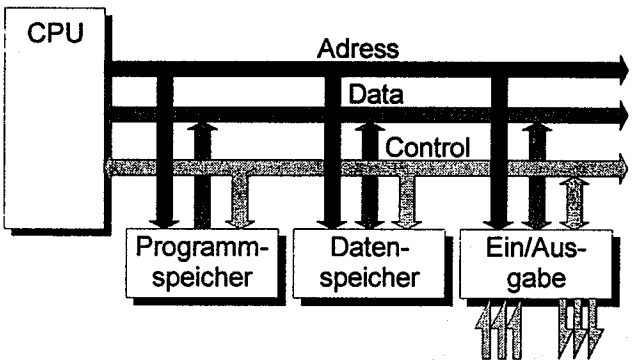
DER MIKROCOMPUTER

Universalität Der Mikrocomputer ist ein universelles System, das durch Programmierung für verschiedenste Anwendungsfälle eingesetzt werden kann!

GRUNDLEGENDER AUFBAU

Überblick

Blockschaltbild



- Kernstück des Mikrocomputers ist die CPU (Central Processing Unit) ➔ *Prozessor*
- Über die Adressen werden einzelne Speicherstellen angesprochen ➔ *Adressbus*
- Die Daten werden zwischen CPU und Speicher sowie Peripherie (Ein-/Ausgabe) transportiert ➔ *Datenbus*
- Für die Steuerung der einzelnen Baugruppen werden Kontrollsignale benötigt ➔ *Steuerbus*

Adressen Die Anzahl der verfügbaren Adressleitungen bestimmt die Grösse des physikalisch ansprechbaren Speichers. Die Adressen werden – im Normalfall – nur von der CPU generiert.

z.B. 20 Adressleitungen ➔ 2^{20} = 1'048'576 Byte = 1 MByte

32 Adressleitungen ➔ 2^{32} = 4'294'967'296 Byte = 4 GByte

Daten Die Anzahl der Datenleitungen ist ein Mass für die Leistungsfähigkeit des Prozessors.

Für kleine Systeme können 4 Bit vollauf genügen, während bei den heutigen Hochleistungsprozessoren 64 Bit als Standard betrachtet werden kann.

Steuersignale Neben der Adressierung der einzelnen Speicherstellen und dem Übermitteln der Daten müssen auch Statussignale zwischen Prozessor, Speicher und Peripherie ausgetauscht werden.

z.B. Memory-Read bzw. Memory-Write

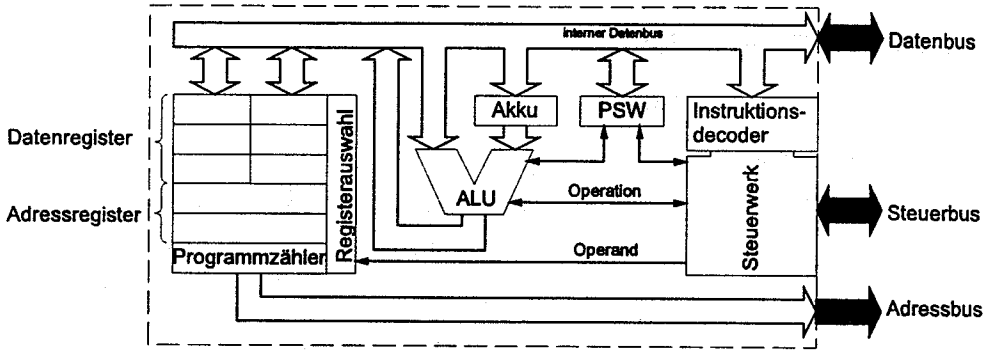
Interruptsignal, Interruptfreigabe usw.

Der Prozessor

Ursprünglich wurden Mikroprozessoren für Berechnungen von (digitalen) Zahlenwerten gebaut. So wurde der erste Prozessor - der 4004 von Intel - vor allem in Registrierkassen eingesetzt.

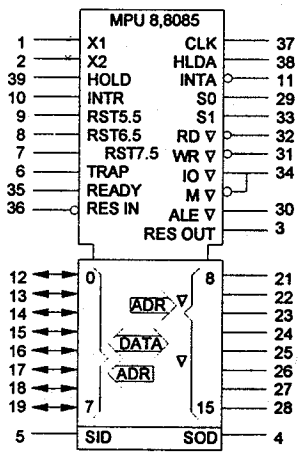
Innenleben

z.B. einfacher 8-Bit Prozessor



Symbol

z.B. Intel-8085



Klassifizierung

Prozessoren werden auf Grund von

- Busbreite
- Prozessortakt
- Befehlsumfang (RISC, CISC)
- Interne Register

klassifiziert, wobei grössere Busbreite und höhere Taktrate auch grössere Prozessorleistung bedeutet.

Die Leistungsfähigkeit wird meist in MIPS (Million Instruction per Second) bzw. FLOPS (Floatin Point Operation per Second) angegeben.

ALU

Sie dient der Verarbeitung der Daten. Wichtiger Bestandteil der ALU (Arithmetic Logic Unit) ist das Additionswerk. Es dient als Grundlage für alle mathematischen Berechnungen. Für die Subtraktion wird das **Zweierkomplement** verwendet.

Aufgabe :

Führen Sie folgende Binäroperationen für die Werte A und B durch

```
A   0 1 0 1 1 0 1 1      A   1 0 0 1 0 0 1 0
B + 1 0 0 0 1 0 0 1      B + 1 1 1 0 0 1 1 1
    -----1-1-1-1--           ---------1-1----
Res 1 1 1 0 0 1 0 0      Res 1 0 1 1 1 0 0 1
    ===============          ===============

A   0 0 0 1 1 0 1 0      A   1 0 0 1 1 1 0 0
B - 0 0 0 1 0 1 1 1      B - 1 1 0 1 1 0 0 1
    ---------------          ---------------
Res                      Res
    ===============          ===============
```

Weiter werden Logikoperationen wie AND-Verknüpfung aber auch das Schieben von Bitstellen durch die ALU besorgt.

Die Ergebnisse einer Operation werden entweder im Akku zwischengespeichert oder an ein Register überwiesen. Je nach Status der ausgeführten Operation werden bestimmte Bit im Statusregister (PSW = Programm Status Wort) gespeichert wie z.B. negative Zahl, Überlauf usw.

Register

Ein Prozessor muss die bearbeiteten Daten intern speichern können, dazu dienen die Register. Je nach Umfang der Prozessorleistung sind mehr oder weniger dieser Register verfügbar, wobei moderne Typen sogenannt „universelle Register" aufweisen.

Das wichtigste Register wird meist als **Akku** bezeichnet. Es ist das eigentliche Arbeitsregister und wird für alle auszuführenden Befehle benutzt.

Das Arbeiten mit Prozessor-Registern bringt vor allem Geschwindigkeitsvorteile, da diese Daten für die Bearbeitung sofort verfügbar sind. Im Vergleich dazu ist der Zugriff auf den externen Speicher um vieles langsamer

z.B. werden in den aktuellen Prozessoren die Daten intern mit bis zu 400 MHz verarbeitet, während der externe Speicher bestenfalls mit 100 MHz getaktet wird.

Statusregister

Das Programm-Status-Wort oder Flag-Register signalisiert über einzelne Bits Bearbeitungszustände sowie spezielle Werte. Je nach Komplexität verfügt ein Prozessor über unterschiedlich viele solcher Bits. Die wichtigsten Flags sind:

- **Carry**[1]
  Diese Bit wird gesetzt, wenn bei einer arithmetischen Operation ein Überlauf eingetreten ist.
- **Auxiliary-Carry**[2]
  Mit diesem Bit wird bei Berechnungen mit BCD-Zahlen angezeigt, ob ein Überlauf innerhalb der unteren BCD-Stelle stattgefudnen hat.
  In einem Byte (8 Bit) finden zwei BCD-Zahlen Platz!
- **Sign**[3]
  Hiermit wird das Vorzeichen nach einer logischen oder arithmetischen Operation angezeigt.
- **Zero**[4]
  Das Zero-Flag zeigt an, dass das Ergebnis einer vorausgegangenen Opartion gleich Null ist. Ein Ergebnis, dessen Wert im Akku glcih Null ist, kann einen Übertrag ausgelöst haben. Damit wird auch das Carry-Flag gesetzt.
- **Parity**[5]
  Hiermit wird angezeigt, dass die Anzahl der Bits mit dem Wert 1 ein geradzahliger Wert ist.

Programmzähler

Das Programm eines Mikrocomputers wird als eine Abfolge von einzelnen Befehlen dargestellt und auch dementsprechend abgearbeitet. Der Programmzähler erhöht nach der Ausführung eines Befehls seinen Wert um eins und weist somit auf den nächsten Befehlseintrag.

Soll auf Grund eines internen Zustandes (Statusregister) die Programmausführung an einer andern Stelle fortgeführt werden, wird der Programmzähler entsprechend neu geladen.

Adressregister

Neben den Adressen für das Programm werden auch Adressen für den Zugriff auf die Daten im RAM-Speicher sowie die Peripherie benötigt. Dazu verfügt der Prozessor über mehrere spezielle Register.

Je nach Bearbeitungszustand wird am Adressbus der Inhalt des Programmzählers oder aber eines Adressregisters ausgegeben.

---

[1] Übertrag
[2] 4-Bit Übertrag
[3] Vorzeichen
[4] Inhalt des Akku ist 0
[5] Parität

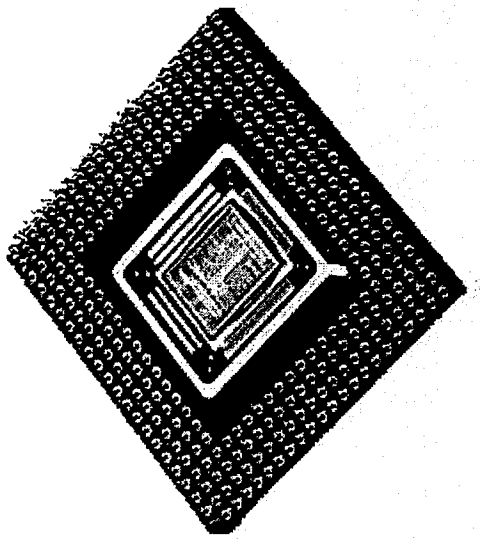
Dekoder

Die gelesenen Befehle werden durch einen Dekoder zerlegt und alle notwendigen internen Aktionen aktiviert. Der Dekoder selber kann als „elektronische Maschine" betrachtet werden, die in mehreren Schritten den anstehenden Befehl ausführt.

Dabei wird der Sprachumfang (die Befehle des Prozessors) als sogenannter Mikrocode im Dekoder einprogrammiert.

Steuerwerk

Hier werden die Steuersignale wie z.B. Memory-Read usw. generiert bzw. externe Signale wie z.B. Interrupts erkannt und dem Dekoder gemeldet.

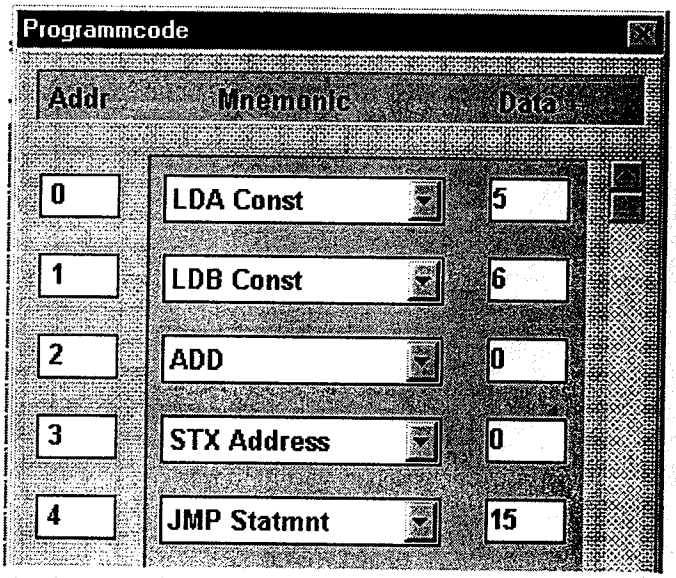


Übung

1. *Verfolgen Sie mit dem „Micro" ein einfaches Programm, welches zwei konstante 4-Bit Werte addiert.*

   Halten Sie alle Ihre Feststellungen auf Arbeitsblatt 1.1.7 fest!

   - starten Sie das Programm Micro aus der Programmgruppe "Fachkunde INFORMATIK/GRUNDLAGEN"
   - laden Sie über "Datei/Programm öffnen"... das Programm "INIT.PRG"; alle Befehle sollten auf NOP lauten
   - schreiben Sie über "Bearbeiten/Programm" folgendes Programm



   - starten und reseten Sie das Programm über das Panel und verfolgen Sie die Werte im Akku sowie die Flags des PSW.

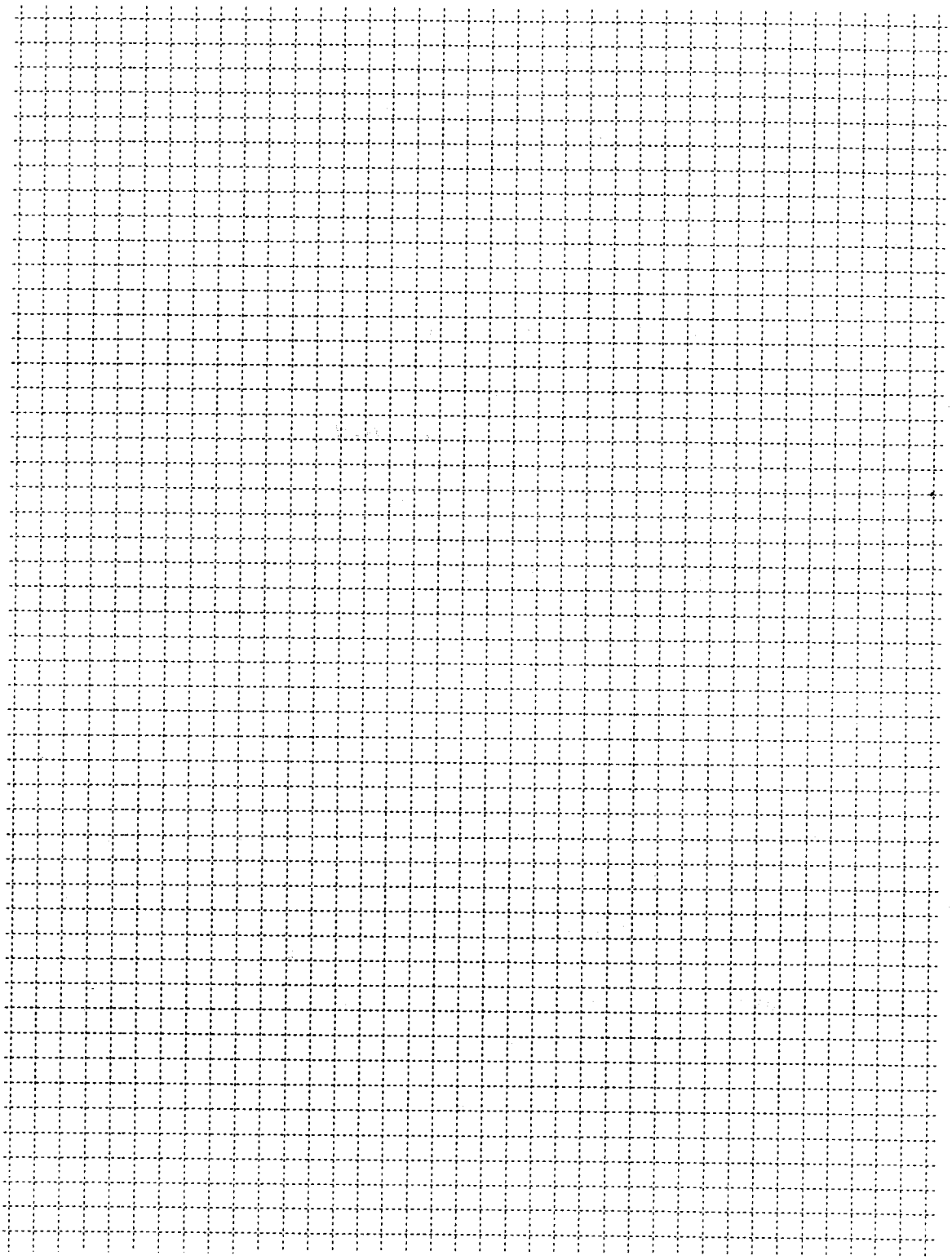
   - ändern Sie den Data-Wert in Zeile 0 auf 9 und verfolgen Sie nun das Programm
   - ändern Sie den Data-Wert in Zeile 0 auf 10 und verfolgen Sie nun das Programm
   - ändern Sie den Data-Wert in Zeile 0 auf 11 und verfolgen Sie nun das Programm

2. *Verfolgen Sie mit dem „Micro" ein einfaches Programm, welches zwei konstante 4-Bit Werte subtrahiert.*

   Halten Sie alle Ihre Feststellungen auf einem Arbeitsblatt 1.1.7 fest!

   - reseten Sie das vorherige Programm
   - ändern Sie den Befehl in Zeile 2 auf SUB und verfolgen Sie nun das Programm
   - ändern Sie den Data-Wert in Zeile 1 auf 10 und verfolgen Sie nun das Programm
   - ändern Sie den Data-Wert in Zeile 1 auf 11 und verfolgen Sie nun das Programm
   - ändern Sie den Data-Wert in Zeile 1 auf 12 und verfolgen Sie nun das Programm

Auswertung



Prozessortypen

Prozessoren sind „universell" einsetzbare Bausteine der Elektronik. Trotzdem werden für die unterschiedlichsten Bedürfnisse auch spezielle Prozessortypen gebaut.

Mikroprozessor Die „eigentliche" Rechenmaschine[6], welche mittels einfacher mathematischer und logischer Befehle Programme abarbeiten kann.

RISC = Reduced Instruction Set Computer: Weist wenige grundlegende Befehle auf, die dafür sehr schnell ausgeführt werden.

~~z.B. Intel-Prozessoren (Pentium II)~~ DEC Alpha

CISC = Complex Instruction Set Computer: Weist sehr viele – auch sehr spezielle – Befehle auf. Mit einem Befehl können sehr komplexe Arbeitsschritte ausgeführt werden.

~~z.B. DEC Alpha~~ PII Intel

Die meisten der modernen Heimcomputer verfügen über einen ganz „normalen" Prozessor.

z.B. IBM-PC, Power-MAC

Mikrokontroller Auch als Single-Chip bezeichnet, da sich in einem Baustein nebst dem Prozessor-Kern auch Ein-/Ausgabe-Schaltungen, Speicher usw. befindet.

Je nach Aufgabenstellung kann ein Single-Chip sehr unterschiedliche Baugruppen beinhalten, so z.B. A/D-Wandler für Messsysteme, Kommunikationssysteme für Netzkarten, Watch-Dog für ausfallsichere Systeme usw.

z.B. Automobile, Waschmaschine usw.

Co-Prozessor Auch als NDP (Numeric Data Processor) bezeichnet. Entlastet den Hauptprozessor bei der Durchführung rechenintensiver Programme. Der NDP erweitert den Befehlssatz des Prozessors.

z.B. CAD, Graphikprogramme usw.

Signalprozessor Dient der zeitkritischen Verarbeitung von Signalen.

z.B. Funktechnik, Telekommunikation usw.

Graphikproz. Entlastet den Hauptprozessor von der Berechnung der Bildschirmausgaben.

z.B. Graphische Benutzerschnittstellen wie Windows usw.

[6] general purpose processor

SPEICHER-BAUSTEINE

Speicherstellen

Adressen

Jede Speicherstelle wird über eine eindeutige Adresse angesprochen, wobei ein einzelner Baustein mehrere solcher Stellen enthält. Pro Stelle können mehrere Speicherzellen selektiert werden, wenn z.B. die „Datenbreite" 8 Bit beträgt.

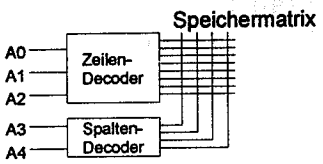
Die Anzahl der Adressleitungen ist mit ein Mass für die Grösse des Speichers.

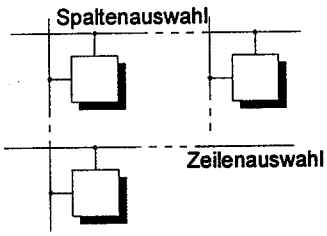
z.B. 12 Adressen ➔ 2^{12} = 4096 Stellen.

Wenn pro Stelle 8 Bit (= ein Byte) aktiviert sind, so spricht man in diesem Fall von einem 4kByte Speicher.

Zeilen/Spalten

Die einzelnen Stellen werden in einer Matrix-Form angeordnet und über Spalten und Zeilen ausgewählt. Die angelegt Adresse muss dazu in einem Decoder „zerlegt" werden.





z.B. werden die fünf Adressleitungen (A0-A4) in 8 Zeilen und 4 Spalten zerlegt, so dass die 32 Stellen (2^5 = 32) adressiert werden können.

Festwertspeicher

Festwertspeicher werden für Programmcode und konstante Werte benötigt, da deren Inhalt auch im stromlosen Zustand erhalten bleiben. Nur so ist es möglich, dass ein Computersystem starten (booten) kann.

Ein weiterer Einsatz stellt das Speichern von Systemdaten (z.B. BIOS-Einstellungen beim PC) dar.

ROM/PROM

Die Daten sind absolut irreversibel (unveränderbar) im Speicher eingeschrieben.

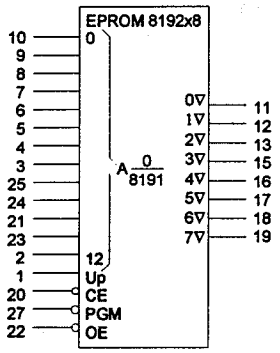
Bei einem ROM[1] werden die Daten (das Programm) bereits bei der Produktion durch den Hersteller eingebrannt, während bei einem PROM[2] die Daten mittels eines Programmiergerätes durch den Anwender eingebrannt werden. Der Begriff „gebrannt" ist vollkommen richtig, da beim Programmieren Verbindungen im Baustein durch einen Stromimpuls erhitzt und zerstört werden.

[1] Read Only Memory
[2] Programmable ROM

EPROM

Um den Ausschuss (Abfall) zu verringern, wurden (durch UV-Licht) löschbare PROM[3] entwickelt.

EPROM sind nach wie vor das wichtigste Medium, um Programme in einem Mikrocomputer-System verfügbar zu machen.



EPROM weisen den Nachteil auf, dass für den Löschvorgang der Baustein aus der Schaltung ausgebaut und in ein Löschgerät gelegt werden muss.

EEPROM

Dies ist bei den elektrisch löschbaren PROM[4] nicht mehr nötig, da auch während dem Betrieb durch das Anlegen einer Spannung einzelne Stellen gelöscht und beschrieben werden können.

EEPROM werden vorwiegend für das Speichern von System- und Konfigurationsdaten (z.B. BIOS-Einstellungen) verwendet.

Flash ROM

Im Gegensatz zum EEPROM können bei Flash ROM's nur ganze Speicherbereiche bzw. der ganze Baustein gelöscht werden.

Flash ROM werden oft für das Speichern von Programmcode – der aber dennoch anpassbar sein muss – verwendet.

Schreib-Lese-Speicher

Die Schreib-Lese-Speicher (=RAM[5]) werden für das (temporäre) abspeichern von Daten verwendet. Der Inhalt der Speicherzellen ist nur solange verfügbar, als auch eine genügend grosse Versorgungsspannung am Baustein anliegt. Nach dem Einschalten eines Mikrocomputer-Systems ist der Inhalt des RAM-Speichers undefiniert!

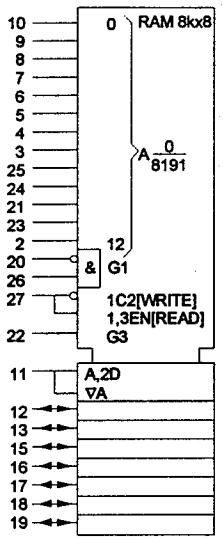
Beachte

Bei Disk basierten OS werden die Programme von der Disk ins RAM umkopiert, so dass hier das RAM als Programmspeicher verwendet wird.

[3] Erasable PROM
[4] Electical Erasable PROM
[5] Random Access Memory

SRAM

Statische RAM-Zellen behalten den Inhalt, d.h. dass die einmal eingeschriebene Information (0 bzw. 1) bis zum Überschreiben durch eine andere Information oder das Abschalten der Versorgungsspannung erhalten bleibt.



DRAM

Dynamische RAM-Zellen speichern die Information (0 bzw. 1) in einem Kondensator, weshalb der Inhalt beim Auslesen verloren geht. Dies erfordert einen „Refresh" der zyklisch ausgeführt werden muss.

DRAM vs. SRAM

Bei DRAM ist der Aufbau der einzelnen Zelle mit weniger Elementen möglich als bei SRAM, weshalb auf der gleichen Grundfläche mehr Speicherzellen untergebrachte werden können, was tiefere Kosten bedeutet. Dafür ist aber die Zugriffsgeschwindigkeit klar länger als bei SRAM.

DRAM eignen sich daher vor allem als Arbeitsspeicher, während SRAM für den schnellen Cache-Speicher verwendet wird.

DP-RAM

In ganz speziellen Fällen ist es notwendig, dass von zwei oder mehr Prozessoren aus auf den gleichen Speicher zugegriffen werden kann.

Erfolgt dieser Zugriff über ein gemeinsames Bus-System muss er sequentiell erfolgen. Um diese „Schwäche" zu umgehen werden DP-RAM[6] verwendet. Hier können zwei Prozessoren gleichzeitig auf den Baustein zugreifen, was die Leistungsfähigkeit eines Computersystem erheblich erhöhen kann.

Z.B. sind moderne Graphik-Systeme mit DP-RAM bestückt, so dass der Hauptprozessor und der Graphikprozessor mit voller Geschwindigkeit zugreifen können.

[6] Dual Port RAM

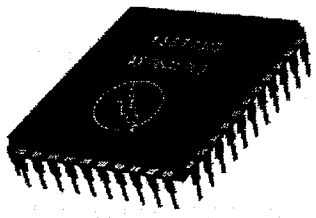
NVRAM[7]

Hier handelt es sich um RAM-Speicher welcher mit einem EEPROM hinterlegt ist. Über eine spezielle Signalleitung kann der Inhalt das RAM ins EEPROM kopiert bzw. vom EEPROM gelesen werden.

Gegenüber einem EEPROM bzw. Flash-ROM verhält sich das NVRAM während des Betriebs wie ein normales RAM, das gelesen und beschrieben werden kann. Bei einem Spannungsausfall wird mit Hilfe von elektronischen Schaltungen der Kopiervorgang aktiviert, so dass die Daten unverlierbar gespeichert sind und bei der Wiederaufnahme des Betriebs der ursprüngliche Zustand hergestellt werden kann.

FIFO

Ein ganz spezieller Speicher stellt der FIFO[8] dar. Hier werden die zu speichernden Daten in eine „Röhre" eingeschoben und am andern Ende wieder ausgelesen, d.h. dass die erst eingeschriebenen Daten auch als Erste wieder ausgelesen werden.



[7] Non Volatile RAM
[8] First In First Out

EIN-/AUSGABE-BAUSTEINE

Allgemeines

EVA

Ein Mikrocomputer ist nur dann sinnvoll einzusetzen, wenn er mit seiner Umgebung Informationen in Form von digitalen Signalen austauschen kann, um diese zu bearbeiten.

Die entsprechenden Bausteine weisen oftmals ein recht grosse Komplexität auf und können teilweise auch bestimmte Aufgaben selbständig lösen.

Adressen

Die Peripherie-Bausteine werden über eine kleine Anzahl von Adressen angesprochen, um die verschiedenen Register zu lesen und zu beschreiben.

Register

Meistens ist ein Status-Register verfügbar. Damit können bestimmte Betriebsmodi eingestellt bzw. Zustände abgefragt werden.

Daneben sind die eigentlichen Daten-Register verfügbar. Hier werden die Daten für die Ausgabe eingeschrieben bzw. als Eingabe eingelesen.

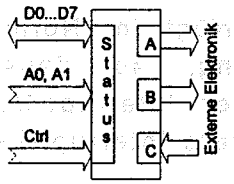
I/O-Port

Port

Die über eine gemeinsame Adresse angesteuerten Ein-/Ausagbe-Leitungen werden als Port bezeichnet.

Parallel I/O

Hier werden die Daten „Bit-Parallel" ausgegeben bzw. eingelesen. Der Baustein-Typ 8255 kann als der „klassische" Parallel-Port bezeichnet werden. Er verfügt über 3 Port's zu je 8 Bit breite. Mittels des Status-Registers kann festgelegt werden, welche Port's als Eingang und welche als Ausgang verwendet werden.

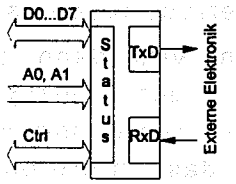


Die Adressierung des Status-Registers sowie der einzelnen Port's erfolgt über zwei Adress-Leitungen ($2^2 = 4$).

Durch Schreiben in den Baustein – auf das entsprechende Port – sind die Daten am Ausgang sofort verfügbar. Durch Lesen aus dem Baustein werden die am Port anliegenden Daten übernommen.

Serielle I/O

Hier werden die Daten „Bit-Seriell" ausgegeben bzw. eingelesen, wobei der Baustein – als UART[1] bezeichnet – die Umwandlung von seriell in parallel (für den Mikroprozessor) selber vornimmt. Der Baustein-Typ 8250 stellt hier den klassischen Seriell-Port dar, der in einer modernisierten Version als 16550 in den meisten PC's Verwendung findet.



[1] Universal Asynchronous Receiver and Transmitter

Wie beim Parallel-Port kann hier auch über das Status-Register der Betriebsmodi – in diesem Fall z.B. Baudrate, Parität usw. – eingestellt werden.

Durch Schreiben in den Sendebuffer wird ein Byte für die Sendung vorbereitet, während durch Lesen des Empfangsbuffers ein eingeschobenes Byte ausgelesen wird.

Interrupt

Da die serielle Leitung mit einer wesentlich tieferen Geschwindigkeit arbeitet als der System-Bus des Mikrocomputers, entstehen Wartezeiten, in denen die Buffer nicht bearbeitet werden dürfen! Damit das System nicht unnötig warten muss, in dem es den Zustand des Bausteins prüft (Busy Wait), wird bei abgeschlossener Operation – ein Zeichen gesendet bzw. empfangen – ein Interrupt ausgelöst.

System

Zeitgeber

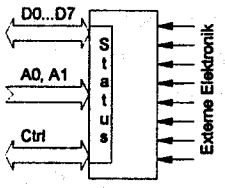
Die meisten Mikrocomputer-Systeme benötigen Zeitgeber bzw. Zähler. So lässt sich z.B. ein preemtives Multitasking[2] nur bewerkstelligen, wenn das OS zyklisch durch einen Timer-Interrupt die Kontrolle erhält.

Weiter können Zeitgeber und Zähler auch im Zusammenhang mit der Ein-/Ausgabe wichtige Funktionen übernehmen.

Die Betriebsart sowie auch Zyklusdauer usw. werden im Status-Register eingetragen.

Interrupt

Die Unterbrechungsanforderung ist ein sehr wichtiges „Werkzeug" für die Programmierung. Nur damit lässt sich auf interne und externe Ereignisse reagieren, die zu einem unbekannten Zeitpunkt auftreten.



Eine CPU verfügt im Normalfall nur über ein Interrupt-Leitung, so dass ein zusätzlicher Baustein – der Interrupt-Controller – benötigt wird, um mehrere Interrupt-Leitungen zusammenzufassen.

Ablauf

Sobald ein externes Ereignis auftritt, wird dieses im Baustein gespeichert und eine Interrupt-Meldung an den Prozessor gesendet. Auf diese Unterbrechung hin wird der Prozesor den Inhalt das Interrupt-Controllers auslesen, damit er feststellen kann, welcher Eingang den Interrupt ausgelöst hat.

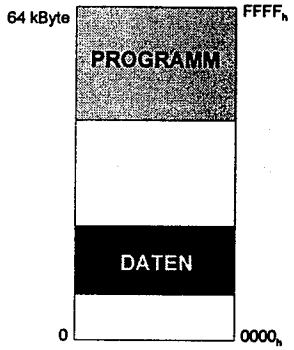
Auf Grund dieser Information wird dann der Prozessor zum entsprechenden Programmteil – der Interrupt-Service-Routine – verzweigen und die nötigen Anweisungen ausführen. Nach Abschluss dieser Arbeit wird das unterbrochene Programm an seiner ursprünglichen Stelle weiter ausgeführt.

[2] quasi parallele Abarbeitung mehrerer Programme

ADRESSIERUNG

Speicheranordnung

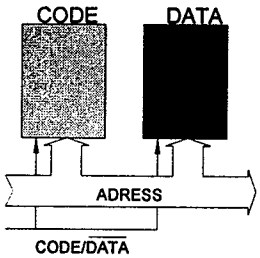
Der von einem Prozessor verwaltete Speicher lässt sich für uns nicht einsehen, so dass als Hilfsmittel eine „Speicherkarte" verwendet wird, an der sich gewisse Vorgänge und Zustände aufzeigen lassen.



Die oben gezeigte Darstellung ist eine allgemeingültige Art, für die Speicherkarte (Memory-Map).

Harvard

Eine Speicher-Architektur die zwei getrennte Speicherbereiche für Daten und Code vorsieht.



Über eine Steuerleitung lassen sich die Adressen entweder dem Daten- oder aber dem Codespeicher zuweisen.

Mit einer begrenzten Anzahl von Adressen lässt sich damit ein doppelt so grosser Speicherraum adressieren. D.h. dass mit 16 Adressen (2^{16} = 65536 ➔ 64 kByte) 128 kByte Speicher ansprechbar sind. Zu beachten ist aber, dass die Speicherbereiche nicht gemischt werden können.

Von Neumann

Bei dieser Architektur sind Code und Daten in ein und demselben Speicherbereich untergebracht. Dies ergibt eine grössere Flexibilität, auch wenn der Speicher insgesamt nur halb so gross ist.

Problematisch kann sich hier aber das Überschreiben von Code auswirken, wenn sich dieser – wie für Disk OS üblich – im RAM befindet!

Die meisten der modernen Mikrocomputer-Systeme verwenden diese Struktur.

Decodierung

Der Speicher setzt sich meist aus mehreren einzelnen Bausteinen (EPROM, RAM, Peripherie usw.) zusammen. Deshalb muss die vom Prozessor generierte Adresse aufgeteilt (dekodiert) werden.

Beispiel :

Für eine Applikation mit dem 8051-Microkontroller werden 28k Byte RAM-Speicher benötigt. Da es sich um ein Massenprodukt handelt, soll der Speicher optimal ausgenutzt werden, d.h. es darf kein 32k × 8 Baustein verwendet werden.

Der Entwickler entschliesst sich mit folgender Konfiguration zu arbeiten: Je einmal 16k × 8, 8k × 8 und 4k × 8.

Aus der Graphik wird ersichtlich, dass die höchsten Bit's (Bit Nr. 15 - 12) für die Selektion des jeweiligen Bausteins massgebend sind, während die tieferen Bit's (alle mit x bezeichneten Stellen) die Adressen des Bausteins ansprechen.

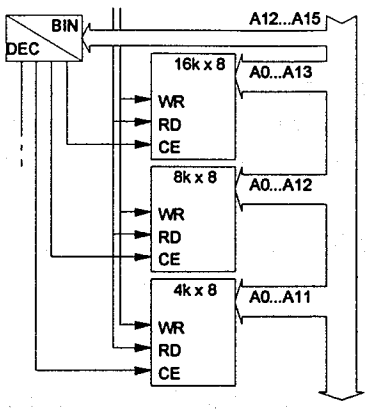
WHT

Dieser Sachverhalt lässt sich auch mit einer Wahrheitstabelle gut wiedergeben:

Adressen							
A15	A14	A13	A12	A11...A0	RAM	Adressen	Anz. Zellen
0	0	X	X	x...x	16kx8	0...16383	2^{14} = 16384
0	1	0	X	x...x	8kx8	16384...24575	2^{13} = 8192
0	1	1	0	x...x	4kx8	24576...28671	2^{12} = 4096
0	1	1	1	x...x		28671...32767	
1	x	x	x	x...x		32768...65535	

(x steht für beliebigen Wert, in diesem Fall 0 bzw. 1)

Blockschaltbild



Der Dekoder wandelt den Binärcode in einen Dezimalcode um.

Beispiel : WHT eines Binär-Dezimal-Dekoders

Eingang			Ausgang							
E2	E1	E0	A7	A6	A5	A4	A3	A2	A1	A0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

Selektierung Somit lässt sich über den CE-Eingang[1] der jeweils gewünschte Baustein auswählen, d.h. dass die tieferen Adressen ab A0-A11 nur für den – auf Grund der höheren Adress A12-A15 selektierten Baustein – gültig sind.

Steuerung Über die Steuerleitungen WR und RD wird dem RAM-Baustein mitgeteilt, ob ein Lese- bzw. Schreibzugriff erfolgt. Die entsprechende Leitung wird auf Grund des im Prozessor dekodierten Befehls durch die CPU aktiviert.

[1] CE = Chip Enable

Mapping

Neben dem Arbeitsspeicher für den Programmcode und die Daten muss auch die Peripherie über Adressen angesprochen werden. Auch hier sind zwei grundlegend verschiedene Methoden bekannt.

I/O-Mapped Wie bei der Harvard-Architektur sind für den Speicher und die I/O verschieden Adressräume vorgesehen. Über eine Steuerleitung wird von der CPU aus mitgeteilt, ob Speicher oder I/O adressiert wird.

Memory-Mapped Hier teilen sich Speicher und I/O den gleichen Adressraum. Somit erfolgt der Zugriff auf die Peripherie über die gleichen Befehle wie beim Zugriff auf den Speicher.

FUNKTIONSABLAUF

Befehl und Daten

Befehlsfolge

Ein Computerprogramm besteht aus einer Liste von Anweisungen, die nacheinander abgearbeitet werden. Die meisten Befehle beziehen sich dabei auf Daten, die im RAM oder einem internen Register der CPU gespeichert sind.

Befehle

Je nach Befehl werden ein bis mehrere Byte's im Programmspeicher benötigt. Dabei hängt die Länge des Befehls mitunter auch von der Komplexität ab.

Beispiel :

Für den 80286 können die Befehl zwischen einem und vier Byte umfassen.

Befehl	Byte	Bedeutung
`PUSH DX`	51_h	Schreibt DX-Register auf den Stack
`JMP START`	$EBF3_h$	Springt um 13 Byte zurück im Programm
`MOV AX,Var`	$A10001_h$	Schreibt `Var` (auf Adresse `100h`) ins AX-Register der CPU
`MOV Var [BX][SI],CX`	9880002_h	Lädt `Var` mit CX-Register der CPU, wobei die Adress von `Var` über einen Index (BX-Register) festgelgt wird

Operationscode

Das erste Byte des Befehls stellt den Operationscode der CPU dar, während die folgenden Bytes sich auf die Speicherstellen beziehen.

Bei einer Grösse von einem Byte für den Operationscode können 256 verschiedene Befehle verfügbar gemacht werden. Bei heutigen CISC-Prozessoren wie dem Pentium II umfasst der OP-Code daher 2 Byte, um die Vielzahl der unterschiedlichen Befehle repräsentieren zu können.

Arbeitsschritt

Jeder Arbeitsschritt besteht aus

• Lesen des Befehls	**Fetch**	Die Adresse des Programmspeichers wird ausgegeben und der Befehl über den Datenbus eingelesen.
• Dekodieren des Befehls	**Execute**	Erfolgt intern in der CPU, so dass Adress- und Datenbus nicht belegt sind.
• Lesen oder schreiben der Daten im RAM		Die Adresse des Datenspeichers wird ausgegeben und die Daten transportiert.

Die Zyklen eines Prozessors

Wie bereits erwähnt, besteht die Abarbeitung eines Befehls aus mehreren Arbeitsschritten.

Taktzyklus

Die CPU ist ein dynamisches System, das step-by-step die Befehle ausführt. Als Grundlage dient ein Taktsignal, welches das interne Zeitverhalten (Timing) steuert.

Der Takt wird (fälschlicherweise) oft als Mass für die Leistungsfähigkeit einer CPU angegeben. Im Endeffekt lässt sich aber aus einer Angaben wie z.B. 400 MHz nichts über die tatsächliche Leistung des Prozessors aussagen.

Maschinenzyk.

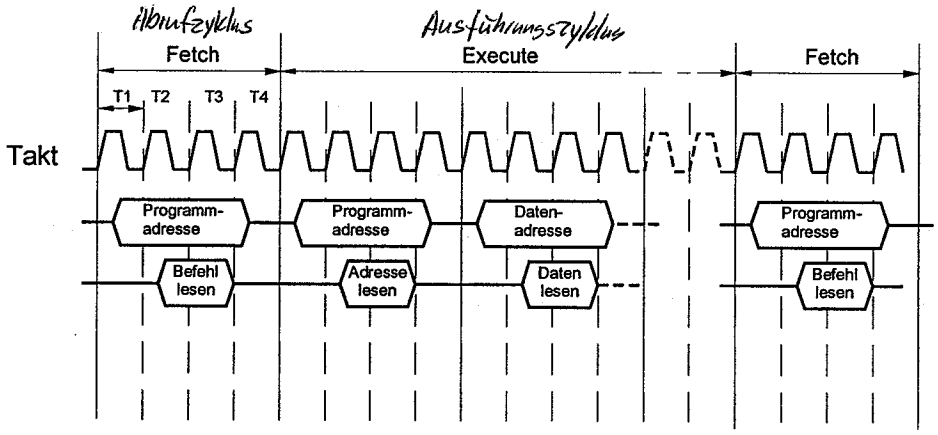
Der Maschinenzyklus beschreibt eine vollständige Aktion wie z.B. „Speicher lesen" innerhalb der Befehlsausführung.

Ein solcher Zyklus wiederum erfordert mehrere Schritte innerhalb der CPU, d.h. er dauert meistens zwischen drei und sechs Taktzyklen.

Befehlszyklus

Der Befehlszyklus beschreibt die Zeitdauer, die das Ausführen des Befehls benötigt. Je komplexer der Befehl ist um so länger dauert die Ausführung.

Jeder Befehlszyklus erfordert mehrere Maschinenzyklen, wobei zuerst immer der Abrufzyklus[1] kommt. Er besteht (normalerweise) aus einem Maschinenzyklus, bei dem der Operationscode vom Programmspeicher eingelesen wird. Danach folgt der Ausführungszyklus[2]. Er umfasst je nach Komplexität des Befehls mehrere Maschinenzyklen.



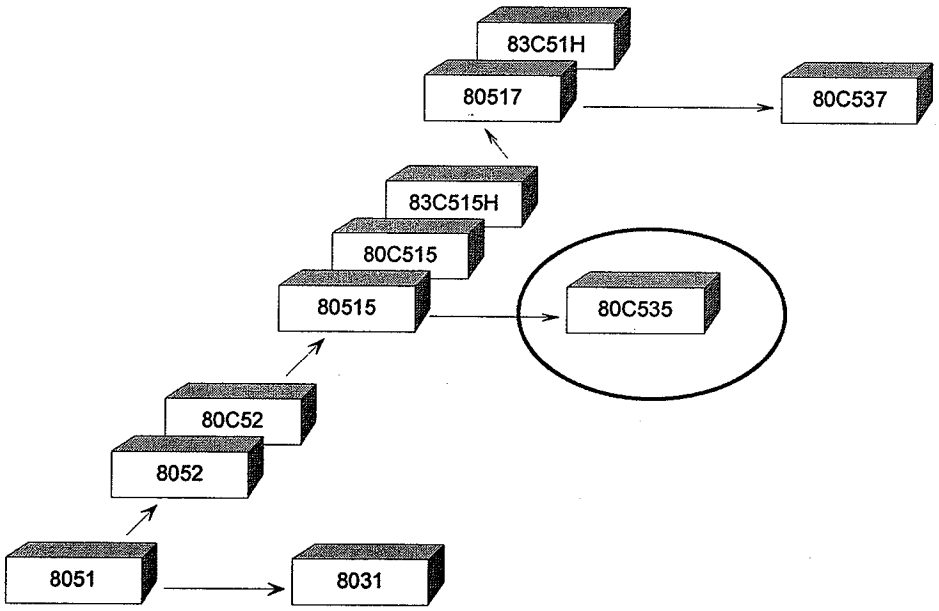
[1] Fetch
[2] Execute

DER 80535 MIKROKONTROLLER

STAMMBAUM

Die 8051-Familie wurde von Intel entwickelt. Ein günstiger Preis und einfache Lizenzierungen führten dazu, dass viele andere Prozessorhersteller wie z.B. Siemens, Dallas Semiconductors usw. diesen Typ erweiterten. Es sind heute Derivate für die verschiedensten Anwendungsgebiete verfügbar, wie z.B. spezialisierte Typen für Netzwerkkarten (mit integriertem CSMA-CD) die Automobilindustrie (mit CAN-Bus-Interface) usw.

Der 8051 besitzt intern bereits RAM (128 Byte) und ROM (4kByte), Timer, Interrupt-Controller sowie auch eine serielle und mehrere parallele Schnittstellen. Als 8031 ist er (billiger) auch als Version ohne ROM erhältlich. Dafür müssen dann aber 2 Ports für den Adress- und Datenbus „geopfert" werden.



ROM-Less

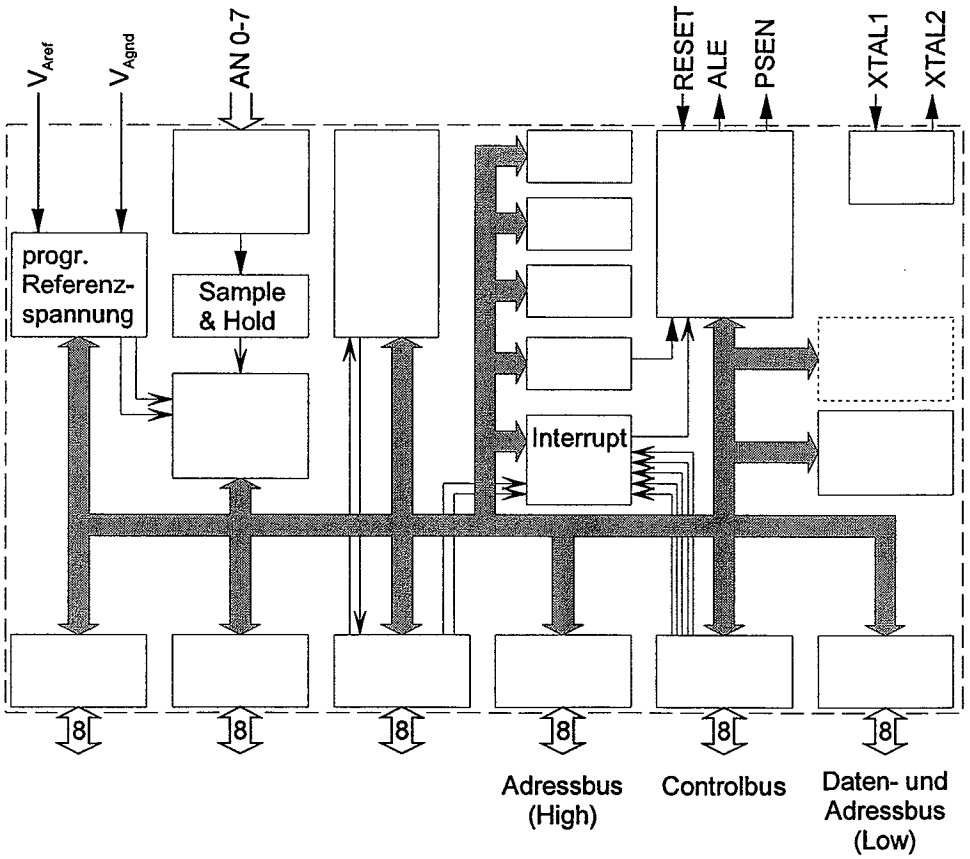
Wir begrenzen uns auf die Familie SAB-80535 von Siemens, da wir diesen Baustein für die praktischen Übungen einsetzen werden.

Typen

Typ	int. ROM	int. RAM	Digital I/O	Analog I/O	Auflösung	Serial I/O	Timer	Div.
8051	4 kB	128 byte	32	-	-	1	2	
8031	-	128 byte	32	-	-	1	2	
8751	4 kB	128 byte	32	-	-	1	2	EPROM
8052	8 kB	128 byte	32	-	-	1	2	
8032	-	128 byte	32	-	-	1	2	
80C515	8 kB	256 byte	56	8	8 bit	1	3	Watch-
80C535	-	256 byte	56	8	8 bit	1	3	dog
80C515A	8 kB	1.2 kB	56	8	8 bit	1	4	Watch-
80C535A	-	1.2 kB	56	8	10 bit	1	4	dog
80C517	8 kB	256 byte	56	12	8 bit	2	5	Watch-
80C537	-	256 byte	56	12	8 bit	2	5	dog
80C517A	32 kB	2.25 kB	56	12	10 bit	2	5	Watch-
80C537A	-	2.25 kB	56	12	10 bit	2	5	dog

DIE ELEMENTE DES 80C535

Blockschaltbild



Register

Akku

Der Akkumulator wird im Assembler mit dem impliziten Namen **A** angesprochen. Da sich der Akkumulator im internen RAM befindet und deshalb über eine Adresse angesprochen werden kann, ist auch noch der Name **ACC** definiert. ACC entspricht der Adresse $F8_h$ im internen RAM.

Der Akkumulator ist das zentrale Register in der CPU, das für praktisch alle arithmetischen und logischen Operationen benötigt wird.

B-Register

Das B-Register wird für die Multiplikation und Division verwendet, die der Befehlssatz des 80535 zur Verfügung stellt.

Allg. Register

Die unteren 32 Byte des internen RAM sind in vier Registerbänke mit je 8 allgemein benutzbaren Registern aufgeteilt. Die Register lassen sich über die impliziten Namen **R0 - R7** ansprechen.

Die Register stehen dem Anwender zur freien Verfügung; allerdings kann zu einem Zeitpunkt immer nur eine Registerbank aktiv sein.

Auch die allgemeinen Register können über Adressen im internen RAM-Bereich angesprochen werden, wobei die Adressierung über die Namen R0 - R7 schneller abgearbeitet wird.

PSW

Dieses Register enthält die Informationen über den aktuellen Programmstatus

- CY; Übertrag bei arithmetischen Operationen
- AC; Übertrag bei BCD-Operationen
- F0, F1; Benutzer-Flag 0 und 1, zur freien Verfügung
- RS0, RS1; Auswahl der Registerbank
- OV; Überlauf
- P; Parität des Akkumulators

Stack Pointer

Der Stackpointer ist 8 Bit breit. Sein Wert wird erhöht bevor ein Datenbyte mittels der Befehle PUSH bzw. CALL auf dem Stack abgelegt wird und erniedrigt, nachdem ein Datenbyte mittels der Befehle POP bzw. RETI vom Stack gelesen wurde.

Der Stackpointer zeigt immer auf das zuletzt abgespeicherte Byte! Der Stack muss sich im Adressbereich von 0..255 befinden!

Datenpointer

Der Datenpointer ist ein 16-Bit Register und dient der Adressierung von externem Speicher. Sobald die Befehle MOVX bzw. MOVC ausgeführt werden, liest der 80535 die entsprechende Adresse aus dem Datenpointer.

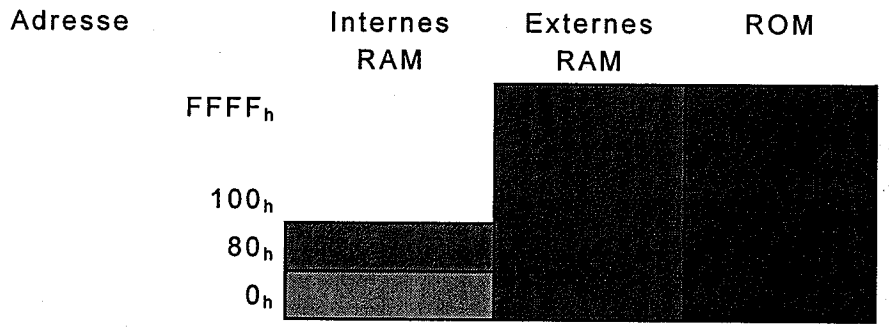
Port

Die Ports des 80535 können über die impliziten Namen P0 - P5 byteweise angesprochen werden. Um ein einzelnes Bit zu lesen wird dessen Position zusätzlich angegeben, z.B. P5.3 um das 4. Bit des Ports 5 zu lesen.

Alle Eingänge die über einen impliziten Namen wie z.B. RxD verfügen, lassen sich auch so ansprechen.

Speichermodell

- Harvardstruktur
- 256 Byte internes RAM (Adressen 00_h...FF_h)
- SFR[1] für Peripherie usw. (Adressen 80_h...FF_h; "shared" mit internem RAM!)
- max. 64 kByte externes RAM (Adressen 0000_h...$FFFF_h$)
- 8 kByte internes ROM (Adressen 000_h...FFF_h)[2]
- max. 64 kByte externes ROM (Adressen 0000_h...$FFFF_h$)



Programm

Während der 80515 über ein internes ROM von 8 kByte verfügt, weist der 80535 kein internes ROM auf. Über den EA-Pin kann festgelegt werden, ob internes oder externes ROM verwendet wird.

Beachte

Sobald ein externes ROM benutzt wird, sind die Ports P0 und P2 durch Adress- und Datenbus sowie Port P1 teilweise durch den Kontrollbus belegt!

Der Adressbereich 03_h...$6F_h$ wird für die **Interrupteinsprung-vektoren** benutzt und sollte daher nicht mit Code belegt werden! Dies lässt sich mit einer **ORG-Direktive** oder einem entsprechenden Linker-Befehl erreichen.

[1] Special Function Register
[2] nur 80515

Die Adressen $00_h..02_h$ stellen den **RESET-Vektor** dar, d.h. dass nach einem Kaltstart der Programmzähler die Adresse 0000_h ausgibt. Meist wird in diesen 4 Byte ein Sprungbefehl zum eigentlichen Programm eingetragen.

Externes RAM

Das externe RAM ist völlig gelöst vom internen, d.h. dass es keinerlei Adresskonflikte gibt, da für den Zugriff ein spezieller Befehl (MOVX) verwendet wird.

Internes RAM

Der "Ur-Vater" der 8051-Reihe verfügte nur über 128 Byte internes RAM, so dass sich keinerlei Probleme mit überlappenden Speicherbereichen[3] ergab.

Sollen Daten in diesem Bereich abgelegt werden, muss dem Assembler bzw. dem C-Compiler mittels einer **DATA-Direktiven** dies mitgeteilt werden! Dieser Speicherbereich lässt sich mit den folgenden Adressierungsarten[4] ansprechen:

- Immediate `MOV DATA_VAL, #12h`
- Direct `MOV DATA_VAL, VAL_ADRESS`
- Indirect `MOV DATA_VAL, @R0`

Die ersten 32 Byte des internen RAM sind den 4 **Registerbänken** zugewiesen. Diese können über die symbolischen Namen R0...R7 angesprochen werden.

Beachte
Es ist nur immer ein der vier Registerbänke aktiv. Die Auswahl erfolgt über 2 Bit im PSW-Register!

Beispiel :

Ein sinnvoller Einsatz der Registerbänke ist z.B. Multitasking, wobei jedem Task 8 "eigene" Register für den Task-Stack zur Verfügung stehen. Dadurch begrenzt sich ein Task-Switch - das Umschalten einer Task A zur Task B - auf das Umbelegen der beiden Bits im PSW.

Beim 80515er wurde das interne RAM um zusätzliche 128 Byte erweitert. Diese sind adressmässig identisch mit den SFR.

Sollen Daten in diesem Bereich abgelegt werden, muss dem Assembler bzw. dem C-Compiler mittels einer **IDATA-Direktiven** dies mitgeteilt werden! Dieser Speicherbereich lässt sich nur mit der Adressierungsart Indirect ansprechen.

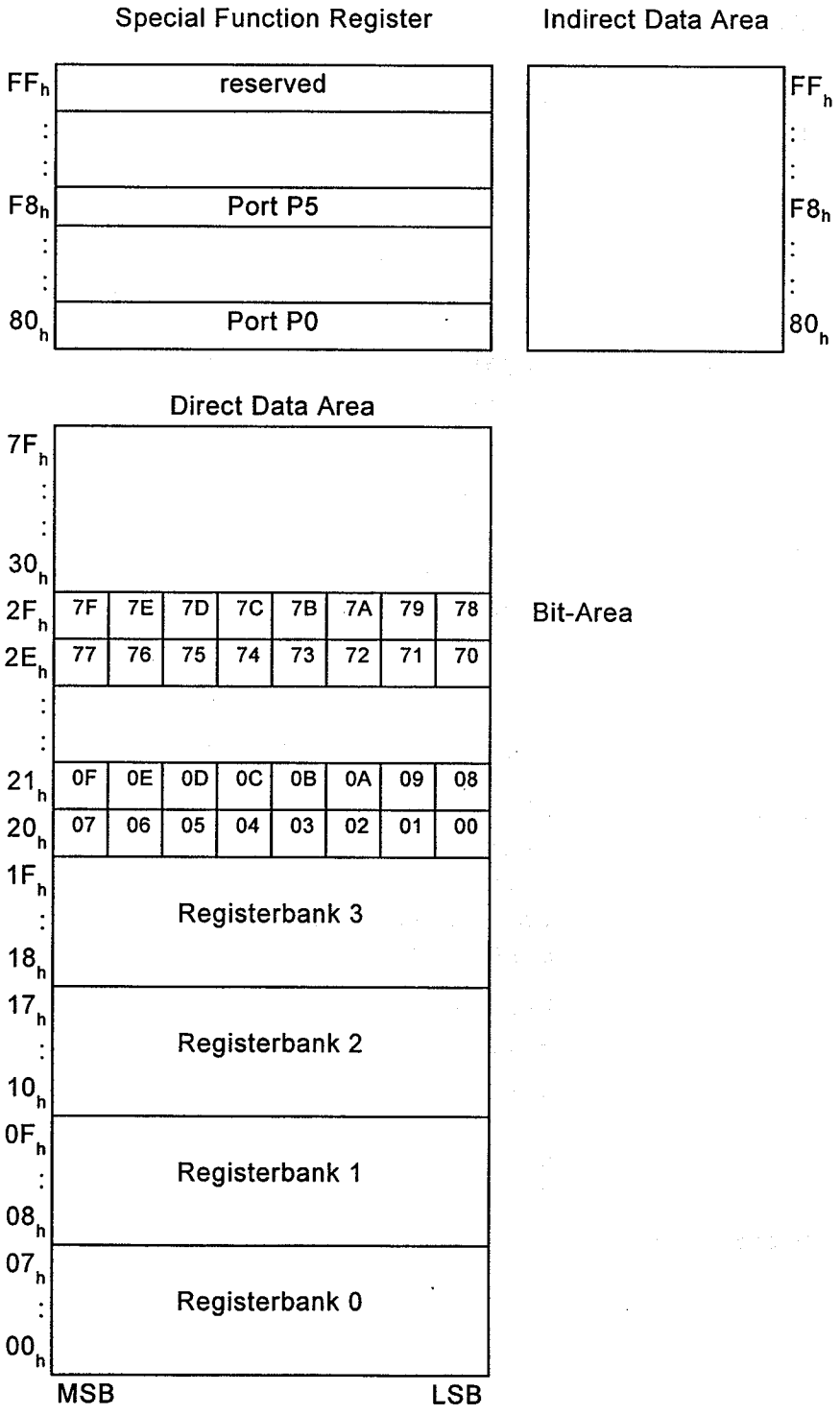
Bitspeicher

Innerhalb des beschriebenen RAM-Bereichs sind die Adressen $20_h..2F_h$ speziell zu beachten, da diese **bitadressierbar** sind, d.h. dass hier jedes der 128 Bit eine eigene Adresse $00_h..7F_h$ erhält.

[3] Shared Memory
[4] Sie werden dazu später genaueres erfahren

Soll das interen RAM als Bit-Speicher benutzt werden, muss auch hier mit der **BIT-Direktiven** dies bei der Programmerstellung bekannt gemacht werden.

Überblick über das interne RAM und die SFR



SFR

Die Special Function Register dienen der Programmierung der Peripherie, d.h. über die SFR kann z.B. auf den Baudratengenerator der seriellen Schnittstelle, das Register eines Timers usw. zugegriffen werden. Sie lassen sich nur über die Adressierungsarten Immediate und Direct ansprechen.

Neben den "einfachen" Registern wie P0-P5, SP[5], DPTR[6] usw. weist der 80535 viele Register für spezielle Aufgaben auf. Diese werden jeweils bei entsprechenden Übungen eingeführt!

Dem Assembler bzw. C-Compiler werden die Namen der Register über eine spezielle Include-Datei bekannt gemacht. Diese trägt z.B. den Namen REG515* oder ähnlich.

Beispiel :

Assembler: `$INCLUDE(C:\MICRO51\ASM\REG515.INC)`
C : `#include <reg515.h> /* SFR from 80515 */`

Timer

Der 80535 verfügt über 3 Timer. Timer 0 und Timer 1 sind allgemein brauchbare Zeitgeber mit 8- oder 16-Bit Wert.

Timer 2 ist mit weiteren Logik-Einheiten des Mikrokontrollers verbunden. Damit lassen sich auch spezielle Aufgaben wie Vergleichsmessungen, Pulsweitenmodulation usw. realisieren.

Interrupt

Bedingt durch die im Mikrokontroller eingebauten Funktionen muss eine „Interrupt-Verwaltung" verfügbar sein.

Die verschiedenen Interrupt-Quellen müssen

- selektiv aktiviert werden können (prinzipiell sind die Interrupts inaktiv)
- über Prioritäten verfügen
- das Abfragen des Zusatndes ermöglichen

Interruptvektor

Jeder Interrupt-Quelle ist eine eindeutige Adresse im Programmspeicher zugeordnet (= Einsprungadresse). Dort liest die CPU die Adresse (= Interruptvektor) der ISR[7] aus um dann zu verzweigen.

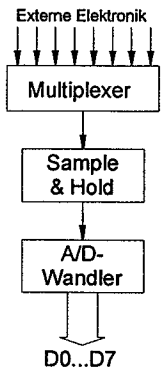
Interrupt Quelle	Auslösendes Moment	Flag	Reset	Einsprung-Adresse
-	Reset	-	-	00_h
Externer Interrupt 0	neg. Flanke / Pegel an P3.2	IE0	HW	03_h
Timer 0 Interrupt	Timer 0 Überlauf	TF0	HW/SW	$0B_h$

[5] Stack-Pointer
[6] 16-Bit Data-Pointer mit DPL und DPH
[7] Interrupt Service routine

Interrupt Quelle	Auslösendes Moment	Flag	Reset	Einsprung-Adresse
Externer Interrupt 1	neg. Flanke / Pegel an P3.3	IE1	HW	13_h
Timer 1 Interrupt	Timer 1 Überlauf	TF1	HW	$1B_h$
Serielle Schnittstelle	Ende der Eingabe an P3.0 Ende der Ausgabe an P3.1	RI TI	SW	23_h
Timer 2 Interrupt	Timer 2 Überlauf od. externer Reload an P1.5	TF2 EXF2	SW	$2B_h$
A/D-Wandler	Ende der Wandlung	IADC	SW	43_h
Externer Interrupt 2	neg./pos. Flanke an P1,4	IEX2	HW	$4B_h$
Externer Interrupt 3 od. Capture 0 Eingang od. Compare 0 Ausgang	neg./pos. Flanke an P1.0 neg./pos. Flanke an P1.0 neg./pos. Flanke an P1.0	IEX3	HW	53_h
Externer Interrupt 4 od. Capture 1 Eingang od. Compare 1 Ausgang	pos. Flanke an P1.1 pos. Flanke an P1.1 neg./pos. Flanke an P1.1	IEX4	HW	$5B_h$
Externer Interrupt 5 od. Capture 2 Eingang od. Compare 2 Ausgang	pos. Flanke an P1.2 pos. Flanke an P1.2 neg./pos. Flanke an P1.2	IEX5	HW	63_h
Externer Interrupt 6 od. Capture 3 Eingang od. Compare 3 Ausgang	pos. Flanke an P1.3 pos. Flanke an P1.3 neg./pos. Flanke an P1.3	IEX6	HW	$6B_h$

A/D-Wandler

Um auch analoge Signale verarbeiten zu könne, verfügt der 80535 einen 12-Bit A/D-Wandler.



Dabei stehen 8 Eingänge zur Verfügung, die über einen Multiplexer ausgewählt und dem A/D-Wandler zugeteilt werden können.

Die Analog-Eingänge lassen sich bei Bedarf auch für digitale Signale verwenden. Dadurch erhöht sich die Anzahl der verfügbaren Port's.

Watch-Dog

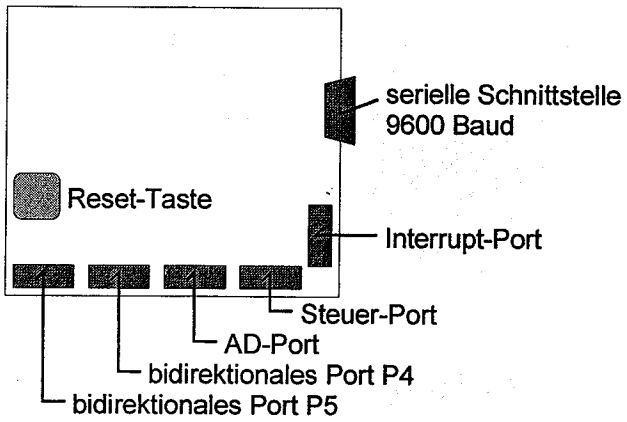
Beim Watch-Dog handelt es sich um einen eingebauten Zähler, der durch die Software laufend wieder neu gestartet werden muss. Erfolgt kein Neustart, so dass der Zähler abläuft, wird ein spezieller Code ausgeführt, der ein geordnetes Tieffahren des Systems ermöglicht → kein Absturz, auch wenn sich das eigentliche Programm „aufhängt".

DAS TBZ-SYSTEM

Die CPU1 enthält einen **80535** mit einem einfachen Monitor-Programm. Dieses erlaubt das Laden eines Programms[1] ab **Adresse 8000_h** im RAM und das Ausführen des Programms. Über die Stecker lassen sich verschiedene Versuchseinrichtungen anschliessen.

CPU-Board

Layout



Pin-Belegung

Für das Anschliessen von Peripherie sowie die Kommunikation mit einem Entwicklungssystem – dem PC – stehen folgende Anschlüsse zu Verfügung:

- Port P5: ① P5.0 ... ⑧ P5.7
- Port P4: ① P4.0 ... ⑧ P4.7
- AD-Port: ① AN0 ... ⑧ AN7
- Steuer-Port

①	T0 Zähler Eingang	P3.4
②	T1 Zähler Eingang	P3.5
③	T2 Zähler Eingang	P1.7
④	T2EX Reload Eingang	P1.5
⑤	WR*	P3.6
⑥	RD*	P3.7
⑦	PSEN	PSEN
⑧	ALE	ALE

- Interrupt-Port

①	INT0	P3.2
②	INT1	P3.3
③	INT2	P1.4
④	INT3	P1.0
⑤	INT4	P1.1
⑥	INT5	P1.2
⑦	INT6	P1.3
⑧	CLK	P1.6

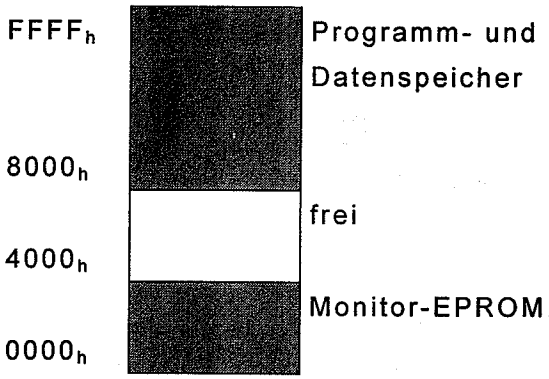
- Speisung ⑨ +5V / ⑩ 0V (fest verdrahtet!)

[1] über die serielle Schnittstelle eines PC's

Beachte
Die Ausgänge sind ungeschützt!
Es dürfen nur die vorgesehenen Versuchseinrichtungen angeschlossen werden.

Speicher-organisation

Die Speicherorganisation der CPU1 entspricht nicht der üblichen Harvard-Architektur der 8051-Familie. Die beiden Steuerleitungen RD- und PSEN- sind so zusammengefasst, dass eine von Neumann-Architektur realisiert wird, d.h. dass der Programm-Code und die Daten *im gleichen Speicherbereich* liegen. Der externe Speicher teilt sich dabei wie folgt auf:



Beachte
Werden Daten im externen RAM abgespeichert, müssen die Segmente so gelegt werden, dass sie sich gegenseitig nicht überlappen. Weder der Assembler noch der Linker können dies sicherstellen, da sie für eine Harvard-Architektur ausgelegt sind.

Der Bereich von 0000_h - $7FFF_h$ ist dabei für ROM-Speicher reserviert und kann nicht beschrieben werden. Im Bereich 8000_h - $FFFF_h$ befindet sich ein 32k x 8 RAM.

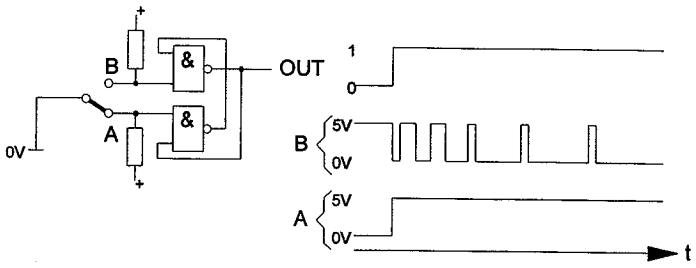
Speisung

Das 80535-Board wird mit einer Wechselspannung von 7.5 Volt versorgt. Diese wird auf dem Print in eine 5 Volt Gleichspannung umgewandelt.

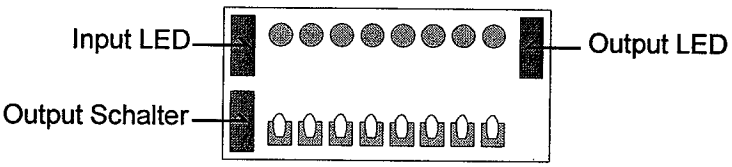
Beachte
Die Speisung der externen Versuchseinrichtungen erfolgt über das CPU-Board.

LED-Versuchsplatine

Sie stellt 8 Ausgänge in Form von LED sowie 8 Eingänge in Form von prellfreien Schaltern zur Verfügung.



Layout

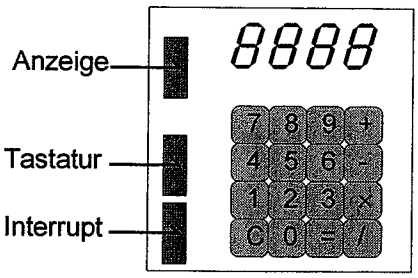


Taschenrechner

Als Experiment ist ein einfacher Taschenrechner mit den vier Grundoperationen +, -, :, × verfügbar. Er weist eine 4-stellige 7-Segment-Anzeige auf, d.h. dass sich der darstellbare Zahlenbereich von –999...9999 begrenzt.

Layout

Die Anzeige wird mit Vorteil über Port 5, die Tastatur über Port 4 angeschlossen. Das von der Tastatur erzeugte Interruptsignal sowie der Steuerausgang des Signalgebers werden über das Interrupt-Port angeschlossen.



Treiber, API

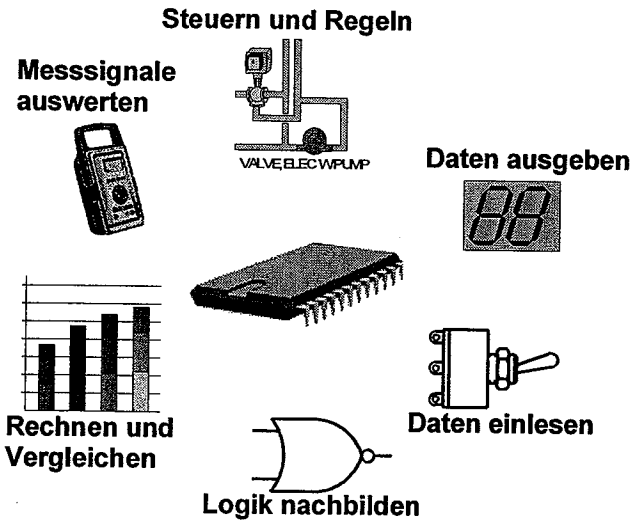
Der Tastatur-Treiber ist verfügbar (weil kompliziert), ebenso auch eine `getc`-Funktion. Der Anzeigentreiber wie auch eine `printf`-Prozedur sind selber zu erstellen.

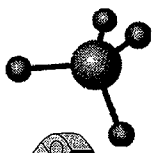
Dokumentation

Jedem Experimentier-Set ist eine umfassende Dokumentation beigelegt, die auch Theorie zu Hard- und Software[2] umfasst. So können Sie mittels einer zweiten Quelle im Selbststudium das Wissen vertiefen.

[2] nur Assembler

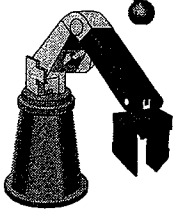
AUFGABEN UND ANWENDUNGEN VON µP





Wissenschaft und Forschung

Spektralanalysegeräte, Manipulatoren, Signalsynthesizer, wissenschaftliche Rechner usw.



Industrie und Handel

Fliessbandsteuerungen, numerisch gesteuerte Werkzeugmaschinen, Kopierautomaten usw.

Nachrichten- und Datentechnik, Energieversorgung

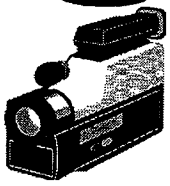
Drucker, intelligente Terminals, Fernsprech-Wählautomaten, Schreibautomaten, Energiezähler usw.

Verkehrstechnik

Autoelektronik (Einspritzung, ABS, KFZ-Diagnose), Tankstellen (Durchflussmessung, Abrechnung) Bahn, Flugzeuge usw.

Medizinische Geräte

Analyseautomaten, EKG- und EEG-Geräte, Röntgengerätesteuerung, Diagnosegeräte usw.



Konsumelektronik

Waschmaschine, Mikrowellenherd, Fernseher, Video-Gerät, Spielautomaten usw.

ASSEMBLER

GRUNDLAGEN

Struktur

Sie haben im Unterricht bereits gelernt, wie ein Hochsprachenprogramm aufgebaut ist. Auch die Assembler[1] kennen eine definierte Struktur, wie ein Programm aufzubauen ist.

Beispiel : 80535-Assembler

```
$TITLE (****)
$NOLIST
$INCLUDE (C:\MICRO\WORK51\REG535N.INC)
$LIST
; Kommentar
NAME      DIRECT
;
; Bekanntgabe der benötigten Segmente
MAIN  SEGMENT   CODE
;
          ORG 8000h
;
; Programm zeigt die indizierte Adressierung, die über ...
; ... die Register PC und DPTR realisiert werden kann.
;
          CSEG AT 8080h
CODE_TAB: DB        01H,02H,04H,08H,10H,20H,40H,80H
                    ; zusammenhängender Speicherbereich im ...
                    ; ... ROM initialisieren; er wird über das...
                    ; ... DPTR-Register angesprochen.
;
;
          RSEG  MAIN
JMP   DS_INIT
;
          CSEG AT 8100h
          ; Da die Transfer-Software am Ende 5 x SWAP A erwartet, muss...
          ; ...der Code hinter die Daten gelegt werden!
          ;
          ; Beginn des eigentlichen Programms
DS_INIT:  MOV  DPTR, #CODE_TAB    ; Indexregister mit Zeiger auf die ...
                                  ; ... Adresse der Daten laden
;              Ziel    Wert
REG_INIT: MOV  A,#0               ; Offsetwert auf Tabelle
;
LOOP:     PUSH ACC                ; Offsetwert in Register speichern ...
          MOVC A,@A+DPTR          ; ...und Tabelle lesen
          MOV  P5,A               ; Wert an Port ausgeben (Reg.-Adressierung)
          CALL WAIT
          POP  ACC                ; Offset verarbeiten
          INC  A                  ;
          CJNE A,#8,LOOP          ; 8 Werte durch arbeiten ...
          JMP REG_INIT            ; ...danach neu beginnen
;
          :
          :
```

[1] Jeder Prozessor weist einen spezifischen Satz von Befehlen auf

Sprache des µP

Binärcode

Der Mikroprozessor verarbeitet die Signale als Spannungswerte im Bereich von 0 Volt und 5 Volt[2]. Diese werden als logische Signal 0 (= 0V) und 1 (= 5V) interpretiert.

Somit müsste ein Befehl inkl. den Daten z.B. wie folgt aussehen

`00110100 01101111` = 34_h $6F_h$

was für die Erstellung eines Programms absolut undenkbar ist; obwohl in den Anfängen der elektronischen Computer die Programm direkt als Binärmuster eingegeben werden mussten.

Eine gewisse „Erleichterung" ist das Schreiben in Hexadezimal-Code, auch wenn das dem Verständnis nicht mehr dienlich ist.

Mnemonic[3]

Durch das Verwenden einer Symbolik lässt sich dies verhindern. So wird obiger Befehl als

`ADDC A,#111` (Addiere Akku mit konstantem Wert, mit Carry)

angeschrieben, was sicherlich um einiges verständlicher ist.

80535-BEFEHLE

Hinweis
Sie werden hier nur eine kurze Übersicht erhalten. Eine ausführliche Zusammenstellung finden Sie in der Dokumentation, die jedem Experimentier-Set beiliegt.

Übersicht

Neben den eigentlichen Befehlen für den Prozessor werden auch spezielle Befehle für die Steuerung des Assemblers und das zuweisen des Speichers benötigt.

Befehle

Der 8051-Befehlssatz gliedert sich in vier Gruppen

- Transportbefehle
- Arithmetikbefehle
- Logikbefehle
- Steuerbefehle

Transportbefehle

Es gibt zwei Transportanweisungstypen:

- Allgemeine Speicheroperationen
- Spezielle Akkumulatorbefehle

Diese Instruktionen beeinflussen bis auf ein POP oder MOV in das PSW-Register dieses nicht. Nach jedem Maschinenzyklus wird das Parity-Bit aus dem Inhalt des Akkumulators gebildet und im PSW abgelegt.

[2] Moderne Chip arbeiten mit tieferen Spannungen, um die Verlustleistung (P=U·I) zu verkleinern.

[3] Mnemonikos = „das Gedächtnis betreffend"

Generelle Transport-Anweisungen

MOV kopiert Bits oder Bytes vom Quell- zum Zieloperanden.

PUSH inkrementiert das Register SP (Stack Pointer) und kopiert dann ein Byte vom Quellen- zum Zieloperanden, der durch den SP adressiert wird.

POP kopiert ein Byte vom durch den SP adressierten Quelloperanden zum Zieloperanden und dekrementiert den SP.

MOV DPTR lädt die angegebene 16-Bit-Konstante in die Zieladressen DPH und DPL.

Spezielle Akkumulator-Befehle

XCH tauscht das Byte des Quelloperanden mit dem Inhalt von A (Akkumulator) aus.

XCHD vertauscht das niederwertige Nibbel (Bits 0-3) im Byte des Quelloperanden mit dem niederwertigen Nibbel des Akkumulators.

MOVX befördert ein Byte zwischen dem externen Datenspeicher und A. Die externe Adresse kann durch DPTR (16-Bit-Data-Pointer), R0 oder R1 (8-Bit-Register der aktuellen Registerbank) spezifiziert sein.

MOVC befördert ein Byte vom Programmspeicher zum A. Der Operand in A wird dabei als Index für eine 256-Byte-Tabelle mit DPTR oder PC als Basis verwendet.

Arithmetik-Operationen

Die vier möglichen arithmetischen 8-Bit-Befehle (Addition, Subtraktion, Multiplikation und Division) berücksichtigen keine Vorzeichen. Mittels des Überlaufflags (OV = Overflow) lassen sich vorzeichenbehaftete Additionen und Subtraktionen durchführen. Ausserdem sind "gepackte" Dezimaloperationen möglich. Gepackt bedeutet zwei BCD-Ziffern in einem Byte (je eine in einem Nibbel).

Addition

INC addiert zum Inhalt des Quelloperanden eine 1 und legt das Ergebnis im Quelloperanden ab.

ADD bewirkt eine Addition des Quelloperanden zum Inhalt von A. Das Ergebnis steht in A.

ADDC addiert den Inhalt des Quelloperanden und des CY (Carry Flag) zum Inhalt von A. A enthält das Resultat.

DAA (Dezimal Adjust) korrigiert das Resultat einer binären Addition im Akkumulator in zwei dezimale Ziffern. Die gepackte Dezimalsumme steht in A. Ist die Summe grösser als 99, wird CY gesetzt, andernfalls gelöscht.

Subtraktion

SUBB subtrahiert den Quelloperanden vom Inhalt in A und subtrahiert zusätzlich eine 1, wenn CY gesetzt war. Das Resultat geht nach A.

DEC subtrahiert eine 1 vom Quelloperanden. Das Resultat wird im Quelloperanden abgelegt.

Multiplikation

MUL multipliziert den vorzeichenlosen Inhalt von A mit dem vorzeichenlosen Inhalt des B-Registers. Vom 2-Byte-Resultat kommt das niederwertige Byte nach A und das höherwertige nach B. Ist das Resultat > 255, wird OV gesetzt (Ergebnis 2 Byte lang), andernfalls wird es gelöscht. CY wird gelöscht, AC wird nicht beeinflusst.

Division

DIV dividiert vorzeichenlos den Inhalt von A durch den Inhalt von B. Der ganzzahlige Quotient kommt nach A und der Rest nach B. Eine Division durch 0 ergibt undefinierte Inhalte von A und B und setzt OV. Andernfalls ist OV gelöscht. CY ist gelöscht und AC bleibt unbeeinflusst.

Logikan-Weisungen

ANL verknüpft zwei Operanden bitweise durch einen logischen UND-Operanden (Bit- und Bytebefehle) und legt das Ergebnis im ersten Operanden ab.

ORL erzeugt mit zwei Operanden bitweise ein logisches ODER (Bit- und Bytebefehle) und legt das Ergebnis im ersten Operanden ab.

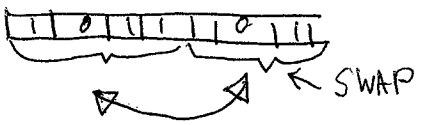
XRL bildet ein bitweises logisches EXKLUSIV-ODER zwischen zwei Operanden (Bytebefehle) und legt das Ergebnis im ersten ab.

SETB, **CLR** und **CPL** setzen, löschen oder negieren ein Bit.

Die Anweisungen (rotate) **RL**, **RLC**, **RR**, **RRC** verschieben den Akkumulatorinhalt nach links oder rechts im Kreis (Rotation!). D.h., dass beim Rechtsrotieren aus der niederwertigsten Stelle im Akkumulator rechts herausgeschobene Bit wird in das höchstwertige Akkumulatorbit links wieder eingefügt. Beim Linksrotieren ändert sich der Ablauf entsprechend in der entgegengesetzten Richtung. Bei beiden Anweisungsarten kann die Information auch durch das C-Bit (zwischen höchst- und niederwertigstem Akkumulatorbit) geschoben werden.

CLR A löscht den Akkumulator, während **CPL A** den Inhalt invertiert.

SWAP führt einen 4-Bit-Kreisshift im Akkumulator aus (tauscht die niederwertigen 4 Bit mit den höherwertigen 4 Bit).



Sprung-Instruktionen

Es gibt drei Sprungtypen:

- Unbedingte Sprünge (Call's, RET)
- Bedingte Sprünge
- Interrupts

Alle Sprungbefehle, einige von Bedingungen abhängig, bewirken die Unterbrechung des kontinuierlichen, sequentiellen Programmablaufs, um an einer anderen Stelle fortzufahren.

Unbedingte Sprünge

AJMP und **LJMP** sind einfache Sprungbefehle mit einem 12- bzw. 16-Bit-Adressteil. Der SJMP (short jump) verzweigt innerhalb eines Bereichs von 256 Byte (8-Bit-Adresse) mit der maximalen Distanz von -128 bis + 127, bezogen auf die Adresse des auf den Sprung unmittelbar folgenden Byte.

ACALL und **LCALL** (Unterprogrammaufruf) inkrementieren den PC um 2 und laden die so ermittelte Adresse des folgenden Befehls in den Stack. Dann verzweigen sie zur Zieladresse. ACALL ist eine 2-Byte-Operation. Sie wird verwendet, wenn die Zieladresse in der momentanen 2-kByte-Seite liegt. LCALL ist ein 3-Byte-Befehl mit einer 16-Bit-Adresse für den gesamten Adressraum.

Im ACALL-Adressteil befindet sich ein unmittelbares 11-Bit-Adressfeld. Die höchstwertigen 5 Bit im PC bleiben unverändert. Dabei ist auf folgendes zu achten! Befindet sich ein ACALL in den 2 letzten Byte einer 2-kByte Seite des Programmspeichers, springt das Programm zur nächsten Seite, weil der PC vor der Befehlsausführung inkrementiert wird und sich damit die höherwertigen 5 Bit im PC verändern.

RET führt die Rückkehr aus einem Unterprogramm durch. Die Rücksprungadresse (2 Byte) wird dabei über den SP aus dem Stack in den PC übernommen und der SP um 2 dekrementiert.

JMP @A+DPTR springt relativ zur Adresse im DPTR. Der Operand A dient als Index (0 bis 255). Vor der Befehlsausführung errechnet sich die Zieladresse durch die Summe der Inhalte von A und DPTR. Das Ziel kann überall im Programmspeicher liegen.

Bedingte Sprünge

Die Ausführung dieser Verzweigungen ist von Bedingungen abhängig. Die Zieladresse liegt immer im Bereich von 256 Byte. Die maximale Sprungweite ist -128 bis +127 Byte, bezogen auf die Adresse des auf den "JMP" unmittelbar folgenden Byte.

Befehl	*Sprung wird ausgeführt, wenn Inhalt*
JZ	von A gleich O ist
JNZ	von A ungleich O ist
JC	von C (Carry = > gesetzt)
JNC	von C (Carry = > O)

JB des direkt adressierten Bit 1 ist

JNB des direkt adressierten Bit O ist

JBC des direkt adressierten Bit 1 ist, das Bit wird gelöscht.

CJNE (Vergleiche und springe, wenn nicht gleich): vergleicht den 1. Operanden mit dem 2. Operanden und springt, wenn sie nicht gleich sind. CY wird gesetzt, wenn der 1. Operand kleiner als der 2. Operand ist; andernfalls wird es gelöscht. Vergleiche sind zwischen A und direkt adressierbaren Bytes im Datenspeicher oder zwischen Konstanten und entweder A, einem Register in der selektierten Registerbank, oder indirekt adressierbaren Byte im Datenspeicher möglich.

DJNZ (Dekrementiere und springe, wenn nicht 0): "DJNZ" dekrementiert den Inhalt des Quelloperanden und legt das Ergebnis in den Quelloperanden zurück. Es wird gesprungen, wenn das Ergebnis nicht 0 war. Der Quelloperand des DJNZ kann ein direkt adressiertes Byte im internen Datenspeicher oder ein Register sein.

Interrupt-"RETURN"

RETI (Rückkehr aus der Interrupt-Service-Routine): Dieser Befehl entspricht der Anweisung RET. Zusätzlich wird die aktuelle Prioritätsebene wieder freigegeben.

PSW-Register

CY wird gesetzt, wenn die Operation einen Übertrag vom oder zum resultierenden höherwertigen Bit ergibt. Andernfalls ist es gelöscht.

AC wird gesetzt, wenn die Operation einen Übertrag (während der Addition) von den niederwertigen 4 Bit oder einen Untertrag (borrow) von den höherwertigen 4 Bit (während der Subtraktion) ergibt. Andernfalls ist AC gelöscht.

OV wird für die vorzeichenbehaftete Arithmetik benötigt. OV steht auf 1, wenn bei einer Addition oder Subtraktion der vorzeichenbehaftete Zahlenbereich (-128 (80_H) bis 127 ($7F_H$)) verlassen wird und bei einer Division durch 0.

P-Flag wird gesetzt, wenn die Modulo-2-Summe (Zahl der gesetzten Bits) im Akkumulator 1 ist (ungerade). Andernfalls wird es gelöscht (geradzahlig). Ein in das PSW geschriebener Wert beeinflusst P nicht, da P in jedem Maschinenzyklus aus dem Inhalt des Akkumulators neu ermittelt und entsprechend gesetzt wird.

Direktiven

Neben den Prozessorbefehlen weist ein Assembler auch sogenannte Direktiven und Steuerbefehle auf. Das sind Befehle, die keinen Einfluss auf den Programmablauf haben, dem Programmierer aber gewisse Hilfsmittel zur Programmerstellung geben.

EQU Definieren eines Symbols, vergleichbar einer Konstante bei Hochsprachen.

Beispiel:
```
MAX_VAL   EQU  0FH
LIMIT     EQU  MAX_VAL - 1
```

SET Definieren eines Symbols, wobei dieses aber an beliebiger Stelle im Programmcode später mittels eines SET geändert werden kann.

Beispiel:
```
VALUE     SET  100
VALUE     SET  VALUE / 2
```

BIT Definieren eines Symbols (Adresse einer Variable), welchem eine Bitadresse zugewiesen wird.

Beispiel:
```
ALARM     BIT  CTRL.0
XON  BIT  60H
```

DATA Definieren eines Symbols (Adresse einer Variable) im internen Datenspeicher. Der Ausdruck muss eine Zahl zwischen 00_h und FF_h besitzen.

Beispiel:
```
RESULT    DATA 40H
RESULT2   DATA RESULT + 2
```

IDATA Definieren eines Symbols (Adresse einer Variable), das sich auf eine indirekt adressierbare interne Datenadresse bezieht.

Beispiel:
```
BUFFER    IDATA     A0H
BUF_LEN   EQU  20H
BUF_END   IDATA     BUFFER + BUF_LEN
```

XDATA Definieren eines Symbols (Adresse einer Variable) im externen Datenspeicher.

Beispiel:
```
TIME  XDATA    2000H
PLACE XDATA    TIME + FFH
```

CODE Definieren eines Symbols als Sprungadresse im Programmspeicher.

Beispiel:
```
RESTART   CODE 00H
INTVECT_0 CODE 03H
RS232_INT CODE 23H
```

DS Reservieren eines Speicherbereichs im RAM mit der entsprechenden Anzahl Stellen, wobei die erste Stelle unter dem Namen eines optionalen Labels ansprechbar ist.

Beispiel:
```
SIZE EQU  4
RESULT:   DS   SIZE
```

DB Initialisieren einer Folge von Bytes im Programmspeicher. Die Speicherstelle kann über ein optionales Label angesprochen werden.

Beispiel:
```
REQUEST:  DB   'Enter any key',0
TABEL:    DB   0,1,2,4,8
CHARS:    DB   'A','B','C'
```

DW Initialisieren einer Folge von Words im Programmspeicher.

Beispiel:
```
TABEL:    DW   1000,1100,1200
STRING2:  DW   'X0','Y0','FF'
```

ORG Die Programmadresse, auf welche das folgende Programmstück "gesetzt" wird.

Beispiel:
```
ORG  8000H
```

NAME Ein optionaler Name für einen Programmcode

Beispiel:
```
NAME DEMO_1
```

Steuerbefehl

Diese Befehle werden bei der Übersetzung durch den Assembler selber ausgeführt. Sie dienen der Formatierung des Listings, dem Einbinden externer Dateien usw.

Beispiel:

$TITLE (Mein Programm)

Erscheint auf jeder Seite des Listing als Titel

$NOLIST

Unterdrückt das erzeugen eines Listings, bis mit $LIST dies wieder zugelassen wird.

$INCLUDE('Pfad'\'Dateiname')

Bindete die angegebene Datei in den Übersetzungsvorgang ein.

$EJECT

Erzwingt einen Seitenvorschub im Listing

Lokalisierung von Symbolen

Segmente

Die einzelnen Bereiche mit Symbolen, Code usw. werden als Segmente bezeichnet. Der Assembler unterschiedet in

- Code-Segmente **CODE**
- Daten-Segmente intern **DATA** und **IDATA**
- Bit-Segment **BIT**
- Daten-Segment extern **XDATA**

Ein Bereich innerhalb eines Programms wird wie folgt definiert:
SegmentName SEGMENT SegmentTyp

Beispiel :

```
int0_code_seg SEGMENT   CODE
              RSEG int0_code_seg
              :
bit_seg_name  SEGMENT   BIT
              RSEG bit_seg_name
              :
```

Relative Adressen

Das Beispiel zeigt, dass nach der Definition des Segments dieses noch aktiviert werden muss. Das erfolgt hier über die **RSEG**-Direktive. Dadurch wird das Segment als **relokativ** bezeichnet, d.h. dass die Adressen des Segments die der Assembler erzeugt relativ zur Adresse 0 sind. Erst beim Binden aller Module wird der **Linker** die gültigen Adressen berechnen.

Absolute Adressen

Soll ein Segment an einer absoluten Adresse stehen, so kann dies über den Linker, aber auch direkt im Assembler erfolgen.

Beispiel :

```
CSEG AT  0BH  ;0BH is address for Timer 0 interrupt
LJMP timer0int
```

Der Code des LJMP-Befehls wird fest auf die Adresse $0B_h$ gelegt! Als Direktiven dienen **CSEG**, **DSEG**, **ISEG**, **BSEG**, **XSEG** für die verschiedenen Bereiche.

Lable

Neben all den vordefinierten Befehlen, Direktiven usw. können auch eigene Namen für Variable, Segmente (vergleichbar mit Funktionen) und Sprungziele angeschrieben werden.

Beispiel :

```
ANZAHL: DW 1              ;Datenwort für Ganzzahl
INIT:   MOV A,#01H        ;Sprungziel eines Jump
```

Aufgabe : Markieren Sie auf Seite 2.1.1

- Direktiven mit grün
- Steuerbefehle mit blau
- 80535-Befehle mit rot
- Lokalisierungen mit braun
- Lable mit orange

EIN BEISPIEL

Anhand eines Blinklichtes wird der ganze Vorgang vom Erstellen (darauf wird jetzt aber verzichtet) über das Übersetzen bis zum Übertragen auf das Zielsystem mittels WORK21 behandelt.

EINRICHTUNG

- Schliessen Sie die LED-Versuchsplatine wie folgt mit zwei Flachbandkabeln an das CPU-Board (vergl. Skizze auf Seite 1.7.1ff)

	LED-Versuchsplatine	CPU-Board
1. Kabel	Input LED	Port 5
2. Kabel	Output Schalter	Port 4

- Verbinden Sie das CPU-Board über das RS-232 Kabel mit dem PC (an COM2)
- Stecken Sie das Netzteil an 220 Volt an.

Achtung
nun sollten alle 8 LED auf der Versuchsplatine leuchten

WORK21

Hier handelt es sich um eine integrierte Entwicklungsumgebung für den 80535-Assembler, die alle wichtigen Funktionen für Erstellung, Übersetzung usw. bereitstellt.

Beachte
Sie müssen darauf achten, dass auch wirklich die 80535-CPU ausgewählt ist
(OPTION/CHANGE CPU)

Dokumentation

Sie finden in der Dokumentation auf Seite 13–19 eine Beschreibung der Funktionen von WORK21.

Dateistruktur

WORK21 muss auf einem lokalen Laufwerk installiert sein, da es auf feste Verzeichnisstrukturen zugreift.

```
C:\-PROGRA~1-+--WORK21--+--CP1_KURS      Übungen
                        +--CP1_MONI
                        +--CP1_TEST
                        +--LINKDEMO
                        +--USERWORK      Ihr Arbeitsverzeichnis
```

userwork Kopieren Sie bei späteren Übungen die von Ihnen bearbeitete Datei jeweils ins Verzeichnis `USERWORK`.

ÜBUNG

Ziel

Sie sollen im Umgang mit WORK21 vertraut werden.

Vorgehen

Anleitung

Sie werden durch Ihren Lehrer „Schritt für Schritt" im Umgang mit WORK21 geführt. Die hier gezeigte Liste soll (nur) als Hilfsmittel und fürs Nachschlagen bei späteren Übungen dienen.

Schritte

Sie finden in der Dokumentation Seite 20 und 23 die ausführlichen Erklärungen.

1. Starten Sie das Programm über den Start-Knopf von Win95!
   (➔ *FACHKUNDE/MIKROPROZESSOREN*)
2. Laden Sie die Datei **BLINK.ASM**!
   (➔ *FILES/LOAD*)
3. Schauen Sie sich nun die Datei an; aber ändern Sie bitte nichts!
   (➔ *FILES/EDIT*)
4. Schliessen Sie den Editor wieder.
5. Übersetzen Sie die Quell-Datei!
   (➔ *ASSEMBLE/ASSEMBLE*)
6. Nun müssen Sie die entstandene Objekt-Datei „linken", damit alle Adressen zugewiesen werden!
   (➔ *LINK/LINK-SOURCE FILE*)
7. Laden Sie das Programm auf die CPU!
   (➔ *TRANSFER/TERMINAL*)

Achtung

Sobald das Terminalprogramm gestartet ist den RESET-Knopf auf dem CPU-Board drücken!
(manchmal muss das mehrmals geschehen ☺)
Mit F2 das Programm transferieren und mit 6 starten!

Juhu!!

So, nun sollte es vollbracht sein und die LED's blinken gemeinsam im Sekundentakt.

Wie weiter?

Sie werden nun das Blinklicht in ein Lauflicht abändern und dabei gleich auch einige verschiedene Möglichkeiten bei der Adressierung von Datenspeichern sehen.

ADRESSIERUNG

Nun wird das Blinklicht in ein Lauflicht abgeändert und anhand dieses Beispiels dann einige verschiedene Adressierungen behandelt.

Bitmuster

Um ein Lauflicht zu erhalten, soll immer nur eine der 8 LED brennen. Dies entspricht einer einzelnen der acht Leitung mit Logikwert 1. Die entsprechenden Bitmuster lassen sich durch folgende Hex-Zahlen darstellen:

Bitmuster	Hex-Zahl
$0000\,'0000_b$	00_h
$0000\,'0001_b$	01_h
$0000\,'0010_b$	02_h
$0000\,'0100_b$	04_h
$0000\,'1000_b$	08_h
$0001\,'0000_b$	10_h
$0010\,'0000_b$	20_h
$0100\,'0000_b$	40_h
$1000\,'0000_b$	80_h

ZIEL EINER OPERATION

Das Ziel einer Operation kann sein

- der Akkumulator — zur weiteren Verarbeitung
- ein Register — als Zwischenspeicher
- eine Speicherstelle — für Variablen

Operand

Als Operand wird das Ziel[1] sowie die Quelle[2] bezeichnet. Befehle die Operanden angeben, haben beim 80535-Mikrokontroller folgenden generellen Aufbau:

Befehl Ziel,Quelle

Beispiel :

```
MOV   A,R3
DJNZ  R5,WAIT
XRL   P5,MASKE
```

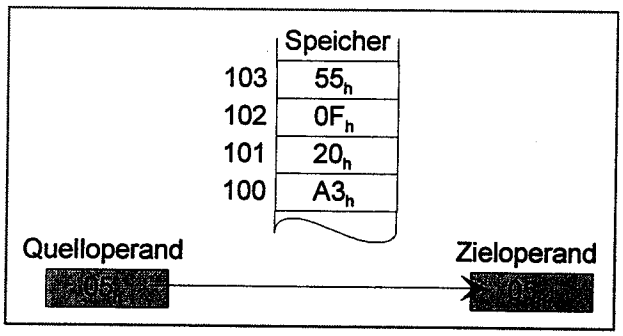
[1] auch als Senke bezeichet

[2] auch als Source bezeichnet

UNMITTELBARE ADRESSIERUNG

Konstante

Bei der unmittelbaren Adressierung (immediat) wird ein konstanter Wert in den Zieloperanden geschrieben, d.h. dass kein Zugriff auf den Speicher erfolgt.



Beispiel :

```
        :
LOOP: MOV   P5,#00h ; Wert an Port ausgeben...
      ACALL WAIT    ; ...und warten
      MOV   P5,#01h ; für alle Stellen
        :
```

Der zu speichernde Wert steht direkt im Befehl, stellt also eine **Konstante** dar. Markiert wird dieser Wert durch das #-Zeichen, so dass der Übersetzer zwischen Wert und Adresse unterscheiden kann.

Übung A

Ziel

Das Lauflicht mittels der unmittelbaren Adressierung realisieren. Das Programm soll ganz einfach gehalten werden.

Pseudocode

```
do_forever
  P5 = 01h; /* 1. LED /
  Wait();
  P5 = 02h; / 2. LED /
  Wait();
   :
   :
  P5 = 80h; /* 8. LED */
  Wait();
end;
```

Konstanten

Schauen Sie auf Blatt 2.1.14 nach, welche Werte Sie an P5 zuweisen sollen, damit ein Lauflicht resultiert!

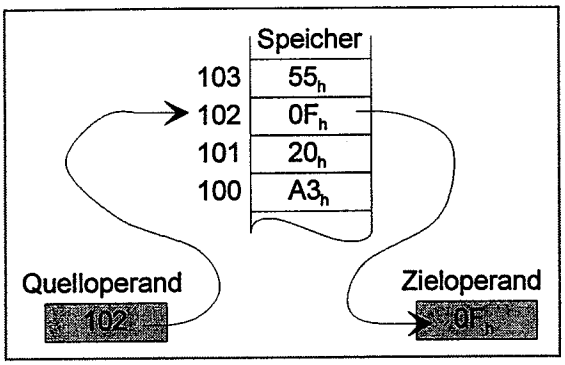
Vorgehen

1. Kopieren Sie die Datei BLINK.ASM aus dem Verzeichnis CP1_KURS als LAUF1.ASM ins Verzeichnis USERWORK!
2. Laden Sie die Datei in WORK21 und führen Sie alle notwendigen Änderungen aus!
3. Kopieren Sie die gültige Fassung auf Ihr Home-Verzeichnis!

DIREKTE ADRESSIERUNG

Variable

Bei der direkten Adressierung (direct) wird der Inhalt einer Speicherstelle bzw. eines Registers – wird durch den Quelloperanden „adressiert" – in den Zieloperanden geschrieben.



Beispiel :

```
        :
WERT:  DS 1             ; ein Byte für den Inhalt...
                        ; ...der an P5 ausgegeben wird
        :
INIT:  MOV WERT,#01h    ; Speicher initialisieren
LOOP:  MOV P5,WERT      ; Inhalt in Port laden...
        :
```

Der zu speichernde Wert ist z.B. Inhalt einer RAM-Zelle, stellt also eine **Variable** dar. Dem Befehl wird die Adresse dieser Variablen mitgegeben. Im Quellencode selber werden aber logische Namen verwendet wie z.B. `WERT`. Erst der Linker fügt hier die nötigen Adressen ein.

Beachte

Die direkte Adressierung des 80535 kennt in der Form der Register-Adressierung einen Spezialfall, bei dem die Daten direkt zwischen Registern des Kontrollers verschoben werden, was sehr schnell geht.

```
MOV A,R0     ; bringe Inhalt von R0 in Akku
```

Übung B

Ziel

Das Lauflicht mittels der direkten Adressierung realisieren. Das Programm soll mit etwas Rafinesse erstellt werden, indem z.B. ein Rotationsbefehl verwendet wird.

Pseudocode

```
WERT = 01h /* Initialisieren /
do_forever
  P5 = WERT;
  Wait();
  SchiebeLinks(WERT); /* neuen Wert berechnen */
end;
```

Variable

Sie müssen den Speicherplatz für eine Variable reservieren! Vergleichen Sie dazu das Beispiel auf der vorherigen Seiten.

Vergessen Sie dabei nicht, dass Sie einen Bereich (➔ Segment) im internen RAM bekannt geben müssen.

Tip

Ab Seite 72 der Dokumentation finden Sie eine Auflistung des Instruktions-Set[3] des 80535. Schauen Sie dort nach, wie der Befehl **RL** beschrieben wird.

Vorgehen

1. Kopieren Sie die Datei BLINK.ASM aus dem Verzeichnis CP1_KURS als LAUF2.ASM ins Verzeichnis USERWORK!
2. Laden Sie die Datei in WORK21 und öffnen Sie die Datei.
3. Erstellen Sie das benötigte Datensegment und deklarieren Sie eine Variable!
4. Erstellen Sie ein Programm, welches dem Pseudocode entspricht!
5. Wenn Sie mit dem RL-Befehl Probleme haben, melden Sie sich beim Lehrer!
6. Kopieren Sie die gültige Fassung auf Ihr Home-Verzeichnis!

EVA-Prinzip

Jedes Programm weist auch im Kleinen das EVA-Prinzip auf, da (primär) nur der Akkumulator für die Verarbeitung von Daten verwendet werden kann.

Eingabe

Der zu bearbeitende Inhalt muss vom RAM in den Akku gebracht werden.

```
mov  A,USERVAR
```

Verarbeitung

Im Akku wird der Wert durch mathematische oder logische Operationen verändert

```
rlc  A
```

Ausgabe

Der geänderte Wert wird wieder ins RAM zurückgeschrieben.

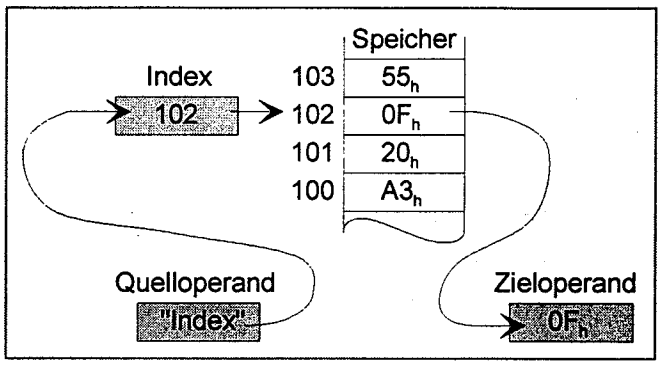
```
mov  USERVAR,A
```

[3] eine Beschreibung aller Befehle des Mikrokontrollers

INDIREKTE ADRESSIERUNG

Zeiger

Bei der indirekten Adressierung (indirect) wird über den Inhalt des Quelloperanden ein Index-Register angewählt, welches auf eine Speicherstelle bzw. ein Register verweist, dessen Inhalt in den Zieloperanden geschrieben wird.



Beispiel :

```
            :
          DSEG AT 20H ; Speicher im internen RAM
WERT: DS 1            ; ein Byte für den Inhalt...
                      ; ...der an P5 ausgegeben wird
            :
INIT:     MOV WERT,#01h ; Speicher initialisieren
REG_INIT: MOV R0,#WERT  ; Indexregister mit...
                        ; ..Adresse laden.
LOOP:     MOV P5,@R0    ; Inhalt in Port laden
          ACALL WAIT    ; ...und warten.
            :
```

Der zu speichernde Wert ist z.B. Inhalt einer RAM-Zelle, stellt also eine **Variable** dar. Dem Befehl wird aber nicht direkt die Adresse der Variable mitgegeben sondern ein spezielles Register (Indexregister), welches auf die entsprechende Speicherstelle verweist.

Beachte

Als Index-Register können **R0** und **R1** der aktuellen Registerbank sowie der Datenpointer **DPTR** dienen.

Übung C

Ziel Das Lauflicht mittels der indizierten Adressierung realisieren.

Pseudocode

```
WERT = 01h /* Initialisieren /
PTR_auf_WERT = Addresse(WERT)
do_forever
  P5 = ^PTR_auf_WERT;
  Wait();
  SchiebeLinks(^PTR_auf_WERT);
end;
```

Zeiger Sie benötigen ein Variable (wie im vorigen Beispiel), um dann deren Wert (die Adresse!) in einem Register zu speichern.

Vorgehen

1. Kopieren Sie die Datei LAUF2.ASM aus Ihrem Home-Verzeichnis als LAUF3.ASM ins Verzeichnis USERWORK!
2. Laden Sie die Datei in WORK21 und öffnen Sie die Datei.
3. Erstellen Sie ein Programm, welches dem Pseudocode entspricht!
4. Passen Sie die Adressierungen an!
5. Kopieren Sie die gültige Fassung auf Ihr Home-Verzeichnis!

Ablauforientiert/Datenorientiert

„Viele Wege führen nach Rom." Diese Aussage stimmt beim Erstellen von Programmen ganz besonders. Neben der Vielzahl von möglichen Abläufen für die Lösung eines Problems lassen sich aber zwei unterschiedliche Vorgehensweisen unterscheiden.

Ablauforientiert Hier werden die Daten durch mehr oder weniger komplexe Verarbeitungsvorschriften so verändert, dass die geforderten Ziele erreicht werden.

Beispiel : In Übung B bzw. C wird der Inhalt einer Variable durch Bearbeitung so abgeändert, dass der gewünschte Ausgabewert resultiert.

Datenorientiert Hier werden mehr oder weniger komplexe Daten bzw. Datenstrukturen bereitgestellt, die durch wenige und einfache Verarbeitungsschritte zum gewünschten Ziel führen.

Beispiel : Die Ausgabewerte für das Lauflicht werden in einer Tabelle bereitgestellt. Durch einfaches Erhöhen eines Zeigers können die Daten direkt gelesen und ausgegeben werden.

Vergleich

Ablauforientiert		*Datenorientiert*	
+	Meist weniger Speicher notwendig	+	Einfachere Programm-struktur
-	Umfassendere Test's not-wendig	+	Anpassungen erfordern nur kleine Programmänderung

Übung D

Ziel

Das Lauflicht mittels der indirekten Adressierung realisieren. Dabei sollen die Ausgabewerte aus einem Datenfeld stammen, das mittels eines Zeigers ausgelesen wird.

Pseudocode

```
Initialisiere Datenfeld; /* mit Schleife */
do_forever
  DataPtr = Adresse(Datenfeld);
  Cntr = 1;
  repeat
    P5 = ^DataPtr;
    Wait();
    Inc(DataPtr);
    Inc(Cntr);
  until Cntr = 8)
end;
```

Datenfeld

Es muss sich um einen Bereich von 8 zusammenhängenden Adressen im internen RAM[4] des 80535 handeln. Die Inhalte müssen im Initalisierungsteil des Programms festgelegt werden.

Sie können die Initialisierung in einer Schleife durchführen – Sie wissen ja nun, wie der Rotier-Befehl einzusetzen ist – oder die Werte einzeln als Konstanten zuweisen.

Zeiger

Der zu definierende Zeiger – verwenden Sie Register R0 – wird mit der Adresse der ersten Stelle im Datenfeld geladen, also z.B. mit dem Wert 20_h. Durch Inkrementieren (erhöhen) des Zeigers wird dann auf das nächste Feld verwiesen, also z.B. auf 21_h.

Vorgehen

1. Kopieren Sie die Datei LAUF3.ASM aus Ihrem Home-Verzeichnis als LAUF4.ASM ins Verzeichnis USERWORK!
2. Laden Sie die Datei in WORK21 und öffnen Sie die Datei.
3. Definieren Sie das benötigte Datenfeld und programmieren Sie die Initialisierung.
4. Erstellen Sie ein Programm, welches dem Pseudocode entspricht!
5. Wenn Sie Probleme mit dem Erhöhen des Zeigers haben, wenden Sie sich an Ihren Lehrer.
6. Kopieren Sie die gültige Fassung auf Ihr Home-Verzeichnis!

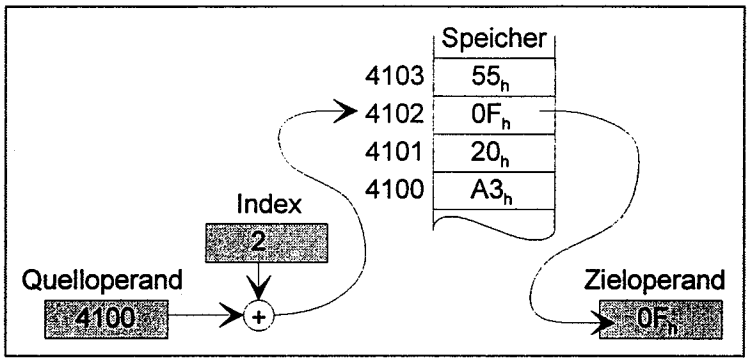
[4] vorteilhaft erst ab Adresse 20_h

INDIZIERTE ADRESSIERUNG

Array

Bei der indizierten Adressierung wird zum Inhalt des Quelloperanden ein Offsetwert addiert. Dieses Ergebnis dient dann als Adresse für den Zugriff auf den Zieloperanden.

Der Quelloperand wird nicht verändert.



Beispiel :

```
      :
CODE_TAB   DB   01H,02H,04H,08H ;Konstanten im ROM
      :
DS_INIT:   MOV DPTR,#CODE_TAB  ; Datenpointer mit...
                               ; ...Adresse laden.
      :
LOOP:   MOVC A,@A+DPTR      ; Wert aus Tabelle..
        MOV  P5,A           ; ...an Port ausgeben.
      :
```

Diese Methode dient bei der 8051er-Familie dem Lesen vom Programmspeicher und erleichtert somit den Zugriff auf Tabellen mit konstanten Werten. Es wird eine **Konstante** in eine **Variable** geschrieben.

Beachte

Der Akkumulator muss zuerst mit dem Offset für den Zugriff auf die Tabelle geladen werden. Dieser Wert wird beim MOVC-Befehl aber überschrieben! Wenn der Offset-Wert nachträglich wieder benötigt wird, muss er zwischenzeitlich gerettet werden. (➔ Stack)

Als Adressregister können der Datenpointer **DPTR** und der Programmzähler **PC** Verwendung finden.

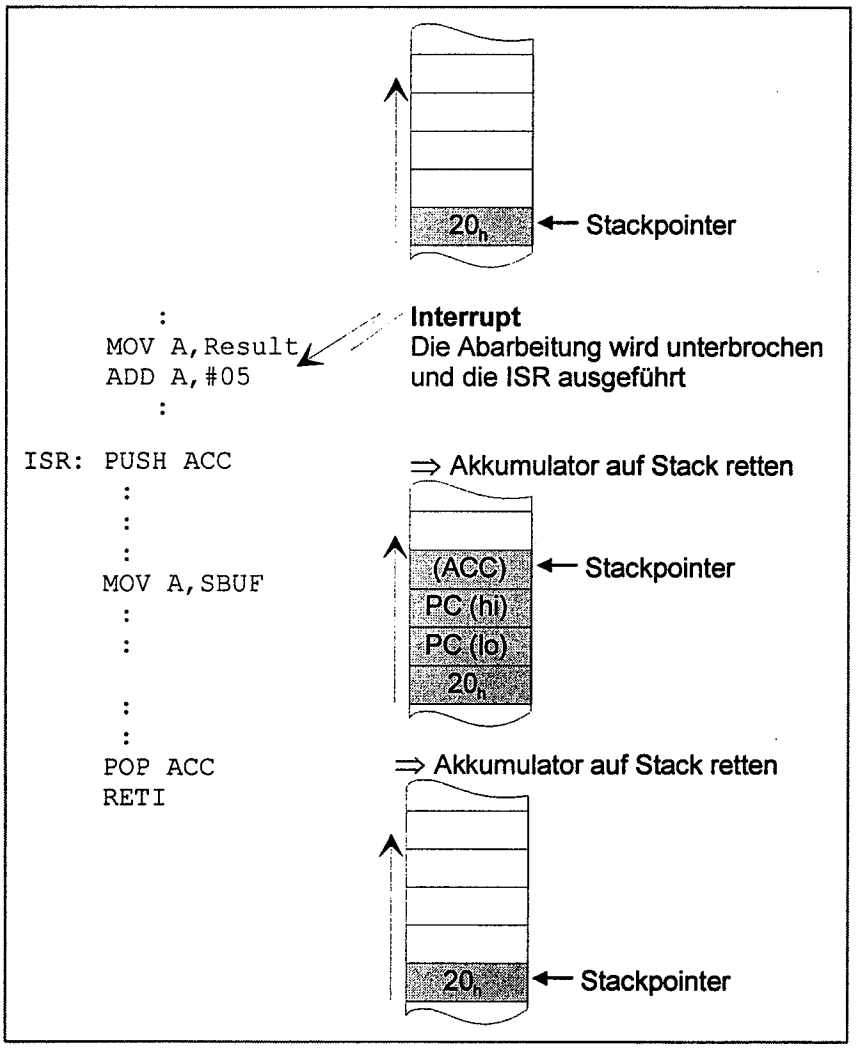
STACK ADRESSIERUNG

LIFO

Eine spezielle Möglichkeit, Werte zu speichern stellt der Stack dar. Dabei handelt es sich um einen reservierten Speicherbereich im internen RAM. Er verhält sich wie ein **Last-In-First-Out** Speicher, d.h. auf die Daten des Stack's kann nicht beliebig zugegriffen werden!

Einsatz

Der Stack dient vor allem dem Retten von Register-Inhalten, wenn z.B. eine Interrupt-Service-Routine[5] aufgerufen wird. Bei Hochsprachen wird die Parameterübergabe von Prozeduren und Funktionen oft auch über den Stack realisiert.



Die Zuweisung `MOV A,SBUF` innerhalb der ISR in obigem Beispiel zeigt keine negativen Folgen, da der mit `MOV A,RESULT` eingeschriebene Wert zu Beginn der ISR auf den Stack gerettet wurde. Beim Befehl `ADD A,#05` ist wieder der richtige Wert im Akkumulator.

[5] ISR

Beachte

Bei jedem Unterprogrammaufruf mit `CALL` und jedem Interrupt wird der Programmzähler (PC) explizit auf den Stack gerettet und mit `RET` bzw. `RETI` wieder zurückgelesen.

Die Adressen ab 08_h im internen RAM werden indirekt über den Stackpointer beschrieben, wenn ein Unterprogrammaufruf erfolgt bzw. ein Interrupt auftritt.
Eine DATA-Direktive sollte daher erst ab der Adresse 20_h Speicher belegen!

Unmittelbare Adressierung

Beispiel: Lauflicht

```
NAME    BLINK
        ORG  8000h
Start:  mov P5; #00h    ; Löscht alle Leds
        call wait       ; warte
        mov  P5; #01h   ; 1. LED
```

Befehl:	Port:	wert:
mov	P5	#00h

Direkte Adressierung

Befehl	Ziel	Quelle
MOV	P5,	OUTPUT
		Inhalt an Adresse "OUTPUT" an P5 ausgeben

Beispiel:

```
NAME    BLINK
        ORG  8000h

VARS    SEGMENT DATA    ; VARS: Segmentname
                        ; Speicherbereich für Daten
        RSEG VARS       ; RSEG: Reloktives Segment
OUTPUT: DS   1          ; reseriere 1 Byte im RAM
:
:
```

```
PROG    SEGMENT  CODE
        RSEG     PROG

Init:   mov  OUTPUT, #01h     ; initialisieren
Start:  mov  P5, OUTPUT       ; akt. Wert ausgeben
        call wait
        :
```

Übung B (Seite 2.4.1

Indirekte Adressierung

Befehl	Ziel	Quelle
MOV	P5,	@R0

R0: Index-Register lade Inhalt der Speicherstelle auf welche das Index-Register verweist

Dis und mit *1 ⇒ von bsp. oben.

```
        mov  R0, #OUTPUT ; Zeiger auf Variable
Start:  mov  P5, @R0     ; über Zeiger-Wert ausgeben
```

```
Akku    mov   A, @R0      ; holt in Akku
        rlc   A           ; verarbeiten
        mov   @R0, A      ; Akku über Zeige Speicher
        jmp   start       ; wiederhole
```

EXTERNE EREIGNISSE

Bis anhin haben Sie noch keinerlei Eingaben verarbeitet, somit also noch keine Datenverarbeitung im eigentlichen Sinn durchgeführt; Sie haben das EVA-Prinzip nicht umgesetzt.

Schalter

Wir werden nun über einen Schalter das Lauflicht stoppen bzw. laufen lassen.

POLLING

Abfrage

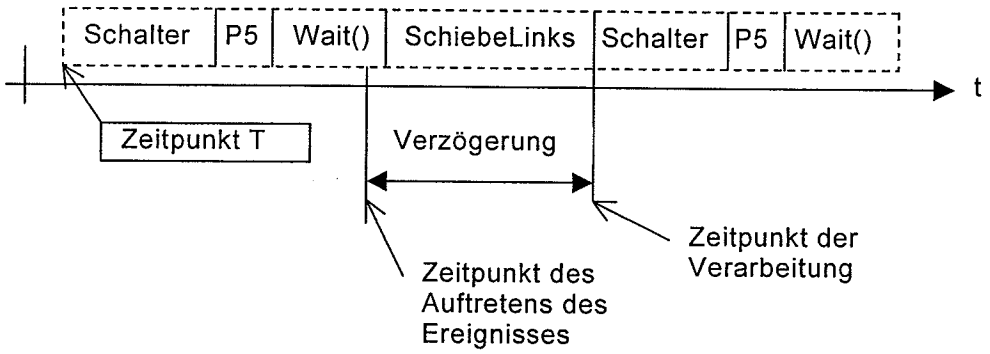
Die einfachste Methode ist das zyklische Abfragen des Schalters. Dabei wird Rechenzeit „verbraucht", da ja laufend der Zustand geprüft werden muss.

Pseudocode

```
do forever
  if Schalter = 0 then
    ; /* Busy-Wait bis Schalter auf 1 steht */
  P5 = WERT;
  Wait();
  SchiebeLinks(WERT); /* neuen Wert berechnen */
end;
```

Zeitraster

Das Ereignis „Schalter" wird nur zu einem bestimmten Zeitpunkt T erkannt, d.h. dass eine gewisse Verzögerung zwischen physikalischem Signal und logischer Verarbeitung erfolgt.



Übung E

Ziel

Das Lauflicht mittels der direkten Adressierung realisieren, wobei über einen Schalter ein go/no-go realisiert wird. Der Zustand des Schalters wird über Polling abgefragt.

Pseudocode

Schauen Sie auf der vorherigen Seite nach.

Tip

Lesen Sie den Schalter an Port P4 ein. Sie müssen dann aber das gewünschte Bit mittels Maskierung explizit auslesen.

Beispiel :

Es soll der Schalter Nummer 7 berücksichtigt werden. Alle andern Schalter sollen ohne Einfluss auf das Programm sein.

Bit-Nr.	7	6	5	4	3	2	1	0
Eingabe an Port	1	0	1	0	0	1	0	0
Maskierung	1	0	0	0	0	0	0	0
AND-Verknüpfung	1	0	0	0	0	0	0	0

Durch die AND-Operation wird nur noch das 7. Bit berücksichtigt. Ist dieses auf logisch 1, so wird auch das Ergebnis eine 1 liefern. Ist es aber 0, wird auch das Ergebnis auf 0 lauten, egal welchen Wert die andern 7 Stellen haben.

Vorgehen

1. Kopieren Sie die Datei LAUF2.ASM aus Ihrem Home-Verzeichnis als POLL.ASM ins Verzeichnis USERWORK!
2. Laden Sie die Datei in WORK21 und öffnen Sie die Datei.
3. Ändern Sie das bestehende Programm so ab, dass Sie den Schalter einlesen und die Ausführung des Lauflichtes nur bedingt durchführen
4. Kopieren Sie die gültige Fassung auf Ihr Home-Verzeichnis!

INTERRUPT

Allgemeines

Um auf die Möglichkeiten der Interrupts zurückgreifen zu können, sind einige Punkte zu beachten:

- Reaktion auf interne oder externe Ereignisse
- sofortiger Unterbruch des ordentlichen Programmablaufs
- keine CPU-Zeit für Polling von Flags bzw. Ports
- aufrufen eines Interrupt-Vektors, je nach Ereignis
- meist Sprung zu einer ISR für die Abarbeitung einer spezifischen Aufgabe
- verschiedene Prioritätsebenen für Interrupts
- Interrupts können selber auch unterbrochen werden (nur bei höherer Priorität)
- Interrupt signalisiert sein Erscheinen durch Request-Flag
- Interrupt muss explizit freigegeben werden

Ablauf

Beim Auftreten eines Interrupts muss die CPU einige Arbeiten implizit (ohne Dazutun des Programmierers) ausführen. Je nach Produkt kann dies Abfolge unterschiedlich ausfallen. Daher werden hier nur die wichtigsten Schritte aufgelistet.

Voraussetzung
Der entsprechende Interrupt ist Initailisiert und „scharf" gemacht!

1. Ein Ereignis tritt ein.
2. Das entsprechende Request-Flag wird gesetzt.
3. Die CPU prüft, ob der Interrupt freigegeben ist.
4. Wenn der Interrupt zugelassen ist, erfolgt ein Sprung zum Interruptvektor; sofern nicht bereits ein Interrupt mit höherer Priorität ausgeführt wird.
5. Wichtige Register wie z.B. der Programmzähler (PC) wird auf den Stack gerettet.
6. Der Interruptvektor wird gelesen und zur entsprechenden ISR[1] verzweigt.
7. Die ISR wird ausgeführt
8. Das Request-Flag wird
   · durch die Hardware selber wieder gelöscht.
   · muss durch die Software explizit gelöscht werden.
9. Alle auf den Stack geretteten Werte werden wieder installiert.

[1] ISR = Interrupt Service Routine

INTERRUPTBEHANDLUNG BEIM 80535

Wird ein Interrupt erkannt, generiert das Interruptsystem einen Aufruf des entsprechenden Interrupt-Vektors, vorausgesetzt, dass keine der folgenden Bedingungen zutrifft:

- der Interrupt ist nicht freigegeben
- ein Interrupt gleicher oder höherer Priorität ist aktiv
- der gerade laufende Befehl ist noch nicht abgearbeitet
- der gerade aktive Befehl ist ein RETI
- es findet ein Zugriff auf das IE[2]- bzw. IP[3]-Register statt

Freigabe

Damit ein ganz bestimmter Interrupt ausgeführt werden kann, muss er explizit freigegeben werde.

Beachte
Ein Interrupt wird nicht generell bearbeitet!

Damit ein Interrupt bearbeitet werden kann

- müssen Interrupts generell zugelassen werden
- muss das spezifische Freigabe-Flag gesetzt sein
- muss das Request-Flag gelöscht und das Enable-Bit gesetzt sein
- darf kein Interrupt mit höherer Priorität ausgeführt werden

Generelle Freigabe

Bei der 8051er-Familie muss das **EA-Bit** auf den logischen Wert 1 (wahr) gesetzt sein, damit Interrupts überhaupt zugelassen sind.

Beachte
Nach dem Power-On ist EA=0!
Sollen Interrupts verwendet werden, muss das EA-Bit gesetzt werden.

Spezifische Interruptquellen

Der 80535 verfügt über eine Vielzahl von Quellen. In der folgenden Liste sind alle relevanten Informationen zusammengestellt.

Quelle	Aktion	Requ. Flag	Ena. Bit	Prio. Bit	Res.	Vektor
Ext. Int. 0	neg. Flanke/Pegel an P3.2	IE0	EX0	(PX0)	H	0003_h
Timer 0	Überlauf des Zählers	TF0	ET0	(PT0)	H	$000B_h$
Ext. Int. 1	neg. Flanke/Pegel an P3.3	IE1	EX1	(PX1)	H	0013_h

[2] IE : Interrupt-Enable-Register (Freigabe der Interrupts)

[3] IP : Interrupt-Prioritäts-Register

Quelle	Aktion	Requ. Flag	Ena. Bit	Prio. Bit	Res.	Vektor
Timer 1	Überlauf des Zählers	TF1	ET1	(PT1)	H	$001B_h$
Serielle Schnittstelle	Ende des Sende-Empfangsvorganges	TI RI	ES	(PS)	S	0023_h
Timer 2	Überlauf des Zählers ext. Reload an P1.5	TF2 EXF2	ET2 EXEN2	PT2	S	$002B_h$
A/D-Wandler	Ende der Wandlung	IADC	EADC	IP0 IP1	S	0043_h
Ext. Int. 2	neg./pos. Flanke an P1.4	IEX2	EX2	IP0 IP1	H	$004B_h$
Ext. Int. 3 Capture 0 Compare 0	neg./pos. Flanke an P1.0	IEX3	EX3	IP0 IP1	H	0053_h
Ext. Int. 4 Capture 1 Compare 1	neg./pos. Flanke an P1.1	IEX4	EX4	IP0 IP1	H	$005B_h$
Ext. Int. 5 Capture 2 Compare 2	pos. Flanke an P1.2	IEX5	EX5	IP0 IP1	H	0063_h
Ext. Int. 6 Capture 3 Compare 3	pos. Flanke an P1.3	IEX6	EX6	IP0 IP1	H	$006B_h$

Reset

H steht für Hardware-Reset, d.h. dass das Request-Falg durch die CPU selbser wieder zurückgesetzt wird.
S steht für Software-Reset, d.h. dass der Programmierer selber für das Rücksetzen des Request-Flag verantwortlich ist.

Beachte

Die genaue Beschreibung der einzelnen Interrupts ist entsprechenden Dokumenten zu entnehmen, da dies den Umfang dieses Skript übersteigen würde.

INTERRUPT-HANDLER

Grundstruktur

Neben den Kenntnissen der Interruptstruktur eines Prozessors ist es auch wichtig, den grundlegenden Aufbau eines Interrupthandlers sowie die notwendigen Initialisierungen zu kennen.

Beispiel : Interrupt-Behandlung mit dem 80535
Hardware-Rücksetzung des Request-Flag

```
;/* Initialisierung des Systems, hier sind keine Interrupts
;   zugelassen, da bei einer Unterbrechung gewisse Werte
;   noch nicht richtig initialisiert sind!                    */
;/* Initalisierung des Interruptsystems
;   spezifische Interrupts freigeben                          */
;/* Beispiel für ext. Interrupt 0 */
SETB  EX0
;/* generelle Freigabe aller Interrupts */
SETB EA
MAIN_LOOP:
    ;/* das eigentliche Programm */
    JMP Main_LOOP
;/*************************************************************/
;/* Interrupthandler mit Hardware-Reset des Request-Flag      */
;/* Beispiel für ext. Interrupt 0 */
ISR_EXT0:
;/* alle relevanten Register auf Stack retten
;   (PC wird explizit gerettet!)                              */
PUSH ACC
PUSH PSW
   :
;/* ausführen der ISR */
;/* die geretteten Register in umgekehrter Reihenfolge
;   wieder installieren                                       */
  :
POP  PSW
POP  ACC
;/* Rücksprung aus Interrupt
;   diesen wieder "scharf" machen                             */
SETB  EX0
RETI
```

Beispiel : Interrupt-Behandlung mit dem 80535
Software-Rücksetzung des Request-Flag

```
;/* Initialisierung des Systems, hier sind keine Interrupts
;   zugelassen, da bei einer Unterbrechung gewisse Werte
;   noch nicht richtig initialisiert sind!                  */
;/* Initalisierung des Interruptsystems
;   spezifische Interrupts freigeben                        */
;/* Beispiel für serielle Schnittstelle */
SETB  ES
;/* generelle Freigabe aller Interrupts */
SETB EA
MAIN_LOOP:
    ;/* das eigentliche Programm */
    JMP Main_LOOP
;/*************************************************************/
;/* Interrupthandler mit Software-Reset des Request-Flag  */
;/* Beispiel für serielle Schnittstelle */
ISR_SERIELL:
;/* alle relevanten Register auf Stack retten
;   (PC wird explizit gerettet!)                            */
PUSH ACC
PUSH PSW
   :
;/* Interrupt-Quelle festlegen */
;/* if TI-Bit gesetzt */
JNB TI,Receive
Tranmit:
;/* then Daten senden */
;/* in Sendebuffer schreiben */
;/* das Request-Flag wieder löschen */
CLR  TI
JMP  ISR_Ende
Receive:
;/* else Daten empfangen */
;/* Empfangsbuffer auslesen und Wert speichern */
;/* das Request-Flag wieder löschen */
CLR  RI
ISR_Ende:
;/* die geretteten Register in umgekehrter Reihenfolge
;   wieder installieren                                     */
  :
POP  PSW
POP  ACC
;/* Rücksprung aus Interrupt
;   diesen wieder "scharf" machen                           */
SETB  ES
RETI
```

Vektortabelle

Das Erzeugen einer Vektortabelle in Assembler besteht darin, festplatzierte Einsprungspunkte vorzugeben.

```
;Vektortabelle für EXT0, TIMER1, SERIEL
CSEG  AT 0000h          ; der Reset-Vektor
      LJMP  MAIN
CSEG  AT 0003h          ; Ext. 0 Interrupt an Adresse 03H
      LJMP  EXT0_ISR
CSEG  AT 001Bh          ; Timer 1 Interrupt
      LJMP  TIMER0_ISR
CSEG  AT 0023h          ; serielle Schnittstelle
      LJMP  SERIEL_ISR
            ;
      RSEG MAIN         ;  automatisch platzieren
MAIN: ......
```

Interrupthandler

Der Interrupthandler kann an beliebiger Stelle im Codespeicher stehen. Wichtig ist, dass der Sprungbefehl die Adresse der ISR erreicht; daher ist es von Vorteil, wenn ein "langer Sprung" (LJMP) eingesetzt wird, der über den ganzen Adressbereich geht.

```
            RSEG  ISR   ; automatisch Platzieren
????_ISR:   PUSH  ACC
            PUSH  .....

            ..........  ; Aktion ausführen

            POP   .....
            POP   ACC
            SETB  ...   ; Interrrupt freigeben
            RETI
```

Übung F

Ziel

Das Lauflicht mittels der direkten Adressierung realisieren, wobei über einen Schalter ein go/no-go realisiert wird. Der Zustand des Schalters wird über eine Interrupt-Routine geprüft.

Pseudocode

```
main()
IT0 = 1; EX0 = 1; EA = 1; /* Initialisieren */
ISR_Flag = 0; Wert = 0x01;
do forever
  P5 = WERT;
  Wait();
  SchiebeLinks(WERT); /* neuen Wert berechnen */
end;

ISR_Extern0()
  if ISR_Flag = 1 then
    ISR_Flag = 0;
    TempWert = Wert;
    Wert = 0;
    P5 = Wert;    /* sofortige Reaktion */
  else
    ISR_Flag = 1;
    Wert = TempWert;
  EX0 = 1;
```

Variable

Wie Sie dem Pseudocode entnehmen können, benötigen Sie 3 Variablen. Plazieren Sie die Variablen selbst im Datensegment; es kann sonst Probleme mit dem Stack geben! Verwenden Sie dazu `DSEG AT 20h`.

Aus

Wenn das Lauflicht stillstehen soll, wird hier sein Wert auf 0 gesetzt, so dass alle LED ausgeschaltet werden. Dadurch lässt sich das Verhalten der Software besser beobachten!

Zudem wird der Wert von P5 sofort auf 0 gesetzt. Sie können dadurch erkennen, dass zwischen dem Auftreten des Ereignis und dessen Bearbeitung keine Zeit verstreicht.

Ein

Hier wird der originale Wert wieder in die „Ausgabe-Variable" `Wert` zurückgeschrieben. Das Port wird aber nicht sofort beschrieben, so dass also auch hier eine gewisse Zeit zwischen dem Ereignis und dessen Auswirkung verstreicht.

TIP

Erhöhen Sie die Wartezeit in der `WAIT`-Prozedure, so dass Sie die Reaktion bei Ein bzw. Aus besser beobachten können!

Pegel/Flanke

Der externe Interrupt 0 kann sowohl auf eine fallende Flanke als auch auf einen 0 Volt-Pegel reagieren. Dies muss der Programmierer über das IT0-Bit einstellen. Setzen Sie dieses Bit auf 1 (`setb IT0`), damit das Programm auf die Flanke reagiert!

Anschluss

Beachten Sie ,dass die LED-Versuchsplatine an Interrupt-Port des Prozessor-Boards angeschlossen werden muss!
Vergl. Sie dazu die Abbildung in Kapitel 1.7

Vorgehen

1. Kopieren Sie die Datei LAUF2.ASM aus Ihrem Home-Verzeichnis als EXTINTR.ASM ins Verzeichnis USERWORK!
2. Laden Sie die Datei in WORK21 und öffnen Sie die Datei.
3. Ändern Sie das bestehende Programm so ab, dass Sie den Schalter einlesen und die Ausführung des Lauflichtes nur bedingt durchführen.
4. Kopieren Sie die gültige Fassung auf Ihr Home-Verzeichnis!

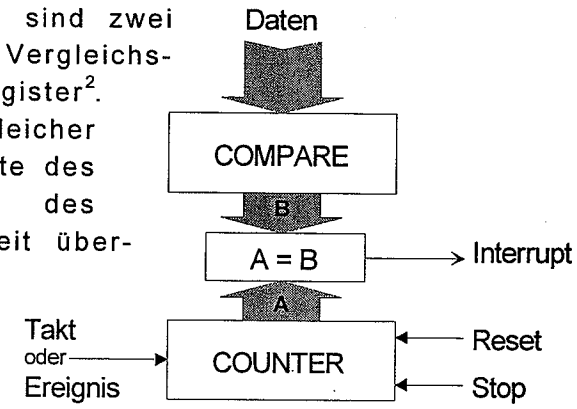
INTERNE EREIGNISSE

TIMER-INTERRUPT

Neben – externen – Signalen die zu einem beliebigen Zeitpunkt auftreten, muss ein Computersystem auch zeitgesteuerte Vorgänge abbilden können. Dazu werden Timer verwendet, bei deren Ablauf ein entsprechender Interrupt generiert wird.

Prinzip

Bei einem Timer-Baustein sind zwei Register vorhanden; das Vergleichs-Register[1] und das Zählregister[2]. Zusätzlich wird ein Vergleicher benötigt, welcher die Werte des Vergleichs-Registers und des Zählregisters auf Gleichheit überprüft.



Im Vergleichs-Register wird durch den Programmierer ein Wert eingeschrieben, der als Schwelle dient.

Das Zählregister wird durch einen Takt hochgezählt, bis sein Inhalt dem des Vergleichs-Registers entspricht. Sobald dies der Fall ist, wird ein Interrupt ausgelöst.

Funktionsvielfalt

Timer können für verschiedenste Funktionen benötigt werden. Im folgenden sollen einige mögliche Einsatzgebiete kurz umrissen werden

Zähler

Bei den meisten Timer-Bausteinen lässt sich ein Zählermodus einstellen. Dabei werden – in diesem Fall externe – Ereignisse gezählt und beim Erreichen einer vordefinierten Schwelle wird der Interrupt generiert.

Timer

Bei dieser Funktionsart wird dem Baustein ein Taktsignal – meist aus dem Systemtakt abgeleitet – zugeführt. Der Zähler zählt solange hoch, bis der Schwellwert erreicht ist, oder ein Stop-Signal an den Zähler angelegt wird.

Reload

Hier handelt es sich um einen speziellen Zähler, bei dem mit dem Auslösen des Interrupt – die Schwelle ist erreicht – auch ein automatischer Reset des Zählers erfolgt. Somit lässt sich eine feste Zeitbasis bilden, z.B. für den Tick eines preemptiven OS.

Baudrate

Timer können auch die Zeitbasis für eine serielle Übertragung liefern.

---

[1] Copmpare-Register

[2] Counter

Übung G

Ziel

Das Lauflicht mittels der direkten Adressierung realisieren, wobei über einen Schalter ein go/no-go realisiert wird. Der Zustand des Schalters wird über eine Interrupt-Routine geprüft.

Zudem erfolgt die zeitliche Steuerung des Lauflichtes nicht mehr über Busy-Wait (mit `call Wait` realisiert), sondern durch Zeitsteuerung über einen Timer.

Pseudocode

```
main()
T2CON = 0X91; ET2 = 1;   /* Timer 2 für Interrupt */
CRCL = 0; CRCH = 0h;     /* Zähler für Timer 2 */
IT0 = 1; EX0 = 1;        /* Intr 0 auf Flanke */
EA = 1;                  /* Interrupts freigeben */
/* Initialisieren der Variablen */
ISR_Flag = 0; Wert = 0x01;
do forever
  ; /* keine Tätigkeit im Hauptprogramm */
end;

ISR Timer2();
  P5 = Wert;
  SchiebeLinks(WERT); /*neuen Wert berechnen */
  TF2 = 0;
  ET2 = 1;

ISR_Extern0();
  if ISR_Flag = 1 then
    ISR_Flag = 0
    TempWert = Wert;
    Wert = 0;
  else
    ISR_Flag = 1;
    Wert = TempWert;
  EX0 = 1;
```

Initialisierung

Der Timer 2 des 80535 kann viele verschiedene Funktionen übernehmen. Für die Lösung unseres Problems benötigen wir einen Reload-Timer. Die Einstellung dazu wird im Register `T2CON` (Wert 91_h) vorgenommen.

Zählerwert

Der 16-Bit Zählerwert steht in den Registern `CRCL2` und `CRCH2`. Bei einer Verarbeitungsfrequenz von 1MHz (typisch für 80535) ergibt sich somit eine maximale Zeitbasis von 65535*1µS = 0.065535 S, was einer Frequenz von 15.25 Hz entspricht.

Über einen Vorteiler kann dem Zähler eine Frequenz von 500 kHz zugeführt werden, so dass sich eine Frequenz von 7.625 Hz. Ergibt.

Beachte
Der 80535 zählt bis 65535 und löst dann einen Interrupt aus!

Zählregister

Das Zählregister besteht aus den zwei 8-Bit Registern `CRCL` und `CRCH`. Für eine maximale Zeitdauer müssen beide auf 0 initialisiert werden.

Akku

Je nach Lösung werden Sie den Akku sowohl beim Timer-Interrupt als auch beim externen Interrupt benötigen. Achten Sie darauf, dass Sie in diesem Fall den Akku mittels `PUSH` und `POP` auf dem Stack retten. (Vrgl. dazu 2.4.9)

Vorgehen

1. Kopieren Sie die Datei LAUF2.ASM aus Ihrem Home-Verzeichnis als TIMINTR.ASM ins Verzeichnis USERWORK!
2. Laden Sie die Datei in WORK21 und öffnen Sie die Datei.
3. Ändern Sie das bestehende Programm so ab, dass Sie den Schalter einlesen und die Ausführung des Lauflichtes nur bedingt durchführen.
4. Kopieren Sie die gültige Fassung auf Ihr Home-Verzeichnis!

```
$NOLIST
$INCLUDE (C:\PROGRA~1\WORK21\REG535N.INC)
$LIST

NAME      BLINK

          ORG 8000h

          DSEG AT 20h
OUTPUT:   DS    1
ISR_FLAG: DS    1
TMP_VAL:  DS    1
;=========================================================
          CSEG AT 8000h
          ljmp Init        ; Programmeinsprung
          CSEG AT 8003h
          ljmp EXT0_ISR    ; Service-Routine für Ext. Int. 0
;=========================================================
PROG      SEGMENT CODE
ISR       SEGMENT CODE

          RSEG   PROG
Init:     setb  IT0          ; auf negative Flanke reagieren
          setb  EX0          ; ext. Interrupt 0 zulassen
          setb  EA           ; alle Interrupt freigeben
                             ;
          mov   ISR_FLAG,#0  ; initalisieren der Variable
          mov   OUTPUT,#01h  ;
Start:    mov   P5,OUTPUT    ; aktuellen Wert ausgeben...
          lcall Wait         ;
          mov   A,OUTPUT     ; ...und neuen Wert rechnen
          rlc   A            ;
          mov   OUTPUT,A     ;
          jmp   Start        ; wiederhole

;=========================================================
          RSEG  ISR
EXT0_ISR: push  ACC              ; Akkumulator wird benötigt...
          mov   A,ISR_FLAG       ; ...für Test auf Zero
          jz    SetFlag          ; if ISR_FLAG = 1 then
ClrFlag:  mov   ISR_FLAG,#0      ;   lösche Flag
          mov   TMP_VAL,OUTPUT   ;   TempVal = Output
          mov   OUTPUT,#0        ;   Output = 0
          mov   P5,OUTPUT        ;   Wert an Port sofort ausgeben
          jmp   EndIf            ; else

SetFlag:  mov   ISR_FLAG,#1      ;   setze Flag
          mov   OUTPUT,TMP_VAL   ;   Output = TempVal -> erst im...
                                 ;   ...Zyklus aktiviert!
EndIf:    setb  EX0              ; Interrupt wieder aktivieren
          pop   ACC              ; Akku installieren
          reti
;=========================================================

Wait:     mov   R5,#0FFh     ; lade 1.Zaehler
Wait1:    mov   R6,#0FFh     ; repeat lade 2.Zaehler
Wait2:    mov   R7,#04h      ;   repeat lade 3.Zahler
Wait3:    djnz  R7,Wait3     ;     repeat, Dec R7, until R7 = 0
          djnz  R6,Wait2     ;   until R6 = 0, Dec R6
          djnz  R5,Wait1     ; until R5 = 0, Dec R5
          ret



          SWAP A         ; End of File fuer Monitor CPU1
          SWAP A
          SWAP A
          SWAP A
          SWAP A

          END
```

C51-SPRACHE

ANSI-C

Bei C51 handelt es sich um einen ANSI-C kompatiblen Compiler der aber über spezielle Erweiterungen verfügt, die auf den 8051-Prozessor zugeschnitten sind!

8051 SPEICHERSEGMENTE

Der 8051 verfügt über diverse physikalische Speicherbereiche (RAM/ROM), die bei der Deklaration von Variablen und typisierten Konstanten angegeben werden müssen.

Segmentbezeichnungen

code Programmspeicher (64 kByte) für typisierte Konstanten
Zugriff über Opcode `MOVC @A+DPTR`

data Direkt adressierbarer interner Speicher (128 Bytes, Adressen 0...127)
Schnellster Zugriff auf Daten

idata Indirekt adressierbarer interner Speicher (256 Bytes, Adressen 0...255)
Zugriff auf ganzen internen Speicher

bdata Bitadressierbarer Bereich des internen Speichers
Ermöglicht Zugriff auf Bit und Byte Ebene

xdata Externer Speicher (64 kBytes) für Variablen
Zugriff über Opcode `MOVX @DPTR`

Beachte
Bei Verwendung der CPU1 muss xdata hinter die Code-Adressen gelegt werden!

pdata Seitenadressierter externer Speicher (256 Bytes, Adressen 0...255)
Zugriff über Opcode `MOVX @Rn`

→ diese Segmente entsprechen (bis auf pdata) den Direktiven des A51-Assembler

Deklaration

Die Segmentbezeichnung muss dem Datentyp vorangestellt werden.

Beispiel :

```
code unsigned char DispNr[4]={0x01,0x02,0x04,0x08};
data int Resultat;
xdata unsigned int Zahlen_Liste[150];
```

ERWEITERTE DATENTYPEN

Neben den in ANSI-C bekannten Datentypen[1] verfügt C51 über spezielle Datentypen für die Bitadressierung.

Bit Typen

bit Reserviert ein Bit im internen Bitspeicher, darf aber auch als Rückgabetyp einer Funktion verwendet werden.

sfr Deklaration von Registern des 8051wie z.B. `P0`, `TCON` usw.

sfr16 Deklaration von Registern neuerer 8051-Derivate, die 16 Bit Werte enthalten wie z.B. `CRCL/CRCH` für Timer 2

sbit Deklaration einzelner Bit's in den Registern des 8051

Beachte
Die Register sind in der Datei REG515.INC bereits deklariert!
Die Datei wird über die `#include`-Anweisung eingebunden.

Deklaration

Beispiel :

```
bit  flag1, flag2;
sbit Speaker = P1^0; /* Bit 0 von Port 1 */
```

SPEZIELLE ANWEISUNGEN

Interrupt

Um die Interruptmöglichkeiten des 8051 direkt aus C zu nutzen, kann bei der Deklaration einer Prozedure mitgeteilt werden, welchem Interrupt sie zugeordnet ist.

Beispiel :

```
void timer2_isr (void) interrupt 5
  {
    :
    :
  } /* Timer2 ISR */
```

Die Interruptvektoren werden durch den C-Compiler erzeugt, so dass der Programmierer (normalerweise) keinerlei Anpassungen in Assembler-Code vornehemen muss.

bei CPU1! Für die CPU1 müssen die Interrupts auf die Adresse 8000_h umgelenkt werden, wozu das Modul INTVECT.A51 verwendet werden muss! Werden Interrupts benötigt, müssen

- die entsprechenden `EXTERN`-Anweisungen aktiviert werden
- die entsprechenden Sprung-Anweisungen (`LJMP ....`) aktiviert und der `RETI`-Befehl gelöscht werden

[1] `char, int, float` usw.

Wenn die Datei INTVECT.A51 abgeändert worden ist, empfiehlt es sich, das ganze Projekt neu zu übersetzen!

Registerbank

Der 8051 bietet für ein einfaches "Multitasking" die 4 Registerbänke an, in welchen die für den jeweiligen Prozess relevanten Daten (PC, DPTR, ACC usw.) zwischengespeichert werden können.

Bei der Deklaration einer Prozedure kann bekannt gemacht werden, welche Registerbank eingesetzt werden soll.

Beispiel :

```
void myfunction (void) using 2
  {
     :
     :
  } /* Prozedure benutzte Registerbank 2 */
```

STRUKTUR

```
#pragma CODE DEBUG SYMBOLS
#pragma OPTIMIZE(3)
#include <reg515.h>
/* */
/* Programmglobale Konstante und Variable */
#define  TRUE   1
#define  FALSE  0
#define  k_MAX_LOOP 8

data unsigned char running=TRUE;
code unsigned char VAL[8]={0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80};
data unsigned char CNT = 0;

ext0_isr () interrupt 0
     { if (running == TRUE) running = FALSE;
       else running = TRUE;
       EX0 = 1; /* Interrupt scharf machen */
     } /*END ext0_isr*/

timer2_isr () interrupt 5
     { TF2 = 0; /* Interruptrequest */
       if (running == TRUE) {
          P5 = VAL[CNT++];
          if (CNT == k_MAX_LOOP) CNT = 0;
       } /* end if */
       ET2 = 1; /* Interrupt scharf machen */
     } /* END timer2_isr */

/* Lauflicht */
void main(void)
{ /* Interrupts initialisieren */
  T2CON = 0x91;  /* autoreload mit kleinster Frequenz */
  CRCL  = 0;
  CRCH  = 0;     /* grösste Reloadzeit einstellen */
  ET2 = 1;
  IT0 = 1;       /* ext0 auf Flanke */
  EX0 = 1;       /* Interrupt freigeben */
  EAL = 1;       /* alle Interrupt zulassen */

  while (TRUE) {
  } /* end while */
} /* end of main */
```

C51-PROJKETE

ACHTUNG
Die beschriebene Methode darf nur von Benutzern verwendet werden, die über ein eigenes Laufwerk H: verfügen!
→ geht nicht für Klassenkonti!

Module Ein ausführbares C51-Programm besteht immer aus mehreren Modulen, die mittels Assembler oder Compiler in einen Objekt-Code – dieser ist noch nicht auf dem Zielsystem ausführbar – übersetzt wird.

Linker Erst durch den Linker werden alle Module miteinander verbunden, so dass die Adressen von Sprunganweisungen aber auch Variablen korrekt zugeordnet werden können.

Projekt Damit nicht jedesmal alle Module einzeln übersetzt und gebunden werden müssen, erstellt man Projekte. Diese beinhalten alle relevanten Angaben, um ein ausführbares Programm zu erhalten.

DIE NOTWENDIGEN MODULE

Startpunkt Jedes C-Programm beginnt mit der Funktion `main`. Daher muss ein Modul (MONSTART.A51) verfügbar sein, das den entsprechenden Sprung zur `main`-Funktion ausführt.

Interrupt-vektoren Da die Interruptvektoren beim vorliegenden System auf die Adressen ab 8000_h gespiegelt sind, muss ein entsprechendes Modul (INTVECT.A51) verfügbar sein, welches die Aufruf der einzelnen Service-Routinen behandelt.

Datei-Ende Das Ende einer Binärdatei wird dadurch erkannt, dass 5 mal das Bitmuster 1010'0100 = $C4_h$ = `SWAP A` auftritt; eine etwas eigenartige Methode, aber sie muss realisiert werden! Es muss daher ein Modul (EOF.A51) verfügbar sein, das die entsprechenden Assembler-Befehle beinhaltet.

C-Programm Und dann soll ja auch noch ein Modul vorhanden sein, welches das eigentliche C-Programm beinhaltet.

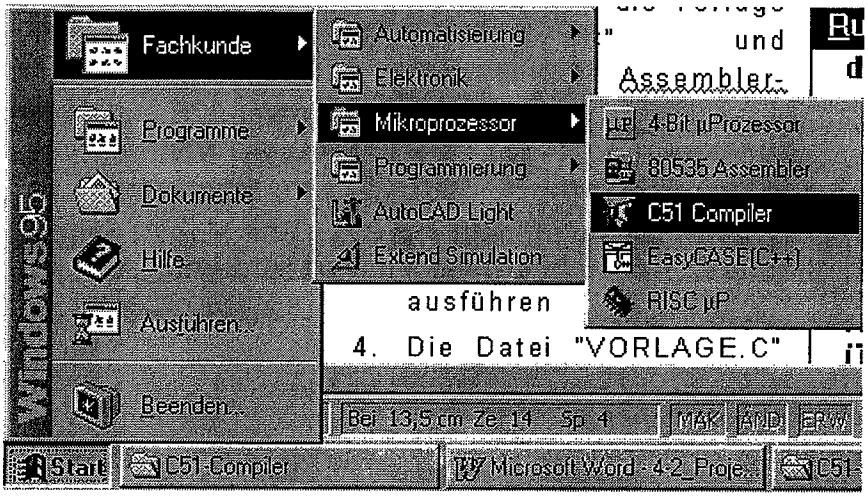
EINRICHTEN EINES PROJEKTES

Arbeitsverzeichnis auf H:

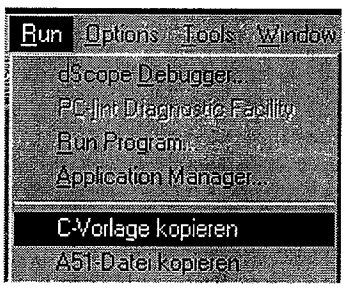
1. Erstellen Sie in Ihrem HOME-Verzeichnis ein Unterverzeichnis **C51** für die Arbeiten mit C51 und wechseln Sie in dieses Verzeichnis. (➔ H:\C51 ist das Ziel aller Laufwerkszugriffe)

Dateien kopieren und laden

2. Starten Sie über den Start-Knopf von Windows 95 den C-51 Compiler.



3. Kopieren Sie die Vorlage „VORLAGE.C“ und wichtige Assembler-Dateien (*.A51) ins Verzeichnis H: (das momentane Unterverzeichnis !). Dazu im Menü "Run" die entsprechenden Befehle ausführen



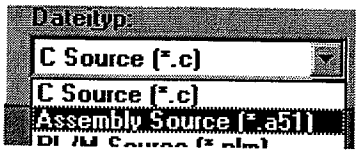
4. Die Datei "VORLAGE.C" umbenennen. Wählen Sie dazu einen **sinnvollen Namen**!
   ➔ Sie müssen in den Dateimanager wechseln!
5. Die C-Quelldatei über "File/Open..."in Editor laden; das Laufwerk H: auswählen!

Beachte

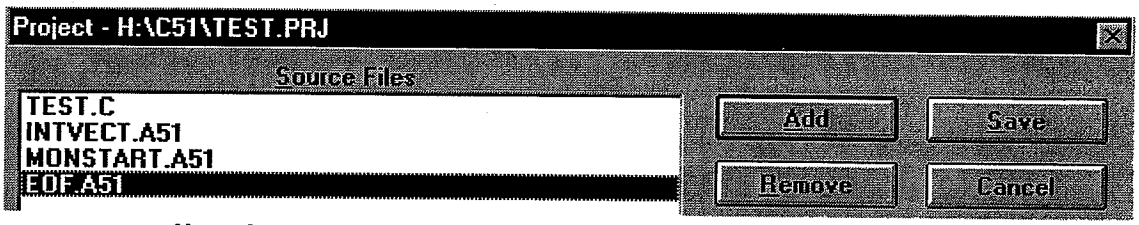
Die Datei VORLAGE.C ist soweit vorbereitet, dass Sie nur noch die Funktionen und das Hauptprogramm (`main`) Ihrer Problemlösung erstellen müssen.

Projekt definieren

6. Mit Menü "Project/New Project..." ein neues Projekt definieren. Verwenden Sie den **gleichen Namen** wie für die C-Source!
7. Die C-Source-Datei über Button "Add" in Projekt einfügen.
8. Die A51-Dateien "INTVECT.A51", "MONSTART.A51" und "EOF.A51" ebenfalls dazu fügen. Dazu unter "Dateityp:" den Assembler auswählen!



ACHTUNG : Reihenfolge unbedingt einhalten!



Ihr eigenes Programm muss nicht zwingen TEST.C heissen!

9. Das Projekt über Befehl "Save" sichern.

Hinweis

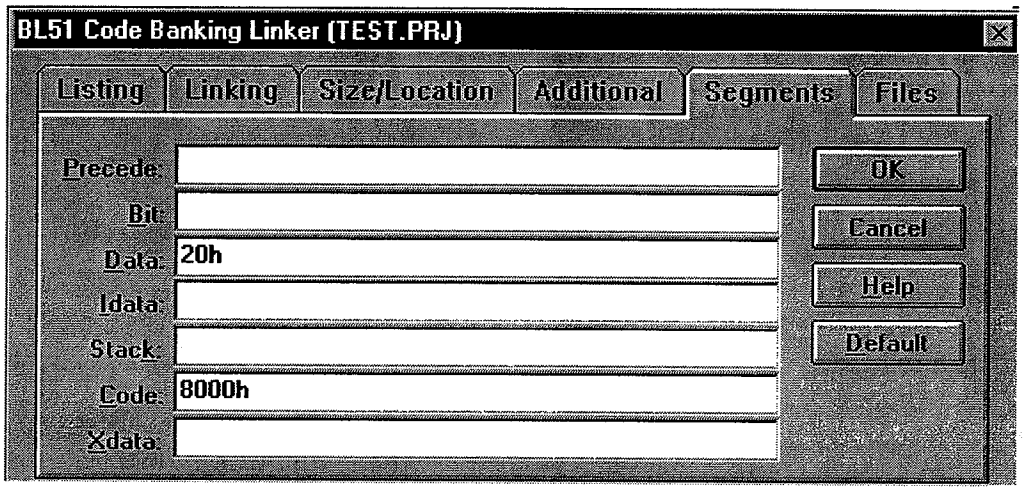
Wenn Sie ein Projekt unterbrechen – d.h. Sie bearbeiten ein bestehendes Programm zu einem späteren Zeitpunkt – müssen Sie immer zuerst das Projekt laden und danach die zu bearbeitende C oder A51 Datei!

LINKER EINSTELLUNGEN

Lokalisierung Damit die einzelnen Segmente richtig plaziert werden, müssen folgende Einstellungen vorgenommen werden. Dazu wählen Sie im Menü "Options" die Funktion "BL51 Code Banking Linker ..." aus und fügen die folgenden Werte ein.

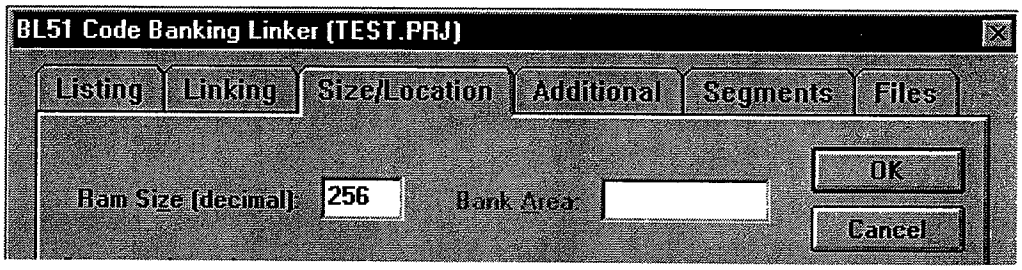
Speicheradressen

10. Die internen Daten werden mit Vorteil erst ab Adresse **20h** zugewiesen, da die ersten 32 Byte des Speichers für die 4 Registerbänke benötigt werden könnten.
11. Der Code soll ab Adresse **8000h** plaziert werden. Dazu im Menü "Options/BL51 Code Banking Linker..." im Griffregister "Segments" die folgenden Werte eingeben:



Speichergrösse

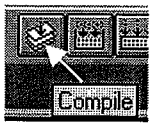
12. Der 80535 verfügt über 256 Byte internes RAM. Den entsprechenden Wert im Griffregister "Size/Location" eintragen.



13. Die Angaben bestätigen (OK-Button)

C-CODE ERSTELLEN

14. Den Quellcode bearbeiten.
15. Den Quellcode übersetzen lassen und allfällige Fehler korrigieren.

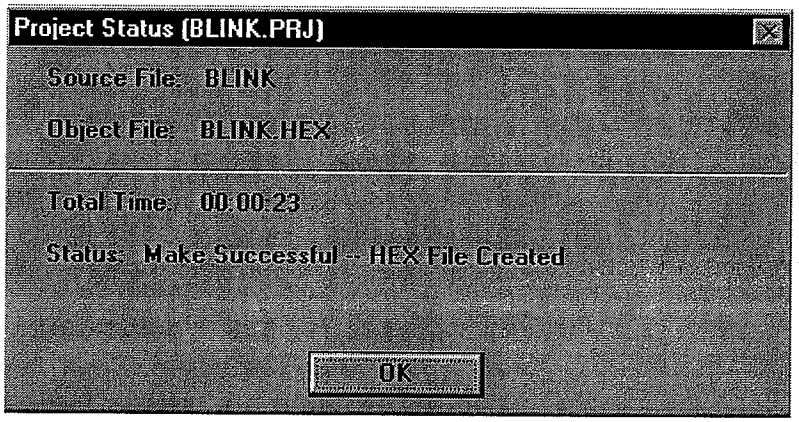


16. Mit ALT F8 das Projekt übersetzen und binden.

Beachte

Sollten Warnings oder gar Errors auftreten, handelt es sich um Fehler bei der Namensvergabe! Sie müssen in den *.A51-Dateien und im C-Code die gleichen Namen für externe Funktionen verwenden. Beachten Sie dazu auch die Kommentare in den entsprechenden Dateien.

Bei erfolgreicher Übersetzung und Bindung des Projektes erhalten Sie folgende Ausgabe:



LINKER OPTIMIEREN

Beachte
Das Segment EOF_CODE muss am Ende des Codes stehen, damit der Monitor der CPU1 das Ende der Datenübertragung erkennt. Dies ist nicht zwingend der Fall!

Link-Datei betrachten

17. Über Menü "RUN" die Link-Datei öffnen.
    Geben Sie als Parameter den Namen der HEX-Datei an. Er entspricht dem ersten Namen, den Sie im Projekt angegeben haben.

18. Die Link-Datei enthält die Adressen der einzelnen Segmente die zusammengebunden wurden. Dabei muss EOF_CODE am Ende der Code-Adressen stehen, da sonst der Transfer zur CPU nicht läuft.

Beispiel :

```
    CODE  8100H   0023H   UNIT   ?PR?_BUSYWAIT?MYPROG
    CODE  8123H   0021H   UNIT   ?PR?MAIN?MYPROG
    CODE  8144H   000CH   UNIT   ?C_C51STARTUP
!   CODE  8158H   0005H   UNIT    EOF_CODE
    CODE  8155H   0008H   UNIT   ?CO?MYPROG
```

19. Die Adresse für die Positionierung von Segment EOF_CODE auf z.B. 8080h festlegen. Dies ist aber nur notwendig, wenn EOF_CODE nicht als letztes Segment aufgeführt ist!

20. Die Linker-Anweisung – Menüpunkt „Options" - entsprechend ergänzen.

21. Das Projekt über Menü "Project/Make: Link Project" neu binden. Hier der Auszug aus der neu erstellten Datei "MYPROG.M51":

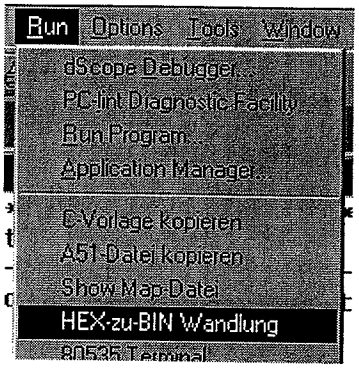
Beispiel :

```
CODE  8044H   000CH   UNIT       ?C_C51STARTUP
CODE  8050H   0008H   UNIT       ?CO?MYPROG
      8058H   0028H              *** GAP ***
CODE  8080H   0005H   UNIT       EOF_CODE
```

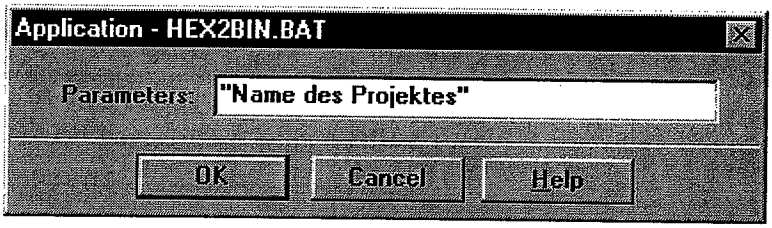
Beachte
Die Schritte 18. Bis 21. müssen wiederholt werden, bis das Segment EOF_CODE am Ende des Codes liegt!

DAS PROJEKT AUF DIE CPU1 TRANSFERIEREN

22. Das Projekt liegt nun als Datei im INTEL-Hex-Format vor und kann mit entsprechender Gerätschaft in ein EPROM gebrannt werden. Für den **Tansfer auf die CPU1** muss das Projekt als **Binär-Datei** vorliegen. Dazu führen Sie im Menü "Run" den entsprechenden Befehl aus.



Geben Sie als Parameter den Namen der HEX-Datei an. Er entspricht dem ersten Namen, den Sie im Projekt angegeben haben.



Beachte

Wenn Sie alle oben beschriebenen Schritte durchgeführt haben, liegt eine ausführbare Datei *.BIN vor, die Sie auf die CPU1 transferieren können.

Die CPU1 benötigt ein spezielles Programm für die Datenübertragung. Es ist in WORK21 eingebunden, weshalb dieses Programm gestartet werden muss, um den Transfer durchführen zu können.

23. Führen Sie im Menü "Run" den entsprechenden Befehl aus. Geben Sie dabei als Parameter wiederum den Namen des Programms an. (➔ vrgl mit 22. oben)

24. Das Programm WORK21 wird gestartet. Wählen Sie nun "File / Load File" aus. Wählen Sie dann im Unterverzeichnis USERWORK die Datei *.ASM aus.
25. Wählen Sie das Menü "Transfer / Terminal" und senden Sie Ihre Binär-Datei wie gewohnt zur CPU1. **Achtung**: *Nicht assemblieren und auch nicht binden!* Das haben Sie bereits erledigt.
26. Verlassen Sie nach dem Testen der Software die Arbeitsumgebung und quittieren Sie die gestellte Frage mit 'j'.

EIN BEISPIEL

Anhand eines Blinklichtes wird der ganze Vorgang vom Erstellen (darauf wird jetzt aber verzichtet) über das Übersetzen bis zum Übertragen auf das Zielsystem behandelt.

EINRICHTUNG

- Schliessen Sie die LED-Versuchsplatine wie folgt mit zwei Flachbandkabeln an das CPU-Board (vergl. Skizze auf Seite 1.1.29ff)

	LED-Versuchsplatine	**CPU-Board**
1. Kabel	Input LED	Port 5
2. Kabel	Output Schalter	Port 4

- Verbinden Sie das CPU-Board über das RS-232 Kabel mit dem PC (an COM2)
- Stecken Sie das Netzteil an 220 Volt an.

Achtung
nun sollten alle 8 LED auf der Versuchsplatine leuchten

C51

Verzeichnis Erstellen Sie das Arbeitsverzeichnis C51 und kopieren Sie danach die Datei BLINK.C vom Pool [P:].

Projekt Gehen Sie nun die Schritte gemäss 4.2.2ff durch, bis Sie das lauffähige Programm auf das System gebracht haben.

ÜBUNG H

Ziel

Das Lauflicht in einer ersten Fassung mit Busy-Wait – Routine aus BLINK.C51 verwenden – und Start/Stop über Schalter erstellen.

Pseudocode

```
main()
do forever
  if Schalter = EIN then
    P5 = Wert[Zaehler];
    Inkrementiere Zaehler;
    if Zaehler > 8 then
      Zaehler = 0;
```

Anschluss

Beachten Sie, dass die LED-Versuchsplatine an Port 4 (Eingang) und Port 5 (Ausgang) des Prozessor-Boards angeschlossen sind.

Initialisierung

Das Datenfeld für die Ausgabe lässt sich unter C am einfachsten als Array initialisieren.

```
data char Wert[] =
{0x00,0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80};
```

Namen

Die Definition der SFR-Namen – der Register des 80535 – finden sich in der Datei REG515.H.

Für diese Übung verwenden Sie die Ports P4 und P5.

Bitweiser Port-Zugriff

C51 ermöglicht einen einfachen Zugriff auf die einzelnen Bit eines Ports. So kann z.B. auf das Bit Nummer 2 von Port 4 über

```
P4^2
```

zugegriffen werden.

Vorgehen

1. Erzeugen Sie ein neues Projekt (z.B. LAUF) und kopieren Sie alle Dateien gemäss der Vorlage auf H:\C51
2. Erstellen Sie das C-Programm
3. Übersetzen und binden Sie das Projekt. Achten Sie unbedingt auf die Position des EOF-Codes! Er muss am Ende des Binärcodes stehen.

Beachte
Sie müssen nach jeder Änderung die Position überprüfen.

4. Übertragen Sie das Programm aufs Zielsystem und führen Sie es aus.

ÜBUNG J

Ziel

Das Lauflicht mit externem und zeitgesteuertem Interrupt realisieren, vergleichbar der Übung G.

Pseudocode

```
main()
/* Initialisieren der Variablen */
ISR_Flag setzen;
T2CON = 0X91; ET2 = 1;   /* Timer 2 für Interrupt */
CRCL = 0; CRCH = 0;      /* Zähler für Timer 2 */
IT0 = 1; EX0 = 1;        /* Intr 0 auf Flanke */
EAL = 1;                 /* Interrupts freigeben */
do forever ; /* keine Tätigkeit im Hauptprogramm */
end;

ext0_ISR()
if ISR_Flag gesetzt then
  ISR_Flag löschen;
else
  ISR_Flag setzen;
EX0=1;

time2_ISR()
if ISR_Flag gesetzt then
  P5 = Wert[Zaehler];
  Inkrementiere Zaehler;
  if Zaehler > 8 then
    Zaehler = 0;
TF2 = 0;
ET2 = 1;
```

Anschluss

Beachten Sie, dass die LED-Versuchsplatine an Port 5 (Ausgang) und am Interrupt-Port (Eingang) des Prozessor-Boards angeschlossen sind.

Initialisierung

Die Interrupts sind identisch wie bei der Assembler-Übung zu initialisieren.

Interrupt Vektoren

In der Datei INTVECT.A51 müssen die benötigten Interrupt-Vektoren aktiviert werden.

Beispiel :

```
    :
EXTRN CODE (TIMER2_ISR)
;EXTRN CODE (AD_ISR)
    :
;
CSEG AT 802Bh ; Timer 2
LJMP TIMER2_ISR
```

Hier Kommentar-Markierung entfernen.

Hier `RETI`-Befehl entfernen und `LJMP` aktivieren

Interrupt Routinen

Die Interrupt-Routinen sind gemäss der Vorlage zu gestalten! Verwenden Sie daher die (umbenannte) Datei VORLAGE.C

Vorgehen

1. Erzeugen Sie ein neues Projekt (z.B. LAUFINT) und kopieren Sie alle Dateien gemäss der Vorlage auf H:\C51
2. Ändern Sie die Datei INTVECT.A51 ab, so dass die beiden benötigten Interrupts aktiviert sind und zu den entsprechenden Interrupt-Routinen verzweigen
3. Erstellen Sie das C-Programm
4. Übersetzen und binden Sie das Projekt. Achten Sie unbedingt auf die Position des EOF-Codes! Er muss am Ende des Binärcodes stehen.

Beachte
Sie müssen nach jeder Änderung die Position überprüfen.

5. Übertragen Sie das Programm aufs Zielsystem und führen Sie es aus.