FAXIMILE MIT DEM MIKROCOMPUTER

von Willi Sicking

Faximile ist eine primär kommerzielle Technik zur Übermittlung von Schriftstücken, Wetterkarten, Satellitenbildern und Pressebildern. Als die ersten ausgemusterten Faxgeräte in größeren Stückzahlen auf den Gebrauchtwarenmarkt kamen, hielt diese Technik auch Einzug in den Amateurfunk. Eine Faxstation läßt sich jedoch auch auf rein elektronischem Wege realisieren. Dazu bietet sich ein Mikrocomputer natürlich an. Um dabei die Vorteile und Grenzen aufzuzeigen, habe ich Anfang 1983 in der Zeitschrift "RTTY" einen Artikel über das Thema verfaßt. Mit freundlicher Genehmigung der Redaktion von "RTTY" sind dieser Zeitschrift ein Großteil des Textes und alle Zeichnungen entnommen.

Die Fax-Anlage besteht aus folgenden Teilen:

1) DAI-Computer 48KB und MDCR
2) Fax-Timer
3) FM zu BCD-Converter
4) Programm

Im folgenden werden die einzelnen Punkte beschrieben:

Der DAI Personal Computer (1)

Mit seiner hohen Farbgrafikauflösung von 255*335*4 Graustufen war der DAI PC zumindest in diesem Punkt lange führend, so daß er für Faximile geeignet schien. Lediglich die 4 Graustufen stellen eine Einschränkung dar. Es ist nur zu hoffen, daß die PC in Zukunft großzügiger mit Bildspeicher ausgerüstet werden. Als Programm- und Bildspeicher dient ein MDCR. Wegen seine hohen Übertragungsrate von 6000 Baud ist er sehr gut zum Laden und Abspeichern der Bilder geeignet.

Der Fax-Timer (2)

In der Fax-Technik wird quarzsynchron gearbeitet, das heißt, daß die Aufzeichnung und Sendung mit einer Genauigkeit von 10^{-5} pro Umdrehung erfolgen muß. Bild 1 zeigt anhand einer Wetterkarte, die direkt von Meteosat 2 empfangen wurde, das Aufzeichnungsformat eines Fax-Bildes. Rechts unten beginnend wird zunächst ein 3 Sekunden langes Startsignal von 300 Hz gesendet.
Darauf folgt das Einphassignal. Es besteht aus 12.5 ms schwarzem und 237.5 ms weißem Bildinhalt. Die nachfolgenden Bildzeilen beginnen mit einem Linestartsignal. Gedacht ist das Signal für den Aufbau einer automatischen Verstärkungsregelung. Oben, am Bildende ist das Stopsignal zu sehen. Es dauert 5 Sekunden und hat eine Frequenz von 450 Hz. Der Fax-Timer dient also zur Synchronisation der Sendung und des Empfangs. Er gibt quarzgenau das Startzeichen für jede Zeile. Die Aufnahmedauer einer Zeile wird durch eine Warteschleife im Programm eingestellt.
Die Schaltung besteht aus einem Quarzoszillator, der hier mit 256 KHz schwingt und 2 Teilern. Ein J-K-Flipflop formt ein Tastverhältnis von 1:1. Das Prinzip des programmierbaren Teilers stammt aus der Zeitschrift UKW-Berichte Heft 4/79 und ist dort ausführlich beschrieben. Der hardwaremäßige Fax-Timer hat gegenüber einer Softwarelösung folgende Vorteile:

2.1 Die Zeit des Timers läßt sich mit 2 Tastern verlängern, um das Bild einzuphasen. (Einphasen bedeutet, das Bild mittig auf den Bildschirm bringen)

2.2 Da das Tastvehältnis 1:1 ist, kann man das Signal als Testsignal vewenden. Mit einer Warteschleife im Programm stellt man die Zeilendauer so ein, daß gerade eine Periode der eingestellten "Drehzahl" den Bildschirm ausfüllt.

2.3 Der Timer kann auch durch andere Signalquellen ersetzt werden. So lassen sich über ein kleines Interface der DAI und ein Telekopierer phasenstarr miteinander verbinden, um Bilder vom Kopierer in den DAI zu überspielen und umgekehrt.

FM zu BCD-Decoder (3)

Die Schaltung des FM zu BCD-Decoders stammt von Volker Wraase. Sie war in der CQ-dl veröffentlicht und ist dort sehr gut beschrieben.
Das NF-Signal wird vom Operationsverstärker IC1 in ein Rechtecksignal umgewandelt. Das nachfolgende Differenzierglied erzeugt an der abfallenden Flanke Nadelimpulse, die einen Oszillator und ein Monoflop (IC5) starten. Diese Oszillatorfrequenz ist ca. die 16-fache Schwarzfrequenz. Oszillator und Monoflop werden mit den Potis VR1 und VR2 so eingestellt, daß der nachfolgende Binärzähler IC6 bei Schwarzfrequnz (1500Hz) bis 16 zählt und bei Weißfrequenz (2000Hz) gerade nicht mehr zählt. Von IC7 werden diese Werte bis zur nächsten Periode gespeichert. Am Ausgang des IC7 ist ein Diodendisplay angeschlossen. Es besteht aus einem 4 zu 16 Decoder, der 16 Leuchtdioden (für 16 Graustufen) ansteuert. Nur so kann man beurteilen, ob richtig abgestimmt ist und ob VR1 und VR2 richtig eingestellt sind.
Mit dem Schalter S2 kann man das Synchronsignal zu Testzwecken ins Bild einblenden.

Programm (4)

Das Programm stammt in der Grundversion von der Softwarefirma Quasar in Haltern, wurde jedoch in wesentlichen Punkten geändert und verbessert.

4.1 Senden

Im Anfang habe ich das Sendesignal direkt mit dem Tonausgang des DAI erzeugt. Inzwischen verwende ich jedoch einen AM- und FM-Modulator, der über 2 Bit des DCE-Bus angesteuert wird. So kann ich Bilder zum Telekopierer überspielen oder über Funk senden.

4.2 Empfangen

Die Auflösung von 225*335*4 Graustufen ist natürlich für Fotofax normalerweise nicht ausreichend. Daher habe ich versucht, Bilder softwaremäßig mit einer höheren Anzahl von Graustufen zu erzeugen. Das Programm legt in dieser Betriebsart immer 2 Punkte zusammen, so daß 9 Graustufen möglich sind. Dabei halbiert sich natürlich die Auflösung.

Im Diagramm (Bild 2) ist zu sehen, daß sich im mittleren Bereich ein starker Fehler ergibt. Ein Unterprogramm eliminiert nun die Zahlen 8,9,10,11 und füllt die restlichen mit 15 auf. So erhält man eine Grauverteilung, die brauchbar ist.

4.3 Testbild erzeugen

4.4 Rufzeichen in die zu sendende Grafik eintragen

4.5 Bild von dem MDCR laden und abspeichern

4.6 Abtastgeschwindigkeit (Drehzahl) eingeben

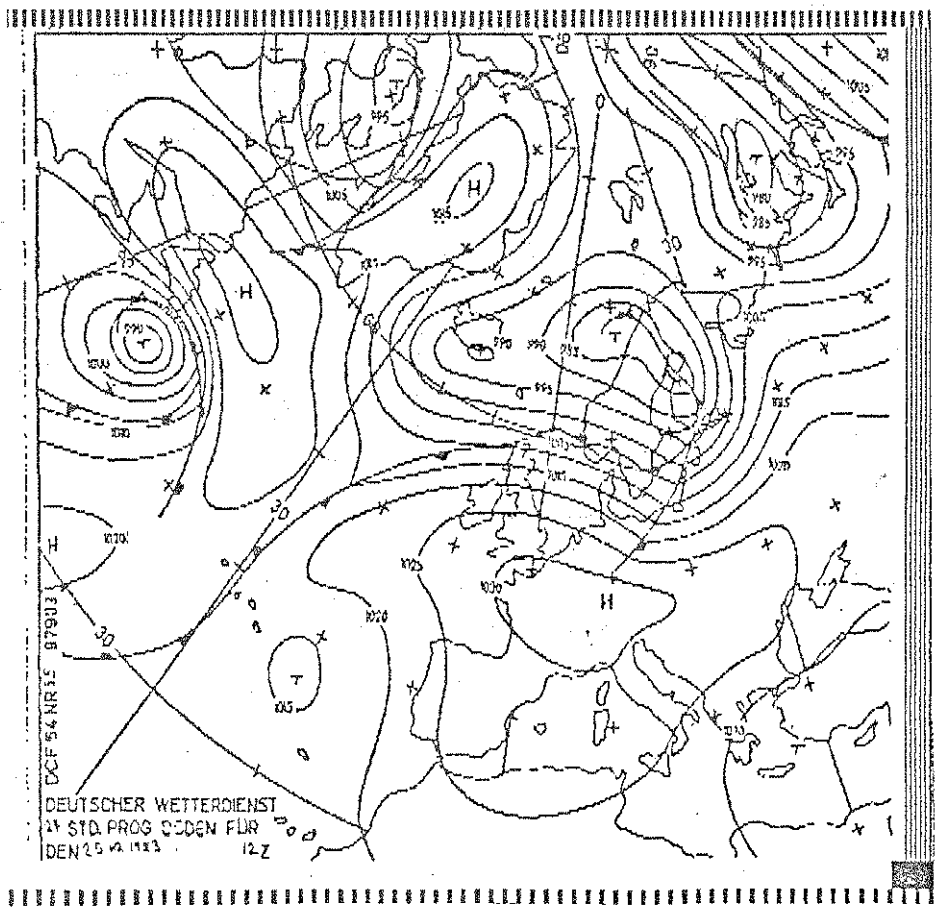
4.7 Bild auf dem Drucker ausgeben.
Die im Anhang gezeigten Bilder sind mit dem neuen Hardcopy-programm des DAI-Club auf einem C.Itho 8510 ausgedruckt. Das Programm stammt von Wilfried Rüsse.

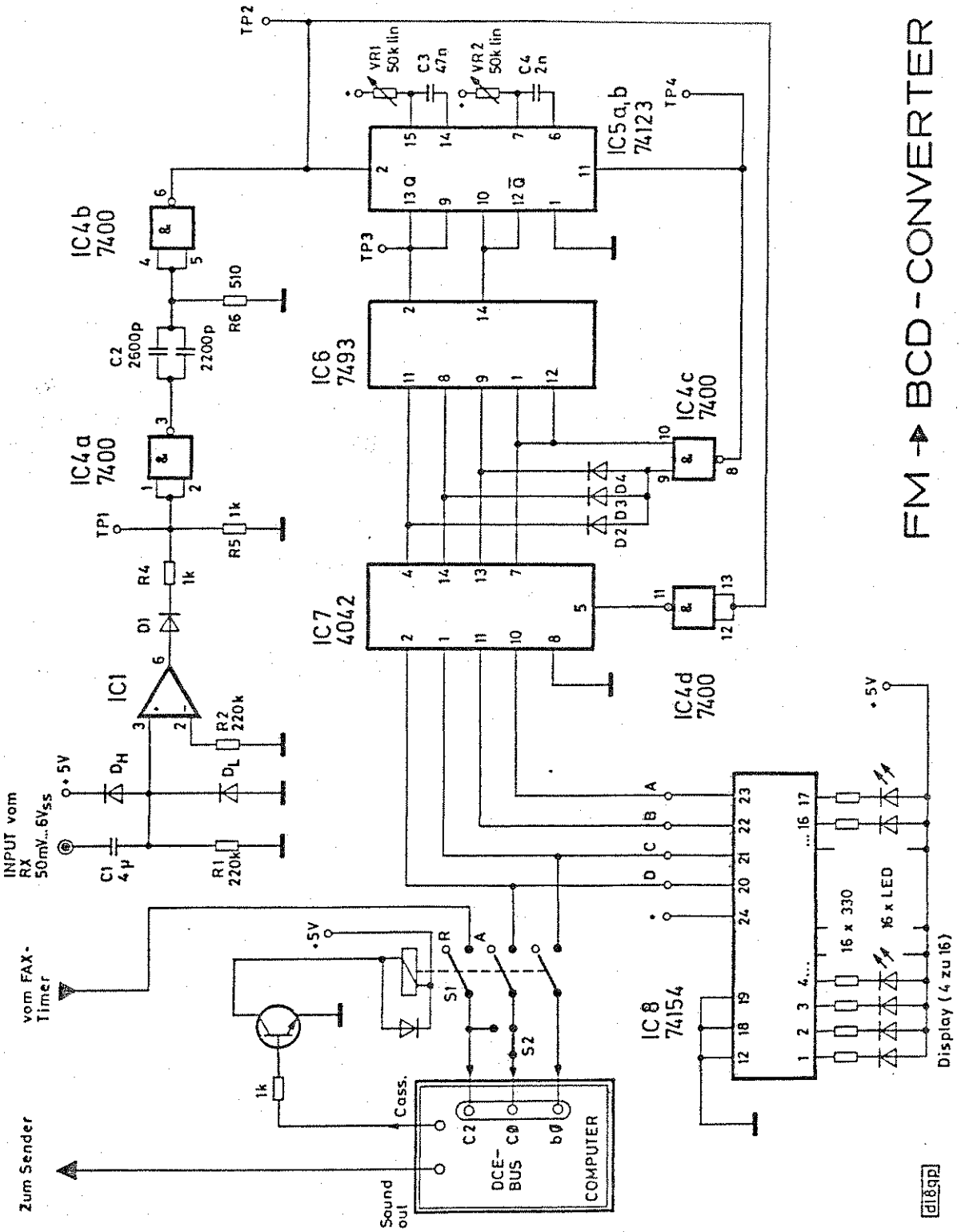
Ich hoffe, daß sich durch diesen Artikel angeregt, mehrere DAI-benutzer mit dem Problem Bildverarbeitung beschäftigen, da dieses Thema in Zukunft immer mehr an Bedeutung gewinnt.

Für weitergehende Auskünfte stehe ich gerne zur Verfügung.

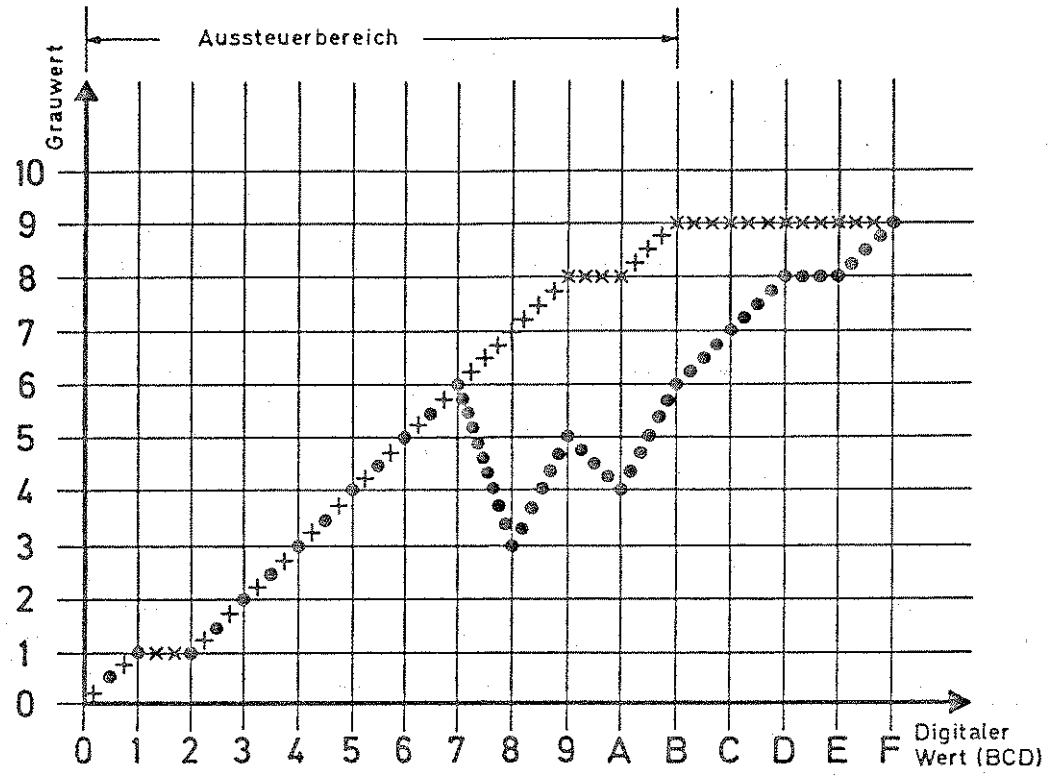
Willi Sicking Jägerweg 31 4423 Gescher 02542/611

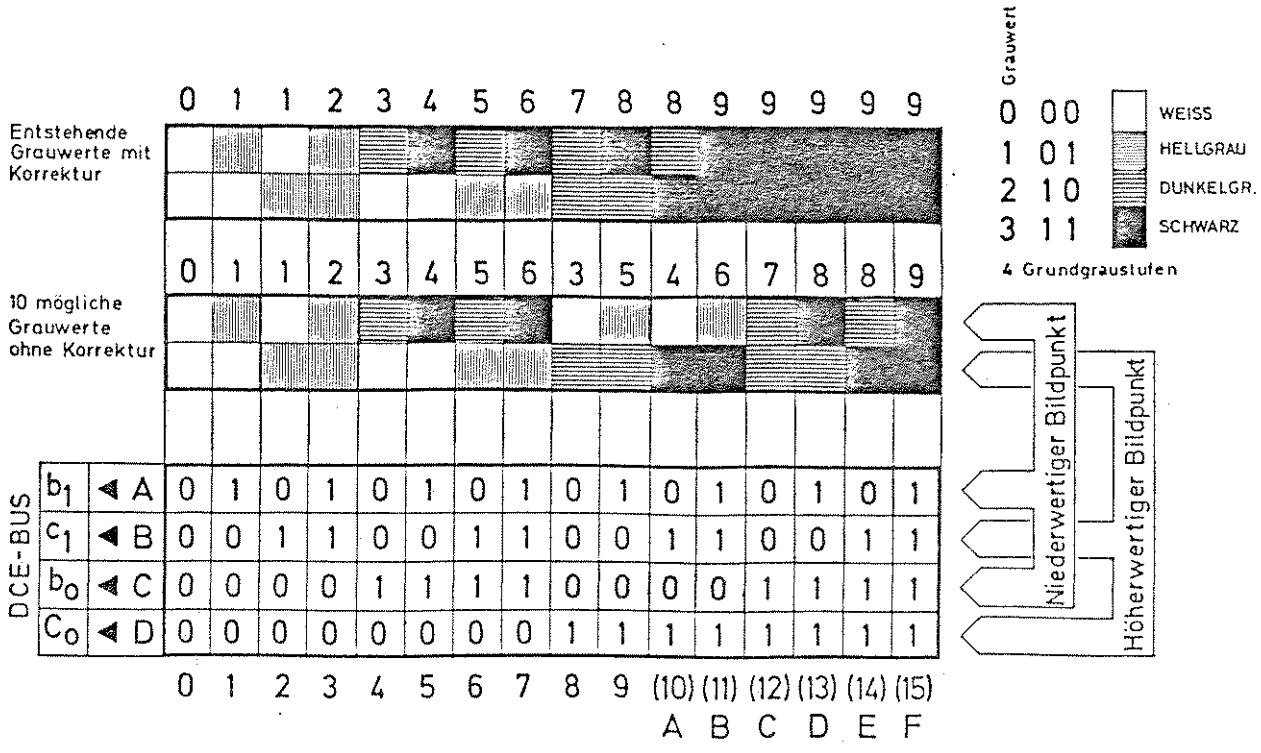
W. Sicking



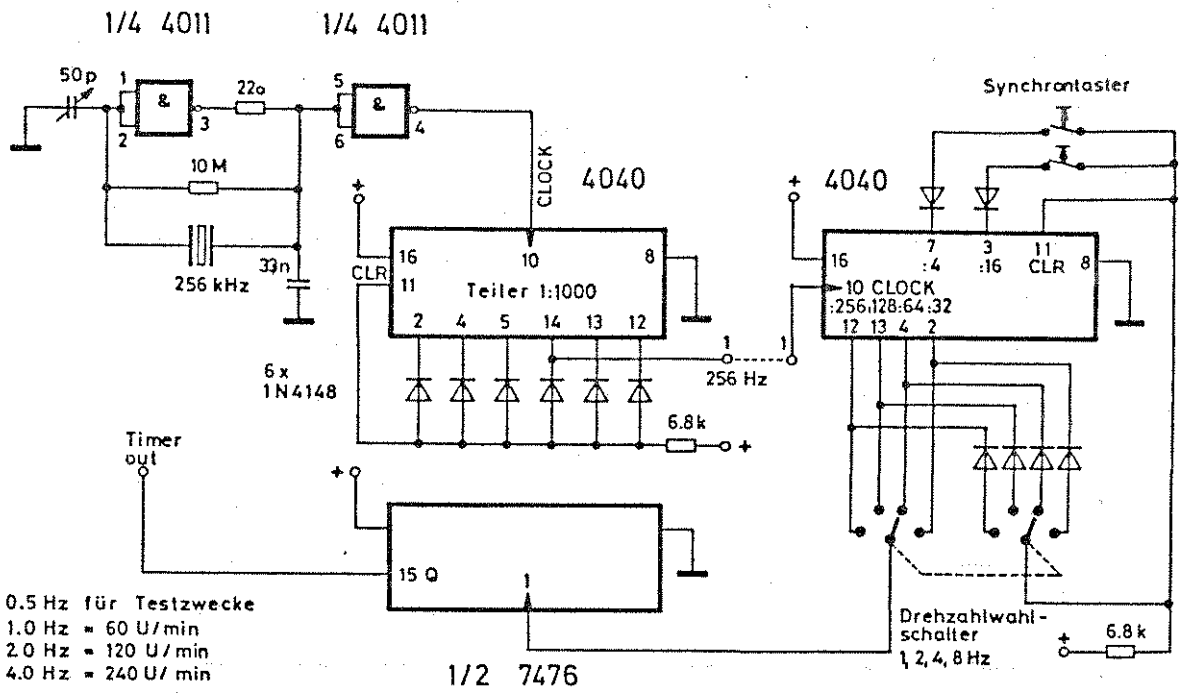

FM → BCD-CONVERTER
INPUT vom RX 50mV... 6Vss
vom FAX-Timer
Zum Sender
Sound out
Cass.
DCE-BUS
COMPUTER
IC1
IC4a 7400
IC4b 7400
IC4c 7400
IC4d 7400
IC5a,b 74123
IC6 7493
IC7 4042
IC8 74154
16 x 330
16 x LED
Display (4 zu 16)
TP1
TP2
TP3
TP4
VR1 50k lin
VR2 50k lin
C1 4µ
C2 2600p
2200p
C3 47n
C4 2n
R1 220k
R2 220k
R4 1k
R5 1k
R6 510
D1
D2
D3
D4
S1
S2
dl8gp

IB 18,4

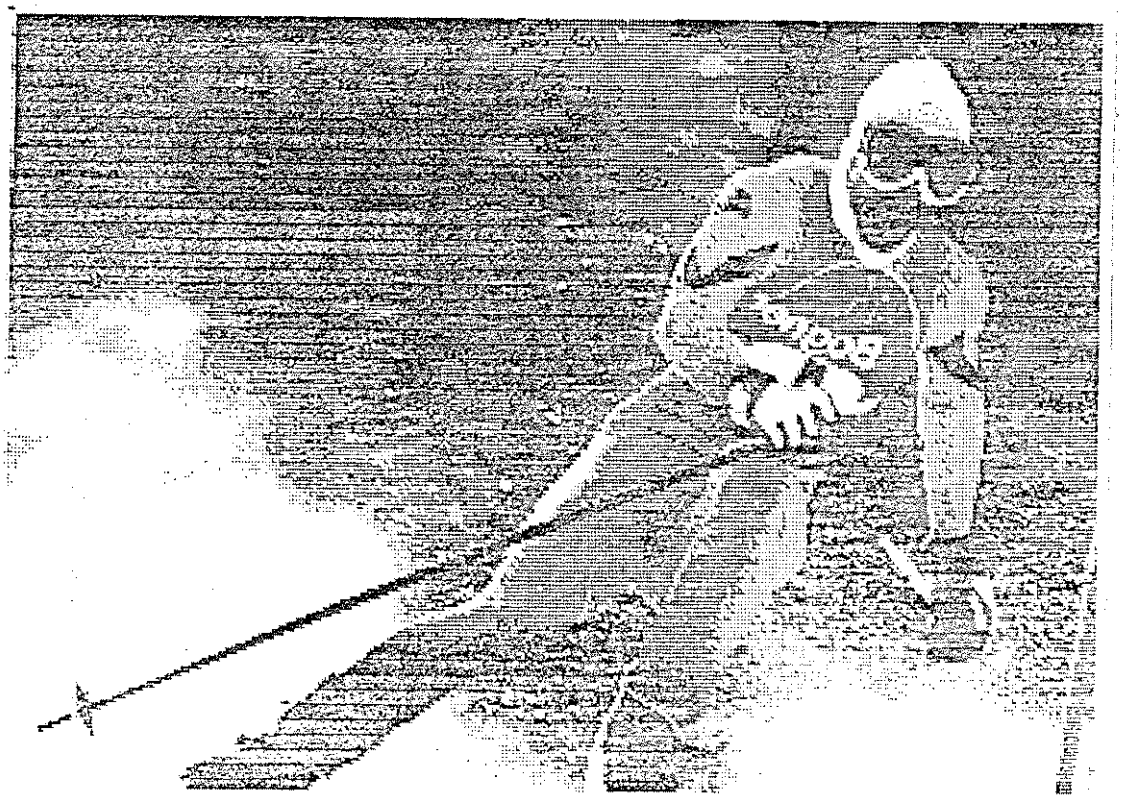


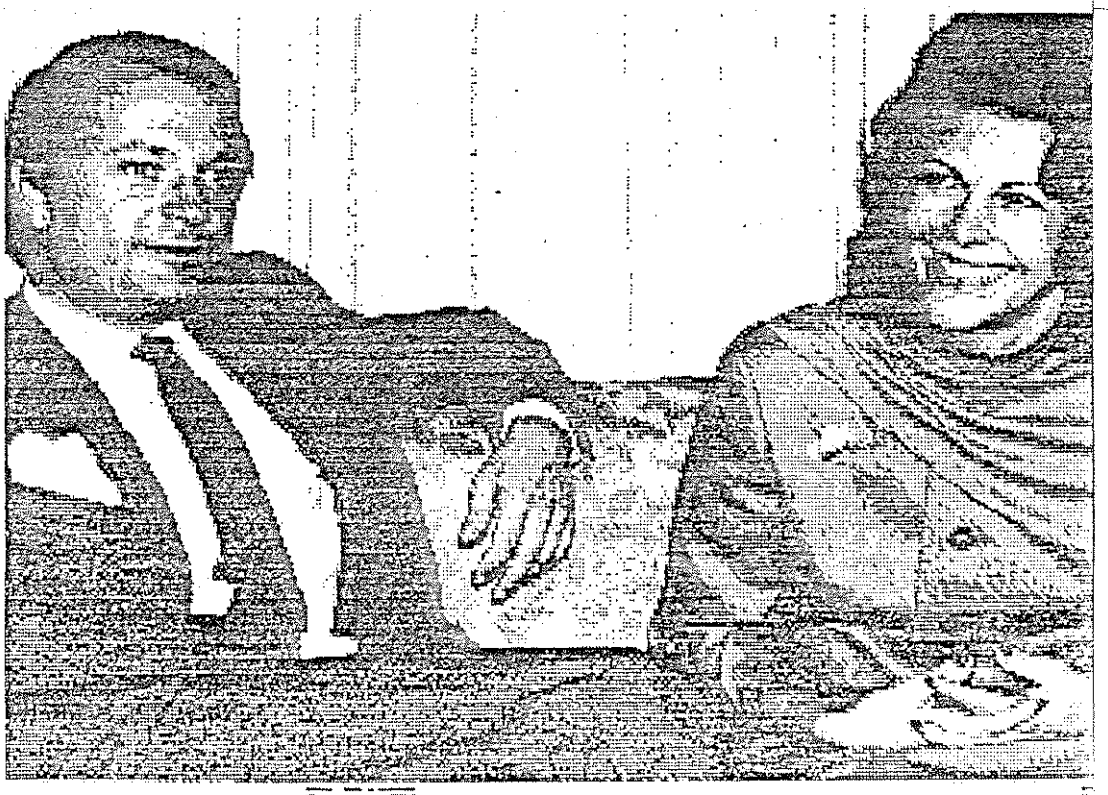


Die [0] gekennzeichneten Nullen werden zu 1 gesetzt.

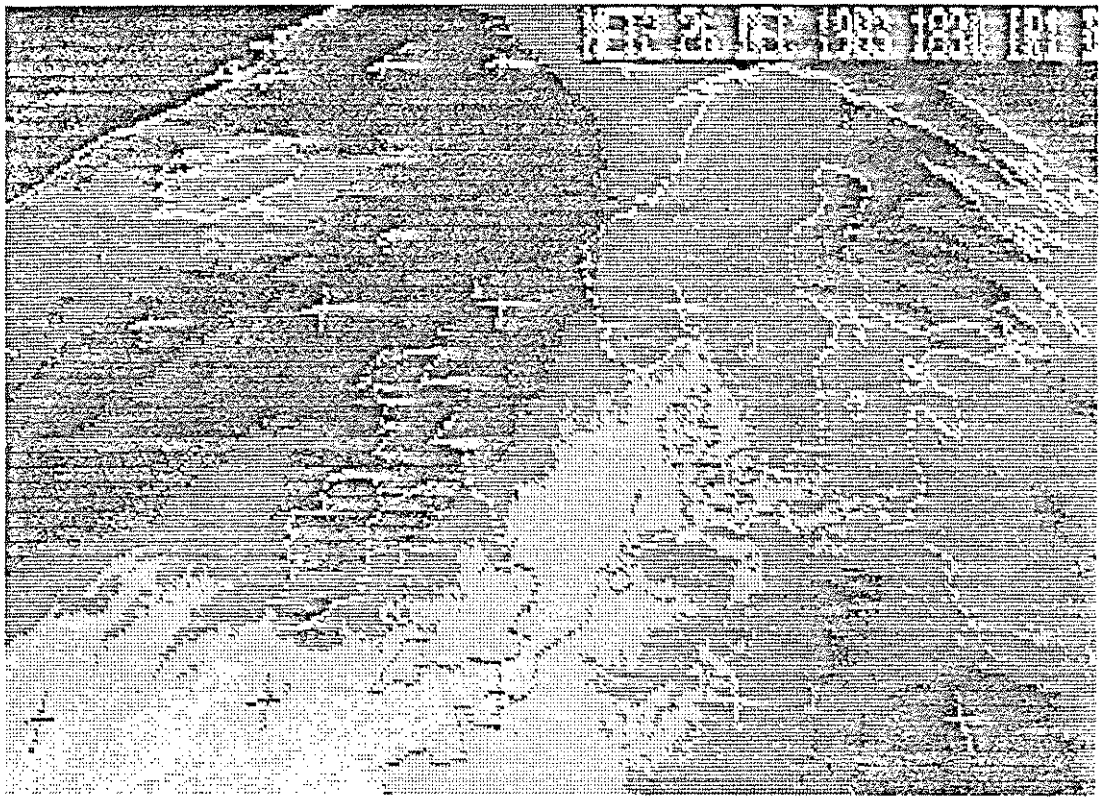

1/4 4011
1/4 4011
50p
22o
10 M
256 kHz
33n
CLOCK
4040
CLR
Teiler 1:1000
6 x
1N4148
256 Hz
6.8k
Timer
out
15 Q
1/2 7476
Synchrontaster
4040
10 CLOCK
:256:128:64:32
Drehzahlwahl-
schalter
1, 2, 4, 8 Hz
6.8k
0.5 Hz für Testzwecke
1.0 Hz = 60 U/min
2.0 Hz = 120 U/min
4.0 Hz = 240 U/min

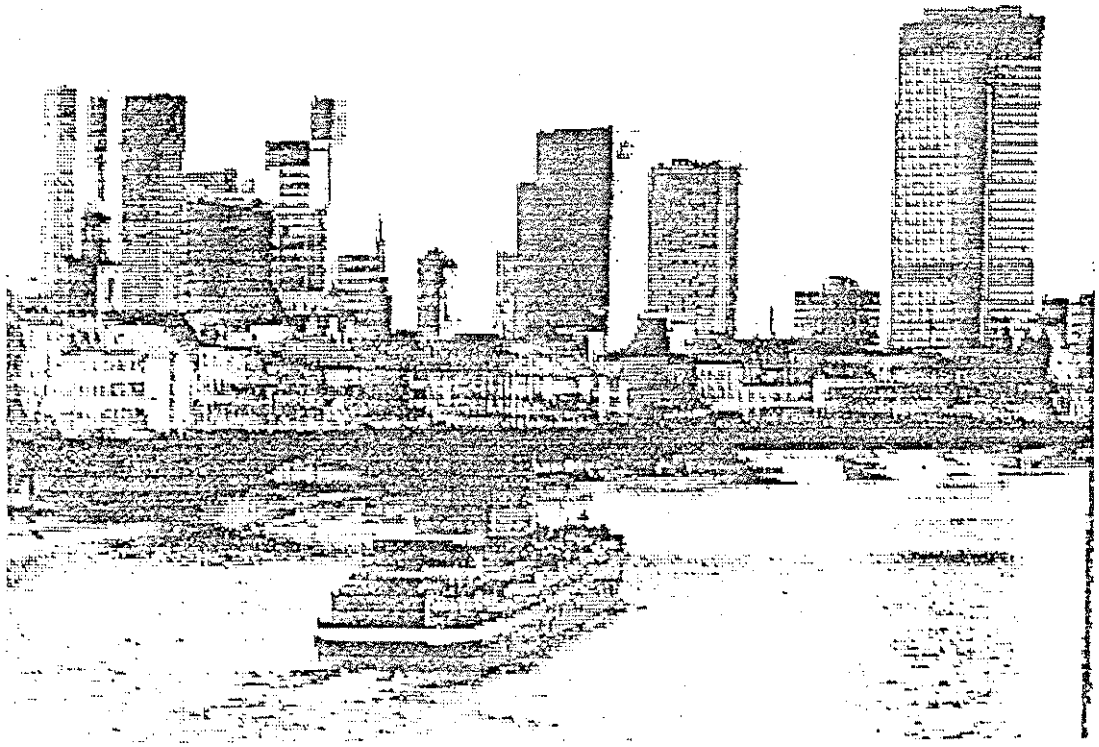
FAX-TIMER





IB 18,7





PRINT Routinen auf dem DAI

Autor: J. Boerrigter

In diesem Artikel werden die meisten für den Benutzer wichtigen Print Routinen beschrieben. Diese Routinen drucken Texte oder Zahlen auf Bildschirm und/oder Drucker aus und sind somit für jeden der häufig in Maschinensprache programmiert von besonderem Interesse.
Für weitergehende Informationen (z.B. wie die Routinen tatsächlich funktionieren) sollten Sie das 'DAI Firmware Manual' benutzen.

In den angegebenen Beispielen wird vorausgesetzt, daß der auszugebende String im Speicher an der Stelle XXXX beginnt. Als Beispiel-String wird immer das Wort "TEST" benutzt.
'LB' dient als Abkürzung für Längenbyte

1. String ausgeben:

1.1 Diese Routine beginnt bei **#DB32**
Beim Aufruf muß HL die Stringadresse enthalten
Der String muß folgendes Format haben:
- Ein Längenbyte
- Der String in ASCII

Programmbeispiel:

```
XXXX 04 - 54.45.53.54
     LB  'T  E  S  T   in Hex-Zahlen

     LXI   H,:XXXX   HL mit der Stringadr. laden
     CALL  :DB32     'TEST' ausgeben
```

Nach Beendigung der Routine zeigt HL auf das nächste Byte nach 'TEST'. Alle anderen Register bleiben erhalten.

1.2 Eine alternative Routine findet man ab **#DB44**. Diesmal zeigt HL direkt auf den String. Die Länge befindet sich im Akku.

Beispiel:

```
XXXX 54.45.53.54      ('TEST' in Hex-Zahlen)
     LXI  H,:XXXX      HL mit der Stringadr. laden
     MVI  A,4          A  mit 4 laden (der String ist ja
                       vier Zeichen lang)
     CALL :DB44        'TEST' ausdrucken
```

Nach Beendigung der Routine zeigt HL wieder auf das nächste Byte nach 'TEST'. Alle anderen Register bleiben erhalten.

2. Einen Text ausgeben

2.1 Diese Routine befindet sich bei Adresse **#DAD4**. Diese Routine beinhaltet viele zusätzliche Optionen. Sie kann benutzt werden um Texte auszugeben, die wiederum einen Verweis zu einem weiteren (Unter) Text besitzen.
Beim Aufruf zeigt HL auf den Textbeginn. Nach Beendigung weist HL wieder auf das Zeichen hinter dem Text. Alle anderen Register werden nicht verändert.
Format eines Textes:
- Die Bytes in ASCII. Alle Zeichen müssen zwischen #01 und #7F liegen
- 00 bedeutet Ende des Textes

Programmbeispiel:

```
XXXX 54.45.53.54 - 00    'TEST' mit Nullbyte am Ende
     LXI  H,:XXXX        HL zeigt auf den Text
     CALL :DAD4          'TEST' ausgeben
```

2.1.2 Format eines Textes mit Verweisen auf andere (Unter-) Texte

- Ein Verweis besteht aus zwei Bytes. Diese werden im folgenden als ein 16 Bit Wort aufgefaßt
- Das erste Byte des Verweises muß >= #80 sein. Dies kennzeichnet den Begin eines Verweises auf einen Untertext
- Wenn in diesem Wort Bit 14 = 1 ist, dann werden die restlichen Bits (0-13) auf die Adresse #C000 auf addiert um den Beginn des Textes zu finden. Dieser Untertext endet wieder mit einem Nullbyte
- Sollte Bit 14 = 0 sein, so werden wieder die Bits (0-13) auf #C000 aufaddiert, nur wird nun an dieser Stelle ein String im Format Längenbyte + Text erwartet.

Beispiele:

```
DD0A  8D.1B  'LOAD'     String - Untertext
      DB.F3  'ING'      Untertext mit Nullbyte Ende
      DC.15  ' ERROR'        "      "      "      "
      20
      00
      LXI  H,:DD0A   Textbegin
      CALL :DAD4     'LOADING ERROR ' ausgeben
```

8D1B Bit 15=1: Kennzeichnung eines Verweises
Bit 14=0: Zeigt auf einen String mit der Adresse:
C000 + 0D1B = CD1B:
CD1B 04 - 4C.4F.41.44
LB L O A D

DBF3 Bit 15=1: Kennzeichnung eines Verweises
Bit 14=1: Zeigt auf einen Text mit der Adresse:
C000 + 1BF3 = DBF3
DBF3 49.4E.47 - 00
I N G

DC15 wieder Bit 14 und Bit 15 = 1. Dies ergibt die Adresse DC15
DC15 20.45.52.52.4F.52 - 00
E R R O R

Viele weitere Beispiele befinden sich in den Adressen **#DB6F - #DD19**

2.2 Eine weitere Print Routine beginnt bei der Adresse **#DAFF**. Diese Routine bearbeitet den Text in der exakt gleichen Weise wie #DAD4, aber die Routine wird etwas anders aufgerufen:

Beispiel:

```
XXXX              (Beginn des Textes, Format vgl. 2.1)
CALL  #DAFF
DBL   XXXX        Die Adresse des auszudruckenden Textes befindet sich als Zwei-Byte-Konstante hinter dem
```

CALL Befehl. Diese Datenblock Adresse wird vom Stack geholt, der Stackpointer wird hinter die Adresse verschoben und der Text wird ausgegeben.
Alle Register bleiben erhalten!

2.3 Eine Spezialform der Routine 2.2 befindet sich ab Adresse **#CEE4**. Diese Routine wird benutzt wenn man sich auf einer umgeschalteten ROM-Bank befindet. Bevor nun Routine 2.2 den Text druckt, wird erst die ROM-Bank 0 angewählt.

3. Verschiedene nützliche PRINT-Routinen

3.1 Routinen, die immer benutzt werden können:

3.1.1 **#CE68** Einen arithmetischen Ausdruck ausgeben, gefolgt von einem Space. BC zeigt auf den Ausdruck

3.1.2 **#CE6B** ein Leerzeichen ausgeben

3.1.3 **#CE70** ein Komma drucken

3.1.4 **#CE75** Einen von Leerzeichen umschlossenen String ausgeben

Beispiel:
```
CALL :CE75
DBL  :XXXX
```

3.1.5 **#DB0D** Cursor aufs nächste Feld setzen. Wird benutzt als TAB auf das nächste Ausgabefeld. Die Feldpositionen sind 0,12,24,36,48

3.1.6 **#DB2A** Cursor auf Spalte 8 setzen

3.1.7 **#DD5E** Ein 'Return' drucken

3.1.8 **#DD60** Ein Zeichen, welches sich im Akku befindet, ausgeben

3.2 Routinen, die nur vom BASIC per CALLM aufgerufen werden können

3.2.1 **#0EFBD-#0EFE0:** Viele nützliche LIST Routinen. Diese Routinen können nur benutzt werden wenn die Routine vom BASIC aufgerufen wurde, weil sie sich in E-ROM Bank 0 befinden. Die auszugebenden Zahlen müssen sich im Mathe-Akku (#D5-#D8) befinden

Beispiel:
```
LXI  H,16      HL mit 16 laden
CALL :EB46     Zahl in den Mathe-Akku bringen
CALL :EFBD     Zahl dezimal ausgeben: ' 16'
```

3.3 Routinen, die nur in vom UTILITY aus gestarteten Maschinenprogrammen genutzt werden können:

3.3.1 **#ED18** Eine zwei-Byte Zahl, die sich in HL befindet ausgeben

3.3.2 **#ED1D** Eine ein-Byte Zahl aus dem Akku ausgeben

3.3.3 **#ED2F** Einen Text drucken; HL zeigt auf den Text. Das Format ist: Text - Ende durch Nullbyte

3.3.4 **#ED3A** 'Return' ausgeben

3.3.5 **#EEB4** Ein Zeichen drucken. Das Zeichen befindet sich im C-Register im ASCII Format

3.4 Routinen um Zahlen aus dem Mathe-Akku auszugeben

3.4.1 **#DB4A** Ausgabe im HEX-Format

3.4.2 **#DB53** " " Integer Format

3.4.3 **#DB59** " " Gleitkomma Format (FPT)

Anmerkung der Redaktion:
Falls jemand andere ROM Routinen entschlüsselt hat und diese erklären kann, so sendet doch bitte diese Erkenntnisse zur Veröffentlichung ein, da es bestimmt Mitglieder gibt, denen diese Beiträge bei der Maschinenprogrammierung sehr nützlich sein können, bzw. auch zum besseren Verständnis der Vorgänge im DAI beitragen.

Tips zu Ken-Dos (und Basic)

Das Ken-Dos Betriebssystem, obwohl es inzwischen einigermaßen ausgereift ist, enthält noch ein paar Besonderheiten, zum Teil auch kleinere Fehler, die, wenn man darüber nicht Bescheid weiß, zu ziemlich ärgerlichen und vor allem verwirrenden Programmabstürzen führen können. Auch Basic ist nicht ganz fehlerfrei. Über einige dieser "Ticks" wollte ich hier berichten.

Eine Besonderheit betrifft die Behandlung von verschiedenen Dateitypen auf Diskette. Das normale DAI Kassettenbetriebssystem kennt die drei Dateitypen "0" (Basic Programm), "1" (Utility Datei), und "2" (Array), obwohl Maschinenspracheprogramme auch beliebige weitere Typen verwenden können. Die Standardkassettendateien, die von Basic SAVE und SAVEA oder Utility W geschrieben werden, haben alle den gleichen Grundaufbau: sie bestehen aus einem Dateikopf mit Typ und Namen der Datei (geschrieben durch ein Programm namens WOPEN), aus zwei Datenblöcken (jedes geschrieben durch WBLK), und aus einer Dateiendekennung (geschrieben durch WCLOSE). Bei Basicprogrammen (Typ "0") bestehen die zwei Datenblöcke aus dem (verschlüsselten) Programmtext und der Symbolentabelle. Bei Typ "1" und "2" werden die gesamten Daten in den zweiten Block geschrieben; der erste Block ist für Typ "1" nur zwei Byte lang und enthält die Ladeadresse der Datei, bei Typ "2" ist dieser Block gar nur ein Byte lang und enthält den Arraytyp (Integer, Floatingpoint oder String). Die Adresse und Länge des zu schreibenden Blocks wird bei jedem Aufruf von WBLK in den HL und DE Registern übergeben.

Ken-Dos, im Gegensatz zum Kassettensystem, kennt von Haus aus zusätzlich zu den Typen "0","1","2" noch die Typen "3" (SRC, d.h. Assemblerquelltext), "4" (RND, d.h. Random- oder nichtsequentielle Datei), "5" (TXT, also Textdateien), und "6" (DBS, d.h. Datenbasis). Allerdings kennt es auch für Maschinenspracheaufrufe nur diese Typen: alle anderen Typen werden in den Bereich "0"-"7" umgewandelt. (Der noch theoretisch mögliche Typ "7" ist nicht implementiert **und sollte nicht verwendet werden**, da er zum Aufhängen des DIRectorybefehls führt.)

Es ist natürlich nicht sinnvoll, für eine Datei mehrere Datenblöcke auf Diskette zu schreiben, erst recht nicht kurze Blöcke von einem oder zwei Byte, zumal ein auch nur partiell beschriebener Sektor für weitere Daten gesperrt ist. Deshalb behandelt Ken-Dos die Aufrufe zu WBLK und dem entsprechenden Leseprogramm RBLK etwas anders als das Kassettensystem. Zur Kompatibilität mit dem DAI-Betriebssystem erwartet Ken-Dos zwar auch zwei Aufrufe zu WBLK, aber erst beim zweiten Aufruf werden die Daten tatsächlich geschrieben; dies gilt auch für Typ "0". Beim ersten Aufruf zu WBLK werden bei Typ "0" nur die Angaben über die Größe der beiden Blöcke in die **Directory** geschrieben! Bei Typen "1" und "3"-"6" werden beim ersten Aufruf von WBLK gerade zwei Bytes Daten erwartet, und diese werden auch in die Directory geschrieben, also nicht auf den Datenteil der Diskette. Diese zwei Bytes werden beim Lesen entsprechend wieder ausgespuckt. Bei Typ "1" und "3"-"6" verhält sich Ken-Dos, von außen gesehen, genauso wie das Kassettensystem, solange der erste Datenblock wirklich nicht länger als zwei Byte ist. Programme, die diese Typen verwenden und mit Kassette funktionieren, laufen unverändert auch mit Ken-Dos.

Diese Kompatibilität gilt aber nicht für Typen "0" und "2". D.h., Standardanwendungen dieser Typen, also die Basic Befehle LOAD, SAVE, LOADA, SAVEA, funktionieren wohl auch unter Ken-Dos, aber *Nichtstandard*anwendungen (wie etwa die Typ "2" Dateien des AHT-Assemblers) funktionieren garantiert nicht, und Nichtstanddarddateien müssen auf Diskette unter einer der Typen "3"-"6"

(oder "1") abgespeichert werden. Die Programme, die solche Dateien erzeugen, müssen natürlich auch für Ken-Dos umgeschrieben werden! Dies ist besonders ärgerlich, weil es unnötig ist.

Ein Grund für die Inkompatibilität bei Typ "2" ist, daß Ken-Dos die beim ersten Aufruf von WBLK übermittelten Daten überhaupt nicht wörtlich auf die Diskette schreibt (auch nicht in die Directory), und somit kann es diese Daten beim Lesen auch nicht wiedergeben. Nur der Datentyp des vermeintlichen "Arrays" wird ermittelt und durch setzen eines Bits mit in den Dateityp "2" in der Directory eingepackt: also aus eventuell zwei Bytes Daten wird nur ein Bruchteil der Information behalten!

Eine weitere Abweichung vom Kassettenprotokoll bei Typ "2" und "0" ergibt sich dadurch, daß Ken-Dos (bei diesen Typen) auch nicht die Information über Blocklage und Blocklänge in den CPU-Registern verwendet, sondern diese Angaben bei Typ "0" aus den Basiczeigern in den Speicheradressen 29B-2A4 entnimmt, und bei Typ "2" aus der Adresse 1CE (wo SAVEA den Arraytyp ablegt). Dies ist zu beachten beim Schreiben und Lesen dieser Typen etwa mit einem Maschinenprogramm.

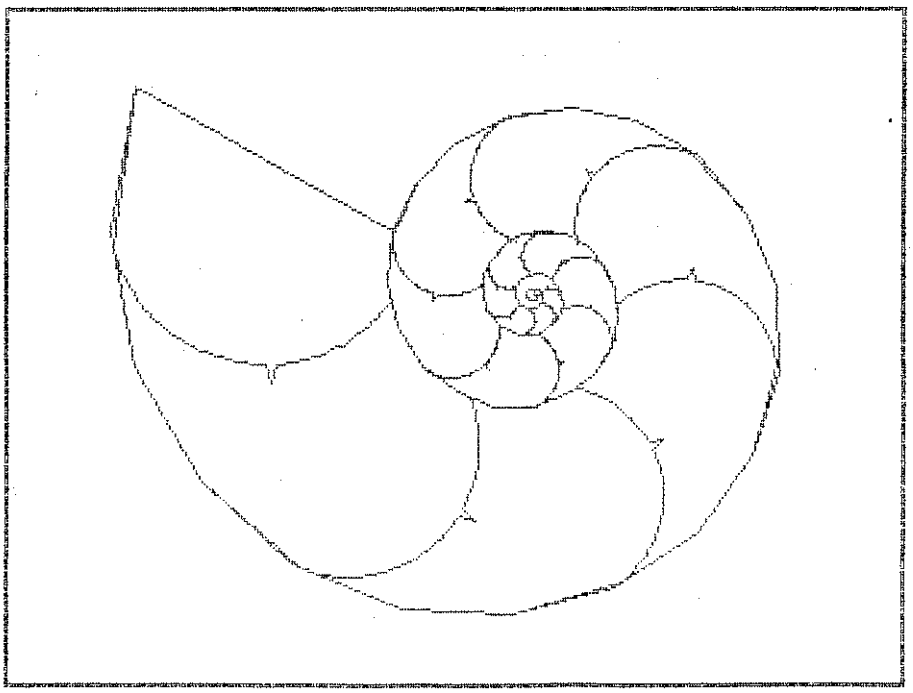
In der Ken-Dos Betriebsanleitung wird beschrieben, wie man in einem Programm zwischen Ken-Dos und DCR Betrieb umschalten kann. Hierbei können Systemabstürze auftreten, wenn man nicht vorsichtig ist! Das liegt daran, daß Ken-Dos, wenn die Diskette angesprochen worden ist, die Motoren noch eine Weile nachlaufen läßt, damit bei häufigem Zugriff der Kopf nicht immer wieder geladen und abgeladen wird. Zu diesem Zweck wird der Cursorinterrupt (seine Adresse steht in #70-71) umgeleitet zu einem Unterprogramm bei #F726 in Ken-Dos Bank 0, das einen Zähler herunterzählt und zur rechten Zeit die Motoren abschaltet und den Cursorinterrupt wieder nach #D9A9 zurückschaltet. Wenn man aber zu DCR umschaltet, während die Disketten noch laufen, dann wird die Ken-Dos Bank umgeschaltet, und bei #F726 steht nun etwas ganz anderes! Es kommt beim nächsten Interrupt zum Absturz. Also, vor dem Umschalten auf DCR immer erst den Inhalt des Vektors bei #70-71 kontrollieren, und abwarten, bis diese Adresse nicht in den Ken-Dos Bereich zeigt (also, bis #71 nicht mehr #F7 enthält). Die gleiche Empfehlung gilt natürlich für jede andere Anwendung, die Ken-Dos Bänke umschaltet!

Nicht nur Ken-Dos, sondern auch DAI-Basic kann manchmal Ärger machen. Z.B.: manche Basic Programme verschieben sich selber im Speicher durch die Befehlsfolge POKE #29B,#xx:POKE #29C,#xx:CLEAR xxxxx. Dadurch soll ein Speicherbereich für ein Maschinenspracheunterprogramm frei gemacht werden, oder etwa ein in einer REM Zeile verstecktes Maschinenprogramm an eine ganz bestimmte Adresse gebracht werden. Leider funktioniert dieses Verfahren nicht immer, und die Konsequenzen können ganz verschieden sein! Z.B., ein Spielprogramm, das zuerst einen kurzen Gruß auf den Bildschirm schreibt und sich dann verschiebt, wiederholt den Gruß anschließend (obwohl das im Programm nicht vorgesehen ist)! Ein anderes Programm, das sich verschiebt und dann mit DLOAD ein Typ "1" Datenblock in den Speicher liest, blieb nach dem Einlesen aus unerklärlichen Gründen stehen, bis ich auf die Idee kam, nach dem Verschieben einfach ein GOTO auf die nächste Basiczeile einzufügen! Seitdem läuft das Programm. Und schließlich ein kurzes Testprogramm (nur Verschieben, anschließend STOP) brachte den Rechner zum Absturz, wenn er gerade erst eingeschaltet worden war, aber lief beim zweiten Mal oder nach Reset einwandfrei! Den genauen Grund für dieses merkwürdige Verhalten kann ich nicht sagen, aber es scheint wirklich ein Problem von Basic, nicht von Ken-Dos, zu sein. Auf jeden Fall, Vorsicht bei obigem Verfahren, Basicprogramme zu verschieben!

Noch ein Paar Tips und Anmerkungen zu Ken-Dos Hardware: Ich habe gelegentlich gehört, und mit meinem eigenen Ken-Dos System und einem fremden DAI auch selber erlebt, daß Ken-Dos nach Einlöten des erforderlichen Drahts vom X-Bus zum DCE-Bus überhaupt nicht läuft. Kenneth Gooswit behauptet, daß die in manchen DAIs eingebauten 8255 ICs für Ken-Dos nicht geeignet sind, und daß das Problem durch Austausch dieses ICs gelöst werden kann. Ich weiß auch von einem Fall (dem System von Jean Marchand), wo ein nichtfunktionierendes Ken-Dos System nach Austausch des 8255 einwandfrei funktionierte. Man sollte aber beim Kauf von Ken-Dos auf dieses Problem achten und nach Möglichkeit das System vor dem Kauf mit dem eigenen Rechner ausprobieren und den 8255 notfalls umlöten lassen. Ein Paar weitere Tips: die Drähte, die für Ken-Dos Betrieb auf die DAI Platine gelötet werden müssen, sollten nicht unter das Netzteil geführt werden, sondern am besten außen herum--das reduziert die Störanfälligkeit. Und falls Rechnerausfälle und Störungen auftreten, kann es manchmal helfen, gesockelte ICs auf richtigen Sitz zu prüfen und den X-Bus Stecker und die Epromsockel mit Kontaktspray zu behandeln (anschließend vor Einschalten des Rechners die Lösungsflüssigkeit verdunsten lassen!). Die beiden letzten Maßnahmen haben auf jeden Fall geholfen, als bei mir der Rechner gelegentlich stehenblieb. Übrigens, Kontaktspray ist auch ein sehr wirksames Mittel gegen eine prellende Tastatur!!

In meinem letzten Testbericht über Ken-Dos habe ich bemängelt, daß es keine dokumentierte Möglichkeit gab, die Directory von einem Programm aus zu lesen--etwa eine Woche nach Redaktionsschluß erhielt ich eine neue, gedruckte Bedienungsanleitung, die auch über die interne Directorystruktur genaue Auskunft gibt. Übrigens haben meine Hardwareprobleme in der neuen Anleitung auch ihre Spur hinterlassen, in Form von sehr viel vorsichtiger gefaßten Empfehlungen zu den diversen Lötarbeiten an dem DAI, die für Ken-Dos Betrieb erforderlich sind.

Bochum, den 6. November 1984 Gordon Wassermann



Neues ROM IC72 (mittleres ROM)

Das ROMBP (EPROM 2764 mit Adaptersockel) ist grundsätzlich in zwei verschiedenen Versionen erhältlich:

1. deutsch 2. ASCII

Die beiden Versionen unterscheiden sich nur in der Tastaturbelegung (näheres siehe dort!).

Das EPROM ist ohne Probleme auch in Rechnern mit BASIC V1.0 einsetzbar.

Die Source-Listings der Änderungen sind auf Wunsch gesondert erhältlich. Der Source für den Grafik-Teil ist allerdings ohne viel Kommentar, benutzt aber die Label aus dem Firmware Manual.

Dieses ROM enthält die ROM-Banks 2 und 3.
Bank 2 enthält alle Bildschirm-Routinen für MODE 0 und die Grafik, sowie den Editor.
Bank 3 enthält die Compilier-Routinen des BASIC, die Tastaturdekodierung, die UTILITY, sowie die DCE-Initialisierungs-Routine.

Beschreibung der Änderungen gegenüber BASIC V1.1:

1. Bank 3:

1a) Compiling für MODE 7 und 8

Byte bei Adresse E1E0 von 07 auf 09 geändert, um das Compilieren von "MODE 7(A)" und "MODE 8(A)" zu erlauben.

```
E1DF  FE 09      CPI  :09
```

1b) Fehler bei IMP STR:

Bei IMP STR a-z trat bisher beim Compilieren der Fehler auf, daß für String-Variable 4 Bytes in der Symboltabelle reserviert wurden. Das führte z.T. zu Programmabstürzen. Dieser Fehler ist mit folgender Änderung korrigiert:

```
E60B  C3 AF E4   JMP  :E4AF

E4AF  7C         MOV A,H        ;VARIABLEN-TYP
E4B0  FE 20      CPI :20        ;=STRING?
E4B2  C2 0F E6   JNZ :E60F      ;ALLES OK, WEITER
E4B5  1E 02      MVI E,:02      ;2 BYTES RESERVIEREN
E4B7  C3 0F E6   JMP :E60F      ;OK, WEITER
```

1c) DCE-INIT:

Ein kleiner Fehler in der DCE-Init-Routine, der allerdings keine Auswirkungen hatte, ist korrigiert:

```
EFE4  32 03 FE   STA :FE03
```

1d) Neue Tastaturdekodierung mit CTRL-Taste.

Das gleichzeitige Drücken von CTRL und einer Buchstaben-Taste liefert nun den Code Buchstabe-#40, d.h. CTRL-A liefert 1, CTRL-Z liefert 26, und zwar unabhängig vom eingeschalteten Groß- Klein-Modus. Dieser wird jetzt durch SHIFT-CTRL umgeschaltet.
In der deutschen Version mit Umlauten werden bei der Umschaltung von Groß- auf Kleinschrift und umgekehrt auch die Buchstaben Ä,ö und ü mit umgeschaltet, weiterhin liefern CTRL-Ä,-ö,-Ü die Codes 27,28,29.

Die deutsche Tastatur sieht so aus:

```
...     )   ß   *   =   +   BREAK       (mit SHIFT)
...     9   0   :   -   ;   BREAK       (ohne SHIFT)

...       o   p   ü   TAB               (mit SHIFT)
...       O   P   Ü   RETURN            (ohne SHIFT)

...     k   l   ö   ä   ^   REPT        (mit SHIFT)
...     K   L   ö   Ä  DEL  REPT        (ohne SHIFT)
```

Die ASCII-Tastatur enthält nur einige Zusätze gegenüber der DAI-Standard-Tastatur:

```
SHIFT-CH DEL   = #5F  (Unterstreichung)
SHIFT-RETURN   = #1B  (ESCAPE)
SHIFT-0        = #40  (APE)
SHIFT-TAB      = #0C  (FORM-FEED)
```

Durch diese Änderung ist der Speicherbereich E935..E999 und EFF2..EFFF komplett anders belegt, die Tastaturmatrix ist natürlich immer noch bei E8C3..E934. Sie kann weiterhin wie üblich ins RAM verschoben werden. Hierbei ist allerdings zu beachten, daß das Byte für CTRL (Adr E8FA) unbedingt 0 sein muß!

1e) UTILITY

Die UTILITY heißt nun PC UTILITY V4.0, was darauf schliessen lassen soll, daß sie komplett überarbeitet wurde. Sie belegt, wie vorher, den Bereich #EA00-#EFBF. Hier ist keine Routine an ihrem alten Platz, d.h. da viele Programme zumindest die R- und W-Routinen benutzen, müssen diese geändert werden.
Die Adressen #50, #60 und #61 werden nicht mehr benutzt.

Die Eingabe erfolgt nun wie im BASIC, d.h. CHAR DEL ist möglich und jede Eingabe wird erst durch RETURN abgeschlossen oder durch BREAK ignoriert. Jedes Kommando ist genau ein Zeichen lang und zwischen den einzelnen Parametern können beliebig viele Leerzeichen stehen, jedoch außer zwischen Kommando und 1. Parameter immer mindestens eins. Werden als Eingabe Hex-Zahlen gebraucht, so sind (wie in der alten UT) immer nur die letzten 4 bzw. bei Bytes 2 Ziffern gültig. Neuer Prompt ist ')', weil die Sache jetzt etwas mehr abgerundet ist. Es gibt 3 verschiedene Fehlermeldungen:
SYNTAX ERROR (alle Syntax- oder Parameterfehler a2>a1)
LOADING ERROR (alle Ladefehler)
BUFFER OVERFLOW (falls 'bytes' mehr als 127 Bytes bei P,?,W,R)

PC UTLITY V4.0 Kommandos:

Zur Syntax: Parameter in Klammern sind optional.

a1	2-Byte-Adresse
b1	1-Byte-Wert
w1	2-Byte-Wert
x	eins der folgenden Zeichen: F A B D H S P
v	eins der folgenden Zeichen: 0 1 2 3 4 5 6 7
string	beliebige Zeichen zwischen zwei '"'
:word	2-Byte-Wert beginnend mit ':'
bytes	eine beliebige Mischung von byte, :word und string. (max. 127 Bytes) Z.B.: CD :DAFF :406 C9 "TEST" 0

B
Rückkehr ins BASIC

C
MODE 0, CLEAR SCREEN

Da1 (a2)
Hex-Dump von a1 bis a2 oder bis #FAFF.
Die Ausgabe kann mit BREAK abgebrochen oder durch eine beliebige Taste angehalten und mit SPACE fortgesetzt werden (wie LIST in BASIC).

Aa1 (a2)
Hex-dump mit ASCII von a1 bis a2 oder bis #FAFF. Um mit 60 Zeichen/Zeile auszukommen, werden je 4 Byte ohne Zwischenblank dargestellt, an einen Drucker wird es jedoch ausgegeben.

Fa1 a2 b1
Fülle a1 bis a2 (incl.) mit b1

Ma1 a2 a3
Verschiebe a1 bis a2 (incl.) nach a3

Z
Setze die Interrupt-Vektoren auf die Normalwerte, die Register auf 0 und den Stackpointer auf #F900. Vor G und L nicht mehr nötig!

#w1 w2
Gib die Summe und die Differenz der beiden Werte aus (w1+w2, w1- w2).

Sa1
Ändere den Inhalt von a1 und den folgenden Adressen. Dabei wird für jedes Byte eine neue Zeile benutzt. Nicht ändern=RETURN, Abschluß=BREAK

X(x)
Zeige die Registerinhalte oder ändere Register x und die folgenden (wie 'S'). Register C E und L existieren nicht mehr, d.h. B, D und H sind 2-Byte-Register

V(v)
Wie bei 'V' für die Interrupt-Vektoren. Die Vektoren M, T und G existieren nicht mehr, da sie sowieso nur Unheil anrichteten.

Pa1 bytes
Ersetze den Speicherinhalt ab a1 durch die angegebenen Bytes. Danach erscheint ')P' und die nächste nicht benutzte Adresse auf dem Schirm, sodaß mit der Eingabe direkt fortgefahren werden kann. Dies kann durch CHAR DEL weggelöscht oder durch BREAK unterbunden werden. Beispiel: 'P400 1 2 3 "ABC" (RETURN)', dann erscheint in der nächsten Zeile ')P0406 ' und man ist im Input-Modus.
Die Bytes werden zuerst in den Input-Buffer gelesen und erst danach im DI-Modus zur angegebenen Adresse verschoben, falls ein Fehler auftritt, ist also noch kein Byte gesetzt!

?a1 a2 bytes
Suche den Speicherbereich von a1 bis a2 (incl.) nach allen Vorkommen von bytes ab. Ein '?' in bytes dient als 'wildcard', d.h an der Position wird Gleichheit simuliert.
Z.B. ?400 B34F CD "?" EF sucht alle CALLs im Bereich #400-#B34F in den Bereich #EF00-#EFFF.
Die Adressen werden zu fünft pro Zeile ausgegeben (Pause und Stop wie üblich).

Wa1 a2 (a3) (string)
Schreibe den Speicherinhalt von a1 bis a2 (incl.) mit dem Filenamen string auf das 'eingeschaltete' Speichermedium. Ist a3 mit angegeben, so wird diese als Startadresse für Maschinenprogramme mit abgespeichert. Ist keine Startadresse angegeben, so ist der 1. Block beim Speichern (wie bei alten UT-Files) 2 Byte lang und enthält die Anfangsadresse des Speicherbereichs, andernfalls ist der 1. Block 4 Byte lang und enthält zusätzlich die Startadresse. File-Typ ist 'I', wie gehabt.
string ist eigentlich bytes, jedoch muss aus Syntax-Gründen zumindest am Anfang ein string sein, und sei es ein Leerstring.

R(wi) (string)
Lies den UT-File mit dem Namen string vom 'eingeschalteten' Speichermedium in den Speicher. Ist kein string (=Name) angegeben, so wird der nächste UT-File gelesen. Falls eine Startadresse enthalten ist, wird sie angesprungen. Durch RET kommt das Programm dann wieder in die UTILITY. Ist ein Offset (wi) angegeben, werden die Daten um wi Bytes höher eingelesen, als sie abgespeichert wurden. In diesem Fall wird das Programm natürlich nicht gestartet.

Anmerkung zu dem neuen File-Format: Alle Files können sowohl von der alten wie von der neuen UTILITY gelesen werden. Neue Files mit Startadresse werden natürlich von der alten UT nicht gestartet!

G(a1)
Ist eine Adresse angegeben, wird eine RET-Adresse zur UT auf den Stack gelegt, die Register geladen und zu a1 gesprungen, andernfalls werden die Register geladen und zu PC gesprungen. G ohne Startadresse sollte nur nach abgebrochenem LOOK oder zu 'never-comeback'-Programmen erfolgen! Ein Z-Kommando ist nicht mehr nötig!!

L(al) wl wZ oder L

Durchlaufe das Programm ab al oder PC im Einzelschritten. Ist die momentane Instruktion im 'Fenster' wl wZ, so werden die Registerinhalte ausgegeben. Beim ersten Aufruf müssen immer alle 3 Parameter angegeben sein, damit eine RET-Adresse auf den Stack gelegt wird und das Fenster definiert ist. Dei Ausgabe kann jederzeit durch einen Tastendruck angehalten oder durch BREAK abgebrochen werden. Danach kann mit L, Lwl wZ (neues Fenster) oder G fortgefahren werden. Der Interrupt-Vektor Ø und die Interrupt-Maske werden durch das Kommando automatisch gesetzt, 'Z' unnötig!

Bsp: I=0400 F=56 A=03 B=1234 D=0000 H=0500 S=F8F2 P=040i

2. Bank 2:

In der Bank 2 ist jetzt sicher kaum noch ein Byte auf seinem alten Platz, deshalb ist ein Umschreiben von Programmen, die auf diesen Bereich zugreifen, unbedingt erforderlich (dies betrifft vornehmlich Screencopy- und Textverarbeitungsprogramme).

Die RST-Vektoren sind selbstverständlich unverändert.

2a) Grafik-Routinen:

Die Firmware ist so geändert, daß nun

MODE 7, MODE 7A, MODE 8 und MODE 8A

direkt vom BASIC aus aufrufbar sind. Sie haben eine Auflösung von 512*242 Punkten und sind genau wie alle anderen Grafik-Modi mit DRAW, SCRN(), etc. benutzbar.

Zu beachten ist, daß das Verhältnis Höhe zu Breite der einzelnen Punkte 1.5 / 1 beträgt, d.h. Kreise werden zu Ellipsen.

2b) Text-Routinen:

Jedes druckende Zeichen, das über RST 5, DATA #03 (d.h. bei PRINT oder Direkteingabe) auf den Bildschirm soll, wird vorher mit dem Inhalt der Speicherstelle #00C8 'oderiert'. Beim Auslesen eines Zeichens aus dem Bildschirm mittels RST 5, DATA #15 (z.B. bei Direkteingabe) wird es mit #7F 'undiert'.

Dies dient dazu, um nach einer Hardware-Änderung, die den Einsatz von 256 verschiedenen Zeichen im Zeichengenerator erlaubt, auf einfache Weise auf den zweiten Zeichensatz zugreifen zu können.

Der zweite Zeichensatz wird nun mit POKE #C8,#80 erreicht.

Beim RESET wird #C8 automatisch auf Ø gesetzt. Andere Werte als Ø oder #80 können zu großen Problemen führen, die nur noch durch RESET zu beseitigen sind! Die Speicherstelle #00C8 war bisher vom Betriebssystem nicht benutzt; es ist jedoch nicht auszuschließen, daß sie von einzelnen Programmen benutzt wird.

2c) Editor:

Der Editor ist komplett neu geschrieben und enthält folgende Neuerungen, die in zwei Stufen beschrieben werden:

2cI) Normaler EDIT-Mode des BASIC:

Das Einfügen und Löschen von Zeichen ist jetzt bei großen Texten schneller. D.h. man kann auch noch editieren, wenn viel Text 'dahinter' ist.
Das Schreiben am Textende ist unabhängig von der Textlänge immer gleichschnell.
Jedes Zeichen, das von der Tastatur erreichbar ist, kann in den Text eingefügt werden (außer CHR$(0)!).
Zusätzlich zu den Cursor- und SHIFT-Cursor-Tasten gibt es nun einige Tasten, die das Editieren stark vereinfachen:
Hierbei bedeutet CYLIN die Anzahl der sichtbaren Zeilen im EDIT-Fenster (normal 24) und CXCHR die Anzahl der Zeichen pro Zeile (normal 60).

Taste	Code	Wirkung
CTRL-A	#01	Fenster um CYLIN Zeilen höher
CTRL-Z	#1A	Fenster um CYLIN Zeilen tiefer
CTRL-X	#18	Fenster um CYLIN Zeichen nach links
CTRL-C	#03	Fenster um CYLIN Zeichen nach rechts
CTRL-E	#05	Cursor um CYLIN/4 Zeilen höher
CTRL-D	#04	Cursor um CYLIN/4 Zeilen tiefer
CTRL-F	#06	Cursor um CXCHR/4 Zeichen nach links
CTRL-G	#07	Cursor um CXCHR/4 Zeichen nach rechts
CTRL-B	#02	Cursor ans Zeilenende, unabhängig ob er links oder rechts davon steht
CTRL-N	#0E	Setze ein Flag, daß das nächste ankommende Zeichen direkt, ohne Auswertung als Steuerzeichen, in den Text eingefügt wird.

2cII) Nutzung durch Maschinenprogramme:

Hierbei gibt es die Möglichkeit, das Fenster oder den Cursor beliebig weit in jede Richtung zu bewegen. Dazu dient die Routine CURWIN bei Adresse EC9C (Bank2!), die als Eingabe folgendes benötigt:

im Akkumulator muss der Code der Bewegungsart sein, z.B. #10 = Fenster hoch ... #17 = Cursor rechts; andere Werte führen zum Programmabsturz!
Im Registerpaar DE muss die gewünschte Schrittweite sein, bei Bewegungen in X-Richtung muss D=0 sein!
Die Routine ist jedoch nicht direkt aufrufbar, dies kann z.B. mit folgender Routine bewerkstelligt werden:

```
MCWL    MVI  A,CODE     ;BEWEGUNGSART
        LXI  D,ANZ      ;SCHRITTWEITE
MCW     PUSH B
        PUSH D
        PUSH H
        PUSH PSW        ;STACK KORRIGIEREN
        CALL :E392      ;CURSOR LÖSCHEN!
        JMP  :EC9C      ;CURWIN
```

Nach CALL MCW ist die Bewegung ausgeführt, der neue Textausschnitt auf dem Schirm und der Cursor blinkt an der richtigen Position. Ist bei der Rückkehr das Carry gesetzt, war alles ok, andernfalls ist eine Bewegung in die gewünschte Richtung nicht möglich.

Weiterhin kann man die Lage und Größe des EDIT-Fensters auf dem Bildschirm durch das Setzen bestimmter Werte beliebig bestimmen. Dazu sind folgende Speicherstellen zu initialisieren:

MCYLIN:	#00BB	Anzahl der Zeilen (mind. 2)
MCXCHR:	#00B9	Anzahl der Zeichen pro Zeile (mind. 2, aus Hardwaregründen maximal 64)
ECHS:	#00BA/BB	Adresse des ersten sichtbaren Zeichens oben links (bei 'Normal-EDIT': #BFE7)
ELINOF:	#00BC/BD	-(Offset vom letzten Zeichen einer Zeile zum ersten Zeichen der nächsten Zeile minus 2) (bei 'Normal-EDIT': #FFF2 =-14)

Die Editor-Initialisierung mittels RST 5, DATA #2A setzt die obigen Adressen für einen 60*24-Schirm. Will man eine eigene Lage und Größe, so sind die obigen Werte zu setzen und danach CALL EDIT1 (#EC0F) auszuführen, was den Text ab Anfang darstellt

Will man den Text ab einer beliebigen Position darstellen, so kann man dies durch CALL EDIT2 (#EC27) erreichen. Dazu müssen allerdings vorher EWINX, EWINY, ECURX, ECURY, TOFPT und CURLB sinnvoll gesetzt sein; der CURPT muß stimmen oder CURPT+1 muß 0 enthalten (=CURPT ungültig, wird neu berechnet). Nach dem Ändern der TAB-Positionen kann es zu Komplikationen kommen, deshalb sollte in diesem Fall der CURPT ungültig gemacht werden!

Beschreibung der benutzten RAM-Adressen:

EBUFR:	#00A2/A3	Zeiger auf den Anfang des Textes
EBUFN:	#00A4/A5	Zeiger hinter das letzte Zeichen des Textes (immer ein Null-Byte!)
EBUFS:	#00A6/A7	Zeiger auf das Ende des Puffers
EWINX:	#00A8	X-Position des äußersten linken Zeichens im Fenster bzw. Anzahl der Zeichen, die links nicht sichtbar sind
EWINY:	#00A9/AA	Y-Pos der obersten Schirmzeile
ECURX:	#00AB	X-Position des Cursors im Text
ECURY:	#00AC/AD	Y-Position des Cursors im Text
CURPT:	#00AE/AF	Zeiger auf die Cursor-Position im Text Falls (#AF)=0 ==> CURPT ungültig

CURLS:	#00B0/B1	Zeiger auf das erste Zeichen der Cursor-Zeile im Bildschirm Achtung: früher Zeiger auf Control-Byte
CURLB:	#00B2/B3	Zeiger auf das erste Zeichen der Cursor-Zeile im Text. Achtung: im Gegensatz zu früher jetzt immer gültig!
TABTP:	#00B4/B5	Zeiger auf die TAB-Tabelle. Die Fositionen müssen in aufsteigender Folge geordnet sein und mit #00 enden.

Neu ab EDIT V4.0:

TOPPT:	#00B6/B7	Zeiger auf den Anfang der obersten sichtbaren Zeile im Textpuffer
MCYLIN	#00B8	Anzahl der Zeilen des Fensters
MCXCHR	#00B9	Anzahl der Zeichen/Zeile des Fensters
ECHS	#00BA/BB	siehe oben!
ELINOF	#00BC/BD	siehe oben!
FLAG	#00BE	normal 0, =#FF: das nächste durch RST 5, DATA #2D übergebene Zeichen wird nicht als Steuerzeichen ausgewertet

Einige nützliche Beispiele für Textprogramme:

Cursor zum Anfang:
```
CALL EDIT1          ;(#EC0F) BANK 2 !!!!
```

zum Textende:
```
MVI A,15H           ;FENSTER RUNTER
LXI D,0FFFFH        ;MAXIMAL
CALL MCW
MVI A,5
RST 5
DATA 2DH            ;CURSOR 1/4 SCHIRM HOCH
DCR A               ;CODE 4
RST 5
DATA 2DH            ;WIEDER RUNTER
MVI A,2H            ;ZUM ZEILENENDE
RST 5
DATA 2DH
```

Dies könnte man bei tieferem Einstieg in die Routinen noch verbessern.

zum Zeilenanfang:
```
MVI A,12H           ;CURSOR LINKS
LXI D,0FFH          ;MAXIMAL
CALL MCW
```

auf nächste TAB-Pos:
```
PUSH B,D,H,PSW
CALL CURDEL         ;(#E392)
XRA A
STA CURPT+1H        ;(#00AF)
JMP E1C30           ;(#EFB8)
```

Programmierung in Assembler
Ein Kurs mit Fortsetzungen von Uwe Wienkop

In der Clubzeitung tauchen immer wieder Programme mit kleinen Assemblerunterroutinen auf. Aber auch gekaufte Programme sind sehr oft entweder ganz in Maschinensprache geschrieben oder sie verwenden Assemblerunterroutinen.
Gerade diese Routinen sind sehr oft das wichtigste am Programm und deshalb ist es schade, wenn man diese Art der Programmierung nicht versteht.
Außerdem werden Programmierer, die sich intensiv mit der Maschinenprogrammierung beschäftigen, oft von anderen mitleidig belächelt, ihre Programmiersprache wäre nicht mehr zeitgemäß usw.
Um diese Vorurteile abzubauen, und um auch Kenntisse in der Nutzung der vielen eingebauten Funktionen des DAIs zu geben habe ich beschlossen diese Reihe zu schreiben. Außerdem möchte ich auch noch zeigen, daß diese Art der Programmierung nicht nur sehr schnell ist, sondern zugleich auch nicht so schwierig ist, wie man immer hört und zu guter Letzt auch recht spannend sein kann.

Bevor nun aber mit der Beschreibung der Befehle begonnen werden kann, müssen zunächst noch ein paar theoretische Dinge behandelt werden.
Es ist notwendig ein wenig auf Zahlensysteme und Umrechnungsverfahren einzugehen:

0.1 Zahlensysteme und Umrechnungsverfahren:

Im DAI BASIC gibt es schon Zahlen in zwei verschiendenen Zahlensystemen: Einmal die normalen Zahlen zur Basis 10 und dann aber Hexadezimalzahlen. Diese Zahlen werden durch voranstellen eines '#' gekennzeichnet. Hexadezimal bedeutet, die Zahl wird zur Basis 16 dargestellt.
Die Basis einer Zahl gibt zum einen an, wieviel relevante Ziffern zur Verfügung stehen, um diese Zahl darzustellen. Im Dezimalsystem, mit dem wir gewöhnlich rechnen, sind dies ja bekannterweise 10 Ziffern (0-9).
Beim teilweise bekannten Hexadezimalsystem müssen also analog 16 Ziffern existieren. Mangels Kenntnis anderer Ziffern als 0-9 nimmt man dafür die Buchstaben A-F, so daß insgesamt hier die Zeichen 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F zur Verfügung stehen.

Beim Computer existiert zudem noch ein weiteres wichtiges Zahlensystem - man könnte fast sagen, es ist das wichtigste überhaupt. Es ist das Binärsystem. Hier gibt es somit nur die Ziffern 0 und 1!

Im BASIC kommt man meistens mit Zahldarstellungen im Dezimalsystem aus. Computer halten jedoch im allgemeinen sehr wenig von diesem Zahlensystem. Sie können intern nur zwei Zustände unterscheiden, z.B. Spannung hoch und Spannung unten. Aus diesen zwei Zuständen können aber durch Hintereinanderreihung größere Zahlen und durch abstraktere Betrachtung auch Zeichen und Worte entstehen.

Somit ist es notwendig jede Zahl in eine entsprechende Binärziffer umzuwandeln. Befehlsfolgen wie z.B. ein PRINT Befehl ist zunächst für einen Computer nur ein abstrakter Begriff mit dem er gar nichts anfangen kann. Erst nach entsprechender Konvertierung und späterer Bearbeitung durch ein Laufzeitsystem wird der Befehl für den Rechner verständlich.
Um jedoch diese Konvertierung und das spätere Arbeiten mit Zahlen besser verstehen zu können ist es sinnvoll schon jetzt die Umwandlungen durchzurechen.

Im Dezimalsystem hat jede Stelle eine entsprechende Wertigkeit, mit der die an dieser Stelle stehende Ziffer multipliziert wird. Aus der Summe der Ziffern ergibt sich schließlich der Wert dieser Zahl. Diese Wertigkeit bildet die andere Bedeutung der Basis: Die Basis mit ihrem entsprechenden Index potenziert ergibt die Wertigkeit. Dies hört sich sehr kompliziert an, ist jedoch ganz einfach. Als Beispiel:
Die Zahl 1243:

Die Indizierung der Ziffern beginnt bei der letzten Ziffer also der 3 und zwar mit Index 0! Die Vier erhält also den Index 1; Die 2 den Index 2 und die 1 den Index 3. Wie oben angeführt werden nun die Wertigkeiten als Potenz der Basis mit dem Index aufgeführt: Hier also

10^3 10^2 10^1 10^0
= 1000 100 10 1

Mit diesen Wertigkeiten werden nun die einzelnen Ziffern multipliziert:

1*1000 + 2*100 + 4*10 + 3*1 = 1243

Hier werden Sie sich jetzt sicher fragen, was soll das Ganze; Das wußte ich schon früher. Jedoch ist dieses Verfahren auch auf Ziffern in einem anderen Zahlensystem anzuwenden, z.B. um herauszufinden, welchen Wert eine Zahl, die zur Basis 16 dargestellt wurde, im Dezimalsystem besitzt.
Nehmen wir hier das Beispiel #BFEF - Diese Zahl gibt die oberste Bildschirmadresse an. Rechnet man nun diese Zahl gemäß dem obigen Verfahren um, so erhält man:

B*16^3	+	F*16^2	+	E*16^1	+	F*16^0	=	
B*4096	+	F*256	+	E*16	+	F*1	=	
11*4096	+	15*256	+	14*16	+	15*1	=	49135

Dies können Sie leicht vom BASIC aus durch PRINT #BFEF überprüfen, denn Print druckt Zahlen immer im Dezimalsystem aus und somit wird die Zahl sofort ins Dezimalsystem überführt und ausgegeben.

Nun wird der umgekehrte Fall betrachtet: Eine Zahl aus dem Dezimalsystem soll in ein anderes Zahlensystem überführt werden.
Das Verfahren hierzu ist auch recht einfach:
Man dividiert die Dezimalzahl ganzzahlig (Nachkommastellen fallen also weg) durch die gewünschte Basis und merkt sich den entstehenden Rest. Dieses Verfahren wird nun solange durchgeführt, bis als Ergebnis der Division nur noch eine Null entsteht. Wenn man nun die entstandenen Reste in umgekehrter Reihenfolge schreibt, so hat man die umgewandelte Zahl zur gewünschten Basis.
Auch hierzu ein Beispiel: Die oben entstandene Zahl 49135 soll ins Hexadezimalsystem überführt werden.

49135	: 16	=	3070	Rest	15	=	#F	
3070	: 16	=	191	Rest	14	=	#E	
191	: 16	=	11	Rest	15	=	#F	
11	: 16	=	0	Rest	11	=	#B	=> #BFEF

Genauso funktioniert auch die Umwandlung ins Binärsystem. Nur ist hier ja die Basis gleich 2: Beispiel 97:

97	: 2	=	48	Rest	1
48	: 2	=	24	Rest	0
24	: 2	=	12	Rest	0
12	: 2	=	6	Rest	0
6	: 2	=	3	Rest	0
3	: 2	=	1	Rest	1
1	: 2	=	0	Rest	1 => 1100001

Nun mag man sich vielleicht fragen, oben wurde gesagt das Binärsystem sei das wichtigste Zahlensystem für den Computer, warum aber wird man dann überhaupt noch mit der Kenntnis eines weiteren Zahlensystems - dem Hexadezimalsytem - belastet?

Die Antwort hierzu ist recht einfach:
Wie man aus den angeführten Beispielen leicht entnehmen kann sind Zahlen im Binärsystem für Menschen recht ungeeignet, da schon bei kleinen Dezimalzahlen recht lange Binärzahlen entstehen. Um diesem Mangel abzuhelfen wurden jeweils vier Binärziffern zu einer Ziffer zusammengefaßt. Man erhält somit eine Hexadezimalziffer. Diese Zahl ist also um den Faktor vier kürzer und somit überschaubarer:
z.B. die obige Zahl 1100001 = #61.

Aus diesem Grund werden im Utility des DAI Computers Register und Speicherinhalte auch in Hexadezimaler Form angegeben.

Das Rechnen im Hexadezimal, bzw. Binärsystem hat noch einen weiteren Vorteil. Durch den Ausbau des Speichers im Computer und dessen Abhängigkeit vom Binärsystem ergeben sich beim Rechnen im Hexadezimal und Binärsystem wesentlich rundere Werte als im Dezimalsystem.
Betrachten Sie hierzu folgendes Beispiel: Der Speicherausbau wird immer in Kilo Byte oder einfach K angegeben. Ein K=1024 Byte; $1024 = 2^{10} = \#400$ usw.

0.2 Informationseinheiten

Wie oben schon kurz angeklungen war, ist eine Binärziffer, die kleinste Informationseinheit; Sie wird auch mit Bit bezeichnet. Da man mit einem Bit jedoch nicht sehr viel anfangen kann, wurden 8 Bits zusammengefaßt. Dies ergibt ein Byte. Mit 8 Bits kann man entsprechend dem obigen Umrechnungsverfahren als größte Zahl die Zahl 255 darstellen.

$$1*2^7 + 1*2^6 + 1*2^5 + 1*2^4 + 1*2^3 + 1*2^2 + 1*2^1 + 1*2^0 = 255$$

Es ergeben sich also insgesamt 256 Möglichkeiten, da ja die Null auch noch mit berücksichtigt werden muß.

Die nächst größere Einheit ist ein 16 Bit Wort. Hier stehen also 16 Bit zur Verfügung, um eine Zahl darzustellen und die größte Zahl ist entsprechend $2^{16}-1 = 65535$.

Es gibt in Maschinensprache selbstverständlich auch negative Werte. Diese Zahlen werden im entsprechenden Abschnitt bei der Subtraktion behandelt.

0.3 Assembler und deren Funktionen:

Wie Sie vielleicht schon aus der Softwarebibliothek des Clubs ersehen haben bietet DAInamic Deutschland einen eigenen Assembler (AHT) an. Ähnliche Produkte sind auch von DAInamic Belgien unter den Namen DNA und SPL erhältlich. Diese Programme unterscheiden sich im wesentlichen durch ihren Bedienungskomfort bei Eingabe, Ausdrucken usw. Ihre Hauptaufgabe erfüllen sie jedoch alle gleich.

Wie oben schon erklärt wurde, müssen alle Befehle in eine binäre Form konvertiert werden. Da diese Form einen Befehl zu schreiben sehr unübersichlich ist - man muß ja die Zahl mit einem Befehl identifizieren - wurde eine Assemblersprache für Maschinenbefehle eingeführt. Der Assembler hat nun die Aufgabe einen verständlichen und lesbaren Befehl in diese Binärziffern zu überführen. Da allerdings der Befehlssatz der einzelnen gängigen Mikroprozessoren sehr stark variert und entsprechend von den Herstellern unterschiedliche Assemblersprachen eingeführt wurden, gibt es für jeden bekannten Prozessor eine eigene Assemblersprache mit sehr unterschiedlichen Befehlen.

Dies war ua. auch ein Grund dafür, daß die sogenannten höheren Programmiersprachen eingeführt wurden. Sie erleichtern nicht nur die Bearbeitung von Programmen, sondern sind auch zum größten Teil übertragbar. So kann man ein BASIC Programm, daß z.B. auf einem APPLE, ohne auf die spezifischen Eigenschaften dieses Computers einzugehen, geschrieben wurde relativ leicht auf einen anderen Computer übertragen.

Nun jedoch zurück zur Assemblersprache. Hier unterscheidet man zwischen zwei Programmarten:

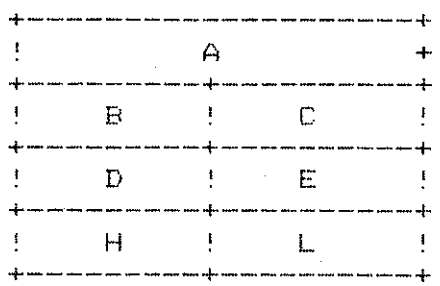
1) Sourcecode: Der Quelltext eines Programmes beinhaltet den lesbaren Programmtext, den Sie zur Erstellung eines Programmes eingegeben haben. Dieser Text kann dann wie im BASIC Editor korrigiert werden usw. Allerdings kann man diesen Text nicht direkt ausführen. Er muß erst assembliert werden. Dies führt zum
2) Objectcode: Dies ist also der assemblierte Quelltext und kann sofort ausgeführt werden. Der Objectcode hat den Vorteil, daß dieser Code gegenüber dem Sourcecode um einen Faktor von 3 oder 4 kürzer ist, jedoch kann dieser Code nur umständlich korrigiert werden, da die Befehle nun in übersetzter Form vorliegen. Sie können ja mal zum Spaß die Utility des DAIs aufrufen und von hieraus sich einmal den Bereich von #C000 bis #EFFF 'ansehen'. Hier befinden sich 24 Kilo Byte Objectcode.

Die Befehle des Sourcecodes werden auch Mnemonics genannt, denn sie dienen ja als 'Gedächnisstütze' für diesen entsprechenden Maschinenbefehl.
Ein Assembler hat zwar neben dem Übersetzen weitere wichtige Aufgaben, jedoch werden diese erst an entsprechender Stelle behandelt.

1 Variablen und Wertzuweisungen

1.1 Variablen auf Maschinenebene

Vom BASIC her ist man es gewohnt, einfach per INPUT Anweisung Variablen spezielle Werte zuzuweisen, diese dann zu verarbeiten und dann auf irgendeine Weise wieder auszugeben. Von diesem einfachen Schema muß man sich bei der Programmierung in Maschinensprache erst einmal loslösen, denn die Ein-/Ausgabe ist ein recht komplexes Thema. Aber eins haben beide Programmierungsarten gemeinsam. Zahlen werden in Variablen untergebracht. In der 8080 Maschinensprache stehen dem Programmierer im wesentlichen 7 Variablen zur Verfügung; Dies sind die internen Register der CPU. Diese Register wurden einfach mit einem Buchstaben als Namen belegt:



Das A Register nimmt hierbei eine herausragende Position ein. Es ist mit einer Einschränkung das einzige Register mit dem arithmetische Operationen durchgeführt werden können. Alle Operationen, wie Vergleichs- und Additions, bzw. Subtraktionsoperationen beziehen sich immer auf dieses Register. Es wird deshalb auch Akkumulator genannt. Die restlichen Register können sowohl einzeln als auch als Paar angesprochen werden. Jedes einzelne Register für sich betrachtet ist 8 Bit breit. Man kann jedoch auch zwei nebeneinanderliegende Register zu einem Registerpaar zusammenfassen um so einen größeren Zahlenbereich zu erhalten Die Registerpaare werden mit dem Buchstaben ihres ersten Registers angesprochen. Das Registerpaar BC also durch B, DE durch D und HL durch H. Die Unterscheidung, ob es sich um ein 8 oder 16 Bit breites Register handelt, wird durch entsprechende Befehle vorgenommen.

Alle anderen Register - abgesehen vom Akkumulator - sind relativ gleichwertig, nur wird das HL Register wegen einiger besonderer Möglichkeiten häufig zur indirekten Adressierung benutzt, jedoch hierzu unten mehr.

Da man in der Praxis wohl nur in den seltensten Fällen mit den internen Registern auskommen wird, besteht die Möglichkeit den Wert einer oder zwei Speicherzellen aus dem RAM des Computers zu lesen, bzw. in diese Register einen Wert zu schreiben. Somit können also sehr viele Variablen definiert werden.

Dies ist auch die bislang am meisten praktizierte Methode, um einem Maschinenprogramm vom BASIC aus Werte zu übergeben. Mit einem POKE Befehl wird in eine Speicherzelle ein Wert geschrieben. Diese Speicherzelle ist aber auch gleichzeitig eine der Variablen des Maschinenprogramms, so daß nach dem Aufruf des Maschinenprogramms der Wert hierfür bereit liegt.

1.2 Wertzuweisungen

Um jetzt überhaupt diese Register mit Werten belegen zu können gibt es mehrere Möglichkeiten: Eine davon ist:
Einem Register oder einem Registerpaar soll eine Konstante zugewiesen werden. In BASIC sähe dies folgendermaßen aus:
10 A=10 oder LET A=10
Bei der Maschinenprogrammierung gibt es einen ähnlichen Befehl:
MVI Register,Konstante, z.B. MVI A,10

Dies soll bedeuten, dem 8 Bit breiten A Register wird der Wert 10 zugewiesen. Es kann mit dem MVI Befehl jedem der oben angeführten Register ein Wert zugewiesen werden.
MVI steht übrigens für move immediate. Es wird also dem nach dem MVI Befehl angegebenen Register der unmittelbar nachfolgende Wert zugewiesen. Da es sich hier nur um eine Zahl ohne vorangestellte Kennung (für ein anderes Zahlensystem) handelt wird die Zahl als Dezimalzahl angenommen und natürlich automatisch beim Assemblieren in eine binäre Zahl überführt.

Bei den beiden Assemblern DNA und AHT werden Hexadezimalziffern im Gegensatz zum BASIC durch einen vorangestellten Doppelpunkt gekennzeichnet (!), z.B.
MVI E,:10
Im AHT Assembler ist auch die Verwendung von Binärziffern vorgesehen. Diese werden dann durch ein %-Zeichen gekennzeichnet
Also z.B. MVI B,%1011101

Analog kann einem 16 Bit breiten Registerpaar ein Wert zugewiesen werden. Nur muß hier auch eine 16-Bit Konstante folgen. Die Zuweisung geschieht mit dem Befehl:
LXI Registerpaar,Konstante, z.B. LXI B,2304
Ebenfalls wäre der Befehl LXI H,0 zugelassen, die Konstante ist zwar nicht 16 Bit breit, jedoch wird dies vom Assembler automatisch beim Übersetzen zu LXI H,0000 korrigiert.

Bitte beachten Sie an dieser Stelle den wichtigen Unterschied zwischen dem LXI und dem MVI Befehl. Dies soll noch einmal am Beispiel MVI B,5 und LXI B,5 durchgesprochen werden.
Der MVI B Befehl weist dem B Register den 8 Bit Wert 5 zu. Das nebenstehende C Register wird davon nicht beeinflußt. Der LXI-B Befehl weist jetzt allerdings dem BC Registerpaar einen Wert zu. Da hierbei das B Register das höherwertige Registe ist wird die 5 nicht ins B, sondern ins C Register geschrieben und das B Register wird gleichzeitig auf Null gesetzt:

	B	C
vorher z.B.	32	14
nach LXI B,5	00	05

	B	C
vorher z.B.	32	14
nach MVI B,5	05	14

Durch den LXI Befehl werden also immer zwei Register gleichzeitig verändert, während durch den MVI Befehl nur ein Register einen anderen Wert erhält.

1.3 Benutzen der externen Variablen

Oben wurde schon angesprochen, daß der gesamte zur Verfügung stehende Speicher für externe Variablen genutzt werden kann. Es ist einleuchtend, daß man die Speicherzellen nicht einfach wie die internen Register mit Namen wie A,B,C usw. ansprechen kann, da dies unter Umständen sehr viele Register sein könnten. Deshalb werden die externen Register mit der Adresse angesprochen, mit der sie im Computer organisiert sind, z.B. vom BASIC aus sprechen Sie mit einem POKE #300,x die Adresse #300 an. Genauso wird dies auch mit den externen Variablen gemacht.

Es stehen dem Programmierer im wesentlichen nur zwei Ein-/Ausgaberegister zur Verfügung. Dies ist zum einen der Akkumulator und zum anderen das HL Registerpaar: Mit den Befehlen LDA Adresse und STA Adresse kann der Wert, der durch die Adresse beschriebenen Speicherzelle ins A Register geladen werden oder aber vom A Register in diese Speicherzelle abgespeichert werden. LDA steht als Abkürzung für LoaD Accu Direct und STA für STore Accu direct.

Beispiel:

#300

!	!	!	!	!#10	!	!	!	!	!	!

Dies soll einen Ausschnitt des Speichers darstellen:

LDA :300 (Bitte beachten Sie die Doppelpunkte stehen im AHT und DNA Assembler für Hex Ziffern!)
Dadurch wird ins A Register der Wert der Speicherzelle #300 gelesen. Der neue Wert im Akkumulator ist nun also #10=16

Durch den Befehl STA :301 wird nun ein Wert in die Speicherzelle #301 geschrieben. Z.B:

```
MVI   A,34 = #22
STA   :301
```

#300 #301

!	!	!	!	!#10	!#22	!	!	!	!	!

Oder als weiteres Beispiel:

```
LDA   :301    Der Wert von #301 wird in den Akku geladen
STA   :300    und sofort wieder nach #300 abgespeichert
MVI   A,10    Dem Akku wird die Konstante 10=#0A zugewiesen
STA   :301    Diese wird dann nach #301 abgespeichert
```

#300 #301

!	!	!	!	!#22	!#0A	!	!	!	!	!

Als zweite Möglichkeit den Wert externer Variablen zu lesen oder aber einen Wert in diesen abzuspeichern ist das HL Register. Der Aufruf dieses Befehls verläuft analog zu den LDA/STA Befehlen. Nur wird das HL Register nicht mit dem Wert einer Adresse - wie bei LDA - sondern mit den Werten zweier aufeinanderfolgender Adressen geladen, denn beim HL Registerpaar handelt es sich ja um zwei Register, bzw. ein 16 Bit breites Registerpaar.

Die Befehle hierzu lauten LHLD und SHLD. LHLD steht für Load HL-register Direct und SHLD entsprechend für Store HL-register Direct.

Hier muß allerdings auf eine Besonderheit der 8080 CPU hingewiesen werden: Die Adressierung erfolgt in low-high order. Dies bedeutet, daß das niederwertige Byte einer 16 Bit Zahl zuerst und dann erst das höherwertige Byte abgespeichert würde. Man könnte die Befehle ja in zwei Teilschritte unterteilen:
Hier wird demnach also zuerst das L-Register nach Adresse abgespeichert und dann erst das H-Register nach Adresse+1 gebracht. Auch hierzu ein Beispiel:

#300 #301 #303 #304

!	!	!	!	!	!	!	!#34	!#12	!	!

```
LXI   H,:BFEF   Dem HL-Registerpaar wird der Wert #BFEF
                zugewiesen
SHLD  :300      Dieser Wert wird dann in #300 und #301
                abgelegt
```

```
                          #300 #301       #303 #304
+----+----+----+----+----+----+----+----+----+----+----+
!    !    !    !    !#EF !#BF !    !#34 !#12 !    !    !
+----+----+----+----+----+----+----+----+----+----+----+
```

LHLD :303 Ins HL-Registerpaar werden die Werte aus den Registern #303 und #304 gelesen HL= #1234 = 4660

Mit diesen Befehlen sind auch schon die Möglichkeiten der 8080 CPU in Bezug auf direktes Ansprechen von Speicherzellen erschöpft. Aber es gibt noch eine weitere Adressierungsart, die weiter unten beschrieben wird.

1.4 Befehlslängen:

Wie Sie vielleicht schon aus den ersten Befehlen ersehen haben, unterscheiden sich die Befehle hinsichtlich ihrer Struktur z.T erheblich: Dem MVI Befehl folgt eine 8 Bit Konstante dem LXI eine 16 Bit Konstante und der XCHG hat überhaupt keine nachfolgenden Werte.

Tatsächlich gibt es bei der 8080 CPU drei unterschiedliche Befehlslängen: ein, zwei oder drei Bytes lang. Der XCHG und die MOV (s.u) Befehle sind 1 Bytes lang (In den MOV Befehlscode werden die nachfolgenden Registerangaben gleich mit hineincodiert, so daß auch dieser Befehl nur ein Byte lang ist).

Die MVI Befehle benötigen zwei Bytes. Je eines für den Befehl und eins für die nachfolgende Konstante.

Die LXI, Lade und Abspeicherbefehle sind jeweils 3 Bytes lang, denn nach dem Befehlscode folgt eine 16 Bit lange Konstante, bzw. es folgt eine ebensolange Adresse.

Diese Angaben brauchen Sie sich jetzt noch nicht unbedingt zu merken, sie sind jedoch für das Verständnis des nachfolgenden Kapitels von Vorteil.

1.5 Anmerkungen zur Verwendung von externen Variablen und Labeln

Wie vielleicht schon aus den einfachen Anwendungsbeispielen ersichtlich wurde, ist es recht mühsam, sich die Bedeutung einer Adresse zu merken, vor allem, wenn mit größeren Programmen auch die Anzahl der Variablen steigt. Hier bietet der Assembler eine wesentliche Erleichterung: Es ist möglich dem Assembler einmal im Programm mitzuteilen, daß die Adresse X z.B. die Bedeutung 'Länge' haben soll. Wenn jetzt nun in einem Programm hinter den LDA/STA oder LHLD/SHLD Befehlen eine Adresse benötigt wird, so kann man nun z.B. LDA Länge schreiben, so daß unmittelbar aus dem Befehl die Bedeutung klar wird. Beim Assemblieren wird dann Länge wieder durch die ihm früher schon zugewiesene Adresse ersetzt.

Man bezeichnet diesen Namen Länge als Label, denn es steht ja als Synonym für eine ihm früher schon zugewiesene Adresse.

Diese Label erhalten später noch einmal eine noch wichtigere Rolle beim Programmieren von Schleifen und Unterprogrammen. Auch hier können nur Absolutadressen verwendet werden. Diese haben den Nachteil, wenn ein Programm verschoben werden soll müssen alle Adressen neu berechnet werden. Verwendet man allerdings Label, so muß nur die Startadresse geändert und das Programm neu assembliert werden und schon ist es wieder ablauffähig!

Nach diesen Vorbemerkungen zu den Label können wir auf die Struktur eines Assemblerprogramms eingehen:
Ein 8080 Assemblerprogramm wird grundsätzlich in vier Spalten unterteilt:
Label ! Mnemonics ! Daten ! Kommentar
Diese Struktur ist im gesamten Programm einheitlich. Wird jedoch eine Spalte nicht verwendet, so bleibt sie leer.
Unsere vorangegangen Beispiele benutzten keine Label, dies wird sich jedoch ab sofort ändern, da ein Programm hiermit wesentlich übersichtlicher wird.

Im AHT z.B. werden externe Variablen auf die folgende Weise definiert:

```
Anfang    RES   1
ENDE      RES   1
Länge     RES   2   usw.
```

Das RES steht für Reserve: Es werden somit für Anfang und Ende jeweils 1 Byte und für Länge zwei Byte reserviert. Es ist zu beachten, daß diese "Mnemonics" keine eigentlichen Befehle sind. Sie geben nur dem Assembler die Anweisung die nächsten Bytes entsprechend der Zahl nach dem RES freizuhalten - Aus diesem Grund werden diese Befehle oft auch Pseudomnemonics genannt.

Im "Standard"-Intel Mnemonic Format haben diese Befehle einen anderen Namen. Hier werden sie als DB und DW bezeichnet. DB steht für Define Byte und DW für Define Word.

Es ist sinnvoll an dieser Stelle noch auf zwei weitere Pseudo-mnemonics einzugehen: ORG und EQU

ORG legt den Anfang des Programms fest (ORG = origin). D.h. alle nach diesem ORG folgenden Befehle werden beim Übersetzen fortlaufend in den Speicher ab dieser Startadresse geschrieben.

```
          ORG   :400
          LDA   Anfang
          MVI   A,4
          MVI   C,7
            :
```

Hiermit wird beim Assemblieren der codierte LDA Befehl nach #400 geschrieben. Nun folgen zwei Bytes für die nachfolgende Adresse. Diese werden nach #401 und #402 geschrieben und der MVI A,4 Befehl kommt nach #403 usw.

Als ORG kann eigentlich jede beliebige Adresse gewählt werden, jedoch werden als ORG häufig die Adressen #300 oder #400 gewählt, da sich bei diesen Adressen ein BASIC Programm leicht hinter das Maschinenprogramm schieben läßt und somit auch das Maschinenprogramm gegen ungewolltes Überschreiben schützt. Die Adresse #400 hat noch einen weiteren Vorteil: Selbst nach einem RESET bleibt das Programm erhalten und kann wieder gestartet werden!

Der EQU (=equal) Befehl weist einem Label - oder richtiger wäre es in diesem Fall von Konstante zu sprechen - einen Wert zu: Bsp:
PRINT EQU :DAD4

Diese Befehle Mnemonics, Pseudomnemonics, Startadressen usw mögen momentan noch ein wenig verwirrend klingen, zumal Sie mit diesen Dingen noch recht wenig anfangen können. Jedoch ist es manchmal notwendig ein wenig vorzugreifen, um nachfolgende Befehle leichter erklären zu können.

2 Einfache Verschiebebefehle

Im LDA/STA Beispiel des vorangegangenen Kapitels wurde gezeigt, wie man den Wert einer Speicherzelle auf eine andere verschieben kann. Bei diesem Verfahren wird natürlich der Wert, der sich vorher in der Zieladresse befand, überschrieben.

Für den relativ häufig auftretenden Fall, daß man einen Wert von einem Register (hier sind jetzt die sieben CPU Register gemeint) in ein anderes bewegen möchte, gibt es die Verschiebe und Austauschbefehle:

Die Verschiebebefehle heißen dementsprechend auch **MOV** für move und sie schieben den Wert von einem Register in ein anderes. Der MOV Befehl hat folgende Struktur:
MOV Register,Register; z.B. MOV A,B MOV H,L MOV C,E usw. Hier wird der Wert vom zweiten angeführten Register in das erste Register verschoben. Der Wert des ersten Registers wird damit überschrieben, aber der Wert des zweiten Registers bleibt erhalten!

Z.B.: MOV A,B bedeutet: Der Akkumulator wird mit dem Wert des B Registers überschrieben und B bleibt erhalten. Dies ist mit dem LET Befehl des BASIC vergleichbar:

```
LET  A=B
MOV  A,B
```

Es können die Werte zwischen allen Registern hin und hergeschoben werden. Es gibt somit 7 x 7 = 49 Kombinationsmöglichkeiten!

Bitte beachten Sie: Der MOV Befehl verschiebt immer nur 8 Bit große Daten. Es ist nicht möglich ein ganzes Registerpaar zu verschieben, z.B. DE nach BC oä. Dies muß dann durch zwei MOV Befehle gemacht werden; also

```
MOV  B,D
MOV  C,E
```

Ein wichtiges Anwendungsbeispiel für den MOV Befehl wird aus der Beschränktheit auf externe Speicherzellen zugreifen zu können deutlich. Man kann ja einen 8 Bit Wert nur in den Akku laden. Benötigt man diesen Wert jedoch im B Register, so muß dieser Wert dann noch durch MOV B,A dorthin verschoben werden.

Neben den MOV Befehlen gibt es noch einen Austauschbefehl: Der XCHG = exchange. Hiermit werden die Inhalte der Registerpaare (!) HL und DE ausgetauscht. Also hat HL nun den Wert von DE und umgekehrt. Hierbei gehen also keine Werte verloren.

3 Additions und Inkrement Befehle

3.1 Die normale 8-Bit Addition

Wie Eingangs erwähnt wurde besitzt der Akkumulator eine herausragende Bedeutung unter den internen Registern. Nur in Verbindung mit diesem Register können Additionen und Subtraktionen durchgeführt werden. Einer der Operanden befindet sich also schon im Akkumulator. Der andere Operand wird durch eine Registerangabe nach dem Additionsbefehl ADD spezifiziert. Der vollständige Befehl lautet also

    ADD Register, z.B. ADD B

Dies bedeutet der Inhalt des Registers B wird zu dem Inhalt des Registers A aufaddiert und das Ergebnis befindet sich nach der Ausführung im Register A; Register B wird nicht verändert. Die Addition ist mit allen Registern möglich. Der Befehl ADD A ist ebenfalls zugelassen: A=A+A = 2*A

Außerdem ist es noch möglich eine Konstante zum Inhalt des A Registers aufaddieren. Dies wird mit dem Befehl ADI (Add immediate) durchgeführt. Z.B: ADI 10 ==> A=A+10

Nun ist es allerdings unbefriedigend nur eine Addition auszuführen: Manchmal ist es wesentlich herauszufinden, ob das Ergebnis überhaupt noch gültig ist, z.B. im Akku befindet sich die Zahl 200. Die Addition ADD A gibt als Ergebnis 400. Diese Zahl ist aber in 8 Bit überhaupt nicht mehr darstellbar. Um dies abtesten und darauf reagieren zu können wurde der Registersatz der CPU noch um ein Flag Register erweitert. Dieses Register ist allerdings nicht mit den anderen Registern zu vergleichen. Hier kann man keine Werte hineinschreiben. Dieses Register liefert nur Werte für Abfragen wie Bereichsüberlauf, Zahl gleich Null, Zahl negativ usw. Von den 8 Bit dieses Registers sind nur 5 belegt!

Diesees Bedingungsregister hat folgendes Aussehen

Bit	7	6	5	4	3	2	1	0
	N	Z		H		P		C

Die Bitpositionen 1, 3, und 5 sind nicht bestimmt.

Bedeutungen: (1=gesetzt; 0=ungesetzt)

N=0 : Vorzeichen positiv
N=1 : Vorzeichen negativ
Z=0 : Ergebnis nicht Null (Zero)
Z=1 : Ergebnis gleich Null
H=0 : Kein Halbübertrag von Bit 3 nach Bit 4
(siehe hierzu den Befehl DAA)
H=1 : Halbübertag von Bit 3 nach Bit 4
P=0 : Summe aller 1er Bits ungerade (Parität)
P=1 : Summe aller 1er Bits gerade
C=0 : kein Übertrag (Carry)
C=1 : Übertrag

Sie können sich den Wert dieses Registers übrigens ausgeben-lassen, wenn Sie die Utility des DAIs aufrufen und hier X Return eingeben. Wenn Sie das F (=Flag) Register in binärer Form schreiben, können Sie das Ergebnis einer Operation sofort ablesen.
Die bisher in den Kapiteln 2 und 3 beschriebenen Befehle beeinflußten dieses Bedingungsregister nicht!

Nun zurück zum Beispiel: Die Addition 200+200 würde als Ergebnis 144 liefern und das Carry Bit wäre gesetzt. Dies bedeutet der Zahlenbereich von 255 wurde überschritten. An einer Rechnung wird dies deutlich:

```
        200 = #C8 =    1100 1000
                      +1100 1000
                      ----------

         Wert   ---> 1 1001 0000  = 1*2^7 + 1*2^4 = 144
       für das         --8 Bit--
    Carry Register
```

Der entstehende Übertrag wird abgeschnitten und nur die letzten 8 Bit werden betrachtet und der Übertrag wird ins Carry Register geschrieben. Das Carry Register kann nun abgefragt werden und gegebenfalls wird eine Fehlerbehandlung durchgeführt. Jedoch wird nicht automatisch wie im BASIC die Bearbeitung abgebrochen und die Meldung NUMBER OUT OF RANGE ausgegeben!

3.2 Addition unter Berücksichtigung des Carrys

Bei der gewöhnlichen Addition durch ADD wird keine Rücksicht auf den vorherigen Stand des Carry Registers genommen. Um auch den Übertrag des alten Ergebnisses mit in die Rechnung einbeziehen zu können gibt es den Befehl ADC (add with carry) Hier wird je nach Stand des Carry Flags noch eine 1 aufaddiert oder nicht!
Ein Standardbeispiel hierfür ist die Addition der Registerpaare BC und DE - das Ergebnis soll sich nachher in DE befinden:

```
(1)      MOV   A,E     Der Inhalt von E wird nach A gebracht
(2)      ADD   C       Hierauf wird C (ohne Berücksichtigung
                       des Carrys) aufaddiert, denn dies ist
                       ja der Anfang der Rechnung
(3)      MOV   E,A     Das Ergebnis wird nach E gebracht
(4)      MOV   A,D     Nun erhält A den Wert von D
(5)      ADC   B       Hierauf wird B unter Berücksichtigung
                       des Carrys aufaddiert
(6)      MOV   D,A     Das Ergebnis der zweiten Addition
                       wird nach D geschoben
```

Zur Verdeutlichung noch ein Beispiel:

```
+------+------+
!  B   !  C   !      1)  A = E = #C8
+------+------+      2)  A = A+C= #C8+#C8 = #90 ; Carry gesetzt
! #04 ! #C8  !      3)  E = A = #90
+------+------+      4)  A = D = #14
!  D   !  E   !      5)  A = A+B+Carry = #04+#14+1 = #19
+------+------+      6)  D = A = #19
! #14 ! #C8  !      +------+------+
+------+------+      !  D   !  E   !
          ===>      +------+------+
                    ! #19  ! #90  !
                    +------+------+
```

Auch bei dem ADC Befehl gibt es eine entsprechende Immediate Form: Dieser Befehl heißt ACI und addiert auf den Akku eine Konstante Zahl unter Berücksichtigung des Carrys.
Durch einen ADD oder ADC Befehl werden alle Flags beeinflußt, also nicht nur das Carry Flag!

3.3 Inkrement und Dekrement - Befehle

Zusätzlich zur Addition besteht beim Programmieren oft der Wunsch den Inhalt eines beliebigen Registers um eins zu erhöhen oder zu vermindern - man denke etwa an eine Bedeutung als Schleifenvariable wie in einer FOR Schleife im BASIC. Hierbei wäre es natürlich sehr umständlich erst den Inhalt des Registers in den Akku zu bringen, dann einen ADI 1 durchzuführen und anschließend alles wieder zurückzuschieben. Für solche Anwendungen gibt es die Befehle INR (=increment register) und INX (inkrementiere ein Registerpaar). Entsprechend hierzu existieren auch die Befehle DCR (decrement register) und DCX (dekrementiere ein Registerpaar).
Diese Befehle verändern im Flag Register bis auf das Carry Flag alle anderen Flags. Es kann also auf Null oder Minus abgefragt werden.
Beispiele:

```
DCR  B      B = B-1
DCX  B      BC=BC-1
INR  H      H = H+1
INX  H      HL=HL+1
DCR  A      A = A-1
INR  A      A = A+1
```

3.4 Der DAD Befehl:

Anhand des Beispiels DE = DE + BC wurde vielleicht schon einmal die Bedeutung von längeren Zahlen mit einer Breite von 16 Bit deutlich: Man kommt zwar häufig bei Schleifen mit Schleifenvariablen bis 255 aus, jedoch muß auch sehr oft auf größere Zahlen zurückgegriffen werden. Wenn diese dann jedesmal über die Sequenz von MOV und ADD Befehlen addiert werden müßten, würde dies einen nicht unerheblichen Aufwand bedeuten. Für solche Anwendung gibt es den Befehl DAD (=double add). Hier wird auf den Inhalt des HL (!) Registerpaars der Inhalt eines anderen Paares aufaddiert. Das Ergebnis befindet sich wieder in HL.

```
Beispiel:   DAD   D     HL=HL+DE
            DAD   B     HL=HL+BC
            DAD   H     HL=HL+HL = 2*HL
```

3.5 Der DAA Befehl

Neben der Codierung der Dezimalzahlen als Binärziffern gibt es noch eine andere Darstellungsweise. Man denke sich ein 8 Bit Register in zwei 4 Bit Register aufgeteilt. Mit vier Bit kann man einen Zahlenbereich von 0 bis 15 darstellen. Von diesen 16 Kombinationsmöglichkeiten werden nun nur 10 verwendet. Der Rest ist "Verschnitt". Insgesamt werden in einem Register also 2 Dezimalzahlen dargestellt. Man nennt diese Darstellungsform deshalb auch BCD Code (= binary coded decimals).
Dieser Code hat bei Rechnung mit reellen Zahlen erhebliche Vorteile, da hierbei keine Rundungsfehler beim Konvertieren von einem Zahlensystem in ein anderes auftreten können. Dieser Vorteil wird jedoch mit einem höheren Speicherbedarf erkauft - von 256 Möglichkeiten pro Byte werden nur 100 genutzt!

Die 8080 CPU unterstützt die Rechnung mit BCD Zahlen nur recht unzureichend. Es gibt hierfür keine speziellen Befehle für Addition und Subtraktion, allerdings wird beim Addieren mit ADD und ADC das H-Flag verändert. Dies zeigt einen Übertrag von der 3. auf die 4. Stelle an. Hiermit ist es möglich das Additionsergebnis so zu korrigiern, daß das Ergebnis wieder eine BCD Zahl ist, daher auch der Name des Befehls: DAA (=decimal adjust accumulator)

```
Beispiel: Addieren der BCD Ziffern aus den Register B und C
          Das Ergebnis befindet sich danach wieder in B!
          MOV   A,B     Zahl von B nach A bringen
          ADD   C       C aufaddieren
          DAA           Das Ergebnis entsprechend dem BCD-Code
                        korrigieren
          MOV   B,A     Das BCD Ergebnis nach B bringen
```

4 Subtraktion

4.1 Darstellung von negativen Zahlen in Binär-Form

Bis jetzt wurde bei allen Operationen davon ausgegangen, daß wir mit Zahlen in einem Darstellungsbereich von 0 bis 255 bei einer Breite von 8 Bit gearbeitet haben. Aber schon bei einer so einfachen Rechnung wie 1-2 = ??? muß man sich fragen, wie soll die -1 dargestellt werden. Hierfür unterteilt man den zur Verfügung stehenden Zahlenbereich und man nimmt das höchstwertige Bit als Vorzeichenindikator. Ist jetzt dieses Bit gesetzt, so handelt es sich um eine negative Zahl, ansonsten wird angenommen, die Zahl wäre positiv. Allerdings beginnt man nicht noch ein zweites Mal bei der Aufzählung mit der Zahl 1, sondern genau andersherum. Dies wird an der folgenden Rechnung deutlich:

```
      00000001      1
    - 00000010    -2
    ==========
    1 11111111  - Ergebnis, d.h. die Zahl 255 wird als -1
  /               betrachtet
Carry
Flag
```

Diese Art der Rechnung bezeichnet man als Rechnen mit 2er Komplement:
Ein Zahl kann entsprechend dem folgenden Verfahren in eine negative Zahl überführt werden. Man "drehe" alle Ziffern um, jedes Bit wird also invertiert. Auf das entstandene Ergebnis wird noch die Eins aufaddiert.

```
Beispiel:      3  =  00000011
          invertiert 11111100
              +1  => 11111101    =  -3
```

Der Zahlenstrahl stellt sich somit wie folgt dar:

```
    -3        -2        -1         0         1         2
-----+---------+---------+---------+---------+---------+---------->
  11111101 11111110 11111111 00000000 00000001 00000010
```

bzw. der andere Übergang von plus nach minus

```
    126       127      -128      -127      -126      -125
-----+---------+---------+---------+---------+---------+---------->
  01111110 01111111 10000000 10000001 10000010 10000011
```

Zusammenfassung dieser Erkenntnisse:
Beim Rechnen zeigt also nach einem Subtraktionsbefehl ein aufgetretenes Carry Flag einen Bereichsunterlauf an, d.h. die Zahl ist negativ. Das Vorzeichen des Ergebnisses kann ebenfalls noch über das N-Flag getestet werden. Ist dieses Flag gesetzt, so ist das Ergebnis negativ!

4.2 Die Subtraktionsbefehle

Nach den Vorbemerkungen von Abschnitt 1 kommen wir nun zur eigentlichen Subtraktion. Diese wird durch den Befehl SUB Register aufgerufen.
Auch hier wird wieder das A Register als Rechenregister verwendet und alle anderen Register können als Operanden auftreten:
Beispiel: SUB B => A=A-B
Außerdem gibt es ebenfalls eine Immediate Form dieses Befehls:

```
SUI Konstante  =>  A=A-Konstante
```

Analog zum ADD Befehls existiert auch für den SUB Befehl noch eine Befehlsform, die ein evtl. aufgetretenes Carry aus einer früheren Rechnung berücksichtigt: SBB (sub with borrow).

```
SBB H           => A=A-B-Carry
SBI Konstante   => A=A-Konstante-Carry
```

Die Notwendigkeit des SBB wird am Beispiel der Subtraktion zweier 16 Bit Zahlen offensichtlich. Hierfür gibt es keinen speziellen Befehl der 8080 CPU!

```
Beispiel:   HL=HL-DE
   (1)      MOV   A,L
   (2)      SUB   E      Die erste Subtraktion darf keine Rück-
                         sicht auf das Carry nehmen
   (3)      MOV   L,A
   (4)      MOV   A,H
   (5)      SBB   D      Beim zweiten Mal muß jedoch ggf. das
                         Carry noch mit abgezogen werden, denn
   (6)      MOV   H,A    das würde bedeuten E>L

Rechnung: Beispiel:      HL=#321 und DE = #231
   1)   A = 0010 0001
          - 0011 0001
        ==============
   2)     1 1111 0000  = -#10  = -16  ==> L-Register; Carry!
   4)   A = 0000 0011
          - 0000 0010
          - 0000 0001 <- Carry
        ==============
            0000 0000  => H

      HL = 0000 0000 0001 0000   =   16
```

4.3 Die Vergleichsbefehle:

Eine Subtraktion mit den hierbei gelieferten Statusflags des Bedingungsregisters kann als Vergleich aufgefaßt werden. Dies verdeutlicht folgende Umformung:

```
A < B ???  <=>  A-B < 0  ???
```

Man könnte somit den Vergleich auf die Subtraktion zurückführen und z.B. mit Abfrage auf Carry abtesten, ob A denn nun größer als B war oder nicht, denn ist A kleiner als B, so ist die Differenz negativ und damit ist auch das Carry gesetzt.

Für die Abfrage A=B könnte man dies ebenfalls benutzen, denn dies würde bedeuten A-B=0 und im Falle der Gleichheit wäre das Zero Flag gesetzt usw.

Dieses Verfahren hat nur den Nachteil, daß der Wert des Akkumulators zerstört werden würde, da sich ja nun im Akku das Ergebnis der Subtraktion befinden würde - was natürlich nicht immer erwünscht ist.

Aus diesem Grund gibt es dem CMP (compare) Befehl. Dieser arbeitet genauso wie die Subtraktion und beeinflußt damit auch die Flags, jedoch wird der Akku nicht mit dem Ergebnis überschrieben, sondern bleibt erhalten. Es werden somit nur die Flags verändert!

Natürlich hat auch dieser Befehl seine Immediate Variante.
Diese lautet CPI Konstante
Der Akku wird also in diesem Fall nur mit einer Konstante verglichen, während er beim CMP Befehl mit einem nachfolgend aufgeführten Register verglichen wird, z.B. CMP H

Anmerkung:
Die 8080 CPU besitzt keine weiteren arithmetischen Befehle, wie z.B. Multiplikation oder Division, von weitergehenden Operationen ganz zu schweigen. Diese Funktionen müssen dann durch geeignete Programme aufgebaut werden!

5 Die Sprungbefehle

Die bisherigen Beispiele waren gezwungenerweise nicht immer sehr sinnvoll, da wohl jeder eine Addition oder Subtraktion leicht in BASIC durchführen würde, bevor er sich an die Arbeit machen würde und dieses Programm in Assembler eingibt. Die Bedeutung der Maschinensprache liegt ja vielmehr in ihrer wesentlich höheren Geschwindigkeit und in der Möglichkeit auf Hardwarestrukturen wesentlich besser zugreifen zu können, als dies vom BASIC aus möglich wäre.
Nun sind allerdings die wenigsten Programme, die auf einem Computer laufen, linear geschrieben, d.h. sie würden keine Wiederholungen oder Verzweigungen benutzen.
Um derartige Programmstrukturen aufbauen zu können verwendet man Sprünge.

5.1 Unbedingte Sprünge:

Diese Strukturen lassen sich gut mit den im BASIC implementierten GOTOs und IF GOTOs vergleichen. Das GOTO bildet einen unbedingten Sprung zu einer Programmzeile, d.h. das Programm wird nach Auftreffen auf den GOTO Befehl in der Zeile weitergeführt, die durch die Zeilennummer nach dem GOTO Befehl bezeichnet wird.

In Maschinensprache gibt es hierzu einen analogen Befehl:
JMP Adresse. (jump)
Nach der Ausführung des JMP Befehls wird das Programm an der neuen Adresse weiterbearbeitet.

An dieser Stelle möchte ich noch einmal an die Bedeutung und Verwendung von Labeln erinnern. Auch hier kann die Adresse nach dem JMP durch ein Label ersetzt werden, so daß man sich nicht mehr um die Errechnung einer absoluten Adresse kümmern muß.

5.2 Bedingte Sprünge

Mit unbedingten Sprüngen läßt sich schlecht ein Programm aufbauen, denn dieses Programm würde nie anhalten, denn es existiert ja keine Anweisung, die das Programm aus dieser Endlosschleife herausreißt.

Hierfür sind nun die bedingten Sprünge zuständig. Bedingt heißt hier, der Sprung wird nur durchgeführt, wenn eine bestimmte Bedingung erfüllt ist. Alle diese Bedingungen beziehen sich auf das Bedingungsregister, in dem sich alle Flags befinden.

Es gibt bei der 8080 CPU acht bedingte Sprungbefehle:

```
JC   Springe, wenn das Carry-Flag gesetzt ist
JNC  Springe,  "     "    "     "   nicht gesetzt ist
JZ   Springe,  "     "  Zero Flag gesetzt ist
JNZ  Springe,  "     "    "     "  nicht gesetzt ist
JM   Springe,  "     "  N(egativ) Flag gesetzt ist
JP   Springe,  "     "  N(egativ) Flag nicht gesetzt ist
JPE  Springe,  "   die Parität gerade ist
JPO  Springe,  "     "    "      ungerade ist
```

Mit den bisher bekannten Befehlen kann man bereits die alle wesentlichen Schleifentypen und Verzweigungen aufbauen:

```
Bsp 1: eine Schleife die genau zehnmal durchlaufen wird:
       MVI   B,10     B=Schleifenvariable
LOOP   beliebige Befehle ausführen, die jedoch das B Register
       nicht verändern
       DCR   B        B wird bei jedem Durchlauf um eins vermindert
       JNZ   LOOP     Dies wird solange wiederholt, bis B
                      schließlich = 0 ist
Dies entspricht in BASIC der FOR Schleife:
FOR B=10 TO 1 STEP -1:  ------   :NEXT B


Bsp 2: eine Schleife, die aufsteigend durchlaufen wird
       MVI   B,1
LOOP   wieder eine beliebige Befehlsfolge
       INR   B        B um eins erhöhen
       MOV   A,B      B für den Vergleich in den Akku bringen
       CPI   11       mit dem Endwert vergleichen
       JNZ   LOOP     falls dieser noch nicht erreicht ist,
                      die Befehle wiederholen
Dies entspricht in BASIC
FOR B=1 TO 10 : -------- : NEXT B


Bsp 3: Ein vollständiger Vergleich:
In BASIC:  100    IF C<B GOTO 150
           110    Befehlsblock1 für den Fall, daß C>=B ist
                  durchführen. Dies wäre der Sonst (ELSE) Fall
                  für die Abfrage
           140    GOTO 200
           150    Befehlsblock2 für den Fall, daß C<B ist
                  durchführen. THEN Fall
           200    Programm gemeinsam weiterführen

       MOV   A,C
       CMP   B
       JC    THEN
       Befehlsblock 1 (C>=B => Kein Carry) - ELSE Fall
       JMP   CONT   Den THEN Teil überspringen
THEN   Befehlsblock 2 (C<B  => Carry       - THEN Fall
CONT   Ab hier wird das Programm gemeinsam weitergeführt
```

Bsp 4: Vergleich mit einer Konstante:
In BASIC: 100 IF B=10 THEN D=H bzw. umgeformt
100 IF B<>10 GOTO 110: D=H

```
        MOV   A,B
        CPI   10
        JNZ   CONT
        MOV   D,H
CONT    Normal weiter
```

5.3 Der PCHL Befehl

Während der Ausführung eines Programms zeigt ein weiteres CPU Register immer auf die Stelle im Hauptspeicher, an der sich der nächste Befehl befinden würde. Dieser Zeiger wird deshalb auch Programmzähler (program counter = PC) genannt. Bei den Sprüngen wird der Programmzähler einfach mit der Adresse hinter dem JMP Befehl geladen und schon dadurch wird das Programm an einer anderen Stelle weiterbearbeitet.

Bei normalen Befehlen wird der Programmzähler automatisch entsprechend der jeweiligen Befehlslänge erhöht, so daß er jedesmal auf den nächsten Befehl zeigt!

Manchmal besteht jedoch der Wunsch zu einer berechneten, also nicht unbedingt festen, Adresse wie bei den JMP Befehlen zu springen! Dies ist mit dem ON GOTO des BASIC zu vergleichen. Hier wird je nach dem Wert des Ausdrucks hinter dem ON zu einer der nach dem GOTO stehenden Zeilennummern gesprungen.

In Maschinensprache könnte so etwas durch den PCHL Befehl realisiert werden (PCHL=program counter HL). Dem Programmzähler wird also der Wert des HL Registerpaares zugewiesen, oder anders ausgedrückt: Das Programm wird ab der Adresse weiterbearbeitet, auf die das HL Registerpaar gerade zeigt. Diese Adresse kann z.B. aus einer Tabelle oä genommen werden.

6 Logische- und Rotationsbefehle

Im täglichen Leben werden sehr häufig die Redewendungen "Wenn das und das ..." oder "Wenn dieses oder jenes ..." verwendet. Es werden damit zwei Aussagen auf verschiedene Weisen mit einander verknüpft. Noch deutlicher wird dies z.B. in einer IF Abfrage in BASIC: IF A<B AND C>D THEN Die Anweisungen des THEN werden nur dann ausgeführt, wenn sowohl A<B, als auch C>D ist. Dies kann genauer und übersichtlicher in einer Wertetabelle dargestellt werden:

Wenn man zwei Aussagen zu verbinden hat, so ergeben sich insgesamt vier Kombinationsmöglichkeiten. Der Wahrheitsgehalt wird dabei durch Nullen und Einsen dargestellt. Null bedeutet Falsch und Eins bedeutet Richtig!

A	B	A und B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A oder B
0	0	0
0	1	1
1	0	1
1	1	1

Wie leicht ersichtlich ist, gibt es im UND Fall nur eine Möglichkeit, daß das Ergebnis wahr ist: A und B müssen richtig sein. Im ODER Fall ist es genau umgekehrt. Das Ergebnis ist wahr, sobald mindestens einer von beiden richtig ist.

Außerdem gibt es noch die Inversion (NOT) und das exclusive ODER (XOR). Das NOT dreht genau die Aussage um. Beim exclusiven ODER ist das Ergebnis genau dann richtig, wenn nur eins von beiden richtig ist (Entweder ... oder ...)

A	NOT A
0	1
1	0

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Wie Sie ja schon aus den ersten Kapiteln wissen, werden in unserer CPU immer 8 Bit gleichzeitig bearbeitet. Es ist hier also möglich alle 8 Bit eines Registers mit denen des Akku über eine der bishervorgestellten Operationsarten zu verknüpfen. Hierbei werden alle Bedingungsregister verändert!
Die Bedeutung dieser Verknüpfungen wird anhand von Beispielen in den entsprechenden Abschnitten erläutert.

6.1 Die Verknüpfung per UND

Der Befehl um ein Register mit dem Akku per UND zu verknüpfen lautet ANA (AND Accumulator) bzw. ANI (AND Immediate

Meiner Einschätzung nach wird der ANI Befehl von beiden am meisten benötigt. Er wird meistens dazu verwendet bestimmte Bits aus einem Byte "auszublenden" - diese Bits können damit auf Null gesetzt werden und der Rest bleibt in der ursprünglichen Form erhalten.

Beispiel: Tastaturabfrage
Im Akku befindet sich das Zeichen der Taste, die gedrückt wurde. Nun soll aber nicht zwischen Groß- und Kleinschrift unterschieden werden, sondern es dürfen nur große Buchstaben vorkommen. Dies kann man einfach mit einem

```
ANI :DF erreichen
```

z.B. das Zeichen 'A' besitzt die Wertigkeit 65=#41 und 'a' entsprechend 97=#61. Stellt man das Ganze in binärer Form dar, so wird es noch etwas deutlicher:

```
   0100 0001   = 'A'
   0110 0001   = 'a'
     ^
```

Nur dieses eine Bit unterscheidet das kleine 'a' von dem großen: Wird dies jedoch über die Maske #DF mit UND verknüpft, so wird dieses Bit auf Null gesetzt. Dem großen 'A' passiert natürlich nichts, da hier das Bit ja bereits auf Null ist:

```
   0110 0001   = 'a'
U  1101 1111   = #DF   (Maske)
============
   0100 0001   = 'A'
```

Dies ist in BASIC übrigens durch ein A=GETC IAND #DF zu erreichen!

6.2 Die Verknüpfung per ODER

Entsprechend zum ANA Befehl lautet hier der Befehl ORA (= OR Accumulator), bzw. ORI Konstante.
Die Bedeutung ist hier ganz entsprechend. Mit dem OR Befehl ist es möglich ein bestimmtes Bit zu verändern - natürlich können durch eine bestimmte Maske auch alle Bits verändert werden.

Bsp 1: Ein Zeichen verändern
Beim CP/M wird ein schreibgeschütztes Programm dadurch gekennzeichnet, daß das erste Zeichen des Typs mit MSB=1 geschrieben wird. Das MSB ist das höchstwertige Bit eines Bytes (most significant Bit). Dieses ist normalerweise bei Eingaben von der Tastatur nicht gesetzt, aber in diesem Fall kennzeichnet es das Programm als schreibgeschützt:
z.B. Life.BAS (BAS ist der Typ des Programms)
Dieses 'B' kann auf zwei verschiedene Arten geschrieben werden: Binär:

```
0100 0010  - normales 'B'
1100 0010  - 'B' mit MSB gesetzt, d.h.
^            unter CP/M schreibgeschützt
```

Um ein normales Zeichen in eines mit MSB gesetzt zu überführen, verwendet man den Befehl: ORI :80

```
   0100 0010  - normales 'B'
O  1000 0000  - #80
   ==========
   1100 0010  - 'B' mit MSB gesetzt
```

Bsp 2: Der Befehl ORA A / ANA A
Mit diesen Befehlen wird eigentlich kein Bit verändert, nur beeinflußen diese Befehle ja auch das Bedingungsregister, so daß über den Wert im Akku konkrete Aussagen gemacht werden können ohne erst einen der anderen Vergleichsbefehle benutzen zu müssen, z.B:

```
LDA   Zahl
ORA   A
```

Da ja der Befehl LDA das Bedingungregister nicht verändert ist auch keine Aussage über die Zahl im Akku möglich. Nach dem ORA A ist möglich auf gleich/ungleich Null oder Zahl positiv/negativ usw. zu überprüfen und entsprechend zu handeln.
Außerdem setzt dieser Befehl das Carry Flag wieder auf Null!

6.3 Die Verknüpfung per XOR

XOR wird verwendet um in einem Byte bestimmte Bits umzudrehen.
In der Mnemonic Schreibweise lautet dieser Befehl
XRA Reg., bzw. XRI Konstante
Eine Besonderheit bietet der Befehl XRA A. Dieser Befehl setzt den Akkumulator auf Null und ist dabei nur ein Byte lang - im Gegensatz zum MVI A,0 der zwei benötigt!

```
   Wirkungsweise des XRA A:
   1011 1010   beliebiger Wert im Akku
X  1011 1010
   ==========
   0000 0000
```

Hierzu betrachte man am besten noch einmal die Wertetabelle der XOR Verknüpfung. Da immer gleiche Bits mit einander verknüpft werden - also Nullen mit Nullen und Einsen mit Einsen ist das Gesamtergebnis immer Null!

Der Befehl XRI :FF invertiert das A Register, da ja alle Werte mit Eins verbunden werden:

```
1 XOR 1 = 0
0 XOR 1 = 1
```

6.4 Invertierung des Akkus

Neben dem Befehl XRI :FF gibt es für diesen Anwendungsfall noch einen Spezialbefehl: CMA (=complement Akkumulator) Dieser Befehl verändert zudem auch die Bedingungsregister nicht!

Bitte beachten Sie jedoch: Um die negierte Zahl im Akku zu erhalten muß nach dem CMA der Akku noch um eins erhöht werden! Also:

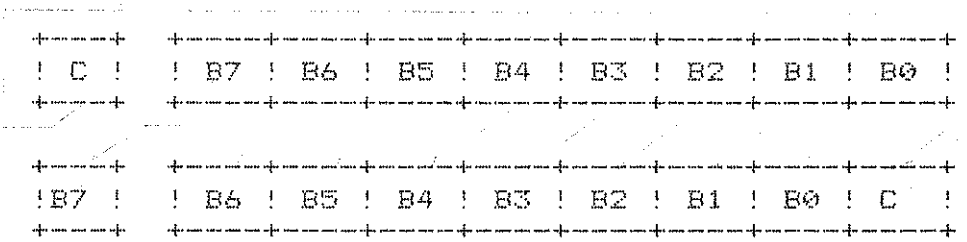
```
CMA
INR  A
```

7 Rotationsbefehle und Carry-Befehle

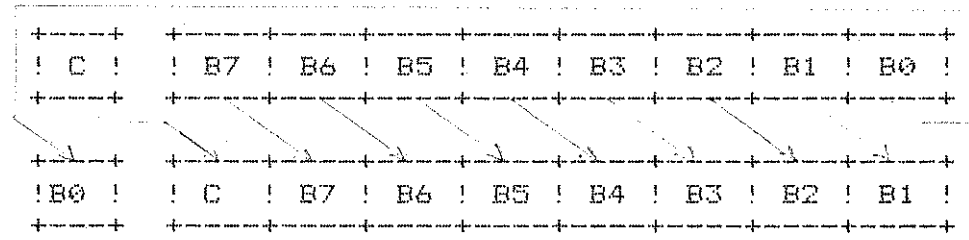
Die Rotationsbefehle verschieben den Inhalt des Akkumulators zyklisch um ein Bit. Das Rotieren ist sowohl nach rechts als auch nach links möglich! Die 8080 CPU kennt nur zwei verschiedene Arten der Rotation: Einmal wird das Carry Flag als quasi 9. Bit mitberücksichtigt und beim anderen Mal wird es nicht mitverschoben!

7.1 Zyklisches Schieben des Akkus und des Carry-Bits

Dies wird um besten an einer Skizze deutlich:



Akkumulator und Carry links:
RAL (rotate akkumulator left)



Akkumulator und Carry rechts:
RAR (rotate akkumulator right)

Das zyklische Rotieren nach links (bei nicht gesetztem Carry-Flag) entspricht einer Multiplikation mit 2!

Dementsprechend kann durch die Befehle

```
ORA A
RAR
```

eine Division durch zwei erreicht werden, denn der ORA A setzt das Carry Flag zurück (d.h. auf Null) und der RAR Befehl schiebt das Register dann um eins nach rechts = Divison durch 2!

Aber auch das Rotieren mit nicht zurückgesetztem Carry Flag hat seine Bedeutung. Dies zeigt folgendes Beispiel: Division des HL Registers durch zwei:

```
 (1)      MOV   A,H     H-Register in den Akku bringen
 (2)      ORA   A       Carry zurücksetzen
 (3)      RAR           Akku rotieren; das letzte Bit des
                        H Registers befindet sich nun im
                        Carry Flag !
 (4)      MOV   H,A     Ergebnis nach H bringen
 (5)      MOV   A,L     L-Register in den Akku bringen
 (6)      RAR           Akku rotieren; das Carry Bit wird nun
                        ins höchste Bit des Akkus geschoben
 (7)      MOV   L,A     Ergebnis nach L bringen
```

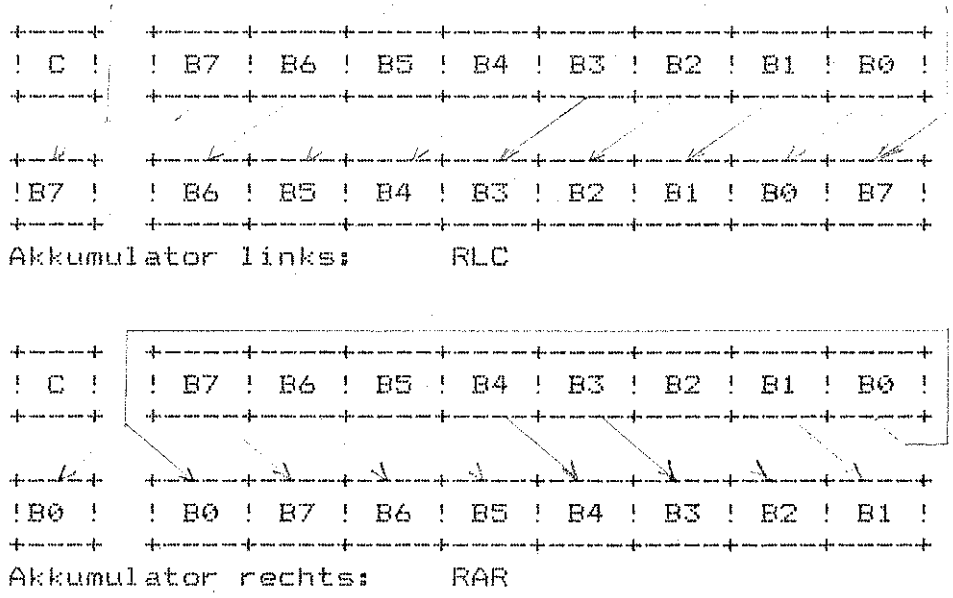
In diesem Beispiel diente das Carry Flag also als Zwischenspeicher für das letzte Bit des H Registers!

```
Bsp:     HL=   1011 0111  0010 1011
1)        A=   1011 0111
2)                              Carry auf Null setzen
3)        A=   0101 1011       Carry = letztes Bit = 1
4)        H=   0101 1011
5)        A=   0010 1011
6)        A=   1001 0101       Carry = letztes Bit = 1
7)        L=   1001 0101
         HL=   0101 1011  1001 0101
```

Die 1 im Carry Flag zeigt, daß die Division einen Rest ergab, also nicht glatt aufging!

7.2 Zyklisches Schieben des Akkus allein

Skizze:



Die Anwendungsbeispiele sind hier ähnlich wie bei den beiden anderen aufgeführten Befehlen: Multiplikation und Division auf 8 Bit Ebene. Auf 16 Bit Ebene sind sie schlecht zu gebrauchen, da der Übertrag von einem Byte auf ein anderes nicht berücksichtigt werden kann.

7.3 Setzen des Carry Flags

Mit dem Befehl STC (set carry) kann das Carry Flag auf einen definierten Wert (=1) gesetzt werden. Im DAI BASIC wird dies sehr häufig verwendet. Kehrt eine Routine mit Carry gesetzt zurück, so bedeutet dies, während der Bearbeitung ist ein Fehler aufgetreten.

7.5 Rücksetzen des Carry Flags

Dieser Befehl wurde schon früher aufgeführt, aber er sei hier der Vollständigkeit wegen noch einmal genannt: ORA A Dadurch daß bei den logischen Befehlen immer nur zwei Bit direkt miteinander verknüpft werden, kann bei diesen Operationen kein Übertrag entstehen (=> Carry=0).
Um keine Mißverständnisse aufkommen zu lassen. Es werden zwar zwei jeweils 8 Bit breite Register miteinander verknüpft, aber es treten zwischen den Bits keine Überträge auf die nächste Stelle auf und somit werden effektiv jeweils gegenüberliegende Bits miteinander verknüpft!

7.6 Complementieren des Carry Flags

Dies geschieht mit dem Befehl CMC (complement carry). Hiermit wird also dieses Flag "umgedreht", also aus Null wird Eins und umgekehrt.

8 Indirekte Adressierung

Bisher haben wir zwei verschiedene Adressierungsmöglichkeiten kennengelernt.
Dies waren zum einen die direkte Adressierung, z.B. bei LDA oder SHLD - eine Speicheradresse wurde also direkt angesprochen.

Zum anderen stießen wir auf die unmittelbare Adressierung (Immediate). Dies wurde vor allem bei Wertzuweisungen benötigt, z.B. MVI H,34 LXI B,12 aber auch bei mathematischen und logischen Operationen, wie ADI 10 CPI 45 ANI :DF usw. Diese Art der Adressierung erhielt ihren Namen dadurch, daß der Wert unmittelbar dem Register (bzw. Befehl) folgt, auf den er sich bezieht.

Diese beiden Möglichkeiten allein sind jedoch zumeist nur sehr unbefriedigend für den Programmierer , da immer nur auf die gleichen Speicherplätze zugegriffen werden kann. In BASIC hat man deshalb die Möglichkeit Arrays zu verwenden. Man kann jeden einzelnen Speicherplatz eines Arrays unter seinem Index aufrufen. Diese Art der Adressierung erhielt deshalb den Namen 'Indirekte Adressierung'. Eine ähnliche Konstruktion ist in Maschinensprache natürlich auch möglich:

8.1 Indirektes Laden und Abspeichern:

Hierfür stehen zwei Befehle zur Verfügung: Diese heißen LDAX und STAX und als Indexregister können hier das BC oder das DE Register verwendet werden.
Die beiden Registerpaare kann man sich hierfür als Zeiger auf eine Speicherzelle im Hauptspeicher denken. Mit LDAX wird nun der Wert der Speicherzelle in den Akku geladen, auf den dieser Zeiger gerade zeigt. Mit STAX wird der Akku entsprechend in diese Adresse abgespeichert. Danach kann der Zeiger verändert werden und schon kann auf eine neue Adresse zugegriffen werden.
Mit STAX und LDAX gibt es also insgesamt vier Möglichkeiten

LDAX	B	STAX	B	Für das HL Registerpaar gibt keinen
LDAX	D	STAX	D	Befehl dieser STAX/LDAX Reihe, da dieses andere Möglichkeiten hierfür kennt.

8.2 Indirekte Adressierung mit dem HL Registerpaar

Unter Verwendung des HL Registerpaares sind noch viele weitere Anwendungen möglich, die weit über die der LDAX und STAX Befehle hinausgehen. Hierfür existiert noch ein zusätzliches Register, daß eigentlich gut neben den anderen sieben CPU Registern stehen könnte. Es handelt sich hierbei um das M-Register (M-Memory).
Dieses Register enthält immer den Wert, der Speicherzelle auf das HL gerade zeigt. Der Inhalt dieses Registers kann dann mit MOV in ein anderes Register verschoben werden, mit ADD M kann der Inhalt dieses Registers auf den Akkumulator aufaddiert werden usw.

Die vollständige Liste der Befehle die mit dem M Register durchgeführt werden können:

MOV	A,M	MOV	M,A	ADD	M	CMP	M
MOV	B,M	MOV	M,B	ADC	M		
MOV	C,M	MOV	M,C	SUB	M	INR	M
MOV	D,M	MOV	M,D	SBB	M	DCR	M
MOV	E,M	MOV	M,E	ANA	M		
MOV	H,M	MOV	M,H	ORA	M	MVI	M,Konst
MOV	L,M	MOV	M,L	XRA	M		

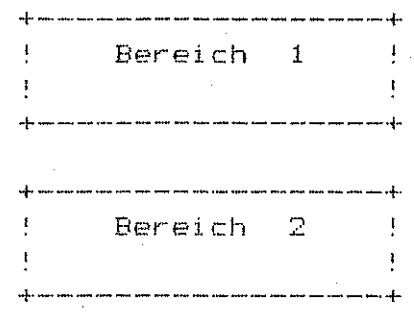
Mit den Befehlen INR M und DCR M wird also ein Inkrement, bzw. Dekrement direkt auf Hauptspeicheradressen durchgeführt und die Werte müssen nicht erst in die CPU Register eingelesen werden.

Schließlich kann noch mit dem Befehl MVI M,Konstante ein Wert direkt in den Hauptspeicher geschrieben werden ohne erst einen MVI A,Konstante und STA Adresse durchführen zu müssen.

Bitte beachten Sie in diesem Zusammenhang: Der Wert des M Registers ist abhängig vom Wert des HL Registerpaares. Wird also HL geändert, so ändert sich M ebenfalls!

Bsp 1: Verschieben eines Speicherbereichs an eine andere Stelle

Hierzu werden die Startadressen der beiden Bereiche und die Länge des Feldes benötigt. Außerdem wird angenommen, daß die Startadresse des zweiten Bereiches mindestens um die Konstante Länge höher im Speicher liegt als die erste Adresse, da sonst die ersten verschobenen Zeichen schon die letzten Zeichen des ersten Bereiches überschreiben:



Die Konstante ADR1 beinhaltet die Startadresse des 1. Blocks

ADR2 entsprechend die des 2. Blocks

```
        LXI   D,ADR1      Das DE Registerpaar zeigt also auf
                          den Anfang des ersten Blocks
        LXI   H,ADR2      und HL auf den des zweiten
        MVI   B,Länge     Das B Register enthält die Länge
                          des Blocks
LOOP    LDAX  D           Ein Byte in den Akku lesen
                          (in Abhängigkeit von DE)
        MOV   M,A         Dieses Byte wieder abspeichern
                          (in Abhängigkeit von HL)
        INX   D           DE und HL um eins erhöhen
        INX   H           Sie zeigen also auf die nächsten Adr.
        DCR   B           Die Schleifenvariable um 1 vermindern
        JNZ   LOOP        Fall B<>0 ist soll weiter verschoben
                          werden.
```

Bsp 2: Darstellung aller verfügbaren Zeichen auf dem Bildschirm

Da der Bildschirm nur einen speziellen Bereich des Speichers darstellt kann dieser Bereich genauso mit diesen Befehlen angesprochen werden, wie der sonstige Bereich!

Auf den Bildschirm, beginnend ab der 3. Zeile von oben, sollen alle auf dem DAI darstellbaren Textzeichen geschrieben werden.

```
TOP       EQU    :BFEF-268-20 Bildschirmstartadresse für das
                                Programm
          ORG    :300         Das Programm soll ab #300
                              beginnen
        LXI   H,TOP       HL einmal die Startadresse zuweisen
        LXI   D,-70       (1)
        MVI   C,0         (2)

LBL1    MVI   B,16        Es sollen 16 Zeichen pro Zeile
                          dargestellt werden!
LBL2    MOV   M,C         Ein Zeichen in den Bildschirm schreiben
        INR   C           (3)
        DCX   H           (4)
        DCX   H
        DCX   H
        DCX   H
        DCR   B           alle Zeichen in dieser Zeile dar-
        JNZ   LBL2        gestellt?
        DAD   D           Übergang zur nächsten Zeile
        MOV   A,C         (5)
        ORA   A
        JNZ   LBL1
        RET               (6)
```

Erklärungen:

Zu 1) Die Konstante -70 erhält man aus folgender Rechnung: Es sollen pro Zeile 16 Zeichen dargestellt werden, wobei zwischen zwei Zeichen noch ein Leerzeichen überigbleiben soll. Insgesamt werden also pro Zeile 32 Zeichen benutzt. Beim DAI wird darüber hinaus nur jedes 2. Byte für Textdarstellung verwendet - die anderen Adressen dienen als Farbbytes. Somit werden also 64 Bytes pro Zeile durch das Programm angesprochen.
Eine Textzeile besteht im DAI normalerweise aus 134 Zeichen, so daß noch 70 abgezogen werden muß, um wieder die 134 (=die nächst tiefere Zeile) zu erreichen. Der Text wird ja von oben nach unten auf den Schirm geschrieben und die oberste Zeile hat die höchste Adresse, deshalb wird subtrahiert!

Zu 2) Das C Register wird als Variable für das nächste darzustellende Zeichen verwendet. Durch den MOV M,C wird ein Zeichen in den Bildschirm geschrieben.

Zu 3) Anschließend wird durch den INR C (3) diese Variable um eins erhöht - also befindet sich nun das nächste Zeichen im C Register usw.

Zu 4) HL wird um vier vermindert, um auf die nächste Textposition zeigen zu können. Es werden jeweils zwei für das Zeichen und zwei für die Leerstelle abgezogen!

Zu 5) Der Zahlenbereich für das 8-Bit breite C-Register ist ja 256 Zahlen groß (0-255). Pro dargestellter Zeile wird das C Register (beginnend bei Null) jeweils um 16 erhöht (In einer Zeile werden ja 16 Zeichen dargestellt). Nach genau 16 Zeilen muß der Wert im C Register somit wieder Null sein (16 x 16 = 256 = 0 (!)), denn 256 ist in 8 Bit nicht mehr darstellbar. Es würde zwar ein Übertrag entstehen, aber der Wert im Register ist Null! Durch den MOV A,C wird dieser Wert in den Akku gebracht und mit ORA A abgetestet, ob er gleich Null ist. Wenn nicht, so wird das Verfahren wiederholt!

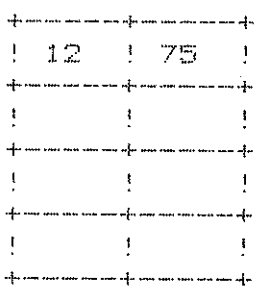
Zu 6) Der RET Befehl wird im nächsten Kapitel erklärt!

Noch ein Tip: probieren Sie dieses Programm ruhig einmal aus! Nach dem Assemblieren können Sie es z.B. vom BASIC aus durch CALLM#300 aufrufen.

Bitte beachten Sie auch die Geschwindigkeit, mit der das Programm abläuft! Es ist nicht sichtbar, daß die Zeichen einzeln auf den Schirm gebracht werden! Eine genaue Berechnung der Geschwindigkeit eines Maschinenprogramms wird in Kapitel 12 durchgeführt. Für dieses Programm liegt die Bearbeitungszeit bei ca. 12,9 Milli Sekunkunden! Versuchen Sie ein ähnliches Programm einmal in BASIC zu schreiben und vergleichen Sie die Ausführungszeiten!!!

9 Verwendung des Stacks

Die 8080 CPU besitzt einen sogenannten Stack oder Stapel. Auf diesem Stack können Werte aus den CPU Registern kurzzeitig zwischengespeichert werden, um nach einer Berechnung dann den Wert sofort zurückholen zu können. Der Stack hat somit auch die Funktion von Hilfsvariablen. Er befindet sich beim DAI im Adreßbereich von #F800 bis #F8FF und ist somit 256 Bytes groß - genügend Platz um 128 mal ein Registerpaar zwischenspeichern zu können! Der Stack wird bei der 8080 CPU von oben nach unten im LIFO Verfahren aufgefüllt. Dies ist folgendermaßen zu verstehen: Der Stack kann als Skizze wie folgt dargestellt werden:



Um sich die Stelle zu merken an der der nächste Eintrag abgespeichert, bzw. von der der nächste Eintrag gelesen werden soll, besitzt die 8080 CPU einen Stapelzeiger (Stack-pointer SP). Soll jetzt ein Wert auf dem Stack abgespeichert werden, so wird zuerst der Stapelzeiger um eins vermindert, dann wird ein Register des angesprochenen Paares abgelegt, danach wird der SP wieder um eins erniedrigt und nun das zweite Register abgespeichert. Dies kann - wie gesagt - bis zu 128 mal durchgeführt werden. Das LIFO Verfahren besagt, der Wert, der zuletzt auf den Stapel gebracht wurde (Last In) wird zuerst wieder gelesen (First Out) - Dies wird insbesondere bei Unterprogrammen sehr häufig benötigt, doch dazu unten mehr.

9.1 PUSH und POP Befehle

Diese beiden Befehle erledigen die Aufgabe ein Registerpaar auf den Stack abzulegen bzw. diesen Wert wieder einem Registerpaar zuzuweisen!

PUSH Registerpaar:
Der PUSH Befehl schreibt den Inhalt des angegebenen Registerpaares auf den Stack und verändert den Stackpointer automatisch mit. Als Registerpaare können hier auftreten:
PUSH H
PUSH D
PUSH B
PUSH PSW
PSW ist eine Bezeichnung für den Akku zusammen mit dem Flag Register (=Proccessor Status Word).

Mit diesen vier Befehlen können somit alle Register "gerettet" werden - so daß Sie nach Ausführung eines Befehls unverändert vorliegen!

POP Registerpaar:
Mit dem POP Befehl "holt" man den nächsten Wert vom Stapel. Dieser Wert wird dann im angegebenen Register abgelegt. Denken Sie hierbei an das LIFO Prinzip - der letzte Wert wird wieder zuerst gelesen!!!

Bsp 1:

```
        PUSH    PSW     Alle Register retten !!!
        PUSH    B
        PUSH    D
        PUSH    H
        Befehlsfolge
        POP     H       Die Register in umgekehrter Reihen-
        POP     D       folge wieder einlesen LIFO
        POP     B
        POP     PSW
```

Bsp 2:
Addition von HL und DE - das Ergebnis soll sich in DE befinden; HL darf nicht zerstört werden und andere Register stehen auch nicht zur Verfügung!

```
        PUSH    H       HL retten
        DAD     D       Addition durchführen
                        - HL enthält das Ergebnis -
        XCHG            HL und DE vertauschen
        POP     H       den alten Wert von HL zurückholen
```

Anmerkung:
Das Abspeicher- bzw. Leseregister müssen nicht unbedingt identisch sein. Es ist z.B. möglich das HL Register auf dem Stack abzulegen, aber dann mit POP B den ursprünglichen Wert des HL Registers nach BC zu laden!
Dies ist auch die einzige Möglichkeit, auf den Inhalt des Bedingungsregisters als Ganzes zugreifen zu können, nämlich durch die Folge:

```
    PUSH  PSW
    POP   H
```

Im HL Registerpaar befindet sich nun neben dem Wert des Akkus auch der Wert des Bedingungsregisters!

9.3 Der XTHL Befehl

Dieser Befehl vertauscht den Wert, auf den der Stackpointer gerade zeigt mit dem Wert des HL Registers. Es gehen dabei keine Daten verloren, denn die Werte werden ja echt ausgetauscht. XTHL = eXchange Top of stack with HL

```
Bsp:    LXI     H,1234
        PUSH    H
        LXI     H,9876
        XTHL                HL enthält nun 1234 (der letzte Eintrag)
        POP     H           und nun wieder 9876 (!)
```

9.4 Der LXI SP, SPHL und DAD SP Befehl

Bis jetzt wurde immer davon ausgegangen, daß sich der Stack im Bereich von #F800 bis #F8FF befindet. Für spezielle Anwendungen, die unter Umständen mehr Platz für Daten benötigen, kann der Stack auch in einen anderen Speicherbereich verlegt werden. Dies geschieht einfach mit dem Befehl

```
        LXI SP,Adresse
```

Von nun an werden alle Einträge in den Stack von der angegebenen Adresse an abwärts eingetragen. Alle vorher gemachten Einträge auf den Stack sind natürlich verloren!

Der Befehl SPHL funktioniert ähnlich, nur wird hiermit der Stack an die Stelle verlegt, auf die gerade das HL Register zeigt!

Der DAD SP Befehl addiert auf den Inhalt des HL Registers den Wert des Stackpointers auf. Dies ist die einzige Möglichkeit an den aktuellen Wert des SPs heranzukommen, denn hierfür existiert kein spezieller Ladebefehl! Um den Wert also in das HL Register zu bekommen benötigt man folgende Befehlsfolge:

```
LXI   H,0
DAD   SP
```

Bemerkung: Wenn sich der Stack in seinem normalen Bereich von #F800 bis #F8FF befindet, so wird ein Überschreiten dieses Bereiches hardwaremäßig abgefangen und die Meldung 'STACK OVERFLOW' wird ausgegeben. Dies geschieht dadurch, daß ein Schreiben in den Bereich von #F000 bis #F7FF als Fehler interpretiert wird. Befindet sich der Stack jedoch in einem tieferen Bereich, so kann diese Überprüfung nicht mehr durchgeführt werden und der Programmierer muß hierfür selbst Sorge tragen, daß der Stackbereich nicht überschritten wird!

10 Unterprogramme

Auch für Unterprogrammstrukturen läßt sich in BASIC leicht eine Analogie finden. In BASIC sind dies Programmstücke, die mit GOSUB angesprochen werden. Man benutzt häufig diese Konstruktionen, um Programmteile, die von mehreren Stellen des Hauptprogramms aus benötigt werden nur einmal hinschreiben zu müssen. Der Programmfluß verzweigt bei einem GOSUB zur angegebenen Zeilennummer des Unterprogramms und kehrt nach dem RETURN wieder zum Hauptprogramm zurück. Auch in Maschinensprache wird dieses Verfahren sehr gerne benutzt, zumal als Unterprogramm auch alle Routinen des DAI BASIC ROMs möglich sind, so daß man hierdurch viel Arbeit einsparen kann.

10.1 Der unbedingte Unterprogrammaufruf

Genauso wie bei den Sprungbefehlen (JMP) gibt es auch bei den Unterprogrammaufrufen eine entsprechende Form, die das Unterprogramm in jedem Fall aufruft - also ohne eine Bedingung abzutesten! Dieser Befehl lautet Call Adresse. Dieser Befehl bewirkt im einzelnen:
Der gegenwärtige Programmzähler (PC) wird auf dem Stack abgespeichert und dann wird ein Sprung zur angegebenen Adresse durchgeführt. Der PC wird also mit der neuen Adresse geladen und das Unterprogramm wird aufgerufen.

10.2 Der bedingte Unterprogrammaufruf

Natürlich gibt es hier auch Unterprogrammaufrufe, die nur dann durchgeführt werden, wenn eine bestimmte Bedingung erfüllt ist. Als Bedingung können hierbei alle Konstruktionen wie bei den bedingten Sprüngen auftauchen. Dies sind

```
CC    Call, falls Carry gesetzt
CNC   Call, falls Carry nicht gesetzt
CZ    Call, falls Zero Flag gesetzt
CNZ   Call, falls Zero Flag nicht gesetzt
CM    Call, falls Zahl negativ
CP    Call, falls Zahl positiv
CPE   Call, falls Parität gerade
CPO   Call, falls Parität ungerade
```

Dies ist in BASIC nur durch eine IF ... THEN GOSUB Konstruktion zu erreichen!

10.3 Die Rücksprünge

In BASIC gibt es hierfür das RETURN Kommando. Dies ist in Maschinensprache zwar auch möglich, jedoch gibt es neben dem unbedingten RET auch noch differenziertere Möglichkeiten:

RET	Return	ohne Bedingung
RC	"	falls Carry gesetzt
RNC	"	falls Carry nicht gesetzt
RZ	"	falls Zero Flag gesetzt
RNZ	"	falls Zero Flag nicht gesetzt
RM	"	falls Zahl negativ
RP	"	falls Zahl positiv
RPE	"	falls Parität gerade
RPO	"	falls Parität ungerade

Jeder Rücksprung bewirkt folgendes:
Ein Wert wird vom Stack gelesen. Dies war die Adresse des Programmzählers vor dem Aufruf des Unterprogramms. Anschliessend wird der Programmzähler wieder auf diesen alten Wert gesetzt und das Programm wird einfach weitergeführt!

Hieraus ergeben sich einige Forderungen, die unbedingt zu beachten sind:
Die Anzahl der PUSH und POP Befehle müssen in einem Unterprogramm identisch sein, da die 8080 CPU keine Trennung zwischen Daten und Rücksprungsadressen vornimmt und somit auch keine zwei verschiedenen Stapel existieren, würde sonst das Datum eines PUSH Befehls als Rücksprungadresse betrachtet und das Programm würde an diese Stelle "zurückkehren" - na ja und was weiter geschehen würde bleibt dem Zufall überlassen.
Diese Regel (gleiche Anzahl von PUSH und POP Befehlen) sollte eigentlich immer beachtet werden, da auch das Hauptprogramm vom BASIC aus ähnlich wie ein CALL aufgerufen wird und auch hier hätten unterschiedliche PUSH / POP Anzahlen verheerende Folgen!!!

10.4 Die RST Befehle

Diese Befehle arbeiten wie ein Unterprogrammaufruf - bestehen allerdings im Gegensatz zu den CALL Aufrufen nur aus einem ein-Byte langen Programmcode. Dafür bilden Sie aber auch nur einen Sprung zu einer festen Adresse, die sich in ersten 64 Bytes des Hauptspeichers befindet. Es gibt insgesamt 8 RST (=ReSTart) Befehle (0-7), die bei den Adressen 0,8,16,24 usw (jeweils 8 höher) beginnen, von diesen Adressen aus wird dann weitergesprungen und der abschließende Befehl muß in jedem Fall ein RET sein!

Beim DAI sind von den RST Befehlen besonders die Befehle RST 1, RST 4 und RST 5 interessant. Über den RST 1 wird das Decodieren von BASIC Eingaben, über RST 4 die Mathematikfunktionen und über RST 5 die Bildschirmoperationen durchgeführt. In diesen drei Fällen wird dem RST noch ein Datenbyte angehängt. Durch dieses Datenbyte wird unterschieden, welche der Funktionen durchgeführt werden soll, z.B.

```
RST   5
DATA  3
```

Ausgabe des Zeichens im Akku an den Bildschirm!
Die vollständige Liste aller RST Funktionen finden Sie im Anhang bei der Erläuterung der wichtigsten ROM Routinen:

11 Spezialbefehle

11.1 Der NOP Befehl

Dieser Befehl führt keine Operation aus. Deshalb auch der Name NOP (No OPeration). Er hat die Bedeutung eines Platzhalters, da er keine Funktion ausübt kann er also beliebig im Programm auftauchen, ohne etwas zu verändern, aber zu späterer Zeit kann hierfür dann z.B. eine Operation eingesetzt werden.

11.2 Der HLT Befehl

HLT = (Halt) veranlaßt den Prozessor an dieser Stelle mit der Abarbeitung des Programms zu stoppen und in einen Wartezustand zu gehen. Der Prozessor kann beim DAI nur durch RESET aus diesem Wartezustand geholt werden. Dieser Befehl hat trotzdem seine Berechtigung! Es kommt manchmal beim Programmieren vor, daß man einen so vertrackten Fehler im Programm hat, daß man nicht sagen kann, ob das Programm noch an einer bestimmten Stelle ankommt oder nicht. Um dies jetzt herauszufinden setzt man an diese Stelle den HLT Befehl.
Wird dieser Befehl erreicht, so bleibt der Cursor stehen und man hat sein Ergebnis - die Stelle wird erreicht! Ansonsten muß man weiter versuchen den Fehler einzukreisen. Dieses Verfahren ist garnicht so schlimm, wie es sich anhört, da Maschinenprogramme im allgemeinen durch einen RESET nicht zerstört werden, so daß das Programm nicht jedesmal neu geladen werden muß!

11.3 Die Befehle DI und EI

Während der Bearbeitung eines jeden Programms läufen normalerweise noch ab und zu andere Programme im Hintergrund. Als bekanntes Beispiel hierzu ist das Cursorblinken zu nennen. Selbst während ein Programm läuft, blinkt der Cursor munter weiter. Sein Blinken wird durch ein Programm ausgeführt, daß über eine Uhr alle 20 ms gestartet wird. Als weitere Beispiele hierfür ist das periodische Abfragen der Tastatur oder auch der SOUND Befehl zu nennen. Diese Programme werden per Interrupt aufgerufen. Der Interrupt - eine Programmunterbrechung - wird durch verschiedene Uhren im DAI gesteuert, die nach bestimmten Zeitintervallen einen solchen Interrupt auslösen, das eigentliche Programm somit unterbrechen, dann ihre Aufgabe durchführen und schließlich das Programm weiterführen lassen.

Für bestimmte zeitkritische Anwendungen, wie z.B. beim Einlesen oder Abspeichern vom Cassettenrecorder aus ist dies jedoch unerwünscht, bzw. würde das Programm mit diesen Interrupts nicht mehr funktionieren. Deshalb gibt es den Befehl DI = Disenable Interrupt. Somit wird kein Interrupt mehr gegeben und das Programm wird ohne Unterbrechungen durchgeführt.

Um nach Beendigung der zeitkritischen Routine, so elementare Dinge wie Tastaturabfragen wieder zuzulassen kann man den Interrupt wieder einschalten. Dies geschieht durch den Befehl EI (=Enable Interrupt). Dannach läuft das Programm mit Unterbrechungen normal weiter.

Außer den bisher aufgeführten Befehlen gibt es noch zwei Befehle IN und OUT, die jedoch beim DAI nicht unterstützt und aus diesem Grund auch hier nicht aufgeführt werden.

12 Berechnung des Zeitbedarfs eines Maschinenprogramms

Da Maschinenprogramme meist sehr schnell ablaufen ist es in fast allen Fällen unmöglich die Rechenzeit für ein Programm zu ermitteln, wenn man die eingebaute Uhr (über #1BE/#1BF) benutzen will, da diese Uhr nur auf 20 ms genau stoppt. Jedoch kennt man für jeden Maschinenbefehl die Zeit, die dieser zur Bearbeitung benötigt. Hieraus kann man leicht durch Summation und ggf. durch Multiplikation (bei Schleifen) den Zeitbedarf errechnen, den das Programm benötigt. Im folgenden wird deshalb eine Liste aller Befehle mit ihren Befehlscodes, ihrem Zeitbedarf und aller beeinflußten Bedingungsregister aufgeführt.

Der Begriff 'Zeitbedarf pro Befehl' ist eigentlich nicht richtig, da sich die Angaben auf Takt Zyklen beziehen. Wird der Prozessor doppelt so schnell getaktet, so dauert der Befehl natürlich auch nur halb so lange. Beim DAI ist diese Rechnung jedoch recht einfach. Im RAM kann mit einer Taktfrequenz von effektiv 1 MHz gerechnet werden. Dieser Wert ist zwar nicht ganz genau, jedoch kann er ruhig als Minimum angesehen werden, d.h. das Programm ist sicher in dieser Zeit fertig!

Es ist natürlich auch möglich ein (kleines) Programm im (unteren) Stack Bereich abzulegen. Da es sich hier um statische RAMs handelt, die keinen Refresh benötigen und die auch nicht durch den Video Scanner abgetastet werden kann hier mit einer Frequenz von ca. 2 MHz gerechnet werden - also doppelt so schnell!

Man rechnet also um die reale Geschwindigkeit zu erhalten, die Anzahl der benötigten Taktzyklen aus und dividiert diese Zahl durch 1.000.000 (ggf. auch durch 2.000.000) und erhält somit die benötigte Rechenzeit!
Dieses Verfahren lohnt sich jedoch nur für wirklich zeitkritische Anwendungen, z.B. Bildinitialisierung. Liegt die Zeit hierfür z.B. unter 1/10 sec., so kann die Routine verwendet werden, sonst müßte man sich evtl. Gedanken darüber machen, wie dies zu verbessern wäre.

Befehlsliste des INTEL 8080

Mnem	Wirkung	umsitt.		Akku A		B-Reg		C-Reg		D-Reg		E-Reg		H-Reg		L-Reg		M-Reg		Bedingungen				
		Op	B Z	Op	B Z	Op	B Z	Op	B Z	Op	B Z	Op	B Z	Op	B Z	Op	B Z	Op	B Z	N	Z	H	P	C
MVI	r=konst			3E	2 7	06	2 7	0E	2 7	16	2 7	1E	2 7	26	2 7	2E	2 7	36	2 10					
MOV	A=reg			7F	1 5	78	1 5	79	1 5	7A	1 5	7B	1 5	7C	1 5	7D	1 5	7E	1 7					
MOV	B=reg			47	1 5	40	1 5	41	1 5	42	1 5	43	1 5	44	1 5	45	1 5	46	1 7					
MOV	C=reg			4F	1 5	48	1 5	49	1 5	4A	1 5	4B	1 5	4C	1 5	4D	1 5	4E	1 7					
MOV	D=reg			57	1 5	50	1 5	51	1 5	52	1 5	53	1 5	54	1 5	55	1 5	56	1 7					
MOV	E=reg			5F	1 5	58	1 5	59	1 5	5A	1 5	5B	1 5	5C	1 5	5D	1 5	5E	1 7					
MOV	H=reg			67	1 5	60	1 5	61	1 5	62	1 5	63	1 5	64	1 5	65	1 5	66	1 7					
MOV	L=reg			6F	1 5	68	1 5	69	1 5	6A	1 5	6B	1 5	6C	1 5	6D	1 5	6E	1 7					
MOV	M=reg			77	1 5	70	1 5	71	1 5	72	1 5	73	1 5	74	1 5	75	1 5	76	1 7					
INR	rg=rg+1			3C	1 5	04	1 5	0C	1 5	14	1 5	1C	1 5	24	1 5	2C	1 5	34	1 10	x	x	x	x	
DCR	rg=rg-1			3D	1 5	05	1 5	0D	1 5	15	1 5	1D	1 5	25	1 5	2D	1 5	35	1 10	x	x	x	x	
ADI	A=konst	C6	2 7																	x	x	x	x	x
ADD	A=A+K			87	1 4	80	1 4	81	1 4	82	1 4	83	1 4	84	1 4	85	1 4	86	1 7	x	x	x	x	x
ACI	A=A+K+c	CE	2 7																	x	x	x	x	x
ADC	A=A+rg+c			8F	1 4	88	1 4	89	1 4	8A	1 4	8B	1 4	8C	1 4	8D	1 4	8E	1 7	x	x	x	x	x
SUI	A=A-K	D6	2 7																	x	x	x	x	x
SUB	A=A-rg			97	1 4	90	1 4	91	1 4	92	1 4	93	1 4	94	1 4	95	1 4	96	1 7	x	x	x	x	x
SBI	A=A-K	DE	2 7																	x	x	x	x	x
SBB	A=A-rg			9F	1 4	98	1 4	99	1 4	9A	1 4	9B	1 4	9C	1 4	9D	1 4	9E	1 7	x	x	x	x	x
ANI	A=A u K	E6	2 7																	x	x	x	x	x
ANA	A=A u rg			A7	1 4	A0	1 4	A1	1 4	A2	1 4	A3	1 4	A4	1 4	A5	1 4	A6	1 4	x	x	x	x	x
ORI	A=A o K	F6	2 7																	x	x	x	x	x
ORA	A=A o rg			B7	1 4	B0	1 4	B1	1 4	B2	1 4	B3	1 4	B4	1 4	B5	1 4	B6	1 4	x	x	x	x	x
XRI	A=A x K	EE	2 7																	x	x	x	x	x
XRA	A=A x rg			AF	1 4	A8	1 4	A9	1 4	AA	1 4	AB	1 4	AC	1 4	AD	1 4	AE	1 4	x	x	x	x	x
CPI	A - K	FE	2 7																	x	x	x	x	x
CMP	A - rg			BF	1 4	B8	1 4	B9	1 4	BA	1 4	BB	1 4	BC	1 4	BD	1 4	BE	1 4	x	x	x	x	x

Registerpaarfunktionen:

Mnem	Wirkung	B (B,C)		D (D,E)		H (H,L)		SP		PSW, A		Bedingungen				
		Op	B Z	Op	B Z	Op	B Z	Op	B Z	Op	B Z	N	Z	H	P	C
LXI	rp=K	01	3 10	11	3 10	21	3 10	31	3 10							
INX	rp=rp+1	03	1 5	13	1 5	23	1 5	33	1 5							
DCX	rp=rp-1	0B	1 5	1B	1 5	2B	1 5	3B	1 5							
DAD	HL=HL+rp	09	1 10	19	1 10	29	1 10	39	1 10							x
PUSH	rp-Stack	C5	1 11	D5	1 11	E5	1 11			F5	1 11					
POP	Stack-rp	C1	1 10	D1	1 10	E1	1 10			F1	1 10					

Registerpaar HL:

Mnem	Wirkung	Op	B Z
LHLD	HL(=Sp	2A	3 16
SHLD	HL=)Sp	22	3 16
XCHG	HL(=)DE	EB	1 4
XTHL	HL(=)St	E3	1 18
PCHL	HL=)PC	E9	1 5
SPHL	HL=)St	F9	1 5

Sprungbefehle:

	Sprung			Unterprogramm			Rücksprung		
Bedingung	Mnem	Op	B Z	Mnem	Op	B Z	Mnem	Op	B Z
keine	JMP	C3	3 10	CALL	CD	3 17	RET	C9	1 10
Z = 1	JZ	CA	3 10	CZ	CC	3 11/17	RZ	C8	1 5/11
Z = 0	JNZ	C2	3 10	CNZ	C4	3 11/17	RNZ	C0	1 5/11
C = 1	JC	DA	3 10	CC	DC	3 11/17	RC	D8	1 5/11
C = 0	JNC	D2	3 10	CNC	D4	3 11/17	RNC	D0	1 5/11
N = 1	JM	FA	3 10	CM	FC	3 11/17	RM	F8	1 5/11
N = 0	JP	F2	3 10	CP	F4	3 11/17	RP	F0	1 5/11
p = 1	JPE	EA	3 10	CPE	EC	3 11/17	RPE	E8	1 5/11
p = 0	JPO	E2	3 10	CPO	E4	3 11/17	RPO	E0	1 5/11

RST Befehle:

Mnem	Op	B Z	Adresse
RST 0	C7	1 11	0000
RST 1	CF	1 11	0008
RST 2	D7	1 11	0010
RST 3	DF	1 11	0018
RST 4	E7	1 11	0020
RST 5	EF	1 11	0028
RST 6	F7	1 11	0030
RST 7	FF	1 11	0038

Akkumulator:

Mnem	Wirkung	Op	B Z	Bedingungen N	Z	H	P	C
LDA	A=Sp	3A	3 13					
LDAX B	A=Sp(B)	0A	1 7					
LDAX D	A=Sp(D)	1A	1 7					
STA	Sp=A	32	3 13					
STAX B	Sp(B)=A	02	1 7					
STAX D	Sp(D)=A	12	1 7					
RLC	L. rot.	07	1 4					x
RRC	R. rot.	0F	1 4					x
RAL	L. rot.	17	1 4					x
RAR	R. rot.	1F	1 4					x
CMA	A= n A	2F	1 4					
DAA	Dez.Kor	27	1 4	x	x	x	x	x
XRA A	A= 0	AF	1 4	0	1	0	0	0
ORA A	Carry=0	B7	1 4	x	x	x	x	0
STC	Carry=1	37	1 4					x
CMC	Cy=n Cy	3F	1 4					x
EI	Int.erl	FB	1 4					
DI	Int.unt	F3	1 4					
HLT	anhalt.	76	1 7					
NOP	keine	00	1 4					

13 Erweiterung der eingebauten Funktionen

Wie schon in den Kapiteln über Addition und Subtraktion bemerkt wurde, sind dies die beiden einzigen arithmetischen Funktionen. Weitergehende Operationen sind nicht im Befehlssatz enthalten und müssen daher selber implementiert werden. Für Multiplikation und Division soll dies in diesem Kapitel geschehen.
Alle aufgeführten Funktionen sind als Unterprogramme ausgelegt worden. Sie werden also per CALL oä aufgerufen!

13.1 Multiplikation

Die Multiplikation kann vereinfacht auf wiederholte Addition zurückgeführt werden. Dieses Verfahren ist jedoch sehr zeitintensiv, da unter Umständen sehr häufig addiert werden muß. Um diesen Aufwand verkleinern zu können benutzt man ein Verfahren, daß wohl jedem vom handschriftlichen Rechnen her bekannt ist.
Beispiel:

```
0101 x 0101
-----------
0000
 0101
  0000
   0101
--------
0011001
```

Bei 8 Bit Argumenten kommt man mit diesem Verfahren mit höchstens 8 Durchläufen aus und dies kann unter Umständen eine erhebliche Zeitersparnis bedeuten!

Das folgende Unterprogramm führt eine Multiplikation zweier 8 Bit Zahlen durch. Die beiden Argumente befinden sich im D, bzw E Register. Das 16 Bit Ergebnis befindet sich nachher im HL Register. Dieses Programm benötigt ca. 0.31 ms für die Multiplikation

```
MUL     LXI   H,:200
        MOV   A,D
        MVI   D,0
LOOP    ADD   A
        JNC   SHIFT
        DAD   D
SHIFT   DAD   H
        JNC   LOOP
        ADD   A
        RNC
        DAD   D
        RET
```

Das HL Register bildet in diesem Programm zugleich den Ergebnisparameter und die Abbruchbedingung! Die :200, die dem HL Register am Anfang zugewiesen wird setzt das L Register auf Null und im H Register wird das 2. Bit gesetzt. Dieses Bit wird durch den DAD H solange nach links geschoben, bis ein Carry auftritt (7-Durchläufe) und damit ist die Schleife beendet. Durch den DAD D wird dieses Bit nicht verändert da auf das HL Register immer nur eine 8 Bit Zahl addiert wird!

Das folgende Programm führt eine 16 Bit Multiplikation durch. Die beiden jeweils 16 Bit breiten Argumente befinden sich in den Registerpaaren BC und DE. Das entstehende 32 Bit breite Produkt befindet sich in den Registern DE/HL, wobei sich die beiden höchstwertigen Bytes in den Registern D und E befinden. Die benötigte Zeit beträgt hierfür 1023 Takt Zyklen, also ca eine Milli Sekunde!

```
MULT    MOV   A,E
        PUSH  D
        CALL  BMULT
        XTHL
        PUSH  PSW
        MOV   A,H
        CALL  BMULT
        MOV   D,A
        POP   PSW
        ADD   H
        MOV   E,A
        JNC   NC1
        INR   D
NC1     MOV   H,L
        MVI   L,0
        POP   B
        DAD   B
        RNC
        INX   D
        RET
```

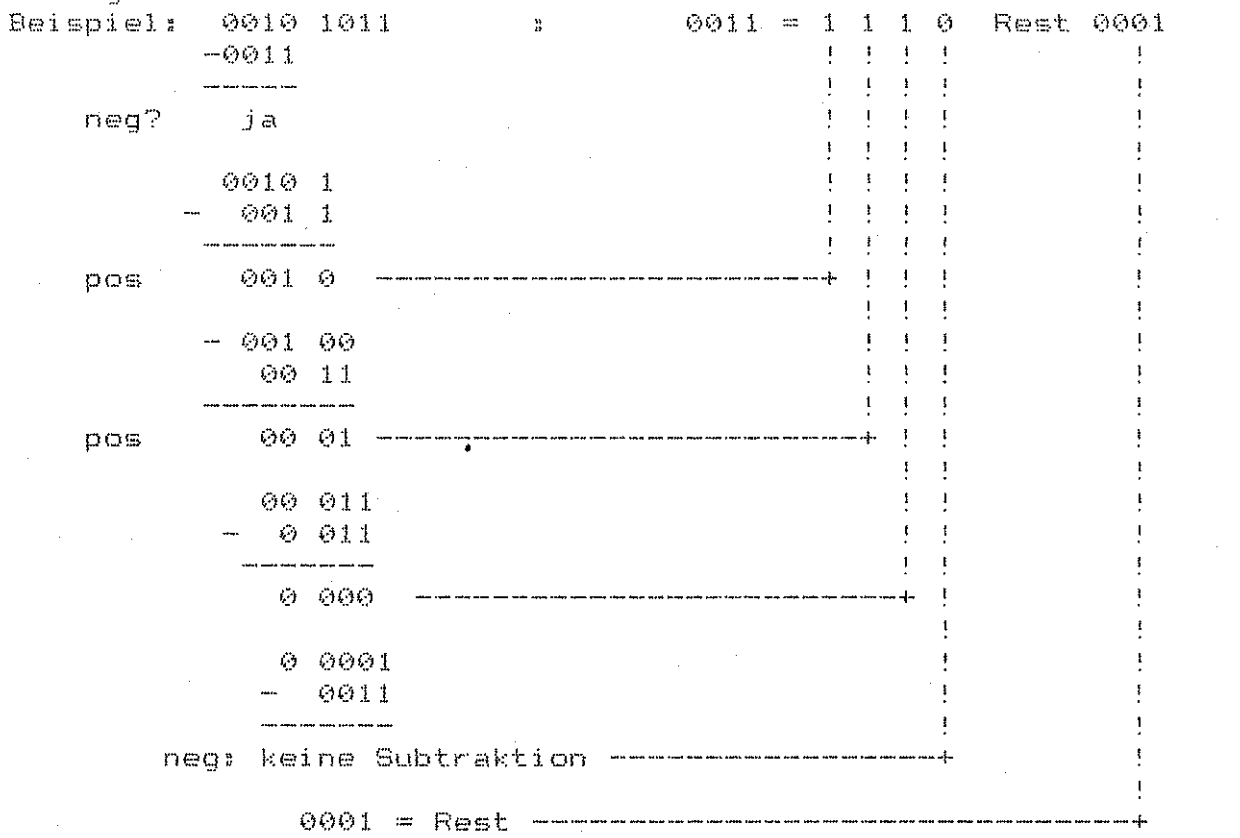
BMULT führt eine 8 x 16 Bit Multiplikation durch. Im BC Registerpaar wird ein 16 Bit und im Akku ein 8 Bit Faktor erwartet. Das Produkt steht nach der Multiplikation in den Registern A-HL. Diese Prozedur benötigt im schlechtesten Fall 424 Takt Zyklen - also ca. 0.42 ms.

```
BMULT   LXI   H,0
        LXI   D,7
        ADD   A
LOOP1   JNC   ZERO
        DAD   B
        ADC   D
ZERO    DAD   H
        ADC   A
        DCR   E
        JNZ   LOOP1
        RNC
        DAD   B
        ADC   D
        RET
```

Eine weitere Multiplikationsroutine befindet sich im ROM an der Adresse #DE8F. Diese Routine führt eine 16 x 8 Bit Multiplikation der Register HL und A durch. Das Ergebnis befindet sich im HL Register! Ein Carry zeigt an, ob bei der Rechnung ein OVERFLOW entstanden ist, denn 16 x 8 Bit = max. 24 Bit großes Ergebnis! Außer dem HL Register bleiben alle anderen Register erhalten!

13.2 Division

Auch die Division kann auf mehrfache Subtraktion zurückgeführt werden, jedoch ist auch hier das Verfahren sehr zeitaufwendig, deshalb wird hier ebenfalls das handschriftliche Verfahren durchgeführt.



Die folgende Routine teilt eine 16 Bit Zahl im HL Register durch eine 8 Bit Zahl im Akku. Das Ergebnis befindet sich im L Register und der evtl. entstehende Rest steht im H Register. Das Carry Flag zeigt einen Fehler an: Ein Fehler kann auftreten, wenn der Quotient = 0 (DIVISION BY ZERO) ist, oder der Quotient größer als 8 Bit wird! Im Fall eines aufgetretenen Fehler wird das Carry Flag gesetzt!

```
DIV     PUSH  B
        MOV   B,A
        MVI   C,8
        MOV   A,H
        CMP   B
        JC    D1
        POP   B
        STC
        RET
D1      DAD   H
        JC    D2
        MOV   A,H
        CMP   B
        JC    D3
D2      MOV   A,H
        ORA   A
        SBB   B
        MOV   H,A
        INX   H
D3      DCR   C
        JNZ   D1
        POP   B
        ORA   A
        RET
```

14 Ein-/ Ausgaben

Dies ist ein recht schwieriges Problem bei der Maschinenprogrammierung. Für das Thema Ausgaben verweise ich auf den Artikel von Jan Boerrighter in der Ausgabe 6/84. Hier wurden alle wichtigen PRINT Routinen vorgestellt. Sie können diese Routinen einfach über einen CALL mit entsprechender Startadresse aufrufen und so Ihre Ausgaben auf Bildschirm und/ oder Drucker bringen!

Außerdem möchte ich an dieser Stelle überhaupt auf das Buch von Jan Boerrighter 'DAI FIRMWARE MANUAL' verweisen. In diesem Buch sind alle ROM Routinen dokumentiert enthalten. Zu jeder Routine wird angegeben, was sie ausführt und welche Register hierbei verändert werden. Wenn Sie mehr über ROM Routinen wissen wollen ist dieses Buch eine sinnvolle Anschaffung!

14.1 Ein Zeichen von der Tastatur lesen

Dies ist wohl von allen Eingabemöglichkeiten die wichtigste Operation. Hierzu steht eine GETC Funktionen wie in BASIC zur Verfügung. Diese Funktion liefert als Ergebnis im Akku auch analoge Ergebnisse:

A=0 - keine Taste gedrückt,
sonst enthält der Akku den ASCII Wert der gedrückten Taste. Diese Routine wird durch CALL :D6BB aufgerufen. Diese Funktion hat nur den gleichen Nachteil wie das GETC der BASIC Version 1.0 - diese Funktion führt häufig zum Prellen der Tasten - ein Zeichen wird also doppelt oder dreifach erzeugt, obwohl nur eine Taste gedrückt wurde!

Dies läßt sich mit der Routine ab :D6BE unterbinden! Jedoch muß hierbei erst das Flag :2B9 auf Null gesetzt werden! Dies braucht jedoch nur einmal im Programm an beliebiger Stelle geschehen. Das Programm um die Tastatur abzufragen lautet also wie folgt:

```
        XRA   A
        STA   :2B9
         :
GETC    CALL  :D6BE
        JZ    GETC
        JC    BREAK
        CPI   usw.
```

Wie aus diesem Programmstück ersichtlich ist, liefert diese, wie auch die :D6BB Routine als Ergebnis neben dem Zeichen auch die Statusflags Zero und Carry. Carry gesetzt bedeutet in dieser Routine: die BREAK Taste wurde gedrückt. Es wird also nicht sofort wie im BASIC die Meldung 'BREAK IN LINE' ausgegeben. Vielmehr wird durch das GETC auch ein HARD BREAK unterdrückt, da der Break Zähler wieder zurückgesetzt wird!

14.2 Eingabe eines Strings

Hierfür gibt es keine ROM Routine, die einfach anzusprechen wäre.
Aus diesem Grund stelle ich hier meine eigene Routine vor. Sie wird von mir häufig z.B. zum Einlesen des Namens eines einzulesenden Files oä verwendet.

Da diese Routine noch zwei Unterroutinen zum Setzen und Löschen des Cursors verwendet - welche sich allerdings auf E-ROM Bank 2 befinden, mußten noch Prozeduren zum Umschalten der ROM Banken geschaffen werden.

Der Aufruf erfolgt durch LXI H,Abspeicheradresse
CALL STR
bzw. wenn sich das Programm sowieso schon auf E-ROM Bank2 befindet genügt auch ein LXI H,Abspeicheradresse
CALL STRING
Nach dem Aufruf befindet sich an der angegebenen Abspeicheradresse eine Textkette mit vorangestelltem Längenbyte, welches aussagt, wieviel Zeichen noch folgen.
Wurde während der Eingabe BREAK gedrückt, so kehrt das Programm mit gesetztem Carry Flag zurück!

Als Besonderheit dieser Routine ist aufzuführen, daß diese Routine nur Eingaben bis zu einer bestimmten Länge (hier 45 Zeichen) zuläßt. Diese Zahl kann beliebig verändert werden, nur sollte die Startposition der Abfrage plus diese Maximalbreite den rechten Rand der Zeile nicht überschreiten, da hierauf dann keine Rücksicht genommen wird!
Diese Routine schützt übrigens auch den Text, der sich links von der 1. Abfrageposition befindet. Ein möglicher Aufruf wäre z.B.

```
        LXI     H,TEXT
        CALL    :DB32
        LXI     H,BUFFER
        CALL    STR
        JC      BREAK
         :
TEXT    DT$     0C"Filename: "
```

Das DT$ ist ein Pseudomnemonic des AHT Assemblers. Der nachfolgende Text wird in Hex-Ziffern übersetzt, die Länge des Textes wird berechnet und vor diese Textkette wird diese Länge geschrieben.

```
CUR.POS  EQU    :72
CUR.DEL  EQU    :E36B
CUR.SET  EQU    :E330
         /
STR      CALL   BANK2
         CALL   STRING
         JMP    BANK0
         /
```

```
STRING  SHLD  ADR
        LHLD  CUR.POS   CUR.POS enthält die akt. Cursor Pos.
        MVI   C,0
        PUSH  H
ST:GET  CALL  GETC      Tastaturabfrage
        JC    ST:EX     BREAK ?
        CPI   13
        JZ    ST:RET    RETURN ?
        CPI   8
        JZ    ST:DEL    CHAR DEL ?
        CPI   32
        JC    ST:GET    Textzeichen kleiner 32 ignorieren
        MOV   B,A
        MOV   A,C
        CPI   45        45 Max. Eingabelänge
        JZ    ST:GET  - kann jedoch beliebig verändert werden
        INR   C
        LHLD  CUR.POS
        CALL  CUR.DEL
        MOV   M,B       eingelesene Zeichen auf den Bildschirm
        DCX   H         bringen. Cursor auf nächste Position
        DCX   H         setzten
        CALL  CUR.SET
        JMP   ST:GET
ST:DEL  MOV   A,C       Zeichen löschen !
        ORA   A         Cursor am linken Rand => ignorieren
        JZ    ST:GET
        DCR   C
        CALL  CUR.DEL   letzte eingegebene Zeichen
        INX   H         mit Leerzeichen =32 überschreiben
        INX   H
        MVI   M,32
        CALL  CUR.SET
        JMP   ST:GET
ST:RET  MOV   A,C       RETURN !
        LHLD  ADR
        MOV   M,A       Textlänge abspeichern
        ORA   A
        POP   H
        RZ              falls nur RETURN gedrückt wurde=> RET
        XCHG
        LHLD  ADR
        INX   H
        XCHG
S:RL    MOV   A,M       sonst den Text vom Bildschirm
        STAX  D         zur angegebenen Adresse kopieren
        INX   D
        DCX   H
        DCX   H
        DCR   C
        JNZ   S:RL
        RET
ST:EX   POP   H         BREAK !
        RET
        /
BANK0   PUSH  PSW
        DI
        LDA   :40
        ANI   :3F
        STA   :40
        STA   :FD06
        EI
        POP   PSW
        RET
        /
```

```
BANK2   PUSH  PSW
        DI
        LDA   :40
        ANI   :3F
        ORI   :80
        STA   :40
        STA   :FD06
        EI
        POP   PSW
        RET
```

14.3 Ausgabe einer Textzeile

Diese Routine schreibt einen String direkt ab einer angegebenen Adresse in den Bildschirm. Wenn im Text ein Zeichen mit MSB=1 geschrieben wurde, so wird dieses Zeichen mit Farbbyte :FF geschrieben; Dadurch wird die zweite Farbbank angewählt und das Zeichen oder auch mehrere können hervorgehoben dargestellt werden!

Außerdem wird der Rest der Textzeichen dieser Zeile überschrieben - dies kann jedoch durch den Aufruf

```
        STC
        CALL  LINE+3
```

unterbunden werden!
Beim Aufruf dieses Unterprogramms muß das DE Registerpaar die Anfangsbildschirmadresse enthalten und das HL Registerpaar muß auf das Längenbyte eines Strings zeigen!

```
Bsp:    LXI   D,:BFE5
        LXI   H,TEXT
        CALL  LINE
         #


LINE    ORA   A
        PUSH  PSW
        MOV   B,M
        INX   H
        MVI   C,60        Zeilenbreite wird als 60 angenommen
PR.L    MOV   A,M
        STAX  D           Ein Zeichen in den Bildschirm bringen
        INX   H
        DCX   D
        DCX   D
        DCX   D
        ADD   A           Hintergrundfarbbyte setzen!
        SBB   A
        STAX  D
        INX   D
        DCR   C
        DCR   B           alle Zeichen abgespeichert?
        JNZ   PR.L
        POP   PSW
        RC                Carry=Rest der Zeile nicht löschen
        XCHG
PR:L2   MVI   M,32        Den Rest der Zeile mit Blanks
        DCX   H           überschreiben!
        DCX   H
        DCX   H
        MVI   M,0
        INX   H
        DCR   C
        JNZ   PR:L2
        RET
```

14.4 Eingabe einer Zahl

Diese Routine erwartet die Eingabe einer Zahl aus einem beschränkten Zahlenintervall und einem gleichzeitig begrenzten Bildschirmbereich! Es ist also möglich als Eingabewert z.B. nur Zahlen aus dem Intervall [5,10] oä zuzulassen.

Die Aufruf erfolgt folgendermaßen:

1) Der Kommentartext zur Eingabe ist auf dem Bildschirm aus zugeben.
2) An der Stelle, wo die Eingabe geschehen soll, sind Punkte (mit MSB=1) zu schreiben, hierfür kann die oben vorge stellte Routine LINE verwendet werden! Nur auf diesen Punkten werden Eingaben angenommen!!!
3) Der Cursor ist dann auf den ersten Punkt zu setzen.
4) In den Registerpaaren BC und DE befinden sich die Eingabe-schranken [BC,DE]
5) Anwählen der BANK2
   Aufruf der Routine ZAHLINP
   ggf. wieder BANK0 anwählen
6) Wenn das Carry Flag gesetzt ist, wurde BREAK gedrückt, sonst befindet sich im HL Register die eingegebene Zahl.

```
Bsp:        LXI   H,LOGO
            LXI   D,:BFE5
            CALL  LINE
            CALL  BANK2
            CALL  CUR.DEL
            LXI   H,:BFE5-12
            CALL  CUR.SET
            LXI   B,5           Bereich von 5 bis 999
            LXI   D,999
            CALL  ZAHLINP
            JC    BREAK
             :

LOGO        DT$   "Zahl: "'"..."
```

Der Apostroph vor einer Zeichenkette gibt im AHT Assembler an, das die nachfolgenden Zeichen mit MSB=1 dargestellt werden sollen!

```
ADD         EQU   :DE30       HL = HL + A
COM         EQU   :DE14       Vergl  HL < DE ?
GETC        EQU   :D6BE
ZAHLINP     PUSH  B
            PUSH  D
            LHLD  CUR.POS
            PUSH  H
Z.1         CALL  GETC        Zeichen einlesen
            JZ    Z.1
            JC    Z.E         BREAK ?
            CPI   13          RETURN ?
            JZ    Z.R
            CPI   8           CHAR DEL ?
            JZ    Z.D
            CPI   48          Es sind sonst nur Zahleingaben zu-
            JC    Z.1         gelassen!
            CPI   "9"+1
            JNC   Z.1
```

```
        MOV   B,A
        LHLD  CUR.POS
        CALL  CUR.DEL
        DCX   H
        DCX   H
        DCX   H          Ziffer nur dann auf den Schirm
        MOV   A,M        schreiben, wenn das Hintergrund
        INR   A          Farbbyte dieses Zeichens =FF ist
        JNZ   Z.2
        INX   H
        INX   H
        INX   H
        MOV   M,B        Zeichen auf den Bildschirm bringen
        DCX   H
        DCX   H
        CALL  CUR.SET
        JMP   Z.1
Z.2     INX   H
        INX   H
        INX   H
        CALL  CUR.SET
        JMP   Z.1
Z.D     LHLD  CUR.POS    Ziffer löschen!
        DCX   H
        MOV   A,M
        INR   A
        JNZ   Z.1        Jedoch nur auf dem markierten Bereich
        INX   H
        INX   H
        INX   H
        CALL  CUR.DEL    letzte Ziffer mit einem Punkt über-
        MVI   M,"."      schreiben
        CALL  CUR.SET
        JMP   Z.1
Z.R     POP   H
        PUSH  H
        XCHG
        CALL  CUR.DEL
        LXI   H,0        HL soll das Ergebnis enthalten
Z.R1    LDAX  D
        MOV   B,A        Ein Zeichen wieder aus dem Bild-
        CPI   "."        schirm lesen
        JZ    Z.RE
        DCX   D
        DCX   D
        DCX   D
        LDAX  D
        INR   A
        JNZ   Z.RE
        INX   D
        MVI   A,10       HL mit Zehn multiplizieren
        CALL  MUL
        MOV   A,B
        SUI   48
        CALL  ADD        und die nächste Ziffer aufaddieren
        JMP   Z.R1
Z.RE    XCHG
        POP   H
        CALL  CUR.SET
        XCHG
        POP   D
        POP   B
        INX   D
        CALL  COM        Vergleichen, ob die Zahl in den
        DCX   D          erlaubten Grenzen liegt
        JNC   Z.NE
```

```
          PUSH   D
          MOV    D,B
          MOV    E,C
          CALL   COM
          POP    D
          RNC
Z.NE      LHLD   CUR.POS    Sonst die gemachten Eingaben wieder
          CALL   CUR.DEL    (durch Punkte) löschen
          PUSH   H
Z.N1      MVI    M,'"."
          DCX    H
          DCX    H
          DCX    H
          DCX    H
          DCX    H
          MOV    A,M
          INR    A
          JNZ    $+9
          INX    H
          INX    H
          INX    H
          JMP    Z.N1
          POP    H
          CALL   CUR.SET
          JMP    ZAHLINP
Z.E       POP    H          BREAK gedrückt !
          POP    D          Abbruch
          POP    D
          JMP    CUR.SET
```

Im folgenden nun die vollständige Liste aller RST 1,4,5 DATAs:

RST 1 / DATA x

```
00:   Eine Basic Zeile encodieren
03:   Eine Zeilennummer encodieren
06:   Eine Konstante encodieren
09:   UT Aufruf durchführen
0C:   Disk Bootstrap
0F:   Heap Verwaltung
12:   Ein Tastaturzeichen decodieren
15:   Eingaben von Tastatur oder einer entsprechenden
      Benutzerroutine holen
```

RST 5 / DATA x

```
00:   Initialisieren nach RESET
03:   Ausgabe des Akkus an den Bildschirm
06:   COLORT (HL)
09:   CURSOR L,H
0C:   L=CURX, H=CURY, DE=Breite/Höhe des Bildschirms
0F:   CURSOR-Mode setzen
12:   Cursor <=> Zeichen blinken
15:   Lese Zeichen vom Bildschirm in den Akku an der Pos C
18:   MODE (A)
1B:   COLORG (HL)
1E:   DOT HL,C A
21:   DRAW HL,C DE,B A
24:   FILL HL,C DE,B A
27:   A=SCRN (HL,C)  / DE=XMAX / B=YMAX
2A:   Initialisieren des Editors
2D:   Ausgabe des Zeichens im Akku an den Editor
```

RST 4 / DATA x (ROM Bank 1 - Mathematik)

00:	FADD	x + y	3F:	ASIN	arcsin x
03:	FSUB	x - y	42:	ACOS	arccos x
06:	FMUL	x * y	45:	ATN	arctan x
09:	FDIV	x / y	48:	FIX	x -> n
0C:	GETMEM	macc = (HL)	4B:	FLOAT	n -> x
0F:	PUSTMEM	(HL) = macc	4E:	IADD	n + m
12:	GETREG	macc = ABCD	51:	ISUB	n - m
15:	PUTREG	ABCD = macc	54:	IMUL	n * m
18:	FABS	abs x	57:	IDIV	n / m
1B:	FNEG	-x	5A:	MOD	n mod m
1E:	INT	int x	5D:	IABS	abs n
21:	FRAG	frac x	60:	INEG	-n
24:	POWER	x ^ y	63:	IAND	n iand m
27:	LOG	ln x	66:	IOR	n ior m
2A:	EXP	exp x	69:	IXOR	n ixor m
2D:	LOGT	log x	6C:	INOT	inot n
30:	ALOG	10 ^ x	6F:	SHL	n shl m
33:	SQR	sqrt x	72:	SHR	n shr m
36:	SIN	sin x	75:	SAVEA $	$-Array auf-bereiten
39:	COS	cos x			
3C:	TAN	tan x			

x: FPT, im macc n: INT, im macc
y: FPT, in (HL) m: INT, in (HL)

Hiermit endet nun diese kleine Einführung in die Maschinenprogrammierung des 8080 Prozessors.
Ich hoffe, daß Sie ein wenig Gefallen an dieser Art der Programmierung gefunden haben und sich vielleicht nun auch selbst (kleinere) Programme zutrauen. Oder sehen Sie doch einmal in älteren Ausgaben der Clubzeitung nach. Hier finden Sie viele nützliche Tips und Anregungen und zum Teil auch gut dokumentierte Programmlistings. Und auch hier gilt: Übung macht den Meister! In diesem Sinne wünsche Ihnen noch viel Spaß beim Programmieren in Assembler!

Uwe Wienkop

Abspeichern von Daten von Maschinenprogrammen aus:

Autor: Uwe Wienkop

Der DAI enthält in seinem ROM viele nützliche Routinen, die man auch sehr gut von eigenen Maschinenprogrammen her nutzen kann. Vielfach besteht der Wunsch die - durch ein MP erzeugten - Daten auf externen Medien abzulegen und sie später wieder zu benutzen. Dieser Beitrag soll nun erläutern, wie man diese ROM Routinen nutzen kann und welche Konventionen man einhalten muß:

Daten (d.h. Programme, UT-Files und Arrays) werden beim DAI immer nach dem gleichen Schema behandelt:

1. Der Name wird abgespeichert; Hierbei wird auch gleich die Kennziffer (d.h. 0-Basic,1-UT-File und 2-Array) mit abgespeichert
2. Nun folgen zwei Blöcke mit Daten:
   - Beim Basic Programm beinhaltet der erste Block den Programmtext bis zum Anfang der Symboltabelle und der zweite Block beinhaltet die Symboltabelle
   - Bei UT-Files steht im 1. Block nur die Startadresse des Files. D.h. ab dieser Adresse wird ein UT-File später wieder eingelesen.
     Der 2. Block beinhaltet dann die eigentlichen Daten
   - Arrays tragen im 1.Block die Array Kennziffer (#00-FPT, #10-INT und #20-String) und im 2.Block folgen wieder die Daten
3. Es folgt ein Write Close, d.h. die Datei wird geschlossen Der Endepiepser wird geschrieben und der Cassetten (MDCR) Motor wird abgeschaltet.

Dies sind die Konventionen, die vom Basic erzeugt werden. Hieran braucht man sich bei eigenen Routinen jedoch nur bedingt zu halten; so kann z.B. eine Abänderung dieses Verfahrens als Kopierschutz verwendet werden (z.B. 3 Blöcke abspeichern usw.)

Ebenfalls ist es möglich auch andere Datei-Typen als die vom Basic vorgegebenen zu verwenden. Beim Check wird lediglich abgefragt, ob der Typ der Datei kleiner als ASC("3") ist. Hiermit kann man also sehr viele Typen für eigene Zwecke verwenden. So benutzt z.B. TINYPASCAL den Typ '$' für seine Files. Der Vorteil besteht darin, daß man seine Files später leichter wiedererkennen kann und daß Verwechselungen mit anderen Filetypen ausgeschlossen sind.

Ich möchte an dieser Stelle aber auch an die 'armen' Diskettenbesitzer verweisen. Diese freuen sich über diese speziellen Dateitypen sicher nicht, denn das DOS kann nur die Standardtypen erkennen und dementsprechend handeln. Es wäre daher sicher gut, wenn Ihr Daten nur entsprechend den Standardkonventionen abspeichern würdet, so daß alle Benutzer des Programms mit Einlesen und Abspeichern keine Probleme hätten.

Nun jedoch zum eigentlichen Verfahren:
Abspeichern:

1) Länge & Name stehen als String im Speicher
   HL zeigt auf die Länge
   A enthält die Kennziffer, z.B. '0', '1' oder '2' in ASCII
   CALL WOPEN (#2C5) <<< Write Open >>>

2) HL zeigt auf die Startadresse des abzuspeichernden Blocks
   DE enthält die Anzahl der abzuspeichernden Bytes
   CALL WBLK (#2C8) <<< Write Block >>>
   Dieser Teil wird zweimal durchgeführt, wegen der Abspeicherung in zwei Blöcken

3) CALL WCLOSE (#2CB) <<< Write Close >>>

Einlesen:

1) HL zeigt auf den Suchnamen;
   bei Länge Null wird das nächstes File genommen
   A enthält bei Arrays den Arraytyp (#00, #10, #20)
   B Filetyp (Kennziffer: '0','1')
   C-00 - d.h. die Namen der überlesenen, bzw. des eingelesenen Files werden nicht ausgedruckt
   C-FF - die Namen werden ausgedruckt
   CALL ROPEN (#2CE) <<< Read Open >>>

2) HL - Adresse, wo der erste Block hingeschrieben werden soll
   DE - Schutzadresse, die nicht mehr überschrieben werden darf
   CALL RBLK (#2D1) <<< Read Block >>>

   DE bleibt bei diesem Aufruf erhalten
   HL steht hinter dem letzten eingeschriebenen Byte

   Carry gesetzt: kein Fehler bei der Datenübertragung
   Carry ungesetzt: Loading Error
   Der Akku enthält in diesem Fall das Fehler Byte:
   A=0,1,2,3 entsprechend dem Basic LOADING ERROR

3) HL und DE entsprechend b)
   CC RBLK (#2D1)
   CALL RCLOSE (#2D4) <<< Read Close >>>
   EI
   Danach sollte eine Fehlerbehandlung wie bei 2)
   beschrieben durchgeführt werden.

Ich habe hier immer den Aufruf über die Vektoren aufgeführt: D.h. in den Adressen von #2C5 bis #2D6 befinden sich Sprünge zu den Abspeicher- und Lade Routinen. Als alternative Möglichkeit könnte man auch die Routinen direkt anspringen. Hiervon ist aber unbedingt abzuraten, denn diese Vektoren werden automatisch beim Wechsel auf ein anderes Speichermedium mit umgeschaltet und sie zeigen dann auf die neuen Routinen. So werden diese Zeiger z.B. mit dem DCR Befehl auf die Routinen im TOS umgelegt, so daß nun bei den Schreib-/Lesebefehlen automatisch die DCR angesprochen wird. Das gleiche gilt natürlich auch für die Diskette. Wenn man nun anderen das Programm gibt und diese Personen arbeiten mit einem anderen Speichermedium, so muß nicht das Programm geändert werden, sondern das Betriebssystem übernimmt diese Aufgabe.
Noch ein Wort zum Abspeichern: Die Speicherzellen #100 u. #101 bilden ein Doppel-Flag. Ist dieses Flag=0, so wird momentan kein BASIC-Programm abgearbeitet, d.h. das System nimmt an es wäre in der Kommandostufe. In diesem Fall wird beim Abspeichern der Text 'SET RECORD, START TAPE, TYPE SPACE' ausgegeben und der Computer wartet auf das Betätigen der Leertaste. Ist dieses Flag <> 0, so wird dieser Text nicht ausgegeben und die Daten werden sofort abgespeichert. Die Diskette reagiert sinnvollerweise nicht auf dieses Flag; ebenso wird bein Einlesen ein #FF im C-Register ignoriert, da hier sowieso nur mit Namen gelesen werden kann

Zur Verdeutlichung noch ein kleines Beispiel:
Ich möchte die Daten, die mein Programm erzeugt hat und die sich in den Adressen von #3000 bis #3FFF befinden extern abspeichern. Als Datentyp wähle ich '1' für UT-Files. Der Name soll hierbei "Testdatei" heissen

Abspeichern:

```
WOPEN   EQU  :2C5
WBLK    EQU  :2C8
WCLOSE  EQU  :2CB
HELP    RES  2                  Help ist ein Hilfsregister,
         :                      in dem die Startadresse abge-
         :                      speichert wird
         :
        LXI  H,:3000            Startadresse nach HL bringen
        SHLD HELP               und den Wert von HL in Help
        /                       abspeichern
        LXI  H,TEXT             HL mit der Adresse des Datei-
        /                       namens laden
        MVI  A,"1"              ASC("1") in den Akku bringen
        CALL WOPEN              Write open durchführen
        LXI  H,HELP             HL zeigt nun auf die Hilfs-
        /                       variable Help
        LXI  D,2                Es sollen zwei Bytes im 1. Block
        /                       abgespeichert werden
        CALL WBLK               1. Block abspeichern
        LXI  H,:3000            HL zeigt nun auf den Anfang der
        /                       Daten
        LXI  D,:3FFF-:3000+1    DE enthält die Anzahl der abzu-
        /                       speichernden Bytes
        CALL WBLK               2. Block abspeichern
        CALL WCLOSE             Datei schließen
         :
TEXT    DT$  "Testdatei"        DT$ ist ein Spezial Befehl des
         :                      AHT Assemblers; Der Text wird
         :                      mit vorangestelltem Längenbyte
         :                      abgespeichert
```

Einlesen:

```
ROPEN   EQU  :2CE
RBLK    EQU  :2D1
RCLOSE  EQU  :2D4
        :
        LXI  H,TEXT             Es soll das File Testdatei
        /                       wieder eingelesen werden
        MVI  B,"1"              Filetyp "1"
        MVI  C,:FF              Der Name soll ausgegeben werden
        CALL ROPEN              Lesen starten
        LXI  H,HELP             Die Adresse wird nach Help geschrieben
        LXI  D,HELP+2           HELP+2 darf nicht mehr über-
        /                       schrieben werden
        CALL RBLK               1. Block lesen
        LHLD HELP               Startadresse aus Help lesen
        LXI  D,:4000            :4000 darf nicht mehr über-
        /                       schrieben werden
        CC   RBLK               falls beim Lesen des ersten Blocks
        /                       kein Lesefehler auftrat, dann
        /                       auch den zweiten Block lesen
        CALL RCLOSE             Lesen schließen
        EI                      Interrupts wieder erlauben
        JNC  ERROR              Fehlerbehandlung durchführen
```

Die RS-232-C - Schnittstelle

Ein Standard und grosse Verwirrung ?!-

bearbeitet von R.Hahn

In diesem Beitrag, der aus mehreren Folgen besteht, wollen wir die RS-232-C-Schnittstelle unter die Lupe nehmen, und die Verachtung und das Missverstaendnis gegenueber dieser Schnittstelle nach und nach abbauen.

Gewoehnlich werden sanftmuetige Menschen zum Wahnsinn getrieben, wenn sie ihren Computer mit einem Drucker oder gar einem anderen Rechner ueber diese Schnittstelle verbinden wollen. Wenn beide Einheiten mit einem Standard EIA-RS-232-C-Kabel fuer Gegenbetrieb (Full Duplex) verbunden werden, weigert sich nicht nur der Drucker zu drucken, sondern er legt sogar den Rechner lahm.

Da mit neuen angebotenen Systemen fuer unseren alten DAI, sowie fuer den Betrieb mit Akustikkopplern diese Schnittstelle mehr und mehr benutzt wird, wollen wir mehr ueber sie erfahren.

Was bedeutet RS-232-C, die fuer die Verbindung von Mikrocomputer und Zusatzgeraeten verantwortlich ist:

Recommended Standard Number 232, Revision C from Electronic Industry Association

Die meisten Schwierigkeiten, die mit der RS-232-C-Schnittstelle verbunden sind, haben ihre Ursache darin, dass sie fuer Aufgaben eingesetzt wird, fuer die sie gar nicht konstruiert wurde.

Also beginnen wir diese Schwierigkeiten zu verstehen !

Setzen wir voraus, dass uns bekannt ist was ein Bit ist. Obwohl dieses Bit eine intellektuelle Konstruktion, ist es physikalisch doch nur eine Spannung, deren Groesse den Wert dieses Bit angibt.

Innerhalb unseres Computers werden diese Bit im 8-Bit-Format (1 Byte) mit nebeneinander liegenden Leitungen uebertragen. Diese Uebertragung bezeichnen wir als parallelen Transfer. Da alle acht Bits zur gleichen Zeit ihr Ziel erreichen, kann dieser parallele Datentransfer mit sehr hohen Geschwindigkeiten durchgefuehrt werden.

Fuer diesen Datentransfer ist eine kontrollierte Umgebung, die Beachtung elektrischer Eigenschaften, wie Widerstand, Kapazitaet und Induktivitaet erforderlich.

Solange der parallele Datentransfer innerhalb unseres Computers stattfindet, ist diese Umgebung stabil, und bleibt von der feindlichen Aussenwelt unberuehrt.

Die Uebermittlung der Daten von unserem Computer zu einem Geraet ausserhalb heisst **Ausgabe**,der umgekehrte Weg **Eingabe**, der gesamte Prozess wird als Ein-/Ausgabe oder E/A bezeichnet.

Sehen wir die interne Umgebung unseres Computers als angenehm an, so herrscht ausserhalb das totale Chaos. Die Uebertragung paralleler Daten in diesem Chaos ueber lange Strecken koennte verheerende Folgen haben. Der Empfang eines Schriftstueckes auf dem Drucker koennte sich als Geheimtelegramm vom KGB erweisen.

Um Daten ueber laengere Strecken sicher transportieren zu koennen, erfanden die Konstrukteure fuer den Austausch zwischen einer Datenendeinrichtung und einem Modem die serielle Datenuebertragung und mit ihr eine Standardschnittstelle.

Da der technische Fortschritt mit rasender Geschwindigkeit fortlief und mit ihm die Computer kamen, wurde diese Schnittstelle, aus Kompatiblitaetsgruenden, von den Computerherstellern missbraucht, ja sogar auf ihre Beduerfnisse umkonstrukturiert.

1969 wurde von der EIA (Electronic Industries Association), den Bell Laboratorien und den Herstellern von Kommunikationsanlagen gemeinsam der EIA RS-232-Standard formuliert und herausgegeben, der spaeter mit kleinen Aenderungen der RS-232-C-Standard wurde.

Dieser Standard wurde ebenfalls von der CCITT (Consultativ Commitee on International Telegraphy and Telephony) uebernommen und als V24-Empfehlung herausgegeben. Diese Empfehlung wurde 1972 ueberarbeitet mit einer Liste der Schnittstellenleitungen. Die elektrischen Werte wurden in der Empfehlung V28 angegeben.
V24 und V28 zusammen entsprechen der RS-232-C bzw. der DIN 66020.

Der Zweck dieser Schnittstelle ist unzweideutig beschrieben :

Schnittstelle zwischen Datenendeinrichtung und Datenuebertragungseinrichtung zum Austausch serieller binaerer Daten

Jedes Wort ist hier wichtig:

Gemeint ist die Schnittstelle zwischen einer Datenendeinrichtung **(Data Terminal Equipment oder DTE)** und einem Modem **(Data Communication Equipment oder DCE)** zur Uebertragung serieller Daten.

Die Beschreibung besteht aus vier Teilen :

Eigenschaften der elektrischen Signale

Spannungen zur Darstellung der logischen Null und der Eins werden hier definiert.

Mechanische Eigenschaften der Schnittstelle

Hier wird festgelegt, dass die Schnittstelle aus einem Stecker und einer Steckdose besteht, und dass die Steckdose auf der Seite des DCE sein muss.
Ebenso die Nummerierung der Stifte wird beschrieben. Das eigentliche Verbindungselement wurde nicht festgelegt, jedoch hat sich der D-formige DB-25-Stecker fuer diesen Standard durchgesetzt.

Der DB-25 ist in einem anderem Standard, dem ISO (International Standard Organziation) genormt. Dieser Stecker duerfte jedem Computerfreund bekannt sein, sodass er hier nicht extra beschrieben werden muss.

Funktionale Beschreibung der Schnittstellenleitungen

Dieser Teil definiert und bezeichnet die Funktionen der zu benutzenden elektrischen Signale. So liegt zum Beispiel SENDEDATEN immer auf Stift 2. 21 solcher Definitionen sind vorhanden, aber nur wenige von ihnen sind fuer Mikrocomputer von Bedeutung.

Standardschnittstellen fuer ausgewaehlte Kommunikations-systemkonfigurationen

Hier wurden Rezepte gekocht fuer uebliche Arten von Terminal/Modem-Verbindungen.

Nun, ist Ihnen bis hierhin alles klar ?!. DTE/DCE alles prima, aber wo steht geschrieben, wie ich meinen RS-232-C-kompatiblen Drucker anschliesse ?. Ich kann Ihnen gleich sagen, nirgendwo. Fahren wir also zum besserem Verstaendnis der Schnittstelle mit unserer Beschreibung fort.

Grundlagen

Mit zwei Leitungen zum Datentransport und einer Leitung fuer die Betriebserde haben wir eine RS-232-C-Schnittstelle in ihrer einfachsten Form. Die Betriebserde hat allerdings nichts mit Erde oder Null zu tun. Sie ist nur die absolute Referenzspannung fuer alle Schaltkreise der Schnittstelle, der Punkt, von dem alle Spannungen gemessen werden. Diese Betriebserde ist bei jeder Verbindung, und ist sie noch so komplex oder nicht, immer zwischen den Stiften 7 bei jedem Stecker herzustellen.

In Abbildung 1 (siehe Anhang) sehen wir ein typisches DTE-Geraet. Die Nummern im Kasten beziehen sich immer auf die Nummern der Stifte eines Steckers.

Betrachten wir uns diese Verbindung zwischen einer DTE und dem urspruenglichen Geraet DCE, damit wir den Begriff SENDEDATEN besser definieren koennen. Wie wir sehen kommen an Stift 2 DTE die SENDEDATEN heraus und werden an Stift 2 DCE empfangen. Also koennte es sich hier doch auch um Empfangsdaten handeln. Tatsaechlich handelt es sich hier um SENDEDATEN, und ob die gesendet oder empfangen werden, haengt ganz davon ab, von welchem Geraet aus Sie die Sache betrachten.

So werden die Daten an Stift 2 des DTE gesendet, waehrend die selben Daten an Stift 2 DCE empfangene Daten werden. (Siehe hierzu Abbildung 2 im Anhang).

Nun kann aber die DCE auch Daten zuruecksenden, die DTE Daten empfangen. Wir koennen hier gleich den Unterschied zwischen DTE und DCE festhalten :

DTE senden auf Stift 2 und empfangen auf Stift 3

DCE senden auf Stift 3 und empfangen auf Stift 2

Sehen Sie hierzu Abbildung 3. Wir haben hier zum besseren Verstaendniss die interne Bezeichnung der Daten in Kleinschrift angegeben. Sie sind nur fuer uns gueltig und gehoeren nicht zur Schnittstellenbeschreibung.

Sie werden sich fragen, wenn dies alles so einfach ist, warum dann 21 Leitungen. Nun ja zu einer ordnungsgemaessen Kommunikation gehoert eine ordnungsgemaesse Verwaltung. Ja, ja, schon wieder Verwaltung. Aber wo kaemen wir hin, wenn bei dieser Verbindung alle durcheinander reden wuerden, wir haetten ein neues Chaos.

Also musste man sich eine Verstaendigung einfallen lassen, die dem Partner sagt wann er reden soll und wann er den Mund zu halten hat. Und hier faengt wie im richtigem Leben der eigentliche Aerger an.

Stellen Sie sich vor, sie schuetteln Ihrem Partner die Hand um ihm mitzuteilen, dass er sprechen darf. Auf englisch nennt man dieses **Handshaking.**

Genau solch ein Verfahren, bzw. noch etwas komplexer, hat man bei der Schnittstelle eingefuehrt, auch hier sprechen wir von **Handshaking**, und meinen damit die geordnete Kommunikation waehrend der Verbindung von RS-232-C-Schnittstellen.

Gehen wir auf diese Problematik genauer ein.

Wir unterscheiden bei der Schnittstelle zwischen 2 Handshaking-Verfahren, dem Software-Handshaking und dem Hardware-Handshaking.

Beim Software-Handshaking handelt es sich um Steuerzeichen im Datenstrom.
Die Steuerzeichen sind **ETX** fuer End of Text und **ACK** fuer Acknowledge (Bestaetigung).

Hardware-Handshaking wird durch Steuersignale auf eigenen Verbindungsleitungen realisiert. Die Bezeichnung der Stifte fuer die Steuersignale sind bei DTE und DCE gleich.

Betrachten wir diesen Fall zwischen der DTE und der DCE in Abbildung 4, und nehmen wir an dass es sich hier um ein Terminal und ein Modem handelt.

In diesem Fall ist die Spannung an Stift 20 null Volt, wenn die Stromversorgung des Terminals ausgeschaltet ist. Diese null Volt am Stift 20 des Modems sagen diesem, dass es nicht betriebsbereit sein darf. Wird das Terminal nun eingeschaltet aktiviert es an Stift 20 eine Spannung > 3 Volt. Dieses sagt dem Modem, dass es betriebsbereit sein darf.

Halten wir fest: Eine positive Spannung an Stift 20 der DTE sagt der DCE, dass sie ruhig sein soll. Eine Spannung > 3 Volt an Stift 20 der DTE sagt der DCE,dass sie reden darf.

Beachten Sie bitte, dass Stift 20 der DTE ein Ausgabestift und Stift 20 der DCE ein Eingabestift ist.

Verfolgen wir die Sache weiter. Ebenso koennte das Terminal wissen wollen, ob das Modem eingeschaltet ist, bzw. das Modem dem Terminal mitteilen, dass es betriebsbereit ist. Dies geschieht ueber Sift 6. Siehe hierzu Abbildung 5.

An Stift 6 ist der Austausch von Signalen identisch zu denen zwischen den Stiften 20 : Das Signal eines Ausgabestiftes wird von der Eingabe des anderen Geraetes erkannt. Der einzige Unterschied ist, dass die Ausgabe (!) an der DCE erzeugt wird, waehrend die Eingabe (?) auf der DTE erkannt wird.

Zusaetzlich zu seinem offiziellen Namen hat jedes Signal eine inoffizielle Abkuerzung:

TRANSMITTED DATA (SENDEDATEN)	=	**TxD**
RECEIVED DATA (EMPFANGSDATEN)	=	**RxD**
DATA TERMINAL READY (DE-Betriebsbereit)	=	**DTR**
DATA SET READY (DUE-Betriebsbereit)	=	**DSR**

Da an der DTE und DCE die Sifte exakt die gleichen sind, koennen wir die Bezeichnungen auch zwischen die Geraete schreiben. Siehe hierzu Abbildung 6.

Bevor wir fortfahren lassen Sie uns die wichtigsten Punkte nachmals festhalten :

1. Daten koennen Byte fuer Byte uebertragen werden, diese Uebertragungsart nennt man parallele Uebertragung.

2. Daten koennen Bit fuer Bit uebertragen werden, diese Uebertragungsart nennt man serielle Uebertragung.

3. Um einen sicheren seriellen Informationsaustausch mit der Aussenwelt zu ermoeglichen wurde die RS-232-C-Schnittstelle erfunden.

4. Handshaking ist der Prozess, in dem ein Geraet den Zustand des anderen ueberwacht und entsprechend handelt.

Software-Handshaking geschieht durch Steuerzeichen im Datenstrom.

Hardware-Handshaking geschieht mit Hilfe von Steuersignalen auf eigenen Verbindungsleitungen.

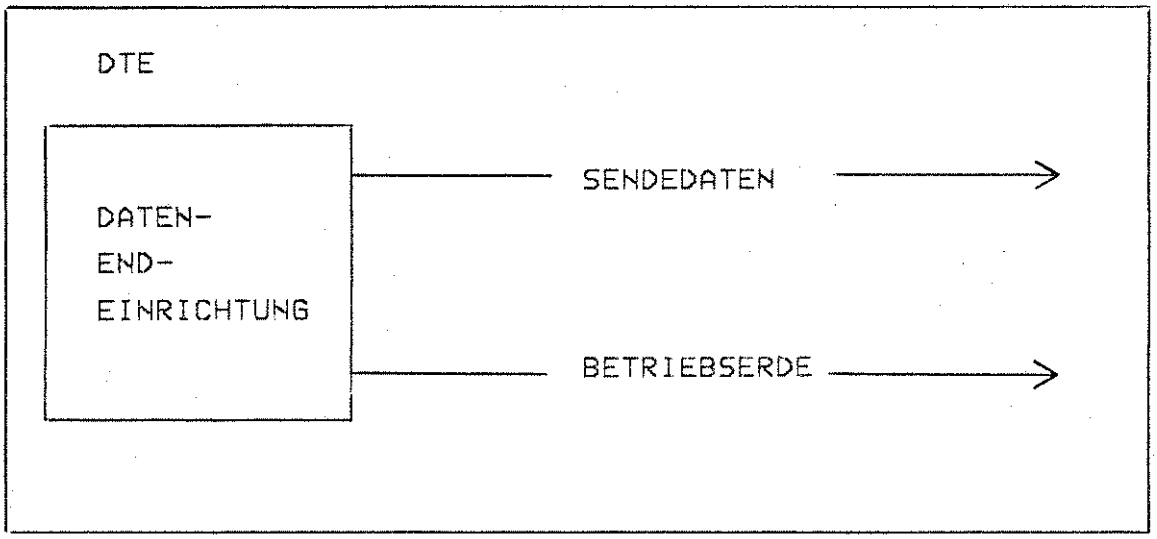


ABBILDUNG 1

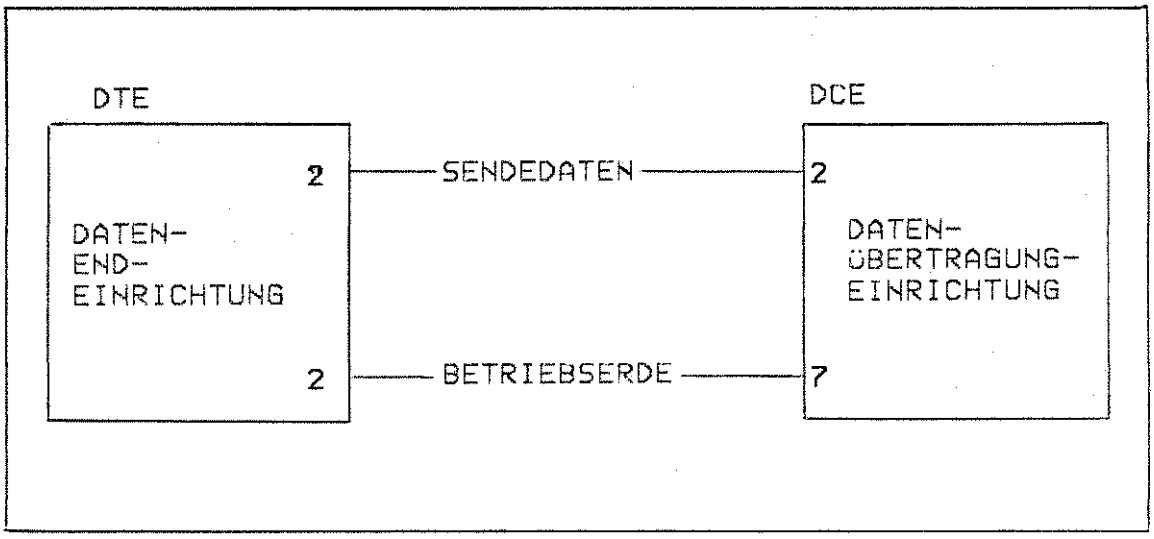


ABBILDUNG 2

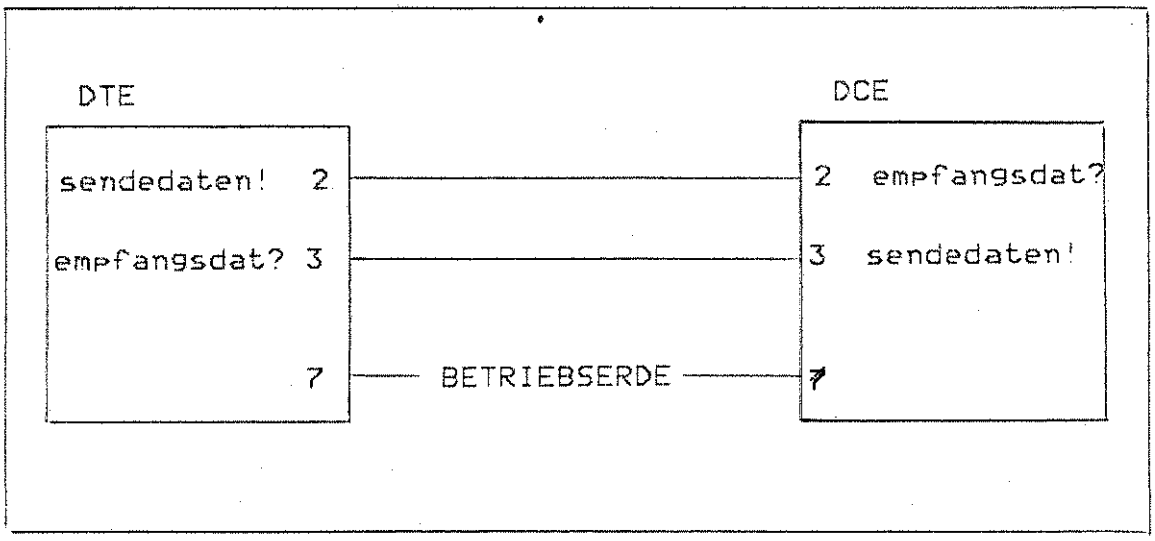


ABBILDUNG 3

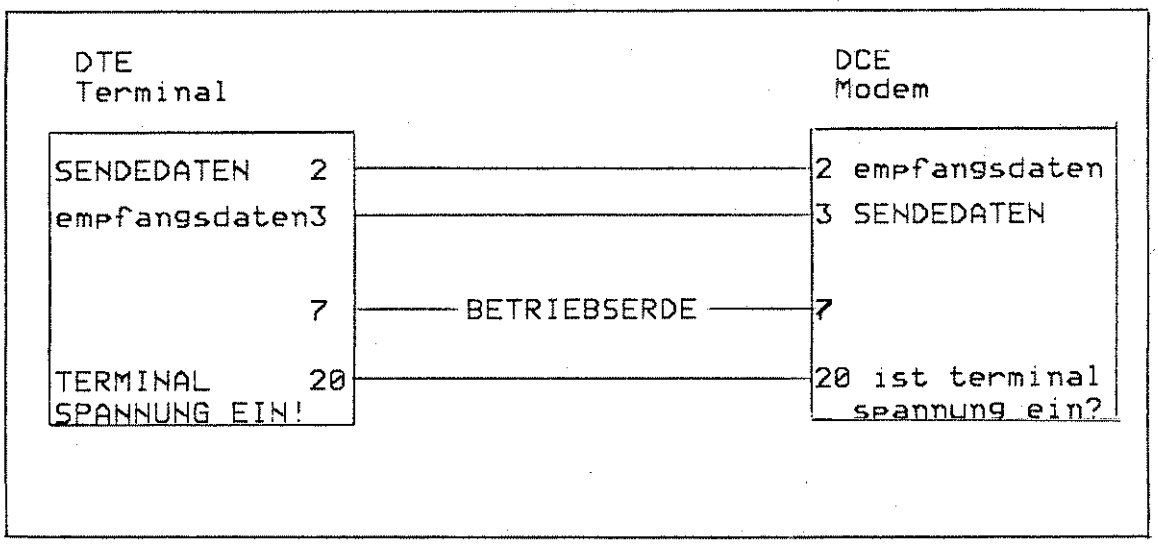


ABBILDUNG 4

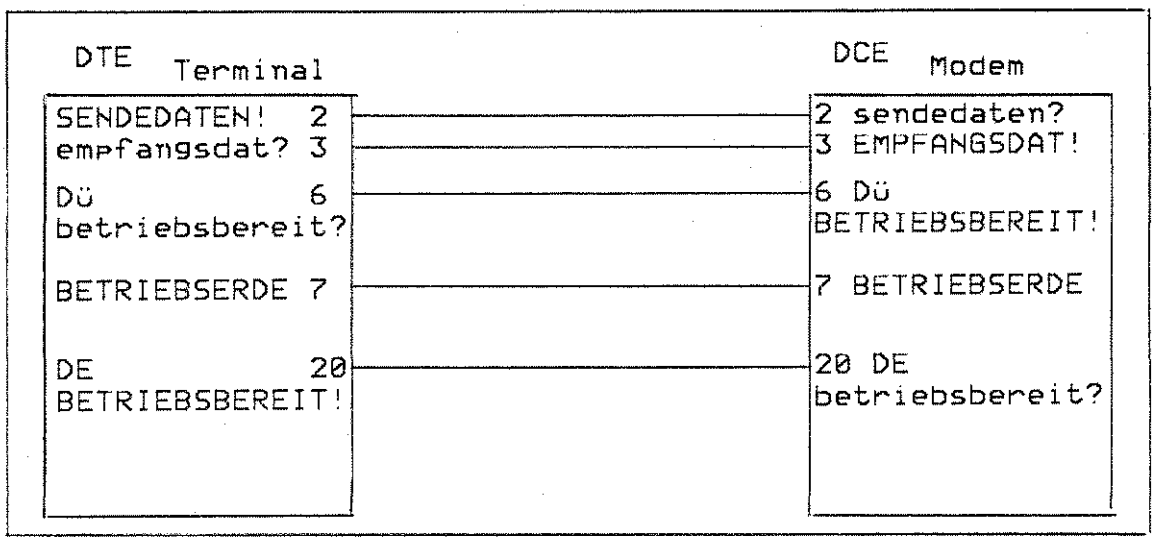


ABBILDUNG 5

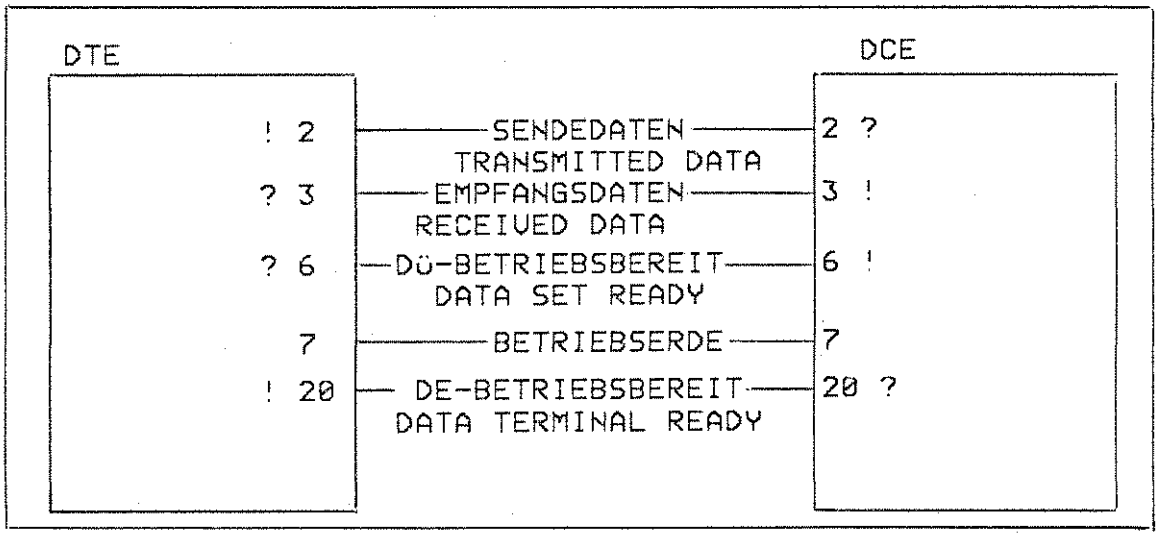


ABBILDUNG 6

DER ARITHMETIK PROZESSOR 9511

von Fabien Fojud

Wie aus dem Handbuch ersichtlich, wissen wir, daß der DAI mit einem 9511 ausgestattet werden kann. Aber wie sieht es mit seinem Gebrauch, seinen Eigenheiten usw. aus ... ?

Der AMD 9511 ist ein 24 poliges IC. Er wird im Innern des DAIs, auf den freien Sockel links neben den ROMs, gesteckt. Seine "Richtungsnase" zeigt dabei zum X-Bus, so wie es bei allen ROMs der Fall ist.

Wenn Sie auf PRINT PEEK(#FB00) die Anwort "DIVISION BY ZERO" bekommen, dann haben Sie einen 9511 im DAI. Aber das Lesen in #FB00 ist verboten (durch Hardware) und auch recht ungesund, so daß jetzt nur noch ein RESET hilft.

Ein anderer Punkt ist beachtenswert: Einige Programme, z.T. in Maschinensprache, können unter Umständen mit dem Mathepro-zessor nicht mehr richtig laufen (z.B. PADDLE, Matheunterpro-gramme usw.)
Muß man nun alle diese Programme umändern, oder den Mathepro-zessor heraus nehmen? Glücklicherweise nicht, es genügt in das Flag (#D4) die folgenden Werte zu schreiben:

```
POKE #D4,0     => 9511 "ausschalten"
POKE #D4,#7B   => Prozessor wieder aktivieren
```

In dieses Flag dürfen außer diesen beiden keine anderen Werte geschrieben werden, sonst ist man gezwungen zur Reset-Taste zu greifen ...

Nun zur Demonstration der Leistungsfähigkeit noch ein kleiner Geschwindigkeitstest:

```
    IMP FPT
10  FOR A=0 TO 10000
20  X= (siehe unten)
30  NEXT A
```

Funktionen:	Inhalt von Zeile 20	Arbeitszeit Mit 9511	Ohne 9511
ABS	X=ABS(-253.7)	33sec 16	35sec 12
ACOS	X=ACOS(0)	54sec 46	3min 18sec 11
ALOG	X=ALOG(0.3020)	48sec 26	8min 19sec 70
ASIN	X=ASIN(1)	27sec 53	31sec 80
ATN	X=ATN(A)	52sec 52	6min 01sec 58
COS	X=COS(A)	47sec 78	7min 15sec 58
EXP	X=EXP(-1)	51sec 53	7min 32sec 82
LOG	X=LOG(A+1)	55sec 58	6min 40sec 32
LOGT	X=LOGT(A+1)	56sec 42	7min 10sec 50
SIN	X=SIN(A)	45sec 90	7min 07sec 64
SQR	X=SQR(A)	28sec 06	2min 49sec 59
TAN	X=TAN(A)	51sec 71	14min 56sec 52
^	X=A^3	1min 15sec 40	13min 24sec 80
*	X=A*A	31sec 38	49sec 51
/	X=1/(A+1)	49sec 75	1min 36sec 73
ohne Zeile 20	----------	7sec 04	6sec 99

In einer alten CHIP (Ausgabe 11 und 12 1980) wurde der Mathe-matikprozessor AMD 9511 besprochen. Hierbei wurden noch einige interessante Entdeckungen gemacht:
Es gibt bei den Arithmetik-Prozessoren zwei baugleiche Typen, den 8231 von INTEL und den 9511 von AMD (Advanced Micro Devices).
Der 8231 könnte also prinzipiell anstatt dem AMD 9511 in den DAI eingesetzt werden. Wo man diesen Baustein erhalten kann und ob, und wenn wieviel dieser Baustein billiger als der 9511 ist war bislang nicht zu erfahren. Sollten jedoch konkrete Zahlen vorliegen, so werden wir Euch natürlich informieren!

********** Übersetzung **********
********* aus *DAICLIC* Nr. 1 *********
**** Zeitschrift des International DAI Club, Belgien ****

RANDOM

LASST UNS ZUFALLSZAHLEN ERZEUGEN < 1 >

von Raymond VANLATHEM (Club CAROLODAI)

Einführung

Wir haben alle schon einmal einen Zufallszahlengenerator beansprucht, mit Hilfe der Operation RND. Diese Zahlen werden laufend verwendet bei der Lösung gewisser Aufgaben, um Simulationen durchzuführen (z.B. in der Industrie), im Verlauf der meisten Spiele, im Unterricht bei der Auswahl von Übungsaufgaben, und dergleichen mehr. In jedem Fall wünschen wir, daß die Ereignisse, die sich abspielen, für den Benutzer unerwartet sind, als ob sie nur vom Zufall abhingen. Per Definition sind Zufallszahlen eben solche, die nur vom Glück abhängen, und die daher nicht vorhersehbar sind.

Und was macht Ihr DAI?

Wie die meisten Rechner, kann er sogenannte Zufallszahlen erzeugen, denn die Folgen der erzeugten Zahlen sind in Wirklichkeit nur scheinzufällig, auch wenn sie zufällig aussehen. In der Tat, diese Zahlen stammen aus einer mathematischen Formel, die ja von ihrer Natur aus deterministisch ist, und somit fern davon, zufällig zu sein.

Um eine neue Folge zu starten, muß zuerst eine Zahl eingebracht werden, genannt die Keimzahl oder Quellzahl (SEED auf englisch), entweder durch den Benutzer oder aber vom Rechner selber, was den Anschein hergibt, die Zahlenfolgen seien vollkommen zufällig.

Normalerweise wird Ihr DAI beim Einschalten automatisch die Keimzahl der Folge erstellen und anschließend die erste Zufallszahl erzeugen. Bei jeder Ausführung der Operation RND erzeugt der DAI die nächste Zufallszahl der Folge.

Probieren wir einmal folgendes Programm, gleich nach dem Einschalten:

```
100   FOR I%=1 TO 5 : PRINT RND(1), : NEXT
```

und lassen wir es zweimal hintereinander ausführen. Wir erhalten auf dem Bildschirm:

```
RUN
0.345451     0.917845     0.703101     0.633126     0.13235
RUN
0.183837     0.527697     0.651674     0.451856     0.531174
```

also zwei verschiedene Zufallszahlreihen, bestehend aus den ersten 10 Zahlen der Folge.

Wir bemerken, daß man nicht durch einen RESET diese zwei Reihen von neuem erhalten kann; die Ausführung des obigen Programms läßt die nächsten Zufallszahlen am Bildschirm anzeigen. Man muß notwendigerweise den DAI ausschalten und vor dem Wiedereinschalten mindestens 15 bis 20 Sekunden warten.

Der DAI wird die Folge initialisieren und bei der ersten Ausführung von RND(1) die Anfangszahl 0.345451 erzeugen. Wenn das nicht der Fall ist, dann hat Ihr DAI nicht die Keimzahl geliefert, die beim Kalteinschalten vorgesehen ist.

Wir haben gerade das Programm ausprobiert mit dem Argument X von RND gleich 1. Wenn wir X einen anderen positiven Wert geben, erhalten wir die gleichen Folgen, aber die Zahlen sind mit dem Wert des Arguments multipliziert. So sind bei RND(10) die ersten angezeigten Zahlen:

```
3.45451      9.17845      7.03101      6.33126      ...
```

Ob man A = RND(10) oder A = 10 * RND(1) schreibt, kommt auf's gleiche hinaus.

Und wenn X einen negativen Wert annimmt?

Was passiert dann? Das Programm wird ständig die Anfangszahl erzeugen. So erhalten wir mit X=-1 bei jedem RUN fünf mal die gleiche Zahl -0.571069, auch ohne den DAI auszuschalten: diese Folge heißt eine "wiederkehrende Folge" im Gegensatz zu "fortlaufende Folge", welches die Folge der Zahlen bezeichnet, die von einem positiven Wert des Arguments X erzeugt werden.

Was für X = -1 gilt, gilt ebenso für jeden negativen Wert des Arguments: die erzeugte Zahl ist negativ, obwohl eine andere als oben (je nach dem Wert von X), aber sie wird immer wiederholt.

Im Fall von X = -10 erhalten wir immer -3.21069. Man beachte, daß A = RND(10) nicht gleichbedeutend ist mit A = 10 * RND(-1).

In der Tat, 10 * (-0.571069) = -5.71069, also anders als vorhin. Im Gegensatz, RND(-5) liefert als erzeugte Zahl -1.60534 (mal 2 = -3.21068): wir sehen, daß hier A = RND(-10) das Gleiche liefert wie A = 2 * RND(-5), bis auf die Rundung der fünften Dezimalstelle. Wir werden später den Grund dafür sehen.

Beim ersten Blick erscheint es, daß es bei der Programmierung überhaupt keinen Nutzen haben kann, wiederkehrende Folgen mit negativen Argumenten zu bekommen. Nur, die so erhaltene Anfangszahl kann beim Austesten eines Programms dazu dienen, daß die bereitgestellten Daten immer die gleichen sind und in der gleichen Reihenfolge.

Ergänzen wir das ursprüngliche Programm durch die folgende Zeile:

```
10    Y = RND(-1)
```

und führen wir das vollständige Programm zweimal hintereinander aus. Der Bildschirm zeigt an:

```
RUN
0.389234     0.359846     0.488753     0.388517     0.171452
RUN
0.389234     0.359846     0.488753     0.388517     0.171452
```

Hier nicht nötig, den DAI auszuschalten, 15 bis 20 Sekunden vor dem Wiedereinschalten zu warten, und das Programm neu zu laden.

Lassen Sie uns die Ausführung unseres Programms etwas verändern. Nach einem ersten RUN, machen wir RUN 100. Wir erhalten am Bildschirm:

```
RUN
0.389234     0.359846     0.488753     0.388517     0.171452
RUN 100
0.969859     0.698164     0.996478     0.145397     0.145897
```

Das zweite RUN erzeugt die nächsten 5 Zufallszahlen aus der fortlaufenden Folge. Bemerken wir allerdings, daß die Folge, die erzeugt wird nach der Ausführung eines RND mit einem negativen Argument (im gegenwärtigen Fall mit X = -1), eine andere ist, als die, die normalerweise durch ein positives Argument erzeugt wird.

Was passiert, wenn wir in Zeile 10 die -1 durch ein anderes negatives Argument ersetzen? Die fortlaufenden Folgen, die erhalten werden, sind verschieden, wenn X ungerade ist.

Z.B., mit X = -2, -4, -8, -16, ... erhalten wir die gleiche Folge, die mit X = -1 erzeugt wird. Und mit X = -3, -6, -12, -24, ... erhalten wir eine andere Folge. Gleichfalls mit X = -5, -10, -20, -40, ... wieder eine andere Folge... Es gibt allerdings eine Beziehung zwischen den Zufallszahlen an gleicher Stelle in zwei verschiedenen Folgen. So unterscheiden sich alle Zahlen der von X = -3 erzeugten Folge um 0.5 von denen aus der Folge, die man mit X = -1 erhält, aber sie unterscheiden sich um 0.25 oder um 0.75 von den Zahlen aus der Folge erzeugt durch X = -5.

```
Wenn X = -1, haben wir:
0.389234     0.359846     0.488753     0.388517     0.171452
Wenn X = -3, haben wir:
0.889234     0.859846     0.988753     0.888517     0.671452
Wenn X = -5, haben wir:
0.639234     0.109846     0.738753     0.138517     0.421452
```

Das kann für gewisse Anwendungen von Interesse sein. Z.B., bei einem Spiel von KOPF [gerundet = 1] oder ZAHL [gerundet = 0], mit X = -1 erhalten wir bei Spielbeginn 5 mal ZAHL, mit X = -3 erscheint 5 mal KOPF, und mit X = -5 haben wir nacheinander KOPF - ZAHL - KOPF - ZAHL - ZAHL.

Bemerken wir noch: um eine fortlaufende Folge von negativen Zufallszahlen zu erhalten, reicht es, den RND Befehl mit positivem Argument durch eine negative Zahl zu multiplizieren; z.B., A = -1 * RND(1), was keinesfalls das Gleiche ist, wie A = RND(-1)!

Kniffe ... und noch mehr Kniffe

Um beim Einschalten eine automatische Rückkehr zu immer der gleichen Anfangszahl zu verhindern, können Sie eine Startroutine verwenden, die erlaubt, die Folge mit der N. Zahl zu beginnen, genannt die Ausgangszahl. Die gleiche Routine erlaubt es auch, zwischen zwei Ausführungen des Programms N Zahlen aus der Folge zu überspringen.

```
20 INPUT "Geben Sie eine Zahl zwischen 1 und 100 ein ";A%
30 FOR J% = 1 TO A% : Z = RND(1) : NEXT
```

Diese Routine liefert in Verbindung mit Zeile 10 bei A% = 2 die Zahl 0.488753 als Ausgangszahl, und ohne Zeile 10 die Zahl 0.703101. Die Startzahl A% kann auf zufällige Weise gewählt werden, z.B.: man öffne ein Telefonverzeichnis auf eine beliebige Seite, zeige blind mit dem Finger auf eine Telefonnummer, und nehme deren zwei letzte Ziffern; oder man schaue auf die Armbanduhr und man verzeichne die Sekunden als Wert von A%.

Dieses Verfahren hat den Nachteil ziemlich langsam zu sein, besonders wenn die eingegebene Zahl (A%) hoch ist.

Wir geben eine wirksamere (da sehr schnelle und sehr vielfältige) Methode, bei der wir 6 Ziffern aus der Telefonnummer verwenden können (statt 2), oder die vollständige Zeitangabe (Stunde, Minuten, Sekunden), die die Uhr anzeigt.

Die Wahl liegt bei Ihnen!

Hier ist also die zweite Routine:

```
20 INPUT "Geben Sie zwei Zahlen ein von Maximal 3 Stellen
          (zwischen 0 und 999), getrennt durch ein Komma, wie
          z.B. 345,78 ";L1%,L2%
30 L% = L1% * 1000 + L2% : A% = L1% * 0.127 + 128
40 B% = L2% * 0.251 : C% = L% MOD 997 / 4
50 POKE #12E,A% : POKE #12F,B% : POKE #130,C%
```

Und schließlich, wenn Sie nur die Zeile 50 verwenden, und dort A%, B% und C% durch den Dezimalwert 255 ersetzen, initialisieren Sie wieder die Zufallsfolge bei jedem RUN, mit ähnlicher Wirkung wie das Kalteinschalten des Rechners: dies ist ein anderes Mittel, um Programme auszutesten.

Wir werden später sehen, worauf wir wirken wenn wir diese Routine verwenden. Aber wir deuten schon gleich darauf hin, daß A% zwischen 128 und 255 liegen muß, während B% und C% kleiner sein müssen als 256.

übersetzt von Gordon Wassermann

(wird fortgesetzt)

********** Übersetzung **********
********** aus *DAICLIC* Nr. 3 **********
**** Zeitschrift des International DAI Club, Belgien ****

LASST UNS ZUFALLSZAHLEN ERZEUGEN 2

von Raymond VANLATHEM (Club CAROLODAI / DAIclic)

Nachtrag Nr. 1

1.- Ein DAI Benutzer hat mich darauf hingewiesen, daß er beim Einschalten seines DAI oder bei der ersten Verwendung der Funktion RND(1) nicht die Folge von Zufallszahlen erhielt, die im ersten Artikel über dieses Thema zur Illustration diente. Wenn dies auch bei Ihnen der Fall ist, beschuldigen Sie nicht Ihren DAI oder nehmen es mir übel.

Die Erklärung ist ganz einfach. Unter gewissen Umständen wird die Initialisierung der Speicherzellen nicht richtig ausgeführt, insbesondere nach einem STACK OVERFLOW. Bevor Sie die RND Funktion verwenden, prüfen Sie den Inhalt der Speicherzellen, wo die Quellzahl des Zufallsgenerators steht, durch einen DISPLAY im UTility: >D12E 130 (der Inhalt von #12D ist nicht wichtig). Diese 3 Speicherstellen sollten FF FF FF enthalten bei der ersten Verwendung der RND-Funktion nach dem Einschalten im Kaltzustand, oder nach einer Wartezeit von mindestens 15 bis 20 Sekunden (ja sogar noch mehr), wenn man seinen DAI ausgeschaltet hat.

Ich habe selber feststellen können, daß in manchen Fällen mindestens einer der FF Bytes durch 7F oder F8 (sogar FC, FE, ...) ersetzt wurde: daher die Erzeugung einer anderen Folge als die im Artikel Angegebene.

Um die gleichen Zahlen zu erhalten, wie die im Artikel Genannten, reicht es, den Inhalt der betroffenen Speicherzellen zu ändern mit dem SUBSTITUTE Befehl im UTility: nach >S12E geben Sie 3 mal FF ein.

2.- Ein DAICLIC Leser hat zu mir geäußert, ich hätte noch nicht von dem berühmten Befehl RND(0) gesprochen, der den mit dem Zustand des Rauschgenerators (NOISE) verbundenen Hardwaregenerator des DAI in Gang bringt. Dieser soll angeblich vollkommen zufällige Zahlen hervorbringen. Aber Versuche ausgeführt auf verschiedenen DAIs erlauben mir nicht ein gültiges Urteil darüber zu schließen, in Anbetracht der schwer kontrollierbaren Ergebnisse (in der Tat, diese Art von Generator produziert nie 2 gleiche Folgen, woher die Schwierigkeit stammt, diesen Generator Vertrauenswürdigkeitstests zu unterwerfen) und in Anbetracht des mangelhaften Zustands mancher NOISE-Generatoren (in den meisten DAIs reicht es, eine kleinere Änderung am NOISE Generator anzubringen, um eine mehr oder minder gültige Folge von Zufallszahlen zu erhalten).

Auf jeden Fall, man hat feststellen können, daß eine von der Hardware des DAI erzeugte Folge weniger vertrauenswürdig ist (größere Streuung) als die Folgen, die von der ROM Software hervorgebracht werden. Wir werden später zu diesem Punkt zurückkehren! Es wäre angebracht zu bemerken, daß DAI Programmierer sehr selten den Befehl RND(0) in ihren Programmen verwenden. Zu Unrecht oder zu Recht?*

---

* (Anm. des Übersetzers) Siehe hierzu meinen Anhang am Ende des Artikels.

PRINZIP des SOFTWAREGENERATORS des DAI

Im ersten Artikel haben wir die Untersuchung der Zufallszahlengeneratoren begonnen und wir haben gesehen, wie der DAI sich allgemein verhält. Laßt uns etwas näher sehen, was in seinem Inneren passiert.

Es ist gut sich daran zu erinnern, daß jede Zahl im DAI gespeichert wird als 4 Bytes von 8 Bits, und daß eine Hexadezimalzahl nicht nur einer ganzen Zahl entsprechen kann, sondern auch einer Familie von reellen Zahlen: es hängt nur von der gewünschten Interpretierung ab. So entspricht die Hexadezimalzahl 02 80 00 00 der ganzen Zahl 41 943 040 und einer Familie von reellen Zahlen benachbart zu 2.0 (von 1.99999988... bis 2.00000079...). In der Tat, da der DAI nur höchstens 6 signifikante Ziffern in Gleitkomma anzeigt, wird die ganze Familie von reellen Zahlen als 2.0 angezeigt. Laßt uns auch bemerken, daß die Hexadezimalzahl 01 FF FF FF, die auch einer Familie von reellen Zahlen benachbart zu 2.0 entspricht, eine sehr verschiedene dezimale ganze Zahl ergibt: 33 422 847. Das stammt von dem unterschiedlichen Binärdarstellungssystem für ganze Zahlen und für reelle Zahlen.

Andererseits ist es gut zu wissen, daß der Wert des positiven Arguments X der RND-Instruktion keine Rolle spielt bei der Bestimmung der Zufallszahl. Alles wird zurückgeführt auf eine RND(1) Routine, die eine Zufallszahl erzeugt, die zum Schluss mit dem Argument X multipliziert wird, bevor sie am Bildschirm angezeigt wird. Alles passiert, als hätten wir den Befehl RND(X) in der Form: X * RND(1).

Die Grundformel, die im DAI verwendet wird, ist:

$$X_{n+1} = AX_n + B$$

wobei A = 59 (in Hexadezimal: 3B) und B = 125 000 001 (in Hexadezimal: 07 73 59 41), mit X_n die vorherige Zufallszahl. Wir bemerken, daß A und B ganze Zahlen sind; X ist eine reelle Zahl (in Gleitkommadarstellung).

Der Wert X_{n+1} wird aufbereitet, bevor er selber wiederum in der Formel verwendet wird, um die nächste Zahl zu erzeugen. Das Aufbereitungsverfahren ist wie folgt: der Hexadezimalwert von X_{n+1} wird abgeschnitten, nur die 3 niederwertigen Bytes werden beibehalten. So wird die Hexadezimalzahl 70 CC 0F 94 zu CC 0F 94. Der DAI verwendet eine AND- (UND-)Maske, nämlich #00 FF FF FF, um dieses Ergebnis zu erhalten: das Vorhandensein eines 1-Bits in einer solchen Maske erwirkt das Kopieren des entsprechenden Bits der Ausgangszahl, während ein 0-Bit das entsprechende Bit der resultierenden Zahl zu Null setzt.

```
70 CC 0F 94 = 0111 0000   1100 1100   0000 1111   1001 0100
  UND-Maske = 0000 0000   1111 1111   1111 1111   1111 1111
00 CC 0F 94 = 0000 0000   1100 1100   0000 1111   1001 0100
```

Das vierte Byte (70) wird dann durch #01 ersetzt, was uns die Zahl 01 CC 0F 94 liefert: dies ist der aufbereitete Wert

X_{n+1}.* Wir bemerken, daß die erste Hexadezimalziffer des 3. Bytes (desjenigen, welches auf die 01 folgt) immer gleich oder größer als 8 sein muß**; gilt das nicht, so addiert der DAI automatisch 8 zu dieser Ziffer. So wird die Hexadezimalzahl 71 7A F0 5D, abgeschnitten als 7A F0 5D, zu 01 FA F0 5D (der DAI addiert 8 zur 7 von 7A: das macht 8 + 7 = F, daher das FA). Der DAI benutzt eine OR- (ODER-)Maske, nämlich #01 80 00 00, um dieses Ergebnis zu erhalten: das Vorhandensein eines 1-Bits in einer solchen Maske reicht, um das entsprechende Bit der resultierenden Zahl zu eins zu setzen, während ein 0-Bit das Kopieren des entsprechenden Bits der Ausgangszahl erwirkt.

```
00 7A F0 5D = 0000 0000   0111 1010   1111 0000   0101 1101
 ODER-Maske = 0000 0001   1000 0000   0000 0000   0000 0000
01 FA F0 5D = 0000 0001   1111 1010   1111 0000   0101 1101
```

Man wird sagen, daß die Zahl X_{n+1}, die man nach der Aufbereitung erhält, als reelle Zahl interpretiert sich in dem Intervall 1.0-2.0 befindet, denn die äußersten Werte, die diese Zahl annehmen kann, sind 01 80 00 00 (reelle Zahl = 1.0) und 01 FF FF FF (reelle Zahl = 2.0 (aufgerundet [Anm. des Übersetzers])).

Die ganze Operation (Berechnung der Formel, plus Aufbereitung) wird 5 mal wiederholt bis sie die Zahl X_{n+5} liefert, die als Ausgangszahl für die folgende Runde dienen wird, und die zur ausgegebenen, d.h., dem Benutzer verfügbaren, Zufallszahl wird. Diese Zahl wird aber einer letzte Veränderung unterzogen, und zwar: die erzeugte Zahl wird zuerst in das Intervall 0.0 - 1.0 zurückgeführt und dann mit dem Argument X multipliziert.

---

* (Anm. des Übersetzers). Diese Aufbereitung dient dem Zwecke, aus der ursprünglichen ganzen Zahl X_{n+1} eine Gleitkommazahl zu machen. Gleitkommazahlen bestehen aus einem Byte Vorzeichen und Exponent (das höchste Bit dieses Bytes ist das Vorzeichen s, die niederen 7 Bits stellen den positiven oder negativen Exponenten e dar als siebenbit zweierkomplement signierte Zahl) und drei Bytes Mantisse M, die als ein Bruch zwischen 0 und 1 aufgefaßt wird; der Wert dieser Gleitkommazahl ist $(-1)^s \cdot 2^e \cdot M$. Bis auf die Zahl 0 (die als vier Bytes 00 dargestellt wird) werden alle Gleitkommazahlen *normalisiert*: die Mantisse wird so lange nach links verschoben (oder, was das gleiche ist, mit 2 multipliziert), bis ihr höchstes Bit 1 wird, und um diese jeweilige Verdoppelung der Mantisse zu kompensieren, wird der Exponent für jede Stelle Verschiebung (d.h. für jede Verdoppelung) um eins vermindert, so daß der gesamte Gleitkommawert erhalten bleibt. Durch die Normalisierung erhält jede Gleitkommazahl eine **eindeutige** Darstellung, und zwar diejenige, bei der alle Mantissenbits signifikant sind, die also die größte Genauigkeit hat! Die Mantisse einer normalisierten Zahl (außer 0) hat einen Wert zwischen 1/2 und 1. Das 01 Byte, mit dem die aufbereitete Zahl beginnt, verleit der Zahl den Exponenten 1, so daß der gesamte Gleitkommawert zwischen 1 und 2 liegen wird. Siehe auch weiter unten im Artikel.

** (Anm. des Übersetzers) Dies bedeutet nichts anderes, als daß das höchste *Bit* dieses Bytes 1 sein muß, und zwar, weil die Mantisse einer Gleitkommazahl normalisiert sein muß--siehe die vorherige Fußnote.

PRAKTISCHES AUSFÜHRUNGSBEISPIEL

Gegebenheiten

Ein Programm verwende RND(10) und die zuletzt ausgegebene Zufallszahl sei 0.359846. Wir erwarten, daß als nächste Zufallszahl 4.88753 (= 10*0.488753) herauskommen wird, wie im vorhergehenden Artikel angegeben (siehe IB 26,2).

REM: Wir hatten am Anfang RND(-1) benutzt, um diese Folge von Zufallszahlen zu erzeugen.

Tabellen

Folgende Tabellen geben die Werte wieder, die man zu den verschiedenen Stadien des Verfahrens zur Erzeugung der Zufallszahl erhält. Tabelle 1 gibt die Werte in hexadezimal an, während die entsprechenden Werte in der Gestalt von ganzen Dezimalzahlen in Tabelle 2 stehen, und in der Gestalt von reellen Zahlen in Tabelle 3. Die Werte von geringerem Interesse (weil für den Anwender ohne viel Sinn) sind in Klammern eingeschlossen.

TABELLE Nr. 1	Spalte A	Spalte B	Spalte C	Spalte D
Zeile 0	04A00000	01AE0F6D		
Zeile 1	631D8E1F	6A90E760	90E760	0190E760
Zeile 2	5C655320	63D8AC61	D8AC61	01D8AC61
Zeile 3	6CEFBA5B	7463139C	63139C	01E3139C
Zeile 4	6F5584F4	76C8DE35	C8DE35	01C8DE35
Zeile 5	694B3637	70BE8F78	BE8F78	01BE8F78
Zeile 6	01BE8F78	7FFA3DE0	039C66AC	

TABELLE Nr. 2	Spalte A	Spalte B	Spalte C	Spalte D
Zeile 0	(77594624)	28184429		
Zeile 1	1662881311	1787881312	(9496416)	26273632
Zeile 2	1550144288	1675144289	(14199905)	30977121
Zeile 3	1827650139	1952650140	(6493084)	31658908
Zeile 4	1867875572	1992875573	(13164085)	29941301
Zeile 5	1766536759	1891536760	(12488568)	29265784
Zeile 6	29265784	(2147106272)	(60581548)	

TABELLE Nr. 3	Spalte A	Spalte B	Spalte C	Spalte D
Zeile 0	10.0	1.35985		
Zeile 1	(8.60168E-10)	(1.34952 E-7)	(0.566031)	1.13206
Zeile 2	(1.15193E-11)	(1.57651 E-9)	(0.84638)	1.69276
Zeile 3	(8.93056E-7)	(1.88974 E-4)	(0.387018)	1.77404
Zeile 4	(2.54868E-6)	(7.66251 E-4)	(0.784641)	1.56928
Zeile 5	(3.50232E-8)	(1.13583 E-5)	(0.744377)	1.48875
Zeile 6	1.48875	0.488753	4.88753	

(Anm. des Übersetzers: die Werte in Tabelle 3 sind die Gleitkommawerte, die man erhält, wenn man die entsprechende Hexadezimalzahl in eine Gleitkommavariable speichert und diese von der ROM Arithmetiksoftware als Gleitkommazahl ausdrucken läßt. Aber die ROM Software wandelt *unnormalisierte* Gleitkommawerte manchmal nicht richtig um, d.h. manche der eingeklammerten Gleitkommawerte sind nicht der wahre Gleitkommawert der entsprechenden Hexadezimalzahl. Wenn man aber den AMD Mathematikprozessor verwendet, erhält man die richtigen Werte. Da falsche Werte aber nur bei unnormalisierten Zahlen, also nicht in den wichtigen Fällen auftreten, und nicht jeder die AMD Werte selber nachvollziehen kann, haben wir die zum Teil falschen Werte aus dem französischen Original beibehalten.)

Kommentar

In Zeile 0, Spalte A: Wert des Argumenten X.

In Zeile 0, Spalte B: Ausgangszahl der Runde = Wert von X_n im Intervall 1.0-2.0. Dies ist der im Laufe der vorherigen Runde erhaltene Wert und befindet sich in den Speicherzellen #12D-#130. Wenn der Inhalt von #12D verschieden von #01 ist, ersetzt ihn der DAI automatisch durch #01; der Inhalt von #12E muß zwischen #80 und #FF liegen (Anm. des Übersetzers: im Original steht: "eine Frage des Intervalls" (in der wohl der Wert liegen muß). Dies ist falsch: der wahre Grund für diese Bedingung liegt in der Normalisierung der Gleitkommamantisse--siehe dazu die Fußnoten oben).
#01AE0F6D hat den Wert 1.35985 (Rundung von 1.359846)
oder 28 184 429 (ganze Zahl).

In Zeile 1, Spalte A: Produkt von X_n mit A.
#01 AE 0F 6D * #3B = #63 1D 8E 1F,
oder 28 184 429 * 59 = 1 662 881 311.

In Zeile 1, Spalte B: Summe von AX_n plus B.
#63 1D 8E 1F + #07 73 59 41 = #6A 90 E7 60,
oder 1 662 881 311 + 125 000 001 = 1 787 881 312.

In Zeile 1, Spalte C: Wert des abgeschnittenen X_{n+1}.
#6A 90 E7 60 wird zu #90 E7 60.

In Zeile 1, Spalte D: Wert von X_{n+1} im Intervall 1.0-2.0.
#90 E7 60 wird zu #01 90 E7 60, oder in Gleitkomma zu 1.13206.

Zeile 2, Spalten A-D: 2. Schleife mit Ergebnis X_{n+2}. Das Verfahren ist ähnlich zu dem der ersten Schleife.

Zeile 3, Spalten A-D: 3. Schleife mit Ergebnis X_{n+3}. Man beachte, daß das Byte #63 (in 3C) zu #E3 wird (in 3D); in der Tat, da 6 kleiner ist als 8, haben wir 6 + 8 = #E (siehe oben).

Zeile 4, Spalten A-D: 4. Schleife mit Ergebnis X_{n+4}.

Zeile 5, Spalten A-D: 5. Schleife mit Ergebnis X_{n+5}.

Zeile 6, Spalte A: Wert identisch mit dem aus 5D. Der in 5D gefundene Wert, nämlich #01 BE 8F 78 (als reelle Zahl 1.48875) wird in der nächsten Runde als Anfangszahl dienen. Sie wird in den Speicherzellen #12D bis #130 aufbewahrt.

Zeile 6, Spalte B: Wert in das Intervall 0.0 - 1.0 zurückgeführt. Das gibt uns #7F FA 3D E0, oder die reelle Zahl 0.488753.

Zeile 6, Spalte C: gesuchte Zufallszahl. Da das Argument X gleich 10 ist, wird die verwendbare Zufallszahl 10 * 0.488753 = 4.88753 sein, was wir erwarteten!

Anhang des Übersetzers

Zu RND(0):

Außer Unkenntnis von seiner Existenz, wird wohl ein Grund für die Unbeliebtheit von RND(0) darin liegen, daß dieser Befehl wesentlich langsamer ist als der Softwarezufallsgenerator; außerdem stimmt es tatsächlich, daß die Hardwarezufallszahlen weniger zufällig sind (nach statistischen Tests) als die Softwarezahlen. Ein guter Trick, um die guten Eigenschaften des Softwaregenerators ausnützen zu können und trotzdem jedes Mal **verschiedene** Zufallsfolgen zu erhalten, ist, den Hardwaregenerator nur einmal zu Anfang zu verwenden, um den Softwaregenerator zu initialisieren, und zwar mit dem Befehl

```
BELIEBIGEVARIABLE = RND(-RND(0)).
```

Es gibt übrigens einen ganz einfachen optischen Test für die Güte des Zufallszahlengenerators, genauer gesagt für das Vorhandensein einer (unerwünschten!) linearen Korrelation zwischen den erzeugten Zufallszahlen: man lasse einfach folgendes kurzes Graphikprogramm laufen:

```
5    MODE 6
10   DOT RND(XMAX),RND(YMAX) 21: GOTO 10
```

Besteht eine lineare Beziehung zwischen sukzessiven Zufallszahlen, so werden die Punkte nicht gleichmäßig verteilt auf dem Bildschirm erscheinen, sondern ein deutliches Streifenmuster bilden! Der Softwaregenerator des DAI besteht diesen Test glänzend, ganz im Gegensatz, interessanterweise, zum IBM PC!

übersetzt von Gordon Wassermann

DRUCKKOPFREINIGUNG

Besitzen Sie einen Epson FX-80 oder anderen Epson Drucker mit auswechselbarem Druckkopf? Und haben Sie schon, wie ich, z.B. erlebt, daß eine Nadel nicht mehr oder nur noch sehr schwach druckt? Das scheint gar nicht so selten vorzukommen.

Wenn ja, dann brauchen Sie wahrscheinlich **nicht** gleich einen neuen Druckkopf zu kaufen. Ich konnte jedenfalls durch eine Reinigung meinen Drucker wieder in Gang bringen, und wie Sie an dem Schriftbild dieses Artikels sehen, druckt er wieder wie neu. Aber es gibt einen kleinen Kniff dabei: Sie müssen nach der Reinigung die Nadeln *einfetten*. Tun Sie das nicht, dann wird das Schriftbild eher schlechter, denn die Reinigung entfernt nicht nur alte Tintenreste, die die Nadeln in ihrer Fahrt behindern können, sondern auch den ursprünglich vorhandenen Ölfilm, ohne den die Nadeln noch stärker behindert werden. Das kann sogar so weit gehen, daß die Nadeln sich im Farbband verfangen und der Druckkopf unter sehr unangenehmen kreischenden Geräuschen zum Stillstand gebracht wird. Dies ist übrigens auch der Grund, warum im Bedienungshandbuch vor einer Reinigung mit Lösungsmitteln gewarnt wird. Was das Bedienungshandbuch aber verschweigt, wohl in der Hoffnung neue Druckköpfe zu verkaufen, ist die Möglichkeit, das weggelöste Öl zu ersetzen!

Hier ist also mein Reinigungsverfahren im einzelnen:

1. Stecker ziehen und nach Anleitung im Bedienungshandbuch Farbband entfernen und Druckkopf aus seiner Halterung lösen. Sie brauchen das Flachbandkabel aber **nicht** aus der Steckerleiste zu lösen.
2. Mit in Isopropylalkohol getränkten Wattestäbchen Tintenreste von den Nadeln gründlich entfernen. Die Nadeln werden im Druckkopf durch eine Zwischenwand geführt und treten dann durch die Vorderwand der Druckkopfnase aus. Die wichtigsten zu reinigenden Stellen sind die Austrittstelle, aber auch **hinter** der Vorderwand der Nase, wo sich viel Tinte ansammelt. Isopropylalkohol ist in Apotheken erhältlich und hat den Vorteil, daß es keine Zusätze enthält und somit keine Rückstände hinterläßt (ich benutze es auch zur Reinigung von Schallplattenabtastnadeln).
3. Das Alkohol verdunsten lassen, und anschließend hinter der Nasenvorderwand und vor der Zwischenwand jeweils einen Tropfen *Nähmaschinenöl* auf die Nadeln geben.
4. Druckkopf wieder einbauen, Stromstecker wieder verbinden, und nun **ohne Farbband** zwei oder drei Seiten Druckerselbsttest drucken lassen, um überschüssiges Öl abzuschütteln. Jetzt ein Farbband einbauen (am besten ein altes, verbrauchtes), und wieder einige Seiten Selbsttest drucken, um die letzten Ölreste durch das Band aufsaugen zu lassen. Die letzten Ölreste machen sich durch unbeabsichtigt **Fett**gedruckte Buchstaben bemerkbar; wenn diese nicht mehr erscheinen, können Sie nach ein oder zwei Seiten den Selbsttest abbrechen.
5. Richtiges Farbband einlegen, fertig! (Bemerkung: es kann sich in der ersten Zeit noch etwas Öl ansammeln, während der Drucker ausgeschaltet ist, das beim nächsten Drucken die ersten paar Zeilen etwas verschmutzt. Dieses Problem hört aber schon beim zweiten oder dritten Mal wieder auf).

Ich hoffe, daß diese Tips einigen Epson Besitzern die Kosten für einen neuen Druckkopf ersparen werden!

Bochum, den 9. 5. 1985 Gordon Wassermann

********** Übersetzung **********
********** aus *DAICLIC* Nr. 2 **********
**** Zeitschrift des International DAI Club, Belgien ****

EINE MAUS FÜR DEN DAI.

von Dominique CARLIER (DAIC).

Als DAI Benutzer war ich schon immer auf der Suche nach Mitteln, die es mir erlauben, den DAI als Nutzmaschine einzusetzen. Da die Steuerknüppel des DAI leider nicht für alle Anwendungen geeignet sind, habe ich mich für eine Lösung entschieden, die mir besser zu sein schien. So habe ich mir eine "Color Mouse" bei Tandy besorgt. Wirft man schnell einen Blick darauf, so stellt sich heraus, daß man an dem fraglichen Gerät einige Anpassungen vornehmen muß, um es auf geeignete Weise mit dem DAI verwenden zu können.

Als erstes muß man den Stecker umbauen. Und dann muß eine Leiterbahn in der Maus durchgetrennt werden. Bild 1 skizziert den elektrischen Plan der Maus und die Farben der Steckerdrähte (so wie ich sie in meinem Exemplar vorgefunden habe). In Klammern finden Sie die entsprechenden Anschlußnummern der DAI Verbindungsbuchse.

Nachdem diese Änderungen ausgeführt wurden, wird die Maus über einen Paddleeingang angeschlossen und wie ein Paddle verwendet.

****
* Bild 1 *
****

TRS-80 COLOR MOUSE AM DAI-PC

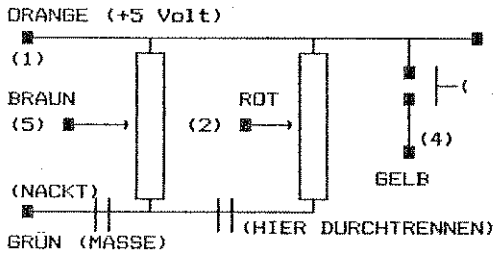
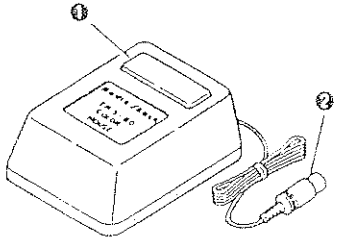


Bild 2 vermittelt einen Eindruck vom Aussehen dieser Maus und in Bild 3 finden Sie die verschiedenen Bewegungen, die möglich sind. Hier ebenfalls einige technische Merkmale:

maximale Bewegung		127 mm
Kabellänge		1,5 m
Maße:	Länge	90 mm
	Breite	62 mm
	Höhe	34 mm
Gewicht		210 g

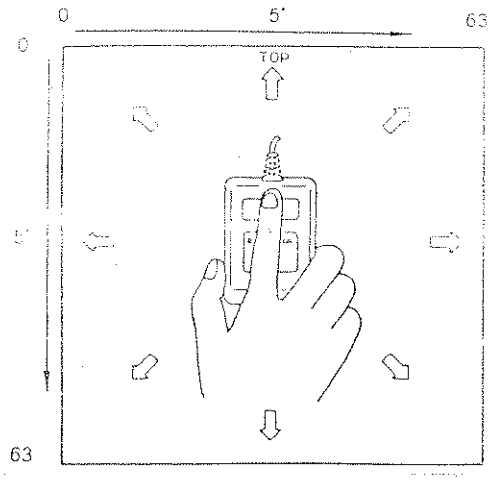
****
* Bild 2 *
****



Diese Maus (Bestellnummer 26-3025) kostete, im Jahre 1984, 2295 belgische Franken (etwa DM 115,) und ist also nicht eine der Billigsten. Wenn Sie nur zweidimensionale Steuerknüppel mit einem Druckknopf wünschen, muß man auf die anderen Tandy Erzeugnisse hinweisen:

- Color Computer Joystick, Bestellnr. 26-3008, für FB 1295 (DM 65,) das Paar.
- Precision Joystick, Bestellnr. 26-3012, für FB 1295 (DM 65,) das Stück.

****
* Bild 3 *
****



Wenn Sie vorhaben, eine Maus anzuschließen, wie vorhin beschrieben, werden Ihnen diese Bemerkungen sehr nützlich sein.

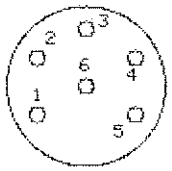
Einige Informationen:

- Die Maus wird am Ausgang PDL 1 oder PDL 2 angeschlossen. Sie liefert, ganz wie ein Potentiometer oder ein Steuerknüppel, einen genauen Wert von 0 bis 255 in X und Y.
- Die Maus ist vollkompatibel mit Programmen, die die Paddles verwenden, und ohne jeglicher Änderung dieser Programme. Sicher, die Verwendung der Maus mit manchen Programmen macht sie schwierig zu benutzen, aber es gibt Andere, wo die Maus besonders geeignet ist, z.B. das ZEICHNUNGSEDITIERPROGRAMM.
- Wir raten Ihnen, die Maus auf einer ebenen Fläche zu verwenden, am besten auf einem Blatt Papier, um eine gewisse Haftung zu haben.
- Es wird angeraten, die Kugel nicht mit den Fingern zu berühren und diese Kugel regelmäßig zu reinigen (entfetten...).

Anpassung:

Den mit der Maus mitgelieferten Stecker (5-polig) entfernen und die abisolierten Drähte in einen neuen Stecker (6-polig) einlöten, wie in Bild a.

```
**********
* Bild a *
**********
        gesehen
        von innen
```



Stifte - Farben
1 - ORANGE
2 - ROT
3 -
4 - GELB
5 - BRAUN
6 - GRÜN

Die Maus wenden (Kugel nach oben) und die 3 Schrauben (am Rand) lösen, die die 'Haube' halten. Vorsichtig öffnen und aufpassen, daß die Kugel oben bleibt. Die Haube mit der Kugel und dem (roten) EVENT Knopf absetzen und darauf achten, die kleine Feder nicht zu verlieren.

Die bedruckte Platine abschrauben, damit Sie die von Masse (grüner Draht) kommende Leiterbahn durchtrennen können, an den zwei Stellen, die in Bild 1 beschrieben sind (siehe oben). Nachdem diese Operation ausgeführt wurde, die Platine wieder befestigen.

Die roten und schwarzen Drähte vertauschen, die an den Enden des Potentiometers anliegen, der die senkrechten Bewegungen mißt.

Das Ganze wieder zusammenbauen, wobei man darauf achten soll, zum einen die Kugel nicht zu viel mit den Fingern zu berühren, und zum anderen den kleinen Stift nicht zu zerstören, auf dem die Feder sitzt.

Und nun, ans Programmieren.

Fabrice DULUINS für D.A.I.C.

übersetzt von Gordon Wassermann September 1985

```
1     REM ************************************
2     REM *   DEMO MAUS IN DER TEXTSEITE     *
4     REM * (c) Alain Mariatte & DAlclic 85 *
5     REM ************************************
6     GOSUB 1000
7     AD%=#BFE7:X%=0:Y%=0:P%=32:GOTO 55
10    X%=PDL(0):Y%=PDL(2)
11    REM JE NACH IHRER DAI VERSION EVENTUELL ABANDERN Y%=PDL(1)
20    X%=X%/4:Y%=Y%/11:IF X%>59 THEN X%=59
21    IF PEEK(#FD00) IAND 32=32 THEN 200
22    IF XX%=X% AND YY%=Y% THEN 10
23    POKE AD%,P%:POKE AD%-3,0
25    CURSOR X%,Y%:AD%=PEEK(#73)*256+PEEK(#72):P%=PEEK(#77)
30    PRINT CHR$(P%):POKE AD%-3,#FF
33    CURSOR 4,23
55    XX%=X%:YY%=Y%
100   WAIT TIME 1:GOTO 10
200   SOUND 0 0 15 0 FREQ(440.0):WAIT TIME 10:SOUND OFF
210   G%=GETC:G%=GETC:G%=GETC
211   G%=GETC:IF G%=0 THEN 211
220   POKE AD%,G%:GOTO 25
1000  MODE 0:COLORT 8 0 10 0:PRINT CHR$(12)
1001  POKE #BA,#69:POKE #BB,#BF
1010  POKE #BA,#69:POKE #BD,#BF
1020  POKE #BC,#E5:POKE #BD,#B3
1030  POKE #84,#D5:POKE #85,#B3
1040  POKE #8E,#D5:POKE #8F,#B3
1050  PRINT CHR$(12):LIST 1-220
1060  RETURN
```