Move A to indirect RAM	1	2
MOV @Ri, direct	Move direct byte to indirect RAM	2	2
MOV @Ri, #data	Move immediate to indirect RAM	2	2
MOV DPTR, #data16	Load DPTR with 16-bit constant	3	3
MOVC A, @A+DPTR	Move code byte relative DPTR to A	1	3
MOVC A, @A+PC	Move code byte relative PC to A	1	3
MOVX A, @Ri	Move external data (8-bit address) to A	1	3
MOVX @Ri, A	Move A to external data (8-bit address)	1	3
MOVX A, @DPTR	Move external data (16-bit address) to A	1	3
MOVX @DPTR, A	Move A to external data (16-bit address)	1	3
PUSH direct	Push direct byte onto stack	2	2
POP direct	Pop direct byte from stack	2	2
XCH A, Rn	Exchange Register with A	1	1
XCH A, direct	Exchange direct byte with A	2	2
XCH A, @Ri	Exchange indirect RAM with A	1	2
XCHD A, @Ri	Exchange low nibble of indirect RAM	1	2
Boolean Manipulation			
CLR C	Clear Carry	1	1
CLR bit	Clear direct bit	2	2
SETB C	Set Carry	1	1
SETB bit	Set direct bit	2	2
CPL C	Complement Carry	1	1
CPL bit	Complement direct bit	2	2
ANL C, bit	AND direct bit to Carry	2	2
ANL C, /bit	AND complement of direct bit to Carry	2	2
ORL C, bit	OR direct bit to carry	2	2
ORL C, /bit	OR complement of direct bit to Carry	2	2
MOV C, bit	Move direct bit to Carry	2	2
MOV bit, C	Move Carry to direct bit	2	2
JC rel	Jump if Carry is set	2	2/4
JNC rel	Jump if Carry is not set	2	2/4
JB bit, rel	Jump if direct bit is set	3	3/5
JNB bit, rel	Jump if direct bit is not set	3	3/5
JBC bit, rel	Jump if direct bit is set and clear bit	3	3/5

Befehlsliste mit der Angabe der verwendete Taktzyklen [DA1] (Fortsetzung)

Mnemonic	Description	Bytes	Clock Cycles
Program Branching			
ACALL addr11	Absolute subroutine call	2	4
LCALL addr16	Long subroutine call	3	5
RET	Return from subroutine	1	6
RETI	Return from interrupt	1	6
AJMP addr11	Absolute jump	2	4
LJMP addr16	Long jump	3	5
SJMP rel	Short jump (relative address)	2	4
JMP @A+DPTR	Jump indirect relative to DPTR	1	4
JZ rel	Jump if A equals zero	2	2/4
JNZ rel	Jump if A does not equal zero	2	2/4
CJNE A, direct, rel	Compare direct byte to A and jump if not equal	3	3/5
CJNE A, #data, rel	Compare immediate to A and jump if not equal	3	3/5
CJNE Rn, #data, rel	Compare immediate to Register and jump if not equal	3	3/5
CJNE @Ri, #data, rel	Compare immediate to indirect and jump if not equal	3	4/6
DJNZ Rn, rel	Decrement Register and jump if not zero	2	2/4
DJNZ direct, rel	Decrement direct byte and jump if not zero	3	3/5
NOP	No operation	1	1

Werden bei den Taktzyklen zwei Zahlen angegeben, so gilt die zweite Zahl für den Fall, dass der Sprung ausgeführt wird.

4 Programmiergrundlagen

Die Hauptaufgabe beim Einsatz von Mikrocontrollern ist neben dem Entwurf der Hardware und ihrer Einbettung in die Applikationsumgebung der Softwareentwurf. Im Folgenden werden grundlegende Kenntnisse in der Programmiertechnik vorausgesetzt. Trotzdem sollen in diesem Abschnitt noch mal einfache Grundlagen angesprochen werden und auch in den Zusammenhang der Mikrocontrollerprogrammierung gestellt werden. Anschließend wird die Erstellung von Struktogrammen und Ablaufdarstellungen als Folgediagramme für einfache Anwendungen erläutert.
Bei tiefergehendem Interesse wird auf die Standardliteratur verwiesen.

4.1 Randbedingungen

Die Entwicklung der Kunst, Software zu erstellen (Software Engineering) wird heute von einer umfangreichen Teildisziplin der Informatik vorangetrieben. Das zentrale Anliegen des Software Engineerings ist die Bewältigung von Problemkomplexität. Die Beschreibung eines Softwaresystems in unterschiedlichen Abstraktionsgraden gehört ebenso zu diesem Aufgabengebiet wie auch die Einhaltung von Qualitätsanforderungen und die Ermöglichung von Projektmanagementmethoden.
Die sinnvolle Aufteilung eines Systems in beherrschbare Teilkomponenten ist oft ein Schlüssel zur erfolgreichen Umsetzung einer Problemlösung.

Beschreibungsmethoden des Software Engineering wie Strukturdiagramme, Flussdiagramme, Struktogramme, Objektorientierung gehören bereits zu dem Handwerkszeug, das bei den Programmierungsgrundlagen vermittelt wird.

Die Entwicklung eines Prozessorsystems erfordert jedoch Betrachtungsweisen, die nicht nur die Software zum Gegenstand haben, sondern auch die Hardware mit berücksichtigen. Insbesondere ist eine Aufteilung der Softwarefunktionen und der Hardwarefunktionen vorzunehmen. Die Aufteilung solcher Aufgaben wird heute mit dem Begriff Hard- und Software Codesign belegt und wird im Systemdesign auch mit Hilfe von Hochebenenbeschreibungen behandelt (z.B. VHDL).

4.2 Erstellen und Testen von Quellcodes

Die Erstellung des eigentlichen Programmcodes stellt eine der letzten Stufen im Designprozess dar.
Es werden, entsprechend den gefundenen Aufteilungen, Unterprogramme entwickelt und zu einem Programmpaket zusammengefasst. An dieser Stelle sollten keine wesentlichen Systemfragen mehr entschieden werden müssen. Die Erstellung von Quellcode ist im einfachsten Fall die Beschreibung des Algorithmus in einem Texteditor in der ausgewählten Sprache. Werden Spezifikationen oder Struktogramme bereits mit Rechnerhilfe erstellt, so sind meist hier Programme vorhanden, die den eigentlichen Codierungsschritt automatisch vornehmen können. Ob und wieweit ein solcher Schritt vollständig durchgeführt werden kann, hängt vom Detaillierungsgrad der erstellten Beschreibungen ab.
Liegt der Quellcode vor, so folgen zunächst die Überprüfung der syntaktischen Richtigkeit und der Vollständigkeit. In einer Iterationsschleife werden dann Fehler vom Compiler (Assembler) angezeigt (Listing) und mittels des Texteditors korrigiert. Aus dem korrekten Quellcode wird der Objektcode erstellt. Der Objektcode enthält die Anweisungen, die der Quellcode enthalten hat, in einer Form, die symbolischen Maschinenbefehlen entspricht. Z.B. sind Preprozessoranweisung von C aufgelöst oder konstante Ausdrücke ausgewertet. Die absolute Position der Befehle und die Einsprungadressen sind jedoch noch nicht ermittelt worden. Diese Aufgabe hat der Linker/Locator, der zusätzliche benutzerdefinierte- oder Standardbibliotheken hinzulädt und mit dem eigentlichen Programm verbindet. Erst nachdem alle diese Informationen vorhanden sind, ist es möglich das Programm mit absoluten Programmadressen zu erstellen. Je nach Anwendungszweck liegt aber auch an dieser Stelle noch nicht das fertige lauffähige Ergebnis vor. Gerade bei Mikrocontrollersystemem werden unterschiedliche Ziele zum effektiven Testen angesprochen.
Die reine Simulation des Programms kann mit Hilfe eines „normalen“ Rechners vorgenommen werden. Es existiert hierzu ein Programm, das den umgesetzten Code lesen kann und die geforderte Aktion virtuell ausführt. Die Programme liefern hierzu Information über die gerade ausgeführte Zeile im Quellcode. Weiter besitzen sie ein Modell des Prozessors und können die jeweiligen Register- oder Speicherinhalte darstellen. Mit diesen Simulatoren ist es möglich Rechenoperationen zu testen und in einfacher Form Ein-/Ausgabeoperationen nachzuvollziehen. Realzeitanwendungen, oder Interruptsteuerungen, die auf externe Ereignisse reagieren müssen, sind auf diese Weise nur bedingt zu testen. Die Überprüfung der Funktionsfähigkeit wird dann mit dem System selbst oder mit einem Emulator durchgeführt. Wird das System selbst verwendet, so existiert meist eine Verbindung zu einem Entwicklungssystem, das beispielsweise über die serielle Schnittstelle in der Lage ist, in den Programmspeicher zu schreiben. Ein Monitorprogramm auf dem eigentlichen System sorgt dann für die Rückmeldung des Prozessorstatus an das Entwicklungssystem. Auf diese Weise kann der Programmablauf direkt verfolgt werden. Die Systeme, die zu diesen Tests verwendet werden, stellen im Speicherbereich modifizierte Systeme des eigentlichen Zielsystems dar, da der Programmspeicher vom Entwicklungssystem geladen werden muss. In der beschriebenen Konfiguration ist der Prozessor auf dem System vorhanden und führt zusätzlich das Monitorprogramm aus. Soll jedoch das System tatsächlich in der geplanten Struktur getestet werden, so können Emulatoren eingesetzt werden. Auch hier existiert ein Entwicklungssystem, das jedoch den gesamten Prozessor in seinem Verhalten nachbildet. Es existiert dann ein Adapter der anstelle des Prozessors im Zielsystem eingesetzt wird. Diese Form des Tests ist meist sehr realitätsnah jedoch auch entsprechend kostspielig. Es wird daher, je nach Anwendung, die angepasste Entwicklungsumgebung eingesetzt.

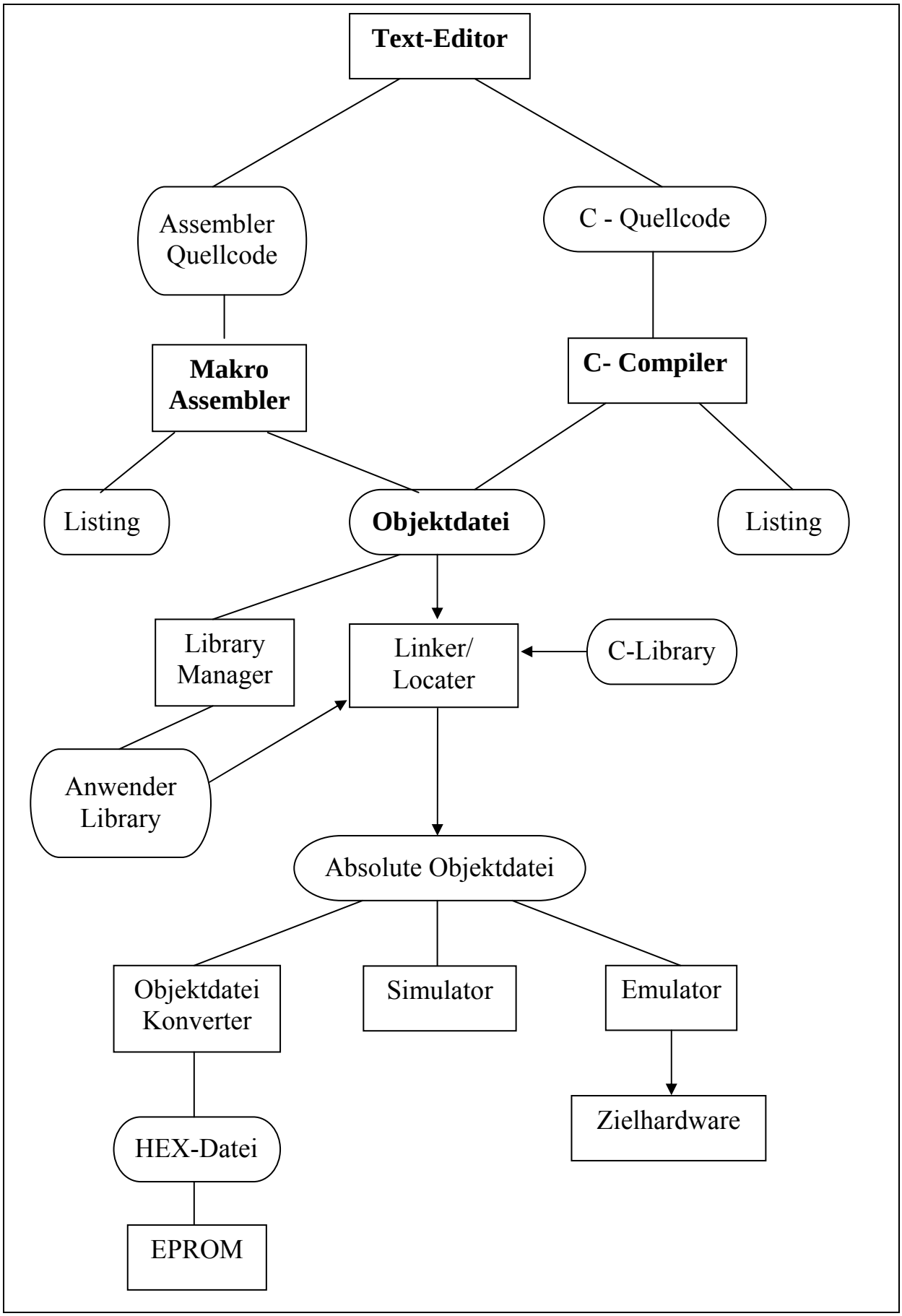
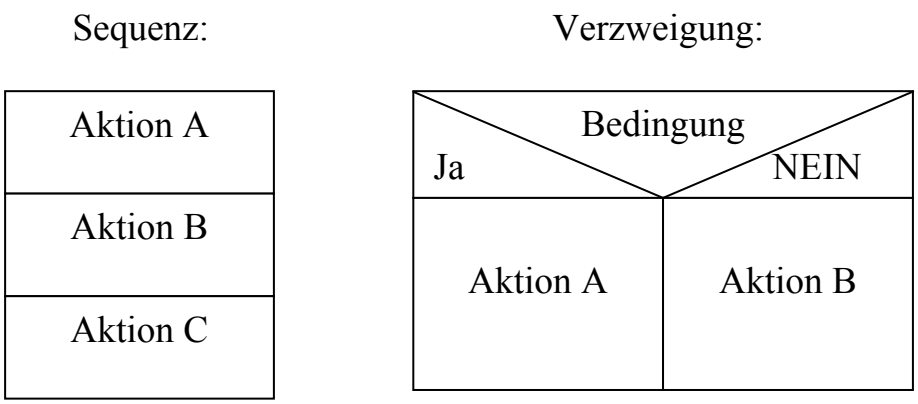
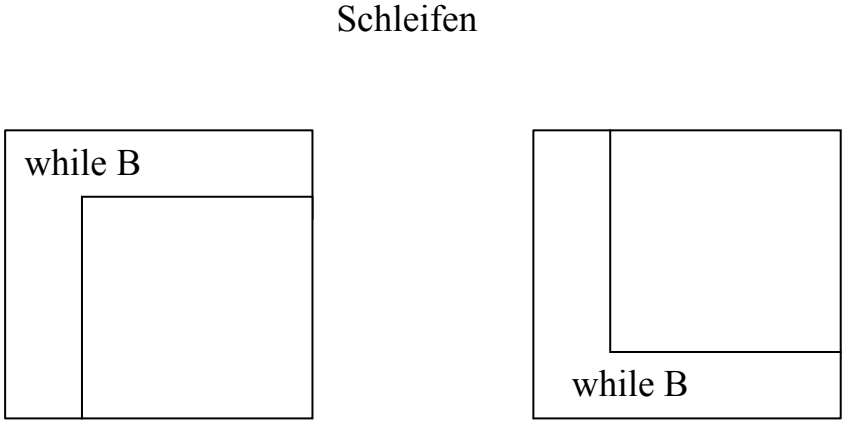


Bild 4.1 Erstellung von Programmcodes

4.3 Darstellung von Struktogrammen

Bei der Entwicklung der Programmiersprache C wurde ein Programmierstil als Grundlage genommen, der als strukturierte Programmierung bekannt ist. Grob beschrieben, wird hier die explizite Anwendung von direkten Sprungbefehlen vermieden. Bei Programmen in Maschinensprache oder auch Assembler kommt man ohne diese Sprungbefehle nicht aus. Wird jedoch der Einsatz der Sprungbefehle analysiert, so kann eine fast ausschließliche Verwendung bei Schleifenkonstruktionen festgestellt werden. Werden jetzt, wie in C vorgesehen, diese Programmstücke in speziellen Schleifenkonstruktionen eingebettet, so sind sie nicht mehr direkt sichtbar. Dies erleichtert die Lesbarkeit von Programmen, da jetzt ein Programm von vorne nach hinten verfolgt werden kann, ohne Sprungbefehle nach hinten betrachten zu müssen. Die sich daraus ergebenden Vorteile sind so eklatant, dass auch bei der Assemblerprogrammierung eine Entwicklungsphase vorgeschaltet wurde, die dem Entwurf der strukturierten Programmierung entspricht. Ein solcher Entwurf kann auch zunächst in C erfolgen und dann eigenhändig in Assembler umgesetzt werden. Eine andere Vorgehensweise verwendet dazu Grafiksymbole, die es erlauben, den Programmablauf programmiersprachenunabhängig zu entwerfen und dann erst in die gewünschte Programmiersprache umzusetzen.
Hierzu wurde eine Anzahl von Vorschriften mit Basissymbolen erstellt, von denen hier aber nur die Wichtigsten vorgestellt werden sollen.





4.2 Grundelemente bei der Erstellung von Struktogrammen

4.4 Ablauffolgediagramme

Ablauffolgediagramme zeigen die Änderung im Programmablauf beim Eintritt von bestimmten Ereignissen. Mikrocontrolleranwendungen ändern ihre Abläufe häufig in Abhängigkeit von äußeren Ereignissen. Da der Zeitpunkt dieser Ereignisse nicht vorhergesehen werden kann, müssen bestimmte Aktionen an ein solches Ereignis geknüpft werden. Zur Darstellung dieser Abläufe sowohl in funktioneller als auch in zeitlicher Hinsicht eignen sich Ablauffolgediagramme. Jede beteiligte Funktion oder jeder Funktionsblock wird durch einen senkrechten Strich dargestellt, der von oben nach unten die Aktion zeitlich darstellt. Sollen die Auswirkungen von Ereignissen dargestellt werden, so sind die Ereignisse in ihrer Zeitordnung aufzuschreiben und der Übergang zu einem anderen Funktionsblock durch einen Pfeil zu kennzeichnen. An die Funktionslinien werden die durchgeführten Aktionen geschrieben und weitere Übergänge dokumentiert. Ablauffolgediagramme beschreiben meist nicht das komplette System, sondern die Aktionen beim Auftreten eines oder mehrerer Ereignisse.

Als Beispiel soll die Entprellung einer Taste dienen. Mechanische Tasten besitzen kein ideales Verhalten in dem Sinne, das sie beim Drücken eine Spannung sofort durchschalten oder beim Loslassen sofort abschalten. Es werden vielmehr mehrere Schließ- und Öffnungsvorgänge vorgenommen, bevor ein Ruhezustand erreicht ist. Werden diese Vorgänge direkt verarbeitet, so wird nicht nur ein Schließ- oder Öffnungsvorgang erkannt, sondern gleich mehrere. Gelöst werden kann dieses Problem durch das Einführen einer Wartezeit beim ersten Erkennen einer Änderung im Schalterzustand und einer endgültigen Abfrage des Zustandes nach einer gewissen Zeit. Dazu stehen im Mikrocontroller spezielle Zeitgeber zur Verfügung, die für diesen Zweck eingesetzt werden können. Ein solcher Ablauf ist in Bild 4.3 dargestellt und demonstriert die Möglichkeiten des Ablauffolgediagramms.

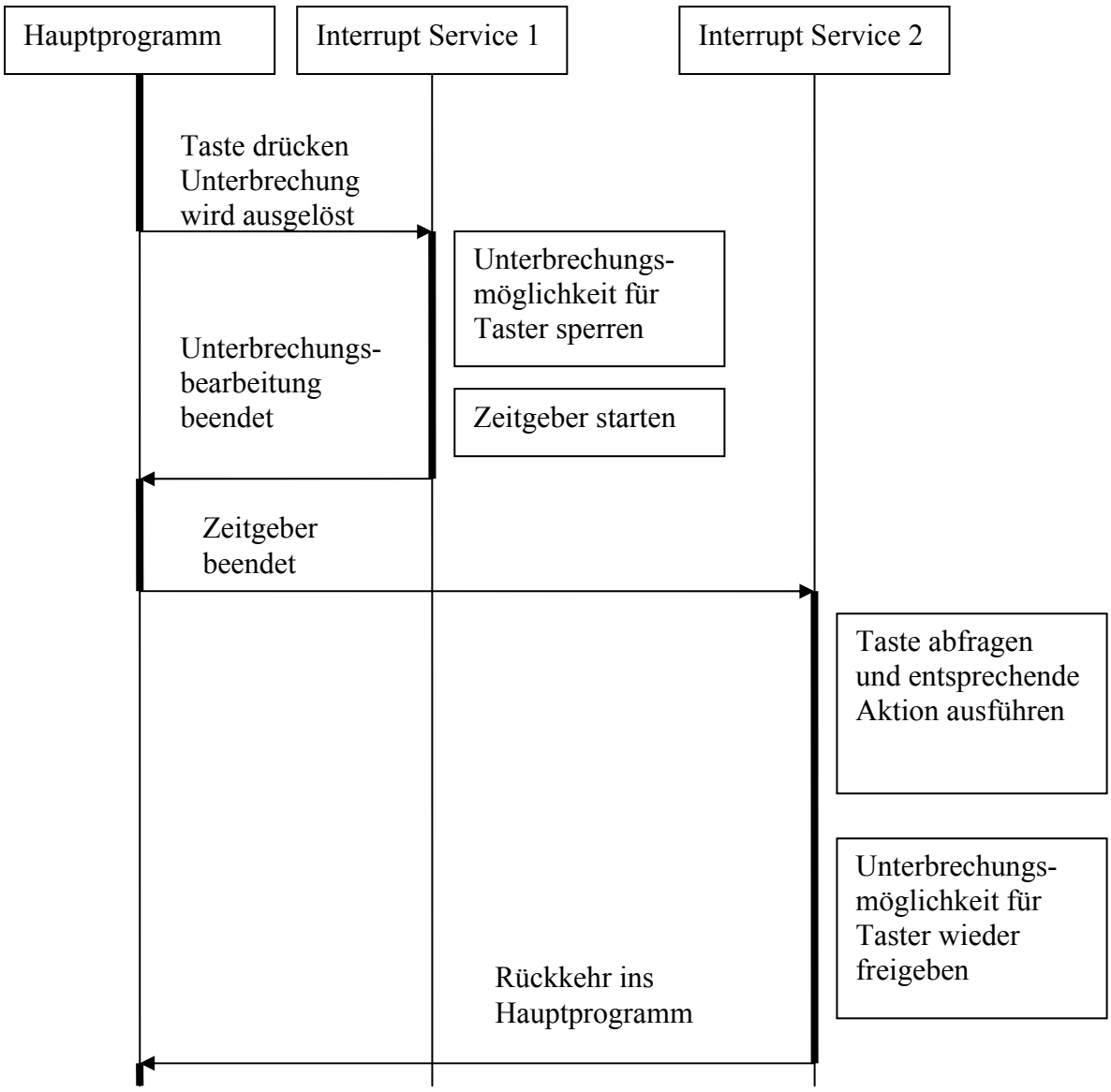


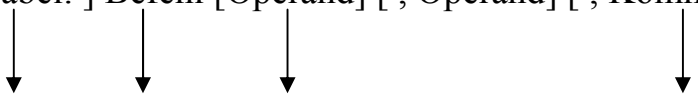
Bild 4.3 Ablauffolgediagramm beim Entprellen einer Taste

4.5 ASSEMBLER-Programmierung

Die Erstellung von Programmlösungen in einem mnemonischen Code birgt die Gefahr die strukturierte Programmierungstechnik zu verlassen. Die direkte Handhabung von Sprungmarken lässt die ursprüngliche Struktur im Assemblercode verschwinden. Es ist deshalb in jedem Fall sinnvoll zunächst den Algorithmus mit Hilfe eines Struktogramms zu beschreiben und dann umzusetzen. Ebenso sollte an Kommentierungen nicht gespart werden, da Assemblerprogramme nicht gerade intuitiv begreifbar sind. Auf den Aufbau des Codes in einem Assemblerprogramm soll im Folgenden eingegangen werden.

Aufbau einer Befehlszeile:

[Label:] Befehl [Operand] [, Operand] [; Kommentar]



Start: JMP Anfang ;Sprung zum Programmanfang

Zahlenkennzeichnungen:

Hexadezimal	H, h	z.B. 0Fh
Dezimal	D, d oder nichts	z.B. 10, 10d
Oktal	O, o, Q, q	z.B. 55Q,56o
Binär	B, b	z.B. 00110011b

Zeichenketten:

Max 2 in Hochkomma eingeschlossene ASCII-Zeichen z.B.

‘A‘ ≙ 41h

‘AB‘ ≙ 4142h

Symbole:

Max 31 Zeichen lang. 1. Zeichen A-Z, a-z oder ?
2.- n. Zeichen A-Z, a-z, 0-9, _, ?

Beispiel: Das_ist_eine_Marke
Loop1

Marken: (Einsprungpunkte)
Symbol + ‘ : ‘

Beispiel: loop1:

Spezielle Assemblersymbole:

A	(AKKU)
R0...R7	(Register)
DPTR	
PC	(Program Counter)
C	(Carry)
AB	(Registerpaar AB)
AR0...AR7	(Absolutadresse der Register R0...R7)

4.5.1 Assembleroperatoren

Assembleroperatoren dienen zur flexiblen Erstellung von konstanten Operanden. Ein Operand kann durch einen Ausdruck erzeugt werden. Der Operand muss vom Assembler berechnet werden können und liegt dann fest. Diese Form der Ermittlung von Konstanten ist hilfreich, wenn das Assemblerprogramm beispielsweise auf die Länge eines Datenfeldes parametrisiert werden soll.

Aufbau eines Ausdrucks ggf. aus:

Arithmetische Operationen:

+, -, ×, ÷, MOD, ()

Logische Operationen:

NOT A	Einerkomplement
HIGH (A)	höherwertige 8 Bits einer 16 Bit Zahl
LOW (A)	niederwertige 8 Bits einer 16 Bit Zahl
SHR, SHL	Rechts/Links Bitschiebeoperationen (Null nachziehen)
A AND B	Logische UND - Verknüpfung
A OR B	Logische Oder - Verknüpfung
A XOR B	Logische EXOR - Verknüpfung

4.5.2 Assembler Direktiven

- Zum Definieren von Symbolen
- Reservieren von Speicherbereichen
- Initialisierung von Speicherbereichen
- Festlegen der Lage des Objektes

Direktiven sind keine Prozessorbefehle. Sie produzieren keinen Objektcode.
Die folgende Aufstellung beschreibt nur den wichtigsten Teil der Assemblerdirektiven und erhebt keinen Anspruch auf Vollständigkeit.

Symboldefinitionen:

Bei den Symboldefinitionen werden Zuordnungen eines Namens zu Speicherbereichen oder Zeichenketten vorgenommen. Die Segmentdeklaration dient beispielsweise zur Erstellung von Programmbereichen oder Datenbereichen, die später verschoben werden können.

Segmentdeklaration

SEGMENT Deklaration eines verschiebbaren Segments
z.B. Code, Daten
Format: Segment_name SEGMENT Segment_Typ
Der Segmenttyp gibt an, in welchem Speicherbereich das definierte Segment liegen soll.

Typ-Code: Programmspeicherbereich
XDATA: externer Datenspeicher
DATA: interner Datenspeicher
IDATA: indirekt adressierbarer interner Datenspeicher
BIT: Bitadressierbarer Datenspeicher (intern)

Beispiel Programmsegment:

```
Programm1 SEGMENT Code
                :
          RSEG Programm1 ; Ansprechen des Segments
Start     MOV A, #0
          ....
```

EQU-Direktive

Definition eines Symbols und Zuweisung eines numerischen Ausdrucks oder eines Registersymbols.

Beispiel:

```
LIMIT   EQU  1200          ;Wertzuweisung
WERT    EQU  LIMIT – 250   ;Wertzuweisung
AKKU    EQU  A             ;Ein anderer Name für
                           ;A wird eingeführt
```

Ein mit EQU definiertes Symbol kann später nicht mehr umdefiniert werden.

Assembler Zustandsdirektiven und Segmentauswahl:

ORG Anweisung:

Mit ORG wird der Adresszähler im aktuellen Segment verändert

```
ORG 100h      ; (Anfangs)adresse 100h
```

USING Anweisung:

- Using 0...3

Angabe, welche Registerbank verwendet werden soll

- Der Assembler berechnet die Adressen für die Symbole AR0..AR7 je nach Auswahl des entsprechenden Registersatzes mit USING. (AR0....AR7 absolute Adressen von R0...R7)
- Default Registerbank ist 0
- Bei Verwendung von R0.....R7 muss RS1, RS0 selbst umgesetzt werden

End Anweisung

- Letzte Zeile des Quellprogramms, darf nur einmal pro Datei verwendet werden.

RSEG-Anweisung

- Auswahl eines vorher definierten verschiebbaren Segments (relokatibles Segment)

Absolute Segmente

Absolute Segmente sind nicht mehr verschiebbar. Der Programmierer legt die Zuordnung fest.

YSEG [AT Absolutadresse]

Y für C = Code,
D = Data,
X = XData,
I = IData,
B = Bit.

4.5.3 Programmbeispiele

Beispiel 1 Messwertberechnung [BLE94]:

Ein vorgegebener dualer Messwert X soll mit dem Faktor 2,5 multipliziert werden. Abrundung auf eine ganze Zahl ist möglich. X < 100.

100*2.5 = 250 <= 255 (größte darstellbare Zahl)

Struktogramm:

Hole den Messwert
Verdopple den Messwert → E1
Halbiere den Messwert → E2
Addiere Teilbeträge E1 + E2
Speichere das Ergebnis

Assemblerprogramm:

```
;Zuordnungen
Messw       EQU 20h
Hilfsw      EQU 21h
Erg         EQU 22h

MESSPROG    SEGMENT CODE    ;Definition eines Programm-
                            ;codesegments
RSEG    MESSPROG            ;Akt. des Programmsegments
ORG     100h                ;Einstellung der akt. Adresse
MOV     A, Messw            ; Messwert holen
RL      A                   ; Messwert verdoppeln
MOV     Hilfsw, A           ; Zwischenerg. retten
CLR     C                   ; Carry 0 setzen
RRC     A                   ; ursprünglichen Wert
RRC     A                   ; Wert halbieren
ADD     A, Hilfsw           ; Zwischenerg. addieren
MOV     Erg, A              ; Ergebnis speichern
END
```

Assemblerlisting:

```
$nolist                         ;Abschalten des Listings
$include (C8051F340.INC)        ;Vordefinierte Symbole laden
$list                           ;Listing wieder einschalten
;//Start-of-Code

Messw       EQU 20h             ;Speicherplatz Messw reservieren
Erg         EQU 21h             ;Speicherplatz f Erg. vorsehen
Hilfsw      EQU 22h             ;Hilfsspeicher anlegen

;-------------------------------------------------------------------------------------
;Initialisierungsabschnitt
        USING   1       ;Verwendung der Registerbank 1
        CSEG AT 0h      ;Startadresse beim Reset
        JMP   Start             ;Programmbeginn
;-------------------------------------------------------------------------------------
;Programmbeginn
                CSEG AT 100h        ;Programmadresse 100h
Start:          MOV  Messw,#(4+3)SHL(4)SHR(4)       ;Testbench f.d. Simulation
                MOV  A,Messw        ;Messwert holen
                RL   A              ;Messwert verdoppeln
                MOV  Hilfsw,A       ;Zwischenergebnis abspeichern
                CLR  C              ;Carry Bit 0-setzen
                RRC  A              ;ursprünglichen Wert
                RRC  A              ;Wert halbieren
                ADD  A,Hilfsw       ;Zum Zwischenergebnis addieren
                MOV  Erg,A          ;Ergebnis speichern

                ;Test zum Verwenden der Using Direktive
                MOV  R0,#1          ;Schreibt an die Adresse 0
                                    ;trotz using
                MOV  AR0,#2         ;Schreibt an die Adresse 8
                                    ;AR0 wird umgerechnet
                SETB RS0            ;Setzen des niederwertigen
                                    ;Registerselectbits
                MOV  R0,#3          ;Jetzt wird auf die Adresse 8
                                    ;geschrieben
        END
;//End-of-Code
```

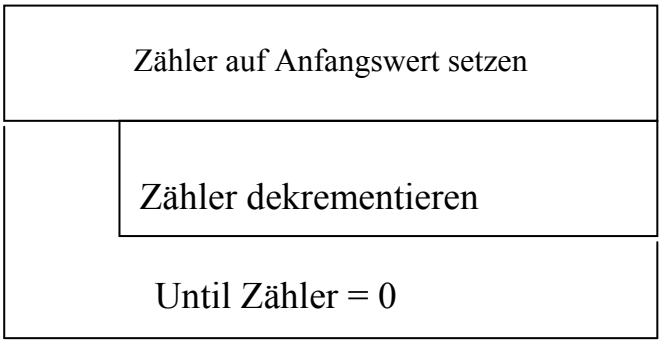
Beispiel 2 Programmieren von Zeitverzögerungen [BLE94]

Zeitverzögerungen oder Aktionen, die zu einem ganz bestimmten Zeitpunkt ausgelöst werden müssen, spielen bei Mikrocontrollern eine große Rolle. Zum Beispiel ist das Drücken einer Taste oft mit einem Wartezyklus des Prozessors verbunden, bevor der Zustand des Schalters als gültig angenommen wird. Es sei jedoch an dieser Stelle bereits darauf hingewiesen, dass Warteprozesse meist mit speziellen Funktionseinheiten(Timern) realisiert werden.

Grundidee bei der Realisierung einer einfachen Warteschleife:

Es wird ein Zähler von einem Startwert auf Null heruntergezählt. Dieser Vorgang benötigt Zeit →Erzeugung einer Zeitverzögerung.

Struktogramm:



Betrachtet man die Architektur des 8051 so kommen folgende Register oder Speicher als Zähler in Frage:

Akku
Arbeitsspeicher R0 – R7
Speicherstelle im RAM (intern)
Datenzeiger (DPTR) (nur aufwärts zählen)

Verwendung eines Registers als Zähler:

Programm:

```
        MOV R0, #ZZ          ; Anfangswert laden
Warte:  DJNZ R0, Warte       ; Zähler –1 bis 0 erreicht ist
```

Die herkömmliche 8051-Architektur benötigt für einen Zyklus 12 Takte

MOV-Befehl	1 Zyklus
DJNZ-Befehl	2 Zyklen

Warteschleife besteht nur aus DJNZ

Gesamtzeit:

$$T = \left(Ladezyklus + ZZ \times \frac{Zyklen}{Durchlauf} \right) \times Zyklendauer$$

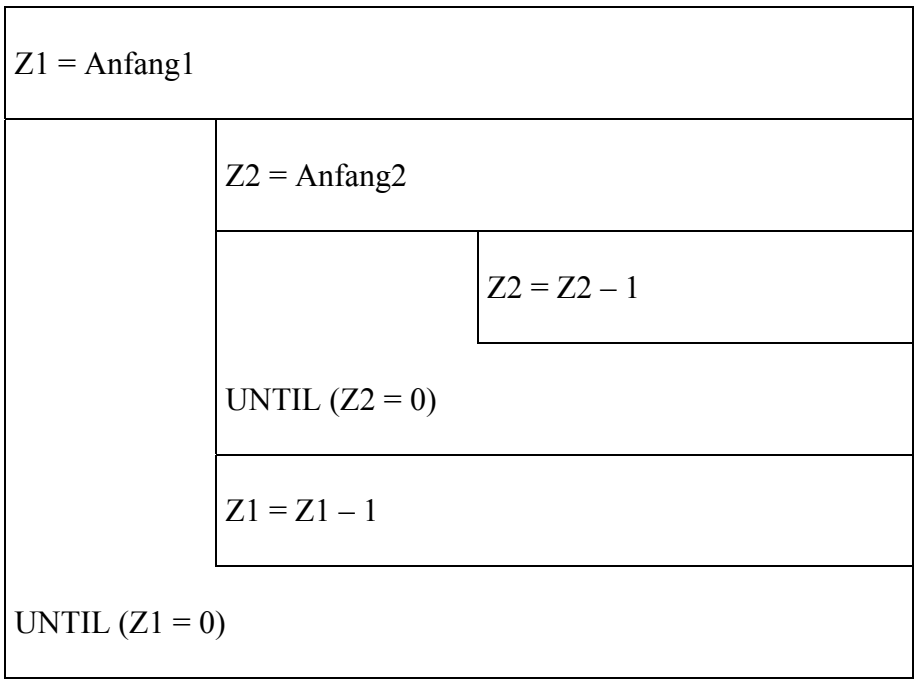
Bei 12 MHz $\rightarrow$ Zyklusdauer = 1µs

$T = (1 + 2 \times ZZ) \times 1\mu s$

Kürzeste Zeit	$\rightarrow ZZ = 01_h$	$\rightarrow T = 3\ \mu s$
Zweitlängste Zeit	$\rightarrow ZZ = FF_h$	$\rightarrow T = 511\ \mu s$
Längste Zeit	$\rightarrow ZZ = 00_h$	$\rightarrow T = 513\ \mu s$
	$T = (1+2*256) * 1\mu s = 513\ \mu s$	

Höhere Verzögerungen erreicht man durch Schachtelung von Zeitschleifen

Struktogramm:



Assemblerprogramm zur Schachtelung von Zeitschleifen

```
ANF1      EQU 200          ; Äußerer Zähler
ANF2      EQU 248          ; Innerer Zähler

          MOV R1 # ANF1    ; Z1 laden
WARTE1:   MOV R0 # ANF2    ; Z2 laden
WARTE2:   DJNZ R0, WARTE2  ; 2. Schleife, Zeitausgleich
          NOP
          DJNZ R1, WARTE1  ;1. Schleife
```

Innere Schleife 500µs (mit NOP)

$Zi = (1 + 2 \times ANF2) + 2 + 1$

↑ ↑

DJNZ NOP

$Zi = 1+2 \times 248+2 = 499$ ohne NOP

$Zi = 1+2 \times 249+2 = 501$ ohne NOP

$Zi = (1 + 2 * 248) + 2 + 1 = 1 + 496 + 2 +1 = 499 + 1 = 500$ mit NOP

$T_{ges} = (1 + ANF1 \times Zi) \times tz$

$T_{ges} = (1\mu s + 200 \times 500\mu s) = 100.001\mu s$

Der C8051F340 benötigt in der Regel pro Byte des Befehls einen Takt

```
        MOV R0, #ZZ             ; Anfangswert laden
Warte:  DJNZ R0, Warte          ; Zähler –1 bis 0 erreicht ist
```

MOV-Befehl	2 Takte
DJNZ-Befehl	2/4 Takte (Weiter/Springen)

Warteschleife besteht nur aus DJNZ

Gesamtzeit:

$$T = \left(\text{Ladetakte} + ZZ \times \frac{Takte}{Durchlauf} \right) \times Taktdauer$$

Bei 48 MHz → Taktdauer = 125/6ns =20,8333ns

$T = (2 + 4 \times ZZ) \times 125/6ns$

Zur Eliminierung des Faktors 1/6 können NOP Befehle eingeführt werden. Ein NOP- Befehl benötigt einen Takt.

```
        MOV R0, #ZZ             ; Anfangswert laden
        NOP
        NOP
        NOP
        NOP

Warte:  NOP
        NOP
        DJNZ R0, Warte          ; Zähler –1 bis 0 erreicht ist
```

$T = ((2+4) + (2+4) \times ZZ) \times 125/6ns$
$T = (1+ ZZ) \times 125ns$

Kürzeste Zeit	→ $ZZ = 01_h$	→ T = 250 ns
Zweitlängste Zeit	→ $ZZ = FF_h$	→ T = 32000 ns
	T = (1+255) * 125ns = 32000 ns	
Längste Zeit	→ $ZZ = 00_h$	→ T = 32125ns
	T = (1+256) * 125ns = 32125 ns	
Für 20µs	→ ZZ=20µs/125ns-1=159	

Beispiel 3 Initialisierung eines Feldes:

Nach dem Einschalten stehen beliebige Werte im Speicherbereich.

Je nach Problemstellung müssen Vorbelegungen initialisiert werden.

Beispiele:

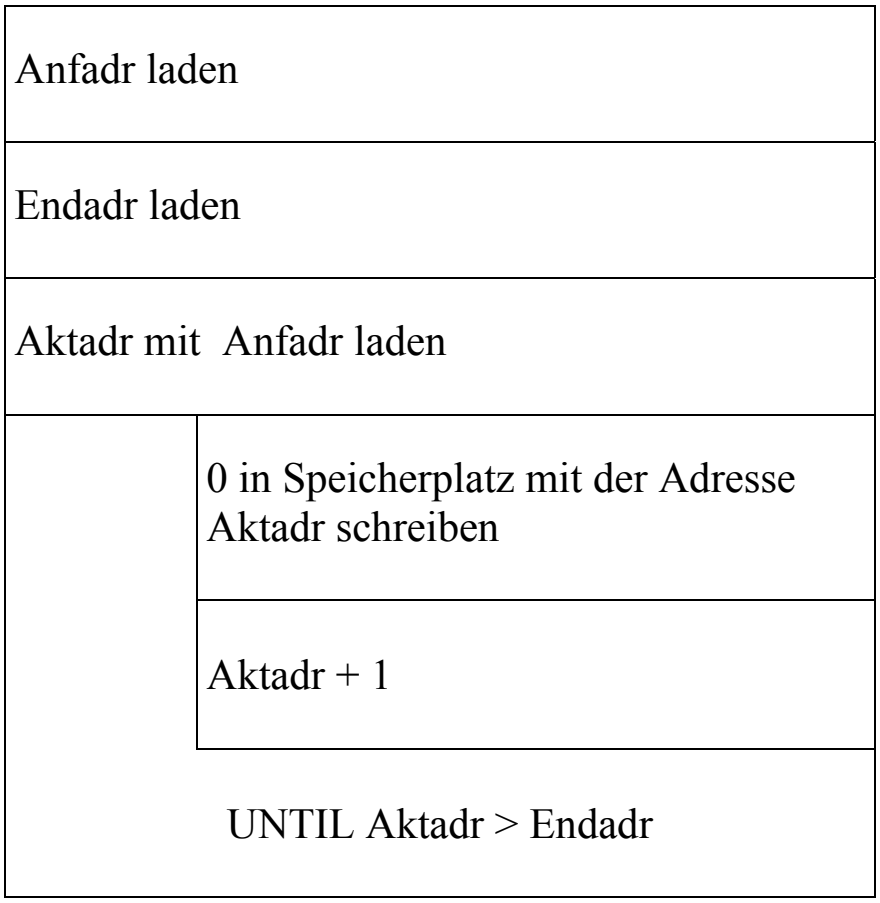
Zur Kettenaddition	Vorbelegung mit 0
Zur Kettenmultiplikation	Vorbelegung mit 1

Beispiel: Vorbelegung mit 0

Gegeben:	Anfangsadresse	Anfadr
	Endadresse	Endadr

Problem:	Es existiert kein Löschbefehl für Speicherplätze
Problemlösung :	Einspeichern einer 0

Struktogramm:



Assemblerprogramm:

```
Len       EQU 10
Anfadr    EQU 2Ah
Endadr    EQU Anfadr + Len

          MOV R0, #Anfadr      ; Anfangsadresse laden
          MOV A,  #Endadr      ; Endadresse laden
LOE1:     MOV @R0, #0          ; Speicherplatz 00
          INC R0               ; Adresse erhöhen
          CJNE A, R0, LOE1     ; Abfrage, ob Endadr. erreicht ist?
          ;Weiter bis END
          :
          :                    ;weiterer Programmtext
          END
```

Funktion des Befehls CJNE OP1,OP2 Adr:
Vergleicht die Operanden OP1 OP2 und veranlasst einen Sprung zu Adr, wenn die Operanden ungleich sind. Gleichzeitig wird das Carry Flag gesetzt.

Carry = 1 wenn OP1 < OP2 sonst = 0

Mit der zusätzlichen Abfrage JC (Jump on Carry) können dann auch die anderen Bedingungen abgefragt werden.

Zusatzfrage: Was macht das oben angegebene Programm falsch?

Verbessertes Programm:

```
          MOV R0, #Anfadr          ; Anfangsadresse laden
          MOV A,   #Endadr         ; Endadresse laden
          CJNE A,  Anfadr, TEST    ; Vergleich mit Anfadr

TEST:     JC Fehler                ; Endadr < Anfadr
          INC A                    ; Endadr wird mit verarbeitet
LOE1:     MOV @R0, #0              ; Speicherplatz <- 00
          INC R0                   ; aktuelle Adresse erhöhen
          CJNE A, R0, LOE1         ;Wunsch aber kein existierender
                                   ;Befehl.
                                   ;Realisierung->Programmlisting
          JMP Weiter
Fehler:   :
          :
Weiter:   :
          :
          END
```

```
$nomod51
$nolist
$include(C8051F340.INC)
$list

Len       EQU  10
Anfadr    EQU  2Ah
Endadr    EQU  Anfadr+Len

USING     0              ;Auswahl der Registerbank 0

CSEG at 0h               ;Startadresse festlegen
     LJMP Start          ;Einsprungpunkt

CSEG at 100h

Start:                   ;Programmstart

     MOV  R0, #Anfadr    ;Anfangsadresse laden
     MOV  A,  #Endadr    ;Endadresse laden

     CJNE A, Anfadr,Test ;Vergleich mit Anfangsadresse
Test:JC   Fehler         ;ja  :Endadresse < Anfangsadresse
     INC  A              ;nein     :Endadresse +1
                         ;sonst erlaubt die Abfrage:
                         ;CJNE A,AR0,LOE1
                         ;kein Zugriff auf das letzte Feld

LOE1:MOV  @R0,#0FFh      ;Speicherplatz auf ff setzen
     INC  R0             ;Adresse erhöhen
     CJNE A,AR0,LOE1     ;Solange akt. Adresse<>Endadresse+1
                         ;springe nach LOE1
                         ;AR0 ist die Nachbildung von R0 über
                         ;eine Direktadresse
     JMP  Weiter         ;Überspringen der Fehlerbehandlung
                         ;letzter Befehl der Initialisierungs-
                         ;sequenz

Fehler:NOP               ;Hier muss man etwas Sinnvolles tun

Weiter:NOP               ;Hier folgt das restliche Programm

     END
```

Beispiel 4: Addition zweier 12 stelliger BCD - Zahlen (Listenverarbeitung)

Zwei 12stellige BCD – Zahlen sollen addiert werden. Ablage im gepackten Format [BLE94].

⇒ 2 BCD Ziffern in einem Byte

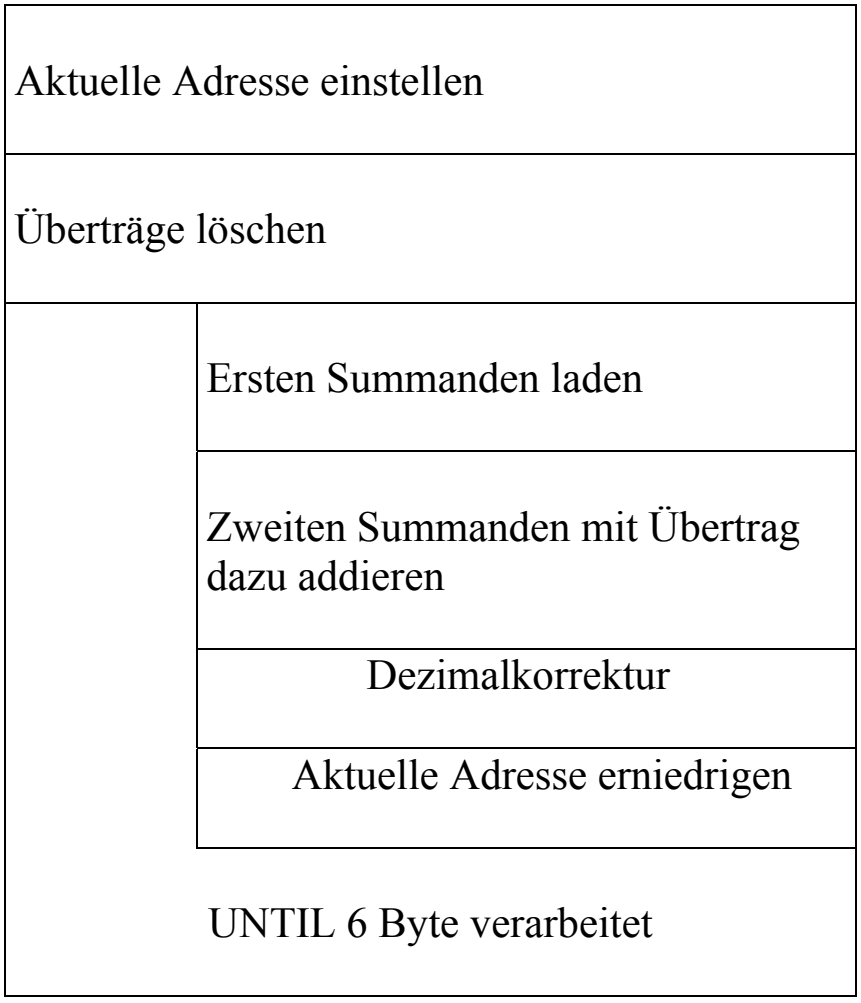
⇒ Jede Zahl umfasst 6 Bytes

⇒

Speicherorganisation:

	Z1	Z1 + 1				Z1 + 5
1. SUMMAND	MSB					LSB
	Z2	Z2 + 1				Z2 + 5
2. SUMMAND	MSB					LSB
	Z3	Z3 + 1				Z3 + 5
3. SUMME	MSB					LSB

Struktogramm:



Programm:

```
        Z1      EQU 20h
        Z2      EQU 26h
        SUM     EQU 2Ch
        REG0    EQU 00h          ; identisch mit R0, damit Befehl
                                 ; möglich MOV R0, R

        MOV R1, #Z1 + 5          ; Adr LSB 1. Summand
        MOV R2, #Z2 + 5          ; Adr LSB 2. Summand
        MOV R3, #SUM + 5         ; Adr LSB Summe
        CLR F0                   ; Übertrag Merkbit

Addlab: MOV C, F0                ; Übertrag festlegen (zurück)
        MOV A, @R1               ; Byte des 1. Summanden
        MOV REG0, R2             ; Byte des 2. Summanden
        ADDC A, @R0              ; add. des 2. Summanden + C
        DA A                     ; BCD Korrektur
        MOV F0, C                ; Carry retten
        MOV REG0, R3             ; Adresse der Summe
        MOV @R0, A               ; Ergebnisabspeicherung

        DEC R1                   ;Adresse für nächste Addition
        DEC R2                   ; Dekrementieren (-1)
        DEC R3                   ;
        CJNE R1 #(Z1-1), Addlab  ; weiter, bis MSB Adr von Z1
                                 ; unterschritten ist.
```

4.5.4 Unterprogrammtechnik

Die Definition von immer wieder verwendeten Programmstücken als Unterprogramme ist aus den höheren Programmiersprachen üblicherweise bekannt. Hier werden neben der Zusammenfassung von Befehlen noch eine Reihe anderer Aufgaben von Unterprogrammbeschreibungen übernommen u.a.:

- Definition der Parameterübergabe
- Lokale Variable können vereinbart werden

Diese Mechanismen dienen dem Programmierer als Schutz vor eventuellen Programmfehlern durch die klare Strukturierung des Programmcodes.

Unterprogrammaufrufe lösen meist eine interne Umorganisation des Speichers aus.
z.B. muss die Rücksprungadresse abgelegt werden.
Interne Variable werden bereit gestellt.
Doppelt belegte Speicherplätze werden gesichert.
Dies geschieht mehr oder weniger unbemerkt vom Benutzer.

Bei der Assemblerprogrammierung stehen hier nur rudimentäre Hilfsmittel zur Verfügung

- Der Stack zum Abspeichern von Daten obliegt der Selbstverwaltung des Programmierers
- Unterprogrammaufrufe und Rückkehrbefehle.

Daher muss an dieser Stelle auf die grundsätzliche Funktion des Stack bei Unterprogrammaufrufen und die Assemblerbefehle zum Erreichen und Verlassen eines Unterprogramms eingegangen werden. Spezielle Unterprogramme werden bei der Behandlung externer Ereignisse durch Interrupts angesprungen, sogenannte Interrupt Service Routinen (siehe spätere Kapitel).

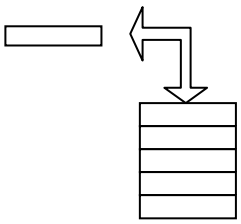
4.5.4.1 Datenverwaltung

Verwendung des Stacks zur

- Abspeicherung von Rücksprungadressen
  z.B. für Unterprogrammaufrufe oder Interrupts
- Abspeichern von Registerinhalten der CPU

Organisation als

LIFO : ***L***ast ***I***n ***F***irst ***O***ut Speicher



Operationen:

- Push Register
  Stackpointer erhöhen
  Daten abspeichern

- Pop Register
  Datum holen
  Stackpointer erniedrigen

Als Datum wird der Inhalt des Registers verwendet.

Der Stack wird indirekt adressiert. Die Adresse wird im Stackpointer (SP) abgelegt.
Startadresse beim Rechnerstart . 07h → 1 Platz → 08h → Registerbank 1

Beispiel für das Retten einer Adresse:

```
PUSH  DPH
PUSH  DPL
  :
POP  DPL   ; alten Zustand
POP  DPH   ; herstellen
```

Vertauschen von Registerinhalten

```
PUSH      ACC
PUSH      B
POP       ACC
POP       B
```

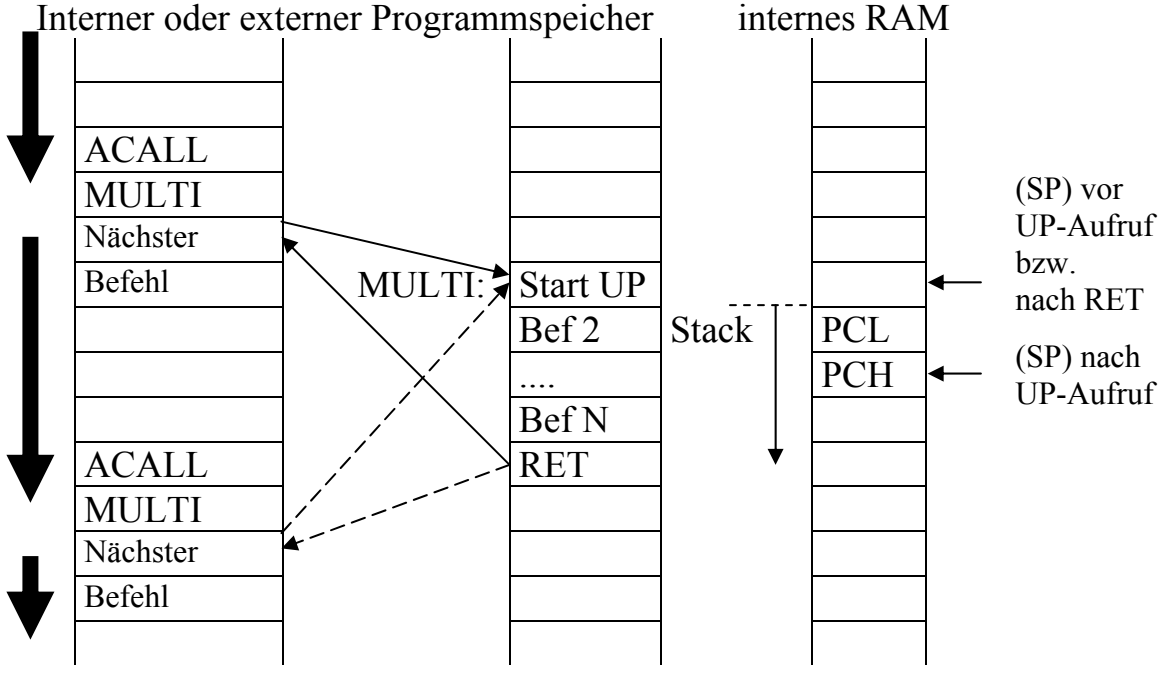
4.5.4.2 Unterprogrammablauf

Zur Erinnerung:
Ein Unterprogramm stellt eine Zusammenfassung einer Befehlsfolge dar, die an mehreren Stellen im Hauptprogramm verwendet werden kann.
Es dient als Organisationshilfe von Befehlen, zu einer höherwertigeren Operation.
Zum Aufruf einer solchen Befehlsfolge stehen folgende Befehle im Assembler zur Verfügung:

ACALL	adr11	Nur die letzten 11 Bit des PC werden verändert. 2-Byte Befehl
LCALL	adr16	voller Adressbereich 3-Byte-Befehl
CALL	adr	ist ein Assembler Befehl. Der Assembler wandelt den Aufruf um in ACALL oder LCALL

Der Unterprogrammsprung bedeutet das Retten des gerade aktuellen Befehlszählers und das Laden des Befehlszählers mit dem Operanden des CALL-Befehls. Das Retten des PC erfolgt mit Hilfe eines Stacks, 2 Bytes werden abgelegt.
Das Herstellen des alten Zustandes erfolgt beim Verlassen des Unterprogramms mit RET (RETURN). Der Wert wird vom Stack geholt.
Direkte Sprünge in andere Programmteile sind verboten. Die Verwendung weiterer PUSH- und POP-Operationen muss konsistent durchgeführt werden.
Mindestforderung: Es müssen gleich viele PUSH- wie POP-Befehle vorhanden sein.

Aufrufe von Unterprogrammen in Unterprogrammen sind erlaubt.
Die folgende Darstellung veranschaulicht den Programmablauf grafisch:



Beispiel: Addition mit Sättigung

Bei dem Entwurf von Regelsystemen wird eine Funktion benötigt, die eine Addition zweier Zahlen vornimmt und dabei einen Grenzwert nicht überschreitet. Eine solche Funktion soll als Unterprogramm zur Verfügung gestellt werden.
Bei Unterprogrammen muss neben der Funktion auch die Position der Übergabeparameter und des Rückgabewertes bestimmt werden. In diesem Beispiel sollen die Register R1 und R2 als Parameter verwendet und der Rückgabewert in R0 abgelegt werden. Der Sättigungswert ist 120 fest vorgegeben. Die Zahlen in R1 und R2 sind immer positiv anzunehmen.
Bei dieser Aufgabenstellung muss beachtet werden, dass bei einem Ergebnis > 255 der Betrag durch den Überlauf wieder kleiner als 120 sein kann.
Das Struktogramm des Unterprogramms kann folgendermaßen angegeben werden:

Unterprogramm: UP_AddSat

<table>
<tr><td colspan="3">Daten retten</td></tr>
<tr><td colspan="3">A=R1+R2</td></tr>
<tr><td>ja</td><td colspan="2">A>255</td><td>nein</td></tr>
<tr><td>R0=120</td><td colspan="2">ja A>120 nein</td></tr>
<tr><td></td><td>R0=120</td><td>R0=A</td></tr>
<tr><td colspan="3">Daten zurück</td></tr>
<tr><td colspan="3">return</td></tr>
</table>

Assemblerprogramm

```
UP_AddSat:
            Push  ACC         ;Daten retten
            Push  PSW
            Mov   A,R1        ;1. Operand laden
            Add   A,R2        ;R1+R2
            JC    SET120      ;Ergebnis >255
            CJNE  A,#120,Weit;Vergleich <=120
            JMP   SETR0_A     ;A==120
Weit:       JC    SETR0_A     ;A<120
SET120:     MOV R0,#120       ;A>120
            JMP EXIT
SETR0_A:    Mov   R0,A        ;Ergebnis zuweisen
EXIT:       Pop   PSW         ;Daten zurück
            Pop   ACC
            Ret               ;Rücksprung
Aufruf:
      Mov   R1,#10            ;1. Operand setzen
      Mov   R2,#20            ;2. Operand setzen
      Call  UP_AddSat         ;Unterprogramm aufrufen
      Mov   10,R0             ;Ergebnis abspeichern
```

4.5.5 Verarbeitung externer Ereignisse

Die Verarbeitungsreihenfolge der Befehle wird durch die Anordnung der Befehle im Programmspeicher bestimmt. Die lineare Abarbeitung der Befehle wird durch die Sprungbefehle oder Unterprogrammaufrufe unterbrochen. Soll jedoch auf Ereignisse reagiert werden deren Auftreten nicht vorgesehen werden kann, so besteht die Möglichkeit, ständig die Signale an den Ports abzufragen (Polling). Dies ist jedoch eine sehr uneffektive Methode, vor allem dann, wenn die Ereignisse sehr selten auftreten oder eine schnelle Reaktion erforderlich ist. Die meisten Prozessoren bieten hier eine Möglichkeit, das laufende Programm zu unterbrechen und ein besonderes Programmstück zu aktivieren, das dann entsprechende Aktionen ausführen kann.

Zur Bearbeitung solcher Unterbrechungen (Interrupts) sind jedoch zusätzliche Maßnahmen notwendig:

Zusätzlicher Hardwareaufwand ist notwendig:

- zur Aufnahme einer Unterprogrammaufforderung
- zur Steuerung der Annahme
- zur Unterbrechung des laufenden Programms
- zur Synchronisierung mit dem aktuellen Ablauf der Bearbeitungszyklen

Spezielle Software:

Programme zur Reaktion auf externe Ereignisse sind mit zusätzlichen Eigenschaften zu erstellen (Interrupt-Service-Routinen):

1. Gewährleistung des ursprünglichen Zustandes des Prozessors bei der Rückkehr
2. Zeitlich angepasstes Verhalten (kurze Reaktionszeit)

Das Zusammenspiel zwischen den speziellen Hardwarekomponenten und der benötigten Software soll im Folgenden untersucht werden. Dazu wird zunächst der Ablauf der Aktionen beim Auftreten eines Interrupts dargestellt.

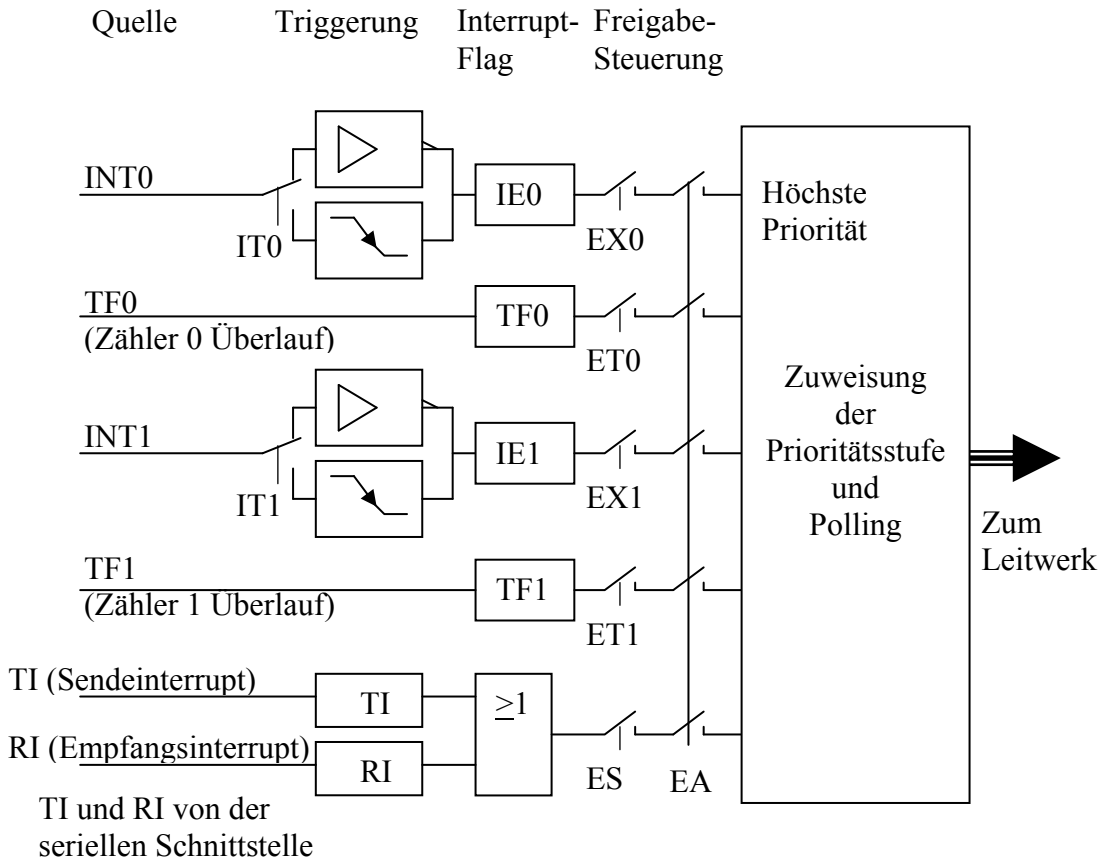
Abfolge der Aktionen bei der Interruptverarbeitung

1. Annahme der Unterbrechungsanforderung
2. Vollständige Bearbeitung der laufenden Instruktionen
3. Sprung zur Interrupt-Service-Routine
4. Retten des aktuellen Zustandes auf dem Stack

   Minimallösung : PC und Zustandsregister
5. Ausführung der Aktion zur Bearbeitung des externen Ereignisses
6. Wiederherstellen des alten Zustandes
7. Ausführung des Rückkehrbefehles ***RETI***
8. Fortfahren im unterbrochenen Programm

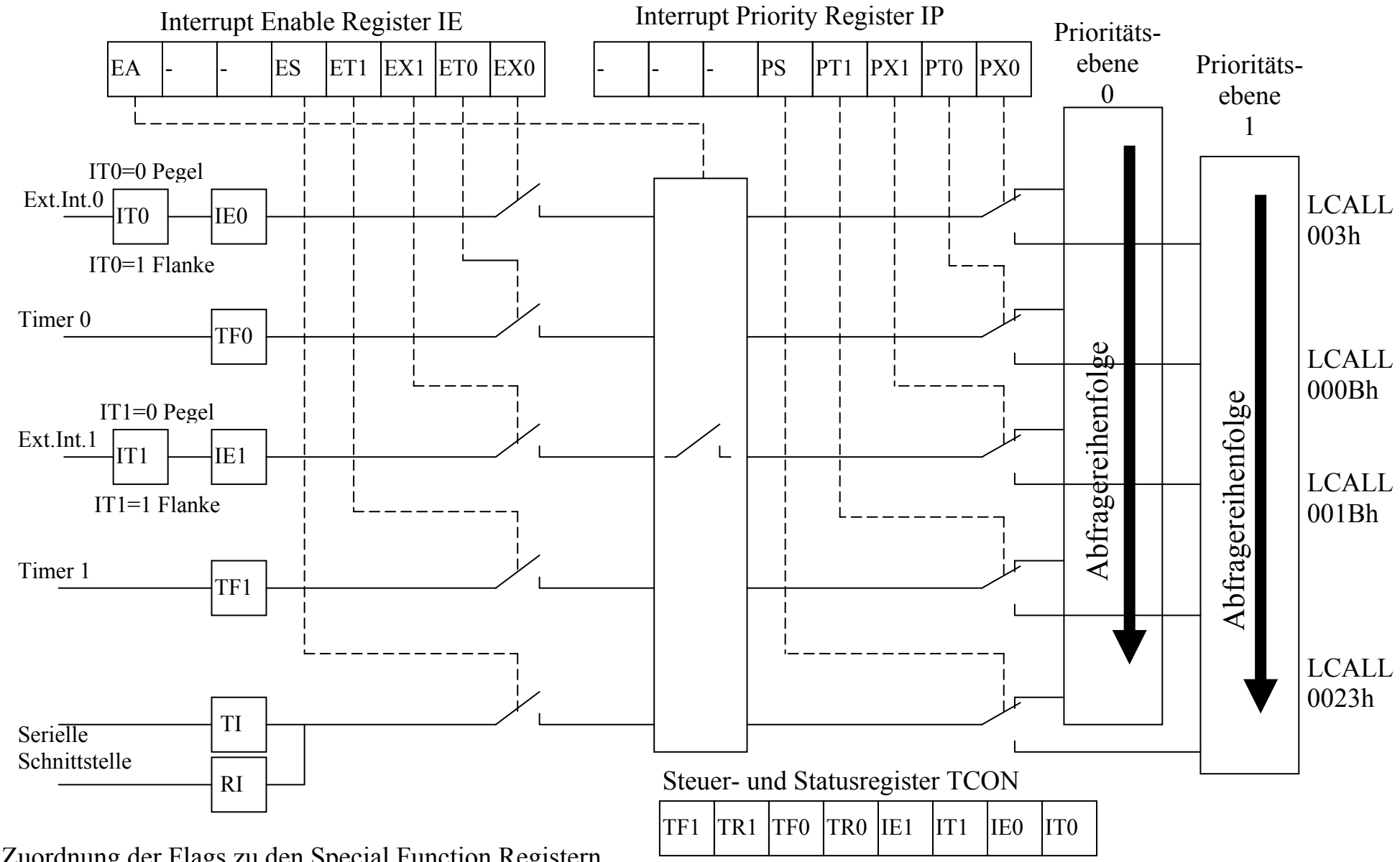
Zu 1. Annahme der Unterbrechungsaufforderung

Zur Bearbeitung externer Ereignisse stehen spezielle Einheiten zur Signalerkennung und zur Zwischenspeicherung zur Verfügung. Dies sind beispielsweise die Flags IE0 oder IE1 zur Kennzeichnung des Auftretens externer Interrupts. Weiter kann die zu erkennende Signalart bei bestimmten Anschlüssen über besondere Steuerbits ausgewählt werden. Externe Interrupts können z.B. flankengesteuert oder pegelgesteuert aktiviert werden ($\overline{INT0}$ $\overline{INT1}$).
Ebenso stehen Auswahlmechanismen zu Verfügung, die es erlauben, auftretende Ereignisse zu filtern d.h. sie können zugelassen oder gesperrt werden oder in ihrer Wichtigkeit bewertet werden.
Die Steuerbits und die Anzeigeflags sind in den Special Function Registern im internen RAM untergebracht und können dementsprechend gelesen oder geschrieben werden (zum Teil auch bitweise). So können die Flags IE0 und IE1 zu Testzwecken auch per Software gesetzt werden.
IE0 und IE1 werden automatisch gelöscht, sobald die Interruptserviceroutine angesprochen wird.



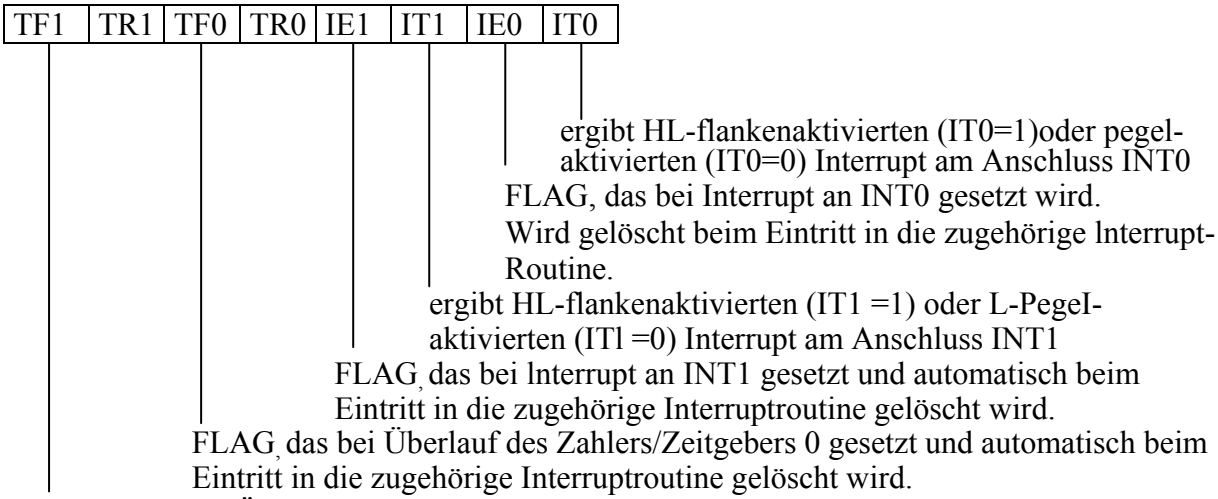
Basisstruktur der Interruptsteuerung für den 8051

Das automatische Löschen der Interruptanforderung ist nicht in jedem Fall gegeben, so müssen beispielsweise die Flags der seriellen Schnittstelle per Software gelöscht werden.
Die Freigabe der Interrupts kann sowohl selektiv als auch im Gesamten vorgenommen werden. Nach dem Einschalten oder dem Reset sind zunächst alle Interrupts gesperrt. Dies ist sinnvoll, um den Prozessor zunächst in einen Zustand zu bringen, in dem er sinnvoll auf äußere Ereignisse reagieren kann.
Die Einzelfreigabe ist auch dann sinnvoll, wenn für einen bestimmten Zeitraum der Interrupt gesperrt sein soll z.B. beim Prellen einer Taste.

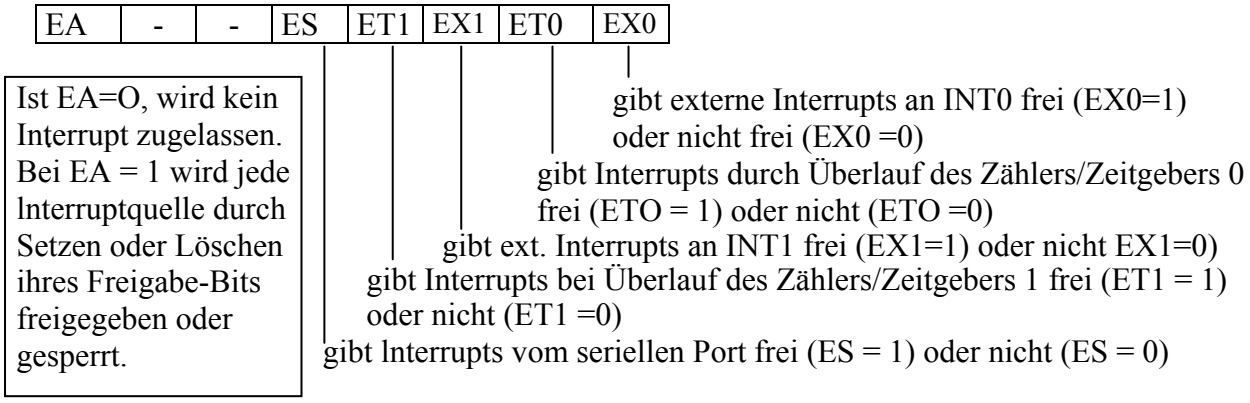


Zuordnung der Flags zu den Special Function Registern

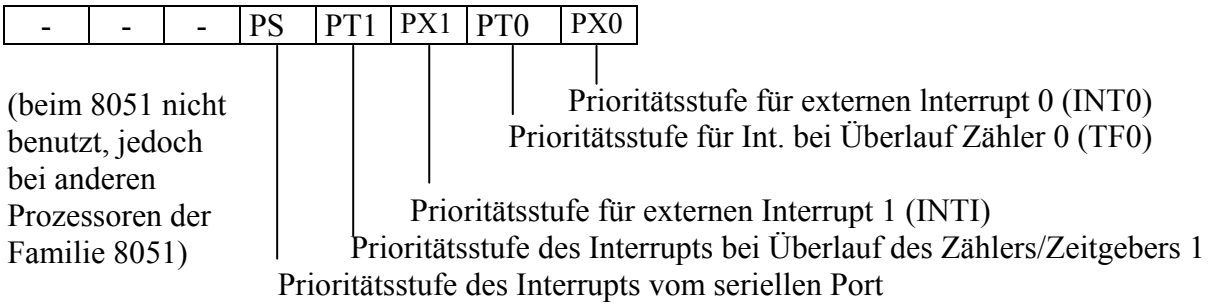
Zeitgeber/Zähler-Steuer und Statusregister (TCON)



Interrupt-Freigaberegister (IE)



Interrupt-Prioritätsregister (IP)



Die Special Function Register (SFR) der Interruptsteuerung

Zu 2. Vollständiges Bearbeiten des laufenden Befehls

Die Interrupt-Flags werden erst kurz vor Ende eines Befehls abgefragt. Praktisch heißt das, dass ein in Ausführung befindlicher Programmbefehl nicht abgebrochen werden kann. Es vergeht noch ein weiterer in dem eine Prioritätenzuordnung stattfindet. Damit verbunden ist die Wahl der Startadresse für die Interruptroutine.

Ein Interrupt wird abgeblockt wenn:

- Ein Interrupt der gleichen oder höherer Priorität verwendet wird.
- Die Ausführung des Befehles noch nicht beendet ist.
- Die aktuelle Instruktion RETI ist, oder ein Zugriff auf die Register IEN0, IEN1, IEN2, IR0, IR1 erfolgt. Mindestens eine Instruktion wird dann noch ausgeführt.

Zu 3. Sprung zur Interrupt Service Routine

Hardwaremäßig wird ein Sprung an eine zum Interrupt gehörigen Adresse vorgenommen. Dies entspricht einen LCALL + Ablegen der Rücksprungadresse auf den Stack. Es werden keine weiteren Register gerettet.

Adressen, die im Programmspeicher bei den verschiedenen Interrupts angesprungen werden:

FFFFh	Programmspeicher	
.....	.....	
002Bh	Timer 2 Interrupt	
0023h	Serielle Schnittstelle	
001Bh	Timer 1	
0013h	Externer Interrupt 1	
000Bh	Timer 0	
0003h	Externer Interrupt 0	8 Byte
0000h	Reset	

Zu 4, 5 und 6

Der Interrupt kann an jeder beliebigen Stelle erfolgen.
Die Interrupt Service Routine muss daher so gestaltet sein, dass alle von ihr verwendeten Register gesichert werden. (PUSH Operationen)
(PSW NICHT VERGESSEN)

Nach der Sicherung kann die eigentliche Verarbeitung erfolgen.

Anschließend muss der ursprüngliche Status der Register wiederhergestellt werden (POP Operationen).

Zu 7

Rücksprung mit RETI

- Die Rücksprungadresse wird vom Stack geholt und in den PC geladen.
- Die Interrupteinheit erkennt die Abarbeitung des Interrupts (nicht bei RET)

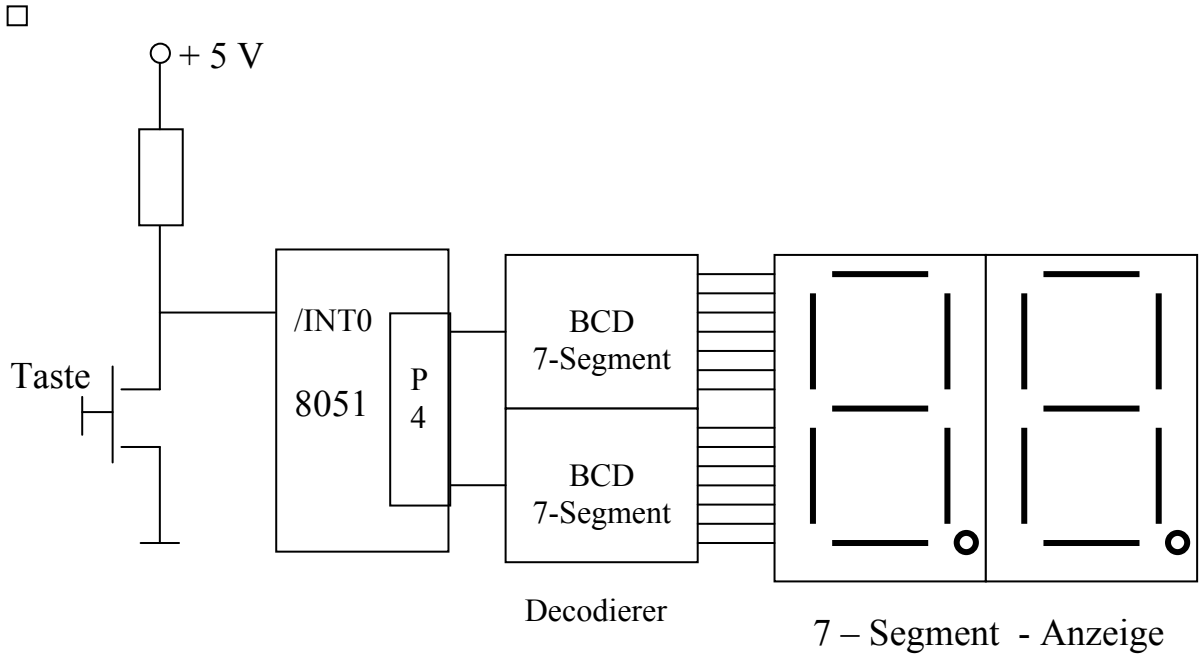
Interrupttabelle des C8051F340 [DB1]:

Interrupt Source	Interrupt Vector	Priority Order	Pending Flag		Bitaddr.?	Cleared by HW?	Enable Flag		Priority Control	
Reset	0x0000	Top	None		N/A	N/A	Always Enabled		Always Highest	
External Interrupt 0 (/INT0)	0x0003	0	IE0	(TCON.1)	Y	Y	EX0	(IE.0)	PX0	(IP.0)
Timer 0 Overflow	0x000B	1	TF0	(TCON.5)	Y	Y	ET0	(IE.1)	PT0	(IP.1)
External Interrupt 1 (/INT1)	0x0013	2	IE1	(TCON.3)	Y	Y	EX1	(IE.2)	PX1	(IP.2)
Timer 1 Overflow	0x001B	3	TF1	(TCON.7)	Y	Y	ET1	(IE.3)	PT1	(IP.3)
UART0	0x0023	4	RI0 TI0	(SCON0.0) (SCON0.1)	Y	N	ES0	(IE.4)	PS0	(IP.4)
Timer 2 Overflow	0x002B	5	TF2H TF2L	(TMR2CN.7) (TMR2CN.6)	Y	N	ET2	(IE.5)	PT2	(IP.5)
SPI0	0x0033	6	SPIF MODF WCOL RXOVRN	(SPI0CN.7) (SPI0CN.5) (SPI0CN.6) (SPI0CN.4)	Y	N	ESPI0	(IE.6)	PSPI0	(IP.6)
SMB0	0x003B	7	SI	(SMB0CN.0)	Y	N	ESMB0	(EIE1.0)	PSMB0	(EIP1.0)
USB0	0x0043	8	Special		N	N	EUSB0	(EIE1.1)	PUSB0	(EIP1.1)
ADC0 Window Compare	0x004B	9	AD0WINT	(ADC0CN.3)	Y	N	EWADC0	(EIE1.2)	PWADC0	(EIP1.2)
ADC0 Conversion Complete	0x0053	10	AD0INT	(ADC0CN.5)	Y	N	EADC0	(EIE1.3)	PADC0	(EIP1.3)
Programmable Counter Array	0x005B	11	CF CCFn	(PCA0CN.7) (PCA0CN.n)	Y	N	EPCA0	(EIE1.4)	PPCA0	(EIP1.4)
Comparator0	0x0063	12	CP0FIF CP0RIF	(CPT0CN.4) (CPT0CN.5)	N	N	ECP0	(EIE1.5)	PCP0	(EIP1.5)
Comparator1	0x006B	13	CP1FIF CP1RIF	(CPT1CN.4) (CPT1CN.5)	N	N	ECP1	(EIE1.6)	PCP1	(EIP1.6)
Timer 3 Overflow	0x0073	14	TF3H TF3L	(TMR3CN.7) (TMR3CN.6)	N	N	ET3	(EIE1.7)	PT3	(EIP1.7)
VBUS Level	0x007B	15	N/A		N/A	N/A	EVBUS	(EIE2.0)	PVBUS	(EIP2.0)
UART1	0x0083	16	RI1 TI1	(SCON1.0) (SCON1.1)	N	N	ES1	(EIE2.1)	)PS1	(EIP2.1)

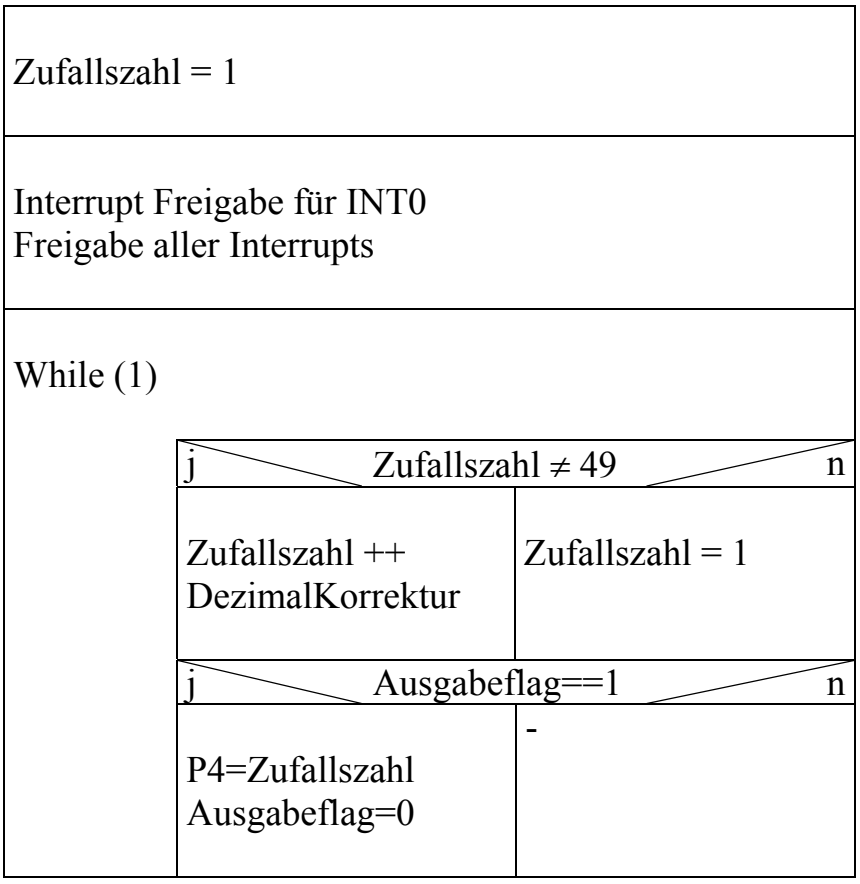
Beispiel :

Es soll ein Lottozahlengenerator entwickelt werden.
Das System soll nacheinander alle Zahlen von 1 bis 49 in einer schnellen Reihenfolge erzeugen. Durch den Druck auf eine Taste wird der Zählvorgang unterbrochen und die gerade aktuelle Zahl wird solange ausgegeben, bis die Taste erneut gedrückt wird. Die Taste soll kein Prellverhalten aufweisen. Für die Ausgabe der Zahl stehen zwei Siebensegmentanzeigen und die dazu gehörigen BCD-Siebensegmentdecoder zur Verfügung. Die Zehnerstelle wird an den höherwertigen Bits und die Einerstelle an den niederwertigen Bits des Ports 4 ausgeben:

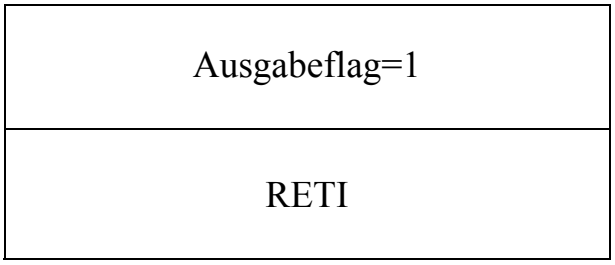
Systemaufbau:



Struktur des Hauptprogramms:



Interruptserviceroutine:



```
CSEG AT   0
LJMP Haupt2

; Interruptroutine
CSEG AT 3h

INT0:     MOV R0, #1          ; Ausgabeflag setzen
          RETI                ; ENDE des Interrupts



; Hauptprogramm

CSEG at 70h

Haupt2:   MOV A, # 01         ; Anfangswert der Zufallszahl
          MOV P4,A            ; erstmalige Ausg. einer Zahl
          SETB EX0            ; Interrupt für INT0 freigeben
          SETB EA             ; globale Interruptfreigabe

Schleife:
          CJNE A, #49h, Plus  ; Maximal 49 ist erlaubt
          CLR A               ; neu initialisieren
PLUS:     ADD A,#1            ; addieren wegen der Flags
          DA A                ; dezimal korrigieren

          CJNE R0,#1, Ausg    ;Ausgabeflag überprüfen
          MOV P4, A           ; Zahlausgabe
          MOV R0,#0           ; Ausgabeflag zurücksetzen

Ausg:     JMP Schleife        ; Endlosschleife
```

4.5.6 Bearbeitung mehrerer Interruptquellen

Verwendung einer Interruptleitung für mehrere Interrupt – Quellen

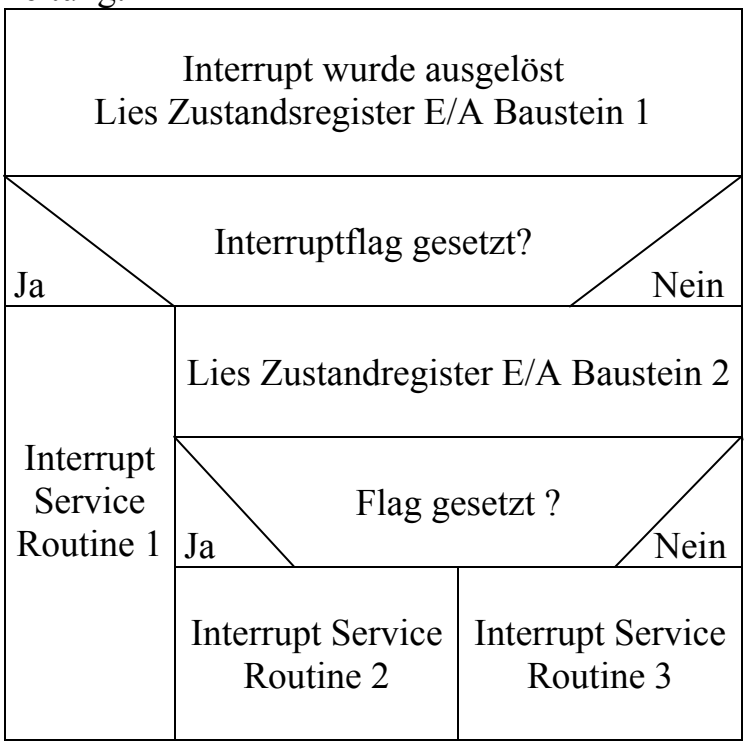
Annahme: Der L – Pegel löst den Interrupt aus

Peripheriebausteine sind an den Interruptausgängen mit offenen Kollektoren (Drains) ausgestattet. Alle Interruptleitungen werden miteinander verbunden und an eine Unterbrechungsleitung des Prozessors geschaltet. Zusätzlich wird ein Pull-Up-Widerstand benötigt.

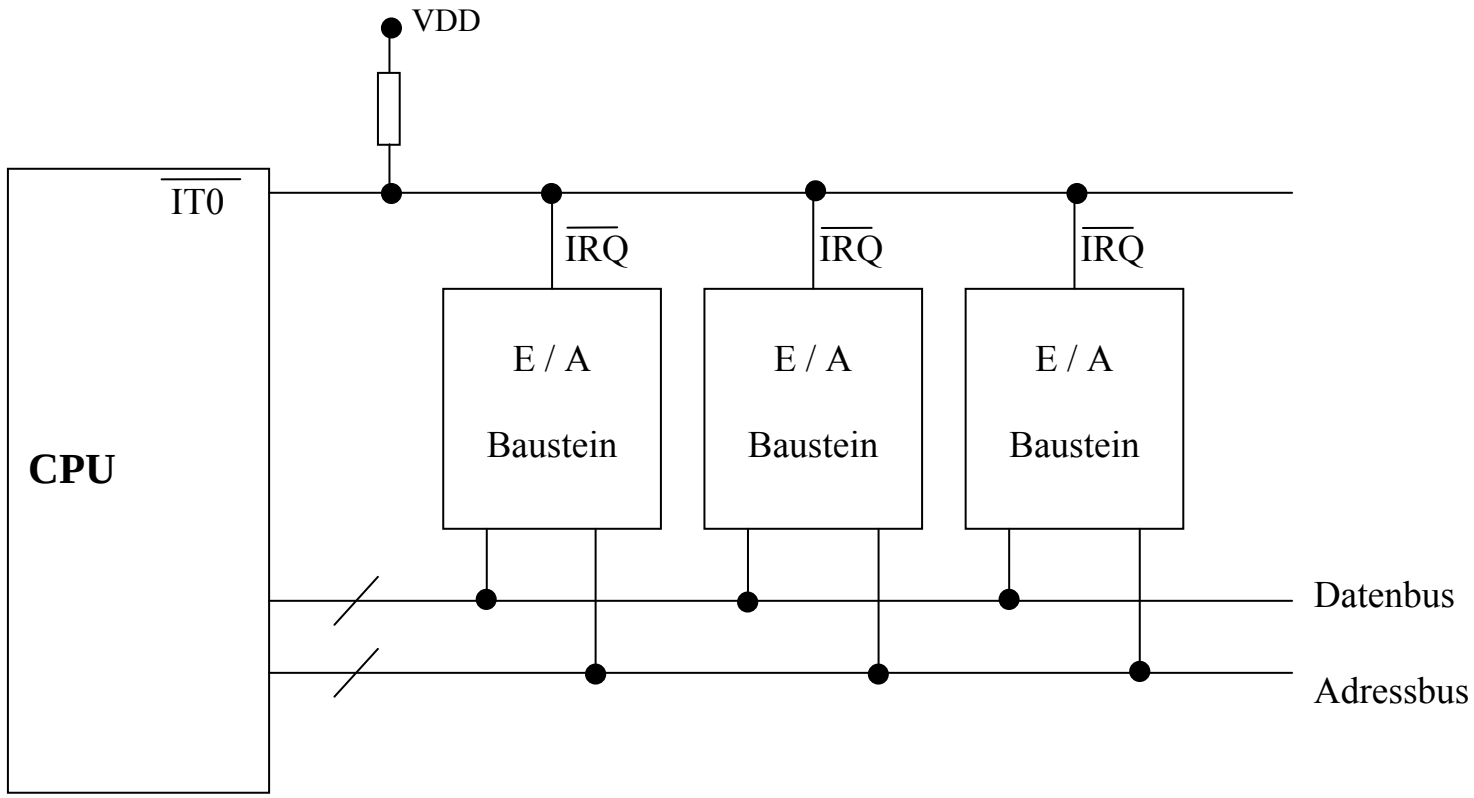
Wirkungsweise:

- Ruhezustand z.B. $\overline{INT0} = 1$
- Bei einer oder mehrerer Interrupts $\overline{INT0} = 0$
- Welche Quelle den Interrupt ausgelöst hat, muss durch Polling herausgefunden werden.
  → Ein Zustandsregister muss im E/A - Baustein vorhanden sein
- Die Interrupt - Quelle muss veranlasst werden können, den Interrupt zurückzunehmen.

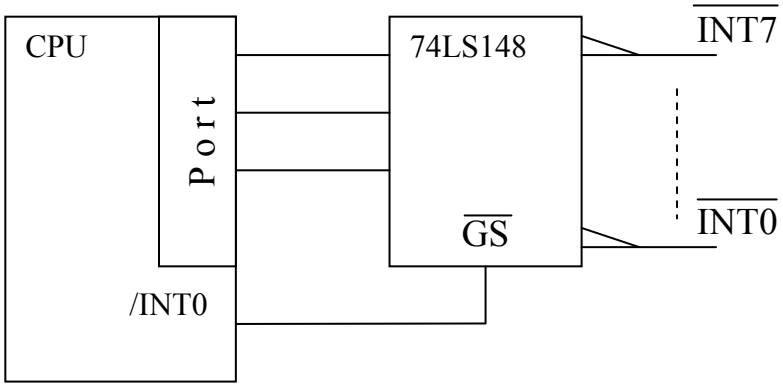
Struktur eines Programms zum Bearbeiten von Interrupts verschiedener Bausteine auf einer Leitung:



Interrupterkennung über Polling



Eine weitere Möglichkeit die Anzahl der Interruptleitung zu erhöhen, ist die Verwendung eines Prioritätsdekoders:



Der Baustein 74148 erlaubt die Überwachung von 8 Interrupts auf ihre aktiven Pegel. Als Ergebnis der Auswertung wird die duale Kennung des höchstrangigsten Interrupts (I7 höchste-, I0 niedrigste Priorität) ausgegeben. Zusätzlich wird an GS ein Wechsel von 1 auf 0 erzeugt. Wird dieser Ausgang z.B. am $\overline{\text{INT0}}$ Anschluss des Prozessors angeschlossen, so löst dieser Vorgang einen Interrupt aus. Die Interruptservice Routine muss dann die Information, welcher Interrupt aktiv ist, am Port lesen und über einen Sprungverteiler zur richtigen Verarbeitungsroutine springen.

4.5.7 Zeitgeber

Bei dem Entwurf von Mikrocontrollersystemen besteht häufig die Anforderung an bestimmten Zeitpunkten Aktionen ausführen zu können z.B.:

- Bei der Messung von Zeiten
- Bei der Bestimmung von Zeiträumen
- Und beim Schalten zu definierten Zeitpunkten.

Die in Frage kommenden Anwendungsbereiche sind sehr vielfältig und reichen von der Fabriksteuerung zu einfachen Messaufgaben

Anwendungsbereiche:

- Prozessteuertechnik
- Haushaltsgräte
- Automobilbereich
- Elektrische Messtechnik

Prozessoren besitzen in der Regel einen hochgenauen Quarz oder einen stabilisierten Oszillator zur Taktung des Gesamtsystems. Verwendet man diesen Zeitabschnitt, der durch die Taktdauer bestimmt wird, so kann man durch Herunterteilen des Taktes genaue Zeitabschnitte definieren, die ein Vielfaches des Systemtaktes sind. Prinzipiell kann eine solche Teilung durch das Schreiben angepasster Programme realisiert werden. Der Prozessor ist dann jedoch nicht mehr in der Lage, andere Aufgaben auszuführen.

Abhilfe:

Zusatzhardware in Form von programmierbaren digitalen Zählern

Zähler können in vielen Anwendungen direkt verwendet werden um Zeit- oder Frequenzaussagen zu machen z.B.:

1. Ereignisse zählen
2. Bei fest gegebenem Takt → Messen von Zeiten
   - Drehzahlmessung
   - Frequenzmessung
   - Uhren
3. Erzeugung von Zeitsignalen
   - Frequenzgenerator
   - Pulsweitenmodulation (PWM)
   - Erzeugung von Pulsfolgen

4.5.7.1 Funktion der Timer

Die Wirkungsweise eines programmierbaren digitalen Zählers soll direkt an den Funktionsbausteinen des Prozessors C8051F340 verdeutlicht werden. Die programmierbaren Zähler heißen hier allgemein Timer. Wird ein solcher Timer vom Systemtakt gesteuert wird er als Zeitgeber betrieben. Er kann jedoch auch von einem externen Ereignis gesteuert werden, um Folgen von Impulsen zu zählen (Zählerbetrieb)

Der Prozessor C8051F340 besitzt:

- 4 Universalzähler
- 1 Vergleichszähler (Programmable Counter Array (PCA))

Im Vergleich dazu besitzt der ursprüngliche Prozessor 8051 nur 2 universelle Zähler.

Der Vergleichszähler (PCA) im Prozessor C8051F340 dient zum simultanen Vergleich mehrerer vorgegebener Bitmuster mit dem Inhalt des Vergleichszählers. Die Übereinstimmung eines Bitmusters wird angezeigt und kann dann zum Anlass für weitere Aktionen dienen. Auf die genaue Funktion des Vergleichszählers soll an dieser Stelle nicht eingegangen werden. Es wird auf das Datenblatt verwiesen.

Funktion der Timer im C8051F340 [DB1]

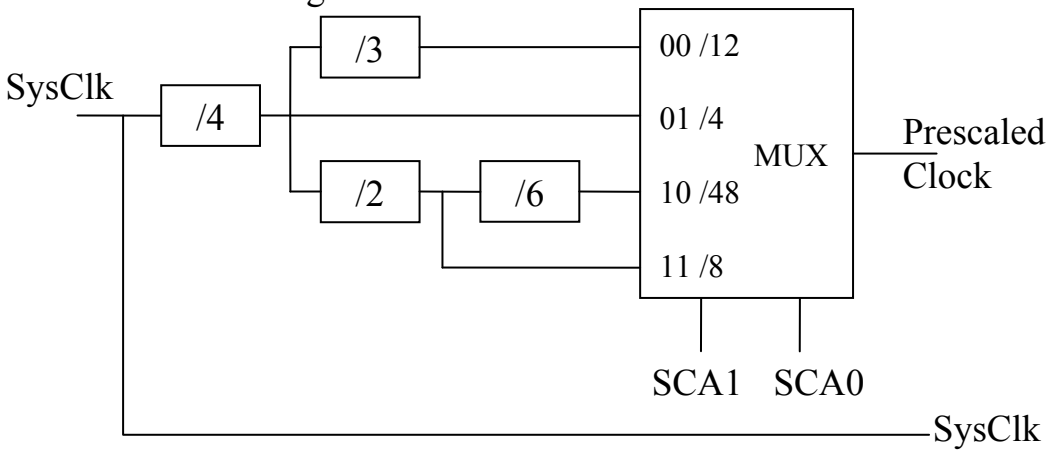
<table>
<tr><th>Timer 0 und Timer 1
Modi:</th><th>Timer 2 und Timer 3
Modi</th></tr>
<tr><td>13-bit Zähler/Zeitgeber</td><td rowspan="2">16-bit Zähler/Zeitgeber
mit auto-reload</td></tr>
<tr><td>16-bit Zähler/Zeitgeber</td></tr>
<tr><td>8-bit Zähler/Zeitgeber
mit auto-reload</td><td rowspan="2">Zwei 8-Bit Zeitgeber
mit auto-reload</td></tr>
<tr><td>Zwei 8-bit Zähler/Zeitgeber
(Nur Timer 0)</td></tr>
</table>

Im Folgenden werden nur Timer 0 bzw. Timer 1 behandelt. Für Timer 2 und Timer 3 soll wiederum auf das Datenblatt verwiesen werden.

Die Timer 0 und Timer 1 können sowohl als Zeitgeber als auch als Zähler eingesetzt werden:

Timerfunktion (Zeitgeber):

Der Systemtakt wird direkt oder durch entsprechende Taktuntersetzer verwendet um den Inhalt des Timerregisters um 1 zu erhöhen.



SCA1	SCA0	Prescaled Clock
0	0	SysClk/12
0	1	SysClk/4
1	0	SysClk/48
1	1	SysClk/8

CKCON: SFR-Adresse: 0x8E **Byte-adressierbar**

R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	Reset
T3MH	T3ML	T2MH	T2ML	T1M	T0M	SCA1	SCA0	00000000
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	

Für CKCON soll im Folgenden der Wert 0x0E (0000 1110) angenommen werden, wenn keine andere Angabe gemacht wird. Damit werden die Timer bei einem Systemtakt von 48 MHz mit 1 MHz betrieben. Damit dauert ein Inkrementvorgang 1 µs.

Zählerfunktion:

Ein HL Übergang am zugehörigen Eingang z.B. T0 oder T1 (P0.0 oder P0.1 für die in der Vorlesung/Labor verwendete Konfiguration) bewirkt eine Erhöhung des Zählerstandes um 1.

Die Feststellung des HL-Übergangs erfolgt in 2 Zyklen:
1. Zyklus H
2. Zyklus L

Die maximale Zählrate ist 1/4 des Systemtaktes.

Möglichkeiten der Steuerung der TIMER T0 und T1

Zur flexiblen Steuerung der Zeitgeber- und Zählerfunktionen können Timer 0 und Timer 1 in mehreren Modi betrieben werden.

Mode 0 13-bit Timer/Zähler

Mode 1 16-bit Timer/Zähler

Mode 2 8-bit Timer/Zähler mit automatischem Nachladen (8bit)

Mode 3

Timer/Zähler 0
- 8-bit Timer mit Highbyte
- 8-bit Timer mit Lowbyte

Timer 1
- Timer zur Steuerung der z.B. seriellen Schnittstellen
- Ohne Interruptmöglichkeit

Einstellung der Betriebsarten in den Registern TMOD und TCON

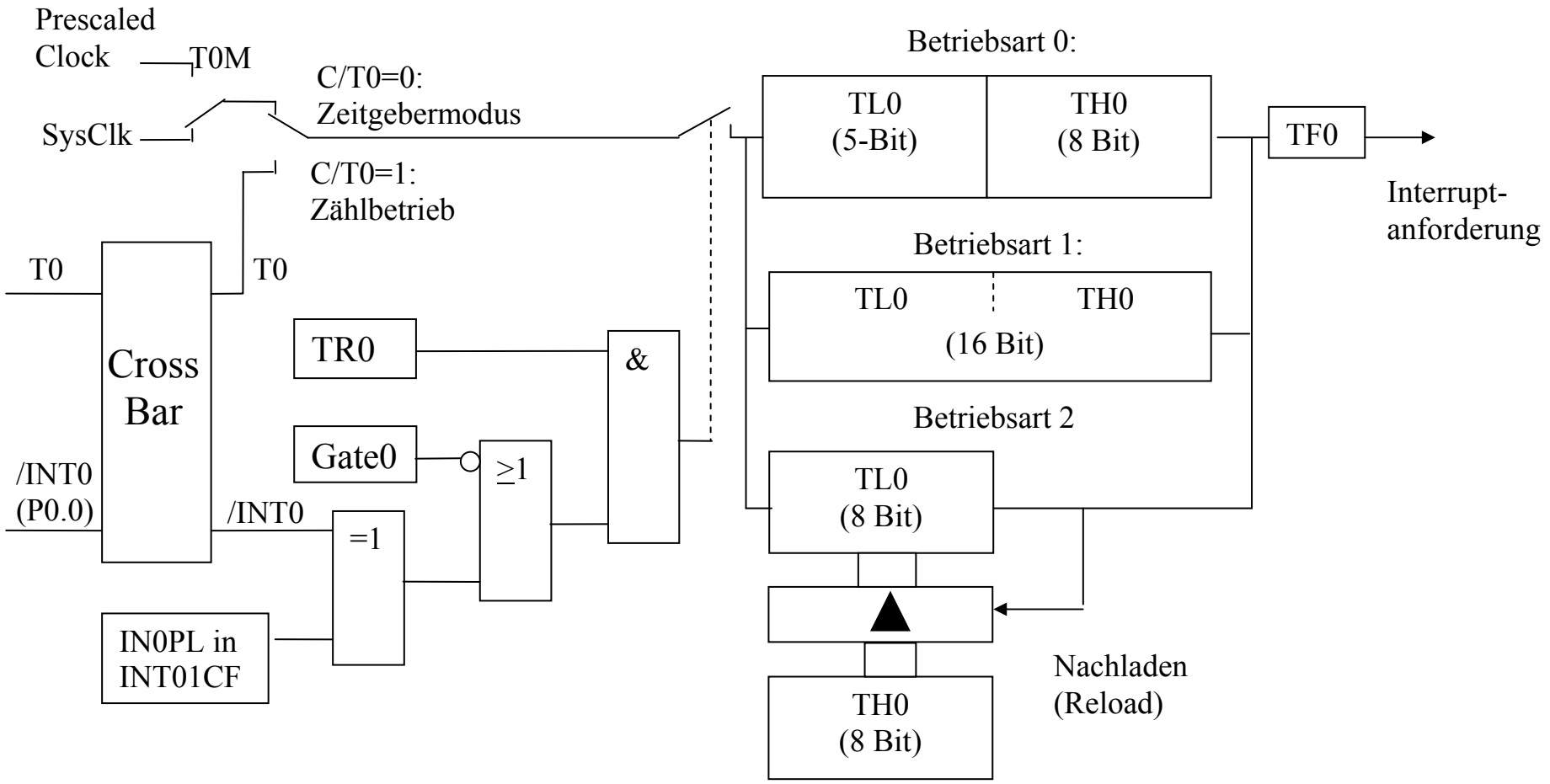
TMOD Timer Mode = Betriebsart
SFR Adresse 0x89, (byteadressierbar)

	Timer 1			Timer 0				
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	Reset
GATE1	C/T1	T1M1	T1M0	GATE0	C/T0	T0M1	T0M0	00000000
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	

TCON Timer Control = Zählersteuerung
SFR Adresse x088, (bitadressierbar)

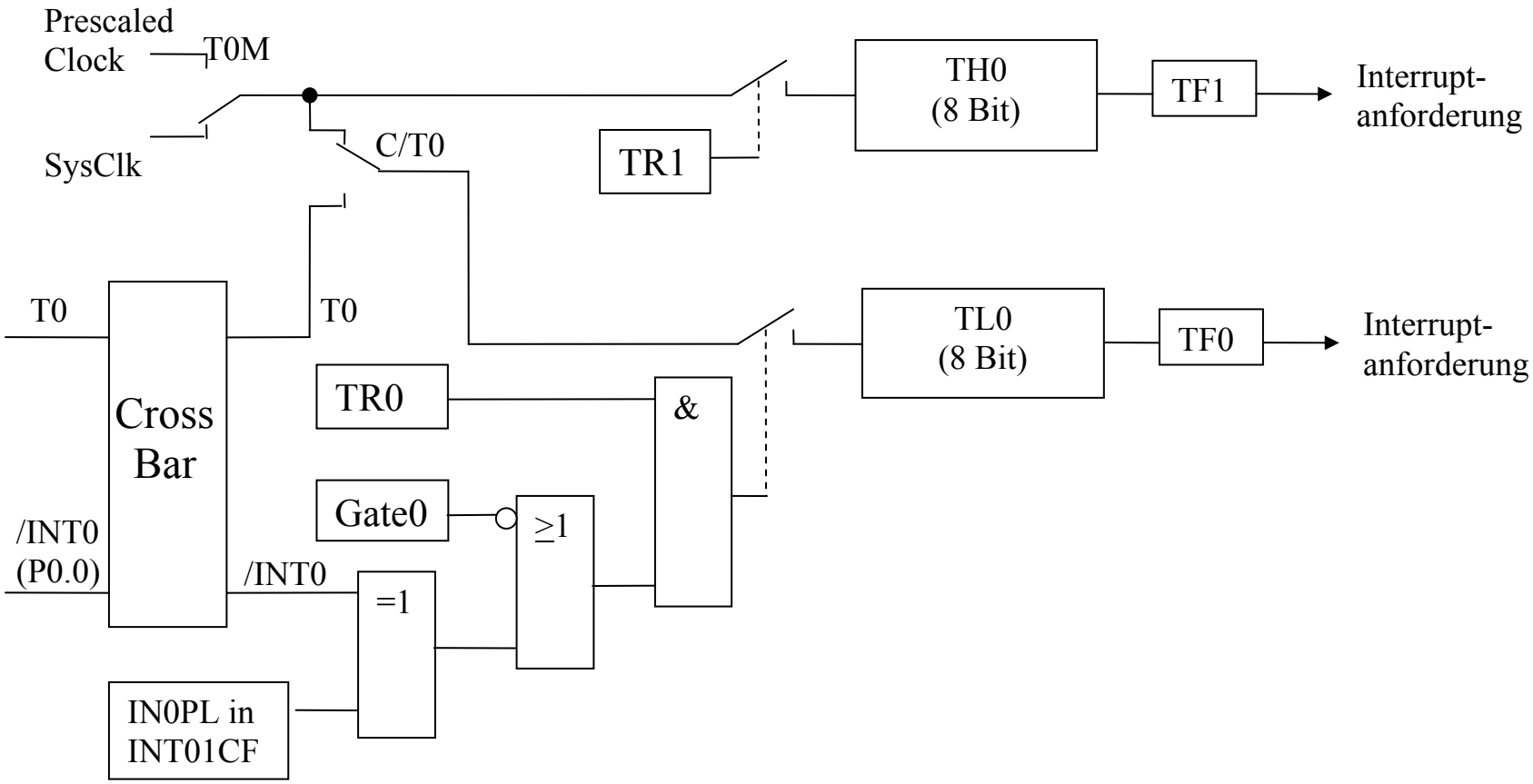
	Timer Control			Externer Interrupt				
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	Reset
TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0	00000000
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	

Das nachfolgende Bild zeigt die Einflüsse der Steuerbits auf die Auswahl der Taktquelle, das Starten und Anhalten der Timer, die verschiedenen Konfigurationsstrukturen (0, 1, 2) für die Timerregister und die Anschaltung des Überlaufbits. Die angegebene Schaltung wurde für den Timer 0 erstellt. Für den Timer 1 gilt die Struktur entsprechend.



Timer 0 Mode 0, 1, 2 Blockdiagramm

Betriebsart 3:



Timer 1 kann in Mode 0, 1, 2 arbeiten, aber ohne die Möglichkeit einen Interrupt auszulösen.

Timer 0 Mode 3 Blockdiagramm

TMOD Timer Mode = Betriebsart
SFR Adresse 0x89, (byteadressierbar)

Timer 1				Timer 0				
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	Reset
GATE1	C/T1	T1M1	T1M0	GATE0	C/T0	T0M1	T0M0	00000000
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	

Bit7: GATE1: Timer 1 Gate Control.
0: Timer 1 enabled when TR1 = 1 irrespective of /INT1 logic level.
1: Timer 1 enabled only when TR1 = 1 AND /INT1 is active as defined by bit IN1PL in register INT01CF.

Bit6: C/T1: Counter/Timer 1 Select.
0: Timer Function: Timer 1 incremented by clock defined by T1M bit (CKCON.4).
1: Counter Function: Timer 1 incremented by high-to-low transitions on external input pin (T1).

Bits5–4: T1M1–T1M0: Timer 1 Mode Select.

	T1M1	T1M0	Mode
	0	0	Mode 0: 13-bit counter/timer
	0	1	Mode 1: 16-bit counter/timer
	1	0	Mode 2: 8-bit counter/timer with auto-reload
	1	1	Mode 3: Timer 1 inactive

Bit3: GATE0: Timer 0 Gate Control.
0: Timer 0 enabled when TR0 = 1 irrespective of /INT0 logic level.
1: Timer 0 enabled only when TR0 = 1 AND /INT0 is active as defined by bit IN0PL in register INT01CF .

Bit2: C/T0: Counter/Timer Select.
0: Timer Function: Timer 0 incremented by clock defined by T0M bit (CKCON.3).
1: Counter Function: Timer 0 incremented by high-to-low transitions on external input pin (T0).

Bits1–0: T0M1–T0M0: Timer 0 Mode Select.

	T0M1	T0M0	Mode
	0	0	Mode 0: 13-bit counter/timer
	0	1	Mode 1: 16-bit counter/timer
	1	0	Mode 2: 8-bit counter/timer with auto-reload
	1	1	Mode 3: Two 8-bit counter/timers

TCON Timer Control = Zählersteuerung
SFR Adresse x088, (bitadressierbar)

Timer Control				Externer Interrupt				
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	Reset
TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0	00000000
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	

Bit7: TF1: Timer 1 Overflow Flag.
Set by hardware when Timer 1 overflows. This flag can be cleared by software but is automatically cleared when the CPU vectors to the Timer 1 interrupt service routine.
0: No Timer 1 overflow detected.
1: Timer 1 has overflowed.

Bit6: TR1: Timer 1 Run Control.
0: Timer 1 disabled.
1: Timer 1 enabled.

Bit5: TF0: Timer 0 Overflow Flag.
Set by hardware when Timer 0 overflows. This flag can be cleared by software but is automatically cleared when the CPU vectors to the Timer 0 interrupt service routine.
0: No Timer 0 overflow detected.
1: Timer 0 has overflowed.

Bit4: TR0: Timer 0 Run Control.
0: Timer 0 disabled.
1: Timer 0 enabled.

Bit3: IE1: External Interrupt 1.
This flag is set by hardware when an edge/level of type defined by IT1 is detected. It can be cleared by software but is automatically cleared when the CPU vectors to the External Interrupt 1 service routine if IT1 = 1. When IT1 = 0, this flag is set to '1' when /INT1 is active as defined by bit IN1PL in register INT01CF .

Bit2: IT1: Interrupt 1 Type Select.
This bit selects whether the configured /INT1 interrupt will be edge or level sensitive. /INT1 is configured active low or high by the IN1PL bit in the IT01CF register
0: /INT1 is level triggered.
1: /INT1 is edge triggered.

Bit1: IE0: External Interrupt 0.
This flag is set by hardware when an edge/level of type defined by IT0 is detected. It can be cleared by software but is automatically cleared when the CPU vectors to the External Interrupt 0 service routine if IT0 = 1. When IT0 = 0, this flag is set to '1' when /INT0 is active as defined by bit IN0PL in register INT01CF .

Bit0: IT0: Interrupt 0 Type Select.
This bit selects whether the configured /INT0 interrupt will be edge or level sensitive. /INT0 is configured active low or high by the IN0PL bit in register
0: /INT0 is level triggered.
1: /INT0 is edge triggered.

4.5.7.2 Timeranwendungen

Beispiel zur Anwendung des Timer 1 als Zeitgeber:

Funktion bei der Verwendung als 16 Bit Timer (Zeitgeber Mode 1):

Der Zähler ist ein Vorwärtszähler → $Z_{t+1} = Z_t + 1$

Beim Übergang von $FFFF_h$ auf 0000h wird das Bit TF1 (Timer Full) im Register TCON gesetzt und ein Interrupt ausgelöst. Die Zählweise bedingt, dass der Zähler mit einem Wert geladen werden muss, dass während des Aufwärtszählens genau die gewünschte Anzahl von Zählvorgängen durchgeführt wird.
Sollen drei Ereignisse gezählt werden, so kann zur Demonstration folgende Rechnung durchgeführt werden.

Der Zähler muss beim letzten Impuls von FFFFh auf 0000h gewechselt haben.

Endwert	0000 0000 0000 0000	Ereignis
-1	1111 1111 1111 1111	3
-1	1111 1111 1111 1110	2
-1	1111 1111 1111 1101	1

Der Ladewert wäre damit FFFDh

Zur direkten Berechnung, kann die folgende Operation durchgeführt werden:

```
  1 0000 0000 0000 0000
- 0000 0000 0000 0011
  1111 1111 1111 1101
```

Diese Operation entspricht aber gerade der Bildung des Zweier Komplements einer binären Zahl:

3	->0000 0000 0000 0011
Negieren	->1111 1111 1111 1100
+1	->1111 1111 1111 1101

Die verwendeten Compiler sind in der Lage eine solche Rechnung während des Übersetzungsvorganges selbst durch zuführen. Es muss daher nur die gewünschte Anzahl von Ereignissen mit einem negativen Vorzeichen versehen werden.

Beispiel:

Nach 100 Takten am Zähler soll ein Interrupt erzeugt werden.
➔ Anfangswert = -100d = FF9Ch

Erzeugung langer Impulse:
Soll ein Timer für die Erzeugung eines 10 ms langen Impulses herangezogen werden, so ergibt eine kurze Rechnung bei einer Taktrate von einem Megahertz (Prescaled Clock) die Angabe von -10000 für den Ladewert des Timers. Es werden damit 2 Bytes zur Aufnahme des Ladewertes benötigt -> Mode 1.

Das folgende Bild zeigt die Struktur des Timers 1 bei der Verwendung als 16 Bit Zähler. Auffällig ist, dass das Zählregister in zwei 8 Bit Register aufgeteilt ist, die getrennt angesprochen werden können (TL1, TH1). Die Taktung des Zählers kann abhängig von den angegebenen Steuerbits ein- bzw. ausgeschaltet werden. Zusätzlich ist die Quelle des Taktes wählbar (Systemtakt oder Prescaled Takt wählbar.

Erläuterung der Auswahlmöglichkeiten:

Aufteilung des Zählers in zwei 8 Bit Worte:

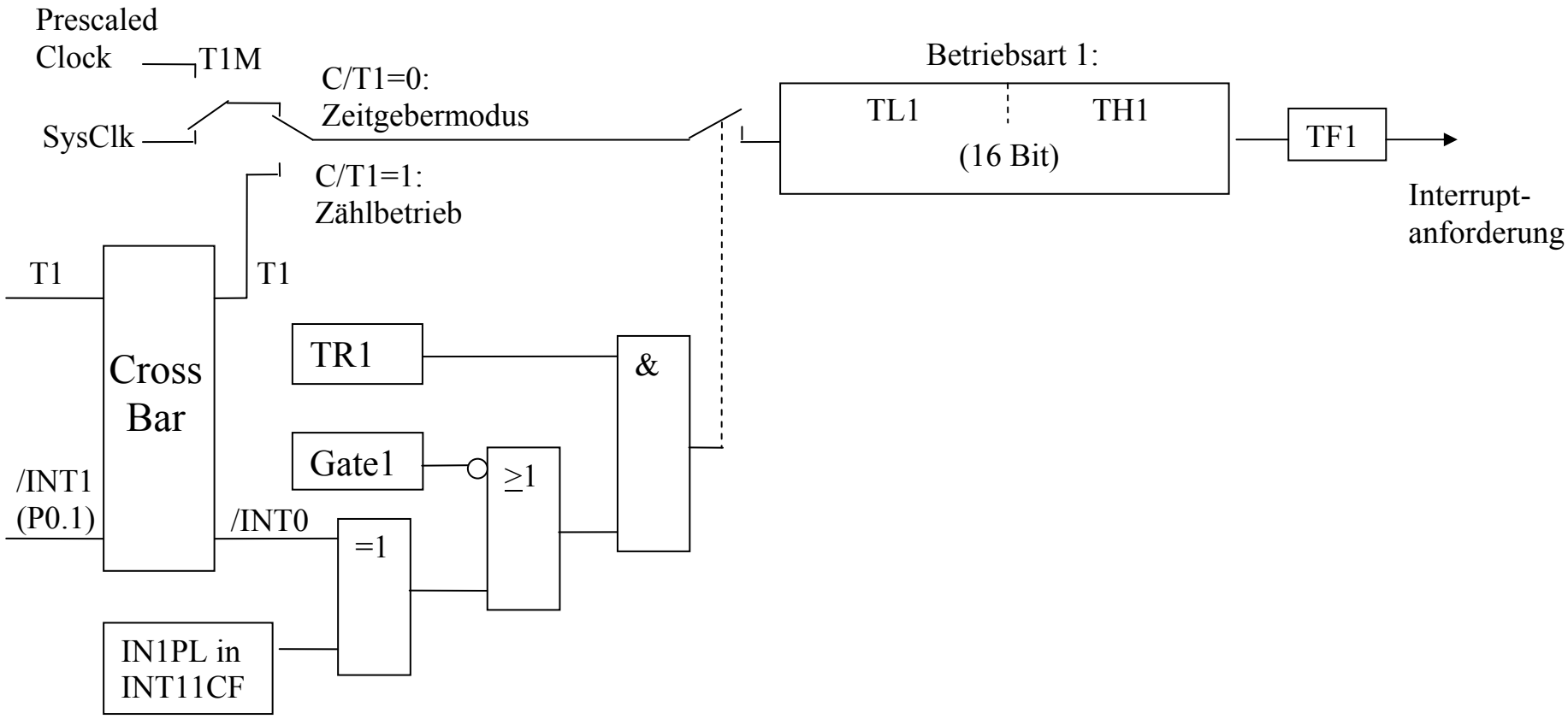
- TL1 Niederwertiges Byte
- TH1 Höherwertiges Byte

Bit C/T liegt im TMOD Register und steuert den Betrieb am Systemtakt oder als Zähler.

Bit TR1 started und stoppt den Zähler bei Gate = 0

Bit Gate1 ermöglicht die zusätzliche Steuerung des Taktes über ein externes Signal (/INT0):

Bit TF1 wird eim Überlauf des Zählers gesetzt. Nach dem Anspringen der Interrupt Service Routine wird das Bit automatisch gelöscht.



Ein Zähler/Zeitgeber des C8051F340 Prozessors (Betriebsart 1)

Vorgehen für die Ausgabe eines 10 ms Impulses

Forderung:

Am Anschluss P3.2 soll 10ms lang ein Impuls ausgegeben werden.
Taktfrequenz ist 12 MHz (prescaled clock)→ Zählertakt 1 MHz → 1 µs

Vorüberlegungen:

Zur Erreichung von 10ms benötigt man 10000 Zählertakte
Der Zähler muss daher wegen seiner Vorwärtszähleigenschaft mit:
Z = - 10000d = -270h =D8F0h geladen werden.

Belegung der Register:

TCON Register: Bitadressierbar sind TF1 und TR1
TF1 zeigt den Überlauf an und löst ggf. einen Interrupt aus.
TR1 hält den Zähler an oder startet ihn.

Überlauf	Start / Stop
TF1	TR1

TMOD-Register (nicht bitadressierbar)

Gate1 = 0 bedeutet der Zähler ist freilaufend
C/T1 = 0 ermöglicht den Zeitgeberbetrieb am Systemtakt
M0=0 und M1=1 wählen die Betriebsart 1 aus (16 Bit Timer)

Timer 1				Timer 0 ...
Gate1	C/T1	T0M1	T0M0	Gate0
0	0	0	1	

Ohne Struktogramm:

```
;Initialisieren des Timers
Starttim:   MOV TH1, #0D8h          ; MSB oder #HIGH(-10000)
            MOV TL1, # 0F0h         ; LSB  oder #LOW (-10000)
            MOV TMOD, #0001XXXX     ; Timer 1 Betriebsart 1
            MOV CKCON, #0000 1110b; ; Clock Prescaler /48
            SETB ET1                ; Interrupt bei Überlauf ermöglichen
            SETB EA                 ; Alle Interrupts ermöglichen
            SETB TR1                ; Zähler läuft, (TCON.6)
            SETB P3.2               ; Pulsbeginn
            RET                     ; Rücksprung

;Interrupt Routine

CSEG at 01Bh                ; Vektor für Timer 1 Interrupt
      TINT: CLR P3.2        ; Impuls beenden
            CLR TR1         ; Zähler stopp
            RETI
```

Bemerkung zur Verwendung von Timern:

Sollen die Zählerwerte direkt aus dem laufenden Programm heraus ausgewertet werden, so muss beim Auslesen der Zählerwerte folgendes beachtet werden.

Bedingt durch die 8-bit Datenpfade müssen die Bytes der 16-bit-Zähler nacheinander abgeholt werden. Tritt dazwischen ein Überlauf auf, so ist eines der beiden Datenbytes falsch.

z.B 0000000011111111
→ 0000000100000000

Lösung:

a) MSB lesen
b) LSB lesen
c) Überprüfung, ob sich das MSB geändert hat.

```
Zeitpkt:    MOV A, TH0              ; MSB von Timer 0
            MOV R0, TL0             ; LSB lesen
            CJNE A, TH0, Zeitpkt    ; notfalls wiederholen
            MOV R1, A               ; MSB gilt
            RET                     ; Rücksprung
```

4.6 Allgemeine Programmiergrundlagen in C

Allgemeine Programmiergrundlagen in C
Zur Programmierung von Mikrocontrollern in der Programmiersprache C werden bestimmte Programmierkonstrukte besonders häufig verwendet. Dies sind insbesondere Schleifen, die Verwendung von Verweisen, Bitverarbeitungsoperationen und Datenfelder.
An dieser Stelle sollen zunächst einfache Bestandteile der Programmiersprache C nochmals erläutert werden, um einen einfacheren Einstieg in die Assemblerprogrammierung zu finden.
Es werden typische Bearbeitungssituationen erläutert und die entsprechenden C-Progammlösungen vorgestellt. Dies soll keine Einführung in die Programmiersprache C ersetzen, sondern Grundlagen festigen, die zur Programmierung von Mikrocontrollern notwendig sind. Weitergehende Betrachtungen zu C-Programmen und ihre Umsetzung in Assemblercode werden in Abschnitt 7 durchgeführt.

4.6.1 Datentypen

4.6.1.1 Skalare Datentypen

Werden skalare Datentypen in ihrem Rechen- und Darstellungsaufwand klassifiziert, so sind Ganzzahltypen (z.B. unsigned char) mit deutlich geringerem Aufwand zu bearbeiten als Gleitkommazahlen. Diese Aussage bezieht sich sowohl auf die zu implementierende Software als auch auf die benötigte Hardware. Bei einfachen Mikrocontrollerarchitekturen sind daher Ganzzahloperationen zu bevorzugen. Zur Abbildung der üblichen Datenbitbreiten (z.B. 8 oder 16 Bit) werden die Datentypen unsigned char oder int verwendet.
Zur einfacheren Darstellung werden häufig Datentypdefinitionen vorgenommen, die eine einfache Notation erlauben:

typedef unsigned char	INT8U
typedef unsigned int	INT16U
typedef signed char	INT8
typedef signed int	INT16

alternativ werden auch folgende Definitionen verwendet:

typedef unsigned char	BYTE
typedef unsigned int	WORD

Im Folgenden sollen jedoch INT8U, INT8, INT16U oder INT16 verwendet werden.

4.6.1.2 Feldtypen

Die im vorherigen Abschnitt vorgestellten skalare Datentypen können dann in Feldern zusammengefasst werden. Es werden hier statische Definitionen bevorzugt. Dynamische allokierte Felder sind schwerer zu testen und bilden auch bei der Prüfung der Betriebssicherheit einen erheblichen Unsicherheitsfaktor.

Die Definition von Feldern wird in C in der folgenden Art vorgenommen:

Datentyp | Variablenname | Dimensionsangabe

Beispiel:
```
#define MAXSIZE 10
INT8U       arr1[MAXSIZE];      //eindimensionales Feld mit 10 Elementen
INT8U       arr2[2][MAXSIZE]    //zweidimensionales Feld mit 20 Elementen
```

Sinnvoll ist es die Feldgrenzen innerhalb einer define Anweisung festzulegen. Dies erlaubt Dimensionen an zentraler Stelle zu verwalten oder aufeinander abzustimmen, ohne den Code selbst zu ändern. An dieser Stelle soll nochmals daran erinnert werden, dass die Indexgrenzen bei 0 beginnen und bis MAXSIZE-1 gültig sind.

4.6.1.3 Verweise

Die Verwendung von Adressen für Variablen spielt bei der Mikrocontrollerprogrammierung eine wichtige Rolle. So gibt es hier auch Speicherbereiche auf die nur über Adressen zugegriffen werden kann. Es sind dabei meist spezielle Speicherbereiche mit einer Adresse zu laden, auf die dann im nächsten Schritt zugegriffen wird. Bei der Verwendung von C zur Programmierung werden diese Schritte jedoch meist vom Compiler eingefügt. Der Umgang mit Verweisen (Pointern) muss jedoch beherrscht werden. Pointer werden in C in der Deklaration von Variablen eingeführt.

1 Beispiel: `INT8U * ptv1;`

Diese Deklaration gibt an, dass in ptv1 eine Adresse auf eine Variable vom Typ INT8U steht. Sollen tatsächlich Daten verarbeitet werden, so muss jetzt der Speicherbereich selbst noch definiert werden:

Beispiel:
```
INT8U      v1;
INT8U * ptv1;
```

Die Adresse von v1 erhält man jetzt durch den sogenannten Adressoperator &. Der Zugriff auf die eigentliche Variable wird mit dem * Operator realisiert.

Beispiel:

```
INT8U      v1;
INT8U * ptv1;

ptv1 = &v1;            //Adressermittlung
*ptv1 = 5;             //Zuweisung
*ptv1 = *ptv1 + 1;     //Zugriff und Zuweisung
```

Felddefinitionen in C werden auch als Verweise betrachtet:

Beispiel: INT8U ar1[10];

Dabei wird eine Variable ar1 definiert, welche die Adresse des ersten Elementes enthält. Die weiteren Elemente werden von dieser Adresse beginnend abgelegt.
Der Zugriff *ar1 liefert den Wert des ersten Elementes zurück, *(ar1+1) den Wert des zweiten Elementes.

ar[0]=5; ar[8]=2; und
*ar=5; *(ar+8)=2 stellen

die gleichen Operationen dar.

4.6.2 Typische Abläufe in Mikrocontrollerprogrammen

4.6.2.1 Bitoperationen

Mikrocontrolleranwendungen werden nicht nur über bestimmte Rechenoperationen definiert, sondern es spielen auch Konfigurationen von speziellen peripheren Einrichtungen, wie zum Beispiel Zeitgeber (Timer) oder Schnittstellenhardware eine Rolle. Diese Konfigurationsregister müssen zum Teil mit Bitoperationen behandelt werden. Hierzu sollen typische Anwendungen beispielhaft vorgestellt werden:

Es wird jeweils angenommen, dass ein Byte manipuliert werden muss.
Da es in C nicht möglich ist, Bitkombinationen direkt anzugeben, sind meist Hexadezimalzahlen oder spezielle define Anweisung gebräuchlich.

Das Bitmuster 00010011 kann damit entweder Hexadezimal als 0x13 dargestellt oder indirekt mit #define b00010011 0x13 notiert werden. Die Definition eines Symbols hat den Vorteil einer besseren Lesbarkeit.

Beispiel 1 Setzen eines Bits in einem Byte

```
INT8U vv=0;

vv = 0x80
vv = vv | 0x08;          //Bitweise Oder-Verknüpfung
Alternativ:
vv |= 0x08;              //Bitweise Oder-Verknüpfung
```

Ablauf:

Nr	vv	Konstante 0x08	
1	0000 0000	0000 1000	Initialisierung
2	1000 0000	0000 1000	vv = 0x80
3	1000 1000	0000 1000	vv=vv | 0x08

Beispiel 2 Löschen von Bits in einem Byte

```
INT8U vv=0;

vv = 0xff
vv = vv & 0xf6;          //Bitweise Und-Verknüpfung
Alternativ:
vv &= 0xf6;              //Bitweise Und-Verknüpfung
```

Ablauf:

Nr	vv	Konstante 0xf6	
1	0000 0000	1111 0110	Initialisierung
2	1111 1111	1111 0110	vv = 0xff
3	1111 0110	1111 0110	vv=vv & 0xf6

Beispiel 3 Bitweise Komplementbildung

```
INT8U vv=0;

vv = 0x55
vv = ~vv; //Bitweise Komplementbildung
```

Ablauf:

Nr	vv		
1	0000 0000		Initialisierung
2	0101 0101		vv = 0x55
3	1010 1010		vv = ~vv;

Beispiel 4 Bitpositionierung mittels Index

```
INT8U vv=0;
INT8U pos=3;

vv = 1 << pos;//Setzen einer 1 an der 4. Position
              //Verschieben einer 1 um 3 Stellen
```

Ablauf:

Nr	vv	pos	
1	0000 0000	0000 0011	Initialisierung
2	0000 1000	3	vv = 1 << pos;

Beispiel 5 Zusammensetzung einer 16 Bit Zahl aus zwei 8 Bit Zahlen

```
INT8U       v1=5;  //Höherwertiges Byte
INT8U       v2=3;  //Niederwertiges Byte
INT16U      v3=0;
v3 = v1 << 8 + v2;     //v1*256 + v2
```

Beispiel 6 Zerlegung einer 16 Bit Zahl in zwei 8 Bit Bytes

```
INT8U       v1=0;  //Höherwertiges Byte
INT8U       v2=0;  //Niederwertiges Byte
INT16U      v3=333;
v1 = v3 >> 8;          //v1=v3/256 ; Rechtsverschiebung
v2 = v3 & 0x00ff       //Ausmaskierung der niederwertigeren Bits
```

4.6.2.2 Arbeiten mit Feldern

Felder werden in C durch die Angabe eines Grundtyps, eines Variablennamens und der Anzahl der Elemente in einem Feld definiert:

Beispiel: Zugriffsarten

```
INT8U ar1[10];      //10 Elemente vom Typ INT8U
                    //Es werden 10 Bytes belegt
```

Der Zugriff auf ein einzelnes Element kann dann entweder durch Angabe einer Indexzahl oder durch Berechnung einer Adresse vorgenommen werden.

```
INT8U ar1[10];      //10 Elemente vom Typ INT8U
                    //Es werden 10 Bytes belegt

ar1[0]=1;           //Zugriff über Index
ar1[1]=ar1[0];

*ar1 = 1;           //Zugriff über Adressberechnung
*(ar1+1)=*ar1;
```

Beispiel: Initialisierung eines Feldes

```
INT8U ar1[10];      //10 Elemente vom Typ INT8U
                    //Es werden 10 Bytes belegt
INT8U ii;           //Indexzähler

for(ii=0;ii<10;ii++)        //Zugriff über Index
        ar1[ii]=0;

for(ii=0;ii<10;ii++)        //Zugriff über Index
        *(ar1+ii)=0;
```

Beispiel: Verschiebung des Feldinhaltes

In diesem Beispiel soll gezeigt werden, wie der Feldinhalt ab einem vorgegebenen Index nach unten verschoben wird. Die Elemente im unteren Bereich werden dabei überschrieben. Die Elemente im frei werdenden oberen Bereich werden neu initialisiert.

```
INT8U ar1[10];      //10 Elemente vom Typ INT8U
                    //Es werden 10 Bytes belegt
INT8U ii;           //Indexzähler
INT8U start=3;      //Startwert
for(ii=start;ii<10-start;ii++)          //Umkopieren
        ar1[ii-start]=ar1[ii];

for(ii=10-start;ii<10;ii++)             //Initialisierung der freien Elemente
        ar1[ii]=0;
```

4.6.2.3 Schleifen

Die Programmiersprache C stellt 3 unterschiedliche Typen von Schleifen zur Verfügung:

1. while-Schleife
2. do-while-Schleife
3. for-Schleife

Vom Programmablauf sind while- und for-Schleifen identisch. Bei der for-Schleife werden typische Aktionen wie die Initialisierung und die Weiterschaltung mit in die Schleifenkonstruktion integriert.

Die Abarbeitung in einer while- oder for-Schleife zeigt Bild 4.4:

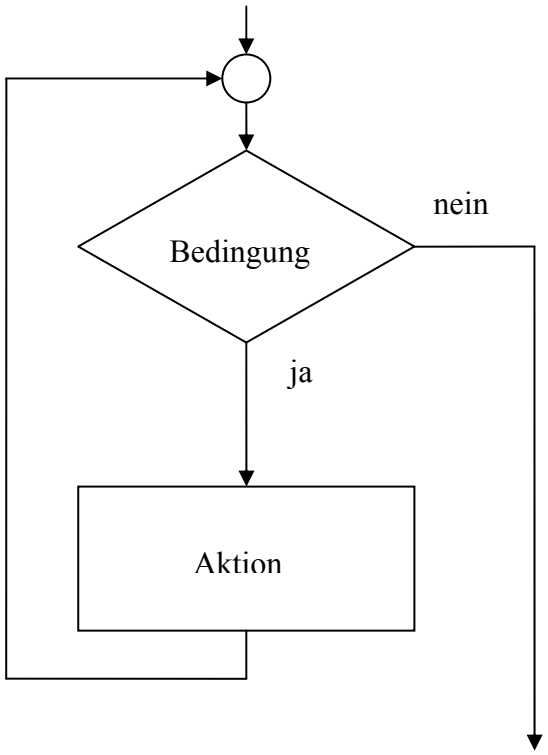


Bild 4.4 Programmablauf in einer while- oder for-Schleife

Entscheidend bei einer while-Schleife ist die Abfrage der Schleifenbedingung am Anfang der Schleife. Ist die Bedingung auch beim Eintritt in die Schleife nicht erfüllt, so wird auch keine Aktion durchgeführt und die Schleife wird sofort verlassen.

Beispiel: Summe der Zahlen von 0 bis 10 mit einer while-Schleife

```
INT8U ii=0;
INT8U sum=0;
while(ii<=10){
        sum = sum +ii;
        ii++;
}
```

Beispiel: Summe der Zahlen von 0 bis 10 mit einer for-Schleife

```
INT8U ii;
INT8U sum;
for(ii=0, sum=0;ii<=10;ii++){
        sum = sum +ii;
}
```

Eine besondere while- oder for-Schleife stellt die Endlos-Schleife dar. In Mikrocontrollerapplikationen ist es notwendig, dass ständig ein gültiger ausführbarer Code vorhanden ist. Es gibt auch zunächst kein Betriebssystem, das Programme startet und anhält. Diese Grundfunktion kann mit einer einfachen Endlosschleife sichergestellt werden.

Endlosschleife while-Version:

```
while(1){
}
```

Endlosschleife for-Version

```
for( ; ; ){
}
```

Die do-while-Schleife vertauscht gerade die Reihenfolge der Abfrage und der Aktionsausführung. Es wird daher die Aktion der Schleife mindestens einmal durchlaufen. Das bedeutet auch, dass auch bei ungültiger Bedingung die Operationen konsistent durchgeführt werden können.

Die Abarbeitung in einer do-while -Schleife zeigt Bild 4.5:

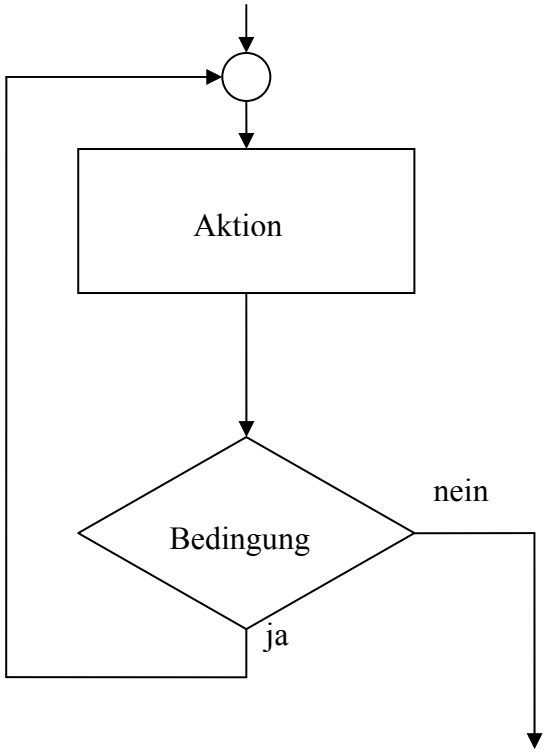


Bild 4.5 Programmablauf in einer do-while-Schleife

Beispiel: Summe der Zahlen von 0 bis 10 mit einer do-while-Schleife

```
INT8U ii=0;
INT8U sum=0;
do{
        sum = sum +ii;
        ii++;
} while(ii<=10)
```

Es stehen zwei weitere Möglichkeiten zur Verfügung Schleifen in ihrem Ablauf zu ändern:

```
break;
continue;
```

Break unterbricht den Schleifenablauf und die Schleife wird komplett verlassen.
Continue veranlasst einen Sprung zum Schleifenende. Die Schleife wird aber nicht verlassen.

4.6.3 Aufgaben zur Selbstkontrolle

1. Bilden sie den Mittelwert der Zahlen in einem Feld INT8U ar[10]. Die Anzahl der gültigen Feldelemente von 0 bis kk-1 soll in der Variable kk gespeichert sein. Überprüfen sie zunächst, ob eine Variable vom Type INT8U zur Summenbildung geeignet ist.

2. Bilden sie die Differenzen zweier benachbarter Zahlen ar[ii+1]-ar[ii] und speichern sie das Ergebnis in ein neues Feld. Verwenden sie zum Zugriff einmal die Indexschreibweise und zusätzlich die Adressenschreibweise.

3. Zur Berechnung eines gleitenden Mittelwertes (moving mean) soll in einem Feld aus jeweils 4 aufeinander folgenden Elementen der Mittelwert gebildet werden und das Ergebnis in ein neues Feld geschrieben werden.

4. In einem vorgegebenen Feld soll das Maximum und das Minimum der abgelegten Zahlen gebildet werden. Das Ergebnis soll in speziellen Variablen abgelegt werden.

4.7 Programme in C für den Mikrocontroller

4.7.1 Anforderungen

Die Programmierung komplexer Algorithmen mit Hilfe einer Assemblersprache ist sehr fehlerbehaftet und meist langwierig. Auch bei Systemen bei denen unterschiedliche Ereignisse in konsistenter Reihenfolge abgearbeitet werden müssen, führen Assemblerprogramme sehr schnell zu unübersichtlichen Beschreibungen. Die Lesbarkeit auch für andere Programmierer ist im Gegensatz zu höheren Programmiersprachen wie C stark eingeschränkt. Dies kann auch durch erhöhten Kommentierungsaufwand nicht wett gemacht werden. Dazu kommen Strukturierungshilfen in den höheren Programmiersprachen, die es erlauben anerkannten Forderungen des Software Engineering (z.B. strukturierte Programmierung) einfach durch die Verwendung der vorhandenen Sprachkonstrukte nachzukommen (While, For usw. aber kein Goto!!). Ebenso wird bei Unterprogrammaufrufen die notwendige Verwaltungsarbeit bei der Übergabe von Parametern oder der Verwendung lokaler Variablen mit erledigt. Der Programmierer braucht sich um solche Standardaufgaben nicht zu kümmern.

Es genügt jedoch nicht nur eine höhere Programmiersprache zu lernen und sein Problem zu formulieren. Bei der Programmerstellung für Mikrocontrollersysteme ist die Architektur des Controllers von großer Wichtigkeit. Zum Ansprechen von peripheren Einheiten ist es beispielweise notwendig, bestimmte Steuerinformationen abzusetzen oder auf externe Anforderungen zu reagieren.
Zu diesem Zweck werden Definitionen eingeführt, die den Zugriff, auf speziell für jeden Prozessortyp andere Speicherzellen, erlaubt. Die Verwendung von Datentypen ist ebenso abhängig von der Architektur des Prozessors. Zwar erlauben die Compiler meist die freie Verwendung von Datentypen, man sollte jedoch nicht vergessen, dass Gleitkommazahlen beispielsweise nicht unbedingt zu den effektiv verarbeitbaren Zahlen eines einfachen Mikrocontrollers gehören. Hier werden dann automatisch Softwareroutinen hinzugenommen, die jedoch den Programmcode erheblich erweitern.

Die Besonderheiten einer Programmierung in einer Hochebenensprache sind daher:

- Spezielle Definition zum Bekanntmachen der Prozessorregister
- Zusatzinformationen bei Unterprogrammen, wenn diese als Interrupt Service Routinen dienen sollen
- Die Abbildung von Aufrufen zur Dateiverarbeitung z.B. zur Bedienung von seriellen Schnittstellen

4.7.2 Verwendung der Datentypen in C

Bei der Verwendung der vorhandenen Datentypen in C soll im Folgenden die Zuordnung für den Prozessor 8051 beschrieben werden. Diese Definitionen sind wichtig, um den Zahlenbereich, der damit definierten Variablen bei der Programmierung berücksichtigen zu können.

Zuordnung der Datentypen zum Speicherplatz und dem Zahlenbereich:

Datentyp	Größe	Wertebereich
bit	1 Bit	0 oder 1
signed char	1 Byte	-128 bis +127
unsigned char	1 Byte	0 bis 255
signed int	2 Byte	-32768 bis +32767
unsigned int	2 Byte	0 bis 65535
signed long	4 Byte	-2147483648 bis +2147483647
unsigned long	4 Byte	0 bis 4294967295
float	4 Byte	±1,176E-38 bis ±3,40E+38
pointer	1-2 Byte	Adresse einer Variablen

Datentypen für Spezial Function Registers (SFR)

Datentyp	Größe	Wertebereich
sbit	1 Bit	0 oder 1
sfr	1 Byte	0 bis 255
sfr16	2 Byte	0 bis 65535

Beispiele für Variablendeklarationen:

```
sfr P0 = 0x80;          //Adressenzuordnung
sbit RI = 0x98;
```

4.7.3 Angabe von Speicherbereichen

Bei der Definition des Datentyps wird der Zahlenbereich der Variablen festgelegt. Soll jedoch zusätzlich auf die Organisation der Datenbereiche Einfluss genommen werden, so können zusätzlich Angaben gemacht werden, die einem Datum als Speicherort den internen oder den externen RAM zuweisen.

Mögliche Speicherbereiche sind:

Speichertyp	Beschreibung
data	direkt adressierbarer interner Datenspeicher (Bereich 00h-7Fh)
idata	indirekt adressierbarer interner Datenspeicher (Bereich 00h-FFh)
bdata	bitadressierbarer interner Datenspeicher (Bereich 20h-2Fh)
xdata	externer Datenspeicher (Bereich 0000h – FFFFh)
code	Programmspeicher (Bereich 0000h – FFFFh)

Beispiele für Variablendeklarationen:

```
char data var1;
char code text[]=”Error Message:”;
unsigned long xdata array[100];
unsigned char xdata vector[10][4][4];
```

4.7.4 Definition von Interrupt Service Routinen

Interrupt Service Routinen werden in C als Unterprogramme ohne Parameter beschrieben. Es ist jedoch ein Zusatz notwendig, um das Unterprogramm mit der Einsprungadresse zu koppeln, die vom Prozessor festgelegt ist.

Beispiel:

```
void Timer_0_isr(void) interrupt 1
```

Nach einem zusätzlichen Schlüsselwort (interrupt) wird eine Nummer (n) angegeben, die mit der hexadezimalen Einsprungadresse des gewünschten Interrupts korrespondiert.

Hierzu gilt folgende Regel: N= (Interrupt-Vektor-Adresse-3)/8

Beispielsweise ergibt sich dann für den Timer 2 Interrupt mit der Einsprungadresse 2Bh=43d

N = (43-3)/8 = 5

4.7.5 Anwendungsbeispiel

In Abhängigkeit von Tastern soll eine Leuchtdiode als Blinklicht gesteuert werden. Die Taster sollen folgende Funktionen auslösen:

Taster 1:	LED soll ständig leuchten	Anschluss:	Port P3.3
Taster 2:	LED ausschalten	Anschluss:	Port P3.2
Taster 3:	LED blinkt	Anschluss:	Port P3.1

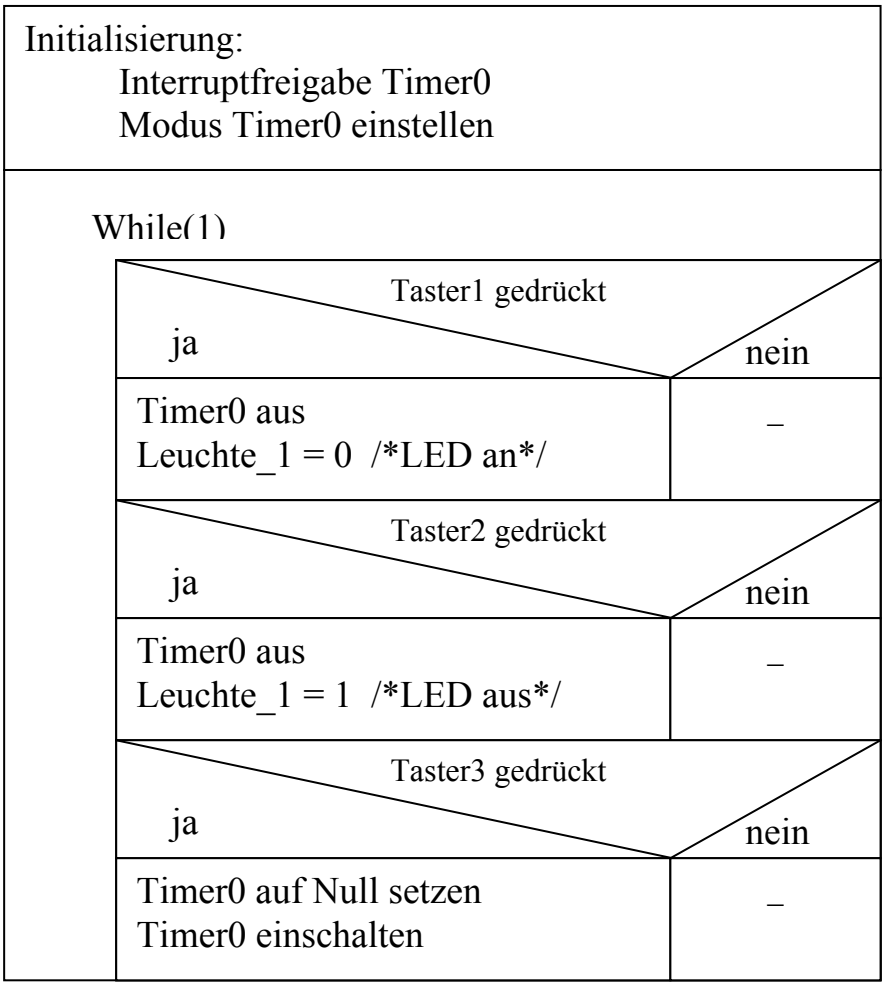
Die Leuchtdiode ist an Port P2.2 angeschlossen.

Zur Verwendung von symbolischen Namen im Programm werden folgende Definitionen vorgenommen:

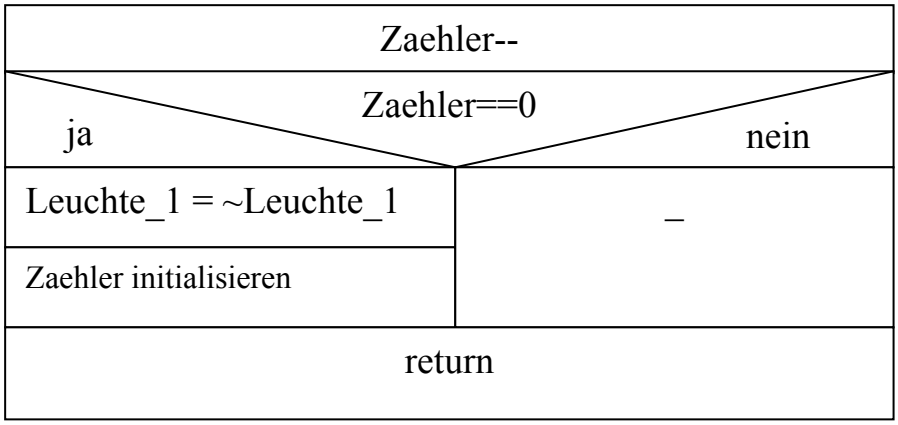
```
/* Taster */
sbit Taster_3 = P3^3;      //Port P3.3
sbit Taster_2 = P3^2;      //Port P3.2
sbit Taster_1 = P3^1;      //Port P3.1

/* Leuchtdiode */
sbit Leuchte_1 = P2^2;   //Port P2.2
```

Struktogramm:

Hauptprogramm:



Interruptserviceroutine



Programmbeschreibung:

```
#include "C8051F340.h"
//Taster
sbit Taster_3= P3^3;        //Port P3.3
sbit Taster_2= P3^2;        //Port P3.2
sbit Taster_1= P3^1;        //Port P3.1
//Leuchtdiode
sbit Leuchte_1= P2^2;     //Port P2.2
#define V_BLINK 10                    /* bestimmt die Blinkdauer */
unsigned char zaehler = V_BLINK;   /* Initialisierung des Zaehlers */
void main (void){
        P2MDOUT=0xff; //P2 als Push-Pull geschaltet
        Leuchte_1 = 1;      /* Lampe ausschalten */
        ET0=1;              //Enable Timer 0 Interrupt
        EA=1;               //Enable alle Interrupts
        TMOD = 0x1;         /* Timer 0, Modus 1, 16 Bit Zähler */
        TCON = 0x0;
        while(1){                       /* Endlos Schleife */
                if(!Taster_1){          /* Taster gedrueckt == 0 */
                        TR0 = 0;        /* Timer anhalten */
                        Leuchte_1 = 0;  /* LED an */
                }
                if(!Taster_2){
                        TR0 = 0;        /* Timer anhalten */
                        Leuchte_1 = 1;  /* LED aus */
                }
                if(!Taster_3){
                        TL0 = 0;        /* Zaehler auf 0 setzen */
                        TH0 = 0;
                        TR0 = 1;        /* Timer an */
                }
        }
}
void Timer_0_ISR(void) interrupt 1          /* Timer 0 Interrupt Routine */
{
        zaehler--;
        if(zaehler==0){
                Leuchte_1 = ~Leuchte_1;     /* Zustandsinvertierung der LED */
                zaehler = V_BLINK;          /* Zaehlerinitialisierung */
        }
        return;
}
```

5 Parallele Ein-/Ausgabe

Der im Prozessor 8051 vorhandene Datenbus transportiert alle Informationen, die während des internen Arbeitsablaufes des Prozessors anfallen.
Damit liegen auch Daten, die ausgegeben werden sollen, zu mindestens temporär auch hier vor. Ausgabedaten müssen jedoch kontinuierlich an den Ports zur Verfügung stehen und dürfen nicht von den internen Arbeitsabläufen abhängig sein.
Sollen die Daten kontinuierlich anliegen sind also Zusatzmaßnahmen notwendig. Zur Ausgabe der Daten bietet es sich an die Wortbreite entsprechend des internen Datenbusses. (8 oder 16 Bit) zu wählen. Die Ausgabedaten werden zwischengespeichert und dann an den parallelen Ausgabekanälen (Ports) nach außen gegeben. Für die Eingabe werden die Werte an den Anschlüssen gelesen und ebenfalls zwischengespeichert.

⇒ Diesen Vorgang bezeichnet man dann als parallele Ein- bzw. Ausgabe.

5.1 Aufbau der Ports

Die Verwendungsmöglichkeiten der parallelen Ein-/und Ausgänge hängen stark von den elektrischen Eigenschaften eines Anschlusses ab. Spannungspegel und Treiberfähigkeiten müssen eingehalten werden um z.B. TTL Logikgatter ansprechen zu können. Ports dienen jedoch nicht nur zum Ansteuern anderer Logikbausteine, sondern werden auch direkt dazu verwendet, Anzeigeeinheiten zu treiben z.B. LEDs um möglichst wenig zusätzliche Bauelemente unterbringen zu müssen. Damit sollen zusätzliche Kosten vermieden werden. Für die Kostenabschätzung beim Systementwurf spielen daher auch die Schaltungsstruktur und die damit verbundenen Möglichkeiten eine wichtige Rolle. Im Folgenden soll daher nicht nur auf die prinzipielle Wirkungsweise eines Ports bei einem Schreib- oder Lesevorgang eingegangen werden, sondern auch die interne Wirkungsweise erklärt werden.

Der Aufbau eines Ports wird durch die Forderung geprägt, sowohl Lesen als auch Schreiben zu können. Weiter muss der Wert der ein- oder ausgelesenen Information zur Entkopplung der Ein-/Ausgabe vom internen Datenbus zwischengespeichert werden können. Eine Struktur, die diese Aufgaben erfüllt, ist im nächsten Bild dargestellt.

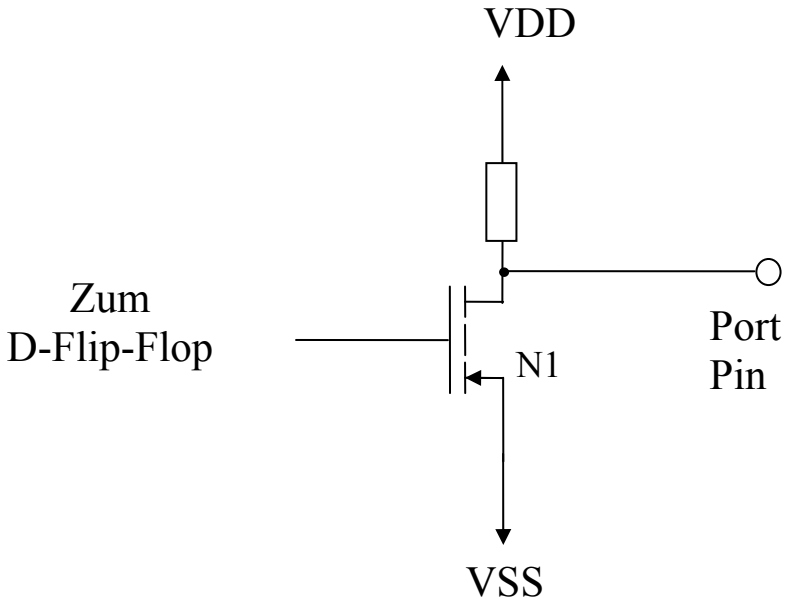
Die Aufgaben und Eigenschaften des Speicherelementes, das als D-FlipFlop ausgeführt ist, sind:

- Speichern eines Bits
- Übernahme von Daten des internen Bus beim Schreibimpuls
- Bestandteil des internen RAM´s (SFR – Special Funktion Register)

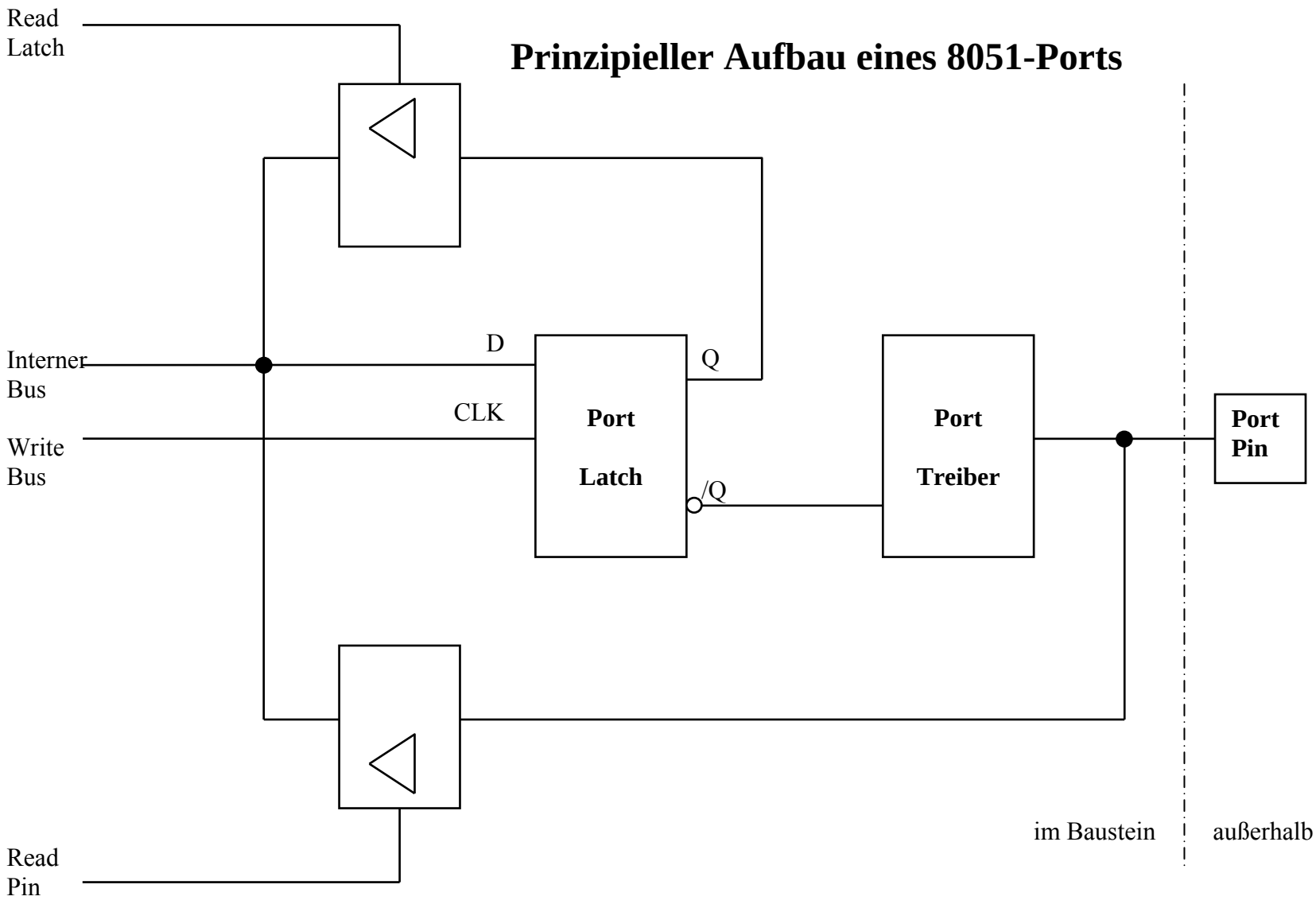
Der Ausgang des D-FF kann auf den internen Bus geschaltet werden. Dies ist notwendig für bestimmte Befehle (Read Modify Write Befehl)

Funktionsweise der Schaltung bei der Ausgabe:
Zum Verständnis der Ausgabe soll die im Blockschaltbild angegebene Treiberstufe durch einen Inverter realisiert sein.
Der Transistor (meist FET n-Kanal) wird vom Ausgang /Q des D-FF gesteuert. Durch die invertierende Eigenschaft des Inverters erscheint Q am Pin.

Funktionsweise der Schaltung beim Einlesen:
Der Transistor des Ausgangstreibers muss sperren, ansonsten würde die Information am Pin zur Masse hin kurzgeschlossen.
Bei Q =1 legt der interne Pullup - Widerstand den Pin auf High (5V). Die externe Quelle kann so den Eingang zur Eingabe einer logischen 0 auf Low (0V) ziehen. Ansonsten wird eine logische 1 erkannt. Der Wert wird dann zum einen auf den internen Bus geschaltet und zum anderen in das vorhandene Speicherelement (D-FF) eingelesen.



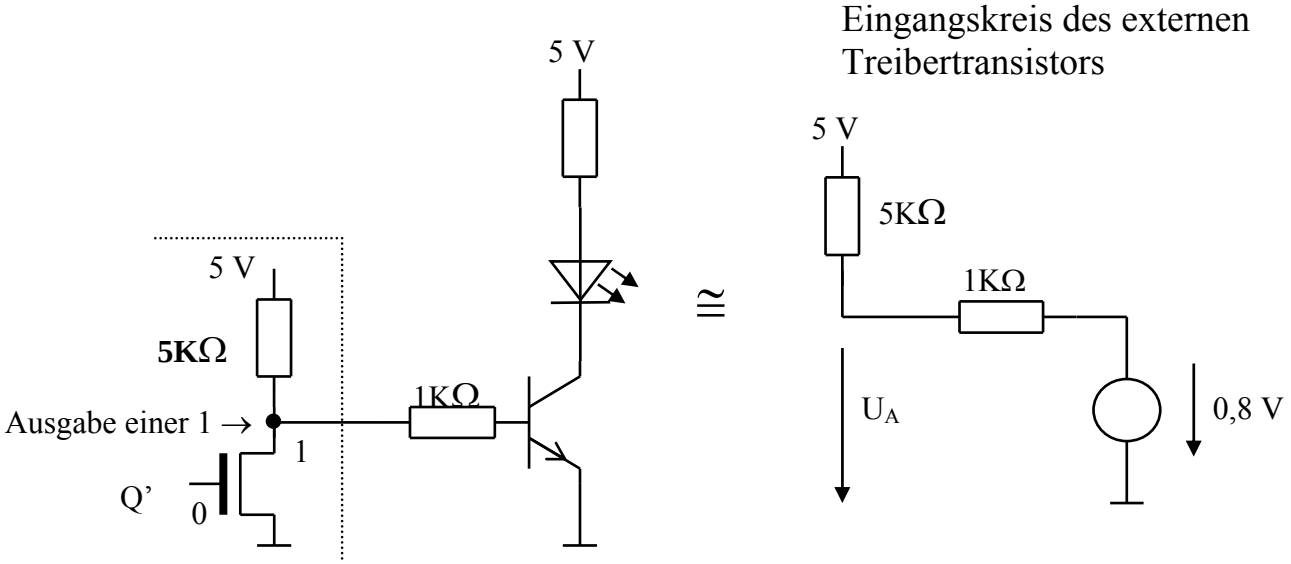
Vereinfachter Aufbau der Treiberstufe


Prinzipieller Aufbau eines 8051-Ports
Read
Latch
Interner
Bus
Write
Bus
D
CLK
Q
/Q
Port
Latch
Port
Treiber
Port
Pin
Read
Pin
im Baustein
außerhalb

Zur Verwendung der Informationen in den Speicherelementen der Anschlüsse muss darauf geachtet werden dass nicht alle Lesebefehle bei einem Port tatsächlich den Eingangswert abfragen. Ausnahmen stellen hier die sogenannten Read-Modify-Write Befehle dar. Bei diesen Befehlen wird der Wert des Speichers gelesen, modifiziert und wieder in das Speicherelement zurückgeschrieben. Je nach Ausgangsbeschaltung (z.B. eine externe Last bei der Anschaltung einer Diode) kann bei diesen Operationen das Einlesen zu einer falschen Information führen. Deshalb wird der Inhalt der Speicherzelle bei den folgenden Operationen als Ausgangspunkt verwendet.

Befehl	Funktion	Beispiel
ANL	Logisches UND	ANL P1,A
ORL	Logisches ODER	ORL P2,A
XRL	Exklusives ODER	XRL P3,A
JBC	Jump if bit is set and clear bit	JBC P1.1, Label
CPL	Complement bit	CPL P3.0
INC	Inkrementiere Byte	INC P4
DEC	Dekrementiere Byte	DEC P5
DJNZ	Dekrementiere und Springe wenn Byte != 0	DJNZ P3,Label
MOV Px.y,C	Bringe das Carry Bit zum Bit y vom Port x	
CLR Px.y	Lösche Bit y von Port x	
SETB Px.y	Setze Bit y von Port x	

Als Beispiel für die Notwendigkeit der Arbeitsweise der Read Modify Write Befehle soll die Anschaltung einer Treiberstufe dienen.



Spannungen für den TTL - Pegel

Eingangsspannung: L = - 0,5V < U_{iL} < 0,9 V
H = 1,9 V < U_{ich} < 5,5 V

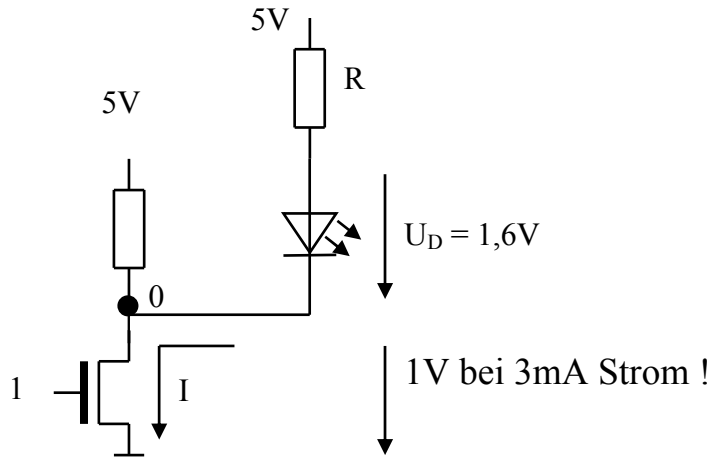
Ausgangsspannung: L = U_{OL} < 0,45 V
H = U_{OH} > 2,4 V

Berechnung des Spannungsteilers:

$$U_A = \frac{1K\Omega}{5K\Omega + 1K\Omega} \times (5V - 0{,}8V) + 0{,}8V = 1{,}5V$$

Die entstehende Spannung sollte als logische Eins beim Einlesen erkannt werden. Sie liegt jedoch unter dem Wert, der sicher als logische Eins erkannt wird.

2. Annahme einer Außenbeschaltung mit LED



Erfahrungswert:

Um eine Diode ausreichend hell leuchten zu lassen:

I_{LED} = 2-3mA

→ U_{OL} > 0,45 V ≈ 1V

Bei einer Betriebsspannung von 3.3 V ergibt sich:

$$R = \frac{U_{CC} - U_{LED} - U_{OL}}{I_{LED}} = \frac{3.3V - 1{,}6V - 1V}{3mA} = 200\Omega$$

Werden Dioden verwendet, die im grünen und blauen Bereich arbeiten, muss mit höheren Durchlassspannungen gerechnet werden. Blaue Dioden liegen dabei im Bereich der Versorgungsspannung von 3.3 V.

5.2 Betrieb der Ports des C8051F340

Bei neueren Prozessortypen, die auf der ursprünglichen 8051 Architektur beruhen, werden zum Teil eine zusätzliche Funktionalität und eine erweiterte Konfigurationsmöglichkeit angeboten. Hier spielen die alternative Verwendung als analoge Eingänge (Komparatoren, AD-Wandler) oder unterschiedliches Verhalten bei digitalen Eingängen eine Rolle. Werden die Anschlüsse beim vorliegenden Prozessor C8051F340 nicht für die Verarbeitung von analogen Signalen oder dem Anschluss von Kommunikationsschnittstellen eingesetzt, so können sie als allgemeine digitale Anschlüsse (GPIO General Purpose Inputs/Outputs) verwendet werden. Die Daten für die Ports P0 bis P4 werden beim Lesen und Schreiben im internen RAM abgelegt und über entsprechende Schaltungen auf und von den Pins des Prozessors weitergeleitet. Die Ports P0 bis P3 sind bit- und byteadressierbar während Port4 nur byteadressierbar ist. Die Einstellungsmöglichkeiten sollen im Folgenden für den Port 0 erläutert werden. Das Verhalten des anderen Ports erfolgt nach dem Datenblatt [DB1] dann entsprechend.

Die Konfiguration der Anschlüsse ist weiter von einem Kreuzschienenverteiler (Cross Bar) bestimmt, der flexibel vorhandene Einheiten wie die serielle Schnittstelle oder den I2C-Bus auf die Anschlüsse verteilt. Ein Kreuzschienenverteiler besteht aus einer Anzahl horizontaler und vertikaler Leitungen die an ihren Kreuzungspunkten verbunden werden können. Damit können die Signale beliebig von den horizontalen Leitungen auf die vertikalen Leitungen verteilt werden oder umgekehrt. Das prinzipielle Schema des Kreuzschienenverteilers beim C8051F340 ist im folgenden Bild dargestellt.

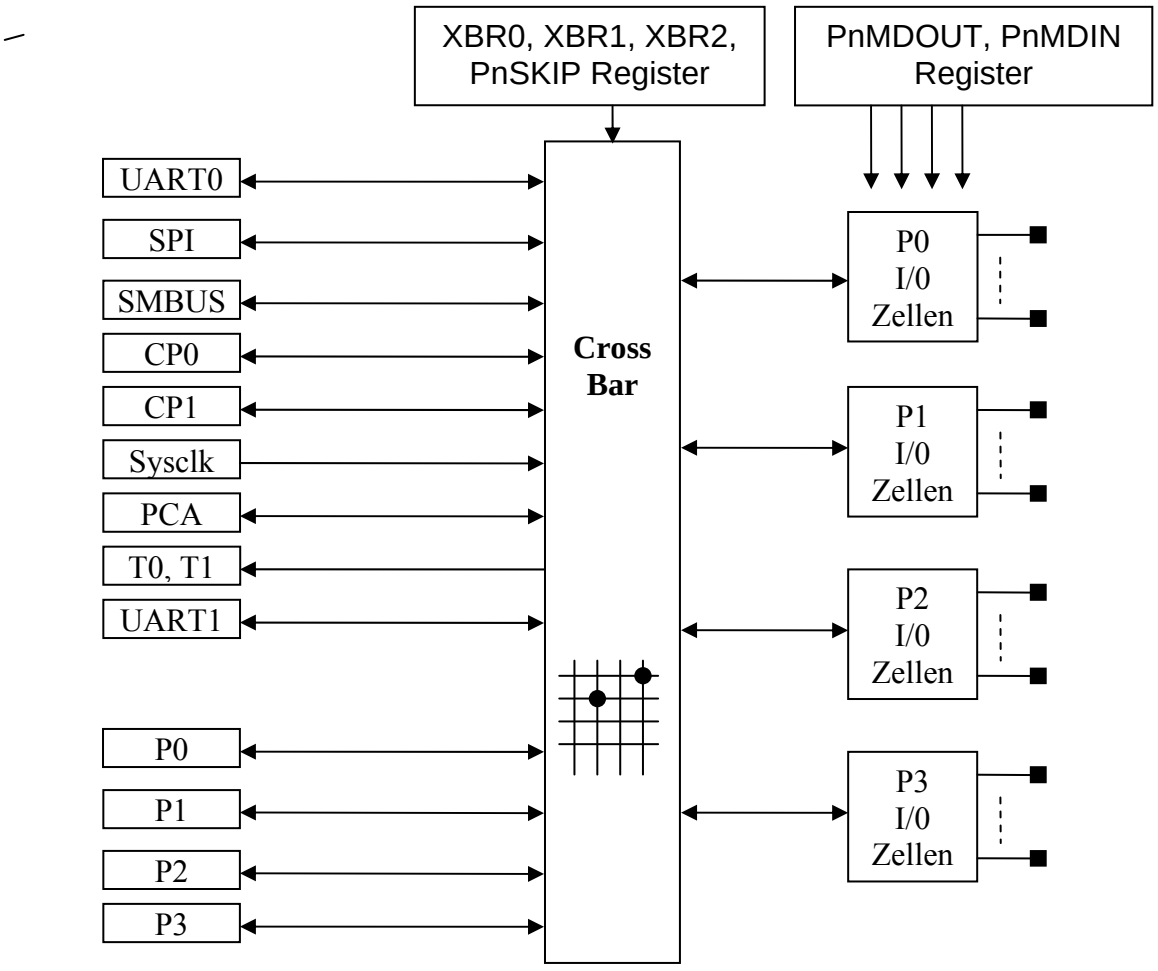
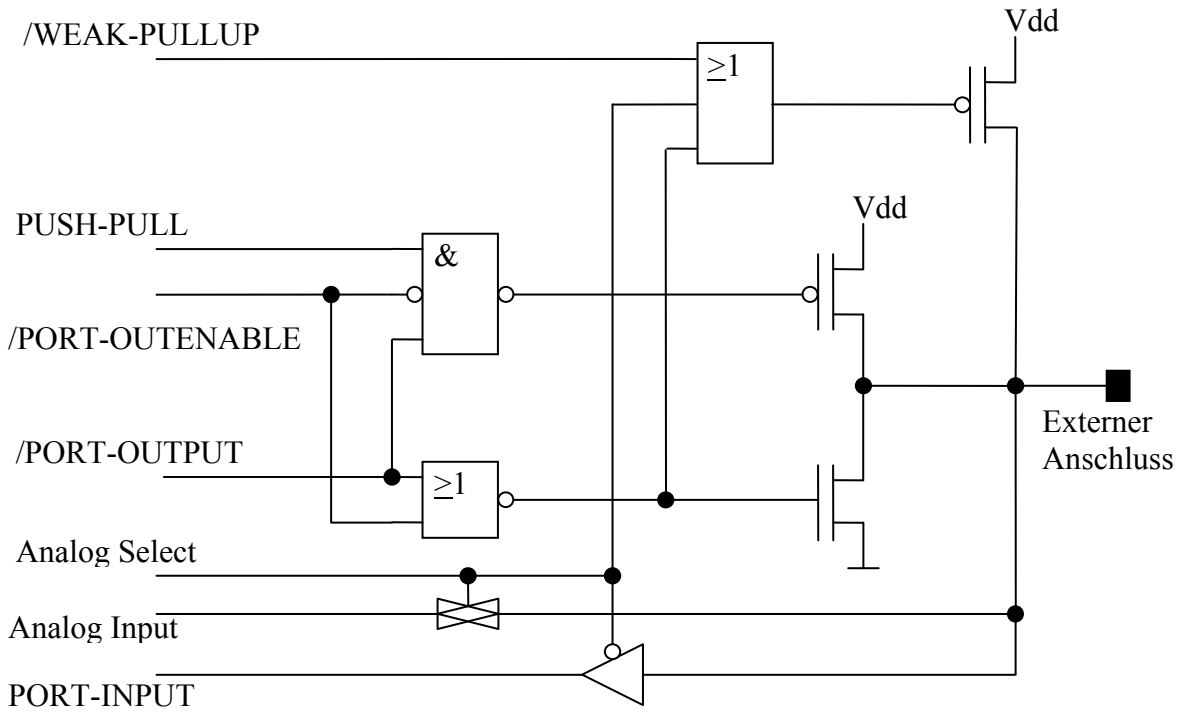


Bild Prinzipielle Funktion des Kreuzschienenverteilers (Cross Bar)

Die folgende Zeichnung zeigt eine Ein/Ausgabezelle für einen externen Anschluss



Schaltung einer Zelle zum Betrieb eines externen Anschlusses

Daten- und Signalleitungen:

PORT-OUTPUT wird vom internen RAM gelesen
PORT-INPUT wird zum internen RAM geleitet
Analog Input wird zur entsprechenden analogen Einheit z.B. A/D-Wandler geleitet

Steuerleitungen

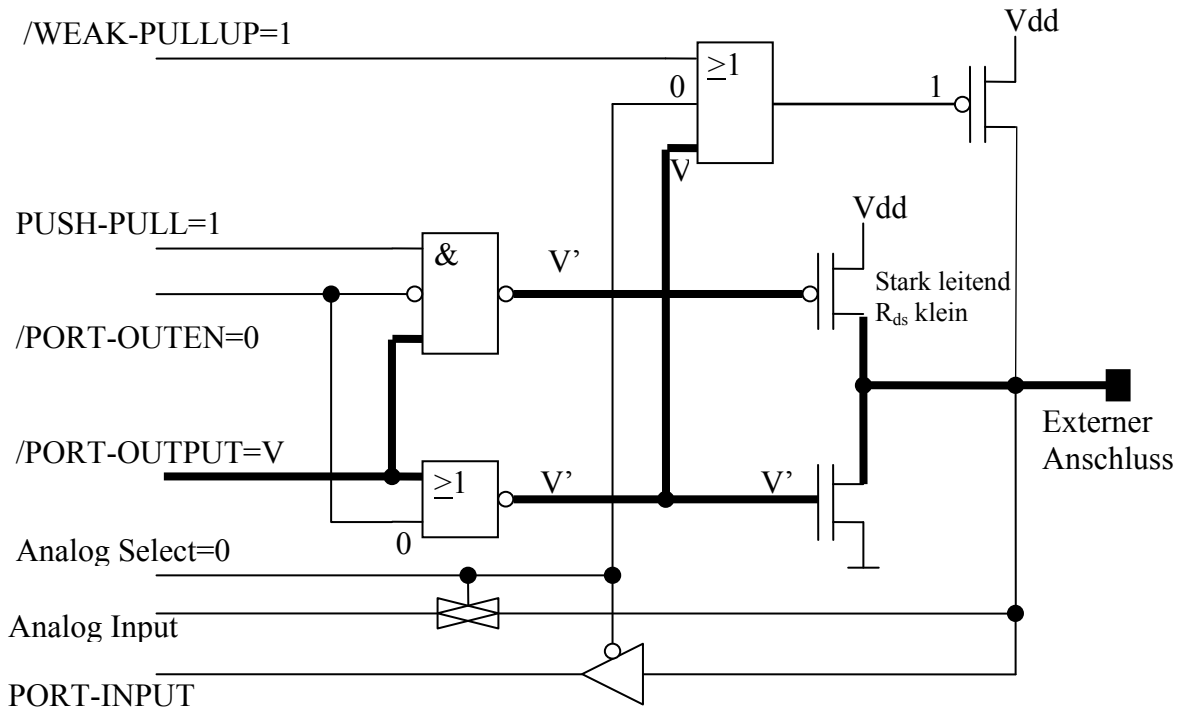
WEAK-PULLUP wird global im SFR-Register XBR1 mit dem Bit WEAKPUD gesetzt.
Bei WEAKPUD=0 ist die Pullup-Funktion eingeschaltet

Analog-Select wird im SFR-Register PnMDIN gesetzt.
Bei PnMDIN.z=0 wird der Eingang als analoger Eingang gesehen. Analog_Select=not(PnMDIN.z)

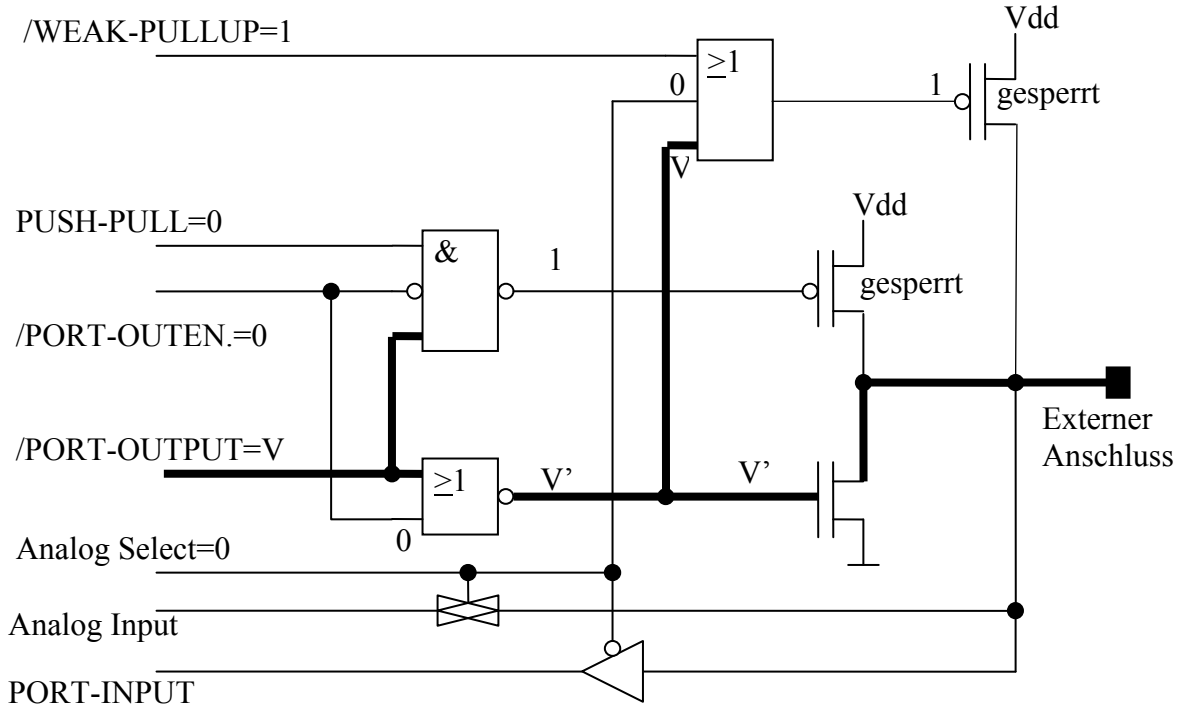
PORT-OUTENABLE wird im SFR-Register XBAR1 durch das Bit XBARE gesetzt

PUSH-PULL wird durch in PnMDOUT gesetzt.
Ist PnMDIN.z = 1 (Default Wert) dann ergibt sich
bei PnMDOUT.z=0 ein Open-Drain-Anschluss
bei PnMDOUT.z=1 ein Push-Pull-Anschluss

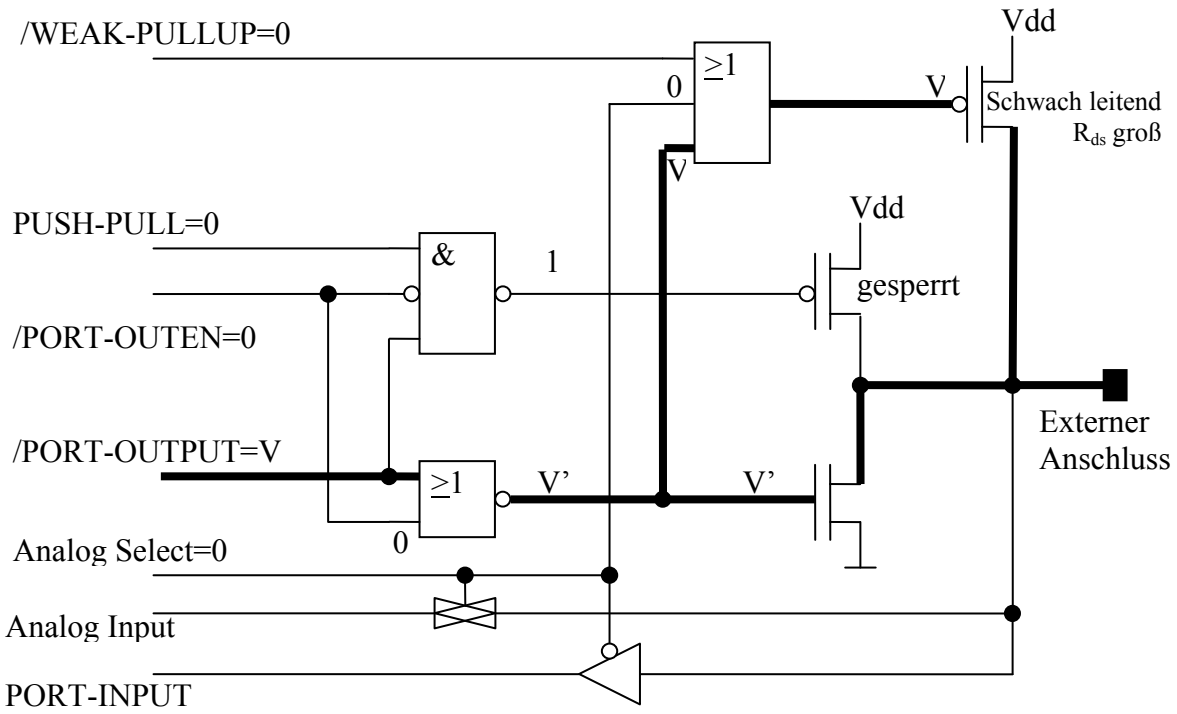
Ausgang als GPIO im Push-Pull Mode:



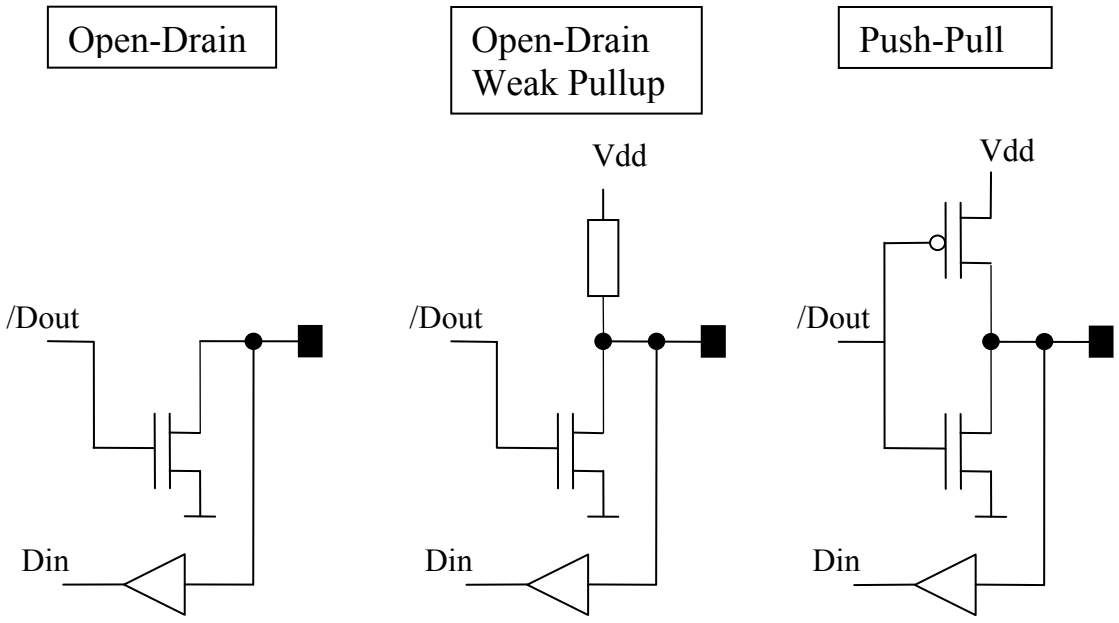
Ausgang als GPIO Open-Drain-Mode:



Ausgang als GPIO Open-Drain-Mode mit Weak Pullup:



Mögliche digitale Ausgangsstufen



P0 Datenspeicher

P0: Bitadressierbar, Resetwert: FFh SFR Adresse: 80h

P0.7	P0.6	P0.5	P0.4	P0.3	P0.2	P0.1	P0.0
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0

Die Werte werden ausgegeben, wenn der Flag XBARE im Crossbar Register 1 gesetzt wird

Crossbar Register 1:

XBR1: Byteadressierbar, Resetwert: 00h SFR Adresse: E2h

…	XBARE	…	…	…	…	…	…
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0

Port0 Input Mode:

P0MDIN: Byteadressierbar, Resetwert: FFh SFR Adresse: F1h

…	…	…	…	…	…	…	…
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0

0: P0.n ist als analoger Eingang konfiguriert.
1: P0.n ist nicht als analoger Eingang konfiguriert.

Port0 Output Mode

P0MDOUT: Byteadressierbar, Resetwert: 00h SFR Adresse: A4h

…	…	…	…	…	…	…	…
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0

Wenn P0MDIN P0.n = 1
0: Der zugehörige P0.n Pin ist ein Open-Drain-Anschluss
1: Der zugehörige P0.n Pin ist ein Push-Pull-Anschluss

Port0 Skip

P0SKIP: Byteadressierbar, Resetwert: 00h SFR Adresse: D4h

…	…	…	…	…	…	…	…
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0

0: Der zugehörige P0.n Pin wird nicht durch die Crossbar ausgelassen.
1: Der zugehörige P0.n Pin wird durch die Crossbar ausgelassen.

Für die Vorlesung und das zugehörige Labor soll die bereits angegebene Pinbelegung zugrunde gelegt werden. Damit müssen der Anschluss des Quarzes von der Crossbar ausgenommen werden (P0SKIP). Zur Verwendung des AD-Wandlers werden die entsprechenden Anschlüsse als analoge Eingänge gekennzeichnet (P1MDIN).
Die Ports P2, P3 werden als Push-Pull-Ausgänge betrieben, Port 4 im Open-Drain-Mode.

Pinbelegung des C8051F340 für die Vorlesung und das Labor

Pin	Port	Verwendung im Labor
6	P0.0	Int0 – Taster 9
5	P0.1	Int1 – Taster 10
4	P0.2	I²C – SDA
3	P0.3	I²C – SCL
2	P0.4	UART – TX
1	P0.5	UART – RX
48	P0.6	NC
47	P0.7	XTAL – externer Oszillator
46	P1.0	GPIO
45	P1.1	ADC – PB4RT_1 oder POTI (über JP1 auswählbar)
44	P1.2	ADC – PB4RT_2
43	P1.3	GPIO
42	P1.4	GPIO
41	P1.5	VREF
40	P1.6	GPIO
39	P1.7	GPIO
38	P2.0	Enable 7-Segment #6
37	P2.1	Enable 7-Segment #5
36	P2.2	Enable 7-Segment #4
35	P2.3	Enable 7-Segment #3
34	P2.4	Enable 7-Segment #2
33	P2.5	Enable 7-Segment #1
32	P2.6	UART – RTS
31	P2.7	UART – CTS
30	P3.0	7-Segment – A
29	P3.1	7-Segment – B
28	P3.2	7-Segment – C
27	P3.3	7-Segment – D
26	P3.4	7-Segment – E
25	P3.5	7-Segment – F
24	P3.6	7-Segment – G
23	P3.7	7-Segment – DP

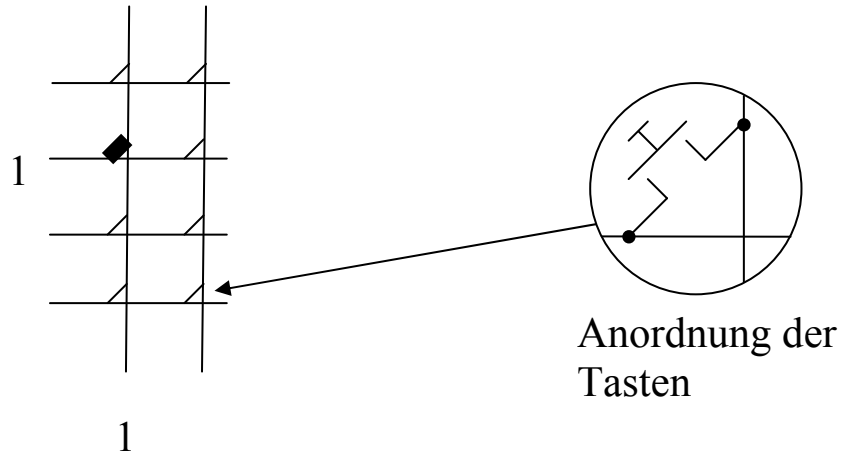
Pin	Port	Verwendung im Labor
22	P4.0	Taster 4
21	P4.1	Taster 3
20	P4.2	Taster 2
19	P4.3	Taster 1
18	P4.4	LED 4
17	P4.5	LED 3
16	P4.6	LED 2
15	P4.7	LED 1
7		GND
8		USB – D+
9		USB – D-
10		VDD (3,3V)
11		REGIN (On-Chip-Voltage-Regulator)
12		USB – VSENSE
13		Reset
14		Debug Interface

```
void Port_IO_Init(){
   //Pin P1.1,2 sind als analoge
   //Eingänge geschaltet
   P1MDIN    = 0xF9;          //1111 1001

   //P2, P3 sind als Push-Pull-Ausgänge geschaltet
   //P4 ist im Open Drain Mode
   P2MDOUT   = 0xFF;          //1111 1111
   P3MDOUT   = 0xFF;          //1111 1111

   //Quarzanschlüsse P0.7, P0.6 werden von der
   // Verteilung in der Crossbar ausgenommen
   P0SKIP    = 0xC0;          //1100 0000

   //Die AD-Wandleranschlüsse werden von der
   //Crossbar ausgenommen
   P1SKIP    = 0x06;          //0000 0110

   //Die Portdaten werden an die Portpins über
   //die Crossbar weitergeleitet XBARE=1
   XBR1      = 0x40;          //0100 0000
   //Bit0: Enable UART0 on Crossbar
   //Bit2: Enable SMBus
   XBR0      = 0x05;          // 0000 0101
}
```

5.3 Matrixtastatur

Zur Abfrage von Schalterstellungen, die an den Ports angeschlossen sind, ist eine Lösung relativ einfach zu finden, wenn wenige Schalter oder Taster angeschlossen sind. Die Schalter werden dann direkt ggf. über einen zusätzlichen Widerstand an die Pins angeschlossen. Die Abfrage kann dann entweder zyklisch erfolgen oder, wenn die Schalter an die Interrupteingänge angeschlossen sind, über die Bearbeitung der Signale in den entsprechenden Interruptservicefunktionen. Die Bearbeitung von mehr als 8 Tasten oder gar von Tastaturblöcken würde auf diese Weise zu aufwendig sein.
Wird eine größere Anzahl von Tasten beim Aufbau eines Mikrocontrollersystems benötigt, so ist eine Anordnung der Tasten in Matrixform mit n-Zeilen und m-Spalten üblich.

Bsp.: n = 4, m = 2



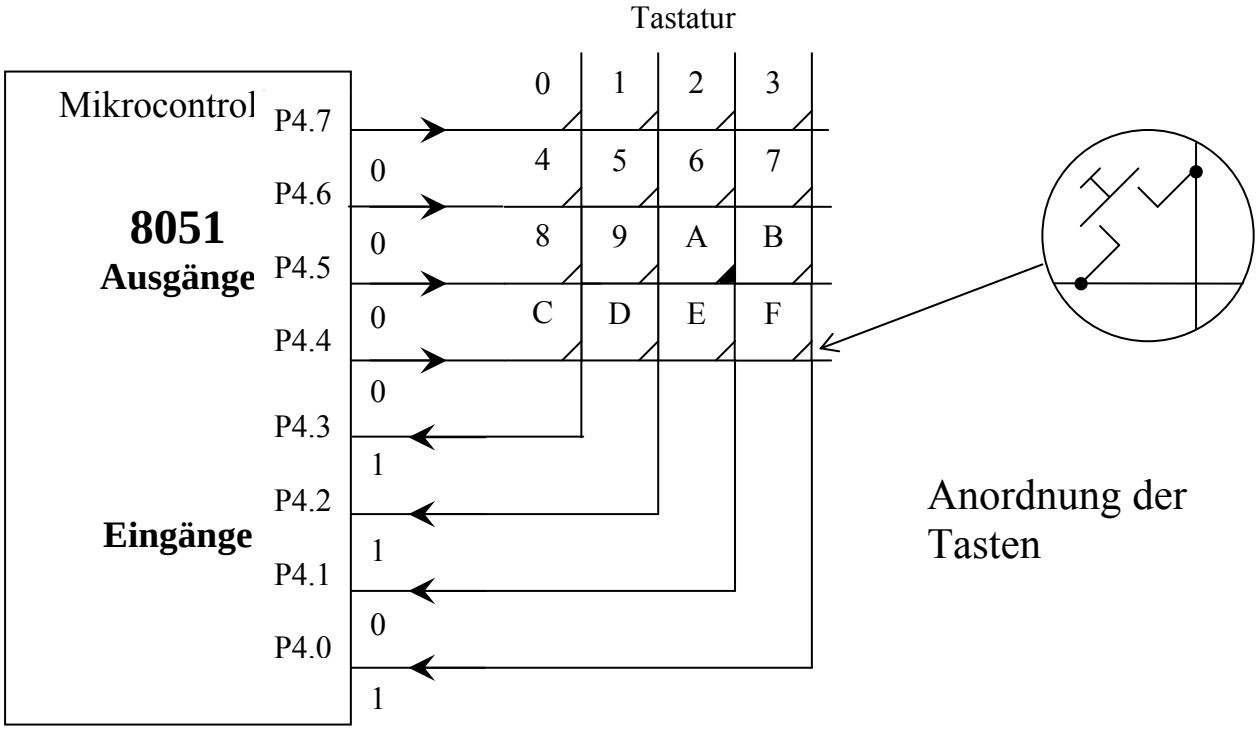
Ein einfaches Verfahren zur Erkennung eines Tastendruckes kann damit folgendermaßen realisiert werden:

- Es werden nacheinander die Zeilen auf 1 gelegt und an den Spalten nachsehen, ob eine ‚1' erscheint. ⇒ Abtasttechnik (Scanning)

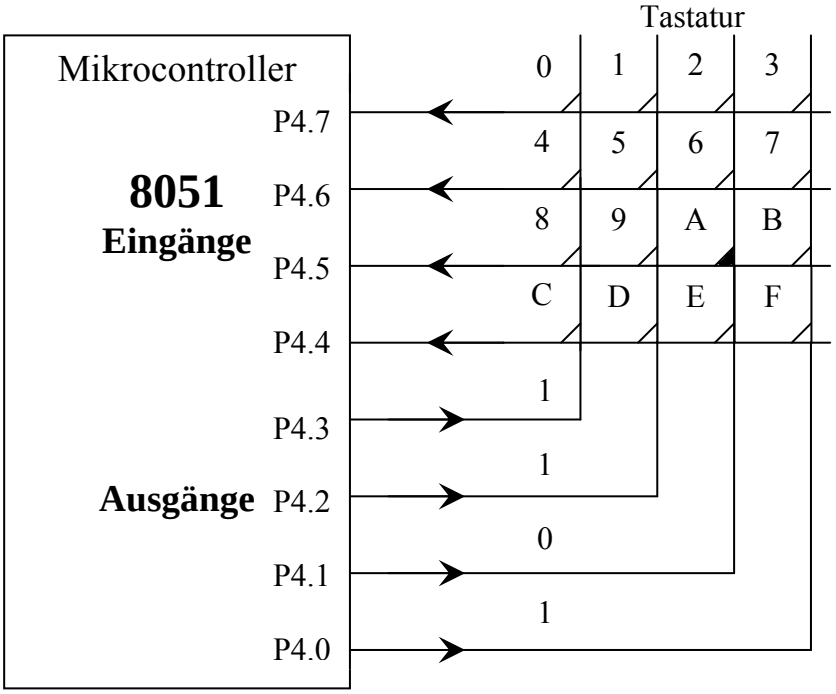
- Ist eine 1 gefunden, so wird wegen möglicher Prellvorgänge 5..20 ms getestet, ob der Zustand andauert. Ist ein dauerhafter Zustand erreicht, wird der Code für die Taste gebildet.

Ein verbessertes Verfahren zur Ermittlung einer gedrückten Taste, ist die Tastaturabfrage mit der Umkehrung der Abfragerichtung.

Abtastung mit Richtungsumkehrung



1. Schritt
   bei gedrückter Taste " A " Ausgabe von 00001111



2. Schritt
   Informationen an den unteren Bits nochmals ausgeben

Tastenabfrage mit Umkehrung der Abfragerichtung.

Aufteilung der Portleitungen in 4 Ausgabe- und 4 Eingabeleitungen.

- Für die Eingabeleitungen muss eine 1 ausgegeben werden.
  (Vorbereitung des Einlesens)
- Die Ausgabeleitungen müssen alle auf 0 gesetzt werden.

1 Schritt: Bitmuster 0000 1111 ausgeben. Ist eine Taste gedrückt, wird eine 0 auf einer Spaltenleitung auftreten.
Es ist nun bekannt,

- dass eine Taste gedrückt wurde
- und zu welcher Spalte die Taste gehört.

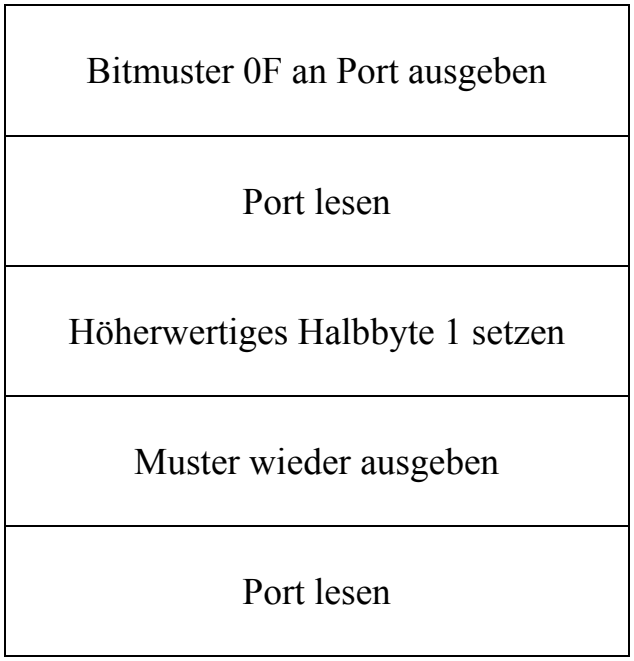
2. Schritt: Ausgabe der gerade empfangenen Information auf derselben Leitung. Die Zeilen werden als Eingabeleitungen geschaltet. Nur an der Stelle, an der sich der geschlossene Schalter befindet, wird die ‘0‘ an die Zeilenleitung gelegt.

Inhalt des Portregisters:

- Die höherwertigen Bits enthalten die Information über die Zeile (Position der 0)
- Die niederwertigen Bits enthalten die Information über die Spalte (Position der 0)

Realisierung des Verfahrens:

Struktogramm:



Unterprogramm Tast:

```
Tast:   MOV P4, #0Fh        ; Zeile auf 0, Spaltenpins sind Eingänge
        MOV A, P4           ; Spalte lesen
        ORL  A,#0F0h        ; obere Bits zum Lesen vorbereiten
        MOV P4, A           ; Zeilenpins sind Eingänge
                            ; Spalte ausgeben
        ; jetzt steht bereits das 8Bit Codewort für die gedrückte Taste fest
        MOV R6, P4          ; Rückgabespeicherplatz
        RET
```

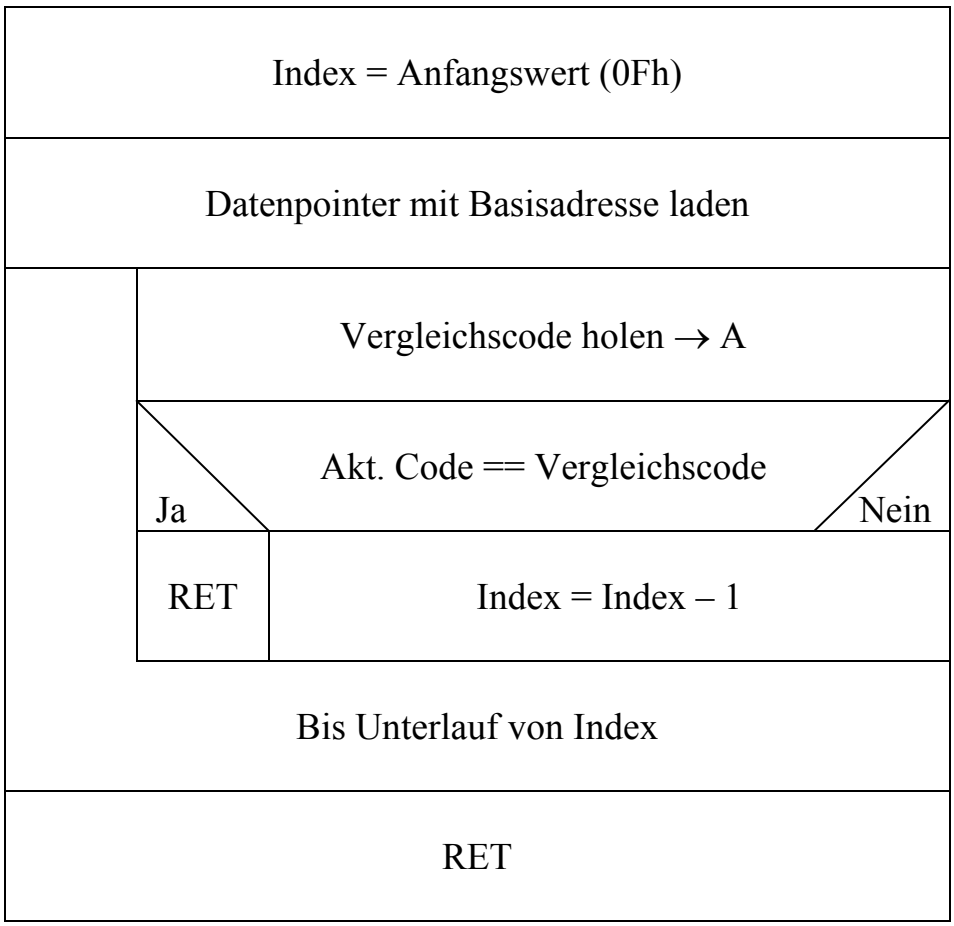
Zusätzliche Dekodierung

z.B. für Taste 0 = 0111 0111 soll Code 00h entsprechen
für Taste 1 = 0111 1011 soll Code 01h entsprechen

Annahme:
Die Codes für ‘0‘, ‘1‘ stehen ab Adresse „Code1“ in aufsteigender Reihenfolge im Programmspeicher.

```
CODE1:  DB 01110111b        ; Tastencode für ‚0‘
        DB 01111011b        ; Tastencode für ‚1‘
        DB 01111101b        ; Tastencode für ‚2‘
              :
        DB 0EEh             ; Tastencode für ‚F‘
```

Struktogramm zur Tastenumkodierung




```
PCode:    MOV R7, #0Fh              ; höchste Tastennummer laden
          MOV DPH, High (CODE1)     ; MSB der Adresse
          MOV DPL, Low (CODE1)      ; LSB der Adresse

NTaste:   MOV A, R7                 ; Nummer ist Index
          MOVC A, @A+DPTR           ; indizierte Adressierung, Inhalt von
                                    ; A ist der Bit-Code der Taste F
          CJNE A, P4, Ungleich      ; Vergleichen, jetzt ist die Nummer
                                    ; der gedrückten Taste in R7
          RET

Ungleich: DEC R7                    ; nächste niedrige Taste
          CJNE R7, #0FFh, NTaste    ; Überlauf hat stattgefunden
          RET                       ; FF in R7 zeigt keine
                                    ; gültige Taste an
```

5.4 Entprellung von Tasten

Entstehung des Prellens:

Mechanisch elastischer Aufprall der Kontaktflächen.

→ Rascher Wechsel des Schaltzustandes für die Dauer von ca. 3.....20 ms.

Siehe Bild Signalverlauf beim Drücken eines Tasters.

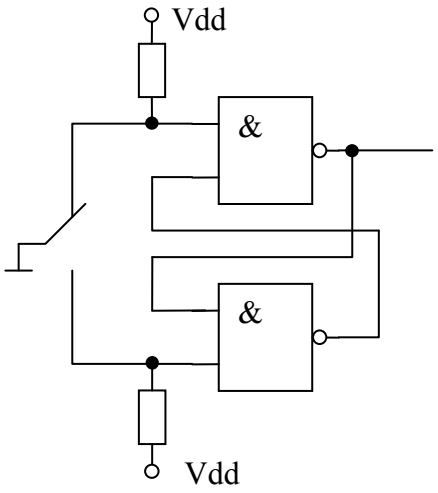
Der Vorgang findet sowohl beim Schließen als auch beim Öffnen statt.

Möglichkeiten der Entprellung:

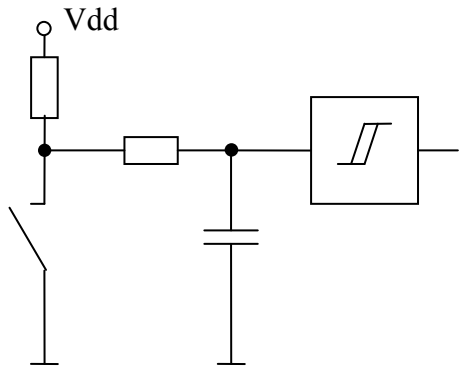
a) Nachschaltung eines RC – Tiefpasses (Aufwendig bei vielen Tasten)

b) Einsatz eines RS - Kippgliedes
(Noch aufwendiger, da Wechselschalter + 2 Gatter benötigt werden)

c) Entprellen durch Software
1. Zeitraum abwarten und danach den Zustand der Taste nochmals überprüfen.
2. Mehrmaliges Abfragen zum Feststellen eines statischen Zustandes.



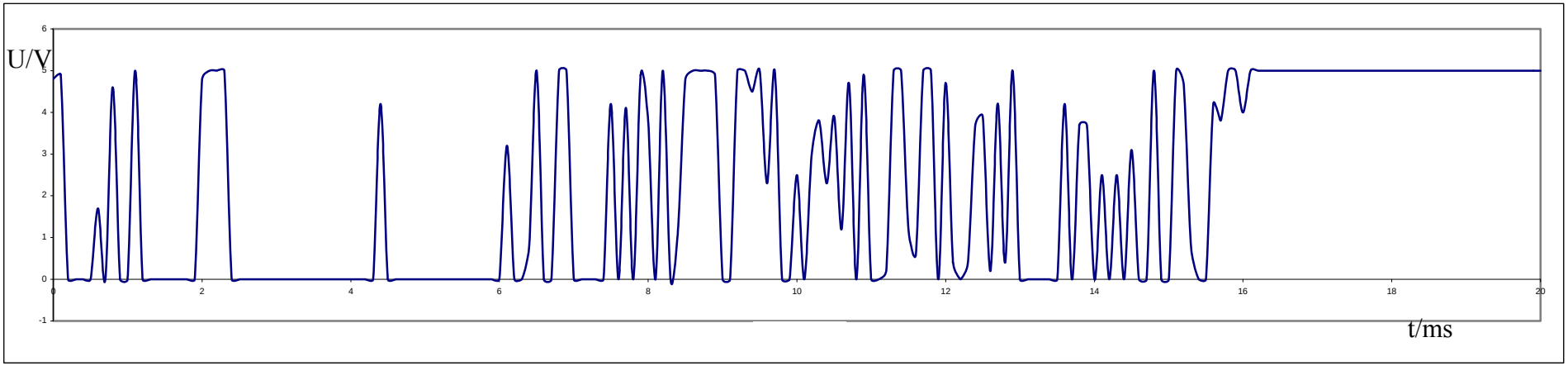
Entprellung mit RS_Flip-Flop



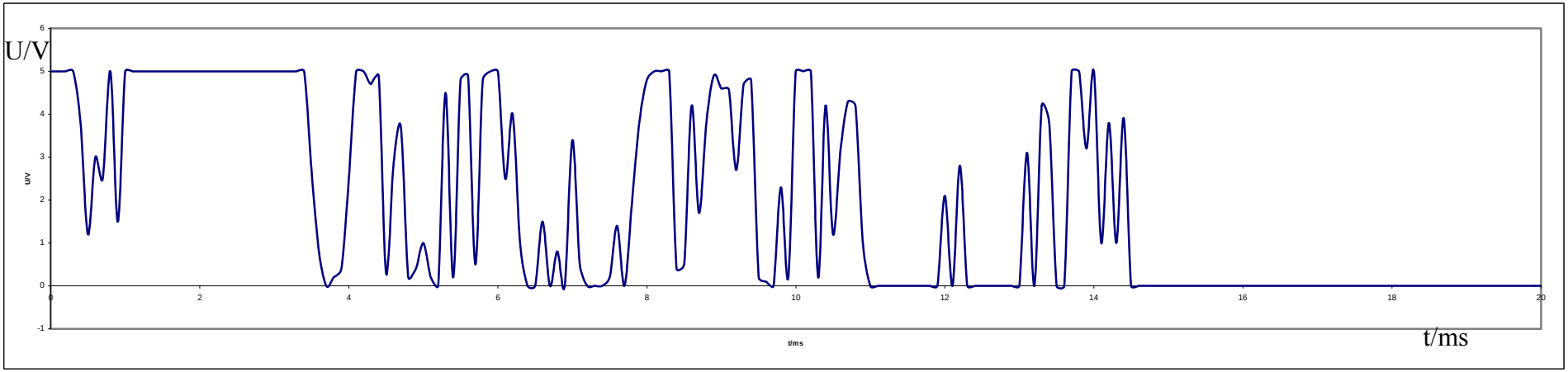
Entprellung mit RC-Glied und Schmitttrigger

Spannungsverlauf beim Drücken und Loslassen einer Taste

Schliessen der Taste

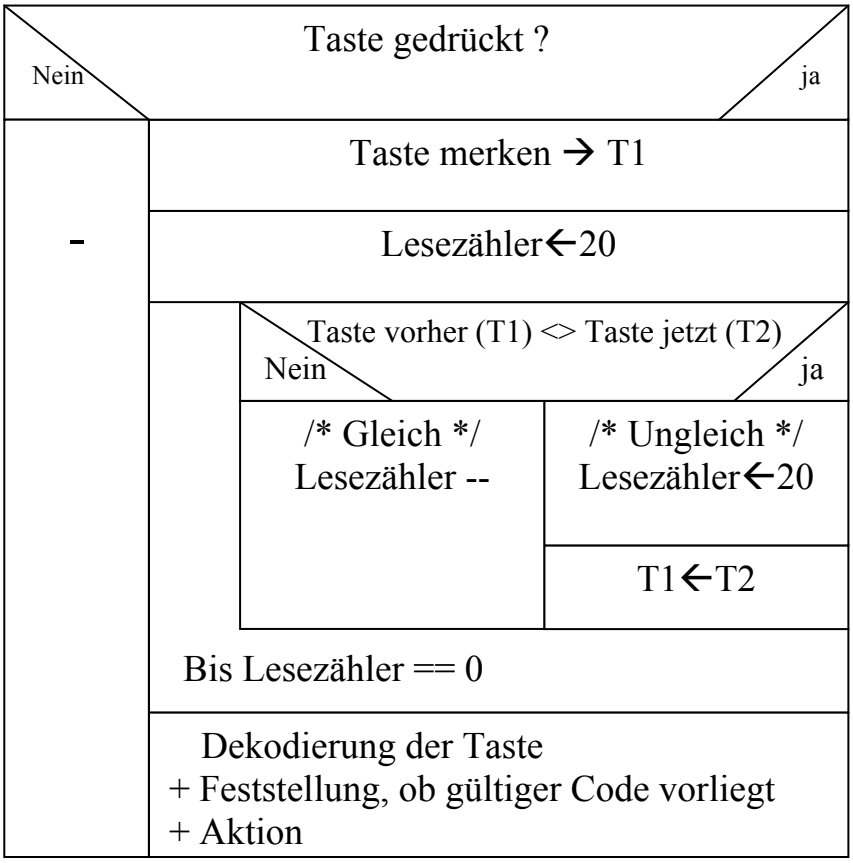


Öffnen der Taste



Abfrage auf den statischen Zustand einer Taste.

Struktogramm:



Programm:

```
Taste:  MOV   P1, #00001111b   ; Zeilen auf 0
        MOV   A, P1            ; Spalten lesen
        ORL   A, #11110000b    ; Spalten auf 0
        MOV   P1, A            ; Codewort in P1
        MOV   A, P1            ; Codewort lesen
        CJNE  A, #0FFh, Merke  ; ist eine Taste gedrückt
        JMP   Exit             ;nein

Merke:  MOV   A, P1            ;ja Tastencode merken
        MOV   R2, #20          ;Lesezähler = 20
Loop:   CJNE  A, P1, Merke     ;Taste ungleich
        DJNZ  R2, Loop         ;20 mal wiederholen
        ;ggf. weitere Aktionen
Exit:   RET
```

6 Serielle Datenübertragung

6.1 Grundlagen

Serielle Datenübertragungen haben den Vorteil, dass nur wenige Leitungen zum Übertragen von Informationen notwendig sind. Das spielt insbesondere dann eine Rolle, wenn weite Entfernungen überbrückt werden müssen. Serielle Übertragungstechniken wurden schon lange vor der Computertechnik (z.B. im Fernschreibebetrieb) verwendet. Sollen sich Geräte von verschiedenen Herstellern verstehen, die an unterschiedlichen Standorten betrieben werden, so ist eine Normung der Signale notwendig z.B.:

DIN 66020
EIA RS232-C (USA)
CCITT V24

Abkürzungen:

- DIN Deutsche Industrie Norm
- EIA Electronic Industries Association
- CCITT Comite Consultatif International Telegraphic et Telephonique

Geräte zum Senden und Empfangen von Daten heißen:

Datenendeinrichtungen DEE

Kopplung: - bei kurzen Entfernungen können die Teilnehmer direkt über Kabel verbunden werden
- bei größeren Entfernungen werden zusätzlich Modulationsverfahren verwendet.

Bei der Modulation und Demodulation in **Datenübertragungseinrichtungen (DÜE)** spricht man von Modems (MOdulator/DEModulator)

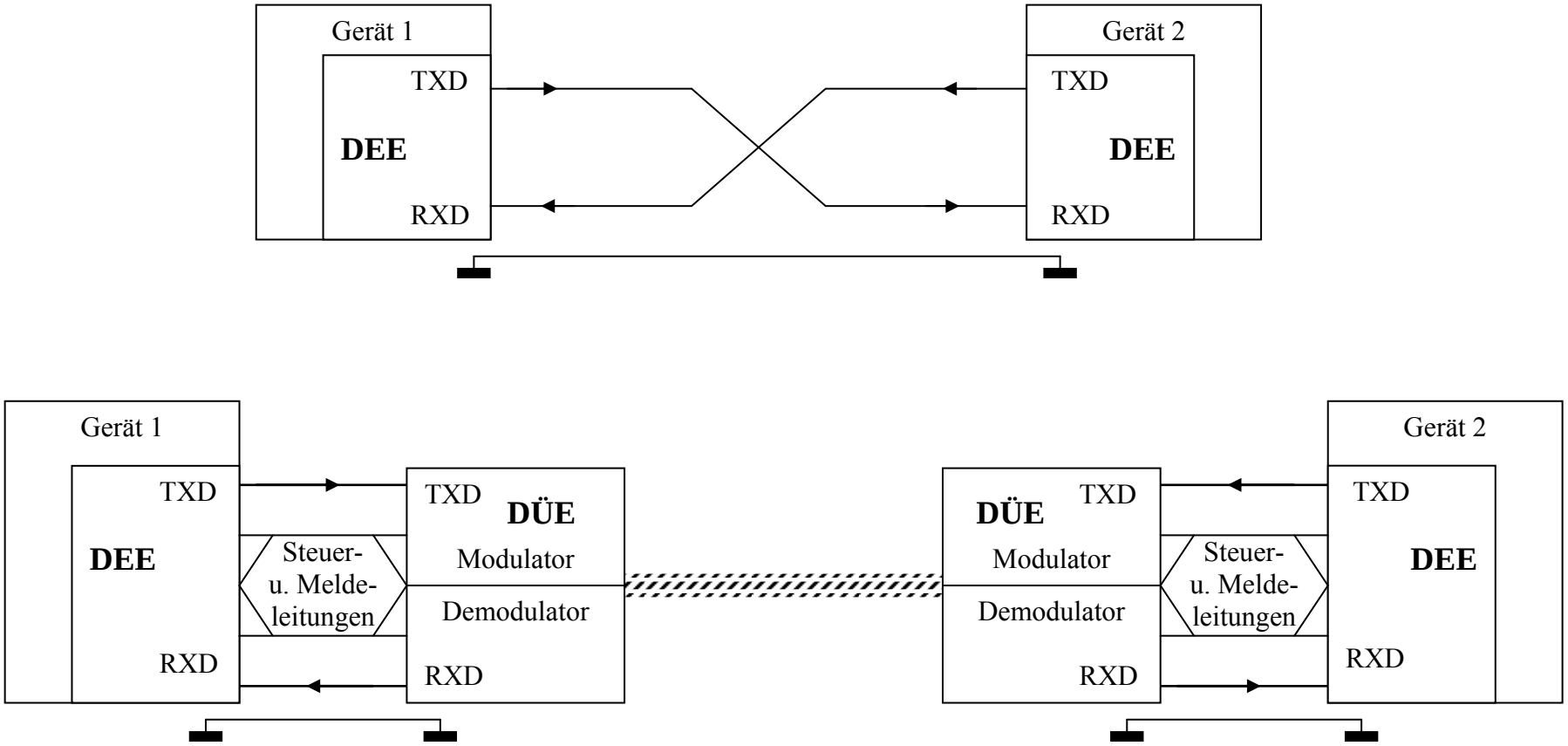
Englisch:

Datenendeinrichtung: **DTE** (Data Terminal Equipment)
Datenübertragungseinrichtung: **DCE** (Data Communication Equipment)

Betriebsarten:

Vollduplexbetrieb: Datenaustausch in beide Richtungen gleichzeitig
Halbduplexbetrieb: Der Kanal wird abwechselnd zum Senden und Empfangen verwendet
Simplexbetrieb: Nur eine Datenübertragungsrichtung ist möglich z.B. Belegdrucker

Datenübertragung bei Fernverbindungen über MODEMs



Wirkungsweise der seriellen Schnittstelle

Parallel vorliegende Daten z. B. 8 Bit im ASCII Code werden in eine serielle Form umgewandelt (Bitfolge). Die Bits werden nach einer Synchronisationsphase nacheinander übertragen.

Voraussetzung:
Sender und Empfänger arbeiten mit der gleichen Geschwindigkeit.

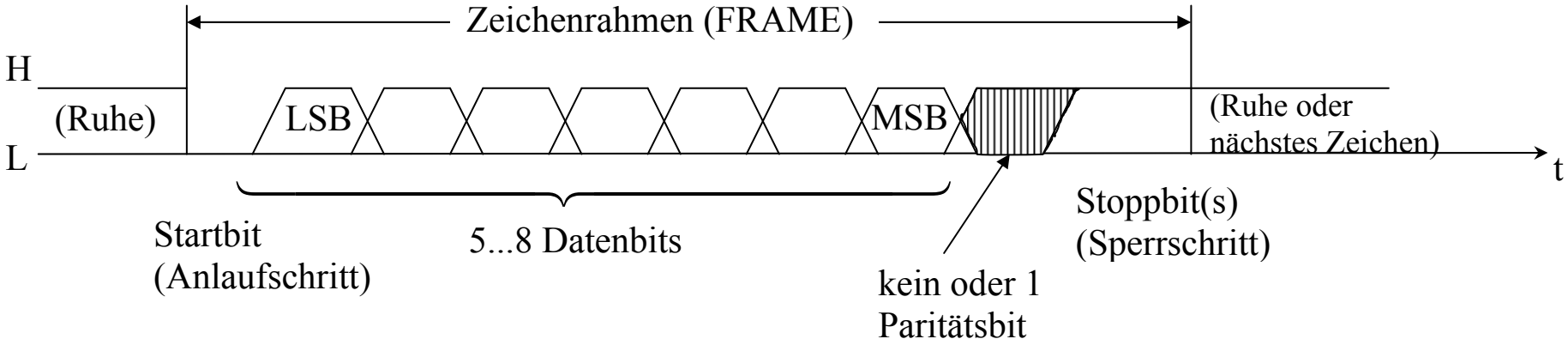
Unterscheidung hinsichtlich der Synchronisation:

- Asynchrone Übertragung (kurze Datensätze) mit Zusatzinformationen am Anfang und Ende der Datensätze zur Synchronisation.
- Synchrone Datenübertragungen erlauben lange Datensätze mit hohen Geschwindigkeiten. Spezielle Codes zur Bestimmung des Taktes werden verwendet.

Aufbau einer asynchronen Übertragung:

1. Startbit (L-Pegel)
2. Datenbits (ca. 8 Bits)
3. Ein oder zwei Stoppbits (H-Pegel)

Die fallende Flanke des Startbits bewirkt die Synchronisation.
Die Stoppbits dienen zur Überprüfung, ob die Synchronisation erfolgreich war, d.h. die Anzahl der vereinbarten Bits wurde empfangen und danach liegt wieder 1-Signal an.



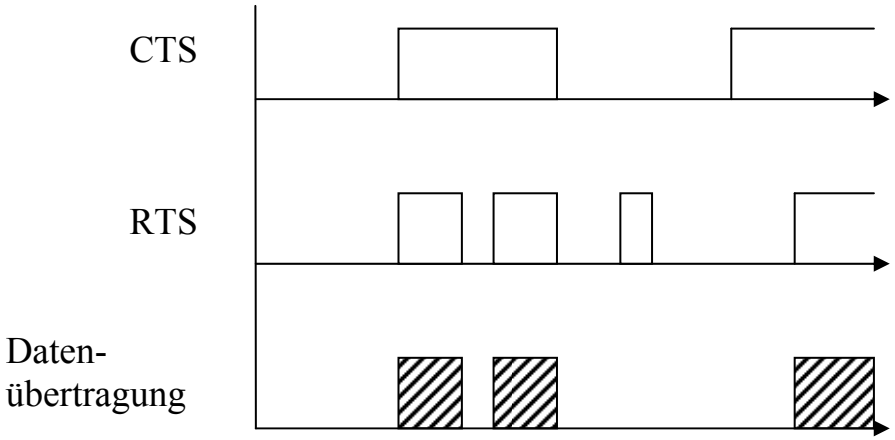
Asynchrones Datenformat (TTL-Pegel)

Das gezeigte Datensignal wird auf einer Leitung mit dem Namen TXD gesendet. (Transmit Data)
Der entsprechende Empfängeranschluss heißt RXD (Receive Data)
Beim Duplexbetrieb werden diese Leitungen doppelt ausgeführt.
Zusätzlich sorgen Steuerleitungen dafür, dass der Datenaustausch keine Überlaufsituationen produziert.
→ Steuerleitung des Datenaustausches mittels Hardware Handshake,
→ RTS/CTS Protokoll.

Bedeutung:

CTS = Clear to send,
Der Sender teilt dem Empfänger mit, dass er weitere Daten zum Senden bereit hält.

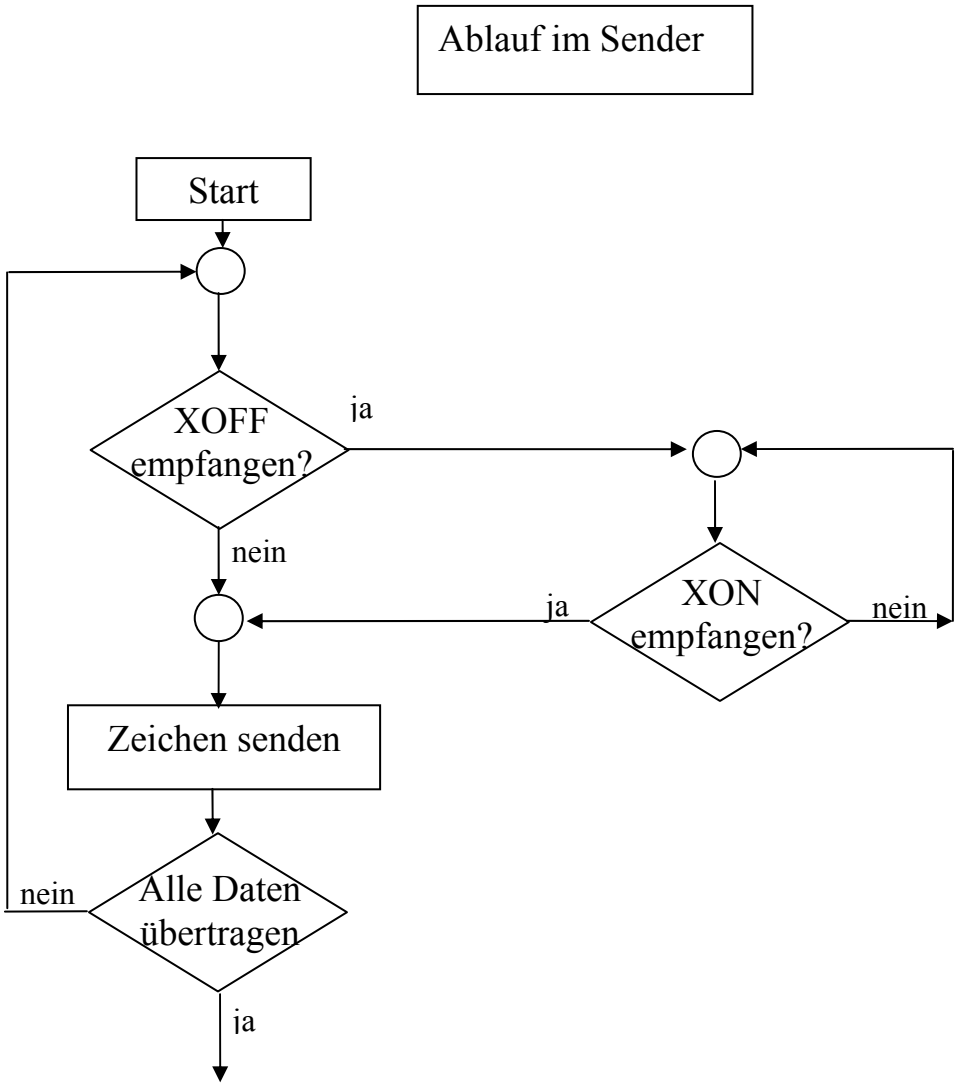
RTS = Request to send
Der Empfänger ist bereit neue Daten zu empfangen. Er fordert über RTS den Sender auf die Übertragung zu beginnen bzw. fortzusetzen.



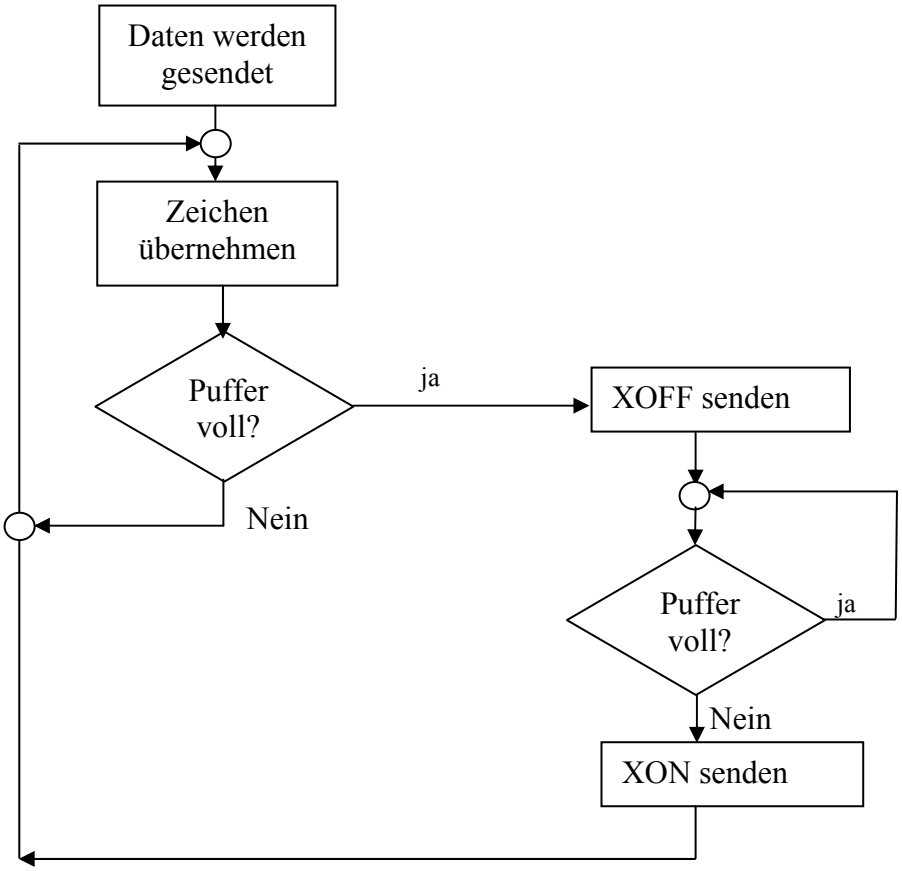
XON/XOFF- Protokoll:

Das XON/XOFF- Protokoll ist ein Software Protokoll zur Steuerung der Datenübertragung. Das Gerät, das die Daten empfängt, sendet ein Besetztzeichen, wenn es keine Daten mehr aufnehmen kann. Ist das Gerät wieder bereit, wird ein Freizeichen gesendet.

Definition der Steuerzeichen:
XON 11h XOFF 13h
Vorteil: Es werden nur 2 Leitungen + GND benötigt.
Nachteil: Findet zwischen jedem zu übertragenen Zeichen ein Handshake statt, sinkt die Übertragungsrate auf ein Drittel.



Ablauf im Empfänger



6.2 Serielle Schnittstellen des Prozessors C8051F340

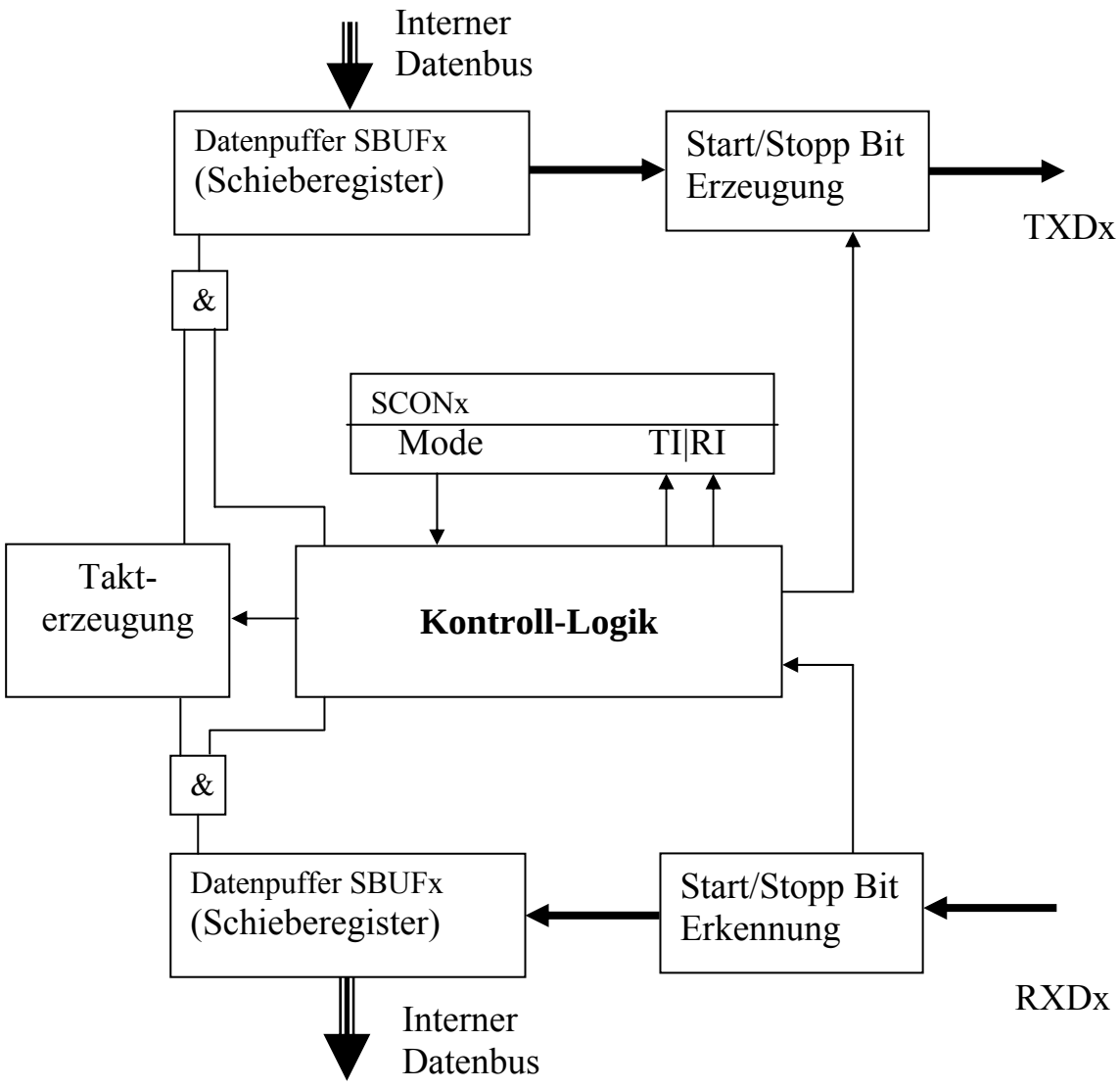
Im verwendeten Prozessortyp C8051F340 stehen zwei serielle Schnittstellen zur Verfügung, die in zwei asynchronen Modi mit variablen Übertragungsgeschwindigkeiten betrieben werden können. Die ursprüngliche Version des 8051 sieht hier noch einen synchronen Übertragungsmodus und eine Möglichkeit vor, mit einer festen Übertragungsgeschwindigkeit asynchron Daten zu übertragen.
Hier wurden schnelle Datenübertragungen angedacht, die inzwischen meist von anderen Verfahren abgelöst wurden (z.B. I2C oder SPI Bus)

Zum Betrieb der Schnittstellen sind mehrere Register im Prozessor vorgesehen.
Zur Einstellung der Parameter der Schnittstelle, insbesondere der gewünschten Übertragungsgeschwindigkeit, werden die im nächsten Abschnitt behandelten Timer mit benutzt. Zunächst soll nur auf die Datenpuffer SBUFx und die Steuerregister SCONx eingegangen werden. Die Datenpuffer werden wie normale Register beschrieben. Der Schreibvorgang in den Datenpuffer löst dann den Ausgabevorgang aus. Der Puffer wird als Schieberegister betrieben und so sind die Bits nacheinander am Ausgang sichtbar. Das Einfügen der Start- und Stoppbits übernimmt eine Kontroll-Logik.

Das Register SCONx enthält zum einen Steuerinformationen, die der Kontrolllogik mitteilen, in welchem Mode die serielle Schnittstelle betrieben werden soll und zum anderen aber auch Kontrollbits, die anzeigen, ob ein Sendevorgang abgeschlossen ist (TI) oder ob ein vollständiges Byte mit seinen Zusatzbits angekommen ist (RI). Die Flags RI und TI dienen auch zum Auslösen der Interrupt Service Routinen für die seriellen Schnittstellen.

Der Ausgabepuffer und der Eingabepuffer werden mit dem gleichen Namen angesprochen, sind jedoch physikalisch als zwei selbstständige Register realisiert. Die Aufgaben der Steuerbits sind in der Beschreibung des zugehörigen Special Function Registers auf den nächsten Seiten zu finden. Das darauf folgende ausführliche Blockschaltbild der seriellen Schnittstelle als peripherer Baustein auf dem Prozessorchip zeigt die Komplexität der Baugruppe.

Vereinfachtes Blockschaltbild der seriellen Schnittstelle



Beschreibung der seriellen Schnittstellen:

Die seriellen Schnittstellen UART0, UART1 sind zwar in ihren Grundfunktionen symmetrisch ausgeführt, der UART1 lässt jedoch weitere Bitbreiten zu und besitzt zusätzliche Pufferspeicher für den Empfang von Daten. Zusätzlich ist ein spezieller Baudrategenerator vorgesehen, der unabhängig von den existierenden Timern eine Taktgenerierung vornimmt. Zum detaillierten Studium wird auf das Datenblatt verwiesen. Im Brennpunkt soll die serielle Schnittstelle 0 (UART0) stehen.

Die zur Verfügung stehenden Betriebsarten beim UART0 unterscheiden sich durch die Anzahl der übertragenen Bits:

Mode 0: 8 Bit UART, Variable Baudrate

Asynchroner Sende und Empfangsmodus. Übertragung von 10 Bits.
1 Bit = Start Bit (0)
8 Bits = Datenbits (LSB zuerst)
1 Bit = Stoppbit (1)

Beim Empfang werden die 8 Datenbits in SBUF0 und das Stoppbit im SFR-Register SCON0 im RB80 Bit abgelegt.

Mode 1: 9-Bit UART, Variable Baudrate

Übertragung von 11 Bits
1 Start-Bit (0)
8 Datenbits (LSB-zuerst)
1 Programmierbares Bit
1 Stopp-Bit

Beim Senden wird das 9te Bit auf den Wert TB80 im SFR SCON0 gesetzt z. B. kann hier das Parity Bit aus dem PSW eingesetzt werden.
Beim Empfang wird das 9te Bit nach RB80 im SFR/SCON0 gespeichert. Das Stopp-Bit wird ignoriert.

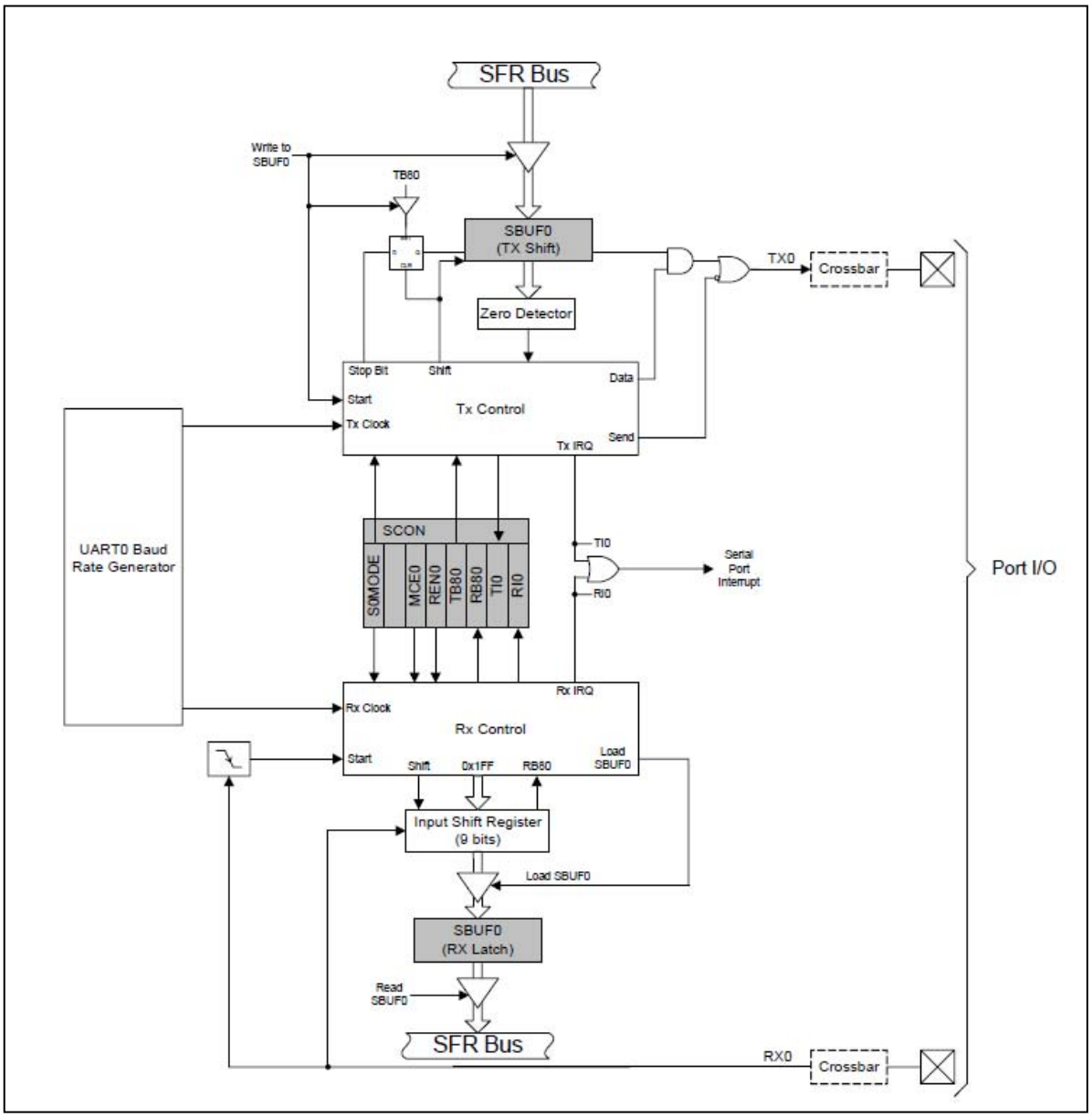
Steuerregister der seriellen Schnittstelle 0

Special Function Register SCON0 (Address 98_H) Reset Value: 00_H
Special Function Register SBUF0 (Address 99_H) Reset Value: XX_H

	MSB							LSB	
No.	$9F_H$	$9E_H$	$9D_H$	$9C_H$	$9B_H$	$9A_H$	99_H	98_H	
98_H	S0MODE		MCE0	REN0	TB80	RB80	TI0	RI0	SCON0

	7	6	5	4	3	2	1	0	
99_H	Serial Interface 0 Buffer Register								SBUF0

Bit	Function
S0Mode	Serial port 0 mode selection bit 0: 8-bit UART, variable baud rate 1: 9-bit UART, variable baud rate
MCE0	Enable serial port 0 multiprocessor communication in modes 2 and 3 In mode2 or 3, if SM20 is set to 1 then RI1 will not be activated if the receives 9^{th} data bit (RB80) is 0. In mode 1, if SM20 = 1 then RI0 will not be activated if a valid stop bit was not received. In mode 0, SM20 should be 0.
REN0	Serial port 0 receiver enable Enables serial reception. Set by software to enable serial reception. Cleared by software to disable serial reception.
TB80	Serial port 0 transmitter bit 9 TB80 is the 9^{th} data bit that will be transmitted in modes 2 and 3. Set or cleared by software as desired.
RB80	Serial port 0 receiver bit 9 In mode 2 and 3, RB80 is the 9^{th} data bit that was received. In mode 1, if SM2 = 0, RB80 is the stop bit that was received. In mode 0, RB80 is not used.
TI0	Serial port 0 transmitter interrupt flag TI0 is set by hardware at the end of the 8^{th} bit timer in mode 0, or at the beginning of the stop bit in the other modes, in any serial transmission. TI0 must be cleared by software.
RI0	Serial port 0 receiver interrupt flag RI0 is set by hardware at the end ot the 8^{th} bit time in mode 0, or halfway through the stop bit time in the other modes, in any serial reception (exception see SM20). RI0 must be cleared by software.



UART 0 Blockdiagramm

Zur Bestimmung der Übertragungsgeschwindigkeit wird der Timer1 im Autoreloadmodus verwendet. Der Interrupt ist hierbei normalerweise abgeschaltet.
Der Timer 1 kann durch mehrere Taktquellen versorgt werden. Zur Bestimmung der Übertragungsrate dient dabei folgende Gleichung:

$$Baudrate = \frac{T1clk}{2*(256-T1H)}$$

Die folgende Tabelle ermöglicht das Ablesen der notwendigen Nachladewerte und Taktquellen für die Standardübertragungsraten.

	Target Baud Rate (bps)	Actual Baud Rate (bps)	Baud Rate Error	Oscillator Divide Factor	Timer Clock Source	SCA1-SCA0 (prescale select)*	T1M*	Timer 0 Reload Value (hex)
SYSCLK = 12 MHz	230400	230769	0.16%	52	SYSCLK	XX	1	0xE6
	115200	115385	0.16%	104	SYSCLK	XX	1	0xCC
	57600	57692	0.16%	208	SYSCLK	XX	1	0x98
	28800	28846	0.16%	416	SYSCLK	XX	1	0x30
	14400	14423	0.16%	832	SYSCLK/ 4	01	0	0x98
	9600	9615	0.16%	1248	SYSCLK / 4	01	0	0x64
	2400	2404	0.16%	4992	SYSCLK/ 12	00	0	0x30
	1200	1202	0.16%	9984	SYSCLK/ 48	10	0	0x98
SYSCLK = 24 MHz	230400	230769	0.16%	104	SYSCLK	XX	1	0xCC
	115200	115385	0.16%	208	SYSCLK	XX	1	0x98
	57600	57692	0.16%	416	SYSCLK	XX	1	0x30
	28800	28846	0.16%	832	SYSCLK / 4	01	0	0x98
	14400	14423	0.16%	1664	SYSCLK / 4	01	0	0x30
	9600	9615	0.16%	2496	SYSCLK/ 12	00	0	0x98
	2400	2404	0.16%	9984	SYSCLK/ 48	10	0	0x98
	1200	1202	0.16%	19968	SYSCLK/ 48	10	0	0x30
SYSCLK=48 MHz	230400	230769	0.16%	208	SYSCLK	XX	1	0x98
	115200	115385	0.16%	416	SYSCLK	XX	1	0x30
	57600	57692	0.16%	832	SYSCLK / 4	01	0	0x98
	28800	28846	0.16%	1664	SYSCLK / 4	01	0	0x30
	14400	14388	0.08%	3336	SYSCLK/ 12	00	0	0x75
	9600	9615	0.16%	4992	SYSCLK/ 12	00	0	0x30
	2400	2404	0.16%	19968	SYSCLK/ 48	10	0	0x30

Timer 0 Ladewerte für Standardübertragungsraten

Der Betrieb einer seriellen Schnittstelle erfordert nicht nur die Einstellung der Betriebsparameter, sondern auch eine Abfolge von Aktionen beim Senden und Empfangen von Daten. Werden die Anzeigeflags RI und TI ständig abgefragt wird das Verfahren Polling genannt.

Sendeaktionen

1. Laden des Sendepuffers (SBUF0)
2. Warten bis der Sendevorgang beendet ist.
   TI zeigt durch den Wert 1 an, dass der Sendevorgang beendet ist.
3. Zurücksetzen von TI, damit ein neuer Sendevorgang gestartet werden kann.

Empfangsaktionen

1. Abfragen des RI Flags auf 1
   Damit wird angezeigt, dass ein Byte vollständig empfangen wurde.
2. RI zurücksetzen, damit ein neuer Empfangsvorgang stattfinden kann.

Programmrealisierung:

Bei der Verwendung der seriellen Schnittstelle 0 kann auf TI und RI direkt zugegriffen werden, da das Register SCON0 bitadressierbar ist.

Assembler

Sendevorgang

```
        …
        Mov  SBUF0, #Datum
Loop:   JNB  TI0, Loop          ;Warten bis der Sendevorgang beendet ist.
        CLR  TI0                ;Sendflag löschen
        …
```

Empfangsvorgang

```
        …
Loop:   JNB RI0, Loop     //Testen ob ein Empfangsvorgang beendet wurde
        CLR RI0           //Empfangflag löschen
        …
```

Ist das Konfigurationsregister nicht bitadressierbar, muss das entsprechende Flag ausmaskiert werden, bevor die entsprechende Abfrage erfolgen kann.

C-Programm

Sendvorgang

```
        …
        SBUF0=buffer;
        while(!(SCON0 & 0x02));      //Warteschleife
        SCON0=SCON0 & 0xFD;          //Sendflag löschen
        …
```

Empfangsvorgang

```
        …
        while(!(SCON0 & 0x01));      //Warteschleife
        SCON0=SCON0 & 0xFE;          //Empfangsflag löschen
        buffer=SBUF0
        …
```

Bei dieser Form der Abfragen wird die Abarbeitung durch die Warteschleifen bestimmt. Die Abfragen können jedoch auch so organisiert werden, dass sie innerhalb der normalerweise vorhandenen globalen Schleife durchgeführt werden und die weiteren Aufgaben noch erfüllt werden können.

Die Empfangsfunktion ist dabei unkritisch. Die entsprechende Abfrage auf das RI Flag führt nur dann zu einer Aktion wenn das RI Flag gesetzt ist. Der Wert im Empfangspuffer wird abgeholt und abgespeichert. Wird dazu eine Funktion realisiert, kann der Wert über die Parameterliste zurückgegeben werden und der Funktionsrückgabewert zur Kennzeichnung eines erfolgreichen Empfangsvorgangs genutzt werden. Das Rücksetzen des RI Flag kann in der Funktion direkt realisiert werden.

```
/*      Function SerIntReceive
*       Global:         SBUF0
*       Output:         cc received Byte
*       Return value:   0 no byte received
*                       1 byte was received
*/
UINT8 SerIntReceive(UINT8 *cc){
        UINT8 retval=0;
        if(SCON0 & 0x01){   //RI==1?
                //Byte was received
                retval=1;
                *cc             =SBUF0;             //get byte
                SCON0           =SCON0 & 0xFE;      //1111 1110 reset RI
        }
        return(retval);
}
```

Die Sendefunktion in der bisherigen Realisierung verlangt ein Warten, bis der Sendevorgang erfolgreich beendet wurde. Wird beim Durchlaufen einer Sendeanforderung überprüft, ob ein Sendevorgang erfolgreich abgelaufen ist, kann danach ein neuer Sendevorgang gestartet werden. Diese Vorgehensweise verhindert ein Warten auf die Beendigung des Sendevorgangs. Für das letzte Byte muss nur überprüft werden, ob der Vorgang erfolgreich war. Die Verhinderung eines neuen Sendevorgangs kann durch die Übergabe eines speziellen Zeichens initiiert werden.

```
/*      Function SerIntTransmit
*       Global:         SBUF0
*       Output:         cc Byte to be send
*                       or special character
*       Return value:   0 preceding byte was not sent
*                       1 preceding byte was sent
*/
UINT8 SerIntTransmit(UINT8 cc){
        UINT8 retval=0;
        if(SCON0 & 0x02){   //TI==1?
                //preceding Byte was sent
                retval  =1;                     //mark last transmit action was ok
                if(cc!='\n'){                   //special character to stop transmission
                                                //Can be any other useful character
                        SBUF0           =cc;                    //send byte
                        SCON0           =SCON0 & 0xFD;      //1111 1101 reset TI
                }
        }
        return(retval);
}
```

Im folgenden Beispiel sollen diese Funktionen eingesetzt werden um Daten aus einem Sendepuffer (transmitbuffer) zu senden und Daten in einem Empfangspuffer abzulegen.
Dazu werden Zähler eingeführt, welche die Anzahl der empfangenen Bytes (receivedBytes) und der bereits gesendeten Bytes protokollieren.
Variablen zur Behandlung der Rückgabecodes der Sende- und Empfangsfunktionen sowie zur Behandlung von Ausnahmefällen sind ebenfalls notwendig.

```
typedef unsigned char UINT8;

#define REBUFFERSIZE 10                    //Größe des Empfangspuffers
#define TRBUFFERSIZE 10                    //Größe des Sendepuffers


UINT8 receivebuffer[REBUFFERSIZE];   //Empfangspuffer
UINT8 receivedBytes=0;               //Anzahl der empfangenen Bytes

UINT8 transmitbuffer[TRBUFFERSIZE];  //Sendepuffer
UINT8 transmittedBytes=0;            //aktuell übertragene Bytes
UINT8 numTransmitBytes=0;            //Anzahl der zu sendenden Bytes

UINT8       codeoverflow=0;//Zustandvariable zur Steuerung des Empfangspufferüberlaufs

UINT8       codetransmit;  //Rückgabecode beim Senden
UINT8       codereceived;  //Rückgabecode beim Empfangen
UINT8       cc;            //Empfangenes oder gesendetes Byte
```

```
//Beispiel:
void main()
{

        while(1){       //Superloop
                //...
                //Aktion zum Laden der Sendedaten in den Sendepuffer
                //transmitbuffer[0]= ...;
                //...
                //transmitbuffer[4]= ...;
                transmittedBytes=0;             //Anzahl der übertragenen Bytes ist 0
                numTransmitBytes=5;                     //5 Bytes sollen übertragen werden
                SCON0         =SCON0 | 0x02;        //0000 0010  TI setzen
                                                //Nur beim allerersten Mal unbedingt notwendig
                                                //Startet die Funktion SerIntTransmit

                //...
                //Aktionen zum Empfangen über die serielle Schnittstelle
                //Empfangsüberprüfung
                codereceived=SerIntReceive(&cc);  //Überprüfung, ob ein Byte gesendet
                                                // wurde und Übernahme
                if(codereceived==1){
                        receivebuffer[receivedBytes]=cc;     //empfangenes Byte abspeichern
                        receivedBytes++;                     //Anzahl der empf. Bytes erhöhen
                        if(receivedBytes==REBUFFERSIZE-1)      //drohender Speicherüberl.
                                codeoverflow=1;
                }
                //Sende Bytes
                if(transmittedBytes<numTransmitBytes){
                        //Wenn das vorausgehende Byte gesendet wurde, wird das
                        // neue Byte in den Sendepuffer S1BUF eingetragen
                        codetransmit=SerIntTransmit(transmitbuffer[transmittedBytes]);
                        if(codetransmit==1){
                                //Das vorausgehende Byte wurde erfolgreich gesendet
                                transmittedBytes++;
                        }else{
                                //nur warten oder Fehler(1);
                        }
                }
                //Überprüfung, ob auch das letzte Byte gesendet wurde
                if(transmittedBytes==numTransmitBytes){
                        codetransmit=SerIntTransmit('\n');
                        if(codetransmit==1){
                                //Verhindert weitere Aktionen bei den Sendeabfragen
                                transmittedBytes++;
                        }else{
                                //nur warten oder Fehler(2);
                        }
                }
```

```
            //Aktionen zur Verarbeitung der empfangenen Bytes
            if(receivedBytes==1){
                  //Byte receivebuffer[0] verarbeiten
            }else{
                  if(codeoverflow==0){
                        //Es wurden Bytes nicht verarbeitet, die inzwischen
                        //angekommen sind.
                        //Daten bereinigen
                        //receivedBytes=0
                  }else{
                        //wenn der Eingabepuffer bereits übergelaufen ist
                        //dann ist codeoverflow==1
                        //Daten bereinigen
                        //receivedBytes=0
                        //
                        //codeoverflow=0 setzen
                  }

            }
            //...

      }//while(1)
} //main
```

7 I²C Bus

Der I²C Bus (Inter Integrated Circuit) wurde ursprünglich von Philips entwickelt um einen Prozessorkern mit seiner Peripherie auf einfache Art verbinden zu können. Der herkömmliche Weg eine Peripherieeinheit mit dem Prozessorkern zu verbinden, verlangt üblicherweise den Anschluss an den Adress- und Datenbus des Prozessors sowie eine Decodierung der Adressen zum Ansprechen des peripheren Bausteins. Mit der Definition eines synchronen Zweidrahtbusses konnte hier gerade in eingebetteten Systemen Abhilfe geschaffen werden und auch eine hohe Flexibilität beim Anschluss benötigter Hardwareperipherie erreicht werden. [I2C1]

Für einen tieferen Einstieg sei auf folgende Dokumentationen hingewiesen [DB1]

1. The I2C-Bus and How to Use It (including specifications), Philips Semiconductor.
2. The I2C-Bus Specification -- Version 2.0, Philips Semiconductor.
3. System Management Bus Specification -- Version 1.1, SBS Implementers Forum.

Eine dem I²C Bus kompatibler Bus ist der SMBus (System Management Bus). In entsprechende Dokumentationen wird auch dieser Namen verwendet.

7.1 Grundstruktur

Zur Datenübertragung werden eine Datenleitung SDA und eine Taktleitung SDL benötigt. Betrieben wird der Bus in einem Master Slave Konzept. Der Master sendet dabei eine Anforderung auf den Bus, die dann von einem Slave beantwortet wird. Der Master stellt dabei den Takt zur Verfügung. Es können dabei mehrere Slaves am Bus betrieben werden, die über Adressen ausgewählt werden. Multi Master Konzepte sind zwar auch vorgesehen, die im Rahmen der Vorlesung allerdings nicht betrachtet werden sollen.
Der Anschluss an den Bus erfolgt über Open-Drain- bzw. Open-Collector-Anschlüsse, die damit nur in der Lage sind, den Bus auf Masse zu ziehen. Der Eins-Pegel wird von Widerständen zu einer Versorgungsspannung für jede Leitung erzeugt. Damit können auch Einheiten, die mit unterschiedlichen Betriebsspannungen arbeiten, angeschlossen werden.

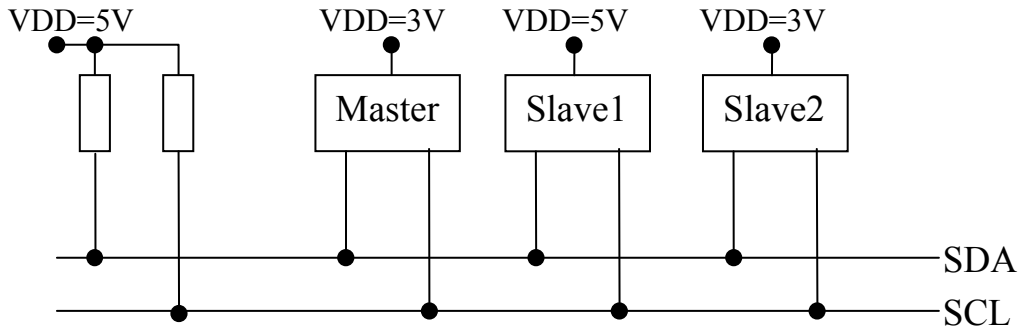


Bild 7.1 Aufbau eines I²C Busses [DB1]

7.1.1 Physikalische Signale

Zur physikalischen Übertragung der Informationen wurden 4 Signalfolgen festgelegt:

1. **Startbedingung**
2. **Stoppbedingung**
3. **Übertragung einer 1**
4. **Übertragung einer 0**

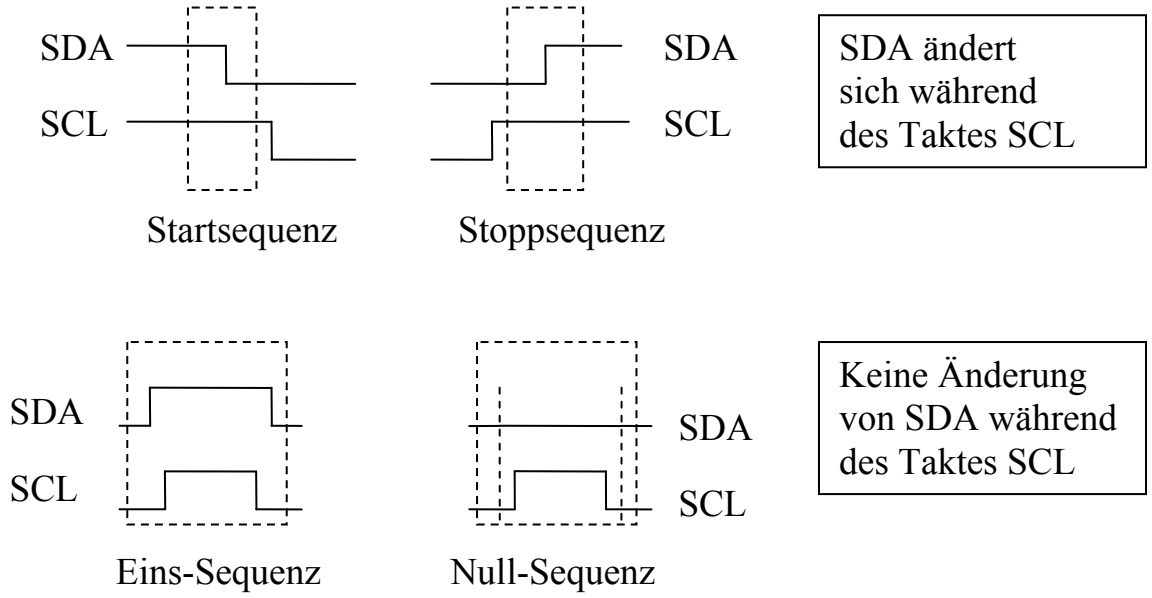


Bild 7.2 Signalfolgen auf dem I^2C Bus [WI211]

Eine Übertragung wird mit einer Startsequenz eingeleitet und mit einer Stoppsequenz beendet. Dazwischen werden die Daten byteweise mit einem Quittungsbit (Acknowledge) (Bild 7.31) übertragen.

Nach der Übertragung von 8 Bit sendet der empfangende Teilnehmer ein Acknowlegde Bit (ACK=0) aus. Ist die Übertragung beendet, so wird vom Sender ein NotAcknowledge Bit gesendet (NAK=1). Der Master kann dann durch eine Stoppsequenz den Bus freigeben.

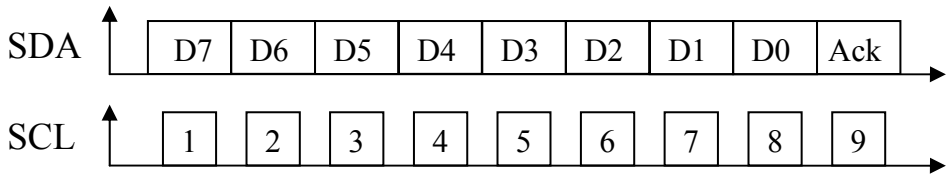
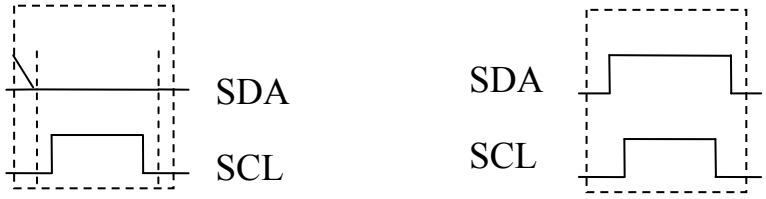


Bild 7.3 Sequenz für ein Datenbyte auf dem I^2C Bus

Dabei haben die übertragenen Bytes in der Reihenfolge in der sie gesendet werden eine bestimmte Bedeutung.



ACK = Null-Sequenz

NACK = Eins-Sequenz

Bild 7.31 ACK- NACK Sequenzen

Das erste vom Master gesendete Byte beinhaltet eine sieben Bit breite Adresse des angesprochenen Busteilnehmers und ein Bit, das die Lese- bzw. Schreiboperation festlegt (R/W Bit). Damit sind 128 verschiedene Adressen möglich, von denen 16 Adressen reserviert sind. Zur Verfügung stehen damit 112 Adressen für mögliche Busteilnehmer. Die Adressen können bei den I^2C-Bausteinen meist nicht völlig frei vergeben werden, mit der Folge, dass die begrenzte Anzahl der Adressen zu Überschneidungen führt. Darum wurde der Adressraum auf 10 Bit ausgeweitet, die eine Teilnehmeranzahl von 1024 Geräten am Bus ermöglicht. Eine gleichzeitige Verwendung von 7-Bit und 10-Bit Adressen ist dabei möglich, da die 10 Bit Adressen alle mit 1111 0 beginnen.
Die Übertragung der Adresse erfolgt dann mit Hilfe zweier Bytes:

1. Byte	**[1**	**1**	**1**	**1 |**	**0**	**A9**	**A8**	**(R/W)**	**]+ ACK**
2. Byte	**[A7**	**A6**	**A5**	**A4 |**	**A3**	**A2**	**A1**	**A0**	**]+ ACK**

Es werden zwei Datentransferoperationen unterschieden:

1. Der Master sendet Daten an den Slave. Dabei liest der Slave die Daten, die der Master gesendet hat.
2. Der Master empfängt Daten vom Slave. Dabei sendet der Slave die Daten, die der Master übernimmt.

In beiden Fällen initiiert der Master den Transfer und stellt den Takt an SCL zur Verfügung.
Eine typische Datenübertragung besteht aus:

- **der Startbedingung,**
- **der Adresse der Slaves (7Bit) + Schreib/Lesekennzeichnung (1 Bit),**
- **der Datenbytes (1 oder mehr) und**
- **der Stoppbedingung**

Die Datenbytes werden im Slave in Registern abgelegt bzw. gelesen, die über Adressen angesprochen werden. Sollen Daten geschrieben werden, wird daher, entsprechend des

verwendeten Bausteins, zuerst ein Byte für eine Registeradresse gesendet, die dann für die folgenden Bytes als Basisadresse zum Abspeichern dient.
Das nächste Byte wird an die um eins inkrementierte Adresse geschrieben. So können relativ effektiv größere Datenmengen übertragen werden.
Beim Lesevorgang wird ebenfalls eine Registeranfangsadresse übermittelt und Daten ab dieser Adresse gelesen.
Bei Bausteinen, die zusätzliche Einstellungen erlauben (z.B. Portexpander) existieren noch Konfigurations- und Kommandoregister, die auf die gleiche Weise beschrieben werden können:

Übergabe einer Registeradresse->Daten für diese Adresse.

7.1.2 Ablauffolgen

Das Ablauffolgediagramm für den Sendevorgang vom Master zum Slave kann bei der Verwendung einer 7-Bit Adresse dann folgendermaßen angegeben werden:

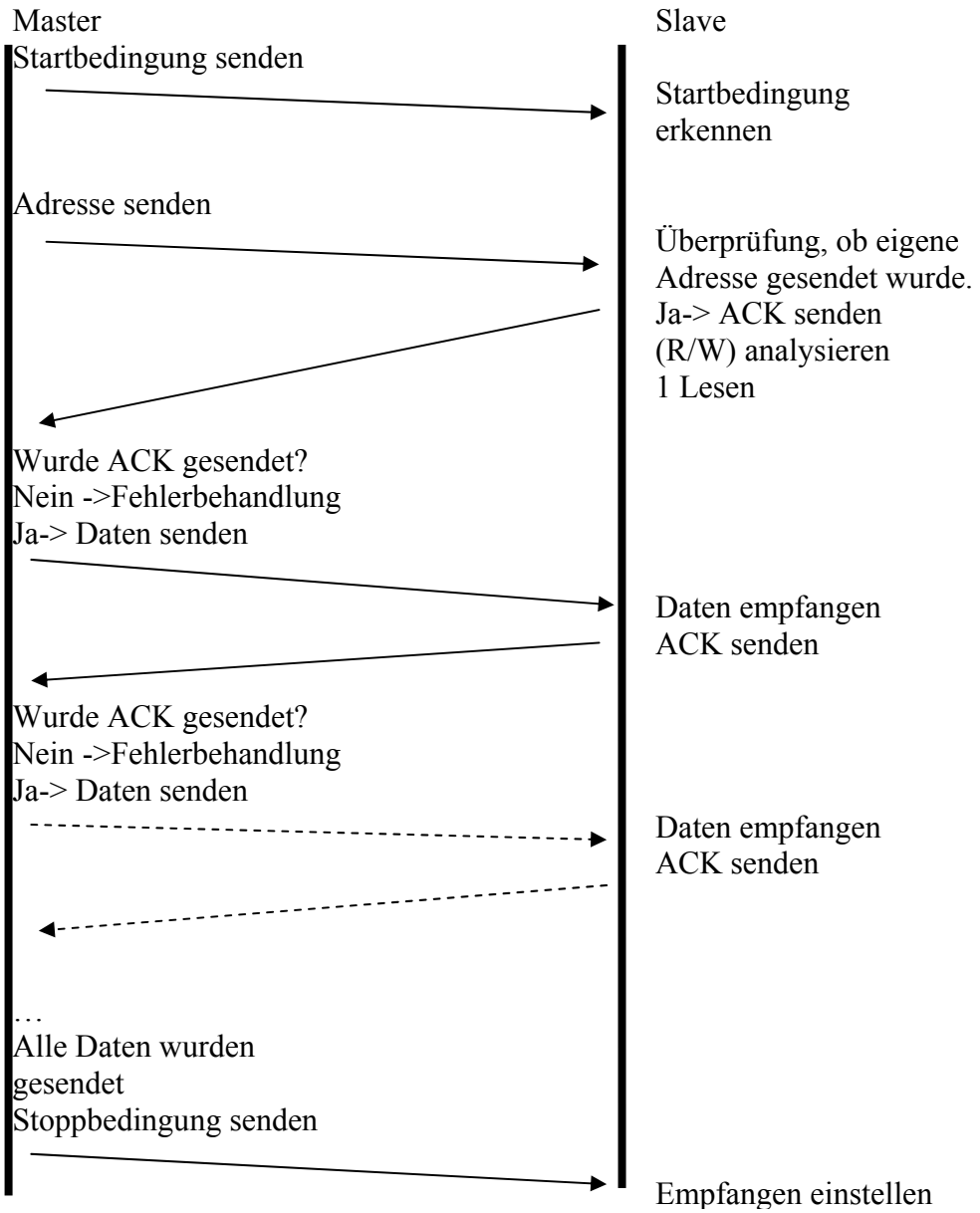


Bild 7.4 Sendevorgang Master zum Slave

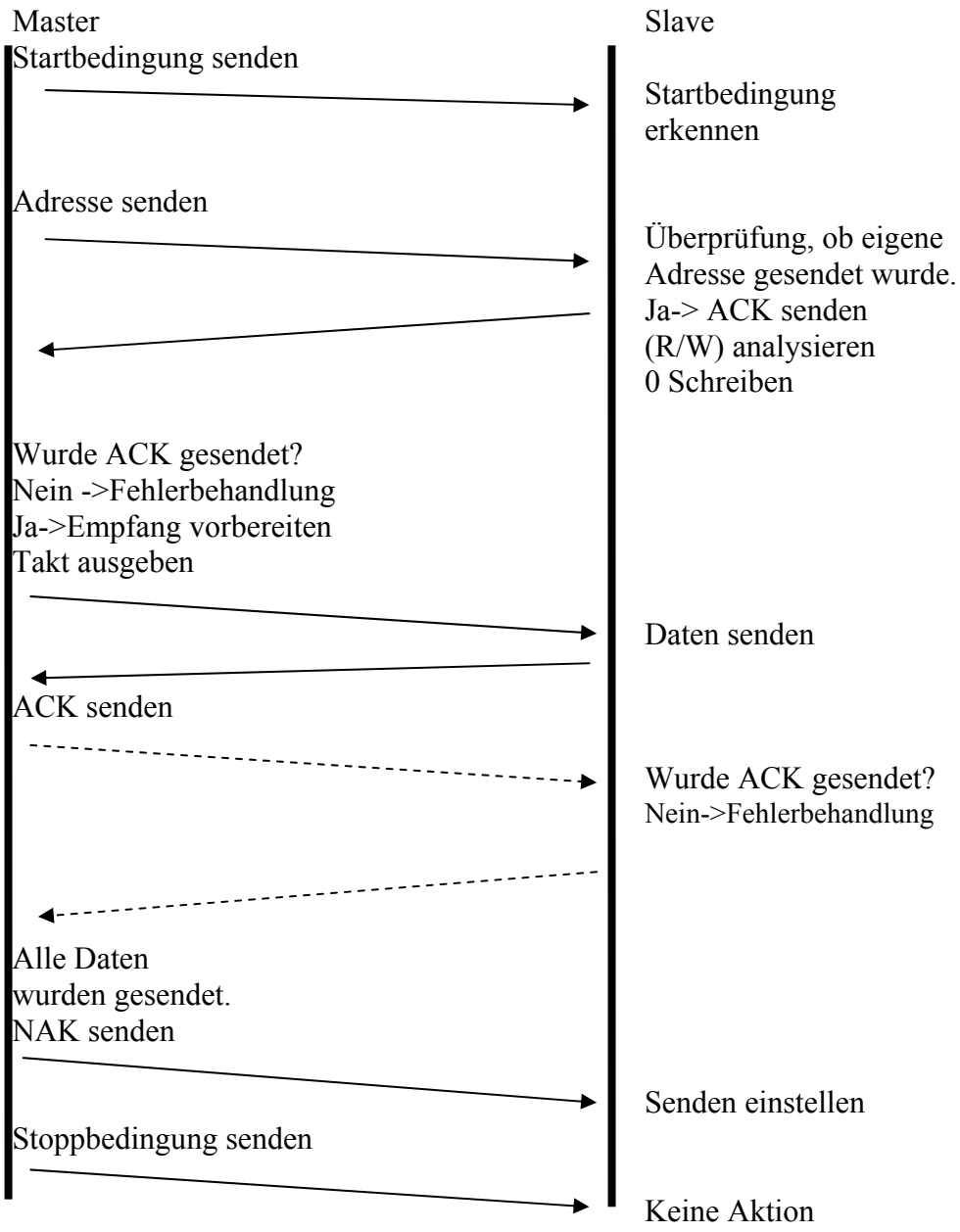


Bild 7.5 Sendevorgang Slave zum Master

7.1.3 Masterrealisierung mittels Software

Programmbeispiel zur Realisierung eines Masters

Zur Realisierung der beiden Anschlüsse SDA und SCL werden Open-Drain-Anschlüsse benötigt. An den Pins müssen dann entsprechend Pull-up-Widerstände dafür sorgen, dass die Anschlüsse auf das Potential der logischen Eins gezogen werden.
Im folgenden Beispiel soll angenommen werden, dass die Anschlüsse P0.2 (SDA) und P0.3 (SDL) entsprechend konfiguriert und beschaltet sind.

```
sbit SDA P0^2;
sbit SDL P0^3;
```

Die Ports werden dann zunächst so initialisiert, dass die Ausgangstransistoren gesperrt sind:

```
SDA = 1;
SCL = 1;
```

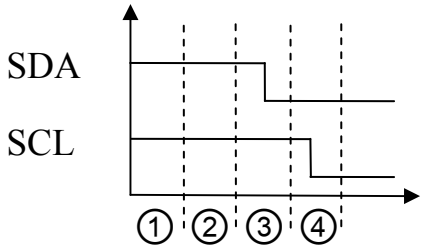
Zur Erzeugung einer Zeitverzögerung zwischen dem SDA und SCL Signal wird eine Funktion definiert, die durch den Aufruf bereits eine Verzögerung erzeugt. Bei Bedarf kann zusätzlich eine Schleife gestartet werden um die Zeitdauer zu erweitern.

```
void i2c_dly(void)
{
}
```

Zur Erzeugung der Start- bzw. Stoppsequenzen werden zwei Funktionen realisiert, welche die benötigten Folgen durch sukzessives Setzen und Löschen der Busausgänge erzeugen. Die zeitlichen Verzögerungen zwischen SDA und SCL Signal wird durch die Verzögerungsfunktion gebildet [ROB11].

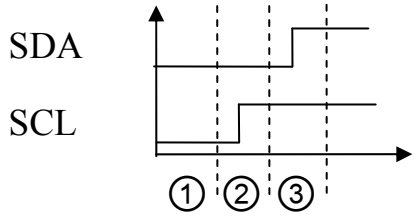
```
//Startsequenz
//SDA                 1100
//SCL                 1110
void i2c_start(void)
{
        // i2c start bit sequence
        SDA = 1;    i2c_dly(); //①//
        SCL = 1;    i2c_dly(); //②//
        SDA = 0;    i2c_dly(); //③//
        SCL = 0;    i2c_dly(); //④//
}
```

Startsequenz



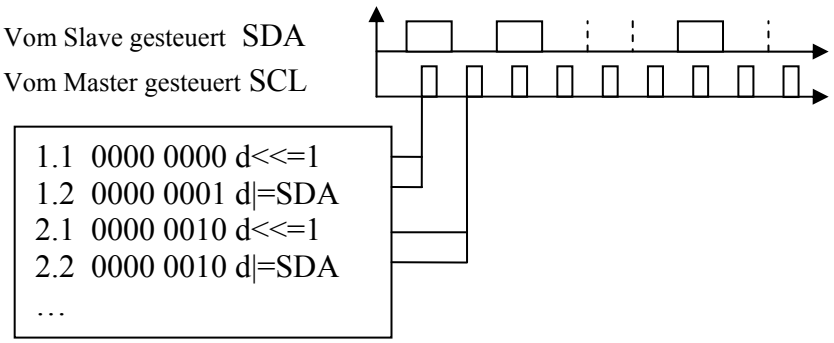

```
//Stoppsequenz
//SDA                 0011
//SCL                 0111
void i2c_stop(void)
{
        // i2c stop bit sequence
        SDA = 0;    i2c_dly();//①//
        SCL = 1;    i2c_dly();//②//
        SDA = 1;    i2c_dly();//③//
}
```

Stoppsequenz



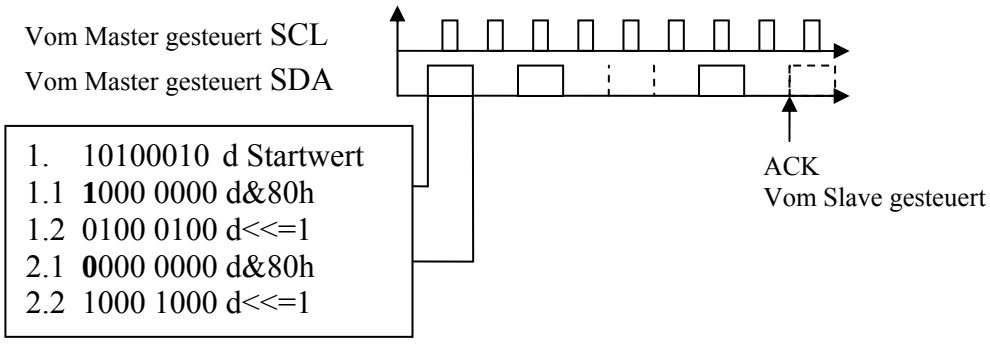
//Funktion zum Empfangen eines Bytes

```
INT8U i2c_rx(INT8U ack)              //ack requirement  act=1->ACK,
                                     //                 act=0-> NACK
{
    INT8U x;                         //loop variable
    INT8U d=0;                       //buffer for bit assembly
    SDA = 1;
    for(x=0; x<8; x++) {
        d <<= 1;                     //①// //Prepare storage of next bit

        //send clock and wait until slave allows to proceed
        do {
            SCL = 1;                 //Master sends SCL=1
        }while(SCL ==0);             //As long as slave overwrites SCL by 0
                                     //wait. SCL clock stretching

        //Now SCL is 1
        i2c_dly();                   //delay between clock and data bit
        if(SDA) d |= 1;              //store received bit
        SCL = 0;                     //prepare next clock cycle
    }
    // send (N)ACK bit
    if(ack) SDA = 0;                 //send ACK
    else    SDA = 1;                 //send NACK

    // create according clock SCL  0,    1,    0,    0
    //                         SDA ack',ack', ack',   1
    i2c_dly();
    SCL = 1;    i2c_dly();
    SCL = 0;    i2c_dly();
    SDA = 1;                         //prepare SDA for next action
    return d;                        //Byte zurückgeben
}
```



//Funktion zum Senden eines Bytes

```
bit i2c_tx(INT8U d)
{
        INT8U x;
        bit b;

        //serialize data and send them
        for(x=8; x; x--) {
                //split one bit
                if(d&0x80)  SDA = 1;
                else        SDA = 0;
                i2c_dly();
                //create according clock
                SCL = 1;    i2c_dly();
                SCL = 0;
                d <<= 1;    i2c_dly();
        }

        //(N)ACK detect
        SDA = 1;                        //Masterassumption

        //create according clock and read ACK Bit
        i2c_dly();  SCL = 1;
        i2c_dly();  b = SDA;            //store possible ACK bit
        SCL = 0;
        return b;
}
```



Die vier angegebenen Funktionen können jetzt zu kompletten Transaktionen kombiniert werden.

In den Beispielen wird von einer Slaveadresse bei 0xE0 ausgegangen.
Im Sendefall soll ein Kommandoregister bei 0x00 angenommen und mit einem Wert geladen werden.

```
i2c_start();            // send start sequence
i2c_tx(0xE0);           // SlaveI2C address with R/W bit clear
i2c_tx(0x00);           // Slave command register address
i2c_tx(0x51);           // send command
i2c_stop();             // send stop sequence
```

Für den Empfangsfall sollen Daten von den Adressen 1 und 2 im Slave gelesen werden. Dazu wird nach dem Ansprechen des Slaves die Anfangsadresse für die Daten auf 1 gesetzt. Beim nächsten Lesevorgang wird vom Slave automatisch auf die nächste Adresse gegangen.

```
i2c_start();            // send start sequence
i2c_tx(0xE0);           // Slave I2C address with R/W bit cleared
i2c_tx(0x01);           // Slave register address
//get data
i2c_start();            // send a restart sequence
i2c_tx(0xE1);           // Slave I2C address with R/W bit set
reg1 = i2c_rx(1);       // get data from internal register address
reg2 = i2c_rx(0);       // get data from internal register address
                        //- note we don't acknowledge the last byte.
i2c_stop();             // send stop sequence
```

7.2 Verwendung des I²C Bus mit dem C8051F340

Zur Entlastung des eigentlichen Rechnerkernes werden auch komplette Einheiten zur Handhabung eines I²C Busses auf dem Mikrocontroller bereits als Hardwarelösung zur Verfügung gestellt. Zur Steuerung der Einheit werden in der üblichen Weise Register zur Übernahme der Daten, Konfigurationsregister und Register zur Darstellung des Zustandes benötigt. Auf dem Mikrocontroller C8051F340 werden drei Register zur Verfügung gestellt um den I²C-Bus zu verwenden. Die grundsätzliche Anordnung dazu zeigt das nächste Bild.

SMB0CF	Konfigurationsregister
SMB0CN	Statusregister
SMB0DAT	Datenregister sowohl zum Senden als auch Empfangen

Tabelle 1: I²C-Register

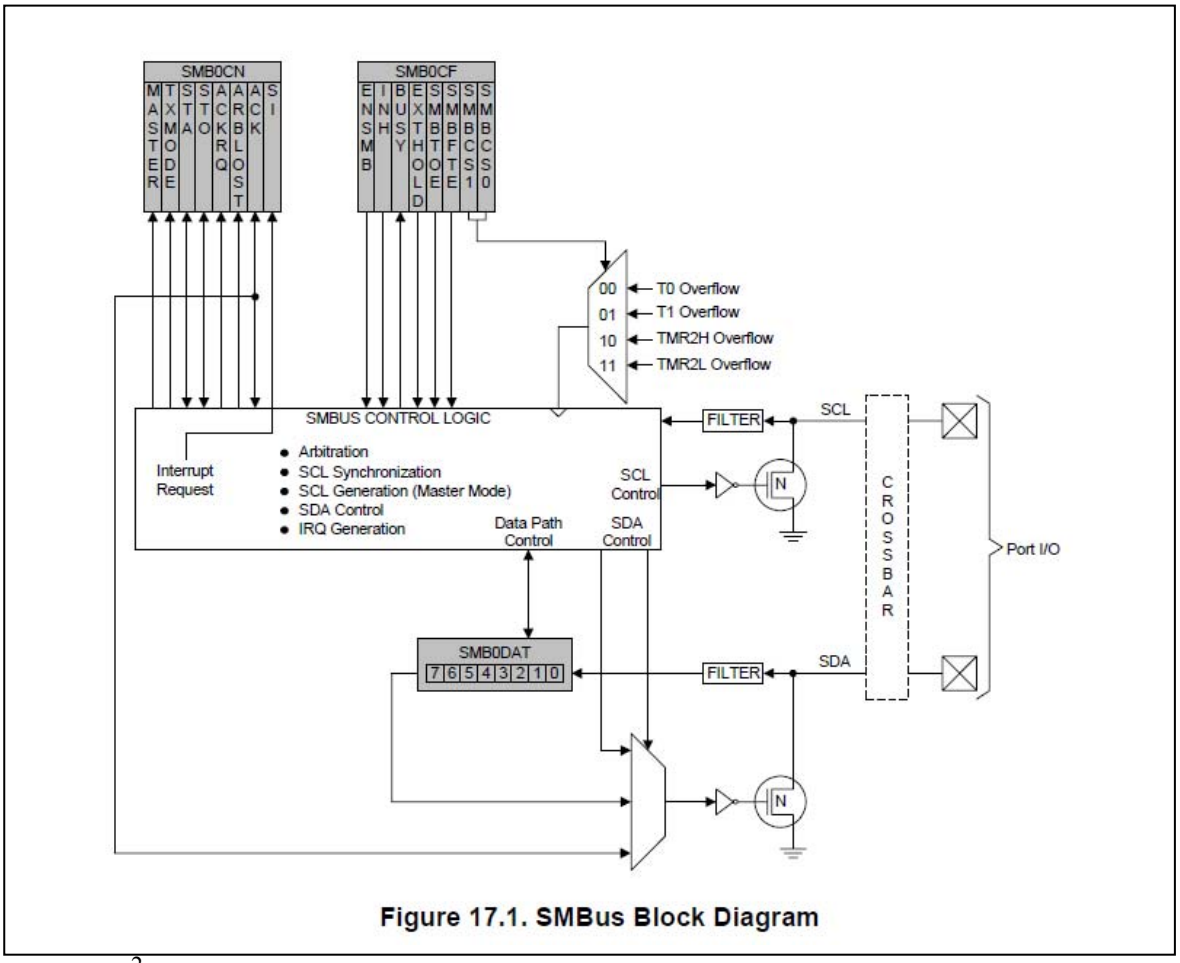


Bild 7.7 I²C Bus Einheit für den C8051F340 [DB1]

SMB0CF

ENSMB	INH	BUSY	EXTHOLD	SMBTOE	SMBFTE	SMBCS1	SMBCS0

Das Konfigurationsregister SMB0CF wird beim Starten des Prozessors auf 0x0 gesetzt. Die beiden niederwertigsten Bit (SMBCS1, SMBCS0) sind für die Auswahl der benötigten Taktquelle vorgesehen. Das Bit ENSMB schaltet die Baugruppe ein. INH lässt nur Interrupts für den Masterbetrieb zu. Für die folgenden Beschreibungen soll für SMB0CF gelten:

SMB0CF = 0xC3; //11000011

SMB0CF

ENSMB	INH	BUSY	EXTHOLD	SMBTOE	SMBFTE	SMBCS1	SMBCS0
1	1	0	0	0	0	1	1

Damit ist der Timer2 als Taktquelle ausgewählt.
Zur vollständigen Beschreibung der Möglichkeiten, die das Konfigurationsregister SMB0CF eröffnet, wird auf die später ebenfalls angegebene, vollständige Beschreibung des Registers SMB0CF hingewiesen.

Für den Betrieb der Schnittstelle werden die Zustände Senden oder Empfangen unterschieden. Die folgenden Diagramme zeigen die Signalfolgen für beide Zustände an. Zusätzlich ist angegeben, an welcher Stelle Interrupts ausgelöst werden.

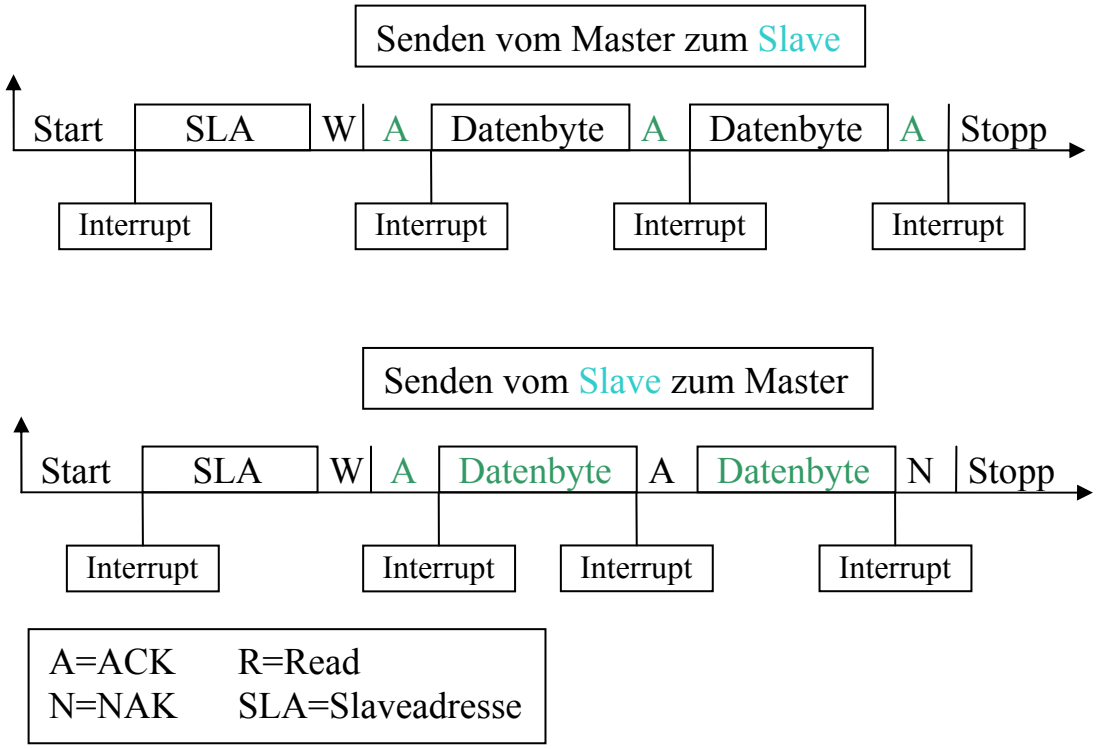


Bild 7.7 Datenübertragung beim I^2C Bus mit Interruptereignissen [DB1]

SMB0CN: SFR-Adresse: 0xC0 Bit-adressierbar

R	R	R/W	R/W	R	R	R/W	R/W	Reset
MASTER	**TXMODE**	**STA**	**STO**	**ACKRQ**	**ARBLOST**	**ACK**	**SI**	00000000
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	

Das Statusregister SMB0CN dient:

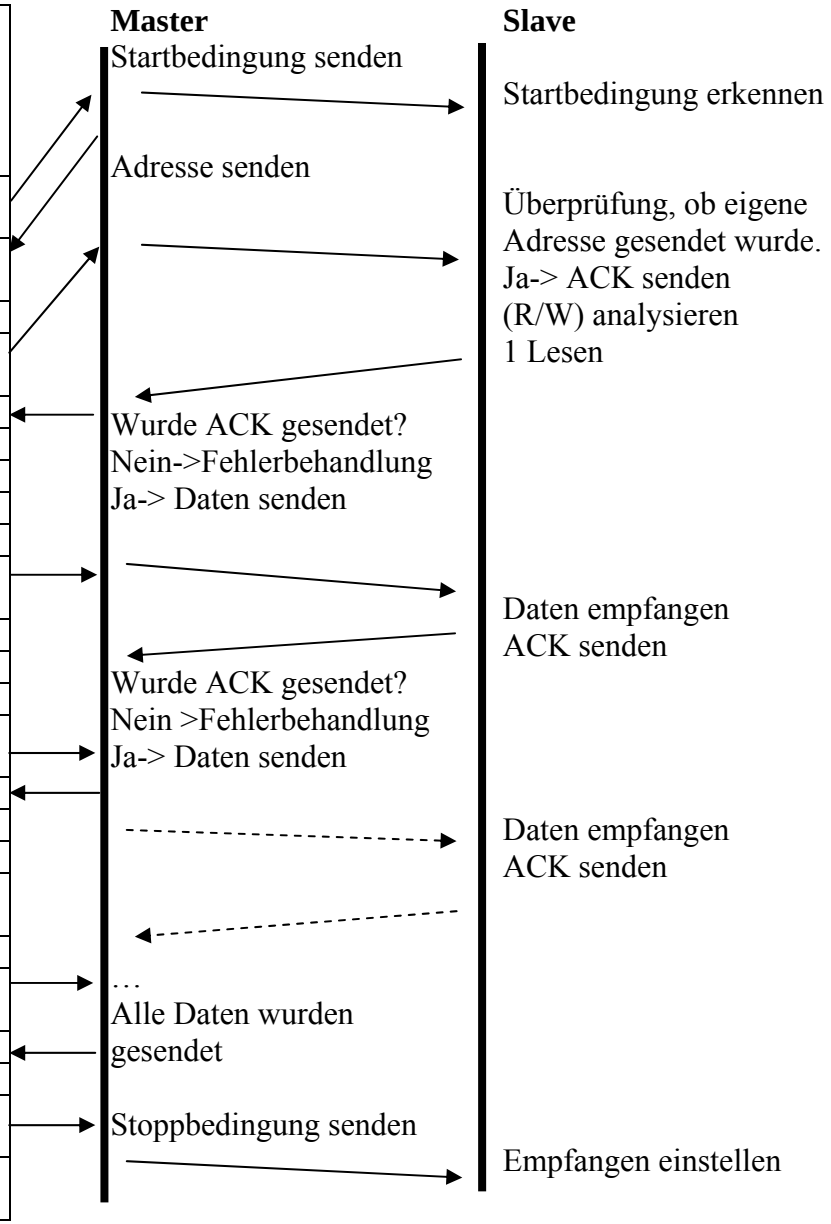
- zur Aufnahme von Daten z.B Acknowledge Bit (ACK)
- zum Initiieren der Start- oder Stoppsequenzen (STA, STO)
- sowie zur Anzeige des Sende- oder Empfangszustandes (TXMODE).
- Das Bit Master wird von der I2C-Einheit gesetzt, wenn eine Startbedingung (STA=1) gegeben wurde und somit angezeigt, dass die Einheit jetzt als Master arbeitet.
- SI zeigt einen aktuellen Interrupt an. Das Flag wird nicht automatisch beim Eintritt in die zugehörige Interruptfunktion gelöscht. Es muss daher per Software gelöscht werden.
- Das Bit ARBLOST zeigt eine fehlerhafte Verbindung an. Zum Rücksetzen der Einheit kann im Register SMB0CF das Flag ENSMB gelöscht und anschließend wieder gesetzt werden.
- ACKRQ =1 fordert das Senden eines Acknowledge

Zur vollständigen Beschreibung der Möglichkeiten, die das Konfigurationsregister SMB0CN eröffnet, wird auf die später ebenfalls angegebene, vollständige Beschreibung des Registers SMB0CN hingewiesen.

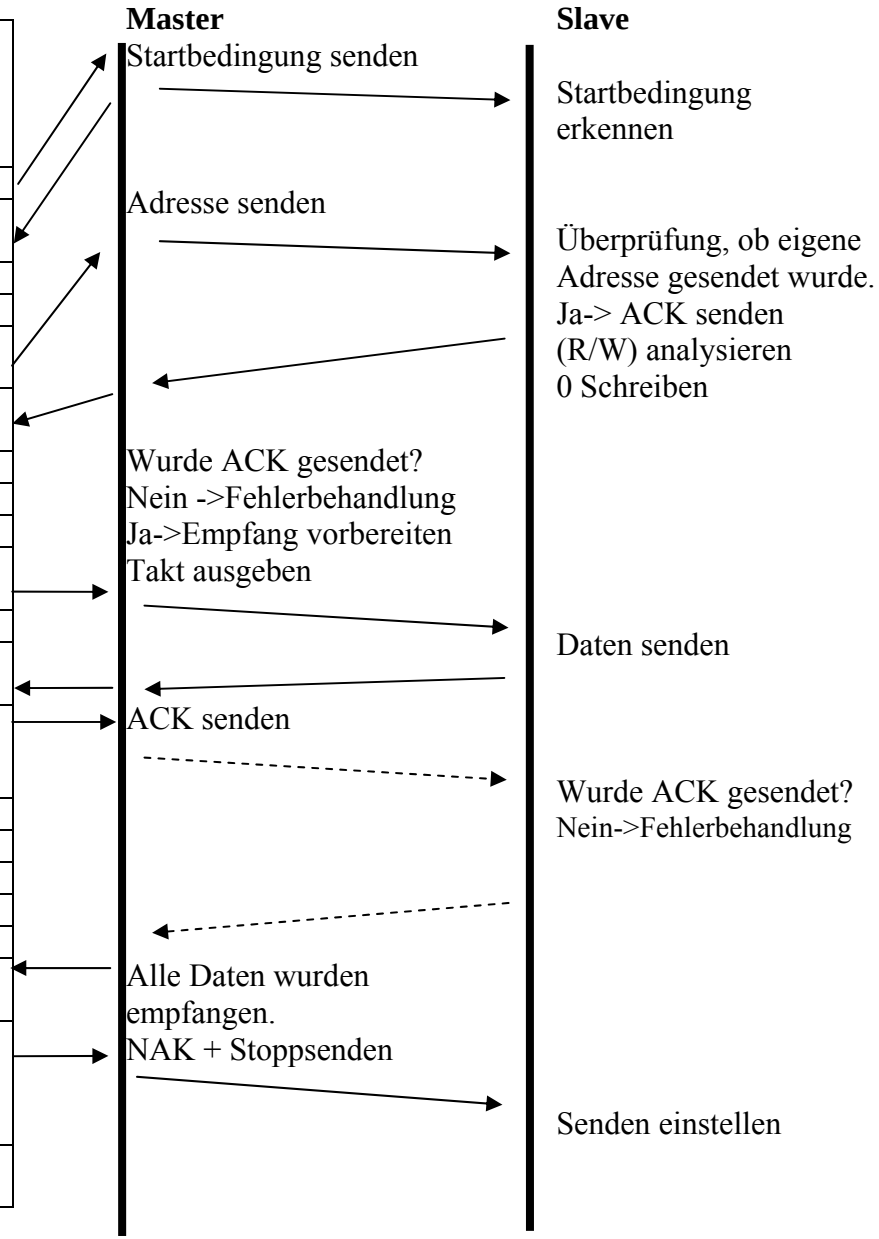
Wird versucht, die notwendigen Aktionen und Antwortmuster für eine einfache Übertragung zu finden, so können die auf den folgenden Seiten angegebenen Muster bestimmt werden:

Im Labor sollen diese Abläufe zur Übung programmtechnisch umgesetzt werden. Weitere Informationen können der Versuchsanleitung entnommen werden.

Nummer	MASTER	TXMODE	STA	STO	ACKRQ	ARBLOST	ACK	SI		SMB0CN Sendevorgang
1S			**1**					**0**		**Start senden**
1E	**1**	**1**	**1**	**0**	**0**	**0**	**X**	**1**	**E1h**	**Start gesendet**
2S			**0**					**0**		**Adresse + R/W=0 in SMB0DAT**
2E	**1**	**1**	**0**	**0**	**0**	**0**	**1**	**1**	**C3h**	**Adresse wurde gesendet**
3S								**0**		**Datum in SMB0DAT wird gesendet**
3E	**1**	**1**	**0**	**0**	**0**	**0**	**1**	**1**	**C3h**	**Datum wurde gesendet**
4S								**0**		**Datum in SMB0DAT wird gesendet**
nE	**1**	**1**	**0**	**0**	**0**	**0**	**1**	**1**	**C3h**	**Datum wurde gesendet**
									…	
nS								**0**		**Datum in SMB0DAT wird gesendet**
nE	**1**	**1**	**0**	**0**	**0**	**0**	**1**	**1**	**C3h**	**Datum wurde gesendet**
mS			**0**	**1**				**0**		**Stopp senden**
iS -> i ter Sendevorgang **jE-> j ter Empfangsvorgang**										



Nummer	MASTER	TXMODE	STA	STO	ACKRQ	ARBLOST	ACK	SI		SMB0CN Empfangsvorgang:
1S			**1**					**0**		**Start senden**
1E	**1**	**1**	**1**	**0**	**0**	**0**	**X**	**1**	**E1h E3h**	**Start gesendet**
2S			**0**					**0**		**Adresse + R/W=1 in SMB0DAT**
2E	**1**	**1**	**0**	**0**	**0**	**0**	**1**	**1**	**C3h**	**Adresse wurde gesendet**
3S								**0**		**Initiiert das Senden der Daten v. Slave**
3E	**1**	**1**	**0**	**0**	**0**	**0**	**X**	**1**	**89h 8Bh**	**Datum wurde in SMB0DAT empfangen**
4S							**1**	**0**		**ACK senden Datenübertragung wird vorgesetzt**
									…	
									…	
									…	
									…	
									…	
4E	**1**	**1**	**0**	**0**	**0**	**0**	**X**	**1**	**89h 8Bh**	**Datum wurde in SMB0DAT empfangen**
nS				**1**			**0**	**0**		**Datenübertragung wird beendet NACK+Stopp senden**
iS -> i ter Sendevorgang jE-> j ter Empfangsvorgang										



Werden die ersten 4 Bit Werte als Status der Schnittstelle betrachtet, der abgefragt werden kann, ergeben sich daraus folgende Aktionen wie in der folgenden Tabelle dargestellt.

	Abfragewerte						Zu setzende Werte		
Mode	Status Vektor	ACKRQ	ARBLOST	ACK	Aktueller Bus Status	Typische Reaktionen	STA	STO	ACK
Master Transmitter	1110	0	0	X	Der Master hat die Startbedingung gesendet	Lade Slaveadresse+ R/W in SMB0DAT	0	0	X
	1100	0	0	0	Ein Daten- oder Adressbyte wurde gesendet. NACK empfangen	STA Setzen für den Transferneustart	1	0	X
						Transfer abbrechen	0	1	X
		0	0	1	Ein Daten- oder Adressbyte wurde gesendet. ACK empfangen.	Nächstes Byte in SMB0DAT laden	0	0	X
						Transferende mit Stopp	0	1	X
						Transferende mit Stopp, Start eines neuen Transfers	1	1	X
						Start noch mal senden	1	0	X
MasterReceiver	1000	1	0	X	Ein Datenbyte wurde empfangen und ein ACK wird gefordert	Empfangenes Byte bestätigen (ACK). SMB0DAT lesen.	0	0	1
						NACK für letztes Byte senden und Stopp senden	0	1	0
						NACK für letztes Byte senden und Stopp gefolgt von Start senden	1	1	0
						ACK mit wiederholtem Start senden	1	0	1
						NACK mit wiederholtem Start senden	1	0	0

SMB0CF: SFR-Adresse: 0xC1 Byteadressierbar Resetwert0x0

R/W	R/W	R	R/W	R/W	R/W	R/W	R/W
ENSMB	INH	BUSY	EXTHOLD	SMBTOE	SMBFTE	SMBCS1	SMBCS0
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0

Bit7

ENSMB: SMBus Enable. Aktiviert und deaktiviert den SMBus.
Ist eine Aktivierung erfolgt werden die SCL- und SDA-Leitungen überwacht.
0: SMBus-Interface deaktiviert.
1: SMBus-Interface aktiviert.

Bit6

INH: SMBus Slave Inhibit.
Wenn diese Option aktiviert ist, werden durch Slave-Events keine Interrupts generiert. Dadurch kann der µC nur noch als Master fungieren.
0: Slave-Modus verfügbar.
1: Slave-Modus nicht möglich.

Bit5

BUSY: SMBus Busy Flag.
Wird durch die Hardware während einem Datentransfer auf 1 gesetzt und gelöscht, wenn eine STOP-Bedingung oder ein Timeout erkannt werden.

Bit4

EXTHOLD: SMBus Setup and Hold Time Extension Enable.
Verlängert die Zeit, die SDA stabil ist, nachdem eine Flanke auf der Taktleitung erfolgt.
0: Verlängerte Setup- und Hold-Zeiten deaktiviert.
1: Verlängerte Setup- und Hold-Zeiten aktiviert.

Bit3

SMBTOE: SMBus SCL Timeout Detection Enable.
Aktiviert die SCL Timeout-Erkennung. Wenn diese Option aktiviert ist, wird Timer3, während SCL high ist, geladen. Wenn SCL low ist, zählt er. Timer3 sollte 25ms lang zählen und in der Interrupt-Routine den SMBus zurücksetzen.
0: SCL Timeout deaktiviert.
1: SCL Timeout aktiviert.

Bit2

SMBFTE: SMBus Free Timeout Detection Enable.
Wenn diese Option aktiviert ist, wird der Bus als frei erkannt, wenn SCL und SDA für mehr als 10 SMBus-Takte High bleiben.
0: Free-Timeout-Erkennung deaktiviert.
1: Free-Timeout-Erkennung aktiviert.

Bit1-0

SMBCS1-SMBCS0: SMBus Clock Source Selection.
Über diese beiden Bits wird die Taktquelle für den SMBus gewählt.

SMBCS1	SMBCS0	SMBus-Takt-Quelle
0	0	Timer 0 Überlauf
0	1	Timer 1 Überlauf
1	0	Timer2 High Byte Überlauf
1	1	Timer2 Low Byte Überlauf

SMB0CN: SFR-Adresse: 0xC0 Bit-adressierbar

R	R	R/W	R/W	R	R	R/W	R/W	Reset
MASTER	TXMODE	STA	STO	ACKRQ	ARBLOST	ACK	SI	00000000
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	

Bit7	MASTER: SMBus Master/Slave Indicator. Zeigt an, ob der SMBus als Master arbeitet. 0: Slave-Modus. 1: Master-Modus.
Bit6	TXMODE: SMBus Transmit Mode Indicator. Zeigt an, ob der SMBus als Sender arbeitet. 0: SMBus in Empfangs-Modus. 1: SMBus in Sende-Modus.
Bit5	STA: SMBus Start Flag. Schreiben: 0: Keine Start-Bedingung generiert. 1: Als Master: eine Start-Bedingung wird gesendet, wenn der Bus frei ist (andernfalls wird auf eine Stopp-Bedingung gewartet und dann ein Start generiert). Wird STA während einem Sendevorgang gesetzt, wird nach dem nächsten ACK ein Repeated-Start generiert. Lesen: 0: Kein Start oder Repeated-Start detektiert. 1: Start oder Repeated-Start detektiert.
Bit4	STO: SMBus Stopp Flag. Schreiben: 0: Keine Stopp-Bedingung wird gesendet. 1: Nach dem nächsten ACK wird eine Stopp-Bedingung gesendet. Die Hardware setzt nach dem Senden STO wieder auf 0. Lesen: 0: Keine Stopp-Bedingung detektiert. 1: Stopp-Bedingung detektiert (Slave-Modus) oder Stopp muss noch gesendet werden (Master-Modus).
Bit3	ACKRQ: SMBus Acknowledge Request. 0: keine Aktion nötig. 1: Der SMBus hat ein Byte empfangen und das ACK-Bit muss mit der korrekten ACK-Antwort beschrieben werden.
Bit2	ARBLOST: SMBus Arbitration Lost Indicator. 0: Bus Fehlerfrei. 1: SMBus hat die Arbitrierung als Sender verloren. Im Slave-Modus wird hierdurch ein Busfehler angezeit.
Bit1	ACK: SMBus Acknowledge Flag. Hier wird die zu sendende ACK-Antwort definiert und die eingehenden Antworten angezeigt. Es sollte nach jedem empfangenen Byte geschrieben, bzw. nach jedem gesendeten Byte gelesen werden. 0: ein „not acknowledge“ wurde empfangen (Sende-Modus) ODER wird gesendet (Empfangs-Modus). 1: ein „acknowledge“ wurde empfangen (Sende-Modus) ODER wird gesendet (Empfangs-Modus).
Bit0	SI: SMBus Interrupt Flag. Wird von der Hardware gesetzt und muss per Software gelöscht werden. Während SI=1 ist ist der Takt auf dem Bus angehalten

Zum Betrieb der I²C Schnittstelle müssen die entsprechenden Konfigurationsregister zunächst gesetzt, die Adressierungen vorgenommen und die Daten entsprechend vorbereitet werden.
Durch eine entsprechende Einführung von verschiedenen Send- und Empfangspuffer und die Einführung eines Befehlsstapelspeichers kann auch der I2C Bus so gesteuert werden, dass weitere Operationen vom Prozessor durchgeführt werden können.
Die Organisation des Betriebs einer I²C Schnittstelle bei mehreren angeschalteten Teilnehmern ist jedoch komplexer als bei der bereits beschriebenen seriellen Schnittstelle.

Da an dieser Stelle jedoch zunächst nur die prinzipielle Funktion bei der Hardwareeinheit gezeigt werden soll, wird die Vereinbarung getroffen, dass beim Senden und Empfangen mit Hilfe der I2C Schnittstelle immer die folgenden Aktionen erfolgen:

1. **Laden der Datenpuffer (Adresse, Daten)**
2. **Warten bis der I2C Bus frei ist.**
3. **Sende- oder Empfangsaktion anstoßen.**
4. **Warten bis der I2Bus nicht mehr belegt ist.**

   Die Wartezeit ist notwendig, wenn die gleichen Datenpuffer für die jeweiligen Sende- und Empfangsvorgänge verwendet werden. Es findet sonst ein vorzeitiges überschreiben statt.

Es soll dabei immer nur ein Gerät am I2C Bus vom Master bedient werden.
Ähnlich der gezeigten Softwarerealisierung können jetzt mit Hilfe der vorhandenen Hardware Sendefunktionen und Empfangsfunktionen angegeben werden.

```
void I2C_Write (INT8U Addr, INT8U num_bytes)
void I2C_Read (INT8U Addr, INT8U num_bytes)
```

Hierbei werden die Adresse des Teilnehmers an der I2C Schnittstelle, die Anzahl der zu sendenden oder zu empfangenden Bytes angegeben.
Die Daten selbst werden in einem globalen Feld (I2C_Buf) abgelegt, das sowohl die Daten zum Senden als auch zum Empfangenen aufnimmt.

```
#define I2CBUFFERSIZE 5
INT8U I2C_Buf   [I2CBUFFERSIZE];
```

Neben den Funktionen zum Initiieren eines Sende- und Empfangsvorgangs muss durch den Betrieb der Hardware bedingt, eine Interruptserviceroutine installiert werden.
Die eigentlichen Sende- und Empfangsoperationen werden dann in der Interrupt-Service-Funktion des I²C Busses ausgeführt.

Die benötigten Daten müssen komplett in globalen Variablen übergeben oder zurückgespeichert werden. Hierzu können folgende Variablen definiert werden:

```
bit     smb_RW;                 //Kennzeichnung RW==0 Senden;
                                //              RW==1 Empfangen
INT8U smb_Target;               //Adresse
INT8U smb_numBytesToProcess;    //Anzahl der zu sendenden Bytes
INT8U smb_numBytesProcessed;    //Aktuell gesendete Bytes
INT8U I2C_Buf[I2CBUFFERSIZE]; //Sende-/Empfangspuffer

bit     smb_busy;               //Benutzerdefiniertes Warteflag
```

Zur Feststellung, ob die I2C Schnittstelle frei ist, kann auf das BUSY Flag (Bit 5) im allerdings nur bytedressierbaren Register SMB0CN zugegriffen werden. Der Zugriff auf dieses Register darf jedoch nach seiner Änderung nur nach einer bestimmten Wartezeit erfolgen. Die Wartezeit kann mit folgender Schleife erzeugt werden:

```
INT8U xx;
for(xx=0;xx<32;xx++);  //Wartezeit
while(SMB0CN & 0x20);       //Warten bis I2C-Schnittstelle frei ist.
```

Bei der Bearbeitung der einzelnen Aktionen in der Interruptserviceroutine kann auf das Register SMB0CN ebenfalls erst nach einer Wartezeit zugegriffen werden.

Wird vom Benutzer nun ein eigenes Flag (smb_busy) eingeführt, das beim Starten der I2C Schnittstelle gesetzt und beim Initiieren der Stoppbedingung zurückgesetzt wird, kann eine Besetztabfrage ohne die Wartezeitschleife durchgeführt werden.
Die Verwendung eines solchen Flags ist in der folgenden prinzipiellen Programmbeschreibung der Interruptserviceroutine angegeben. Im assoziierten Labor soll der konkrete Code dazu erarbeitet werden:

```
void I2C_ISR (void) interrupt 7
{
        if(ARBLOST){        //der Bus ist nicht mehr aktiv
                //Konfigurationregister SMB0CF zurücksetzen
                //Statusregister SMB0CN zurücksetzen
                SMB_busy = 0;        //Benutzerflag "I2C Schnittstelle besetzt" zurücksetzen
        }else{
                for(xx=0;xx<32;xx++);        //Wartezeit für den Zugriff auf SMB0CN
                switch(SMB0CN){
                        case 0xE1:     // Start gesendet, NACK empfangen
                        case 0xE3:     // Start transmitted, ACK empfangen
                                //SMB0DAT mit der
                                //Adresse (smb_Target) und der
                                //Lese-/Schreibinformation (smb_RW) laden
                                //Sart Anforderung löschen
                                break;
                        case 0xC3:     // Slave Adresse oder Daten gesendet, ACK empfangen
                                if (SMB_RW == 0) {//Sendemodus
                                        if(alle Bytes gesendet ){
                                                //Stopp senden
                                                SMB_busy = 0;
                                        }else{
                                                //Im Sendemodus die folgenden Bytes senden
                                        }
                                }
                                break;

                        case 0x89:     // Slave Adresse + Read gesendet, NACK empfangen
                        case 0x8B:     // Slave Adresse + Read gesendet, ACK empfangen
                                //empfangene Daten aus SMB0DAT nach I2Cin laden
                                if(alle Daten wurden empfangen){
                                        //NACK senden
                                        //Stoppbedingung senden
                                        SMB_busy = 0;
                                }
                                break;

                        case 0xC1:     // Slave Adresse oder Daten gesendet, NACK empfangen
                        default:
                                //Nochmaliger Start
                                STO = 1;
                                STA = 1;
                }
        }
        SI = 0; //Interruptflag zurücksetzen und eingeleitete Aktion starten
}
```

Im Hauptprogramm werden der Sende- bzw. der Empfangsvorgang angestoßen. Dies kann aber nur erfolgen, wenn der Bus gerade nicht aktiv ist. Im einfachsten Fall wird jeweils gewartet bis der Bus frei ist, anschließend der Bus durch das Setzen des Flags smb_busy belegt. Danach werden die notwendigen Daten in die globalen Variablen kopiert, der Bus

gestartet und erneut auf eine Freigabe gewartet, die von der Interruptserviceroutine durchgeführt wird. Durch die Verwendung nur eines Puffers muss der aktuelle Vorgang abgewartet werden, da sonst die Datem überschrieben würden. Dazu können Programme erstellt werden, die in ihren Funktionen den angegebenen Programmen I2C_Write und I2C_Read entsprechen:

```
// Senden von Daten
void I2C_Write (INT8U Addr, INT8U num_bytes)
{
        while(smb_busy)  {}                 //Warten bis der Bus frei ist.
                                            (Benutzerdefiniertes Flag)
        smb_busy = 1;                       //Belegung des Busses
        smb_Target = Addr;                  //Adressregister laden (Benutzerdefiniert)
        smb_numBytesProcessed = 0;                  //Zähler (Benutzerdefiniert)
        smb_numBytesToProcess = num_bytes; //Anzahl der zu übertragenden Bytes
        smb_RW = 0;                         //Flag zeigt an, dass geschrieben werden soll
                                            // (Benutzerdefiniert)
        STA = 1;                            // Bus starten

        while(smb_busy)  {}         //Warten bis der Schreibvorgang beendet ist
}
//
// Empfangen von Daten
void I2C_Read (INT8U Addr, INT8U num_bytes)
{
        while(smb_busy)  {}                 //Warten bis der Bus frei ist

        smb_busy = 1;                       //Belegung des Busses (Benutzerdefiniert)
        smb_Target = Addr;                  //Adressregister laden (Benutzerdefiniert)
        smb_numBytesProcessed = 0;                  //Zähler (Benutzerdefiniert)
        smb_numBytesToProcess= num_bytes;  //Anzahl der zu empfangenden Bytes
        smb_RW = 1;                         //Flag zeigt an, dass gelesen werden soll
                                            // (Benutzerdefiniert)
        STA = 1;                            // Bus starten

        while(smb_busy)  {}                 //Warten bis der Lesevorgang beendet ist
}
void main(){
        //einen Wert von einem externen Baustein lesen:
        //Ausgabepuffer mit Registeradresse laden
        I2C_Buf[0] = 0x00;
        I2C_Write(0x42, 1);         //An Adresse 0x42 die Registeradresse senden

        I2C_Read(0x42, 1); //Von der Adresse 0x42 Portinhalt holen

        // I2C_Buf[0] auswerten
}// End Main
```

8 Vergleich C-Code –Assembler

8.1 Arithmetische Operationen

8.1.1 Byte Verarbeitung

Bei der Verwendung verschiedener Datentypen in C werden vom Compiler entsprechende Umsetzungen vorgenommen. Daten im Mikrocontroller 8051 sind zunächst auf 8 Bit begrenzt. Sollen Daten verwendet werden, die mehr als 8 Bit benötigen z.B. int, müssen diese Operationen in mehreren Schritten durchgeführt werden. Operationen wie Addition und Subtraktion sind dabei üblicherweise relativ einfach erweiterbar. Multiplikation und Division sind aufwendiger zu realisieren. Da Ressourcen in einem Mikrocontroller stark begrenzt sind, ist der Umgang mit Datentypen sehr sorgfältig vorzunehmen. Es werden nicht nur Speicher im Sinne von RAM-Speicherplatz benötigt, sondern es muss auch der benötigte Programmspeicherplatz zur Verfügung gestellt werden.
Die einfachste Abbildung von Datentypen in C ist mit dem Type unsigned char möglich. Da dieser Datentyp nur 8 Bit benötigt, kann er direkt in die Datenlänge des Prozessors umgesetzt werden. Im folgenden Beispiel werden drei Variablen von diesem Typ definiert. Zur Abkürzung der Variablendefinition wurde der Typ „unsigned char“ durch die Typvereinbarung „typedef unsigned char BYTE“ in BYTE umbenannt. Die Variable werden dann bei verschiedenen arithmetischen Operationen eingesetzt. Negative Zahlen werden in die Darstellung im K2-Komplement umgewandelt und dann als vorzeichenlose Zahlen behandelt. Nach der Kompilierung entsteht ein Assemblerprogramm, das direkt die zur Verfügung stehenden Befehle aus dem Befehlssatz des Prozessors verwendet um die geforderten Operationen auszuführen: Add, Subb, MUL AB, DIV AB.

```
//File Byte.c
#include <reg51.h>       /* define 8051 registers */

typedef unsigned char BYTE;
void main()
{
     BYTE a, b, y;
     a=5; b=6;
     y=a+b;
     y=a-b;
     y=a*b;
     y=a/b;
     a=5; b=-6;
     y=a+b;
     y=a-b;
     y=a*b;
     y=a/b;
}
```

Zur Demonstration der Arbeitsweise des Compilers [KEI05] wurde das gesamte Listing einschließlich des Include Files dargestellt.

Die im Prozessor vorhandenen Speicherplätze werden über spezielle Definition zur C-Programmierung bekannt gemacht (sfr=Special Function Register, sbit Einzelbitdefinition). Das erzeugte Assemblerprogramm wird mit Bezug auf den ursprünglichen Programmcode angegeben. Anschließend erfolgt die Angabe der Speicherplatzbelegung.

```
C51 COMPILER V6.01 BYTE                                 04/23/2006 12:02:10 PAGE 1
C51 COMPILER 6.01, COMPILATION OF MODULE BYTE
OBJECT MODULE PLACED IN .\Byte.OBJ
COMPILER INVOKED BY: C:\Keil\C51\BIN\C51.EXE .\Byte.c DEBUG OBJECTEXTEND
CODE LISTINCLUDE SYMBOLS
stmt level    source
   1          //File Byte.c
   2          #include <reg51.h>               /* define 8051 registers */
   1      =1  /*-------------------------------------------------------------
   2      =1  REG51.H
   3      =1
   4      =1  Header file for generic 80C51 and 80C31 microcontroller.
   6      =1  All rights reserved.
   7      =1  -------------------------------------------------------------*/
   8      =1
   9      =1  /*  BYTE Register  */
  10      =1  sfr P0   = 0x80;
  11      =1  sfr P1   = 0x90;
  12      =1  sfr P2   = 0xA0;
  13      =1  sfr P3   = 0xB0;
  14      =1  sfr PSW  = 0xD0;
  15      =1  sfr ACC  = 0xE0;
  16      =1  sfr B    = 0xF0;
  17      =1  sfr SP   = 0x81;
  18      =1  sfr DPL  = 0x82;
  19      =1  sfr DPH  = 0x83;
  20      =1  sfr PCON = 0x87;
  21      =1  sfr TCON = 0x88;
  22      =1  sfr TMOD = 0x89;
  23      =1  sfr TL0  = 0x8A;
  24      =1  sfr TL1  = 0x8B;
  25      =1  sfr TH0  = 0x8C;
  26      =1  sfr TH1  = 0x8D;
  27      =1  sfr IE   = 0xA8;
  28      =1  sfr IP   = 0xB8;
  29      =1  sfr SCON = 0x98;
  30      =1  sfr SBUF = 0x99;
  31      =1
  32      =1
  33      =1  /*  BIT Register  */
  34      =1  /*  PSW   */
  35      =1  sbit CY   = 0xD7;
  36      =1  sbit AC   = 0xD6;
  37      =1  sbit F0   = 0xD5;
  38      =1  sbit RS1  = 0xD4;
  39      =1  sbit RS0  = 0xD3;
  40      =1  sbit OV   = 0xD2;
  41      =1  sbit P    = 0xD0;
  42      =1
  43      =1  /*  TCON  */
  44      =1  sbit TF1  = 0x8F;
  45      =1  sbit TR1  = 0x8E;
  46      =1  sbit TF0  = 0x8D;
  47      =1  sbit TR0  = 0x8C;
  48      =1  sbit IE1  = 0x8B;
  49      =1  sbit IT1  = 0x8A;
  50      =1  sbit IE0  = 0x89;
  51      =1  sbit IT0  = 0x88;
  52      =1
  53      =1  /*  IE   */
C51 COMPILER V6.01 BYTE                                 04/23/2006 12:02:10 PAGE 2

  54      =1  sbit EA   = 0xAF;
  55      =1  sbit ES   = 0xAC;
  56      =1  sbit ET1  = 0xAB;
  57      =1  sbit EX1  = 0xAA;
  58      =1  sbit ET0  = 0xA9;
  59      =1  sbit EX0  = 0xA8;
  60      =1
  61      =1  /*  IP   */
  62      =1  sbit PS   = 0xBC;
```

```
 63       =1  sbit PT1  = 0xBB;
 64       =1  sbit PX1  = 0xBA;
 65       =1  sbit PT0  = 0xB9;
 66       =1  sbit PX0  = 0xB8;
 67       =1
 68       =1  /*  P3  */
 69       =1  sbit RD   = 0xB7;
 70       =1  sbit WR   = 0xB6;
 71       =1  sbit T1   = 0xB5;
 72       =1  sbit T0   = 0xB4;
 73       =1  sbit INT1 = 0xB3;
 74       =1  sbit INT0 = 0xB2;
 75       =1  sbit TXD  = 0xB1;
 76       =1  sbit RXD  = 0xB0;
 77       =1
 78       =1  /*  SCON  */
 79       =1  sbit SM0  = 0x9F;
 80       =1  sbit SM1  = 0x9E;
 81       =1  sbit SM2  = 0x9D;
 82       =1  sbit REN  = 0x9C;
 83       =1  sbit TB8  = 0x9B;
 84       =1  sbit RB8  = 0x9A;
 85       =1  sbit TI   = 0x99;
 86       =1  sbit RI   = 0x98;
  3
  4           typedef unsigned char BYTE;
  5           void main()
  6           {
  7   1           BYTE a, b, y;
  8   1           a=5; b=6;
  9   1           y=a+b;
 10   1           y=a-b;
 11   1           y=a*b;
 12   1           y=a/b;
 13   1           a=5; b=-6;
 14   1           y=a+b;
 15   1           y=a-b;
 16   1           y=a*b;
 17   1           y=a/b;
 18   1
 19   1       }
```

```
C51 COMPILER V6.01 BYTE                 04/23/2006 12:02:10 PAGE 3
ASSEMBLY LISTING OF GENERATED OBJECT CODE
             ; FUNCTION main (BEGIN)
                                     ; SOURCE LINE # 5
                                     ; SOURCE LINE # 6
                                     ; SOURCE LINE # 8
;---- Variable 'a' assigned to Register 'R7' ----
0000 7F05               MOV     R7,#05H
;---- Variable 'b' assigned to Register 'R6' ----
0002 7E06               MOV     R6,#06H
                                       ; SOURCE LINE # 9
                                     ;     y=a+b;
0004 EF                 MOV     A,R7
0005 2E                 ADD     A,R6
0006 FD                 MOV     R5,A
;---- Variable 'y' assigned to Register 'R5' ----
                                       ; SOURCE LINE # 10
                                     ;     y=a-b;
0007 C3                 CLR     C
0008 EF                 MOV     A,R7
0009 9E                 SUBB    A,R6
000A FD                 MOV     R5,A
                                       ; SOURCE LINE # 11
                                     ;     y=a*b;
000B EF                 MOV     A,R7
000C 8EF0               MOV     B,R6
000E A4                 MUL     AB
000F FD                 MOV     R5,A
                                       ; SOURCE LINE # 12
                                     ;     y=a/b;
0010 EF                 MOV     A,R7
0011 8EF0               MOV     B,R6
0013 84                 DIV     AB
0014 FD                 MOV     R5,A
                                       ; SOURCE LINE # 13
0015 7EFA               MOV     R6,#0FAH
                                       ; SOURCE LINE # 14
                                     ;     y=a+b;
0017 EF                 MOV     A,R7
0018 2E                 ADD     A,R6
0019 FD                 MOV     R5,A
                                       ; SOURCE LINE # 15
                                     ;     y=a-b;
001A C3                 CLR     C
001B EF                 MOV     A,R7
001C 9E                 SUBB    A,R6
001D FD                 MOV     R5,A
                                       ; SOURCE LINE # 16
                                     ;     y=a*b;
001E EF                 MOV     A,R7
001F 8EF0               MOV     B,R6
0021 A4                 MUL     AB
0022 FD                 MOV     R5,A
                                       ; SOURCE LINE # 17
                                     ;     y=a/b;
0023 EF                 MOV     A,R7
0024 8EF0               MOV     B,R6
0026 84                 DIV     AB
0027 FD                 MOV     R5,A
                                       ; SOURCE LINE # 19
0028 22                 RET             ; FUNCTION main (END)
```

```
C51 COMPILER V6.01 BYTE                          04/23/2006 12:02:10 PAGE 4
NAME                                    CLASS   MSPACE  TYPE    OFFSET  SIZE
====                                    =====   ======  ====    ======  ====
T0 . . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       00B4H  1
AC . . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       00D6H  1
T1 . . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       00B5H  1
EA . . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       00AFH  1
RD . . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       00B7H  1
ES . . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       00ACH  1
RI . . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       0098H  1
INT0 . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       00B2H  1
CY . . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       00D7H  1
TI . . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       0099H  1
INT1 . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       00B3H  1
PS . . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       00BCH  1
OV . . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       00D2H  1
main . . . . . . . . . . . . . . . . .  PUBLIC   CODE   PROC      0000H  ---
--
  a. . . . . . . . . . . . . . . . . .  * REG *  DATA   U_CHAR    0007H  1
  b. . . . . . . . . . . . . . . . . .  * REG *  DATA   U_CHAR    0006H  1
  y. . . . . . . . . . . . . . . . . .  * REG *  DATA   U_CHAR    0005H  1
WR . . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       00B6H  1
BYTE . . . . . . . . . . . . . . . . .  TYPEDEF  -----  U_CHAR    -----  1
IE0. . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       0089H  1
IE1. . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       008BH  1
ET0. . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       00A9H  1
ET1. . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       00ABH  1
TF0. . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       008DH  1
TF1. . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       008FH  1
RB8. . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       009AH  1
EX0. . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       00A8H  1
IT0. . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       0088H  1
TB8. . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       009BH  1
EX1. . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       00AAH  1
IT1. . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       008AH  1
P. . . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       00D0H  1
SM0. . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       009FH  1
SM1. . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       009EH  1
SM2. . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       009DH  1
PT0. . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       00B9H  1
PT1. . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       00BBH  1
RS0. . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       00D3H  1
TR0. . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       008CH  1
RS1. . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       00D4H  1
TR1. . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       008EH  1
PX0. . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       00B8H  1
PX1. . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       00BAH  1
REN. . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       009CH  1
RXD. . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       00B0H  1
TXD. . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       00B1H  1
F0 . . . . . . . . . . . . . . . . . .  ABSBIT   -----  BIT       00D5H  1
MODULE INFORMATION:   STATIC OVERLAYABLE
   CODE SIZE        =     41    ----
   CONSTANT SIZE    =   ----    ----
   XDATA SIZE       =   ----    ----
   PDATA SIZE       =   ----    ----
   DATA SIZE        =   ----    ----
   IDATA SIZE       =   ----    ----
   BIT SIZE         =   ----    ----
END OF MODULE INFORMATION.
```

8.1.2 Word Verarbeitung

Unter einem Wort soll eine 2-Byte Zahl ohne Vorzeichen verstanden werden.
Addition und Subtraktion können mit den Befehlen Addc und Subb relativ einfach auf 2 Bytes erweitert werden. Zur Multiplikation und Division werden vom Compiler eigene Unterprogramme eingefügt.
Eine 2. Bytezahl wird als unsigned int definiert. Um auch hier eine einfache Schreibweise einzuführen, wurde die folgende Typdefinition vorgenommen:

```
typedef unsigned int WORD;
```

Die Auswirkungen der Verwendung eines solchen Datentyps auf das erzeugte Assemblerprogramm soll an folgendem Programm demonstriert werden:

```
stmt level    source

   1          //File Word.c
   2          #include <reg51.h> /* define 8051 registers */
   3
   4          typedef unsigned int WORD;
   5          void main()
   6          {
   7    1      WORD a, b, y;
   8    1      a=5; b=6;
   9    1      y=a+b;
  10    1      y=a-b;
  11    1      y=a*b;
  12    1      y=a/b;
  13    1      }
```

Der erzeugte Assembler Code für Addition und Subtraktion steigt linear mit der Anzahl der zu verarbeitenden Bytes. Die Multiplikation kann ebenfalls relativ einfach realisiert werden, indem die Operanden in MSB und LSB getrennt werden und als Summen ausmultipliziert werden:

```
; Product =  (256* Ahigh + Alow) * (256*Bhigh + Blow)

;Ausmultipliziert ergibt sich dann:
;       Produkt =     (256 * Ahigh * 256 * Bhigh) +
;                     (256 * Ahigh * Blow) +
;                     (256 * Bhigh * Alow) +
;                     (Alow * Blow)
```

Das Ergebnis muss wieder in 2 Bytes passen, sonst liegt eine Zahlenbereichsüberschreitung vor. Die Überschreitung wird nicht explizit abgefangen.

```
C51 COMPILER V6.01 WORD                        04/23/2006 13:46:12 PAGE 2

ASSEMBLY LISTING OF GENERATED OBJECT CODE


             ; FUNCTION main (BEGIN)
                                                ; SOURCE LINE # 5
                                                ; SOURCE LINE # 6
                                                ; SOURCE LINE # 8
0000 750000      R     MOV     a,#00H
0003 750005      R     MOV     a+01H,#05H
;---- Variable 'b' assigned to Register 'R2/R3' ----
0006 7B06              MOV     R3,#06H
0008 7A00              MOV     R2,#00H
                                                ; SOURCE LINE # 9
000A E500        R     MOV     A,a+01H
000C 2B                ADD     A,R3
000D F500        R     MOV     y+01H,A
000F EA                MOV     A,R2
0010 3500        R     ADDC    A,a
0012 F500        R     MOV     y,A
                                                ; SOURCE LINE # 10
0014 C3                CLR     C
0015 E500        R     MOV     A,a+01H
0017 9B                SUBB    A,R3
0018 F500        R     MOV     y+01H,A
001A E500        R     MOV     A,a
001C 9A                SUBB    A,R2
001D F500        R     MOV     y,A
                                                ; SOURCE LINE # 11
001F AE00        R     MOV     R6,a
0021 AF00        R     MOV     R7,a+01H
0023 7D06              MOV     R5,#06H
0025 7C00              MOV     R4,#00H
0027 120000      E     LCALL   ?C?IMUL
002A 8E00        R     MOV     y,R6
002C 8F00        R     MOV     y+01H,R7
                                                ; SOURCE LINE # 12
002E AE00        R     MOV     R6,a
0030 AF00        R     MOV     R7,a+01H
0032 120000      E     LCALL   ?C?UIDIV
0035 8E00        R     MOV     y,R6
0037 8F00        R     MOV     y+01H,R7
                                                ; SOURCE LINE # 13
0039 22                RET
             ; FUNCTION main (END)
```

```
Unterprogramm C?IMUL
;Input     (R6,R7)*(R4,R5) = A*B
; Product =  (256* Ahigh + Alow) * (256*Bhigh + Blow)
```

;Ausmultipliziert ergibt sich dann:

```
;       Produkt =     (256 * Ahigh * 256 * Bhigh) +      (4)
;                     (256 * Ahigh * Blow) +             (3)
;                     (256 * Bhigh * Alow) +             (2)
;                     (Alow * Blow)                      (1)
; Maximum Number in a variable of type WORD N=65535

C?IMUL:
     Mov  A,R7       ;Lower Byte Multiplication (1)
     Mov  B,R5
     Mul  AB
     Mov  R0,B       ;Save MSB of Result
     XCH  A,R7       ;Save LSB of Result to R7 and load LSB of A
                     ;R7 is just the LSB of the Multipl. result
     Mov  B,R4       ;Get MSB of B
     Mul  AB         ;(2)
     Add  A,R0       ;MSB of (1) + LSB of (2) = (5)
                     ;if Carry occur -> range overflow
                     ;Overflow will not be handled
     XCH  A,R6       ;Save LSB of (5) and get MSB of A
     Mov  B,R5       ;Get LSB of B
     Mul  AB         ;(3)
     Add  A,R6       ;(5)+LSB of (3)
                     ;if Carry occur -> range overflow
                     ;Overflow will not be handled
     Mov  R6,A       ;A contains MSB of result
     Ret
```

Bei der Division A/B =(256*G+H)/(256*D+E) werden drei Fälle unterschieden:

1. Beide High-Bytes sind 0-> Der im Prozessor vorhandene Divisionsbefehl kann genutzt werden.
   (256*0+H)/(256*0+E) = H/E
2. Das High Byte von B ist 0-> Für das High Byte von A kann der vorhandene Divisionsbefehl verwendet werden. Der Rest und das Low-Byte muss danach mit dem allgemein bekannten Divisionsalgorithmus verarbeitet werden.

   (256*G+H)/(0+E) = 256*G/E + H/E
   G/E=Q + R/E Q und R ergeben sich direkt aus einer 1 Byte Division
   (256*G+H)/E = 256*Q+256*R/E + H/E = 256*Q+(256*R + H)/E
3. Beide High Bytes sind ungleich 0->Das High Byte von Quotienten ist immer 0. Das Low-Byte muss mit dem bekannten Divisionsalgorithmus errechnet werden.
   Der minimale Wert von D wenn das HighByte von B ungleich 0 ist, ist 1 * 256.
   Der maximale Wert von A ist 256*256 -1. Damit ergibt sich bei der Division:
   (256*256-1)/256 = 256-(1/256)<256. Damit ist das höchstwertigste Byte des Quotienten immer kleiner 0.

Zur Anwendung des Standarddivisionsalgorithmus wird A über 3 Byte dargestellt. (R5,R6,R7) wobei R5 auf 0 initiiert wird. R6 enthält das höchstwertige Byte von A. R7 enthält das niederwertige Byte.

```
Unterprogramm C?UDIV
```

```
;Input    (R6,R7)/(R4,R5) = A/B
     CJNE      R4,#0,MSB_B_N0
     CJNE R6,#0,MSB_A_N0
     Mov  A,R7       ;Both MSB are zero
     Mov  B,R5       ;Perform 8 Bit Division
     Div  AB
     Mov  R7,A       ;R7 is quotient
     Mov  R5,B       ;R5 is Remainder
     Ret
MSB_B_N0:            ;MSB(B) !=0
     Clr  A
     XCH  A,R4       ;Get MSB(B) , R4=0
                     ;Make R4 the MMSB(A)
     Mov  R0,A       ;Save MSB(B) to R0
     Mov  B,08h           ;B=00001000         8 mal schieben
Loop1:
     Mov  A,R7       ; Get LSB(A)
     Add  A,R7       ; 2*LSB(A)
     Mov  R7,A       ; 2*LSB(A) ->R7
     Mov  A,R6       ; Get MSB(A)
     RLC  A          ; 2*MSB(A) + MSBit of 2*LSB(A)
     Mov  R6,A       ; 2*MSB(A) ->R6
     Mov  A,R4       ; Get MMSB(A)
     RLC  A          ; 2* MMSB(A)
     Mov  R4,A       ; 2* MMSB(A)->R4
     Mov  A,R6       ; Get 2*MSB(A)
     SUBB A,R5       ; 2*MSB(A) - LSB(B)
     Mov  A,R4       ; Get 2* MMSB(A)
     SUBB A,R0       ; 2* MMSB(A) - MSB(B)
     JC   M1         ; as long as B>A only shift

     Mov  R4,A       ; Save the reduced MMSB(A)
     Mov  A,R6       ; Recalculate MSB(A)
     SUBB A,R5
     Mov  R6,A       ; Save the reduced MSB(A)
     Inc  R7         ; Use R7 for storing the quotient, too
                     ; after 8 shifts the LSB(A) is removed
                     ; from R7 and the quotient was shifted
                     ; to the correct weight
M1:
     DJNZ B,Loop1
     Clr  A          ;Accu reset
     XCH  A,R6       ;R6->0,if both MSB are set,R6=0 in any case
     Mov  R5,A       ;Remainder to R5
     Ret




MSB_A_N0:            ;MSB(B)==0 ,MSB(A) != 0
     Mov  A,R5       ;Get LSB(B)
     Mov  R0,A       ;R0<=LSB(B)
     Mov  B,A        ;LSB(B) -> B Register
     Mov  A,R6       ;Get MSB(A)
     DIV  AB         ; MSB(A) / LSB(B)
     JB   OV,Ende    ; Overflow -> Ende
     Mov  R6,A       ; Quotient -> R6
     Mov  R5,B       ; Remainder -> R5
     Mov  B,08h           ; B=00001000
Loop2:
```

```
     Mov  A,R7       ;Get LSB (A)
     Add  A,R7       ;2*LSB (A)
     Mov  R7,A       ; 2*LSB (A)->R7
     Mov  A,R5       ;Get Remainder of MSB(A)
     RLC  A          ;2* Remainder of MSB(A) + MSBit of 2*LSB A
     Mov  R5,A       ; Save it->R5
     JC   M3         ;if carry it is surely in
     SUBB A,R0       ; 2*LSB B- LSB of B
     JNC  M2         jmp if it was in
     DJNZ B,Loop2    ;Try shifting as long as possible
     Ret             ;Rücksprung
M3:  CLR  C
     SUBB A,R0       ; 2*LSB B- LSB of B
M2:
     Mov  R5,A       ;Save remainder
     Inc  R7         ;Quotien calculation
     DJNZ B,Loop2    ;Try shifting as long as possible

Ende: Ret
```

Vergleicht man die Codelänge des Programms mit einfachen Bytes als Datentypen mit den jetzt verwendeten doppelt langen Typen, so steigt die Codelänge von 152 Bytes auf 560 Bytes.

8.1.3 Integer Verarbeitung

Im vorliegenden Fall werden Integerzahlen vom Typ int in 2 Bytes abgebildet. Eine Erweiterung auf 4 Bytes ist mit dem Typ long int möglich. Auch hier lassen sich Addition und Subtraktion einfach erweitern. Beginnend vom niederwertigsten Byte hin zum höchstwertigsten Byte werden die Operationen nacheinander ausgeführt und der Übertrag von der jeweils vorhergehenden Operation mit verwendet. Bei der Division und Multiplikation müssen zusätzlich noch die Vorzeichen berechnet werden und gegebenenfalls eine Betragsbildung erfolgen.
Der Vergleich der benötigten Codegrößen für die vier Standardoperationen zeigt den immer größer werdenden Aufwand bei der Verarbeitung dieser Datentypen. Werden die Operationen häufig verwendet, steigt der Aufwand im Programmcode nicht linear mit. Die Operationen werden als Unterprogramme definiert und der Aufruf erfolgt an den benötigten Stellen. Der Rechenaufwand bleibt jedoch erhalten.

Datentyp	**Codegröße in Byte**
BYTE	152
WORD	560
int	718
long int	1445

Beim vorliegenden Compiler werden vom Hersteller die Taktzyklen für 16 Bit Operationen und 32 Bit Operationen angegeben. Sie geben einen Eindruck der Ausführungszeiten [KEI05]:

16 Bit Operationen	**CPU**	**Routine**	**Min.**	**Avg.**	**Max.**
Multiplikation signed/unsigned	8051	IMUL	29	29	29
	80517	Intrinsic	17	17	17
Unsigned Division	8051	UIDIV	16	128	153
	80517	UIDIV517	22	22	22
Signed Division	8051	SIDIV	53	141	181
	80517	SIDIV517	35	52	60
32 Bit Operationen	**CPU**	**Routine**	**Min.**	**Avg.**	**Max.**
Multiplikation signed/unsigned	8051	LMUL	106	106	106
	80517	LMUL517	62	62	62
Unsigned Division	8051	SIDIV	227	497	650
	80517	SIDIV517	36	52	101
Signed Division	8051	SLDIV	267	564	709
	80517	SLDIV517	49	75	141

8.1.4 FLOAT Verarbeitung

Float Zahlen werden üblicherweise in Form einer Mantisse und eines Exponenten dargestellt:
Die Norm IEEE 754 definiert eine Standarddarstellung von binären Gleitkommazahlen in Computern. Diese Standarddarstellung findet auch bei der Umsetzung von C-Programmen mit float Datentypen ihre Anwendung.
Allgemein kann die Darstellung einer Gleitkommazahl, wie angegeben, beschrieben werden:

$x = s * m * b^{e}$

Vorzeichen	s	1 Bit
Exponent	e	r Bit
Mantisse	m	p Bit
Basis des Exp.	b	

S=0 bezeichnet üblicherweise positive Zahlen, s=1 markiert negative Zahlen. Der Exponent ergibt sich aus der in den Exponentenbits gespeicherten nichtnegativen Binärzahl E zur Basis b durch Subtraktion eines festen Grundwertes B:

$e=E-B$

Die Mantisse wird als normierte Gleitkommazahl in der angegebenen Form verstanden:

$m=1+M/2^{p}$

Die erste Eins braucht dann nicht abgespeichert zu werden, da hier immer eine Eins steht.
Bei float –Typen in C werden die Kenngrößen bei Mikrocontrollerapplikationen oft auf folgende Größen gesetzt:

Mantisse	23 Bit	p=23
Exponent	8 Bit	r=8
Biaswert		B=127
Vorzeichen	1 Bit	

Damit ergibt sich die kleinste Einheit in der Mantisse mit $2^{-23}=1.192*10^{-7}$

Der Wertebereich ist damit von:

$2^{-126}=1.175*10^{-38}$ bis $2^{128}=3.403*10^{38}$

Der Exponent -127 wird zur Anzeige von NaN (Not a Number) verwendet.

NaN	0xFFFFFFFF	Not a Number	
+INF	0x7F800000	Positiv Unendlich	(Positiver Überlauf)
-INF	0xFF800000	Negativ Unendlich	(Negativer Überlauf)

Die Werte werden gesetzt, wenn durch Null dividiert wird oder eine Zahlenbereichsüberschreitung vorliegt.

Werden die einzelnen Teile wie im nachfolgenden Bild gezeigt, angeordnet, so kann eine float Zahl wie eine Integerzahl verglichen werden.

V	E	E	E	E	E	E	E	E	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
0	7	6	5	4	3	2	1	0	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
	Exponent								Mantisse																						
	Vorzeichen																														

Beim Vergleich des benötigten Codes mit den bereits bekannten Basisoperationen ergibt sich eine signifikante Zunahme des benötigten Programmspeichers.

```
//File Float.c
#include <reg51.h>                    /* define 8051 registers */
typedef unsigned int WORD;
void main()
{
        float a, b, y;
        a=722.6; b=0.258;
        y=a+b;
        y=a-b;
        y=a*b;
        y=a/b;
}
```

Dies erklärt sich durch die getrennte Betrachtung von Mantisse und Exponent sowie zusätzlichen Normalisierungsmaßnahmen nach den arithmetischen Operationen um die Mantisse wieder auf die gewünschte Form $m=1+M/2^{p}$ zu bringen.

Datentyp	Codegröße in Byte
BYTE	152
WORD	560
int	718
long int	1445
float	2590

Neben den Grundrechenarten, stehen je nach Compiler auch mathematische Operationen zur Verfügung und können durch eine entsprechende Include-Datei bekannt gemacht werden (z.B. <math.h>. Zur Verfügung stehen dann beispielsweise sinus, cosinus, tangens, arcsinus, arccosinus, arctangens, exponential, logarithmus oder Quadratwurzel. Die Ausführungszeiten für ein Programm steigen dabei erheblich. In der folgenden Tabelle sind exemplarisch, die vom Compilerhersteller abgegebenen Taktzyklen genannt [KEI05]:

Operation	CPU	Routine	Min.	Avg.	Max.
Addition	8051	FPADD	8	107	202
	80517	FPADD517	8	107	202
Subtraktion	8051	FPSUB	11	113	214
	80517	FPSUB517	11	113	214
Multiplikation	8051	FPMUL	13	114	198
	80517	FPMUL517	13	86	141
Division	8051	FPDIV	48	687	999
	80517	FPDIV517	48	165	209
Sinus	8051	SIN	1565	2928	3476
	80517	SIN517	1422	2519	3048
Cosinus	8051	COS	1601	2921	3665
	80517	COS517	1458	2514	3180
arcsinus	8051	ASIN	912	6991	8554
	80517	ASIN	912	3984	4717
arccosinus	8051	ACOS	796	7578	8579
	80517	ACOS517	796	4255	4871

8.2 Einsatz von Pointern

Pointer spielen eine wichtige Rolle bei der Programmierung in Mikrocontrolleranwendungen. Felder können mit einem Adresszähler elegant verarbeitet werden. Des Weiteren ist der Zugriff auf bestimmte Speicherbereiche nur über Verweise möglich. Da die Ressourcen in einem Mikrocontroller jedoch begrenzt sind, ist von einem extensiven Einsatz von Feldern abzusehen. Insbesondere ist der RAM-Bereich in dem die Felder abgelegt werden können, sehr unterschiedlich dimensioniert. So stehen beim Prozessortyp 8051 je nach Version im internen RAM nur 128 bzw. 256 Speicherplätze (Data-, IData-Bereich) überhaupt zur Verfügung. Auch der externe Speicher mit seinem nominal 64 KByte Speichergröße steht eigentlich nur in der vollen Ausbaustufe bereit.

8.2.1 Pointer auf Byte

Sollen Pointer auf den einfachsten und damit auch direkt abbildbaren Datentyp (Byte) aufgebaut werden, so können in C folgende Definition vorgenommen werden:

Typdefinition zur einfacheren Notation: **typedef unsigned char BYTE;**
Definition einer Variable auf die verwiesen werden soll: **BYTE v;**
Aufbau eines Pointers **BYTE *ptv;**

Zur besseren Kennzeichnung und zur Steuerung des Compilers ist es sinnvoll, den gewünschten Datenbereich anzugeben. Werden keine Angaben gemacht, wird zum Teil allgemeiner und damit aufwendiger Code für die Verarbeitung von Pointern generiert.

Ein Feld vom Typ Byte wird automatisch als Verweis auf das erste Element gehandhabt:

```
typedef unsigned char BYTE;

void main()
{
        BYTE data   ad[10];
        BYTE data   *ptad;
        BYTE data   e;
        ad[0]=0;
        ad[2]=2;
        ptad=ad;            //gibt die Adresse des ersten Elementes an
        e= * ptad;          //holt das erste Element
        e= *(ptad +2);      //holt das dritte Element
}
```

```
C51COMPILER V6.01 PDATA                 04/26/2006 21:10:42 PAGE 2

ASSEMBLY LISTING OF GENERATED OBJECT CODE


             ; FUNCTION main (BEGIN)
                                                ; SOURCE LINE # 5
                                                ; SOURCE LINE # 6
; ad[0]=0                                       ; SOURCE LINE # 10
0000 E4                 CLR     A
0001 F500         R     MOV     ad,A
; ad[2]=2                                       ; SOURCE LINE # 11
0003 750002       R     MOV     ad+02H,#02H
; v=ptad                                        ; SOURCE LINE # 13
;---- Variable 'ptad' assigned to Register 'R0' ----
0006 7800         R     MOV     R0,#LOW ad
; e= *ptad                                      ; SOURCE LINE # 14
0008 E6                 MOV     A,@R0
;---- Variable 'e' assigned to Register 'R7' ----
; e= *(ptad+2)                                   ; SOURCE LINE # 15
0009 E8                 MOV     A,R0
000A 2402               ADD     A,#02H
000C F8                 MOV     R0,A
000D E6                 MOV     A,@R0
000E FF                 MOV     R7,A
                                                ; SOURCE LINE # 17
000F 22                 RET
             ; FUNCTION main (END)

C51 COMPILER V6.01 PDATA             04/26/2006 21:10:42 PAGE 3

main . . . . . .  PUBLIC   CODE   PROC     0000H  --
  ad . . . . . .  AUTO     DATA   ARRAY    0000H  10
  ptad . . . . .  * REG *  DATA   PTR      0000H  1
  e. . . . . . .  * REG *  DATA   U_CHAR   0007H  1

MODULE INFORMATION:   STATIC OVERLAYABLE
   CODE SIZE        =     16    ----
   CONSTANT SIZE    =   ----    ----
   XDATA SIZE       =   ----    ----
   PDATA SIZE       =   ----    ----
   DATA SIZE        =   ----      10
   IDATA SIZE       =   ----    ----
   BIT SIZE         =   ----    ----
END OF MODULE INFORMATION.
```

Zugriff auf den externen Speicher:

Die Verwendung der Direktive xdata zwingt den Compiler zum Einsatz des Datenpointers (DPTR) zum Zugriff auf die Daten im externen RAM.
Die Wahl des Datenbereiches wurde dabei dem Compiler überlassen. Durch entsprechendes Setzen der Datenpointer kann der Zugriff auf gewünschte Adressbereiche bestimmt werden.

```
stmt level    source

   1          //File xdata.c
   2          #include <reg51.h>    /* define 8051 registers */
   3
   4          typedef unsigned char BYTE;
   5          void main()
   6          {
   7    1      BYTE xdata     ax[10];
   8    1      BYTE xdata     *ptax;
   9    1      BYTE data e;
  10    1
  11    1      ax[0]=0;
  12    1      ax[2]=2;
  13    1      ptax=ax;  //gibt die Adr. des ersten Elementes an
  14    1      e= *ptax;      //holt das erste Element
  15    1      e= *(ptax+2);  //holt das dritte Element
  16    1      }
```

```
C51 COMPILER V6.01 PXDATA                04/26/2006 21:31:10 PAGE
2
ASSEMBLY LISTING OF GENERATED OBJECT CODE
             ; FUNCTION main (BEGIN)
                                                  ; SOURCE LINE # 5
                                                  ; SOURCE LINE # 6
                                                  ; SOURCE LINE # 11
0000 E4                  CLR     A
0001 900000        R     MOV     DPTR,#ax
0004 F0                  MOVX    @DPTR,A
                                                  ; SOURCE LINE # 12
0005 900000        R     MOV     DPTR,#ax+02H
0008 7402                MOV     A,#02H
000A F0                  MOVX    @DPTR,A
                                                  ; SOURCE LINE # 13
;---- Variable 'ptax' assigned to Register 'DPTR' ----
000B 900000        R     MOV     DPTR,#ax
                                                  ; SOURCE LINE # 14
000E E0                  MOVX    A,@DPTR
;---- Variable 'e' assigned to Register 'R7' ----
                                                  ; SOURCE LINE # 15
000F A3                  INC     DPTR
0010 A3                  INC     DPTR
0011 E0                  MOVX    A,@DPTR
0012 FF                  MOV     R7,A
                                                  ; SOURCE LINE # 16
0013 22                  RET
             ; FUNCTION main (END)

C51 COMPILER V6.01 PXDATA                04/26/2006 21:31:10 PAGE 3

NAME                                     CLASS   MSPACE  TYPE    OFFSET  SIZE
main . . . . . .                         PUBLIC   CODE   PROC      0000H  ---
  ax . . . . . .                         AUTO     XDATA  ARRAY     0000H  10
  ptax . . . . . .                       * REG *  DATA   PTR       0000H  2
  e. . . . . .                           * REG *  DATA   U_CHAR    0007H  1

MODULE INFORMATION:   STATIC OVERLAYABLE
   CODE SIZE        =     20    ----
   CONSTANT SIZE    =   ----    ----
   XDATA SIZE       =   ----      10
   PDATA SIZE       =   ----    ----
   DATA SIZE        =   ----    ----
   IDATA SIZE       =   ----    ----
   BIT SIZE         =   ----    ----
END OF MODULE INFORMATION.
```

Einstellung von selbstgewählten Adressen:

```
stmt level    source

  1       //File Pxdata2.c
  2       #include <reg51.h>                    /* define 8051 registers */
  3
  4       typedef unsigned char BYTE;
  5       void main()
  6       {
  7  1        BYTE xdata  *ptax=0x2040;
  8  1        BYTE data    e;
  9  1
 10  1        ptax[0]=1;            //setzt das erste Element
 11  1        ptax[2]=2;            //setzt das dritte Element
 12  1        e= *ptax;             //holt das erste Element
 13  1        e= *(ptax+2);//holt das dritte Element
 14  1     }

C51 COMPILER V6.01 XDATA2                                        04/26/2006
21:58:35 PAGE 2

ASSEMBLY LISTING OF GENERATED OBJECT CODE


             ; FUNCTION main (BEGIN)
                                          ; SOURCE LINE # 5
                                          ; SOURCE LINE # 6
                                          ; SOURCE LINE # 7
;---- Variable 'ptax' assigned to Register 'R6/R7' ----
0000 7F40              MOV     R7,#040H
0002 7E20              MOV     R6,#020H
                                          ; SOURCE LINE # 10
0004 8F82              MOV     DPL,R7
0006 8E83              MOV     DPH,R6
0008 7401              MOV     A,#01H
000A F0                MOVX    @DPTR,A
                                          ; SOURCE LINE # 11
000B A3                INC     DPTR
000C A3                INC     DPTR
000D 04                INC     A
000E F0                MOVX    @DPTR,A
                                          ; SOURCE LINE # 12
000F 8F82              MOV     DPL,R7
0011 8E83              MOV     DPH,R6
0013 E0                MOVX    A,@DPTR
;---- Variable 'e' assigned to Register 'R5' ----
                                          ; SOURCE LINE # 13
0014 A3                INC     DPTR
0015 A3                INC     DPTR
0016 E0                MOVX    A,@DPTR
0017 FD                MOV     R5,A
                                          ; SOURCE LINE # 14
0018 22                RET
             ; FUNCTION main (END)
```

Wurden keine Hinweise auf den zu benutzenden Speicherbereich gegeben, so werden allgemeine Pointer erzeugt (Generic Pointer). Zur Abbildung aller Möglichkeiten werden 3 Byte benötigt.
Das erste Byte enthält eine Kennung, welcher Speicherbereich gemeint ist. Die folgenden Bytes ergeben die Adressdaten.

Diese Form ist zwar sehr allgemein, erfordert aber entsprechend umfangreichen Code und auch mehr Speicherplatz.

Speicher Type	Kennung +0	+1	+2
idata data bdata	0x00	0	Adresse
xdata	0x01	MSB Adresse	LSB Adresse
pdata	0xFE	MSB Adresse	LSB Adresse
code	0xFF	MSB Adresse	LSB Adresse

8.2.2 Verwendung anderer Datentypen

Werden Datentypen mit mehr als einem Byte verwendet, ist es notwendig die einzelnen Bytes nacheinander abzulegen und bei der Adressberechnung entsprechend anzupassen.

Das folgende Beispiel zeigt am Datentyp int wie ein solcher Zugriff realisiert wird.

```
   1          //File Pxdataint.c
   2          #include <reg51.h>   /* define 8051 registers */
   3
   4          void main()
   5          {
   6   1       int xdata *ptax=0x2040;
   7   1       int data  e;
   8   1
   9   1       ptax[0]=1;          //setzt das erste Element
  10   1       ptax[2]=2;          //setzt das dritte Element
  11   1       e= *ptax;      //holt das erste Element
  12   1       e= *(ptax+2);  //holt das dritte Element
  13   1      }
C51 COMPILER V6.01 XDATAINT     04/26/2006 22:05:49 PAGE 2
ASSEMBLY LISTING OF GENERATED OBJECT CODE
;---- Variable 'ptax' assigned to Register 'R6/R7' ----
0000 7F40              MOV     R7,#040H
0002 7E20              MOV     R6,#020H
                                           ; SOURCE LINE # 9
0004 8F82              MOV     DPL,R7
0006 8E83              MOV     DPH,R6
0008 E4                CLR     A
0009 F0                MOVX    @DPTR,A
000A A3                INC     DPTR
000B 04                INC     A
000C F0                MOVX    @DPTR,A
                                           ; SOURCE LINE # 10
000D EF                MOV     A,R7
000E 2404              ADD     A,#04H
0010 F582              MOV     DPL,A
0012 E4                CLR     A
0013 3E                ADDC    A,R6
0014 F583              MOV     DPH,A
0016 E4                CLR     A
0017 F0                MOVX    @DPTR,A
0018 A3                INC     DPTR
0019 7402              MOV     A,#02H
001B F0                MOVX    @DPTR,A
                                           ; SOURCE LINE # 11
001C 8F82              MOV     DPL,R7
001E 8E83              MOV     DPH,R6
0020 E0                MOVX    A,@DPTR
0021 FC                MOV     R4,A
0022 A3                INC     DPTR
0023 E0                MOVX    A,@DPTR
0024 FD                MOV     R5,A
;---- Variable 'e' assigned to Register 'R4/R5' ----
                                           ; SOURCE LINE # 12
0025 EF                MOV     A,R7
0026 2404              ADD     A,#04H
0028 F582              MOV     DPL,A
002A E4                CLR     A
002B 3E                ADDC    A,R6
002C F583              MOV     DPH,A
002E E0                MOVX    A,@DPTR
002F FC                MOV     R4,A
0030 A3                INC     DPTR
0031 E0                MOVX    A,@DPTR
0032 FD                MOV     R5,A
0033 22                RET
```

8.3 STRING Verarbeitung

Zur Ausgabe von Meldungen oder die Zusammenstellung von Daten zur Nachrichtenübertragung werden häufig Zeichenketten verwendet. In der Programmiersprache C sind dazu Verweise auf den Datentyp char vorgesehen. Diese Datentypen sind sehr effizient in Ketten von 8 Bitdaten umzusetzen.

Stringbehandlungsroutinen sind nicht zwangsläufig Bestandteil des Lieferumfanges eines C-Compilers zur Erzeugung von Mikrocontrollerprogrammen.
Im Internet kann jedoch der Code der Standardfunktion für die Stringbehandlung gefunden und unter Beachtung der Copyright Bedingungen verwendet werden. Eine Auswahl der Funktionen, die auf die vorliegende Anwendung zugeschnitten ist, reduziert den Programmcode unter Umständen erheblich.
Ebenfalls kann es möglich sein, dass durch den eigenen Einbau dieser Funktionen im Gegensatz zu Compilerfunktionen eine kleinere Codegröße erreicht wird.

Die folgenden Beispiele zeigen den Einsatz der Stringverarbeitungsfunktionen strcpy zur Belegung eines Speicherplatzes mit Zeichen und der anschließenden Erweiterung des String um eine weitere Zeichenkette mit strcat.
Im ersten Fall wurden die direkt einbindbaren Funktionen verwendet (include string.h). Im zweiten Falle wurden Funktionen aus [STR11] angegebenen Stringbehandlungsfunktionen eingesetzt. Zur Vermeidung von Überschneidungen mit Namen der Stringfunktionen innerhalb der Compilerumgebung wurden die Funktionen mit dem Prefix m_ versehen.

Programm mit Stringfunktionen, die direkt vom Compiler zur Verfügung gestellt wurden.

```
#include <reg517.h>
#include <string.h> //Bekanntmachung der Stringfunktionen
void main ()
{
        char xdata *buf;
        buf=0x0;
        strcpy(buf,"Das ist ein Test");
        strcat(buf," Zusatz");
}
```

Programm mit eigenen Stringbehandlungsfunktionen:

```
#include <c8051F340.h>
char *m_strcpy(char *dst, const char *src)
{
  char *cp = dst;
  while (*cp++ = *src++);
  return dst;
}
char *m_strcat(char *dst, const char *src)
{
  char *cp = dst;
  while (*cp) cp++;
  while (*cp++ = *src++);
  return dst;
}

void main()
{
        char xdata *buf=0x0;
        m_strcpy(buf,"Das ist ein Test");
        m_strcat(buf," Zusatz");
}
```

Der Vergleich der benötigten Speicherplätze für den Code zeigt den Unterschied:

Verfahren	**Codegröße**
Verwendung von Compilerfunktionen	1313 Byte
Eigene Stringbehandlung	841 Byte
Eigene Stringbehandlung ohne return	754 Byte
Stringbehandlung ohne return mit Zusatz DPTR	594 Byte
Eigene Stringbehandlung ohne return mit Zusatz DPTR und Angabe der Speicherbereiche bei den Parametern	574 Byte

Durch die Verwendung von Bytes als Basistypen können ähnlich den Stringbehandlungsfunktionen zusätzlich noch Speicherbereichskopierfunktionen mit angegeben werden. Da hier jedoch nicht vorausgesetzt werden kann, dass eine Endekennung existiert, ist hier die Angabe einer Länge notwendig.

Beispiele hierzu sind die Funktionen: memcpy oder memset …

Die Funktionen sind in [STR11] bei den Stringbehandlungsfunktionen mit angegeben.

z.B. memcpy

```
void *memcpy(void *dst, const void *src, size_t n)
{
  void *ret = dst;

  while (n--)
  {
    *(char *)dst = *(char *)src;
    dst = (char *) dst + 1;
    src = (char *) src + 1;
  }

  return ret;
}
```

Wird die angegebene Funktion ohne return in einem Hauptprogramm aufgerufen, wird dafür eine Codegröße von 499 Bytes benötigt. Zur Vermeidung von Problemen, mit den im Compiler vorhandenen Funktionen, wurde die Funktion in m_memcpy umbenannt.

```
void m_memcpy(void *dst, const void *src, unsigned char n)
{
  void *ret = dst;
  while (n--){
    *(char *)dst = *(char *)src;
    dst = (char *) dst + 1;
    src = (char *) src + 1;
  }
}
void main ()
{
        char xdata *buf1;
        char xdata *buf2;
        m_memcpy(buf1,buf2,10);
}
```

Wird der Code direkt in das Hauptprogramm kopiert, ergibt sich eine Codegröße von 203 Byte anstatt 499 Bytes.

```
void main ()
{
        char xdata *buf1;
        char xdata *buf2;
        char data n=10;
        while (n--){
           *buf1 = *buf2;
           buf1++;
           buf2++;
         }
}
```

8.4 Schleifenkonstruktionen

Durch die Forderungen der strukturierten Programmierungsmethode, die in C umgesetzt wurde, sind zunächst nur zwei prinzipielle Formen von Schleifen zugelassen:

- While Schleife
- Do … While Schleife

Die For Schleife kann als Erweiterung der While Schleife angesehen werden, bei der Initialisierung, Fortschaltung und Endabfrage in einem Statement integriert sind.

8.4.1 While Schleifen

While Schleifen werten eine Bedingung aus, von der abhängig die Schleife abgebrochen oder weitergeführt wird. Die Komplexität der Abfrage ist jedoch im Wesentlichen durch die verwendeten Datentypen bestimmt. Die Schleife selbst wird in Assembler durch entsprechende bedingte und unbedingte Sprünge realisiert. Eigene Schleifenkonstrukte sind hier nur rudimentär vorhanden z.B. der DJNZ Befehl und müssen entsprechend eingepasst werden.

Beispiel mit while-Schleifen mit unterschiedlichen Datentypen:

```
stmt level    source

   1          //File while.c
   2          #include <reg51.h>  /* define 8051 registers */
   3          typedef unsigned char BYTE;
   4          void main()
   5          {
   6   1       BYTE bi=100;
   7   1       int ii=-100;
   8   1       while(bi>0)  bi--;
   9   1       while(ii<=0) ii++;
  10   1
  11   1      }
```

```
C51 COMPILER V6.01 WHILE                          04/25/2006 21:31:55 PAGE 2

ASSEMBLY LISTING OF GENERATED OBJECT CODE


             ; FUNCTION main (BEGIN)
                                             ; SOURCE LINE # 4
                                             ; SOURCE LINE # 5
                                             ; SOURCE LINE # 6
;---- Variable 'bi' assigned to Register 'R7' ----
0000 7F64              MOV     R7,#064H
                                             ; SOURCE LINE # 7
;---- Variable 'ii' assigned to Register 'R4/R5' ----
0002 7D9C              MOV     R5,#09CH
0004 7CFF              MOV     R4,#0FFH
0006         ?C0001:
                                             ; SOURCE LINE # 8
0006 DFFE              DJNZ    R7,?C0001
0008         ?C0003:
                                             ; SOURCE LINE # 9
0008 D3                SETB    C
0009 ED                MOV     A,R5
000A 9400              SUBB    A,#00H
000C EC                MOV     A,R4
000D 6480              XRL     A,#080H
000F 9480              SUBB    A,#080H
0011 5007              JNC     ?C0005
0013 0D                INC     R5
0014 BD0001            CJNE    R5,#00H,?C0006
0017 0C                INC     R4
0018         ?C0006:
0018 80EE              SJMP    ?C0003
                                             ; SOURCE LINE # 11
001A         ?C0005:
001A 22                RET
             ; FUNCTION main (END)

C51 COMPILER V6.01 WHILE                          04/25/2006 21:31:55 PAGE 3

NAME                                    CLASS   MSPACE  TYPE    OFFSET
SIZE
====                                    =====   ======  ====    ======
====


main . . . . . . . . . . . . . . . . .  PUBLIC   CODE   PROC     0000H  ---
--
  bi . . . . . . . . . . . . . . . . .  * REG *  DATA   U_CHAR   0007H  1
  ii . . . . . . . . . . . . . . . . .  * REG *  DATA   INT      0004H  2

MODULE INFORMATION:   STATIC OVERLAYABLE
   CODE SIZE        =     27    ----
   CONSTANT SIZE    =   ----    ----
   XDATA SIZE       =   ----    ----
   PDATA SIZE       =   ----    ----
   DATA SIZE        =   ----    ----
   IDATA SIZE       =   ----    ----
   BIT SIZE         =   ----    ----
END OF MODULE INFORMATION.
```

8.4.2 Do…While Schleifen

Bei der Konstruktion von Do…While Schleifen treten keine grundsätzlich anderen Probleme auf. Der Programmcode muss etwas anders angeordnet werden, da die Schleife nun mindestens einmal durchlaufen werden muss, bis die Schleifen Abfrage erfolgt. Das bereits aus dem vorhergehenden Kapitel bekannte Beispiel wurde hier nochmals mit Do-While Schleifen umgesetzt. Auch am generierten Code können die Ähnlichkeiten erkannt werden.

```
stmt level    source

   1          //File Repeat.c
   2          #include <reg51.h>   /* define 8051 registers */
   3          typedef unsigned char BYTE;
   4          void main()
   5          {
   6   1       BYTE bi=100;
   7   1       int ii=-100;
   8   1       do bi--; while(bi>0);
   9   1       do ii++; while(ii<=0);
  10   1
  11   1      }
```

```
C51 COMPILER V6.01 REPEAT                    04/25/2006 21:35:24 PAGE 2

ASSEMBLY LISTING OF GENERATED OBJECT CODE
             ; FUNCTION main (BEGIN)
                                             ; SOURCE LINE # 4
                                             ; SOURCE LINE # 5
                                             ; SOURCE LINE # 6
;---- Variable 'bi' assigned to Register 'R7' ----
0000 7F64               MOV     R7,#064H
                                             ; SOURCE LINE # 7
;---- Variable 'ii' assigned to Register 'R4/R5' ----
0002 7D9C               MOV     R5,#09CH
0004 7CFF               MOV     R4,#0FFH
0006         ?C0003:
                                             ; SOURCE LINE # 8
0006 1F                 DEC     R7
0007 EF                 MOV     A,R7
0008 D3                 SETB    C
0009 9400               SUBB    A,#00H
000B 50F9               JNC     ?C0003
000D         ?C0006:
                                             ; SOURCE LINE # 9
000D 0D                 INC     R5
000E BD0001             CJNE    R5,#00H,?C0008
0011 0C                 INC     R4
0012         ?C0008:
0012 D3                 SETB    C
0013 ED                 MOV     A,R5
0014 9400               SUBB    A,#00H
0016 EC                 MOV     A,R4
0017 6480               XRL     A,#080H
0019 9480               SUBB    A,#080H
001B 40F0               JC      ?C0006
                                             ; SOURCE LINE # 11
001D 22                 RET
             ; FUNCTION main (END)
main . . . . . .  PUBLIC   CODE   PROC     0000H  ---
--
  bi . . . . . .  * REG *  DATA   U_CHAR   0007H  1
  ii . . . . . .  * REG *  DATA   INT      0004H  2


MODULE INFORMATION:   STATIC OVERLAYABLE
   CODE SIZE        =     30    ----
   CONSTANT SIZE    =   ----    ----
   XDATA SIZE       =   ----    ----
   PDATA SIZE       =   ----    ----
   DATA SIZE        =   ----    ----
   IDATA SIZE       =   ----    ----
   BIT SIZE         =   ----    ----
END OF MODULE INFORMATION.
```

8.4.3 For Schleifen

In der Einleitung wurde bereits erwähnt, dass For Schleifen in ihrer Struktur erweiterte While Schleifen darstellen. In C können die Initiierung, die Endabfrage und die Weiterschaltung in einer gemeinsamen Klammer angegeben werden. Der Schleifenkörper muss entsprechend in die Kontrollbefehle eingepasst werden. Das folgende Beispiel zeigt die Anwendung von For Schleifen. Bei der zweiten Schleife wurde der Initialisierungsteil leer gelassen. Die Initialisierung der Schleifenvariablen erfolgt bei ihrer Definition.

```
stmt level    source

   1          //File for.c
   2          #include <reg51.h>
/* define 8051 registers */
   3          typedef unsigned char BYTE;
   4          void main()
   5          {
   6   1       BYTE bi;
   7   1       int ii=-100;
   8   1       int zz=0;
   9   1       for(bi=100; bi>=0; bi--)zz++;
  10   1       zz=0;
  11   1       for(;ii<=0;ii++)zz++ ;
  12   1
  13   1      }
```

```
C51 COMPILER V6.01 FOR
04/25/2006 21:43:33 PAGE 2

ASSEMBLY LISTING OF GENERATED OBJECT CODE


             ; FUNCTION main (BEGIN)
                                               ; SOURCE LINE # 4
                                               ; SOURCE LINE # 5
                                               ; SOURCE LINE # 7
;---- Variable 'ii' assigned to Register 'R6/R7' ----
0000 7F9C                MOV     R7,#09CH
0002 7EFF                MOV     R6,#0FFH
                                               ; SOURCE LINE # 8
;---- Variable 'zz' assigned to Register 'R4/R5' ----
0004 E4                  CLR     A
0005 FD                  MOV     R5,A
0006 FC                  MOV     R4,A
                                               ; SOURCE LINE # 9
;---- Variable 'bi' assigned to Register 'R3' ----
0007 7B64                MOV     R3,#064H
0009           ?C0001:
0009 0D                  INC     R5
000A BD0001              CJNE    R5,#00H,?C0008
000D 0C                  INC     R4
000E           ?C0008:
000E 1B                  DEC     R3
000F EB                  MOV     A,R3
0010 C3                  CLR     C
0011 9400                SUBB    A,#00H
0013 50F4                JNC     ?C0001
0015           ?C0002:
                                               ; SOURCE LINE # 10
0015 E4                  CLR     A
0016 FC                  MOV     R4,A
0017 FD                  MOV     R5,A
                                               ; SOURCE LINE # 11
0018           ?C0004:
0018 D3                  SETB    C
0019 EF                  MOV     A,R7
001A 9400                SUBB    A,#00H
001C EE                  MOV     A,R6
001D 6480                XRL     A,#080H
001F 9480                SUBB    A,#080H
0021 500C                JNC     ?C0007
0023 0D                  INC     R5
0024 BD0001              CJNE    R5,#00H,?C0009
0027 0C                  INC     R4
0028           ?C0009:
0028 0F                  INC     R7
0029 BF0001              CJNE    R7,#00H,?C0010
002C 0E                  INC     R6
002D           ?C0010:
002D 80E9                SJMP    ?C0004
                                               ; SOURCE LINE # 13
002F           ?C0007:
002F 22                  RET
             ; FUNCTION main (END)
```

```
C51 COMPILER V6.01 FOR
04/25/2006 21:43:33 PAGE 3

main . . . . . . . . . . . . . . . . . .  PUBLIC   CODE   PROC     0000H  ---
--
  bi . . . . . . . . . . . . . . . . . .  * REG *  DATA   U_CHAR   0003H  1
  ii . . . . . . . . . . . . . . . . . .  * REG *  DATA   INT      0006H  2
  zz . . . . . . . . . . . . . . . . . .  * REG *  DATA   INT      0004H  2


MODULE INFORMATION:   STATIC OVERLAYABLE
   CODE SIZE        =     48    ----
   CONSTANT SIZE    =   ----    ----
   XDATA SIZE       =   ----    ----
   PDATA SIZE       =   ----    ----
   DATA SIZE        =   ----    ----
   IDATA SIZE       =   ----    ----
   BIT SIZE         =   ----    ----
END OF MODULE INFORMATION.
```

8.4.4 Unterprogramme

Bei der Programmierung von Mikrocontrollern dienen Unterprogramme nicht nur zur Gruppierung von Befehlen, sondern auch zur Anbindung von Abarbeitungsfolgen an bestimmte Ereignisse wie Interrupts. Diese Anbindung erfordert eine besondere Kennzeichnung des Unterprogramms, verbunden mit der Forderung an ein ganz bestimmtes Aussehen. Interrupt Service Routinen dürfen beispielweise keine Rückgabewerte produzieren und müssen eine leere Parameterliste aufweisen.
Werden Unterprogramme von mehreren Stellen aus verwendet und ist eine gleichzeitige Verwendung nicht vom Programmablauf ausgeschlossen, so wird meist noch die Wiedereintrittsfähigkeit (reentrant) gefordert. Das bedeutet, dass alle Variablen für den Aufruf des Unterprogramms speziell für einen aktuellen Aufruf zur Verfügung stehen. Die Verwendung von beispielsweise globalen Variablen führt unweigerlich zu Datenüberschreibungen und damit in der Konsequenz zu falschen Ergebnissen. Das Gleiche gilt für Verweise auf Speicherbereiche, die damit gleichzeitig benutzt werden. Maßnahmen hierzu sind Kennzeichnungen gemeinsamer Bereiche, die damit gesperrt werden. Hierbei muss jedoch eine besondere Sorgfalt verwendet werden, damit kein wechselseitiges Aussperren erfolgt und die Prozesse sich gegenseitig blockieren (Dead Lock).
Die Wiedereintrittsfähigkeit kann auch durch den zusätzlichen Einsatz lokaler Variablen erreicht werden, da hier die Daten dann exklusive für den aktuellen Aufruf erzeugt werden. Bei Mikrocontrollern sind solche Maßnahmen jedoch immer im Hinblick auf den benötigten Speicherplatz durchzuführen.

Je nach Arbeitsweise des C-Compilers müssen jedoch noch andere Gegebenheiten beachtet werden. Der Keil C51 Compiler verwendet zur Übergabe von Parametern fest vorgegebene Register. Es können damit 3 Funktionsargumente sehr effizient übergeben werden. Dieser Übergabemechanismus führt jedoch zwangläufig zu der Tatsache, dass Unterprogramme, die auf diese Art mit Werten versorgt werden, nicht wiedereintrittsfähig sind. Dies muss durch ein entsprechendes zusätzliches Schlüsselwort erzwungen werden (reentrant).

Verwendung der Register bei der Übergabe von Werten:

Argument Nummer	char, 1 Byte Ptr.	Int, 2 Byte Ptr.	Long, float	Generic ptr
1	R7	R6 & R7	R4-R7	R1-R3
2	R5	R4 & R5	R4-R7	R1-R3
3	R3	R2 & R3		R1-R3

Sind keine Register verfügbar, werden feste Speicherpositionen verwendet.
(Direktive REGPARAMS, NOREGPARAMS).
Auch für die Rückgabe gibt es eine Konvention für die Register

Rückgabetyp	Register	Bedeutung
bit	Carry Flag	
char, u_char, 1 Byte Ptr.	R7	
int, u_int, 2 Byte Ptr	(R6 & R7)	(MSB & LSB)
long, u_long	(R4 - R7)	(MSB in R4 LSB in R7)
float	(R4 - R7)	32 Bit IEEE format
Generic ptr	(R1 – R3)	MType in R3, MSB R2, LSB R1

Zur Anpassung der Funktion an bestimmte Anwendungszwecke z.B. Interrupts, Speichermodelle können zusätzlich Angaben bei der Beschreibung der Funktion gemacht werden:

[SpeicherModell][reentrant][interrupt n][using n]

Es kann mehr als eine Option angegeben werden.

Das Speichermodell kann 3 Typen bezeichnen:
- small
- compact
- large

Small-Speichermodell

Wird keine Angabe gemacht, gilt das Small-Speichermodell. Als Bereiche für Variablen und den Stack wird der interne RAM (Data-, IData-Bereich) verwendet. Es ist das effizienteste Speichermodell, aber auch das Kleinste, was den verfügbaren Speicherbereich angeht.

Compact-Speichermodell

Dieses Speichermodell verwendet 256 Bytes im externen Ram. Dazu muss der Port 2 entsprechend mit einer Adresse versorgt werden, wenn noch andere Zugriffe auf den externen Ram erfolgen sollen. Der Zugriff erfolgt über @R0 und @R1. Er ist weniger effizient als der Zugriff beim Small-Speichermodell aber besser als beim Large-Speichermodell.

Large Speichermodell

Der Stack und die Variablen werden hier im externen Speicher abgelegt. Der Datenpointer DPTR wird hierbei zur Adressierung verwendet und stellt damit die ineffizienteste Methode des Datenzugriffes dar.

Den einzelnen Funktionen können unterschiedliche Speichermodelle zugewiesen werden.

Beispiel:

```
int mtest(int i, int y)//small model is the default
{
        return(i*y*y) ;
}
int large_func(int i int k) large       //large model is enforced
{
        return(mtest(I,k)+2);
}
```

Soll eine Funktion rekursiv aufgerufen werden oder kann sie z.B. bei Interrupts von mehreren Stellen aus aufgerufen werden, ohne das Datenüberschneidungen stattfinden sollen, so wird das Schlüsselwort „reentrant“ angegeben.
Bei Angabe des Speichermodells small wird der zusätzlich benötigte Stack im Idata Bereich aufgebaut. Wird compact oder large angegeben, so wird der Stack im externen Ram angelegt. Bei compact stehen 256 Byte zur Verfügung, bei large prinzipiell der gesamte externe Ram.

Am Beispiel der bereits bekannten Funktion m_memcpy kann der Unterschied im Assemblercode gesehen werden.

Im ersten Fall wird die Funktion ohne den Qualifier reentrant definiert und aufgerufen:

```
//File memcpynore.c
#include <reg517A.h>

void m_memcpy(void *dst, const void *src, unsigned char n)
{
        while (n--)
        {
        *(char *)dst = *(char *)src;
        dst = (char *) dst + 1;
        src = (char *) src + 1;
        }
}
void main()
{
        char data *buf1;
        char data *buf2;
        m_memcpy(buf1,buf2,10);
}
```

Compiler Ergebnis :

```
stmt level    source

   1          //File memcpynore.c
   2          #include <reg517A.h>
   3
   4               void  m_memcpy(void  *dst,  const  void  *src,
unsigned char n)
   5          {
   6   1           while (n--)
   7   1           {
   8   2           *(char *)dst = *(char *)src;
   9   2           dst = (char *) dst + 1;
  10   2           src = (char *) src + 1;
  11   2           }
  12   1      }
  13          void main()
  14          {
  15   1       char data *buf1;
  16   1       char data *buf2;
  17   1       m_memcpy(buf1,buf2,10);
  18   1      }
```

Pointerzuordnung :

```
NAME       CLASS   MSPACE  TYPE     OFFSET  SIZE
  buf1 . . AUTO     DATA    PTR       0000H  1
  buf2 .   AUTO     DATA    PTR       0001H  1
  dst. .   AUTO     DATA    VOID_PTR  0000H  3
  src. . . AUTO     DATA    VOID_PTR  0003H  3
  n. . .   AUTO     DATA    U_CHAR    0006H  1
```

```
C51 COMPILER V6.01 MEMCPYNORE 04/29/2006 16:29:45 PAGE 2
             ; FUNCTION _m_memcpy (BEGIN)
;Parameterübergabe Parameter 1 in den Registern R1-R3:
; dst Type, MSB, LSB
             ; FUNCTION _m_memcpy (BEGIN)
0000 8B00        R     MOV     dst,R3
0002 8A00        R     MOV     dst+01H,R2
0004 8900        R     MOV     dst+02H,R1
                                           ; SOURCE LINE # 4
                                           ; SOURCE LINE # 5
Schleifenanfang
0006         ?C0001:
                                           ; SOURCE LINE # 6
;Parameter n wird aus dem Data Speicher versorgt
;Ist n=1 muss der leere String kopiert werden.
;deshalb direkt Rücksprung
0006 AF00        R     MOV     R7,n
0008 1500        R     DEC     n
000A EF                MOV     A,R7
000B 6032              JZ      ?C0003
                                           ; SOURCE LINE # 7
                                           ; SOURCE LINE # 8
;Verwendung der Register R1-R3 als Übergabeparameter
;für den allgemeinen Pointerzugriff ?C?CLDPTR für den
;Zugriff auf den Quellstring
000D AB00        R     MOV     R3,src
000F AA00        R     MOV     R2,src+01H
0011 A900        R     MOV     R1,src+02H
0013 120000      E     LCALL   ?C?CLDPTR ;Speicherwert in Akku
;Verwendung der Register R1-R3 als Übergabeparameter
;für den allgemeinen Pointerzugriff ?C?CLDPTR für den
;Zugriff auf den Zielstring
0016 AB00        R     MOV     R3,dst
0018 AA00        R     MOV     R2,dst+01H
001A A900        R     MOV     R1,dst+02H
001C 120000      E     LCALL   ?C?CSTPTR; A in Speicherplatz
                                           ; SOURCE LINE # 9
;dst Pointer erhöhen +1, 2 Byte Operation Add (LSB) +Addc (MSB)
001F E500        R     MOV     A,dst+02H
0021 2401              ADD     A,#01H
0023 F9                MOV     R1,A
0024 E4                CLR     A
0025 3500        R     ADDC    A,dst+01H
0027 850000      R     MOV     dst,dst
002A F500        R     MOV     dst+01H,A
002C 8900        R     MOV     dst+02H,R1
                                           ; SOURCE LINE # 10
;src Pointer erhöhen +1, 2 Byte Operation Add (LSB) +Addc (MSB)
002E E500        R     MOV     A,src+02H
0030 2401              ADD     A,#01H
0032 F9                MOV     R1,A
0033 E4                CLR     A
0034 3500        R     ADDC    A,src+01H
0036 850000      R     MOV     src,src
0039 F500        R     MOV     src+01H,A
                                           ; SOURCE LINE # 11
;Schleifenrücksprung
003D 80C7              SJMP    ?C0001
                                           ; SOURCE LINE # 12
```

```
003F         ?C0003:
;Rücksprung
003F 22                 RET
             ; FUNCTION main (BEGIN)
                                                ; SOURCE LINE # 13
                                                ; SOURCE LINE # 14
                                                ; SOURCE LINE # 17
;Sichern des Eingangspointers für dst um die Parameterübergabe
;zu organisieren.
;Nicht als const definiert
0000 A900        R      MOV     R1,buf1
0002 7A00               MOV     R2,#00H
0004 7B00               MOV     R3,#00H
0006 C003               PUSH    AR3
0008 C002               PUSH    AR2
;Versorgung der Parameter für dst und src
;von einer Basisadresse aus gerechnet für die Daten in memcpy
000A 8B00        R      MOV     ?_m_memcpy?BYTE+03H,R3
000C 8A00        R      MOV     ?_m_memcpy?BYTE+04H,R2
000E 850000      R      MOV     ?_m_memcpy?BYTE+05H,buf2
0011 75000A      R      MOV     ?_m_memcpy?BYTE+06H,#0AH
;Rückspeicherung der gesicherten Daten
0014 D002               POP     AR2
0016 D003               POP     AR3
;Funktionsaufruf
0018 020000      R      LJMP    _m_memcpy
                                                ; SOURCE LINE # 18
             ; FUNCTION main (END)
```

```
C51 COMPILER V6.01 MEMCPYNORE 04/29/2006 16:29:45 PAGE 4

NAME                                  CLASS   MSPACE  TYPE     OFFSET  SIZE
====                                  =====   ======  ====     ======  ====
main . . . . . . . . . . . . . . . .  PUBLIC  CODE    PROC     0000H   ---
  buf1 . . . . . . . . . . . . . . .  AUTO    DATA    PTR      0000H   1
  buf2 . . . . . . . . . . . . . . .  AUTO    DATA    PTR      0001H   1
_m_memcpy. . . . . . . . . . . . . .  PUBLIC  CODE    PROC     0000H   ---
  dst. . . . . . . . . . . . . . . .  AUTO    DATA    VOID_PTR 0000H   3
  src. . . . . . . . . . . . . . . .  AUTO    DATA    VOID_PTR 0003H   3
  n. . . . . . . . . . . . . . . . .  AUTO    DATA    U_CHAR   0006H   1


MODULE INFORMATION:   STATIC OVERLAYABLE
   CODE SIZE        =     91    ----
   CONSTANT SIZE    =   ----    ----
   XDATA SIZE       =   ----    ----
   PDATA SIZE       =   ----    ----
   DATA SIZE        =   ----       9
   IDATA SIZE       =   ----    ----
   BIT SIZE         =   ----    ----
END OF MODULE INFORMATION.


C51 COMPILATION COMPLETE.  0 WARNING(S),  0 ERROR(S)
```

Als nächstes wird die Funktion m_memcpy mit dem Zusatz reentrant versehen.

```
//File memcpyre.c
#include <reg517A.h>

void m_memcpy(void *dst, const void *src, unsigned char n) reentrant
{
      while (n--)
      {
      *(char *)dst = *(char *)src;
      dst = (char *) dst + 1;
      src = (char *) src + 1;
      }
}
void main()
{
      char data *buf1;
      char data *buf2;
      m_memcpy(buf1,buf2,10);
}
```

Compilerergebnis:

```
stmt level    source

   1          //File memcpynore.c
   2          #include <reg517A.h>
   3
   4void m_memcpy(void *dst, const void *src, unsigned char n)
                    reentrant
   5          {
   6   1            while (n--)
   7   1            {
   8   2            *(char *)dst = *(char *)src;
   9   2            dst = (char *) dst + 1;
  10   2            src = (char *) src + 1;
  11   2            }
  12   1      }
  13          void main()
  14          {
  15   1       char data *buf1;
  16   1       char data *buf2;
  17   1       m_memcpy(buf1,buf2,10);
  18   1      }
Adresszuordnungen
NAME           CLASS   MSPACE  TYPE    OFFSET  SIZE
====           =====   ======  ====    ======  ====
_?m_memcpy . . PUBLIC   CODE   PROC     0000H  -----
  dst. . . . . AUTO     IDATA  VOID_PTR 0000H  3
  src. . . . . AUTO     IDATA  VOID_PTR 0003H  3
  n. . . . . . AUTO     IDATA  U_CHAR   0006H  1
```

```
C51 COMPILER V6.01 MEMCPYRE04/30/2006 11:26:13 PAGE 2
             ; FUNCTION _?m_memcpy (BEGIN)
;Speziellen Stackpointer verwenden.
;wächst von hohen Adress- zu niedrigen Adresswerten
;3 Elemente reservieren
0000 1500        E     DEC     ?C_IBP
0002 1500        E     DEC     ?C_IBP
0004 1500        E     DEC     ?C_IBP
;Der Stack liegt im indirekt adressierbaren internen Speicher
;Ablegen des ersten Parameters (dst) als generic pointer
;N und der zweite Parameter (src) liegen bereits dort
0006 A800        E     MOV     R0,?C_IBP;Adresse laden
0008 A603              MOV     @R0,AR3;Nachb. Reg.-Reg. Befehl
000A 08                INC     R0
000B A602              MOV     @R0,AR2
000D 08                INC     R0
000E A601              MOV     @R0,AR1
                                                ; SOURCE LINE # 4
;Schleifenstart
0010         ?C0001:
                                                ; SOURCE LINE # 6
;N holen
;Anfangsadresse des Bereichs für m_memcpy holen
0010 E500        E     MOV     A,?C_IBP
;Adresse auf N einstellen Offset=6
0012 2406              ADD     A,#06H
0014 F8                MOV     R0,A
;Wert für N holen
0015 E6                MOV     A,@R0
;Auf 0 zählen und abfragen
0016 16                DEC     @R0
0017 6057              JZ      ?C0003
                                                ; SOURCE LINE # 7
                                                ; SOURCE LINE # 8
;src generic pointer holen ab akt. Stackadresse + 3
0019 E500        E     MOV     A,?C_IBP
001B 2403              ADD     A,#03H
;Ptr laden in (R3,R2,R1)
001D F8                MOV     R0,A
001E 8603              MOV     AR3,@R0
0020 08                INC     R0
0021 E6                MOV     A,@R0
0022 FA                MOV     R2,A
0023 08                INC     R0
0024 E6                MOV     A,@R0
0025 F9                MOV     R1,A
;Auf generic Ptr src zugreifen
0026 120000      E     LCALL   ?C?CLDPTR
;Ergebnis nach R7
0029 FF                MOV     R7,A
;src generic pointer holen ab Stackanfang
002A A800        E     MOV     R0,?C_IBP
;Ptr laden in (R3,R2,R1)
002C 8603              MOV     AR3,@R0
002E 08                INC     R0
002F E6                MOV     A,@R0
0030 FA                MOV     R2,A
0031 08                INC     R0
0032 E6                MOV     A,@R0
0033 F9                MOV     R1,A
;Abzuspeichernder Wert in den Akku
0034 EF                MOV     A,R7
;Auf generic Ptr dst abspeichern
```

```
0035 120000      E     LCALL   ?C?CSTPTR
                                          ; SOURCE LINE # 9
;2 Byte Addition zur Einstellung der neuen Adresse für dst
0038 A800        E     MOV     R0,?C_IBP
003A 8603              MOV     AR3,@R0
003C 08                INC     R0
003D E6                MOV     A,@R0
003E FA                MOV     R2,A
003F 08                INC     R0
0040 E6                MOV     A,@R0
0041 2401              ADD     A,#01H
0043 F9                MOV     R1,A;LSB in R1 zwischenspeichern
0044 E4                CLR     A
0045 3A                ADDC    A,R2;MSB in A
;Wert zurückspeichern
0046 A800        E     MOV     R0,?C_IBP
0048 A603              MOV     @R0,AR3;Speichertyp abspeichern
004A 08                INC     R0
004B F6                MOV     @R0,A;MSB abspeichern
004C 08                INC     R0
004D A601              MOV     @R0,AR1;LSB abspeichern
                                          ; SOURCE LINE # 10
;2 Byte Addition zur Einstellung der neuen Adresse für src
;Akt Stackadresse + 3
004F E500        E     MOV     A,?C_IBP
0051 2403              ADD     A,#03H
0053 F8                MOV     R0,A
0054 8603              MOV     AR3,@R0
0056 08                INC     R0
0057 E6                MOV     A,@R0
0058 FA                MOV     R2,A
0059 08                INC     R0
005A E6                MOV     A,@R0
005B 2401              ADD     A,#01H
005D F9                MOV     R1,A
005E E4                CLR     A
005F 3A                ADDC    A,R2
;Wert zurückspeichern
0060 FA                MOV     R2,A
0061 E500        E     MOV     A,?C_IBP
0063 2403              ADD     A,#03H
0065 F8                MOV     R0,A
0066 A603              MOV     @R0,AR3
0068 08                INC     R0
0069 A602              MOV     @R0,AR2
006B 08                INC     R0
006C A601              MOV     @R0,AR1
```

```
                                                ; SOURCE LINE # 11
;Schleifenrücksprung
006E 80A0               SJMP    ?C0001
                                                ; SOURCE LINE # 12
0070            ?C0003:
;7 Byte wurden auf dem speziellen Stack verwendet
;Diese werden jetzt freigegeben
0070 E500           E   MOV     A,?C_IBP
0072 2407               ADD     A,#07H
0074 F500           E   MOV     ?C_IBP,A
;Rücksprung
0076 22                 RET
             ; FUNCTION _?m_memcpy (END)

             ; FUNCTION main (BEGIN)
                                                ; SOURCE LINE # 13
                                                ; SOURCE LINE # 14
                                                ; SOURCE LINE # 17
;Ersten Speicherplatz des speziellen Stack einstellen
0000 1500           E   DEC     ?C_IBP
;Erstes Element in R0 adressieren
0002 A800           E   MOV     R0,?C_IBP
;N abspeichern
0004 760A               MOV     @R0,#0AH
;buf2 Adresse ablegen
0006 A900           R   MOV     R1,buf2
;Generic Pointer für buf2 (src) anlegen
;und in den spez. Stack kopieren
;3 Plätze anfordern
0008 1500           E   DEC     ?C_IBP
000A 1500           E   DEC     ?C_IBP
000C 1500           E   DEC     ?C_IBP
;aktuellen Platz holen
000E A800           E   MOV     R0,?C_IBP
;Generic Pointer ablegen
0010 7600               MOV     @R0,#00H
0012 08                 INC     R0
0013 7600               MOV     @R0,#00H
0015 08                 INC     R0
0016 A601               MOV     @R0,AR1
;Generic Pointer für buf1 (dst) erzeugen
0018 A900           R   MOV     R1,buf1
001A 7A00               MOV     R2,#00H
001C 7B00               MOV     R3,#00H
;Funktionsaufruf
001E 020000         R   LJMP    _?m_memcpy
                                                ; SOURCE LINE # 18
             ; FUNCTION main (END)
```

```
MODULE INFORMATION:   STATIC OVERLAYABLE
   CODE SIZE        =    152    ----
   CONSTANT SIZE    =   ----    ----
   XDATA SIZE       =   ----    ----
   PDATA SIZE       =   ----    ----
   DATA SIZE        =   ----       2
   IDATA SIZE       =   ----    ----
   BIT SIZE         =   ----    ----
END OF MODULE INFORMATION.
```

Angabe der Registerbank für ein Unterprogramm

Jedes Unterprogramm kann einer Registerbank zugewiesen werden. Dies ist dann günstig, will man das Sichern und das Zurückspeichern von 8 Registern sparen. Die Umschaltung auf eine andere Registerbank macht ein solches Vorgehen unnötig. Sollen jedoch weitere Unterprogramme von hier aus aufgerufen werden, so muss dafür gesorgt werden, dass die Registerbank möglichst erhalten bleibt. Es können vier Registerbänke 0-3 ausgewählt werden.
Beispiel:

```
void rb_function () using 1;
```

Anbindung an Interruptadressen:
Die Anbindung an eine Interrupteinsprungadresse erfolgt über eine Nummer, die über die Heximaladresse (HA) des Einsprungpunktes berechnet wird:

N=(HA-3)/8

So ergibt sich für den externen Interrupt 0 mit HA=3h die Nummer 0
Für den Timer 0 mit HA=Bh die Nummer 1
Usw:

Beispiel:

```
//globals
typedef unsigned int BYTE;
BYTE interruptcnt, second;

//Interruptanbindung des Timers 0
void isr_timer0(void) interrupt 1 using 2
{
        interruptcnt++;
        if(interruptcnt==20){
                second++;
                interruptcnt=0;
        }
}
```

Die Interruptangabe hat folgenden Einfluss auf das Verhalten der Interrupt-Service Routine:

- Der Inhalt der Special Function Register ACC, B, DPH, DPL und PSW werden auf den Stack gesichert, wenn es notwendig ist.
- Alle Arbeitsregister, die in der Interruptroutine verwendet werden, werden auf den Stack gesichert, wenn keine Registerbank angegeben wurde.
- Alle Daten die auf den Stack gesichert wurden, werden zurückgespeichert bevor die Funktion verlassen wird.
- Zum Schluss wird die RETI Funktion aufgerufen.

Beispiel für den Aufbau einer Uhr:
Es soll zum Aufbau der Uhr angenommen werden, dass der Prozessor die Timer mit mit 12 MHz versorgt (Prescaled Clock). Für die Timer können damit Zeiteinheiten von 1 us gezählt werden.
In diesem Beispiel soll der Timer 0 im 16 Bit Mode eingesetzt werden, um eine Grundeinheit von 50 ms zu erzeugen. Für eine Sekunde sind dann 20 Grundeinheiten zu zählen.
Das Stellen der Uhr wird nicht betrachtet.
Die Zeit wird in 3 globalen Variablen abgelegt:
Stunden, Minuten, Sekunden

```
//File clock.c

#include <C8051F340.h>
typedef unsigned char BYTE;
BYTE Stunden=0;
BYTE Minuten=0;
BYTE Sekunden=0;
BYTE ICnt=0;
void main ()
{
        IEN0 =0x92; /* Interruptfreigabe*/
        TMOD = 0x1;         /* Timer 0, Modus 1, 16 Bit Zähler */
        TCON = 0x0;
        TL0= -50000 & 0x00ff;       //Low Byte des Nachladewerts erzeugen
        TH0= -50000 >> 8;           //High Byte des Nachladewerts erzeugen
        TR0 = 1;                    //Timer starten
        while(1);                   //Endlosschleife
}
void ISR_Timer0() interrupt 1
{
        TR0=0;                      //Timer stop
        TL0= (-50000 << 8) >> 8;  //Timer nachladen
        TH0= -50000 >> 8;
        TR0=1;                      //Timer start
        ICnt++;                     //Zwischenzähler erhöhen
        if(ICnt==20)        {ICnt=0;            Sekunden++;       }
        if(Sekunden==60)    {Sekunden=0;        Minuten++;   }
        if(Minuten==60)     {Minuten=0;         Stunden++;  }
        if(Stunden==24)     {Stunden=0;                 }

}
```

```
ASSEMBLY LISTING OF GENERATED OBJECT CODE
             ; FUNCTION main (BEGIN)
                                                ; SOURCE LINE # 9,10,11
0000 75A892              MOV     IEN0,#092H
                                                ; SOURCE LINE # 12
0003 758901              MOV     TMOD,#01H
                                                ; SOURCE LINE # 13
0006 E4                  CLR     A
0007 F588                MOV     TCON,A
                                                ; SOURCE LINE # 14
0009 758AB0              MOV     TL0,#0B0H
                                                ; SOURCE LINE # 15
000C 758C3C              MOV     TH0,#03CH
                                                ; SOURCE LINE # 16
000F D28C                SETB    TR0
0011           ?C0001:
                                                ; SOURCE LINE # 17
0011 80FE                SJMP    ?C0001
                                                ; SOURCE LINE # 18
             ; FUNCTION main (END)
             ; FUNCTION ISR_Timer0 (BEGIN)
0000 C0E0                PUSH    ACC
0002 C0D0                PUSH    PSW
                                               ; SOURCE LINE # 19,21
0004 C28C                CLR     TR0
                                                ; SOURCE LINE # 22
0006 0500          R     INC     ICnt
                                                ; SOURCE LINE # 23
0008 E500          R     MOV     A,ICnt
000A B41405              CJNE    A,#014H,?C0004
000D 750000        R     MOV     ICnt,#00H
0010 0500          R     INC     Sekunden
0012           ?C0004:
                                                ; SOURCE LINE # 24
0012 E500          R     MOV     A,Sekunden
0014 B43C05              CJNE    A,#03CH,?C0005
0017 750000        R     MOV     Sekunden,#00H
001A 0500          R     INC     Minuten
001C           ?C0005:
                                                ; SOURCE LINE # 25
001C E500          R     MOV     A,Minuten
001E B43C05              CJNE    A,#03CH,?C0006
0021 750000        R     MOV     Minuten,#00H
0024 0500          R     INC     Stunden
0026           ?C0006:
                                                ; SOURCE LINE # 26
0026 E500          R     MOV     A,Stunden
0028 B41803              CJNE    A,#018H,?C0007
002B 750000        R     MOV     Stunden,#00H
002E           ?C0007:
002E 758AB0              MOV     TL0,#0B0H      ; SOURCE LINE # 27
0031 758C3C              MOV     TH0,#03CH      ; SOURCE LINE # 28
0034 D28C                SETB    TR0            ; SOURCE LINE # 29
0036 D0D0                POP     PSW            ; SOURCE LINE # 31
0038 D0E0                POP     ACC
003A 32                  RETI
             ; FUNCTION ISR_Timer0 (END)
```

8.5 Zugriff auf feste Adressen

Für den Zugriff auf feste Adressen ist es möglich, über eine Compilerdirektive "_at_" Variablen eine feste Adresse zuzuweisen.

[memory space] type variable_name _at_ constant

Beispiele:

```
idata  int    dat1   _at_ 0x40;
xdata  char   c1[10] _at_ 0x8000;
```

Die Variablen können jetzt direkt angesprochen werden:

```
c1[6]=3;
dat1=2;
```

Eine weitere Variante ist die Definition von Pointern, denen dann direkt Adresswerte zugewiesen werden können. Der Zugriff auf Felder kann wieder in bekannter Weise erfolgen (z.B. a[1]). Bei skalaren Variablen muss der Inhaltsoperator angewendet werden.

Beispiele:

Definition:

```
unsigned char xdata     *arr=0x2020;
unsigned char data      *ptb=0x20;
```

Zugriff:

```
arr[1]=5;

*ptb=7;
```

Solche Vorgehensweisen erlauben zwar ein direktes Ansprechen von Speicherplätzen. Dennoch sind sie sind aber nicht sehr ratsam, wenn vom C Compiler frei Variable angelegt werden. Dies kann zu Überschneidungen führen.

9 Echtzeitsysteme

9.1 Anforderungen

Beim Entwurf von Programmen zur Erfüllung vorgegebener Aufgaben werden funktionelle Zusammenhänge definiert und eine oder mehrere Abarbeitungsreihenfolgen festgelegt. Bei Echtzeitsystemen sind zusätzlich Zeiten zur Bearbeitung einer bestimmten Aufgabe zu berücksichtigen. Hierbei kann zwischen Soft- und Hard-Realzeitsystemen unterschieden werden. Bei „hard real time systems“ müssen bestimmte Zeitpunkte oder feste Zeiträume eingehalten werden. Bei „soft real time systems“ wird eine möglichst schnelle Abarbeitung angestrebt. Es müssen aber keine festen Zeiten eingehalten werden. Die meisten Systeme stellen jedoch eine Kombination der beiden Typen dar.
Sollen beispielsweise Messdaten ermittelt werden, so ist die Aufnahme häufig an einen festen Rhythmus gekoppelt. Die Anzeige bzw. eine Mittelwertbildung unterliegt zunächst keinem festen Zeitrahmen. Sie muss aber ebenfalls zyklisch durchgeführt werden.

9.2 Konzepte

Bevor Konzepte zur Realisierung von Echtzeitsystemen vorgestellt werden, soll zunächst noch einmal auf einfache Verfahrensweisen zur Implementierung auch zeitabhängiger Systeme eingegangen werden. Werden Programmabläufe als serielle Abarbeitung von Funktionen verstanden, die durch Unterbrechungsmechanismen auf besondere Ereignisse reagieren können, führt dies zu sogenannten Foreground/Backgroundsystemen.

9.2.1 Foreground/Backgroundsysteme

Foreground/Backgroundsysteme besitzen eine Hauptschleife (Super Loop) in der nacheinander bestimmte Aufgaben (Tasks) ausgeführt werden. Dieser Bereich wird als Background bezeichnet. Asynchrone Ereignisse werden über Interrupt Service Routinen abgewickelt (Foreground oder Interruptebene). Durch die Eigenschaft der Interruptbearbeitung, den gerade laufenden Task zu unterbrechen und eine notwendige Aktion einzufügen, kann ein Interrupt auch zur Bearbeitung von kritischen Ereignissen verwendet werden. Beim Entwurf der Programme müssen jedoch zeitliche Aspekte beachtet werden. Interrupt Service Routinen müssen üblicherweise so kurz wie möglich gehalten werden um den Hintergrundbetrieb weiter zu gewährleisten. Das Warten auf Ereignisse oder die langwierige Bearbeitung von Daten sollte nicht in einer Interrupt Service Routine vorgenommen werden.
Feste Zeitrhythmen können durch den Einsatz von Timern eingehalten werden. Sollen mehrere Vorgänge zeitlich gekoppelt werden, sind meist Zustandsmaschinen mit zu implementieren.
Bei Änderungen im Programmcode werden oft auch die Zeiten der Ablaufreihenfolge mit geändert. Dies führt dann zu einem erheblichen Analyseaufwand beim Test des Gesamtsystems.
Bei Systemen mit geringer Komplexität, sowohl in der Aufgabenstellung als auch in ihrem zeitlichen Verhalten, fallen diese Nachteile nicht sehr ins Gewicht. So werden auch die meisten hochvolumigen Mikrocontrollerapplikationen auf diese Art entwickelt (Einfache Steuerungen (z.B. Heizung, Spielzeug, Telefon …).

Denkbar sind auch Systeme, die nur auf Ereignisse reagieren. Hier besteht das Backgroundsystem nur aus einer Endlosschleife, die bei jedem Ereignis verlassen wird.

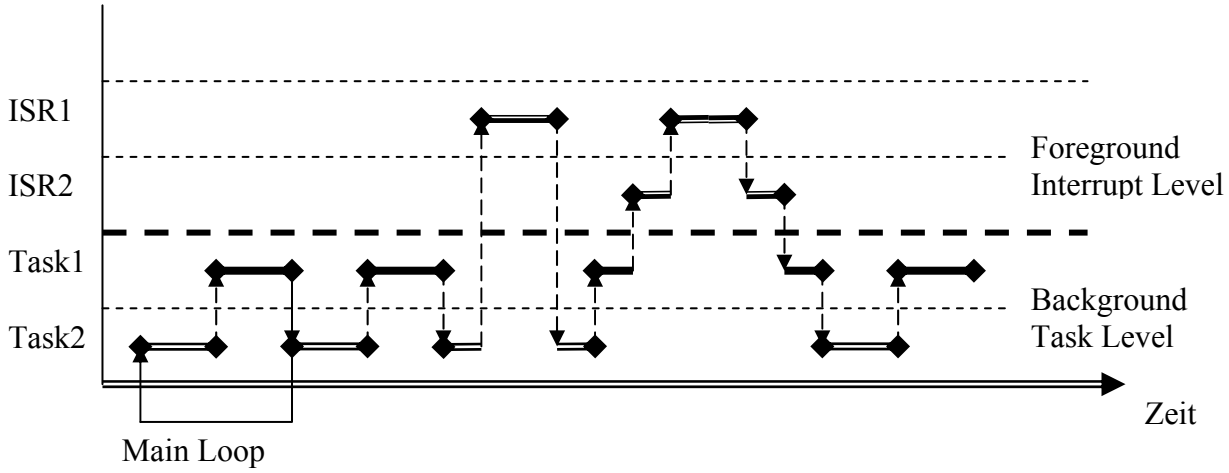


Bild 8.1 Foreground / Background System

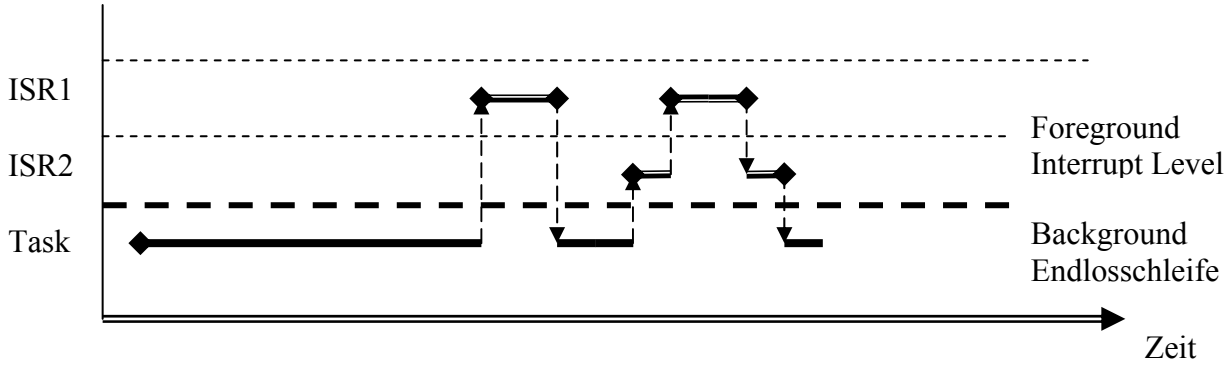


Bild 8.2 Foreground / Background System mit Endlosschleife

Foreground / Background Systeme können ohne größere Probleme mit der Programmiersprache C implementiert werden. Durch die eingeschränkten Ressourcen (z.B. Stack) treten zusätzliche Probleme bei Parameterübergaben oder Unterprogrammaufrufen auf. Aktionen zur Sicherung von Daten benötigen Speicherplatz und Ausführungszeit. Dies verursacht ggf. eine verzögerte Sperrung des Interruptsystems.

Beispiel: Ampelsteuerung für eine Fußgängerampel

Das vorgestellte Beispiel soll nur den prinzipiellen Aufbau eines Foreground / Background - Systems demonstrieren. Kritische Situationen bei der Interruptbehandlung oder der Stackbelegung werden nicht behandelt.

Aufgabenstellung:

Ein Fußgängerüberweg soll mit einer Ampel sicherer gemacht werden. Fußgänger können auf der jeweiligen Seite eine Taste drücken, wenn sie die Straße überqueren wollen. Die Ampel für die Autofahrer wird dann auf rot geschaltet und die Fußgängerampel auf grün. Dies soll mit den üblichen Übergangsphasen geschehen. Nach einer gewissen Zeit soll dann wieder in den Ausgangszustand zurückgekehrt werden.

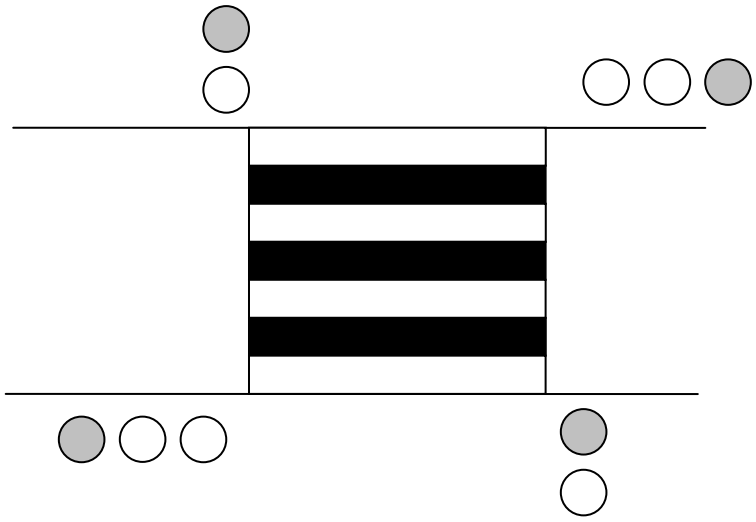


Bild 8.3 Fußgängerampel

Ablauf:

1. Taste wird gedrückt
2. Autoampel->gelb Fußgängerampel->rot 1 sec
3. Autoampel->rot Fußgängerampel->rot 1 sec
4. Autoampel->rot Fußgängerampel->grün 10 sec
5. Autoampel->rot Fußgängerampel->rot 1 sec
6. Autoampel->rot/gelb Fußgängerampel->rot 1 sec
7. Autoampel->grün Fußgängerampel->rot sonst

Das System wird im Sinne eines Foreground/Background Systems realisiert. Der notwendige Zeitrhythmus wird über Timer 0 realisiert, der auf ein Zeitintervall von 50 ms eingestellt wird. Vorausgesetzt wird ein Timertakt von 12 MHz. Diese Zeitintervalle müssen jeweils gezählt werden um ein Zeitraster von 1 sec erzeugen zu können. Ebenso muss eine Zustandsvariable jede Sekunde weitergeschaltet werden um die richtigen Steuersignale aufzubauen. Diese Aufgaben können innerhalb der Interrupt Service Routine des Timers 0 ohne weiteres erledigt werden. Die Abfrage der Tasten und die Ausgabe der Signale zur Steuerung der Ampellampen sollen in einer Endlosschleife (Super Loop) im Hauptprogramm erfolgen.

Das Anschalten der Tasten und der Leistungstreiber soll an Port 4 realisiert werden.

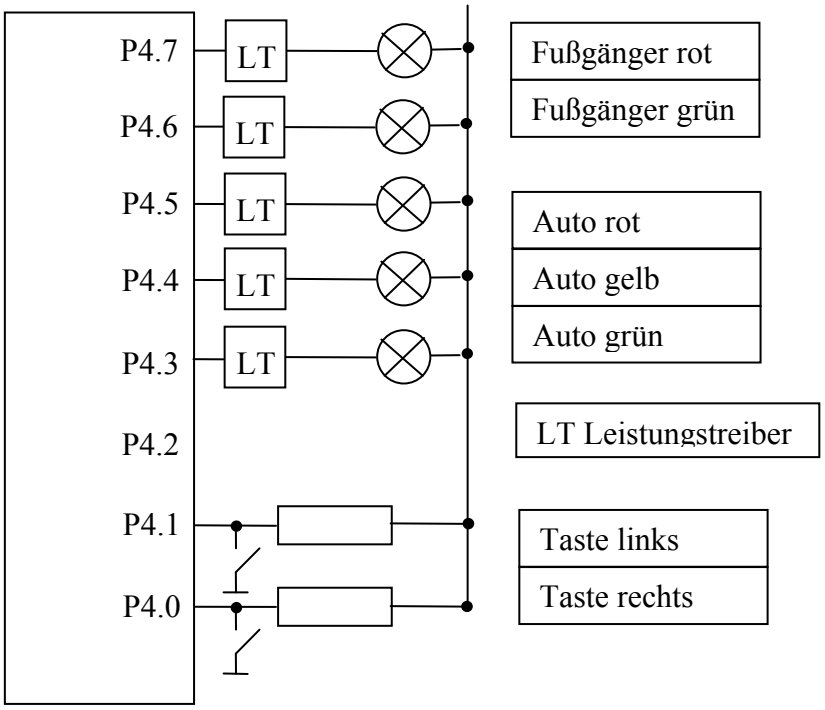


Bild 8.4 Beschaltung des Ports 4 bei der Ampelsteuerung

Programmfestlegungen:

```
//File ampel.c
#include <C8051F340.h>
typedef unsigned char INT8U;

//Timer reload value
#define TIMER0RELOAD -50000
//State descriptors
#define FrotAgruen    0x8B
#define FrotAgelb     0x93
#define FrotArot      0xA3
#define FgruenArot    0x63
#define FrotArt_ge    0xB3
//global Variables
INT8U         astate=0;              //State Variable
INT8U         v1=3;                  //P4 check variable
INT8U         timercount=0;          //Time Tick Untersetzung
```

Funktionen:

```
//Interrupts Service Routine Timer 0
void ISR_Timer0() interrupt 1{
      TR0=0;
      TH0= TIMER0RELOAD >> 8;
      TL0= TIMER0RELOAD & 0x00ff;
      TR0=1;
      timercount=timercount+1;
      if(timercount==3){//Reduziert zum Test in der Simulation (Normal 20)
          astate = astate+1;
          timercount=0;
      }
}
//Timer0 Initialisierung
void Timer0_Setup(){
      TMOD=0x1;
      TH0= TIMER0RELOAD >> 8;
      TL0= TIMER0RELOAD & 0x00ff;
      TR0=0;
}
void  main(){
      Timer0_Setup();          //Timer 0 Initialisierung
      EA=1;                    //Interrupt enable alle
      ET0=1;                   //Timer 0 interrupt an
      astate=0;                //Anfangszustand setzen
      //Super Loop
      while(1){
          if      (astate==0){
                                                  P4=FrotAgruen;
                                                  //Abfrage der Tasten
                                                  do{v1=P4& 0x03;}while(v1==3);
                                                  TR0=1;          //Ampeldurchlauf starten
          }else if(astate==1){                    P4=FrotAgruen;
          }else if(astate==2){                    P4=FrotAgelb;
          }else if(astate==3){                    P4=FrotArot;
          }else if(astate>=4 && astate<=13){      P4=FgruenArot;
          }else if(astate==14){                   P4=FrotArot;
          }else if(astate==15){                   P4=FrotArt_ge;
          }else if(astate==16){
                                                  TR0=0;          //Ampeldurchlauf beenden
                                                  astate=0;
          }else{
                                                  /*Error fehlerhafter Programmzustand*/

          }
      }
}
```

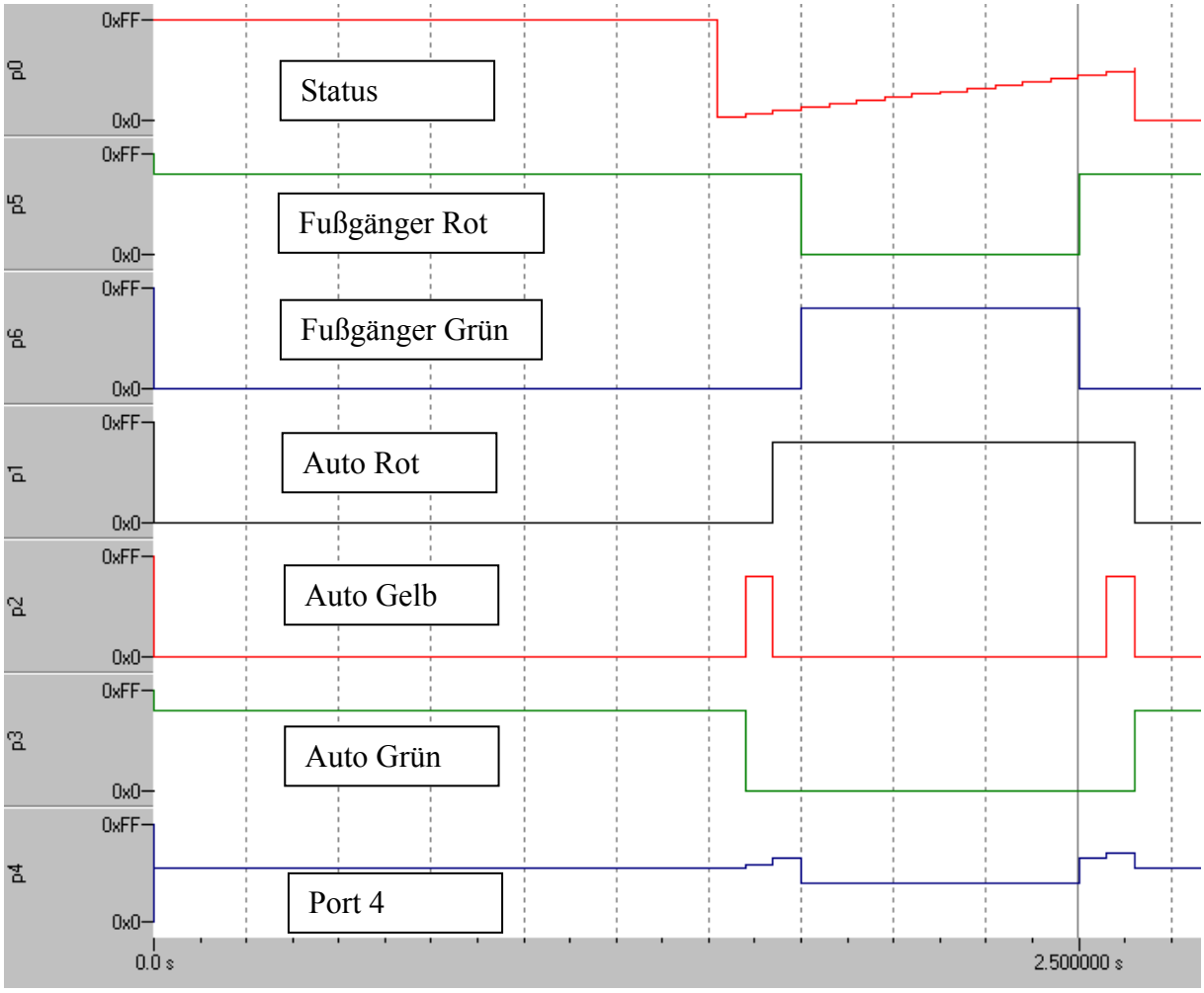


Bild 8.5 Ablauf Ampelsteuerung

9.2.2 Betriebssysteme

Die Planung von Foreground/Backgroundsystemen wird umso schwieriger, je mehr Aufgaben in einem festgelegten Zeitraster erfüllt werden müssen. Probleme treten dann auf, wenn die Durchführung eines Tasks nicht mehr in einer zusammenhängenden Zeitscheibe erledigt werden kann. Eine Umorganisation oder Teilung der Aufgaben kann zwar eine Lösung liefern, ist aber mit einer Änderung der Programmstruktur verbunden. Eine weitere Maßnahme kann die Kopplung des schnellen Prozesses an einen Timerinterrupt sein, der dann eine Unterbrechung des laufenden Tasks vornimmt und die Bearbeitungssequenz damit einschiebt. Damit wird jedoch die Reaktionsfähigkeit des Systems auf äußere Ereignisse gemindert, die wiederum durch Änderung der Interruptprioritäten angepasst werden könnte.

Wünschenswert wäre ein System, das parallel die einzelnen Tasks abarbeiten kann. Mikrocontroller besitzen im Allgemeinen nur die Fähigkeit eine Aufgabe gleichzeitig zu erledigen. Eine quasi Gleichzeitigkeit kann jedoch erreicht werden, wenn die einzelnen Tasks jeweils für kurze Zeit abwechselnd vom Prozessor bearbeitet werden könnten.
Vorteil einer solchen Vorgehensweise ist die Organisation der Tasks in einer Weise, die zunächst keine Änderung im Ablauf der einzelnen Tasks erfordert. Es muss natürlich gewährleistet sein, dass alle Aufgaben vom Prozessor im vorgegeben Zeitfenster erledigt werden können.

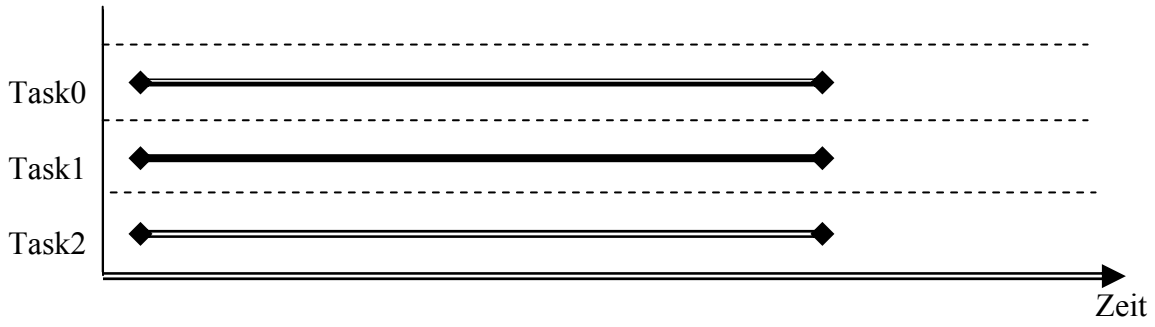


Bild 8.6.1 Wünschenswerter paralleler Ablauf von Tasks

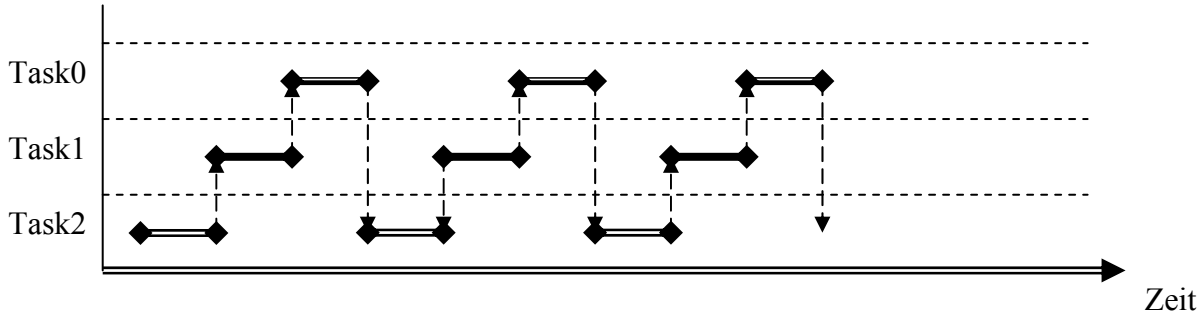


Bild 8.6.2 Quasi paralleler Ablauf von Tasks

Für die Umschaltung zwischen den Tasks müssen zusätzliche Maßnahmen getroffen werden.
Notwendig ist ein übergeordnetes Programm, das die Organisation der einzelnen Tasks übernimmt. Dieses Programm wird Scheduler oder Dispatcher genannt. Er muss dafür sorgen, dass die Tasks die jeweilig benötigten aktuellen Daten vorfinden und eine Auswahl des

nächsten Tasks treffen. Hier spielen in den meisten Systemen Prioritäten eine zusätzliche Rolle.

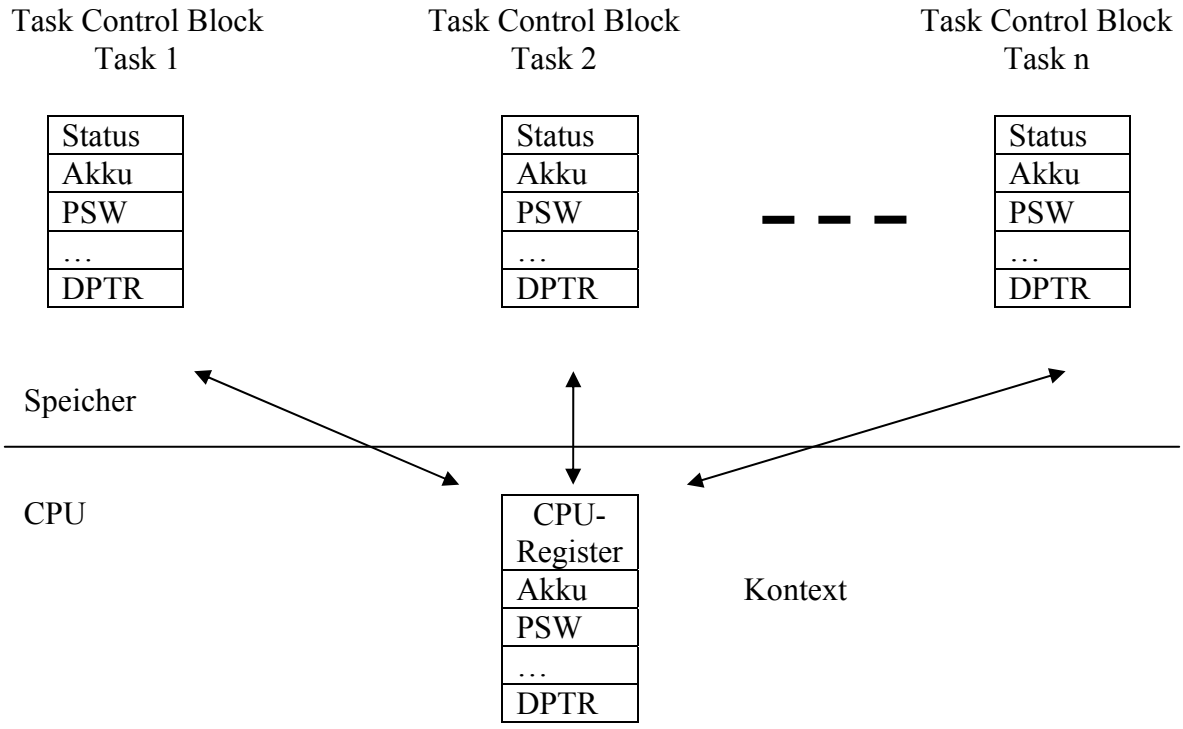


Bild 8.7 Abbildung der Taskdaten in den CPU-Kontext

Bei den Taskdaten selbst wird eine Kennung des gerade aktuellen Zustandes abgelegt. Dies gestattet dem Scheduler entsprechende Auswahlkriterien anzuwenden. Einfache Zustände können sein:

Ready der Task wartet auf die Zuteilung der CPU

Running der Task ist aktiv

Waiting der Task wartet auf ein Ereignis (z.B. Interrupt)

Bei der Steuerung der Tasks wird zwischen „non preemptiven Systemen“ und „preemptiven Systemen“ unterschieden.
Bei „non preemptiven Systemen“ gehen die Tasks in einen Wartezustand und erlauben dem Scheduler einen neuen Task auszuwählen. Die Tasks müssen dabei in einer bestimmten Weise kooperativ sein. Das bedeutet, eine Übergabe der Kontrolle muss hinreichend häufig erfolgen. Interrupt Service Routinen können, wie bekannt, bearbeitet werden, da nach der Bearbeitung der ISR in den gerade aktuellen Task zurückgekehrt wird.

Vorteil von „non preemptiven Systemen“ ist die definierte Übergabestelle der Kontrolle an den Scheduler. Damit verringert sich der Aufwand um Zugriffe auf gemeinsame Speicherbereiche zu gewährleisten. Durch die starre Übergabe müssen jedoch auch Tasks mit

einer höheren Priorität, die beispielweise durch einen Interrupt angestoßen worden sind, auf die Übergabe warten.

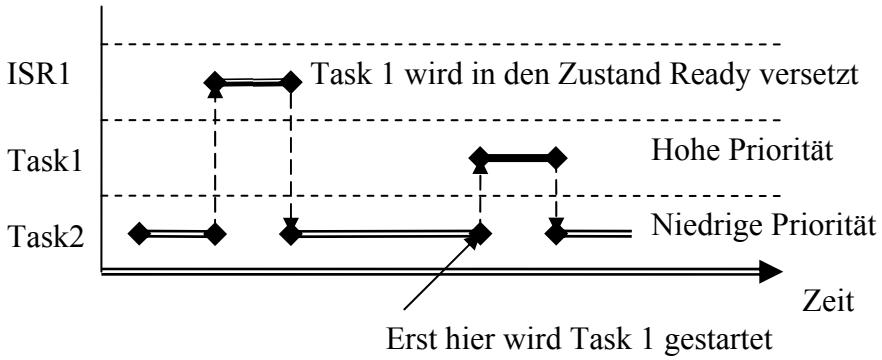


Bild 8.8 CPU-Übergabe bei „non preemptiven Systemen“

Preemptive Systeme werden benötigt, wenn eine geringe Antwortzeit auf ein Ereignis erreicht werden soll. Die meisten verfügbaren Systeme bieten eine solche Möglichkeit an. Der Scheduler muss dann in der Lage sein, von sich aus einen Task zu unterbrechen und einen neuen Task zu starten. Durch diese jetzt nicht mehr absehbare Unterbrechung eines Tasks an einer beliebigen Stelle werden höhere Anforderungen an die Organisation der Daten gestellt.
Der Ablauf beim Auftreten des Ereignisses in Bild 8.8 ändert sich dann in das Profil, das in Bild 8.9 dargestellt ist.

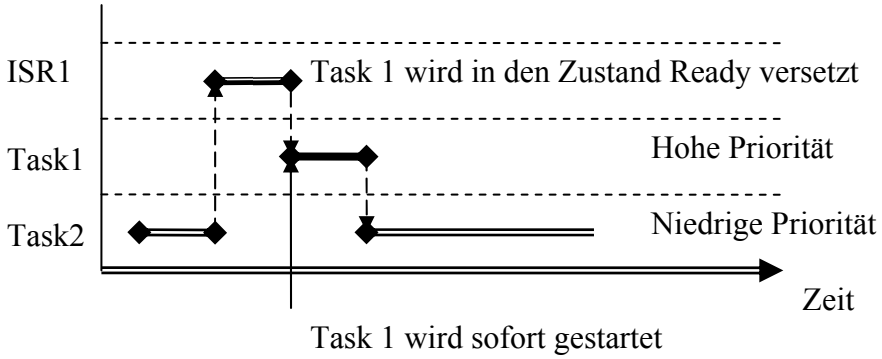


Bild 8.9 CPU Übergabe bei preemptiven Systemen

Bei den bisher dargestellten Abläufen wurde die Taskübergabe jeweils von Ereignissen im laufenden Prozess selbst oder von externen Ereignissen veranlasst. Die Festlegung einer oberen Zeitgrenze für einen Task ist eine zusätzliche Maßnahme, die den Eindruck der quasi parallelen Verarbeitung noch verstärkt. In der einfachsten Form werden Tasks mit gleicher Priorität gestartet und dann nach einer festgelegten Zeit (Quantum) jeweils aktiviert. Dazu muss der Scheduler zyklisch gestartet werden. Dies kann beispielsweise durch den Einsatz einer Interrupt Service Routine erfolgen, die an einen zyklisch arbeitenden Timer angebunden wurde.

9.2.3 Round Robin Verfahren (Time Slicing)

Werden Tasks mit der gleichen Priorität belegt, so wird bei preemptiven Systemen ein Task entweder solange laufen, bis er in einen Wartezustand kommt oder vom Scheduler unterbrochen wird. Bei der Unterbrechung ist es dann notwendig, die Daten des aktuellen Prozesses zu retten und die Daten des neuen Prozesses bereitzustellen.

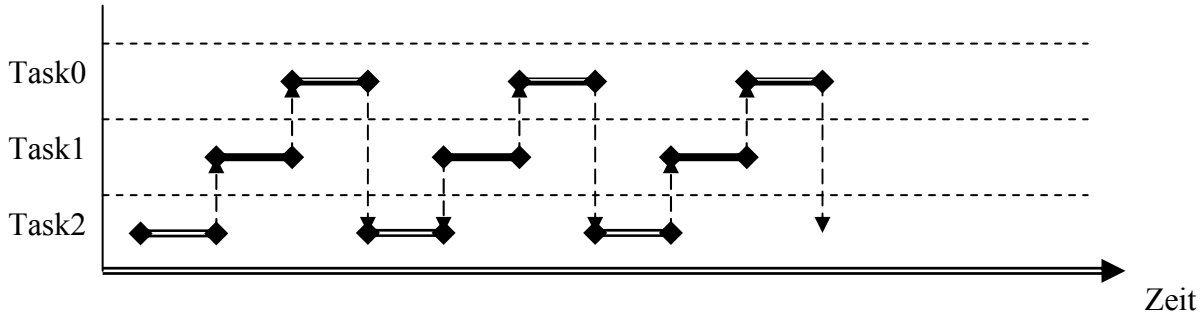


Bild 8.9 Taskablauf beim Round Robin Verfahren bei gleicher Priorität der Tasks

Nach jedem Aktivieren des Schedulers, der innerhalb einer Timer-Interrupt-Service-Routine aufgerufen wird, erfolgt die Unterbrechung des gerade laufenden Tasks und das Starten des folgenden Tasks. Da kein Task eine höhere Priorität hat als ein anderer Task, erfolgt der Wechsel zyklisch. Bei jedem Übergang müssen die Daten des anghaltenen Tasks gespeichert werden (Task Controll Block) und die Daten des neuen Tasks an die zugehörigen Speicherplätze kopiert werden. Diese Aktionen werden alle während der Aktivität des Schedulers durchgeführt.

9.3 Verwendung eines einfachen Echtzeitsystems (HKRO)

Das zugrunde liegende Echtzeitsystem (Hochschule Karlsruhe Real Time Operation System HKRO) bietet die Möglichkeit Applikationen mit Hilfe eines einfachen Betriebssystems zu realisieren und in ihrer Funktion zu verstehen. Das System wurde im Hinblick auf die Verwendung des Prozessors vom Typ 8051 entwickelt.
Echtzeitbetriebssysteme mit einem höheren Anspruch an die Funktionalität werden, so weit es möglich ist, in C geschrieben. Nur ein kleiner Teil zur Umstellung des CPU-Kontextes wird in Assembler programmiert. Der Betrieb des Systems wird über Schnittstellenfunktionen und spezielle Initialisierungstechniken vorgenommen. Für den industriellen Einsatz ist die Verwendung eines vorgegebenen, validierten und von der Größe her passenden Systems mit Sicherheit ein vernünftiger Ansatz. Hierbei ist der innere Ablauf des Systems aufgrund der Komplexität meist jedoch nur schwer in seinen gesamten Abhängigkeiten zu verstehen.
Zum Verständnis auch der inneren Zusammenhänge soll deshalb im Weiteren das bereits erwähnte einfache HKRO System zum Einsatz kommen.

Die Implementierung erfolgte weitgehend in der Programmiersprache C. Für kritische Teile wie z.B. der Schedulersteuerung oder der Datensicherung wurden Assemblerprogramme eingesetzt.
Durch die angestrebte Einfachheit ergeben sich dabei folgende Festlegungen:

- Es können bis zu etwa 6 unterschiedliche Tasks definiert werden.
  (abhängig vom sonstigen Speicherbedarf im IData-Bereich)
- Semaphore können eingesetzt werden.
- Zwischen den Tasks können Nachrichten ausgetauscht werden.
- Tasks können in einen definierten Wartezustand gehen:
  Zeitraum, Unterbrechungsanforderung
- Es sind zunächst keine Unterprogrammaufrufe für die Tasks zugelassen.
- Datenübergaben erfolgen über globale Variable.
- Zur Vermeidung von generischen Pointer bei der Code-Generierung werden alle Variablen mit entsprechenden Speicherbereichsangaben versehen.
- Das System arbeitet im internen RAM sowohl im Data Bereich jedoch hauptsächlich im IData-Bereich.
- Code Größe etwa 800 Byte.

Die Wirkungsweise soll zunächst an einem einfachen Beispiel im nächsten Abschnitt demonstriert werden.

9.3.1 Einführungsbeispiel

Zur Demonstration der Wirkungsweise eines Echtzeitbetriebssystems sollen entsprechend der angegebenen Konfiguration in Bild 8.12 Schalter Anzeigeeinheiten steuern.
Hierzu soll angenommen werden, dass an Port 3 jeweils 4 Schalter und 4 Leuchtdioden angeschaltet sind. Drei Schalter sind für die direkte Steuerung von 3 Leuchtdioden reserviert. Eine weitere Leuchtdiode soll entweder rhythmisch blinken oder ständig leuchten. Dies soll mit dem letzten verbliebenen Schalter kontrolliert werden. Eine Entprellung der Schalter braucht nicht vorgenommen zu werden.

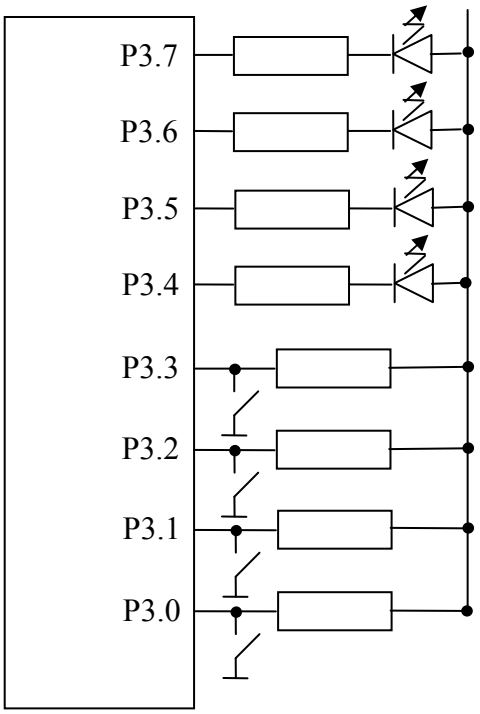


Bild 8.12 Port 4 - Beschaltung

Festlegungen:
Der Schalter an P3.3 und die Leuchtdiode an P3.7 sollen für das Blinken verwendet werden.
Die Wiederholrate des Timers 0 wird mit 20 ms festgelegt.
Zum Betrieb des Echtzeitbetriebssystems müssen zunächst Aufgabengruppen (Tasks) definiert werden.
Im vorliegenden Beispiel soll das Einlesen der Schalter P3.0 bis P3.2 als eine Aufgabe und die Steuerung der blinkenden LED als weitere Aufgabe gehandhabt werden.
Ein Task wird dann als C-Funktion ohne Rückgabewert definiert. Innerhalb des Tasks muss eine Endlosschleife realisiert werden, da ein Task eine Aufgabe darstellt, die ständig ausgeführt werden muss. Innerhalb der Schleife werden die eigentlichen Aktionen ausgeführt.
Damit können die Tasks definiert werden:

Tsk1: - Zyklisches Einlesen der Tasten an P3.0 bis P3.2
- Komplementbildung
- Ansteuerung der Dioden.

Tsk2: -Zyklisches Einlesen der Taste an P3.3
- Davon abhängig Komplementbildung oder Anschalten der LED an P3.7
- Blinken

Programmrealisierung:
Bis auf die while–Schleife unterscheiden sich die Aktionen innerhalb des Tsk1, nicht von einer Funktion innerhalb eines Background-/Foregroundsystems.

```
//Zyklisches Einlesen der Tasten an P4.0 bis P4.2
void Tsk1()
{
        while(1){
                P3_4= ~P3_0;
                P3_5= ~P3_1;
                P3_6= ~P3_2;
        }
}
```

In Tsk2 entsprechen die Abfrage der Taste und die Auswahl der ausgesonderten Aktion auch dem normalen Programmierstandard. Die Verzögerung zum Erreichen der Blinkfunktion wird jedoch über eine spezielle Wartefunktion realisiert. Wird innerhalb des Tasks diese Funktion erreicht, so wird die Kontrolle an den Scheduler übergeben und der Task für ein Vielfaches der Zeitbasis (hier 50*Zeitbasis) des Echtzeitsystems suspendiert. Der Scheduler kann dann einen anderen Task aktivieren. Mit dem ersten Parameter der RO_WaitDel-Funktionen wird eine Zählernummer angegeben, die damit mit dem Warteaufruf assoziiert ist.

```
//Zyklisches Einlesen der Taste an P3.3
//Davon abhängig Komplementbildung oder
//Anschalten der LED an P3.7
//Blinken

void Tsk2()
{
        while(1){
                RO_WaitDel(0,50);   //Warten, in der Simulation 5
                if(P3_3==1)
                        P3_7= ~P3_7; //Komplement bilden bei
                                     //nicht gedrückter Taste
                else P3_7=0;         //Leuchtdiode kontinuierlich an
        }
}
```

Die Initialisierung des gesamten Systems wird im Hauptprogramm vorgenommen und beinhaltet das Aufsetzen der Zeitbasis, das Laden der Task Control Blöcke, das eigentliche Starten des Timers und die Bestimmung des Basistasks.

Im Hauptprogramm wird zunächst der, dem Echtzeitsystem als Zeitbasis zugeordnete, Timer initialisiert

Funktion RO_TimerSetup();

Die Zeitbasis wird über eine define Vereinbarung festgelegt z.B.:

```
#define RO_Tim0rel  -20000          // Bei einem prescaled Takt von 1 MHz
                                    ergibt sich ein Timerereignis alle 20 ms
```

```
//main function
void main()
{
        //HKaRO Initialization
        TMOD=0;
        RO_TimerSetup();

        //Task Initialization
        //RO_StartTask(Startadresse,Priority,ID,    State)
        RO_StartTask  (Tsk1,        1,      1,      RO_STREADY);
        RO_StartTask  (Tsk2,        1,      2,      RO_STREADY);

        //Run Application
        RO_SetCritical();     //Timer 0 start
        TR0=1;
        RO_ExitCritical();

        RO_BaseTask(1);     //Run Base Task
}
```

Zur Festlegung der Taskdaten werden die Taskadressen angegeben und zusätzlich die Priorität sowie eine Kennung und der Startzustand definiert.

Funktion	RO_StartTask(((Startadresse Priority,ID, State)	
Parameter	Startadresse	Verweis auf den Anfang der Taskfunktion
	Priority	Priorität des Tasks 1 niedrigste Priorität, 255 höchste Priorität 0 nicht erlaubt
	ID	Kennzeichennummer des Task z.Zt. nur bei der Funktion RO_BaseTask verwendet
	State	Zustand des Tasks RO_STREADY -> bereit zum laufen

Das gesamte System wird nach Starten der Zeitbasis mit TR0=1 und der Aktivierung des Interruptsystems gestartet. Das Interruptsystem wird im vorliegenden Fall über die Funktionen
RO_SetCritical();(setzt EA=0) und RO_ExitCritical(); (setzt EA=1) ein- und ausgeschaltet.

Der eigentliche Start der Applikation wird über die Aktivierung des Basistasks vorgenommen.

Funktion:	RO_BaseTask(id);	
Parameter	id	Kennzeichennummer des Basistasks

Damit werden die Tasks in Abhängigkeit von ihrem Zustand vom Scheduler aktiviert.
Eine Ablaufsequenz ist in Bild 8.13 dargestellt.

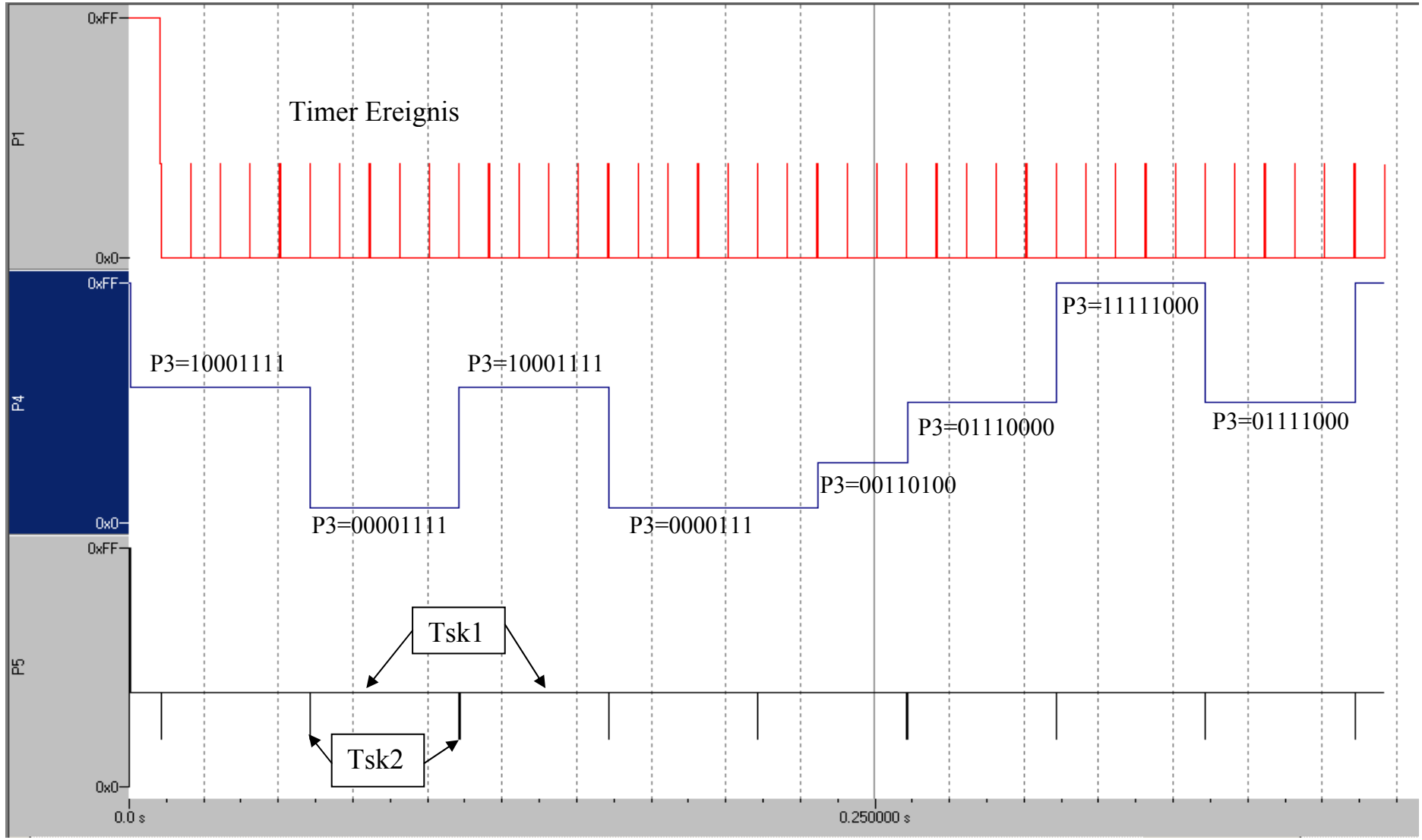


Bild 8. 13 Ablaufsequenz

9.3.2 Prioritäten

Die Verteilung von Prioritäten ist natürlich von der Anwendung abhängig. Erfahrungswerte zeigen jedoch, dass zumindestens eine erste Verteilung der Prioritäten nach groben Regeln erfolgen kann.

Ein entsprechender Vorschlag ist der folgenden Liste zu entnehmen:

Priorität	Task Typ
0	Nicht erlaubt
1	Warte Task
bis 64	Ausgabe Tasks
bis 128	Allgemeine Verarbeitung
bis 192	Eingabe Tasks
bis 255	Interrupt Tasks

Die Priorität 0 wird vom System zur Kennzeichnung von Tasks im Scheduler verwendet. Sie darf daher nicht vom Benutzer benutzt werden. Die niedrigste verwendbare Priorität ist damit 1. Sie wird üblicherweise für den Wartetask verwendet, der immer dann gestartet wird, wenn kein anderer Task lauffähig ist. Das Betriebssystem benötigt zur fehlerfreien Funktion immer mindestens einen Task, der gestartet werden kann.

9.3.3 Wartebedingungen

Die bisher angesprochenen Funktionen sind geeignet, quasi parallele Abläufe von zwei oder mehreren Prozessen (Tasks) zu realisieren. In Echtzeitsystemen spielen jedoch feste Zeitzuordnungen oder ereignisgesteuerte Abläufe eine wesentliche Rolle. Wird die Möglichkeit geschaffen, auf bestimmte Ereignisse zu warten, so lassen sich auch komplexe Vorgänge einfacher beschreiben. Die eigentliche Abfolge wird dann vom Betriebssystem selbst organisiert.

Interrupts:
Interrupts können als solche Ereignisse gehandhabt werden. Es lassen sich so Tasks an Interruptereignisse binden, die nur dann aktiv werden, wenn ein solcher Interrupt ausgelöst wird. Eine umfangreiche Aktion muss so nicht mehr in der Interrupt Service Routine selbst durchgeführt werden. Die Aufgabe einer solchen Routine besteht dann nur noch darin, den entsprechenden Task zu aktivieren. Sollen kurze Antwortzeiten erreicht werden, kann dieser Task mit einer höheren Priorität ausgestattet werden. Bei preemptiven Systemen wird ja dann sofort der Task mit der höchsten Priorität ausgeführt.

Messages:
Entsprechend den Interrupts stellen Nachrichten zwischen Tasks (Messages) eine Möglichkeit dar, in Abhängigkeit von einem bestimmten Zustand die Abarbeitung einer weiteren Aufgabe zu veranlassen. Ist die Weiterführung der Taskaufgabe dann von einem Ereignis in einem anderen Task abhängig, so kann auch hier ein Wartezustand herbeigeführt werden.

Delays:
Zur Erfüllung zyklisch wiederkehrender Aufgaben ist es sinnvoll einen Task eine bestimmte Zeit warten zu lassen. Das Echtzeitbetriebssystem besitzt bereits eine Einheit zur Vorgabe eines Zeitraumes. Dieser Zeitraum kann für einen Task vom Scheduler gezählt werden und nach Ablauf einer bestimmten Zeit eine erneute Aktivierung erfolgen.

In allen Fällen wird auf ein bestimmtes Ereignis gewartet. Zur Kennzeichnung eines Tasks im Wartezustand wird im Statusbyte eine Kennung für die Art des Wartezustands abgelegt und zusätzlich wird eine Kennung ermöglicht, welche die Art des Wartezustands spezifiziert. Damit kann auf unterschiedliche Interrupts, Nachrichten oder Delays gewartet werden.

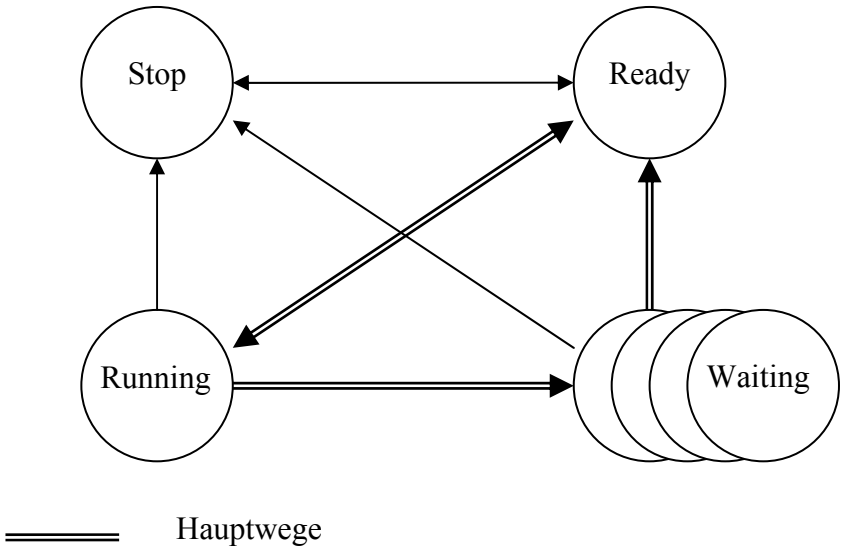


Bild 8.14 Taskübergänge

9.3.3.1 Messages

Nachrichten werden verwendet um wechselseitige Abhängigkeiten von Tasks verarbeiten zu können. Werden Teilaufgaben in einem Task erledigt, auf dessen Ergebnis ein anderer Task warten muss, so können Nachrichten verwendet werden um die Abarbeitungsreihenfolge festzulegen.
Zum Austausch von Nachrichten zwischen Tasks werden zwei Funktionen zur Verfügung gestellt:

```
RO_WaitMsg(mRO_Event);
RO_SendMsg(mRO_Event);
```

RO_WaitMsg versetzt den Task in einen Wartezustand. Konkret wird das Statusbyte mit der entsprechenden Kennung versehen und der Scheduler zur Auswahl eines anderen Tasks aufgerufen.

RO_SendMsg liefert hierzu den entsprechenden Wert und ruft den Scheduler mit der konkreten Nachrichtenkennung in der Variablen gRO_Event auf.
Sendet ein Task eine Nachricht und ist kein weiterer Task vorhanden, der auf diese Nachricht wartet, so geht die Nachricht verloren. Zur Kennzeichnung der Tatsache, dass wenigstens ein Task die Nachricht erhalten hat, wird die Variable gRO_MSGAck eingeführt. Sie wird auf 1 gesetzt, wenn ein Task aufgrund der gesendeten Nachricht in den Ready-Zustand versetzt wurde. Sonst hat die Variable den Wert 0.

```
INT8U  data    gRO_MSGAck;              //Message was accepted by a task
```

9.3.3.2 Delays

Zum Aufbau von zyklischen Abläufen kann die Zeitbasis des Echtzeitbetriebssystems herangezogen werden. Wird ein Task in einen Wartezustand versetzt, so wird der Scheduler veranlasst, einen speziellen Zähler für diesen Task herunter zu zählen. Beim Erreichen der gewünschten Wartezeit wird der Task wieder gestartet.

Funktion:	RO_WaitDel(mRO_Index,mRO_DelInit);	
Parameter	mRO_Index	zugeordneter Zähler
	mRO_DelInit	Wartezeit als Vielfaches der Zeitbasis

Beispiel zu Wartefunktionen

Mit den jetzt bekannten Funktionen kann die Ampelsteuerung in einer programmtechnisch einfacheren Form umgesetzt werden. So ist es möglich die Zeitbasis der Ampel jetzt auch durch den Grundtakt des Echtzeitbetriebssystems zu erzeugen. Dies kann mit in den Task aufgenommen werden, der die Statusauswertung vornimmt. Auch die Abfrage der Tasten kann an eine Wartezeit gebunden werden, um den Prozessor zu entlasten. Soll zusätzlich gewährleistet werden, dass die Autos eine gewisse Mindestzeit fahren sollen, kann ein weiterer Task benachrichtigt werden, der eine erneute Anforderung verzögert. Dies soll über Nachrichten organisiert werden.

Aufgaben der einzelnen Tasks:

- Task 1 Abfragen der Tasten alle 100 ms
- Task 2 Ausgabe neuer Signale und Weiterschalten des Zustandes alle Sekunde
- Task 3 Verzögerung einer direkten neuen Anforderung (Verzögerung 10 s)
- Task 4 Wartetask

Wird mit einer Basiszeit von 20 ms gearbeitet, so ergeben sich folgende Wartezeiten für die einzelnen Aufrufe von RO_WaitDel:

Task	Zeit	Wartezähler	Zusatz
Task 1	100 ms	5	
Task 2	1000 ms	50	
Task 3	10000 ms	500	2*250
Task 4	-	-	

Damit ergeben sich die folgenden Definitionen der Tasks.

Task zum Einlesen der Tasten:

```
//Zyklisches Einlesen der Tasten an P4.0 und P4.1
void Tsk1()
{
        while(1){
                        RO_WaitDel(2,5);
                        v1=P4;
                        v1&= 0x03;
                        if(v1!=3 && astate==0){
                                RO_SendMsg(30);
                                RO_WaitMsg(20);
                        }
        }
}
```

Abarbeitung der Zustände:

```
//Zyklisches Abarbeiten des Zustandes
void Tsk2()
{
   while(1){
            RO_WaitDel(0,5); //RO_WaitDel(0,50); Reduced for simulation
            if     (astate==0){
                                                    P4=FrotAgruen;
                                                    RO_WaitMsg(30)
                                                    astate=1;
            }else if(astate==1){                    P4=FrotAgruen;;};
            }else if(astate==2){                    P4=FrotAgelb;
            }else if(astate==3){                    P4=FrotArot;
            }else if(astate>=4 && astate<=13){      P4=FgruenArot;
            }else if(astate==14){                   P4=FrotArot;
            }else if(astate==15){                   P4=FrotArt_ge;
            }else if(astate==16){                   RO_SendMsg(10);
            }else{                                  /*Error*/
            }
            if(astate!=16)      astate=astate+1;
            else                astate=0;
   }
}
```

Hauptprogramm mit Wartetask und Einschaltverzögerung:

```
//Einschaltverzögerung
void Tsk3()
{
        while(1){
                RO_WaitMsg(10);
                RO_WaitDel(1,25);   //RO_WaitDel(1,250); Reduced for Simulation
                RO_WaitDel(1,25);   //RO_WaitDel(1,250); Reduced for Simulation
                RO_SendMsg(20);
        }
}
//Wartetask
void Tsk4()
{
        while(1){
        }
}

//main function
void main()
{
        //HKaRO Initialization
        TMOD=0;
        RO_TimerSetup();

        //Task Initialization
        //RO_StartTask((PCH,PCL)  ,RO_oP      ,RO_oID      ,RO_oState)
        RO_StartTask(Tsk4         ,1          ,4           ,RO_STREADY);
        RO_StartTask(Tsk1         ,2          ,1,          RO_STREADY);
        RO_StartTask(Tsk2         ,2          ,2,          RO_STREADY);
        RO_StartTask(Tsk3         ,5          ,3,          RO_STREADY);

        //Run Application
        RO_SetCritical();
        TR0=1;
        RO_ExitCritical();
        RO_BaseTask(4);
}
```

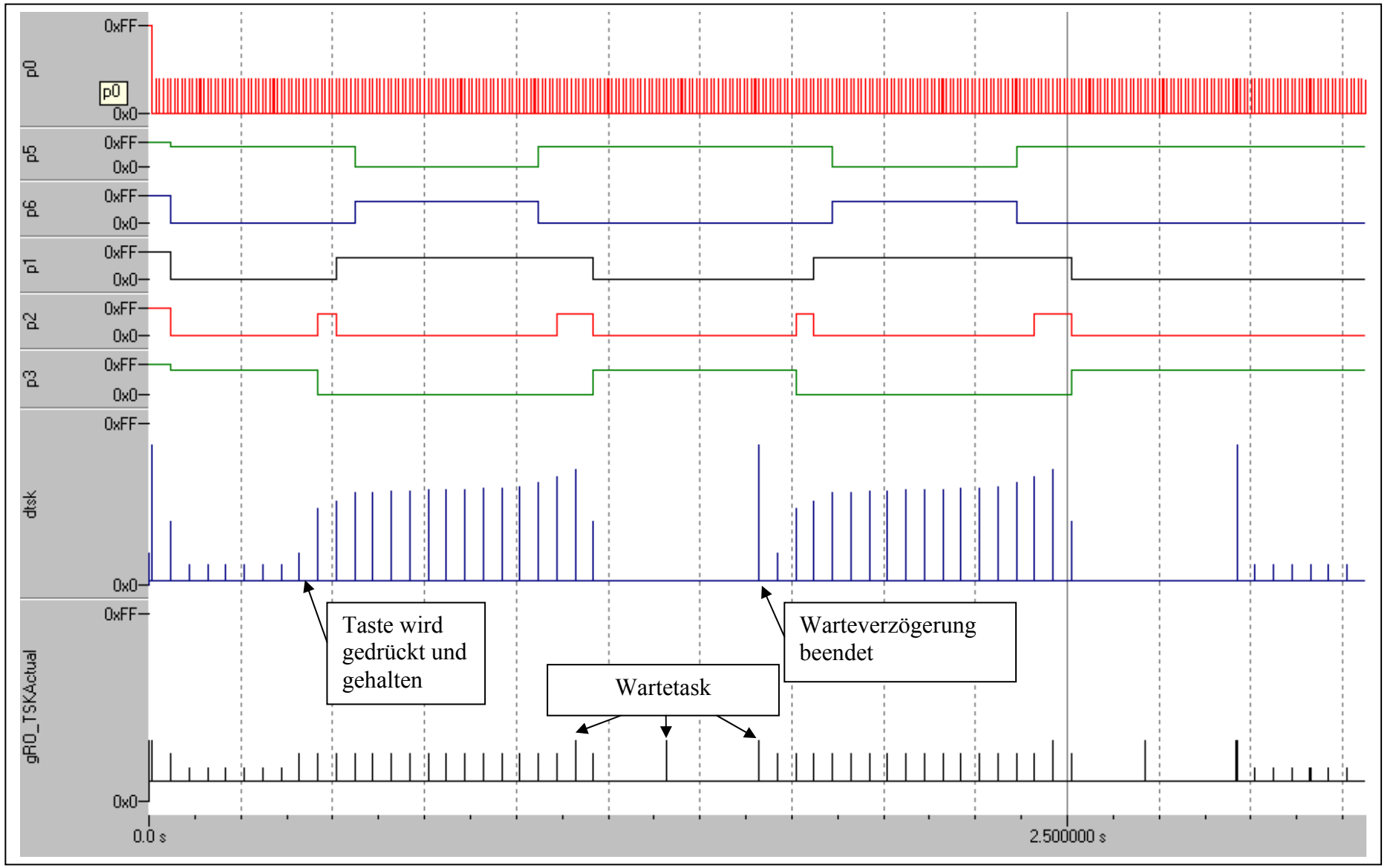


Bild 8.17 Ablauf der Ampelsteuerung beim Einsatz von Nachrichten und Warteoperationen

9.3.3.3 Semaphore

Sollen von verschiedenen Tasks gleiche Ressourcen verwendet werden, so kann der quasi gleichzeitige Zugriff zu Konflikten führen. Soll beispielsweise ein Speicherbereich von einem Task beschrieben und von einem anderen Task zur Ermittlung eines Ergebnisses verwendet werden, so ist es nicht sinnvoll, den Bereich zu bearbeiten, bevor der Schreibvorgang beendet ist. Eine Möglichkeit diese Situation zu regeln, ist das Versenden von Nachrichten zwischen den beteiligten Tasks zur Vermeidung von Zugriffskonflikten.

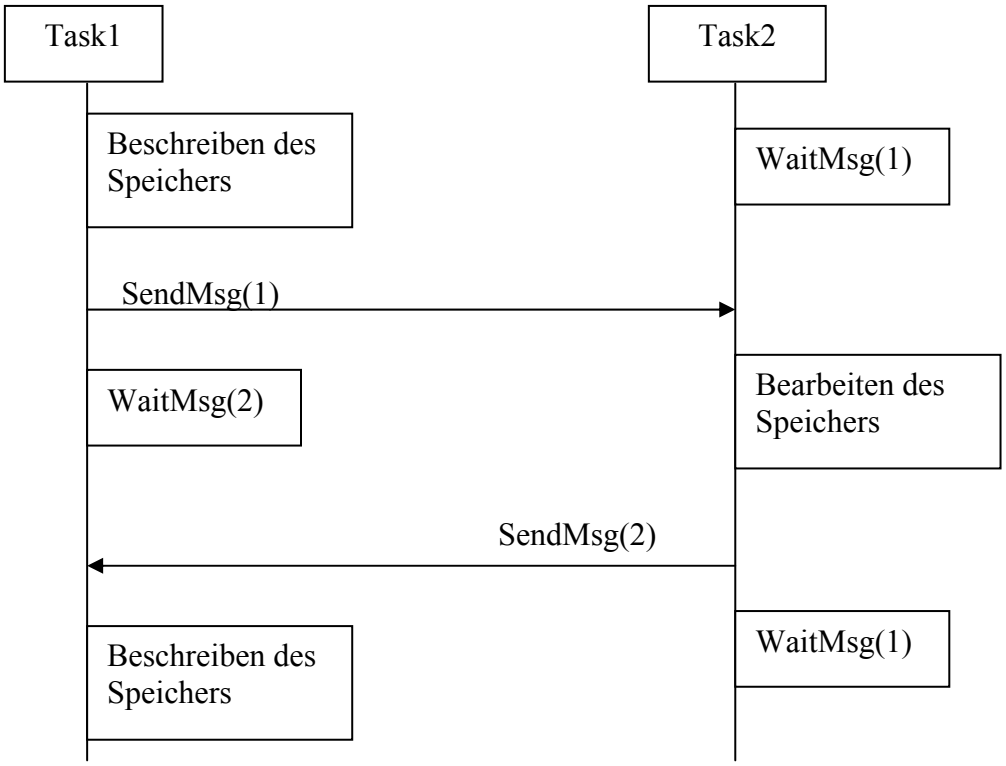


Bild 8.18 Nachrichtenaustausch zur Vermeidung von Konfliktsituationen

Das Verwenden von Nachrichten setzt die genaue Kenntnis der beteiligten Tasks voraus und auch deren Aufgaben.
Eine flexiblere Form der Zugriffsregelung stellen Semaphoren dar. Edgser Dijkstra führte diesen Begriff um 1965 ein. Eine Semaphore kann als Schlüssel verstanden werden, den ein Task braucht, um weiter arbeiten zu können. Nur der Task, der den Schlüssel besitzt, darf dann beispielweise auf eine Ressource zugreifen. Dazu müssen nur die Möglichkeiten geschaffen werden:

- Einen freien Schlüssel zu belegen.
- Den Schlüssel freizugeben.
- Auf die Freigabe des Schlüssels zu warten.

Zur Abspeicherung der Semaphoren wird in dieser Version ein Byte angelegt. Damit können 8 Semaphoren mit den Nummern 0 bis 7 verwendet werden.

Zur Bedienung der Semaphoren werden zunächst zwei Funktionen zur Verfügung gestellt.

Belegen einer Semaphore mit der Nummer mRO_SemN

Funktion RO_GetSem(mRO_SemN);

Die Funktion überprüft, ob die angesprochene Semaphore belegt ist.
Ist sie frei, wird die Semaphore belegt. Wird eine Belegung vorgefunden, geht der Task in den Wartezustand und wartet bis die Semaphore freigegeben wird.

Freigabe einer Semaphore mit der Nummer mRO_SemN

Funktion RO_RelSem(mRO_SemN);

Die Funktionen bergen jedoch 2 Gefahren in sich.

1. Begibt sich der Task in den Wartezustand auf eine Semaphore und die Semaphore wird nicht mehr freigegeben, besteht nur noch durch einen Rücksetzprozess die Möglichkeit den Task wieder zu aktivieren.
2. Warten mehrere Tasks auf eine Semaphore, werden sie alle gleichzeitig bereit gemacht, obwohl nur ein Task vernünftig zugreifen kann.

Die beiden vorgestellten Funktionen sind daher nur für die Kommunikation von zwei Tasks und für Verarbeitungssituationen bei denen die Freigabe der Semaphore in jedem Fall sicher gestellt ist, geeignet.
Ein Ausweg aus dieser Situation würde eine Funktion bieten, die nur einen Versuch unternimmt die Semaphore zu belegen und nicht in einen Wartezustand geht. Wird das Ergebnis des Versuchs dem aufrufenden Task bekannt gemacht, können dort Entscheidungen getroffen werden, ob und in welcher Form und wie lange gewartet werden soll.
Es können damit auch Timeouts programmiert werden. Auch der Zugriff über mehrere Tasks ist jetzt möglich, da bei einem Task im Wartezustand kein Semaphorenereignis für den Scheduler gekennzeichnet wurde.
Die zusätzlich eingeführte Variable gRO_MSGAck zeigt an, ob die Semaphore belegt war.
gR0_MSGAck enthält 0 wenn die Semaphore nicht benutzt wird oder eine Zahl>0 wenn sie benutzt wird.
Diese Variable kann dann vom Task überprüft werden, ob die Belegung erfolgreich war. War die Semaphore frei, wurde sie jetzt bereits für den Task reserviert.

Test, ob eine Semaphore belegt ist - ohne Wartezustand
Funktion: RO_GetNWSem(mRO_SemN,mRO_MSGAck);\

Das folgende Programmbeispiel zeigt den Aufbau einer Timeoutschleife über ein Wait for Delay. Es sind aber auch direkte Verzögerungsschleifen (v=1000;while(v)v--;) oder eine Kombination denkbar.

```
INT8U v1tsk5;
INT8U v2tsk5;
void Task5()
{
        while(1){
                v2tsk5=10;     //10 Zugriffsversuche
                while(v2tsk5){
                        RO_GetNWSem(1, v1tsk5);
                        if(v1tsk5!=0){ //Semaphore 1 wird benutzt
                                RO_WaitDel(2,20);
                        }else{
                                break;
                        }
                        v2tsk5--;
                }
                if(v2tsk5==0){
                        //timeout
                }else{
                        //Aktion
                }
        }
}
```

Beim Zugriff durch mehrere Tasks muss nach dem Wartevorgang eine erneute Anfrage erfolgen. Dies kann in einer Schleife geschehen, die erst verlassen wird, wenn ein erfolgreicher Zugriff durchgeführt wurde.

```
INT8U v1tsk5;

void task5 ()
{
        while(1){
                …
                do{
                        RO_GetSem(1);
                        RO_GetNWSem(1, v1tsk5);
                while(v1tsk5!=0);
                …
        }
}
```

Beispiel:
Die Verwendung von Semaphoren soll am Zugriff auf einen gemeinsam verwendeten Speicher demonstriert werden. Zur Regelung des Zugriffes wird eine Semaphore als Schlüssel verwendet. Ein Task liest Daten von einem Port ein und speichert die Daten in den Speicher. Der Speicher soll eine feste Länge haben und einen Zugriffspointer besitzen. Der eingelesene Wert wird an die Stelle des Zugriffspointers geschrieben und der Zugriffspointer wird erhöht. Wurde der letzte Platz erreicht, so wird wieder am Beginn des Feldes gestartet.
Ein weiterer Task soll zyklisch den Mittelwert aus allen Werten bestimmen.

```
//Speicherbereich:
        INT8U       idata   dat[10];                //Datenbereich
        INT8U       data * idata    ept= dat+10;    //Eins nach dem letzten Element
        INT8U       data * idata    npt= dat;       //Loop pointer
//Zugriffspointer:
        INT8U       data * idata    pt=ept;
//Mittelwert:
        INT16U      data    mean;
```

```
//Bestimmung des Mittelwertes
void Tsk1(){
        while(1){
                RO_WaitDel(0,6);
                RO_GetSem(1);
                npt=dat;
                mean=0;
                while(npt!=ept){
                        mean=mean+ *npt;
                        npt++;
                }
                mean=mean/10;
                RO_RelSem(1);
        }
}
//Einlesen der Werte
void Tsk2(){
        while(1){
                RO_WaitDel(1,2);
                RO_GetSem(1);
                if(pt==ept)pt=dat;
                *pt=P4; pt++;
                RO_RelSem(1);
        }
}
//Wartetask
void Tsk3(){
        while(1){P1=mean;}
}
```

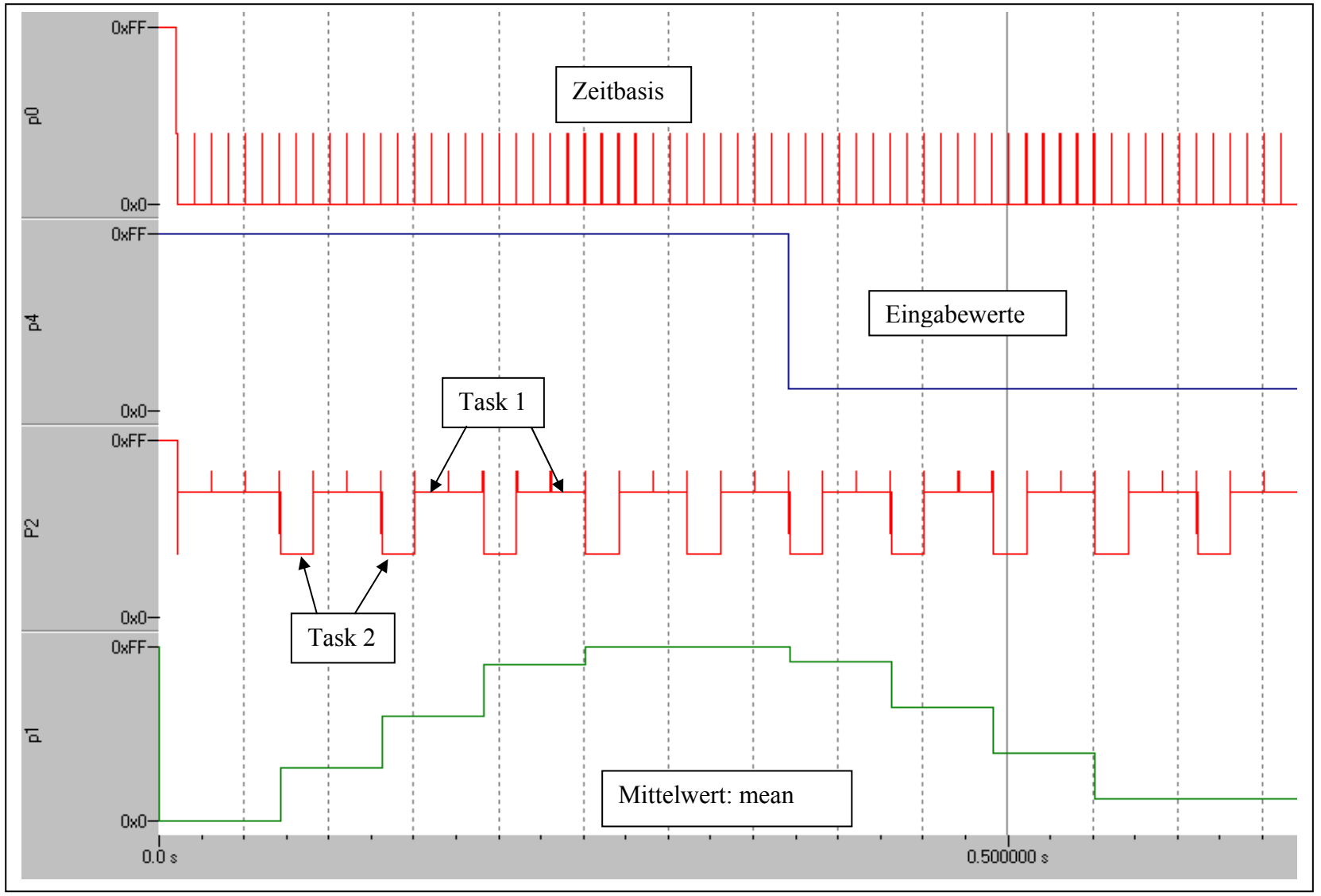


Bild 8.l9 Zugriffsverwaltung eines Speichers mit Semaphoren

9.3.3.4 Interrupts

Für den Aufbau des Grundtaktes wird bereits bei der Zeitbasis ein Interruptsignal verwendet. Grundsätzlich unterscheiden sich andere Interruptquellen nicht von dieser Interruptquelle. Soll auch bei diesen anderen Interruptquellen eine sofortige Reaktion erfolgen können, so kann das bereits verwendete Schema auch hier angewendet werden. Sinnvoll ist das Auffangen des Interrupts in einer Assemblerfunktion, die Bildung einer kritischen Sektion und den Aufruf des Schedulers. Der Scheduler muss jedoch unterscheiden können, ob der Interrupt aus seiner Zeitbasis heraus (Timer 0) oder von einem anderen Ereignis ausgelöst wurde. Hierzu wird eine zusätzliche Variable eingeführt, die in der jeweiligen Assemblerfunktion mit einer entsprechenden Kennung geladen wird. Diese Kennung kann auch dazu verwendet werden um einen Task, der auf einen solchen Interrupt wartet zu starten. Mit diesem Hintergrund wurden folgende Definitionen vorgenommen:

```
INT8U data  gRO_Event      //Event Variable
```

Werden Interrupts von anderen Quellen als von Timer 0 ausgelöst, so enthält die Variable gRO_Event die Art und die Nummer des Interrupts, die sich aus der Durchnummerierung der Interrupts ergibt oder sich aus folgender Formel berechnen lässt:

N=(Interrupt_Einsprungadresse – 3)/8

Für den Interrupt von Timer 1 ergibt sich dann:
010nnnnn = RO_STWAITINT+nnnnn
01000011 = ROa_STWAITINT+3

Beispielhaft ist in der folgenden Auflistung auch die Interruptfunktion für den Timer 1 angegeben.

```
$nomod51
$include(C8051F340.inc)
PUBLIC  ROa_RSTWA  ;Restores working area
PUBLIC  ROa_SAVEWA ;Saves working area
EXTRN code ( RO_Scheduler);Scheduler written in C
EXTRN data ( gRO_Event)    ;Event variable defined in C-Code Type INT8U
EXTRN data ( gRO_PCH)      ;save PCH, variable defined in C-Code Type INT8U
EXTRN data ( gRO_PCL)      ;save PCL, variable defined in C-Code Type INT8U
EXTRN data ( gRO_SP)       ;save Stackptr., variable defined in C-Code Type *INT8U
EXTRN data ( gRO_TSKActual);actual TCP, variable defined in C-Code Type *INT8U
ROa_EVSCHEDULER       EQU   32*1 + 1
ROa_STWAITINT         EQU   32*2
ROa_Tim0rel           EQU   -20000
ROa_TCBSIZE           EQU   18
USa_Tim1rel           EQU   -50000
;------------------------------------------------------------------------
;CSEG at 0003h                  ;  Int 0 - _Int0 Interrupt
;       RETI
CSEG at 0bh                    ;  Int 1 - Timer0 Interrupt
ISR_Timer0:
       clr EAL                 ; Enter critical section
       jmp ROa_ISR_Timer0Code
CSEG at 0013h                  ;  Int 2 - _Int1 Interrupt
       RETI
CSEG at 001Bh                  ;  Int 3 - Timer1 Interrupt
ISR_Timer1:
       clr EAL                 ; Enter critical section
       JMP ISR_Timer1Code;
;Interrupt level context switch entry point
?PR?OSIntCtxSw    SEGMENT CODE
       RSEG  ?PR?OSIntCtxSw
ISR_Timer1Code:          ;          Int 3 - Timer1 Interrupt
       Clr     TR1
       Mov    TH1,#High(USa_Tim1rel)          ;Reload value Timer 1
       Mov    TL1,#Low (USa_Tim1rel)
       SETB TR1
       Mov gRO_Event,#ROa_STWAITINT+3;Set Int. Event: State+Interrupt Number
       call RO_Scheduler                ;Activate Interrupt Process
       Setb EAL                        ;Enable all Interrupts; Leave critical section
       RETI                            ;Return from Interrupt
;------------------------------------------------------------------------
ROa_ISR_Timer0Code:          ;     Int 1 - Timer0 Interrupt
       Clr     TR0
       Mov   TH0,#High(ROa_Tim0rel)          ;Reload value Timer 0
       Mov   TL0,#Low (ROa_Tim0rel)
       SETB TR0
       Mov gRO_Event,#ROa_EVSCHEDULER  ;denote Scheduler Event
       call RO_Scheduler               ;perform context switch
       Setb EAL                        ;Leave critical section
       RETI
```

Sollen auf diese Weise angesprochene Prozesse auch mit höherer Priorität gestartet werden können, so muss der Scheduler zum einen den Task ausfindig machen, der auf ein solches Ereignis reagieren kann und zusätzlich feststellen, ob der damit aktivierte Task auch die höchste Priorität besitzt. Dazu muss der Task sich vorher in einen spezifischen Wartezustand gebracht haben. Der Task ruft hierzu eine spezielle Wartefunktion (RO_WaitInt) auf, die im vorliegenden Fall das Warten auf einen Interrupt kennzeichnen soll:

Wurde der Scheduler durch einen Interrupt aktiviert, so ist es üblich einen Task mit einer höheren Priorität zu starten. Durch die preemptive Eigenschaft des vorliegenden Schedulers wird dieser Task sofort nach dem Interrupt gestartet und damit der gerade aktive Task unterbrochen.

Beispiel:
Die bereits bekannte Ampelsteuerung soll in ihrem zeitlichen Verhalten nicht mehr von der Zeitbasis des Betriebssystems sondern vom Timer 1 abhängig sein. Dazu wird die Interruptannahme über Tsk3 realisiert.

Interruptannahme über den Tsk3

```
void Timer1_Setup()
{       /* setup timer 1 */
        TMOD |= 0x10;                       /* Timer 1, Modus 1, 16 Bit Zähler */
        TL1  = TIMER1RELOAD & 0x00ff;//Low Byte des Nachladewerts erzeugen
        TH1  = TIMER1RELOAD >> 8;     //High Byte des Nachladewerts erzeugen
        ET1  = 1 ;                          /* enable timer 1 interrupt */
        TR1  = 0;                           //Hold Timer 1
}


void Tsk3()
{
        while(1){
                RO_WaitInt(3);
                timercount=timercount+1;
                if(timercount==3){
                        astate = astate+1; if(DEB) dastate= astate <<3;
                        timercount=0;
                }

        }
}
```

Tsk1 und Tsk 2 bleiben erhalten:

```
void Tsk1(){   //Zyklisches Einlesen der Tasten an P4.0 und P4.1
   while(1){
        v1=P4;
        v1&= 0x03;
        if(v1!=3 && astate==0){TR1=1;}
   }
}
void Tsk2(){//Zyklisches Abarbeiten des Zustandes
   while(1){
      if       (astate==0){                    P4=FrotAgruen;
      }else if(astate==1){                     P4=FrotAgruen;
      }else if(astate==2){                     P4=FrotAgelb;
      }else if(astate==3){                     P4=FrotArot;
      }else if(astate>=4 && astate<=13){       P4=FgruenArot;
      }else if(astate==14){                    P4=FrotArot;
      }else if(astate==15){                    P4=FrotArt_ge;
      }else if(astate==16){                    TR1=0; astate=0;
      }else{                                   /*Error*/
      }
   }
}
```

Im Hauptprogramm muss zusätzlich Tsk 3 gestartet werden:

```
void main()
{
        //HKaRO Initialization
        TMOD=0;
        RO_TimerSetup();

        //Task Initialization
        //RO_StartTask((PCH,PCL),RO_oP,RO_oID,RO_oState)
        RO_StartTask(Tsk1,1,1,RO_STREADY);
        RO_StartTask(Tsk2,1,2,RO_STREADY);
        RO_StartTask(Tsk3,3,3,RO_STREADY);
        Timer1_Setup();

        //Run Application
        RO_SetCritical();
        TR0=1;
        RO_ExitCritical();
        RO_BaseTask(1);
}
```

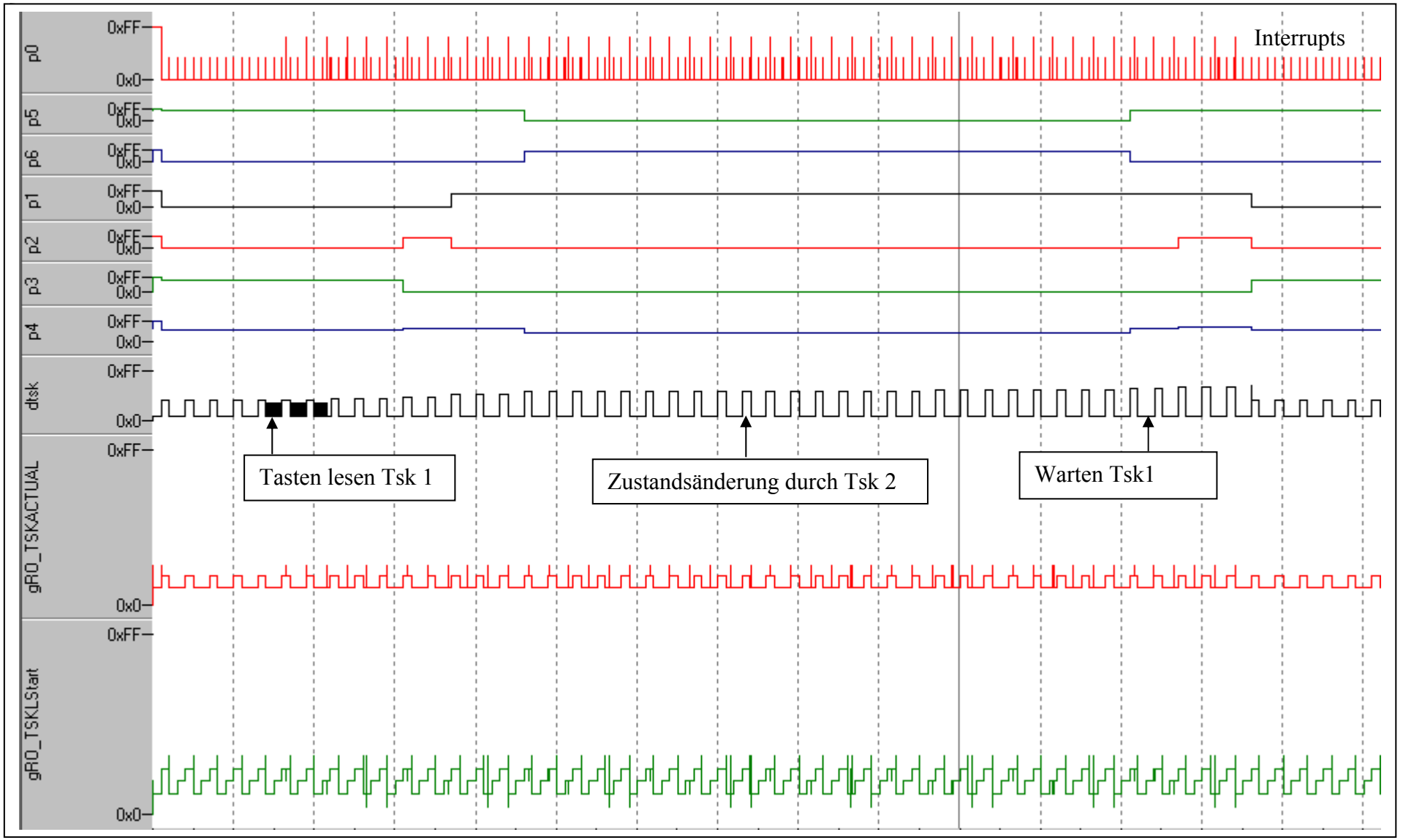


Bild 8.16 Ablauf der Ampelsteuerung mit RO_WaitInt

9.3.4 Unterprogrammaufrufe

In den vorhergehenden Abschnitten wurde angenommen, dass Unterprogrammaufrufe in den Tasks nicht vorkommen dürfen. Bei der allgemeinen Zulassung von Unterprogrammaufrufen wird der Stack bei der Ausführung eines Tasks mit zusätzlichen Daten und Rücksprungadressen belegt. Wird die Zulassung von Unterprogrammaufrufen gefordert, so muss nicht nur die gerade aktuelle Programmadresse, sondern der gesamte Bereich, der jetzt zum Task gehört, mit abgespeichert werden. Das ist grundsätzlich möglich, jedoch relativ aufwendig. Sollen die Daten im Task Control Block untergebracht werden, so kann entweder nur eine feste Anzahl von Daten aufgenommen werden oder der gesamte TCB muss dynamisch erweitert werden können. Bei der Reservierung von festen Datenbereichen führt dies zu einem starken Bedarf an Speicherplatz, da hier die größte Dichte an Unterprogrammaufrufen angenommen werden muss. Die dynamische Verwaltung von beliebigen Speicherplatzblöcken führt zu den bekannten Problemen der Zerstückelung des Speicherbereiches und daher zu einem hohen Aufwand in der Verwaltung. Ein Kompromiss stellt die Verkettung von einzelnen Bytes in Form einer Liste dar. Der Vorteil ist die einfache Verwaltung freier Bytes sowie deren Zugriff. Einen großen Nachteil stellt der erhebliche Bedarf an Verwaltungsspeicherplatz dar. Die Hälfte des Speicherplatzes des Bereiches muss dafür geopfert werden. Trotzdem stellte sich bei eingehender Untersuchung auch noch anderer Organisationsstrukturen diese Form des Speichers als einen guten Kompromiss aus Programmaufwand und Speicherplatzbedarf dar. Dies gilt besonders im Hinblick auf die Anwendung innerhalb eines Mikrocontrollers.
Die in der Vorlesung und dem zugeordneten Labor verwendete Version des Echtzeitbetriebssystems beinhaltet die Möglichkeit der allgemeinen Verarbeitung von Unterprogrammen zunächst nicht. In der eingeschränkten Version ist es jedoch möglich für einen Task Unterprogramme zuzulassen.

9.3.4.1 Eingeschränkte Unterprogrammverwendung

Ist eine Applikation vom Programmablauf so aufteilbar, dass nur in einem Task Unterprogrammaufrufe vorkommen, so können auch in der bis hierher vorgestellten Version des Echtzeitbetriebssystems Unterprogrammaufrufe in eingeschränkter Form eingesetzt werden. Es gibt dann einen Task der Unterprogrammaufrufe in beliebiger Schachtelung enthalten kann, während alle anderen Tasks keine Unterprogramme enthalten dürfen. Werden Tasks ohne Unterprogrammaufrufe aktiviert, können die abgelegten Rücksprungadressen und Daten auf dem Stack verbleiben und es muss nur die gerade aktuelle Programmadresse geändert werden, die dann auch in einem Unterprogramm liegen kann. Diese wird dann in bekannter Form gesichert und später beim Aufruf wieder zurückgespeichert. Für die anderen Tasks gelten die bereits bekannten Vorgehensweisen.

9.3.4.2 Reentrant Funktionen

Bei der Verwendung von Unterprogrammen in Tasks ist es zunächst nicht verboten, Funktionen, die einem allgemeinen Zweck dienen z.B. strcpy (Stringkopierfunktion) in mehreren Tasks zu verwenden. Durch die Aktivierung und Suspendierung der Tasks müssen diese Funktionen eine Eigenschaft haben, die als reentrant bezeichnet wird. Dies bedeutet, dass alle Daten, die eine solche Funktion verwenden, auch exklusiv diesem Aufruf zugeordnet werden können. Werden globale Variable benutzt, ist diese Forderung nicht erfüllt. Beim Keil-Compiler werden Variable üblicherweise auch Speicherplätzen zugeordnet, die dann eigentlich globalen Variablendefinitionen entsprechen. Durch eine Compilerdirektive kann jedoch angegeben werden, dass die Eigenschaft reentrant eingehalten werden muss. Diese Eigenschaft führt aber zu einem erhöhten Programmaufwand und der Verwendung eines gesonderten Stacks, um die Variablen zwischenspeichern zu können.

Beispiel 1:
Die nachfolgende Definition von strcpy kann nicht von mehreren Stellen aus gleichzeitig aufgerufen werden, da die globale Variable c im Kopiervorgang verwendet wird.

```
char c;
void strcpy(char *s1,char *s2)
{
        c= *s2;
        while(c!=0){
                 *s1 = c
                s1++;
                s2++;
                c=*s2;
        }
}
```

Beispiel 2:
Die nachfolgende Definition von strcpy kann von mehreren Stellen aus gleichzeitig aufgerufen werden, da eine lokale Variable im Kopiervorgang verwendet wird und die Eigenschaft reentrant vom Compiler eingefordert wird.

```
void strcpy(char *s1,char *s2) reentrant
{
        char c;

        c= *s2;
        while(c!=0){
                 *s1 = c
                s1++;
                s2++;
                c=*s2;
        }
}
```

9.3.4.3 Prioritätsinversion

Bei der Verwendung von Semaphoren zur Regelung des Zugriffs von mehreren Tasks auf eine Ressource kann es bei unterschiedlichen Prioritäten der Tasks zu einem Effekt kommen, der Prioritätsinversion genannt wird. Wird die Ressource durch einen Task mit einer niederen Priorität belegt, so ist ein anderer Task, auch wenn er eine höhere Priorität hat, nicht in der Lage seine Aufgabe durchzuführen. Er wird also faktisch in der Priorität hinter den Task mit der niedrigeren Priorität zurückgesetzt (Prioritätsinversion).
Bild 8.24 zeigt die Wirkung einer Prioritätsinversion im Ablauf beim Zugriff auf eine Ressource.

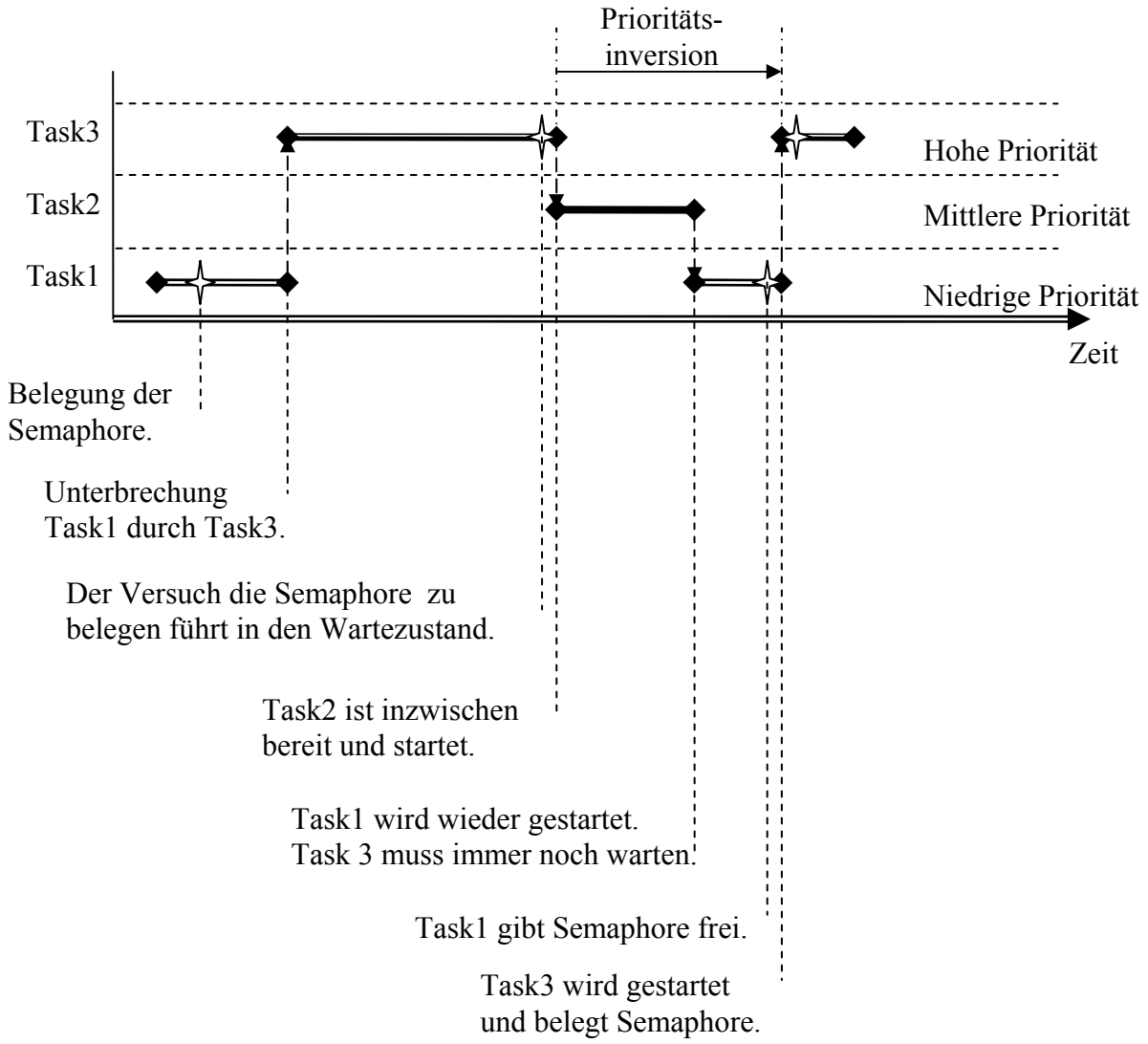


Bild 8.24 Prozessablauf bei einer Prioritätsinversion

Die Priorität des Tasks 3 wurde faktisch auf die Priorität von Task 1 reduziert. Durch den erzwungenen Wartezustand musste Task 3 auch noch auf das Ende der Ausführung von Task 2 warten. Das Problem kann umgangen werden, wenn man kurzzeitig die Priorität von Task 3 so erhöhen würde, dass er die höchste Priorität von allen Tasks besitzt, die um die Semaphore konkurrieren. Wird im vorliegenden Beispiel so vorgegangen, so tritt der gewünschte Effekt ein. Erfolgt die Aktivierung von Task 3 jedoch nicht, so könnte Task 2, obwohl er nichts mit der Semaphore zu tun hat, auch nicht gestartet werden.

Zur Lösung dieses Dilemmas muss je nach Situation die Priorität umgesetzt werden. Dazu kann folgende Vorgehensweise vereinbart werden:

- Ein Task belegt wie bisher die Semaphore
- Erfolgt ein Zugriff auf die Semaphore von einem Task mit einer höheren Priorität, so wird der belegende Task auf die Priorität des Tasks mit der höheren Priorität gehoben.

Diese Vorgehensweise wird Prioritätsvererbung oder priority inheritance genannt.
Die Bearbeitung wird dann mit der neuen Priorität fortgeführt und die Semaphore wird freigegeben.
Der erste Prozess wird auf seine ursprüngliche Priorität zurückgesetzt und der wartende Task wird gestartet.

Im vorliegenden Beispiel würde damit Task 2 erst nach Freigabe der Semaphore von Task 3 gestartet (siehe Bild 8.25).

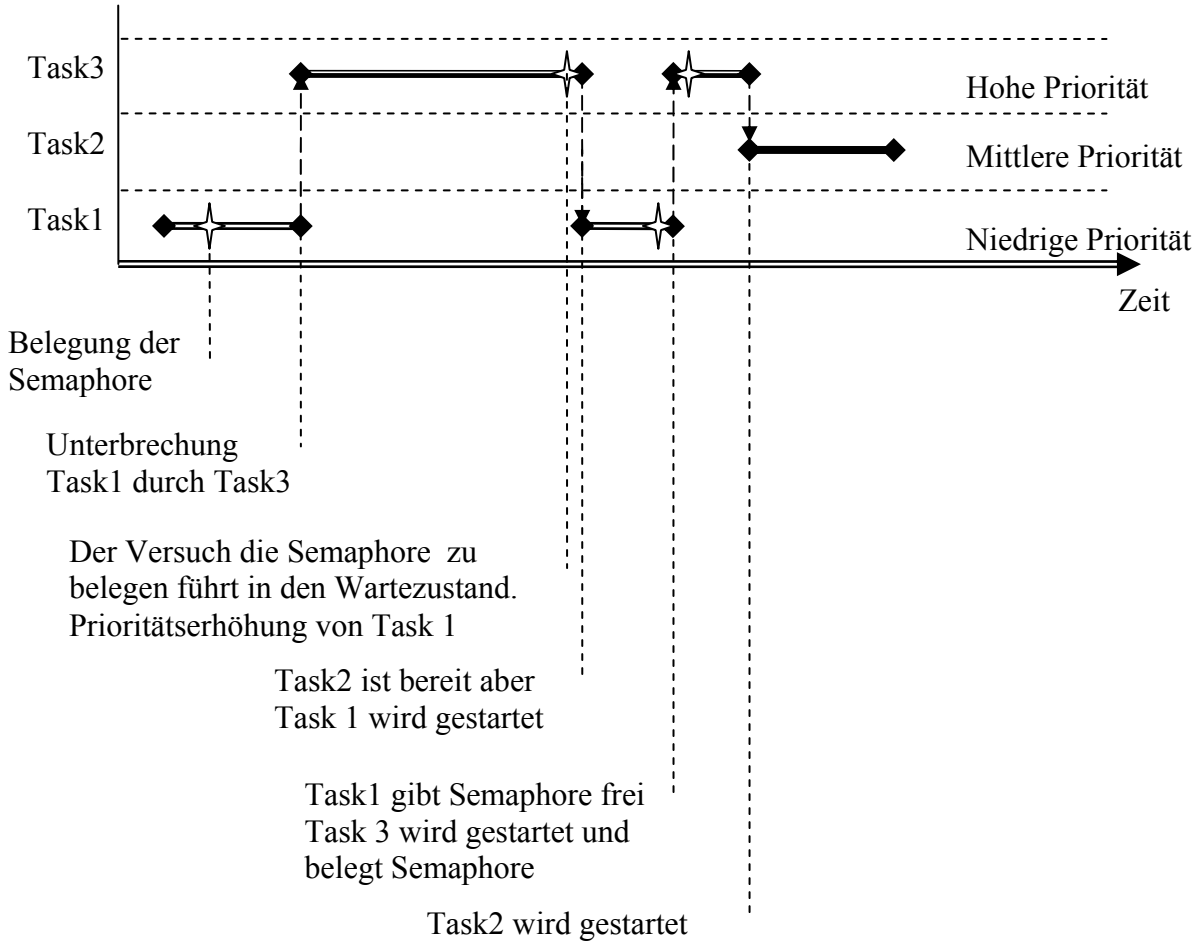


Bild 8.25 Ablauf bei der Verwendung von Semaphoren mit Prioritätsvererbung

Im vorliegenden Echtzeitbetriebssystem steht zur Zeit kein Mechanismus zur Prioritätsvererbung zur Verfügung. Es sollte an dieser Stelle aber auf die Problematik der Prioritätsinversion hingewiesen und Lösungsmöglichkeiten aufgezeigt werden.

9.3.5 HKRO Funktionen

In diesem Abschnitt sollen die für den Benutzer des Echtzeitbetriebssystems relevanten Funktionen noch mal tabellarisch aufgelistet und der Zweck stichwortartig genannt werden.

9.3.5.1 Tabelle der HKRO Funktionen

Funktion	Art	Zweck	Parameter	
Wartebedingungen				
RO_WaitDel	Makro	Ausführungsverzögerung	mRO_Index	Zählerindex
			mRO_DelInit	Wartezeit in Zyklen
RO_WaitInt	Makro	Warten auf Interrupt	mRO_Event	Interruptnummer
Nachrichten				
RO_SendMsg	Makro	Sende Nachricht	mRO_Event	Nachrichtennummer
RO_WaitMsg	Makro	Warten auf Nachricht	mRO_Event	Nachrichtennummer
Semaphore				
RO_RelSem	Makro	Semaphore freigeben	mRO_Event	Semaphorennummer
RO_GetSem	Makro	Wartezugriff auf Semaphore	mRO_Event	
RO_NWGetSem	Makro	Lesezugriff auf Semaphore	mRO_Event	
Initialisierung				
RO_StartTask	Makro	Setzt TCB Werte	mRO_TSK	Code Anfang
			mRO_oP	Priorität
			mRO_oID	Taskidentifizierungsnr
			mRO_oState	Startstatus
RO_BaseTask	Funktion	Startet den ersten Task	id	Taskidentifizierungsnr.
RO_TimerSetup	Funktion	Startet den Basis Timer		
Kritischer Bereich				
RO_SetCritical	Makro	Eintritt in kritischen Bereich		
RO_ExitCritical	Makro	Austritt aus krit. Bereich		

9.3.5.2 Beschreibung der HKRO Funktionen

Funktionsname: RO_WaitDel

Definition:
#define RO_WaitDel(mRO_Index,mRO_DelInit);

Funktion:
Die Funktion veranlasst den Task in einen Wartezustand zu gehen. Es wird ein Wartezähler über einen Index angewählt und mit dem gewünschten Wert initialisiert. Die Zeit berechnet sich aus Anzahl * Zeitquantum. Das Zeitquantum stellt die Zeit dar, die für die Zeitbasis des Echtzeitbetriebssystems eingestellt wurde. Dies ist hier die Wiederholrate von Timer 0.
Die Bestimmung dieser Werte erfolgt über RO_Tim0rel im C-Code und ROa_Tim0rel im Assembler Code.

Parameter:

mRO_Index	Nummer des Verzögerungszählers	Type :INT8U
mRO_DelInit	Anzahl der Zeitquanten	Type :INT8U

Globale Variable:

Aufrufbeispiel: RO_WaitDel(1,5);//Zählerindex 1, Verzögerung 5*Quantum

---

Funktionsname: RO_SendMsg

Definition:
#define RO_SendMsg(mRO_Event);

Funktion:
Versendet eine Nachricht an einen Task, der auf diese Nachricht wartet. Wartet kein Task auf diese Nachricht, so geht die Nachricht verloren.
Diese Tatsache kann in der Variable gRO_MSGAck abgefragt werden. Der Wert 1 zeigt an, dass mindestens ein Task aktiviert wurde. Sonst besitzt die Variable den Wert 0.

Parameter:

mRO_Event	Nummer der Nachricht, die gesendet wird.	Type :INT8U

Globale Variable:

gRO_MSGAck	Zustand, ob Nachricht angenommen wurde.	Type :INT8U

Aufrufbeispiel: RO_SendMsg(4);//Nachricht mit der Nummer 4 wird versendet.

Funktionsname: RO_WaitMsg

Definition: #define RO_SendMsg(mRO_Event);

Funktion:
Nach Aufruf dieser Funktion wird der Task in einen Wartezustand versetzt. Wird die Nachricht von einem anderen Task gesendet, so wird der Task aktiviert und führt seine Aufgabe weiter durch.

Parameter:
mRO_Event Nummer der Nachricht auf die gewartet wird. Type :INT8U

Globale Variable:
gRO_MSGAck Zustand, ob Nachricht angenommen wurde. Type :INT8U

Aufrufbeispiel: RO_WaitMsg(4);//Auf Nachricht mit der Nummer 4 wird gewartet.

Funktionsname: RO_RelSem

Definition:
#define RO_RelSem(mRO_SemN);

Funktion:
Die Semaphore mit der angegebenen Nummer wird freigegeben.
Alle Tasks die auf diese Semaphore warten, werden in den Zustand READY versetzt.

Parameter:
mRO_SemN Nummer, der freizugebenden Semaphore Type :INT8U

Globale Variable:

Aufrufbeispiel: RO_RelSem(5);//Semaphore Nummer 5 wird freigegeben.

Funktionsname: RO_GetSem

Definition:
#define RO_GetSem(mRO_SemN);

Funktion:
Es wird der Versuch unternommen die Semaphore mit der angegeben Nummer zu belegen. Ist die Semaphore belegt, so wird in einen Wartezustand gegangen. Der aufrufende Task bleibt solange in dem Wartezustand bis die Semaphore freigegeben wird.

Parameter:

mRO_SemN	Nummer der anzusprechenden Semaphore	Type :INT8U

Globale Variable:

Aufrufbeispiel: RO_GetSem(5);//Belegung der Semaphore Nummer 5

Funktionsname: RO_NWGetSem

Definition:
#define RO_GetNWSem(mRO_SemN,mRO_MSGAck);

Funktion:
Es wird versucht, die angegebene Semaphore zu belegen. War der Vorgang erfolgreich, so wird in der Variable mRO_MSGAck eine 0 zurückgegeben. Sonst ist der Wert >0.

Parameter:

mRO_SemN	Anzusprechende Semaphore	Type :INT8U
mRO_MSGAck	Rückgabewert	Type :INT8U

Globale Variable:

Aufrufbeispiel:

```
INT8U v1;
RO_GetNWSem(5,v1);      //Semaphore 5 wird angesprochen
                        //In v1 wird das Ergebnis zurückgegeben
```

Funktionsname: RO_StartTask

Definition:
#define RO_StartTask(mRO_TSK,mRO_oP,mRO_oID,mRO_oState)
Funktion:
Initialisiert den Task Control Block mit dem Programmanfangscode, der Priorität, der Identifikationsnummer, und dem Anfangszustand (e.g. RO_STREADY)

Parameter:

mRO_TSK	Code Pointer auf Taskanfang (Funktionsname)	
mRO_oP	Priorität des Tasks	Type :INT8U
mRO_oID	Identifikationsnummer	Type :INT8U
mRO_oState	Anfangsstatus	Type :INT8U
	RO_READY	
	RO_STSTOP	
	RO_WAITINT+nnnnn	
	RO_WAITDEL+nnnnn	
	RO_WAITSEM+nnnnn	

Globale Variable:

Aufrufbeispiel: RO_StartTask(Tsk1,1,1,RO_STREADY);

Funktionsname: RO_BaseTask

Definition:
void RO_BaseTask(INT8U id);

Funktion:
Startet den ersten Task zum Ablauf des Echtzeitbetriebssystems mit seiner Identifikationsnummer

Parameter:

id	Identifikationsnummer	Type :INT8U

Globale Variable:

Aufrufbeispiel: RO_BaseTask (1);//Start des Tasks mit der Identifikationsnummer 1

Funktionsname: RO_TimerSetup;

Definition:
void RO_TimerSetup();

Funktion:
Setzt die Werte im Steuerregister TMOD zum Betrieb des Timer 0 im 16 Bit Modus und lädt die Timerregister mit dem Wert, der unter RO_Tim0rel definiert wurde. Der Timer wird noch nicht gestartet.

Parameter:

Globale Variable:

Aufrufbeispiel: RO_TimerSetup();//Initialisieren des Timer 0

Funktionsname: RO_SetCritical();

Definition:
#define RO_SetCritical();

Funktion:
Setzt das Bit EAL zum Ein- und Ausschalten aller Interrupts auf 0 und sperrt damit alle Interrupts.

Parameter:

Globale Variable:

Aufrufbeispiel: RO_SetCritical(); // Sperren aller Interrupts

---

Funktionsname: RO_ExitCritical();

Definition:
#define RO_ExitCritical();

Funktion:
Setzt das Bit EAL zum Ein- und Ausschalten aller Interrupts auf 1 und hebt die globale Sperre der Interrupts auf.

Parameter:

Globale Variable:

Aufrufbeispiel: RO_ExitCritical();// Ermöglichen aller Interrupts

---

10 Literatur

[STR11] J. Ranggaard http://www.jbox.dk/sanos/source/lib/string.c.html 22.09.2011

[DB1] http://www.silabs.com/Support%20Documents/
TechnicalDocs/C8051F34x.pdf, 22.09.2011

[DB2] www.keil.com/dd/docs/datashts/silabs/c8051f130_short.pdf, 22.09.2011

[WAL96] Jürgen Walter
Mikrocomputertechnik mit der 8051-Controller-Familie
Springer Verlag, ISBN: 3-540-60540-1, 2. Auflage 1996

[FEG87] Otmar Feger
Die 8051 Mikrocontroller Familie
Mark & Technik Verlag AG, ISBN: 3-89090-360-6, 1987

[MAI96] Jürgen Maier-Wolf
8051 Mikrocontroller erfolgreich anwenden
Franzis Verlag, ISBN: 3-7723-6453-5, 2. Auflage 1996

[PON03] Michael J. Pont
Embedded C
Addison-Wesley Verlag, ISBN: 0-201-79523-X, 2003

[KEI05] C51 Compiler
Keil Software, 2005

[WIE04] Jörg Wiegelmann
Softwareentwicklung in C für Mikroprozessoren und Mikrocontroller
Hüthig Verlag, ISBN: 3-7785-2943-9, 3. Auflage 2004

[BAR99] Michael Barr
Programming Embedded Systems
O'Reilly Verlag, ISBN: 1-56592-353-5, 1999

[PRED99] Myke Predko
Programming and Customizing the 8051 Microcontroller
McGraw-Hill Verlag, ISBN: 0-07-134192-7, 1999

[LAB02] Jean J. Labrosse
MicroC/OS-II
CMP Books, ISBN: 1-57820-103-9, 2. Auflage 2002

[BLE94] K. Blecken, Mikroprozessortechnik,
Fachhochschule Heilbronn-Künzelsau, 1994

[GE1]	http://www.rechenhilfsmittel.de/rmhahn.jpg,	31.08.11
[GE2]	http://privat.swol.de/svenbandel/Hollertith2.jpg,	31.08.11
[GE3]	http://www.weller.to/his/img/zuse_z1.jpg,	31.08.11
[GE4]	http://www.at-mix.de/images/glossar/eniac1.jpg,	31.08.11
[GE5]	http://upload.wikimedia.org/wikipedia/commons/thumb/5/53/ Telefunken-tr4.jpg/220px-Telefunken-tr4.jpg,	31.08.11
[GE6]	http://static.nol.hu/media/picture/92/27/00/ 000002792-3500-330.jpg,	31.08.11
[GE7]	http://www.zdnet.co.uk/i/z5/illo/nw/story_graphics/10dec/ intels-victims/intelvics-tms-1000-texasinstruments.jpg,	31.08.11
[GE8]	http://upload.wikimedia.org/wikipedia/commons/2/2c/ KL_Intel_P8048H.jpg,	31.08.11
[GE9]	http://www.rcs.hu/roboshop/Microrobot/js8051a1cpu.htm,	31.08.11
[GE10]	http://cpucollection.ca/IntelN8096BH.jpg,	31.08.11
[GE11]	http://www.technikimbuero.at/Museum/Computer.htm,	31.08.11
[GE12]	http://www.efton.sk/t0t1/history8051.pdf	31.08.11
[ROB11]	http://www.robotelectronics.co.uk/forum/viewtopic.php?f=5&t=461,	22.09.11
[WI211]	http://en.wikipedia.org/wiki/I%C2%B2C,	22.09.11