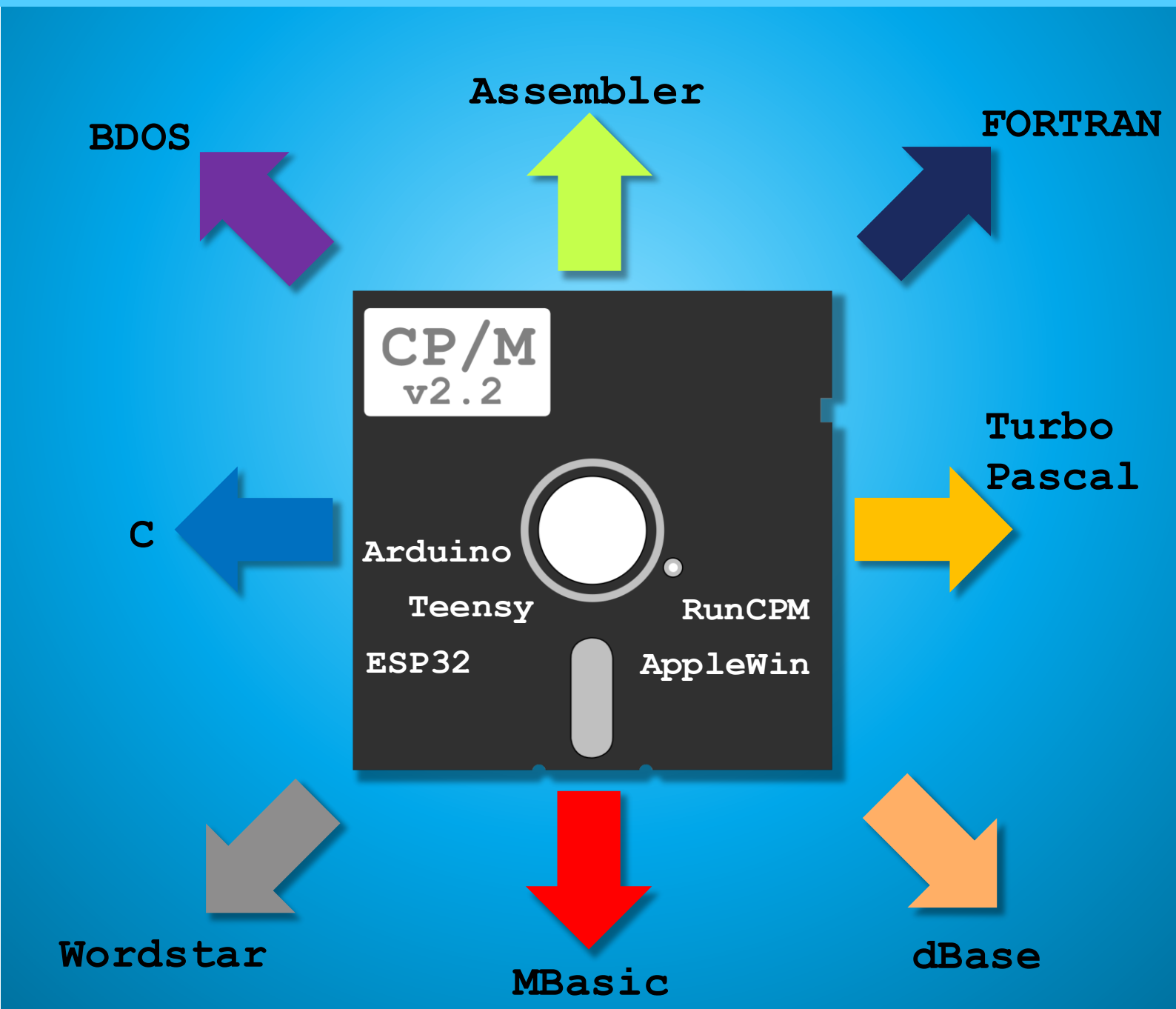
Erik Bartmann

Spaß mit CP/M



Dieses Dokument	
Titel	Spaß mit CP/M
Thema	Das Betriebssystem CP/M
Erstellt am	01.02.2023
Erstellt von	Bartmann, Erik
Version	1.01
Dateiname	Spass mit CPM.docx

Ihr Ansprechpartner	
Name	Erik Bartmann
E-Mail	erk.bartmann@yahoo.de
Internet	https://erik-bartmann.de/

Copyright 2024 – Erik Bartmann

Inhalt

1 Über Sinn und Unsinn

Wenn jemand den Reiz verspürt, sich mit einer etwas älteren, aber immer noch sehr interessanten Technik auseinanderzusetzen, dann ist er hier genau richtig gelandet. Die Überschrift klingt schon etwas merkwürdig und provokant und das ist auch beabsichtigt. Über Sinn und Unsinn sollte man sich heutzutage in allen Bereichen des Lebens Gedanken machen, denn die Wahrheiten, die uns überall ins Gesicht springen, entpuppen sich nicht selten - vorsichtig formuliert - als Halbwahrheiten und sollten hinterfragt werden. Habe ich Halbwahrheiten gesagt?

Ich habe dieses kleine Buch geschrieben, weil es einfach Spaß bereitet, sich mit derartiger Technik auseinander zu setzen und weil jeder es machen kann, der sich dafür interessiert. Wer es als Zeitverschwendung ansieht, der wird es mit großer Wahrscheinlichkeit auch nicht bis hier hingeschafft haben, um dies Zeilen zu lesen.

Es ist für den Einstieg noch nicht einmal erforderlich, ein eigens CP/M-System mit dedizierter Hardware über einen Z80-Prozessor mit RAM und ROM und allem, was sonst noch dazugehört, aufzubauen. Es reicht ein Mikrocontrollerboard wie zum Beispiel

- Teensy 3.6
- Arduino Due
- ESP32
- Raspberry Pi Pico

Das Projekt, was ich nutze, lautet ***RunCPM***. Unter dem folgenden Link sind alle erforderlichen Details zu finden, die zur Umsetzung des Vorhabens notwendig sind.

https://github.com/MockbaTheBorg/RunCPM

RunCPM ist vollständig mit der Programmiersprache ***C*** und auf modulare Weise geschrieben, so dass die Portierung auf andere Plattformen ein Kinderspiel ist. Es sollten laut Programmierer des Projektes keine Änderungen an den Hauptmodulen des Codes notwendig sein und ich kann dem zustimmen, denn ich bin auf keine Probleme bei unterschiedlichen Hardware-Plattformen gestoßen.

Wer sich zudem mit Programmen wie ***Wordstar***, ***dBaseII***, ***mBasic*** und anderen aus jener Zeit befassen möchte, so ist RunCPM natürlich ebenfalls dafür geeignet. RunCPM emuliert ***CP/M*** (Version 2.2 - 3.0 soll in Vorbereitung sein) von ***Digital Research*** so genau wie möglich, mit dem einzigen Unterschied, dass es normale Ordner auf dem Host anstelle von Disk-Images verwendet. Und das ist einfach fantastisch. Über eine SD-Karte, die mit dem Mikrocontrollerboard über den SD-Slot verbunden ist, kann man sich mit Unmengen an Software versorgen, die frei im Internet verfügbar ist. Einfach runterladen, raufkopieren und loslegen. Hier ein paar Eckpunkte, was RunCPM so attraktiv macht.

- Die Ausführungsgeschwindigkeit von CP/M-Anwendungen ist sehr hoch. Bei der Nutzung von Spielen, kann sich eine derart hohe Ausführungsgeschwindigkeit jedoch negativ auswirken
- Es wird zur Speicherung von Programmen und Daten das Windows-Dateisystem genutzt, was ein entscheidender Vorteil gegenüber sonst üblichen CP/M-Disk-Images ist, wo es recht umständlich ist, Daten hinzuzufügen beziehungsweise das jeweilige Disk-Image zu manipulieren

Nachfolgend ist diese Dateisystem zu sehen, wobei der Zusammenhang zwischen ***Laufwerk*** und ***User-Bereich*** noch erläutert wird.

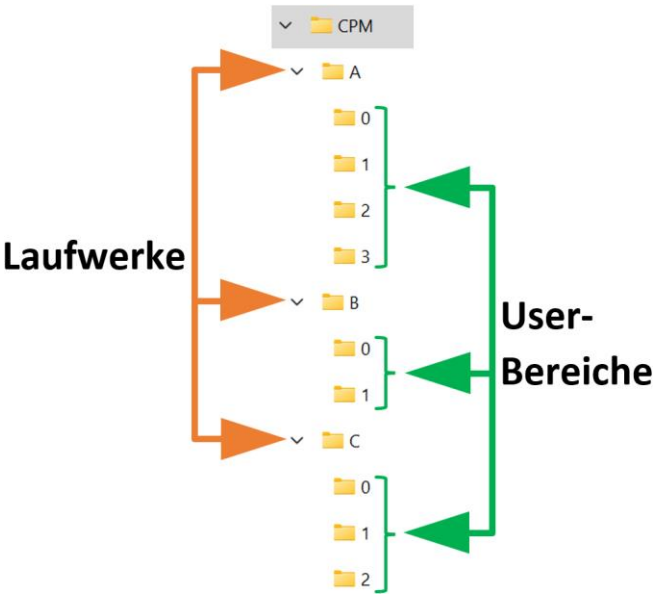


Abbildung 1 - Laufwerke und User-Bereiche

Hört sich alles wie eine Werbeveranstaltung an, nicht wahr?! Auf der folgenden Abbildung ist ein *Teensy 3.6-Board* mit einem SD-Kartenslot zu sehen, was die einfachste Variante darstellt, denn es muss kein separater SD-Kartenslot mit dem Board verbunden werden, wie das zum Beispiel beim *Arduino Due* oder *ESP32* der Fall wäre.

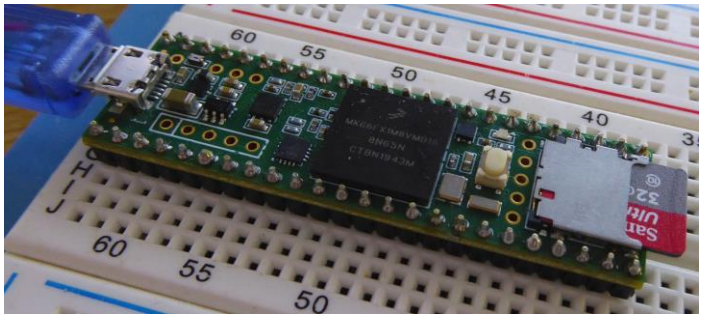


Abbildung 2 - Das Teensy 3.6-Board als CP/M-Computer

Hier die Umsetzung mit dem *Arduino-Due* und einem SD-Karten-Modul.

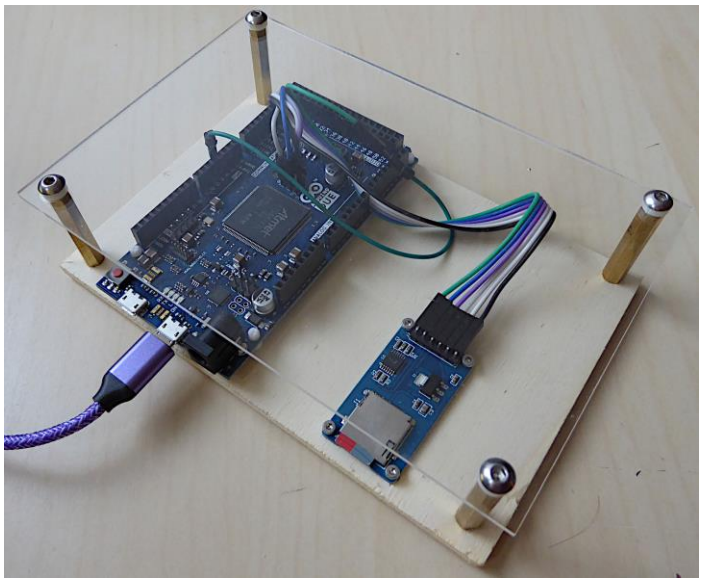


Abbildung 3 - Das Arduino-Due-Board als CP/M-Computer

Hier die Umsetzung mit einem Raspberry Pi-Pico-Board und einem SD-Karten-Modul. Unter dem folgenden Link sind alle Informationen dazu zu finden.

https://github.com/guidol70/RunCPM_RPi_Pico

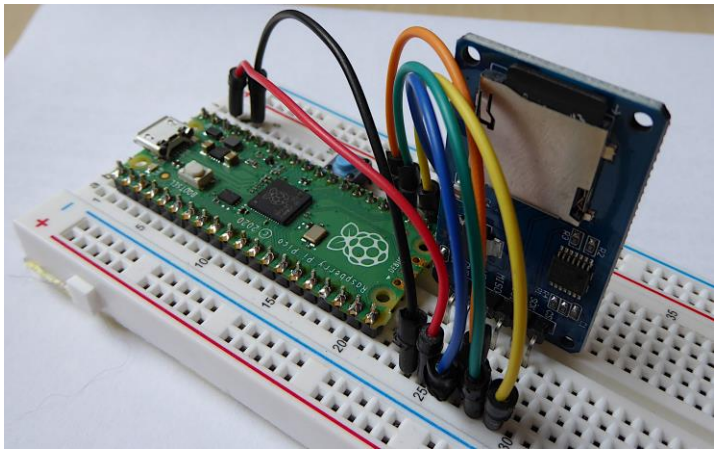


Abbildung 4 - Das Raspberry Pi-Pico-Board als CP/M-Computer

Abschließend zur Realisierung von RunCPM auf verschiedenen Plattformen hier noch der Aufbau mit meinem ESP32-Pico-Discoveryboard und einem SD-Kartenmodul-Adapter. Es ist wichtig, dass hier kein SD-Karten-Modul mit integriertem Levelshifter zum Einsatz kommt.

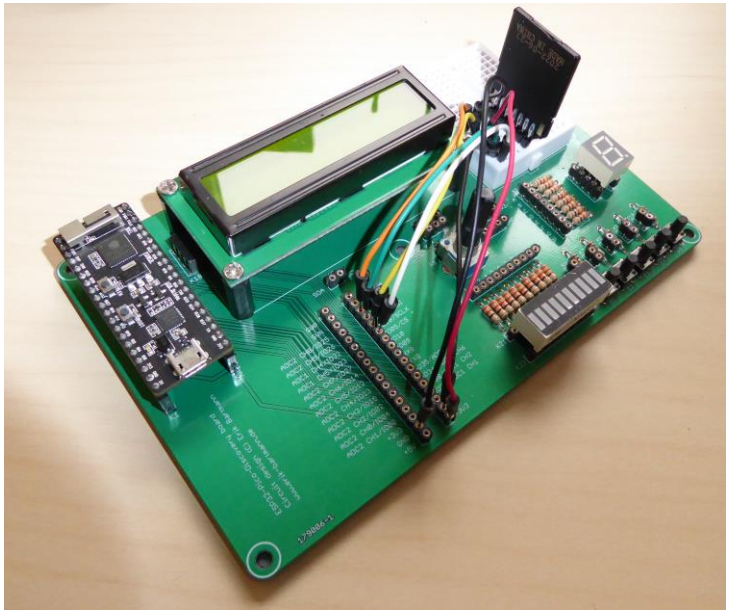


Abbildung 5 - Das ESP32-Pico-Board als CP/M-Computer

Am Ende des Buches gehe ich noch einmal genauer auf die Verschaltung der Module ein, die aber natürlich auch im Internet zu finden sind.

Zudem spreche ich hier zahlreiche Programmiersprachen wie ***Z80***-Maschinensprache, ***C***, ***MBASIC*** oder ***Turbo Pascal*** an. Es geht mir primär nicht darum, eine detaillierte Einführung dieser Sprachen zu liefern, sondern direkt konkrete Beispiele zu zeigen und die darin benutzten Sprachelemente zu beleuchten. Wer sich näher mit diesen Programmiersprachen befassen möchte, dem sei geraten, auf die vielen freien eBooks oder Tutorials aus dem Netz zuzugreifen und wer ein bisschen persönliche Suche investiert, findet Tonnen an Material, das für zwei oder mehr Leben reicht.

Eine wunderbare deutsche Seite nennt sich ***Gaby's Homepage für CP/M und Computergeschichte*** und ist unter der folgenden Internetadresse zu finden.

http://www.gaby.de/

Gaby hat mir die Erlaubnis gegeben, Links zu diversen CP/M-Themen zu nutzen und ich danke herzlich dafür!

In einem eigenen Kapitel - ***CP/M auf realer Hardware*** - werde ich auf die einzelnen Punkte der Installation auf geeigneter Hardware genau eingehen, will aber zu Beginn nicht damit um die Ecke kommen.

Ich möchte hier nicht mit zu vielen Details aufwarten und wer meine bisherigen Bücher kennt, der ist über meine Didaktik bestens informiert. Ich liefere die benötigten Informationen immer genau dann, wenn sie gebraucht werden. Die Verblödungsstrategie, der ich in meiner Jugend in diversen Anstalten ausgesetzt bzw. ausgeliefert war, führe ich nicht fort! Das System ist dem Untergang geweiht und wir arbeiten gerade nach besten Kräften daran.

Ich wünsche jedenfalls viel Spaß mit ***CP/M*** und allem, was dazugehört und das ist eine Menge!

Der Autor, *Erik Bartmann*

2 Das Betriebssystem CP/M

Um mit einem Computer in Kommunikation zu treten, muss nach Möglichkeit eine Software installiert sein, die es dem Nutzer gestattet, mehr oder weniger komfortabel eine Interaktion zwischen Mensch und Maschine zu führen. Man tippt also zum Beispiel Befehle in eine sogenannte Kommandozeile ein und versendet damit Anfragen an den Rechner, der entsprechende Aktionen ausführt und Antworten präsentiert. Natürlich geht das im Zeitalter der modernen Betriebssysteme wie *Windows*, *Mac OSX* oder Linux zumeist über eine grafische Benutzeroberfläche, die über das Eingabegerät Maus sehr einfach Programme startet um dann zum Beispiel über die Tastatur Texte in schicken Fenstern zu erstellen beziehungsweise zu verwalten, Grafiken zu kreieren oder auch Musik zu komponieren und abzuspielen. Das sind nur sehr wenige Beispiel einer Vielzahl von Einsatzmöglichkeiten eines modernen Computers. Die Geschichte der Betriebssysteme hat jedoch etwas angefangen. In den 1970er Jahren gab es einen amerikanischen Bastler und Computer-Freak mit Namen *Gary Kildall*. Man sagt, dass er ein Diskettenlaufwerk bekommen hatte und nun bestrebt war, dieses an seinen selbst gebauten Computer anzuschließen. Sicherlich kein einfaches und alltägliches Unterfangen dieser Tage. Es war jedoch so intelligent, dass er sich kurzerhand selbst eine passende Schaltung – einen Controller – bastelte, um das Laufwerk nutzen zu können. Doch derartige Hardware ist ohne eine geeignete Software eigentlich nicht viel Wert und unbrauchbar. Also programmierte er noch eine – man würde heute sagen – Treibersoftware, die in der Lage war, Informationen beziehungsweise Daten auf eine Diskette zu schreiben und auch wieder zu lesen. Das war quasi die Geburtsstunde für *CP/M*, was für ***C**ontrol-**P**rogram for **M**ikrocomputers* steht. Kildall gründete daraufhin eine Firma mit dem Namen *Digital Research*.

Bei CP/M handelte sich also um das erste *DOS* (Disk Operating System) und war somit der Vorläufer für die heute geläufigen Microsoft- DOS/Windows-Systeme, wobei MS-DOS selber von einem CP/M-Clone namens *QDOS* (Quick and Dirty Operating System) abstammt, das von *Tim Patterson* programmiert und von Microsoft für den ersten IBM PC eingekauft wurde. Microsoft hat sich für das Betriebssystem DOS an CP/M orientiert und diese Grundlagen gnadenlos ausgenutzt, um ein eigenes Produkt auf den Markt zu bringen. Wurde es *„geklaut*"? Das muss jeder für sich selbst entscheiden und recherchieren. Wer die Wahrheit sucht, für den sprechen die Fakten für sich! Es ist nichts so, wie es scheint! Und schon sind wir beim eigentlichen Thema dieses Kapitels angelangt. CP/M wurde sofort zum Standard-Betriebssystem für Prozessoren wie den Z80, 8080 und 8085 erkoren, die sich alle sehr ähnlich sind. Dadurch entstand eine sehr große Anzahl von Programmen, die auf Rechnern liefen, die diese CPUs besaßen. Zahllose Programme wie zum Beispiel Assembler, diverse

Programmiersprachen wie ***BASIC***, ***PASCAL***, ***FORTRAN*** wurden für CP/M entwickelt oder portiert und CP/M erfreute sich dadurch großer Beliebtheit. Auch heute noch – 50 Jahre später – hat dieses Betriebssystem nichts von seinem Charme verloren und viele Entwickler basteln Platinen, die einen Z80 als CPU besitzen, um Software aus vergangenen Zeiten wieder aufleben zu lassen.

CP/M gibt es in unterschiedlichen Versionen für 8-Bit-Systeme wie zum Beispiel das CP/M 2.2, das auch CP/M-80 genannt wird und für Prozessoren wie 8080, 8085 und Z80 verwendet wird. Des Weiteren gibt es CP/M 3.0, das auch CP/M-Plus genannt wird. Im Zuge der Erweiterung auf 16-Bit-Systeme gibt es das CP/M-86 für die Prozessoren 8086 und 8088 beziehungsweise das CP/M-86K für den Prozessor 68000.

Da CP/M nun die Basis für den Betrieb eines derartigen Computers darstellt, werden wir uns einigen grundlegenden Strukturen beziehungsweise Befehlen widmen, damit der Einstieg im Umgang damit auch gelingt und der Spaßfaktor nicht zu kurz kommt. Natürlich wird das hier keine umfassende und komplette Einführung in CP/M werden, doch es reicht dafür aus, erste Erfahrungen zu sammeln. Eine frei verfügbare Buchliste findet sich am Ende meiner Ausführungen. Wird ein CP/M-Rechner eingeschaltet, ist das vorhandene ROM, also der Festwertspeicher, zum Start des eigentlichen Betriebssystems von Diskette erforderlich. Es muss also dafür gesorgt werden, dass immer mindestens ein Diskettenlaufwerk angeschlossen und eine Boot-Diskette enthalten ist, auf der die erforderlichen Dateien des Betriebssystems enthalten sind.

2.1 BIOS, BDOS und CCP

Woraus setzt sich denn CP/M eigentlich zusammen? Nun, bei CP/M handelt es sich sowohl um ein spezielles Programm, als auch um eine Sammlung von Programmen, die den Umgang mit CP/M ermöglicht beziehungsweise erleichtert. Im Grunde genommen wird zwischen zwei Softwaregruppen unterschieden. Da ist zum einen die Systemsoftware, die über das CP/M mit internen Kommandos abgebildet wird und zum anderen eine Sammlung diverser (Dienst)-Programme, auch *Tools* genannt. Prominente Vertreter der Tools sind Programme wie *ED* oder *PIP*, auf die ich natürlich noch im Detail eingehe. Der Unterbau von CP/M ist in drei wesentliche Komponenten gegliedert.

2.1.1 BIOS

BIOS ist die Abkürzung für Basic Input Output System und stellt die Basisfunktionen der Ein-/Ausgabe bereit. Es handelt sich um den systemabhängigen Teil und ist für die Ansteuerung der hardwareseitigen Systemkomponenten verantwortlich.

2.1.2 BDOS

BDOS steht für *Basic Disk Operating System* und ist der Betriebssystemkern, der für grundlegende Laufwerksverwaltung und der gesamten Ein- und Ausgabe verantwortlich ist. Dieser Teil steht in direktem Kontakt mit dem BIOS. Das BDOS ist ein hardwareunabhängiger Teil des Betriebssystems und in allen CP/M 2.2-Computern ist dasselbe BDOS vorhanden. Es stellt dabei einen Satz von

Funktionen zur Verwaltung des Rechners zur Verfügung und nutzt die BIOS-Routinen.

2.1.3 BDOS-Fehlermeldungen

BDOS kennt vier grundlegende Fehlermeldungen.

- **Bdos Err On <Laufwerk>: Bad Sector** (Eine Aufzeichnung konnte infolge eines Diskettenfehlers entweder nicht gelesen oder nicht geschrieben werden)
- **Bdos Err On <Laufwerk>: Select** (Das angeforderte Laufwerk ist nicht vorhanden oder nicht aktivierbar)
- **Bdos Err On <Laufwerk>: R/O** (Das entsprechende Laufwerk ist schreibgeschützt)
- **Bdos Err On <Laufwerk>: File R/O** (Die entsprechende Datei ist schreibgeschützt)

2.1.4 CCP

CCP ist die Abkürzung für *Console Command Processor* und fungiert als Befehls-Bearbeiter-Instanz. Über die Eingabeaufforderung werden die wichtigsten CP/M-Kommandos verarbeitet. Die über die Konsole eingegebenen Zeichen werden angenommen, interpretiert und dann auch - wenn die Syntax stimmt - ausgeführt. Wenn es zum Beispiel darum geht, den Inhalt eines Speichermediums zur Anzeige zu bringen, dann wird *DIR* (DIR: Directory) eingegeben. Soll eine Datei gelöscht werden, kommt *ERA* (ERA: Erase) zum Einsatz. Das sind nur zwei Beispiele

Des Weiteren gibt es noch die folgenden beiden Bereiche.

- **TPA**: *Transient Program Area* - Speicherbereich für Anwenderprogramme
- **System**: *Low-Storage* (Zero-Page) - System-Puffer und Parameter

Man kann sagen, dass CP/M ein Minimum von 20K bzw. 48K erfordern. Die meisten Systeme haben ein Maximum von 64K. Die ersten 256 Bytes von *0000h* bis *00FFh* werden vom Betriebssystem als sogenanntes *Scratchpad* genutzt und enthalten Puffer, Parameter und andere temporäre Daten. Dieser Bereich wird *Low Storage* genannt.

Der Speicherbereich, in dem der Nutzer seine Programme lädt und ausführt, wird - wie schon erwähnt - TPA genannt, wobei die Größe dieses Bereichs von der Größe des Arbeitsspeichers abhängt. Bei einem typischen 64K-System endet die TPA bei *DC00h*, kann aber natürlich variieren.

Wenn ein Programm von der Kommandozeile ausgeführt werden soll, bedeutet es nichts anderes, als dass die Datei in den Speicher beginnend bei Adresse *0100h* geladen und der Code im Anschluss ab Adresse *0100h* ausgeführt wird. Wenn ein Programm mit einem Assembler entwickelt wird, sollte es demnach mit der Zeile ***org 0100h*** begonnen werden.

Der CCP liegt am oberen Ende des freien Speicherbereiches und ist im Gegensatz zum BDOS nicht vor einem Überschreiben geschützt. Es besteht jedoch kein

Problem, wenn längere Programme den *CCP* überschreiben und diesen Speicherplatz für eigene Zwecke nutzen. Die Voraussetzung ist jedoch, dass nach dem Beenden eines derartigen Programms ein Warmstart eingeleitet und somit das Betriebssystem in seinem Urzustand wieder restauriert wird.

Um einen gewissen Überblick zu bekommen, wie diese Bereiche im Speicher angeordnet sind, ist die nachfolgende Grafik sicherlich hilfreich, wenn auch hinsichtlich der Speicheradressen nicht allgemein gültig, denn es gibt unterschiedliche Konfigurationen.

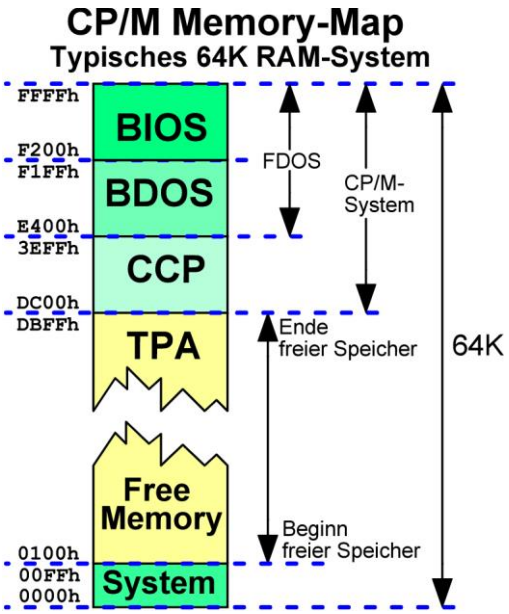


Abbildung 6 - Die Memory-Map für ein 64K-System

Wenn wir später eigene Programme erstellen, dann ist der TPA-Bereich ab Adresse ***0100h*** natürlich sehr interessant, denn dort werden die Programme abgelegt und zur Ausführung gebracht. Doch bevor es soweit ist, müssen wir uns ein paar grundlegende Befehle ansehen, um mit dem CP/M-Betriebssystem überhaupt arbeiten zu können.

Ich möchte an dieser Stelle noch einmal auf die schon erwähnte Zero-Page zu sprechen kommen, die in der zuvor gezeigten Abbildung mit ***System*** gekennzeichnet ist. Die Zero-Page - ein weiterer Namen ist *Base-Page* - ist eine Datenstruktur, die in CP/M-Systemen für die Kommunikation zwischen Programmen und dem Betriebssystem verwendet wird. In 8-Bit-CP/M-Versionen befindet sie sich in den ersten 256 Bytes (0000h bis 00FFh) des Speichers und hat deswegen diesen Namen.

Nun möchte ich nicht alle Speicherstellen mit ihren Bedeutungen durchgehen, doch die hier genannten sind schon recht wichtig und geben Aufschluss über bestimmte Speicherbereiche beziehungsweise Startadressen, in denen relevante Teile des CP/M-Systems zu finden sind.

Offset	Größe	Bedeutung
00 - 02	Code	Sprung zum BIOS
03	Byte	I/O-Byte, eine optionale Funktion, die die Neuzuweisung von Geräten in CP/M 2 ermöglicht

04	Byte	Aktueller Command-Processor (niedrige 4 Bits) und Benutzernummer (hohe 4 Bits)
05 - 07	Code	Sprung zum BDOS-Eintrag - Einstiegspunkt für den Hauptsystemaufruf. Dies ist auch die Adresse des ersten Bytes des vom Programm nicht nutzbaren Speichers

Tabelle 1 - Zero-Page-Adresse (nur die ersten 8 Bytes)

Warum zeige ich gerade diese hier? Nun, es ist gut zu wissen, wo sich der Beginn des ***BDOS*** beziehungsweise des ***BIOS*** befindet. Mit dem Programm ***ddt***, auf das wir noch zu sprechen kommen, können wir uns Speicherbereiche mit deren Inhalten und entsprechenden Assemblerbefehlen ansehen. Wir geben also auf der Kommandozeile ***ddt*** ein und anschließend ***10000*** (ein kleines ***L*** mit vier Nullen). Die Ausgabe schaut wie folgt aus.

```
A0>ddt
DDT VERS 1.4
-10000
  0000  JMP  FE03
  0003  DCR  A
  0004  NOP
  0005  JMP  ED00
  0008  NOP
  0009  NOP
  000A  NOP
  000B  NOP
  000C  NOP
  000D  NOP
  000E  NOP
-
```

Wir sehen an den ersten drei Speicherstellen den Befehl ***jmp FE03***, was ein unmittelbarer Sprung zur gezeigten Adresse bedeutet. Laut Zero-Page-Tabelle handelt es sich also um die Einsprungadresse des *BIOS*. Ab Adresse *0005* ist der Befehl ***jmp ED00*** zu sehen, was einen Sprung zur genannten Adresse bewirkt und laut Tabelle die Einsprungadresse des *BDOS* ist.

Hier aber ein wichtiger Hinweis, denn am Anfang hat mich das auch etwas verwirrt und hier und da tuen sich sicherlich Fragen auf, wobei man doch vorher gedacht hatte, es verstanden zu haben. Bei den meisten der CP/M-Emulatoren, die im Internet zu finden sind, handelt es sich um Z80-Emulatoren. Sie emulieren eine vollständige Z80-Hardware mit Z80-CPU und RAM/ROM/Disk-Controller usw. Sie sind virtuelle Z80-Computer, auf denen ein CP/M-System läuft. Sie haben sie jedoch einen großen Nachteil: Da sie eine vollständige Hardware mit einem Festplatten-Controller emulieren, sind sie gezwungen, Festplatten mit Hilfe von sogenannten Disk-Image-Dateien zu emulieren, die normalerweise die Dateiendungen ***.img*** oder ***.dsk*** besitzen. Daraus ergibt sich eine gravierende Einschränkung, die darin besteht, dass kein voller Zugriff auf die CP/M-Dateien vom Host-Betriebssystem aus besteht. Müssen Dateien hinzugefügt, geändert oder entfernt werden, bedeutet das immer ein Extrahieren beziehungsweise Hinzufügen aus oder zum Disk-Image.

RunCPM emuliert dabei nicht eine Z80-Hardware und ist eigentlich eine "Virtuelle Maschine", die eine Emulation des CP/M-BIOS und -BDOS erstellt und Anfragen in Aufrufe an das Host-Betriebssystem übersetzt, wo immer das möglich ist. Die Dateien werden auf dem Dateisystem des Host-Betriebssystems gespeichert. Die Vorteile, die sich daraus ergeben, bestehen darin, dass jederzeit ein voller Zugriff auf die CP/M-Dateien besteht. Man muss lediglich den entsprechenden Ordner aufrufen und Änderungen nach Belieben vornehmen.

Die Nachteile bestehen darin, dass es sich nicht um eine Hardware-Emulation handelt und dementsprechend, kein ROM, kein BIOS und kein BDOS existieren. Es gibt nur eine virtuelle Maschine, die alle Aufrufe des BIOS oder BDOS entgegennimmt und an das Host-Betriebssystem weiterleitet. Die Adressen für die BIOS- und BDOS-Einträge werden nur aus Gründen der Kompatibilität mit alten Programmen behalten. Es gibt also keine tatsächlichen Z80-BIOS- oder BDOS-Codes an diesen Stellen. Es sind nur Einsprungpunkte, die die Kontrolle an den *RunCPM* (nativen) Code übergeben, wenn diese aufgerufen werden. Der Unterschied eines RunCPM-Systems zu einem realen CP/M-System gestalten sich wie auf der folgenden Abbildung zu sehen ist.

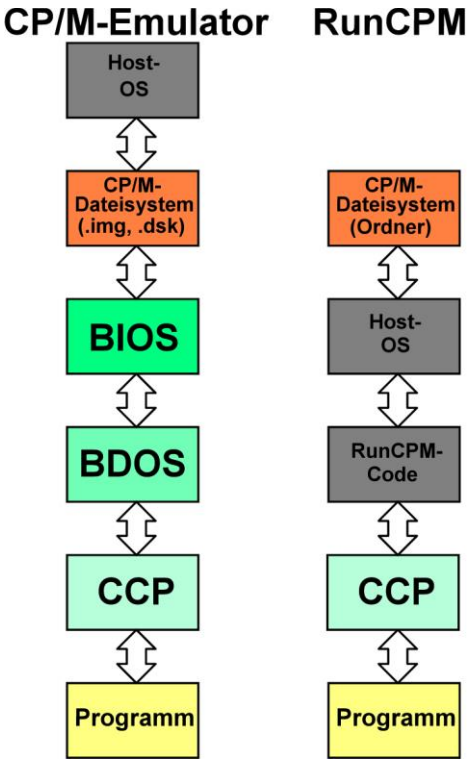


Abbildung 7 - Die Gegenüberstellung eines CP/M-Emulators und RunCPM

2.1.5 Der eigentliche Boot-Prozess von CP/M

Es ist sicherlich zu Beginn etwas verwirrend, wenn man sich die Bereiche von ROM und RAM anschaut. Das Betriebssystem von CP/M befindet sich ja im ROM und doch ist es nach dem Starten von CP/M in einem Bereich zu finden, dass dem RAM zugeordnet ist. Wie kann das sein? Um einen CP/M-Computer zu starten, muss ein sogenannter *Bootup-Code* den eigentlichen CP/M-Code in den Speicher legen und die vorhandene Hardware muss initialisiert werden. Wird der Rechner mit Spannung versorgt, dann beginnt die CPU mit der Ausführung des Programmcode an der Speicherstelle ***0000h***. An dieser Stelle ist auch die erste Adresse des ROMs zu finden. Eine gängige Methode ist, dass der Startup-Code, der sich ja im ROM befindet, das ebenfalls im ROM gespeicherte Betriebssystem an einer bestimmten Stelle in das RAM zu kopieren, um damit den Bootvorgang abzuschließen.

Der Code im ROM sorgt also für den Start des Rechners und erledigt alle erforderlichen Prozeduren für das Booten, um dann nach erfolgreichem

Startvorgang dieses ROM gegen RAM-Speicher auszutauschen. In einem „richtigen" CP/M-System lädt das ROM den CP/M-Bootsektor von der Festplatte und legt ihn zum Beispiel im Speicher des RAMs ab der Speicheradresse ***DC00h*** ab und beginnt dann, den BIOS-Code auszuführen. Ab diesem Zeitpunkt übernimmt das Betriebssystem CP/M die Kontrolle, wobei der *Low-Storage-Bereich*, der in der Abbildung mit *System* gekennzeichnet ist, initialisiert wurde.

Nach dem Start des CP/M-Systems meldet es sich zum Beispiel wie folgt, wobei ich hier eine sehr interessante Lösung über das *Teensy-Board 3.6* mit Namen ***RunCPM*** nutze, wie sollte es anders sein! Primär werde ich dieses Board nutzen, was natürlich auch mit den schon anderen genannten Boards funktioniert.

Abbildung 8 - RunCPM auf einem Teensy-Board

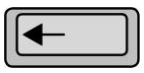
Das Prompt ***A0*** zeigt an, dass wir uns auf dem Laufwerk ***A*** befinden und die nachfolgende ***0*** zeigt den betreffenden User an. Da es bei CP/M noch keine Unterverzeichnisse gab, wurde dieses Manko über unterschiedliche User (0, 1, 2, usw.) realisiert. Später mehr dazu. Ich sprach darüber, dass in CP/M als Systemprogramm interne Kommandos verfügbar sind, die in der sogenannten *CCP* (Console Command Processor) implementiert sind. Es handelt sich dabei um den Kommandozeileninterpreter, der vergleichbar mit der Kommandozeile unter DOS ist. Für DOS wurde hier sehr viel von CP/M entliehen und in der damaligen Zeit hat man sich sehr gerne bei anderen etwas abgeschaut und dann für eigene Zwecke verwendet. Aber das ist reine Spekulation und sicherlich nicht so passiert, denn wir sind ja alles moralisch orientierte Menschen, die keinem etwas Böses wollen.

2.2 Die Eingabe von Steuerzeichen

Bevor wir mit der Eingabe von Befehlen oder anderen Kommandos in CP/M beginnen, ist es sicherlich sinnvoll, über die vorhandenen Steuerzeichen informiert zu sein. Es gibt hier eine ganze Reihe unterschiedlicher Funktionen, die ich jedoch nicht alle zur Sprache bringe. Nachfolgend sind die in meinen Augen wichtigsten zu finden.

2.2.1 Löschen des letzten Zeichens

Wenn es darum geht, dass zuletzt eingegebene Zeichen zu löschen, kann die *Backspace*-Taste genutzt werden.



Der Cursor wird an die Stelle des gelöschten Zeichens gesetzt.

2.2.2 Löschen der gesamten Zeile - Variante 1

Möchte man die ganze Zeile löschen, dann macht es keinen Sinn, die *Backspace*-Taste dir ganze Zeit gedrückt zu halten, bis alle Zeichen gelöscht sind. Bei ein paar wenigen Zeichen macht das zwar noch Sinn, doch bei vielen Zeichen gibt es eine bessere Alternative. Es muss dazu einfach die ***Strg-U***-Tastenkombination gedrückt werden.

Es wird die gesamte Zeile gelöscht und es erscheint ein # am Ende der Zeile. Der Cursor wandert in die nächste Zeile und erwartet eine neue Eingabe.

```
A0>Das ist ja Quatsch hier#
```

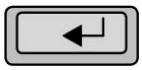
Diese Vorgehensweise hat den Vorteil, dass man den zuvor eingegebenen Text noch sehen kann, um ihn daraufhin korrigiert neu eintippen kann. Es gibt noch eine weitere Variante, die wir jetzt sehen.

2.2.3 Löschen der gesamten Zeile - Variante 2

Über die ***Strg-X***-Tastenkombination kann die gesamte Zeile gelöscht werden und der Cursor befindet sich im Anschluss am Zeilenanfang ohne, dass eine neue Zeile für die erneute Eingabe zur Verfügung steht.

2.2.4 Bestätigung der Eingabe

Um eine Eingabe zu bestätigen, wird die ***RETURN***-Taste gedrückt.



2.2.5 Warmstart für CP/M

Um einen Warmstart in CP/M auszulösen, muss die ***Strg-C***-Tastenkombination gedrückt werden.

2.2.6 Den Cursor nach links und rechts bewegen

Um den Cursor nach links zu bewegen, ohne, dass ein Zeichen wie beim Backspace gelöscht wird, muss die ***Strg-A***-Tastenkombination genutzt werden.

Die Positionierung des Cursors nach rechts erfolgt über die ***Strg-F***-Tastenkombination

2.2.7 Alle Zeichen rechts vom Cursor löschen

Wurde der Cursor zum Beispiel über *Strg-A* nach links positioniert und befindet sich somit in der Mitte irgendeines Textes, dann können mit der ***Strg-K***-Tastenkombinationen alle Zeichen recht vom Cursor gelöscht werden.

Dieser Vorgang wird ***Delete to EOL from cursor*** genannt.

2.2.8 Das letzte Kommando wiederholen

Eine sehr nützliche Tastenkombination ist über ***Strg-W*** zu erzielen. Darüber wird die zuletzt abgesetzte Befehlszeile erneut in der Konsole eingefügt und kann erneut ausgeführt oder editiert werden.

2.2.9 Zwischen Anfang und Ende einer Zeile springen

Über die ***Strg-B***-Tastenkombination kann man zwischen dem Anfang und Ende eines Textes einer Zeile springen.

2.3 Resistente interne Befehle

Nun ist es an der Zeit, sich den internen Befehlen zuzuwenden, um ein Gefühl für das System zu bekommen und damit zu spielen. Hier kommt der schon erwähnte *CCP* (Console Command Processor) ins Spiel, über den einige Befehle zur Verfügung gestellt werden, die den Umgang mit Dateien ermöglichen beziehungsweise erleichtern. Nach einem Kalt- beziehungsweise Warmstart wird das CCP-Programm unterhalb von BDOS geladen und gibt sich mit einem Prompt zu erkennen, über die die Kommunikation mit CP/M gestartet werden kann.

Die folgenden Kommandos sind intern, also fest eingebaut und beziehen sich auf Diskettenoperationen. Diese Befehle sind also nicht im Dateisystem zu finden, sondern sind im Betriebssystem integriert, was deren Ausführungsgeschwindigkeit aufgrund des nicht erforderlichen Zugriffes auf einen Datenträger beträchtlich erhöht.

Kommando	Funktion
DIR	Anzeige des aktiven Disketteninhaltes
ERA	Löschen einer Datei

REN	Umbenennen einer Datei
SAVE	Speichern einer Datei
TYPE	Anzeigen einer ASCII-Datei auf der Konsole
USER	Ändern der Anwendernummer

Tabelle 2 - Interne Befehle

Das zu sehende Promptzeichen ***A0>*** zeigt sowohl den Laufwerksnamen *A*, als auch den User *0* an, wobei CP/M 16 verschiedene Laufwerke von *A:* bis *P:* und User von *0* bis *15* unterstützt. Nach dem Booten ist immer User 0 aktiv. Gehen wir also die essentiell wichtigen internen Befehle einmal durch. Die übrigen Befehle liegen als Programme auf der CP/M-Diskette vor und werden bei Bedarf und auf Anforderung nachgeladen und werden *Transiente Befehle* genannt. Für den „normalen" Anwender ist es eigentlich nicht unbedingt wichtig, zu welcher Kategorie ein Befehl oder ein Programm gehört.

2.3.1 Das DIR-Kommando

Über das ***DIR***-Kommando wird eine Liste der auf dem Standardlaufwerk vorhandenen Dateien erzeugt, die vierspaltig ist. DIR ist die Abkürzung für *Directory*, was übersetzt „Verzeichnis" bedeutet. Wird kein weiterer Parameter angegeben, enthält die Liste alle vorhandenen Dateien. Um eine Filterung nach bestimmten Mustern vorzunehmen, können die Zeichen * beziehungsweise ? als Platzhalter verwendet werden. Ein ? im Suchmuster ist stellvertretend für ein beliebiges Zeichen, ein * für beliebig viele beliebige Zeichen. Es ist dabei zu beachten, dass ein CP/M-Dateiname immer aus 8 Zeichen für den eigentlichen Namen und drei Zeichen für das Suffix besteht. Wird diese Konvention nicht erfüllt, werden Name und Suffix intern mit Leerzeichen aufgefüllt. Eine Datei mit dem Namen

xyz.txt

wird intern gespeichert als

xyz·····.txt

wobei die Punkte für Leerzeichen stehen. Woher hat das Betriebssystem DOS nur die 8.3-Konvention für Dateinamen, die hier zu erkennen ist??? Ein Schelm, der Böses dabei denkt!

CP/M kennt keine Verzeichnisse, wie das von DOS vielleicht geläufig ist. Alle Dateien eines Verzeichnisses liegen demnach auf einer Ebene. Damit entfällt auch die Angabe von Pfadnamen. Soll eine Datei auf einem anderen als dem aktuellen Laufwerk angesprochen werden, so wird der Laufwerksbuchstabe gefolgt von einem Doppelpunkt vor den Dateinamen gestellt, also beispielsweise

B:dump.com

Ein Stern (*) steht also für eine komplette Zeichenfolge und das Fragezeichen (?) für ein einzelnes Zeichen. Hierzu einige Beispiele. Angenommen, es sind die folgenden Dateien auf Laufwerk ***A0***: vorhanden, was über das *DIR*-Kommando ohne weitere Parameter zur Anzeige gebracht wird.

```
A0>dir
A: 1STREAD  ME   |  ASM      COM  |  BDOS     ASM  |  BDOS     LUA
A: BDOS     SUB  |  BDOSEQU  LIB  |  CAL      COM  |  CCP      ASM
A: CCP      SUB  |  CCPZ     SUB  |  CCPZ     Z80  |  CLEAN    SUB
A: CONSOLE7 COM  |  CONSOLE7 Z80  |  CONSOLE8 COM  |  CONSOLE8 Z80
A: DDT      COM  |  DISKDEF  LIB  |  DISPLAY  LIB  |  DUMP     ASM
A: DUMP     COM  |  ED       COM  |  EXIT     COM  |  EXIT     SUB
A: EXIT     Z80  |  FORMAT   COM  |  FORMAT   SUB  |  FORMAT   Z80
A: HELLO    LUA  |  INFO     COM  |  INFO     SUB  |  INFO     TXT
A: INFO     Z80  |  LOAD     COM  |  LU       COM  |  LUA      COM
A: LUA      SUB  |  LUA      Z80  |  LUAINFO  LUA  |  MAC      COM
A: MAKEFCB  LIB  |  MBASIC   COM  |  MDIR     COM  |  MLOAD    ASM
A: MLOAD    COM  |  MLOAD    DOC  |  MOVCPM   COM  |  OPCODES  DOC
A: PIP      COM  |  RSTAT    COM  |  RSTAT    SUB  |  RSTAT    Z80
A: STAT     COM  |  SUBMIT   COM  |  SUBMITD  COM  |  SYSGEN   COM
A: TE       COM  |  UNARC    COM  |  UNCR     COM  |  UNZIP    COM
A: USQ      COM  |  XMODEM   COM  |  XSUB     COM  |  Z2HDR    LIB
A: Z3BASE   LIB  |  Z3HDR    LIB  |  Z80ASM   COM  |  Z80ASM   PDF
A: Z80CCP   ASM  |  Z80CCP   SUB  |  ZCPR2    ASM  |  ZCPR2    SUB
A: ZCPR3    ASM  |  ZCPR3    SUB  |  ZEXALL   COM  |  ZEXDOC   COM
A: ZSID     COM  |  ZTRAN    COM  |  CPM      TXT
A0>
```

Abbildung 9 - Das DIR-Kommando

Nachfolgend werden alle Dateien angezeigt, die die Dateinamenerweiterung *com* besitzen.

```
A0>dir *.com
A: ASM      COM  |  CAL      COM  |  CONSOLE7 COM  |  CONSOLE8 COM
A: DDT      COM  |  DUMP     COM  |  ED       COM  |  EXIT     COM
A: FORMAT   COM  |  INFO     COM  |  LOAD     COM  |  LU       COM
A: LUA      COM  |  MAC      COM  |  MBASIC   COM  |  MDIR     COM
A: MLOAD    COM  |  MOVCPM   COM  |  PIP      COM  |  RSTAT    COM
A: STAT     COM  |  SUBMIT   COM  |  SUBMITD  COM  |  SYSGEN   COM
A: TE       COM  |  UNARC    COM  |  UNCR     COM  |  UNZIP    COM
A: USQ      COM  |  XMODEM   COM  |  XSUB     COM  |  Z80ASM   COM
A: ZEXALL   COM  |  ZEXDOC   COM  |  ZSID     COM  |  ZTRAN    COM
A0>
```

Abbildung 10 - Das DIR-Kommando zur Anzeige von COM-Dateien

Mit dem Voranstellen einer oder mehrerer Buchstaben in Verbindung mit dem Stern können Dateien angezeigt werden, die mit diesen Buchstaben im Namen beginnen. Es ist zu sehen, dass alle Dateien, die mit einem ***c*** beginnen, aufgelistet werden.

```
A0>dir c*.*
A: CAL      COM  |  CCP      ASM  |  CCP      SUB  |  CCPZ     SUB
A: CCPZ     Z80  |  CLEAN    SUB  |  CONSOLE7 COM  |  CONSOLE7 Z80
A: CONSOLE8 COM  |  CONSOLE8 Z80  |  CPM      TXT
A0>
```

Über das Fragezeichen **?** kann erreicht werden, dass Dateien angezeigt werden, die an der betreffenden Zeichenposition ein beliebiges Zeichen besitzen.

```
A0>dir c*p.asm
A: CCP      ASM
A0>
```

Nähere Informationen dazu sind unter der folgenden Internetadresse zu finden.

http://www.gaby.de/cpm/manuals/archive/cpm22htm/ch1.htm#Section_1.4.2

2.3.2 Das ERA-Kommando

Über das ***ERA***-Kommando können vorhandene Dateien vom Datenträger entfernt, also gelöscht werden. ERA ist die Abkürzung für *Erase*, was übersetzt „Löschen" bedeutet. Es können bei diesem Kommando die gleichen Filtereinstellungen, wie beim DIR-Kommando verwendet werden. Um alle Dateien in einem Laufwerk zu löschen, was jedoch gut überlegt sein sollte, muss das folgende Kommando abgesetzt werden.

```
A0>era *.*
```

Dieses Kommando wird auf unterschiedlichen Plattformen verschiedene Auswirkungen, Auf meinem *RunCPM*-System werden unmittelbar und ohne weitere nachfragen alle Dateien gelöscht. Es kann aber auch sein, dass eine vorherige Sicherheitsabfrage erscheint, ob die Löschung aller Dateien auf A1 wirklich durchgeführt werden soll. Fall Ja, muss das mit ***Y*** (Ja) bestätigt werden. Andernfalls sollte schnell die Taste ***N*** (Nein) gedrückt werden, denn es gibt nach einer versehentlich durchgeführten Löschung kein Zurück mehr.

Nähere Informationen dazu sind unter der folgenden Internetadresse zu finden.

http://www.gaby.de/cpm/manuals/archive/cpm22htm/ch1.htm#Section_1.4.1

2.3.3 Das REN-Kommando

Über das ***REN***-Kommando können vorhandene Dateien umbenannt werden. REN ist die Abkürzung für *Rename*, was übersetzt „Umbenennung" bedeutet. Die Umbenennung erfolgt dabei nach dem folgenden Schema

REN <Neu>=<Alt>

Zuerst wird also der neue Name der Datei, dann der alte genannt und zwischen den beiden Namen steht ein Gleichheitszeichen. Wenn die Umbenennung auf dem gerade aktuellen Laufwerk erfolgen soll, bedarf es keiner Laufwerksbezeichnung vor die Dateinamen. Wenn der neue Name einer Datei schon existieren sollte, erfolgt ein entsprechender Warnhinweis. Im Folgenden wird die Datei ***info.txt*** in ***readme.txt*** umbenannt.

```
I0>dir *.txt
I: INFO    TXT
I0>ren readme.txt=info.txt
I0>dir *.txt
I: README  TXT
I0>
```

Es muss darauf geachtet werden, dass kein Leerzeichen vorkommen darf. Die Ausnahme ist das Leerzeichen direkt hinter dem ***ren***-Kommando.

Nähere Informationen dazu sind unter der folgenden Internetadresse zu finden.

http://www.gaby.de/cpm/manuals/archive/cpm22htm/ch1.htm#Section_1.4.3

2.3.4 Das SAVE-Kommando

Das ***SAVE***-Kommando speichert die angegebene Anzahl *n* von Seiten des transienten Programmbereichs auf der Festplatte unter dem angegebenen Dateinamen und wird hauptsächlich von Programmieren benutzt, die die sich intensiv mit der Assembler-Programmierung befassen. Nähere Informationen dazu sind unter der folgenden Internetadresse zu finden.

http://www.gaby.de/cpm/manuals/archive/cpm22htm/ch1.htm#Section_1.4.4

2.3.5 Das TYPE-Kommando

Über das ***TYPE***-Kommando kann der Inhalt einer Datei im *ASCII*-Format angezeigt werden.

```
RunCPM Version 5.7 (CP/M 2.2 60K)

I0>type cities.txt
Berlin
London
Tokio
Rom

I0>
```

Wenn man sich den Inhalt der Datei über das *DUMP*-Programm (das Programm wird später noch genauer erklärt) anschaut, dann sind die einzelnen Buchstaben mit ihren jeweiligen ASCII-Codes zu sehen.

```
I0>dump cities.txt

0000 42 65 72 6C 69 6E 0D 0A 4C 6F 6E 64 6F 6E 0D 0A
0010 54 6F 6B 69 6F 0D 0A 52 6F 6D 0D 0A 1A 1A 1A 1A
0020 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A
0030 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A
0040 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A
0050 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A
0060 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A
0070 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A

I0>
```

Nähere Informationen dazu sind unter der folgenden Internetadresse zu finden.

http://www.gaby.de/cpm/manuals/archive/cpm22htm/ch1.htm#Section_1.4.5

2.3.6 Das USER-Kommando

Da es unter CP/M 2.2 noch keine Unterverzeichnisse gibt, die zur Organisation der einzelnen Dateien sinnvoll wäre, stellt das ***USER***-Kommando eine erste Form der Strukturierung dar. Über dieses Kommando wird es ermöglicht, ein Laufwerk in 16 Bereiche zu unterteilen, wobei sich die Bezeichnung der Bereiche von 0 bis 15 erstreckt. Eine derartige Aufteilung war seinerzeit recht nützlich, wenn sich mehrere Benutzer einen Computer teilen und die jeweiligen Benutzerdateien nicht durcheinanderkommen sollten. Um in einen anderen Bereich zu wechseln, wird das USER-Kommando, gefolgt von der entsprechenden Nummer eingegeben. Das Prompt ändert dann die angehängte Ziffer, die zu Beginn immer den Bereich 0 repräsentiert. Im nachfolgenden Beispiel wurde der Inhalt von *F0*: über das ***DIR***-Kommando aufgelistet, dann auf ***F1***: gewechselt und erneut der Inhalt angezeigt.

```
F0>dir
F: CONFIG   COM  |  CREF80   COM  |  DUP      COM  |  HELLO    BAS
F: HELLO    COM  |  INFO     TXT  |  KEYDEF   COM  |  L80      COM
F: LIB80    COM  |  M80      COM  |  MDIR     COM  |  NSWEEP   COM
F: PIP      COM  |  SCIFN    BAS  |  SUBMIT   COM  |  ZBASIC   COM
F: ZBASIC   HLP  |  ZBASIC   PDF
F0>user 1
F1>dir
F: ADDENDUM PDF  |  CPMIO    MAC  |  CRCKLIST CRC  |  CREF80   COM
F: DSKDRV   MAC  |  DTBF     MAC  |  F80      COM  |  FCHAIN   MAC
F: FORLIB   REL  |  HELLO    COM  |  HELLO    FOR  |  HELLO    REL
F: INFO     TXT  |  INIT     MAC  |  IOINIT   MAC  |  L80      COM
F: LIB      COM  |  LPTDRV   MAC  |  LUNTB    MAC  |  M80      COM
F: MANUAL   PDF  |  TTYDRV   MAC
F1>
```

Nach jedem Kaltstart von CP/M startet das Betriebssystem im Benutzerbereich Null (0). Nähere Informationen dazu sind unter der folgenden Internetadresse zu finden.

http://www.gaby.de/cpm/manuals/archive/cpm22htm/ch1.htm#Section_1.4.6

2.4 Grundlegende Transiente Befehle

Kommen wir nun zu den *Transienten Befehlen*. Natürlich kann ich an dieser Stelle nicht alle Programme, die jemals für CP/M programmiert wurden, auflisten und dennoch gibt es einige Standard-Programme, die eigentlich immer Teil des CP/M-Systems sind. In der Regel befinden sich diese Programme im Default-Laufwerk, was nach dem Booten in der Regel das Laufwerk A: ist. Um nun diese Programme auch aus anderen Laufwerken heraus nutzen zu können, wenn man sich zum Beispiel auf Laufwerk ***C:*** befindet, ist es nicht erforderlich, diese Programme überall an die Stellen zu kopieren, an denen sie benötigt werden. CP/M schaut nach der Eingabe eines Befehls zuerst auf Laufwerk ***A:*** nach und startet es beim Auffinden von dort.

2.4.1 PIP - Dateien kopieren

Wenn es um das Kopieren von Dateien geht, dann ist das Programm ***PIP*** (*Peripheral Interchange Program*) gefordert. PIP kann auch Daten zu oder von einem der seriellen Geräte kopieren. Es kann sowohl im Einzelbefehlsmodus, als auch im interaktiven Modus arbeiten. Wenn PIP ohne weitere Parameter aufgerufen wird, geht das Programm in den interaktiven Modus über und zeigt ein Sternchen als Eingabeaufforderung an. Der interaktive Modus kann wieder verlassen, indem an der Eingabeaufforderung ohne Parameter die Eingabetaste gedrückt wird. In beiden Modi sind die Parameter von PIP jedoch gleich. Die grundlegende Syntax für eine PIP-Übertragungsanforderung lautet

destination=source[options]

Das Ziel (Destination) und die Quelle (Source) können dabei logische Geräte oder Dateinamen sein. Hier einige Beispiele für PIP-Kommandos.

Kommando	Bedeutung
PIP B:STAT.BAK=B:STAT.COM	Kopiert die Datei *STAT.COM* auf dem von Laufwerk B: in eine neue Datei namens *STAT.BAK* auf demselben Laufwerk B:
PIP B:=A:DUMP.COM	Kopiert die Datei *DUMP.COM* von Laufwerk A: in die Datei *DUMP.COM* auf Laufwerk B:

PIP C.TXT=A.TXT,B.TXT	Fügt die Dateien *A.TXT* und *B.TXT* zusammen und speichert sie unter *C.TXT* ab

Tabelle 3 - PIP-Beispiele

Wir haben gesehen, dass es in CP/M der Version 2.2 keine Unterverzeichnisse gibt und die Strukturierung der Dateien auf bestimmte User (0 bis 15) je Laufwerk verteilt wird. Wie ist es aber möglich, Dateien über User-Grenzen hinaus zu kopieren? Sehen wir uns dazu ein Beispiel an.

Ziel ist es, die Datei *color.c*, die sich auf Laufwerk ***C1:*** befindet, in das Laufwerk ***D6:*** zu kopieren. Die Ausgangssituation ist also wie folgt.

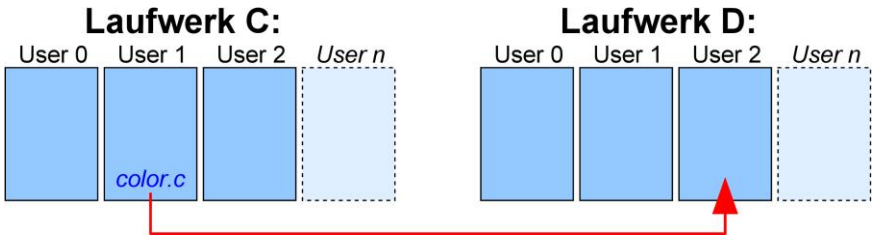


Abbildung 11 - Das Kopieren einer Datei über User-Grenzen hinweg

Wir begeben uns zunächst auf das Ziellaufwerk in den gewünschten User-Bereich.

```
A0>d:
D0>user 2
D2>
```

Jetzt wird das folgende *pip*-Kommando abgesetzt und nachfolgend mit einem *dir* nachgesehen, ob die Datei auch kopiert wurde.

```
D2>pip d:=c:color.c[G1]

RunCPM Version 6.0 (CP/M 60K)

D2>dir *.c
D: COLOR    C
D2>
```

Über den Schalter beziehungsweise Zusatz ***[G1]***, wobei die ***1*** für den User-Bereich steht, auf dem sich die Quelldatei befindet, wird diese vom Laufwerk ***C1:*** in das aktuelle Laufwerk ***D2:*** kopiert. Nachfolgend noch ein weiteres Beispiel. Wir befinden uns aktuell in Laufwerk ***G3:*** und es soll die Datei *info.txt* von Laufwerk ***C5:*** nach Laufwerk ***A1:*** kopiert werden.

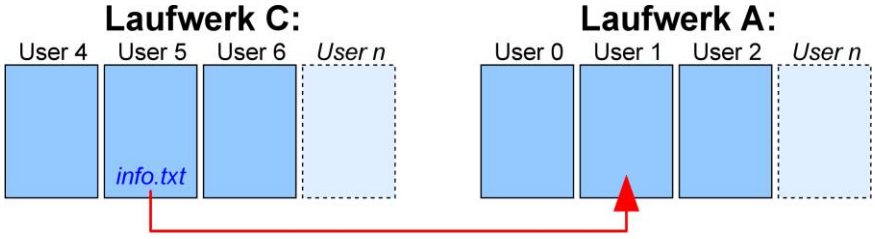


Abbildung 12 - Das Kopieren einer Datei über User-Grenzen hinweg

Mit dem nachfolgenden Befehl ist diese Kopieraktion zu bewältigen. Nachfolgend kontrollieren wir das gewünschte Ergebnis mit einem *DIR* auf Laufwerk ***A1:***

```
G3>pip a:[G1]=c:info.txt[G5]


RunCPM Version 6.0 (CP/M 60K)

G3>a:
A3>user 1
A1>dir *.txt
A: INFO     TXT
A1>
```

2.4.2 PIP - Verkettung von Textdateien

Mit dem PIP-Programm können auch mehrere Textdateien verkettet, also zusammengefügt werden. Die allgemeine Syntax dazu lautet.

pip Zieldatei = Quelldatei1, Quelldatei2, ...

Sehen wir uns das an zwei vorhandenen Textdateien ***text1.txt*** und ***text2.txt*** an, die in die Datei ***text3.txt*** zusammengefügt werden sollen.

```
A0>type text1.txt
Das ist die erste Zeile

A0>type text2.txt
Das ist die zweite Zeile

A0>pip text3.txt = text1.txt, text2.txt


RunCPM Version 6.0 (CP/M 60K)

A0>type text3.txt
Das ist die erste Zeile
Das ist die zweite Zeile

A0>
```

Nähere Informationen dazu sind unter der folgenden Internetadresse zu finden.

http://www.gaby.de/cpm/manuals/archive/cpm22htm/ch1.htm#Section_1.6.4

2.4.3 STAT - Statistics

Das Programm ***STAT*** dient zur Überprüfung oder Veränderung von Systemeigenschaften. Nachfolgend ein paar Beispiele.

- Laufwerkstatus und des freien Plattenspeicherplatzes anzeigen
- Alle Dateien eines Laufwerks mit Länge und Attributen anzeigen
- Plattenlaufwerksparameter anzeigen
- Aktive Benutzerbereiche eines Laufwerks anzeigen
- Setzen des Schreibschutzes für ein Laufwerk
- Setzen von Datei-Attributen

Die Anzeige der verfügbaren STAT-Funktionen kann über die zusätzliche Angabe von ***VAL:*** als Argument erfolgen.

```
A1>stat val:

Temp R/O Disk: d:=R/O
Set Indicator: d:filename.typ $R/O $R/W $SYS $DIR
Disk Status  : DSK: d:DSK:
User Status  : USR:
Iobyte Assign:
CON: = TTY: CRT: BAT: UC1:
RDR: = TTY: PTR: UR1: UR2:
PUN: = TTY: PTP: UP1: UP2:
LST: = TTY: CRT: LPT: UL1:
A1>
```

Wird STAT ohne Argumente abgesetzt, erfolgt die Anzeige des freien Plattenspeicherplatzes, wobei der Laufwerksstatus *R/W* ist, was für *Read/Write*, also Lesen und Schreiben steht. In der folgenden Anzeige ist lediglich das Laufwerk ***A:*** aufgeführt.

```
A0>stat
A: R/W, Space: 7464k

A0>
```

Es sind jedoch viele weitere Laufwerke am System angeschlossen. Warum werden diese nicht aufgeführt? CP/M weiß nach einem System-Neustart lediglich von einem einzigen aktiven Laufwerk ***A:***, wobei es natürlich zum Beispiel auch das Laufwerk ***C:*** gibt. Setzen wir doch einmal den DIR-Befehl auf Laufwerk ***C:*** ab und sehen uns im Anschluss die Ausgabe von STAT erneut an.

```
A0>c:
C0>stat
A: R/W, Space: 7464k
C: R/W, Space: 7464k

C0>
```

CP/M hat sich jetzt also den Inhalt von Laufwerk ***C:*** angesehen und somit ist es bekanntgemacht worden. Um einen Schreibschutzstatus zum Beispiel für Laufwerk ***F:*** einzurichten, muss dieser von R/W auf R/O (Read/Only) gesetzt werden, was über den folgenden Zusatz erfolgt.

```
A0>stat f:=R/O

A0>stat
A: R/W, Space: 7464k
F: R/O, Space: 7464k

A0>
```

Mit STAT können sich die wichtigsten Dateiparameter anzeigen lassen. Nachfolgend mache ich das für alle *COM*-Dateien.

```
F0>stat *.com

 Recs  Bytes  Ext Acc
   90    12k    1 R/W F:CONFIG.COM
   31     4k    1 R/W F:CREF80.COM
   54     8k    1 R/W F:DUP.COM
   78    12k    1 R/W F:HELLO.COM
    7     4k    1 R/W F:KEYDEF.COM
   84    12k    1 R/W F:L80.COM
   37     8k    1 R/W F:LIB80.COM
  157    20k    1 R/W F:M80.COM
   24     4k    1 R/W F:MDIR.COM
   51     8k    1 R/W F:NSWEEP.COM
   58     8k    1 R/W F:PIP.COM
   10     4k    1 R/W F:SUBMIT.COM
  193    28k    1 R/W F:ZBASIC.COM
Bytes Remaining On F: 7580k

F0>
```

Die linke Spalte ***Recs*** zeigt an, wieviel 128-Byte-Sätze die jeweilige Datei belegt. In der Spalte ***Bytes*** wird angezeigt, wieviel Platz die Datei insgesamt auf dem Datenträger belegt, wobei die Angabe in KByte (1024 Bytes) erfolgt. Die Spalte ***Ext*** gibt die Anzahl der im Inhaltsverzeichnis gespeicherten Einträge für die betreffende Datei an. Die Spalte ***Acc*** gibt an, auf Weise Zugriff auf die Datei genommen werden kann, wobei R/W der Standard ist. Der Schreibschutz ist - wie schon erwähnt - mit R/O versehen. Nähere Informationen dazu sind unter der folgenden Internetadresse zu finden.

http://www.gaby.de/cpm/manuals/archive/cpm22htm/ch1.htm#Section_1.6.1

2.4.4 ED - Ein grauenvoller Editor

Es handelt sich um den standardmäßig unter CP/M vorhandenen und einfach grausamen und - bringen wir es auf den Punkt - unbrauchbaren Texteditor, der diesen Namen im Grunde genommen nicht verdient. Deswegen gehe ich nicht auf ihn ein und übernehme auch keine Haftung, für etwaige psychischen Folgen, die bei der Nutzung auftreten können! Ich nutze später den Text-Editor über das Programm ***TE***, was um ein Vielfaches besser ist und keine bleibenden Schäden verursacht. Das ist aber nur meine persönliche Einschätzung und wer es auf die harte Tour mag, der möge sich damit auseinandersetzen! Nähere Informationen dazu sind unter der folgenden Internetadresse zu finden.

http://www.gaby.de/cpm/manuals/archive/cpm22htm/ch1.htm#Section_1.6.5

2.4.5 DUMP - Programminhalte anzeigen

Das Programm ***DUMP*** gibt den Inhalt der Datenträgerdatei in hexadezimaler Form auf der Konsole aus. Der Dateiinhalt wird in Form von sechzehn Bytes auf einmal aufgelistet, wobei die absolute Byte-Adresse links von jeder Zeile in hexadezimaler Form aufgeführt wird. Im weiteren Verlauf dieses Manuals wird auf DUMP ausführlich eingegangen und die erste Berührung hatten wir ja schon beim Type-Kommando. Nähere Informationen dazu sind unter der folgenden Internetadresse zu finden.

http://www.gaby.de/cpm/manuals/archive/cpm22htm/ch1.htm#Section_1.6.8

2.4.6 SUBMIT - Befehlsfolgen

Die meisten kennen sicherlich die Funktionalitäten von sogenannten Batch-Dateien, die die Möglichkeit bieten, verschiedene Kommandos stapelweise zu verarbeiten. Es handelt sich dabei um eine reine Textdatei, die mit auszuführenden Befehlen versehen wurde. Sehen wir uns dazu ein Beispiel im Zusammenspiel mit dem ***SUBMIT***-Befehl an, der den automatischen Ablauf von Befehlsfolgen erlaubt. Zuerst muss natürlich eine Textdatei erstellt werden, wobei ich den Editor ***TE*** nutze, auf den ich noch zu sprechen komme. Die Datei habe ich *folge.sub* genannt und besitzt folgenden Inhalt.

```
te: FOLGE.SUB                          --- | Lin:0004/0004/0512 Col:10/76 Len:09
    !...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...
  1|stat a:
  2|stat c:
  3|c:
  4|dir *.com
```

Es sollen also die gezeigten vier Befehle nacheinander abgearbeitet werden. Um die Abarbeitung zu starten, ist es nicht möglich, nach der Erstellung einfach den Namen dieser Datei in der Kommandozeile einzugeben. Es muss dazu das erwähnte *SUBMIT*-Programm mit der Angabe der erstellten Textdatei gestartet werden. Das schaut dann wie folgt aus.

```
A0>submit folge.sub


RunCPM Version 6.0 (CP/M 60K)

A0$STAT A:

Bytes Remaining On A: 7428k

A0$STAT C:

Bytes Remaining On C: 7424k

A0$C:
C0$DIR *.COM
C: $EXEC    COM  |  C-ORIG   COM  |  C309-8   COM  |  C309     COM
C: CGEN     COM  |  COLORS   COM  |  CPP      COM  |  CREF     COM
C: DEBUG    COM  |  DEHUFF   COM  |  FRACTAL  COM  |  HELLO    COM
C: LIBR     COM  |  LINQ     COM  |  OBJTOHEX COM  |  OPTIM    COM
C: OPTIMH   COM  |  P1       COM  |  ROVECTOR COM  |  SCAN     COM
C: SYMTOAS  COM  |  ZAS      COM
C0>
```

Innerhalb von SUBMIT-Dateien können bis zu drei Variablen verwendet werden, die sich $1, $2 und $3 nennen. Sie stehen als Platzhalten und können beim Aufruf als Argument angehängt werden. Nachfolgend ist ein sehr einfaches Beispiel zu sehen, das es erlaubt, aus den Verzeichnissen *A:* und *C:* Dateien mit bestimmten Dateinamenerweiterungen anzeigen zu lassen. Abschließend erfolgt ein Rücksprung zu Laufwerk *A:*.

```
te: FOLGE2.SUB                         --- | Lin:0005/0005/0512 Col:03/76 Len:02
    !...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...
  1|a:
  2|dir *.$1
  3|c:
  4|dir *.$1
  5|a:
```

Es ist zu sehen, dass ich nach der Dateinamenerweiterung ***h*** über das *DIR*-Kommando filtere. Auf Laufwerk *A:* sind keine Dateien zu finden, wohin auf Laufwerk *C:* die angezeigten vorhanden sind.

```
A0>submit folge2.sub h


RunCPM Version 6.0 (CP/M 60K)

A0$A:
A0$DIR *.H
No file
A0$C:
C0$DIR *.H
C: ASSERT   H    |  CONIO    H    |  CPM      H    |  CTYPE    H
C: EXEC     H    |  FLOAT    H    |  HITECH   H    |  LIMITS   H
C: MATH     H    |  OVERLAY  H    |  SETJMP   H    |  SIGNAL   H
C: STAT     H    |  STDARG   H    |  STDDEF   H    |  STDINT   H
C: STDIO    H    |  STDLIB   H    |  STRING   H    |  SYS      H
C: TIME     H    |  UNIXIO   H
C0$A:
A0>
```

Nähere Informationen dazu sind unter der folgenden Internetadresse zu finden.

http://www.gaby.de/cpm/manuals/archive/cpm22htm/ch1.htm#Section_1.6.7

Weitere transiente Befehle folgen in Kürze, wenn die erforderlichen Grundlagen dafür geschaffen wurden.

2.5 Power - Das Schweizer Taschenmesser

Ich möchte auf ein sehr nützliches Programm hinweisen, das als Schweizer Taschenmesser für CP/M angesehen werden kann. Es nennt sich ***POWER*** und wurde von *Pavel Breder* programmiert. Man kann mit Recht behaupten, was der *Norton Commander* für das Betriebssystem *DOS* ist, stellt *POWER* für CP/M dar. Es ermöglicht es auf einfache Weise, Dateien nach Nummern zu manipulieren, Speicherinhalte zu ändern, fehlerhafte Medien zu testen und zu reparieren, Programme zu debuggen und besitzt viele andere hilfreiche Funktionen, die allesamt in einem einzigen Programm angeboten werden. Es ist zum Beispiel unter den folgenden Internetadresse zu finden.

http://www.z80.eu/power.html

Nach dem Start meldet sich das Programm mit dem Prompt in Form eines Gleichheitszeichens hinter der Laufwerksbezeichnung.

```
A0>power


        POWER 3.08 on CP/M 2.2  Z

Copyright (c) 1981, 82, 83, 84 by PAVEL BREDER
        All rights reserved. 84-05-19

-from-COMPUTING!-SAN-FRANCISCO-CA--94123-USA--

A0=
```

Nähere Informationen zur Handhabung sind über den folgenden Link zu bekommen.

http://www.robotron-net.de/Service/Dokus/power.pdf

Nachfolgend ein kleines Beispiel zur Anzeige von Speicherinhalten ab einer angegebenen Adresse. Der Befehl ***DS 100*** zeigt den Inhalt ab Adresse 100 an,

wobei die Anzeige in <**H**> hexadezimal, <**D**> dezimal, <**B**> binär oder <**A**> ASCII erfolgt.

```
A0=ds 100
addr:Hex Dec  Binary Ascii Enter
0100: C3 195 11000011 - C  <H>
0101: 10  16 00010000  ^P  <H>
0102: 03   3 00000011  ^C  <H>
0103: 00   0 00000000  ^@  <H>
0104: 02   2 00000010  ^B  <H>
0105: 20  32 00100000      <H>
```

2.6 Unterschiedliche Dateitypen

In einem Computersystem gibt es die unterschiedlichsten Dateien. Allgemein können wir sagen, dass eine Datei eine Ansammlung von Daten ist, die auf einem Datenträger gespeichert ist. Jede dieser Dateien muss einen eindeutigen Namen besitzen, denn diese werden in einem Inhaltsverzeichnis (Directory) je Laufwerk organisiert. Auf dem Laufwerk ***A:*** können also nicht zwei Dateien mit gleichem Namen abgelegt sein. Es gibt eine Vielzahl von verschiedenen Datei-Typen, die sehr unterschiedliche Informationen speichern können. Schaut man sich jedoch eine Datei auf Bit-Ebene an, kann man dieser Datei nicht ansehen, für welchen Zweck sie angelegt wurde und um welchen Inhalt es sich handelt. Es kommt immer auf den Kontext an, für den sie genutzt wird. Hat man eine ausführbare Datei mit der Dateiendung ***.com***, dann wird diese nach dem Aufruf ausgeführt, wohingegen eine Testdatei zum Beispiel mit der Dateiendung ***.txt*** nicht ausführbar ist und kein Programm im klassischen Sinn darstellt. Eine derartige Datei kann nur über einen Text-Editor oder ein Text-Verarbeitungsprogramm bearbeitet werden und macht nur in diesem Kontext einen Sinn. Die folgende Tabelle beinhaltet die gängigsten Datei-Endungen, wobei diese nicht vollständig ist, denn diese können Hersteller spezifisch abweichen.

Datei-Endung	Bedeutung
com	Befehls- bzw. Programmdatei
asm	Assembler-Datei (Quelltext)
bak	Backup-Datei (Sicherungsdatei)
sub	Stapelverarbeitungs-Datei
bas	Basic-Programm
bin	Binär-Datei unter Basic
prn	Datei für Listenausdruck assemblierter Maschinenprogramme
hex	Datei im Intel-HEX-Format
$$$	Temporäre Datei

Tabelle 4 - Unterschiedliche Dateiendungen

2.7 Der Text-Editor TE

Ich möchte auf den Umstand zurückkommen, den ich beim Standard-Text-Editor ED von CP/M genannt hatte. Dieser ist nicht zu gebrauchen und aus diesem Grund werde ich ab jetzt nur noch den Text-Editor ***TE*** nutzen, der sich bei RunCPM auf Laufwerk ***A:*** befindet, was jedoch kein Problem darstellt, da CP/M zuerst immer auf Laufwerk *A:* nachschaut, ob sich das aufgerufene Programm dort befindet. Nach der Eingabe von ***te*** öffnet sich also der Text-Editor, der sich wie folgt gestaltet.

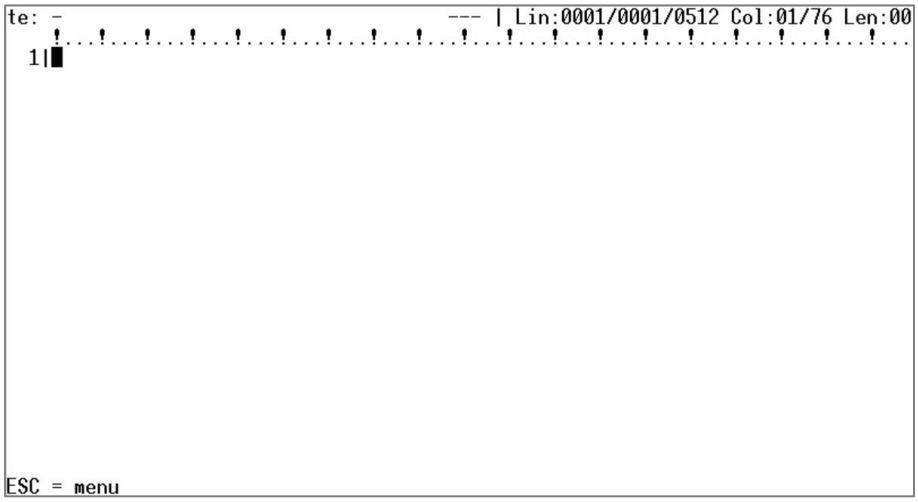


Abbildung 13 - Der Text-Editor TE

Es wird die momentan einzige vorhandene Zeilennummer 1 mit dem Cursor angezeigt. Nun kann der gewünschte Text oder Quellcode eingegeben werden, doch ich muss zuvor noch ein paar Dinge erklären. Unter den vermeintlich modernen Betriebssystemen gibt es natürlich die Pfeiltasten, die es einem ermöglicht, den Cursor *rauf/runter* beziehungsweise *links/rechts* zu positionieren. Die ersten CP/M-Rechner besaßen jedoch keine derartigen Tasten, sodass es mit dem Navigieren hier nun etwas anders ausschaut. Es wurden dazu die Tasten ***E*** und ***X*** beziehungsweise ***S*** und ***D*** genutzt, deren Positionen beziehungsweise Anordnung quasi dem Navigationskreuz der Pfeiltasten entspricht. Zusätzlich muss natürlich die Steuerungstaste ***Strg*** gedrückt werden, um die Eingabe von der normalen Texteingabe zu unterscheiden.

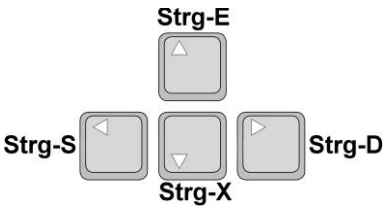


Abbildung 14 - Die Navigation im Text-Editor TE

Ich würde zur Navigation vorschlagen, dennoch die Pfeiltasten einmal auszuprobieren, denn es besteht Hoffnung, dass diese funktionieren, was nicht immer der Fall ist. Auf dem *Arduino Due* hat es bei mir nicht funktioniert, wogegen es beim Teensy keine Probleme gab.

Für das Löschen eines Zeichens - je nachdem, wo sich der Cursor gerade befindet - können sowohl die ***Entfernen***- als auch die ***Backspace***-Taste genutzt werden. Um eine Datei zu speichern muss das Menü über die ***ESC***-Taste aufgerufen und später ***Save*** gewählt werden. Die gezeigten Großbuchstaben zeigen das jeweilige Tastenkürzel an. Dort sind auch andere wichtige Optionen zu finden.

Abbildung 15 - Die Optionen des Text-Editors TE

Alle wichtigen Menü-Befehle sind unter der Option ***Help*** zu finden.

2.8 Der Umgang mit Bits und Bytes

Da wir jetzt gesehen haben, wie es über den Text-Editor TE sehr einfach ist, Text einzugeben, wollen wir uns den Inhalt einer derartigen Datei mit geeigneten Werkzeugen näher ansehen. Beim *Type*-Kommando haben wir schon gesehen, wie sich der Inhalt einer Text-Datei anzeigen lässt und das sogar mithilfe des *Dump*-Kommandos runter bis auf Byte-Ebene. Werfen wir einen Blick darauf. Ich möchte die beiden Textzeilen

- CP/M macht
- grossen Spass

in einer Textdatei speichern. Nach dem Aufruf des Text-Editors in einem beliebigen Verzeichnis muss zunächst der oberste Menüpunkt ***New*** gewählt werden, in dem die Taste ***N*** gedrückt wird. Bei jedem menüpunkt ist ein Buchstabe großgeschrieben, der für den Aufruf des jeweiligen Punktes steht.

Abbildung 16 - Eine Neue Datei anlegen

Nach der Eingabe des Textes schaut es wie folgt im Editor aus.

Um den Text in einer Datei zu speichern, wird zuerst die ***ESC***-Taste gedrückt, um darüber erneut das Menü zur Anzeige zu bringen. Um die Speicherung zu ermöglichen, muss die Taste ***A*** gedrückt werden, die den Menüpunkt ***save As*** aufruft.

Abbildung 17 - Die Datei unter einem neuen Namen speichern

Im Anschluss wird im unteren Bereich des Text-Editors folgendes eingeblendet, was uns die Möglichkeit gibt, dort den gewünschten Dateinamen inklusive Datei-Erweiterung einzugeben und die Eingabe mit der ***RETURN***-Taste zu bestätigen.

```
Filename (ESC = cancel): cpm.txt
```

Im Anschluss muss über das Drücken der ***ESC***-Taste erneut das Menü aufgerufen werden, um dort über die Taste ***X*** den untersten Punkt ***eXit te*** zum Verlassen des Text-Editors auszuwählen.

Abbildung 18 - Den Text-Editor verlassen

Danach ist das entsprechende Prompt des Laufwerks zu sehen und wir sehen nach, ob die Datei dort zu finden ist. Zuerst sehen wir mit dem ***DIR***-Kommando mit dem Filter ****.txt*** nach und führen dann das ***TYPE***-Kommando zur Anzeige des Inhaltes aus.

```
F0>dir *.txt
F: INFO     TXT  |  CPM      TXT
F0>type cpm.txt
CP/M macht
grossen Spass

F0>
```

Abbildung 19 - Das Anzeigen des Datei-Inhaltes mittels TYPE

Es ist zu erkennen, dass die Datei angelegt wurde und den eben vergebenen Inhalt widerspiegelt. Das hat also funktioniert. Im nächsten Schritt nutzen wir das ***DUMP***-Kommando zur Anzeige der Informationen hinsichtlich des ASCII-Codes. Beim *ASCII*-Code handelt es sich um den ***American Standard Code for Information Interchange*** mit einer 7-Bit-Zeichenkodierung. Die Ausgabe schaut wie folgt aus, wobei ich den für uns relevanten Teil in den betreffenden beiden Zeilen markiert habe.

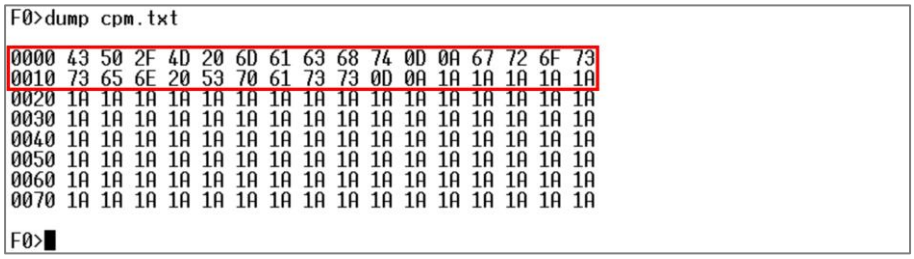


Abbildung 20 - Das Anzeigen des Datei-Inhaltes mittels DUMP

Beim Aufruf von DUMP ist die Anzeige in zwei Blöcke gegliedert und es werden die zweistelligen Werte, die je ein einzelnes Byte repräsentieren zu je 16 Stück in einer separaten Zeile zur Anzeige gebracht und am linken Rand durchgezählt. Die Zahl *0000* der ersten Zeile gibt den ersten Wert wieder, die Zahl *0010* der zweiten Zeile den Wert 16, die Zahl *0020* der dritten Zeile den Wert 32 usw.

Werfen wir nun einen Blick auf die Werte, die im hexadezimalen Format angezeigt werden. Ich übertrage sie in eine Tabelle mit entsprechenden Erläuterungen.

Hex-Wert	Zeichen	Hex-Wert	Zeichen
43	C	6F	o
50	P	73	s
2F	/	73	s
4D	M	65	e
20	SP	6E	n
6D	m	20	SP
61	a	53	S
63	c	70	p
68	h	61	a
74	t	73	s
0D	CR	73	s
0A	LF	0D	CR
67	g	0A	LF
72	r		

Tabelle 5 - ASCII-Werte mit Zeichen

Wenn man sich die Werte von oben nach unten, die in je zwei Spalten zu lesen sind, anschaut, dann ist der eben eingegebene Text zu erkennen. Neben den druckbaren Zeichen sind zudem Steuerzeichen zu sehen. ***SP*** bedeutet ***Space*** und steht für ein Leerzeichen und ***CR*** für ***Carriage Return*** (Wagenrücklauf) beziehungsweise ***LF*** für ***Linefeed*** (Zeilenvorschub). Unter CP/M wird also jedes Zeilenende durch das ASCII-Paar *CR* und *LF* - kurz: *CRLF* - gekennzeichnet. Warum werden aber die restlichen Zeichen hinter dem letzten 0A zur Anzeige gebracht? Diese Werte ***1A*** sind ja nicht Bestandteil des eingegebenen Textes!? Einen Umstand, den ich noch nicht zur Sprache gebracht habe, ist die Tatsache, dass CP/M die zu speichernden Daten in 128-Byte-Einheiten organisiert, die ***Records*** genannt werden.

Ich komme nun zu einem weiteren Programm, das sich ***DDT*** nennt. Die Informationen, die DUMP zutage fördert, sind zwar schön und gut und zeigen

lediglich die rohen Daten in Form der vorhandenen Bytes. Für eine genauere Analyse von Text-Dateien oder auch Programmen ist dies jedoch zu wenig. Aus diesem Grund gibt es das DDT-Programm.

2.8.1 DDT - Dynamic Debugging Tool

Die drei Buchstaben DDT stehen für ***Dynamic Debugging Tool*** und es handelt sich bei diesem Programm um ein Werkzeug, was zur Fehlersuche in Programmen genutzt werden kann. Mit seiner Hilfe ist es möglich, den Inhalt einer Datei in hexadezimaler Form inklusive der ASCII-Bedeutung anzeigen zu lassen. Wir machen das einmal mit der gerade erstellen Textdatei *cpm.txt*.

```
F0>ddt cpm.txt
DDT VERS 1.4
NEXT  PC
0180 0100
-d
0100 43 50 2F 4D 20 6D 61 63 68 74 0D 0A 67 72 6F 73 CP/M macht..gros
0110 73 65 6E 20 53 70 61 73 73 0D 0A 1A 1A 1A 1A 1A sen Spass.......
0120 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A ................
0130 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A ................
0140 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A ................
0150 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A ................
0160 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A ................
0170 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A ................
0180 1A 84 12 13 C3 69 01 D1 2E 00 E9 2A 7C 1D EB 0E .....i.....*|...
0190 1A CD 67 1B C9 3E 0C D3 01 3E 08 D3 01 DB 01 07 ..g..>...>......
01A0 07 07 1F DA A9 08 C3 9D 08 DB 03 E6 7F C9 21 83 ..............!.
01B0 1D 70 2B 71 2A 82 1D 44 4D CD A1 07 0E 3A CD 86 .p+q*..DM....:..
-
```

Abbildung 21 - Das DDT-Programm

Man kann das Programm DDT direkt mit dem Namen der zu untersuchenden Datei als Argument angeben und aufrufen und dann im Anschluss hinter dem DDT-Prompt (-) den Buchstaben ***D*** (D=DUMP) eingeben. Somit ist die gerade gezeigte Ausgabe zu sehen. Es ist zu sehen, dass hier viel mehr Informationen geliefert werden, wobei sich die Anzeige in drei Blöcke gliedert. Auf der linken Seite in der ersten Spalte werden die Adressen aufgelistet und enthalten die Informationen, an welcher Stelle sich die Datei im Speicher befindet. In der Mitte sind die Werte in hexadezimaler Schreibweise zu sehen, die man schon von DUMP kennt und auf der rechten Seite in der letzten Spalte ist die entsprechende Auflistung der ASCII-Entsprechungen zu sehen. Man mag sich vielleicht fragen, warum dort so viele Punkte zu sehen sind. Diese zeigen an, dass es sich dabei nicht um druckbare ASCII-Zeichen handelt und nicht dem ASCII-Code zuzuordnen sind.

Hinsichtlich des Unterschiedes zwischen *DUMP* und *DDT* kann festgehalten werden, dass DUMP lediglich den Dateiinhalt auflistet und eine Aussage darüber macht, wie die Daten auf dem Datenträger vorliegen. Im Gegensatz dazu liefert DDT die Informationen darüber, wie der Dateiinhalt an welcher Stelle in den Speicher geladen wurde, was bei einem Standard-CP/M immer die Hex-Adresse ***0100h*** ist.

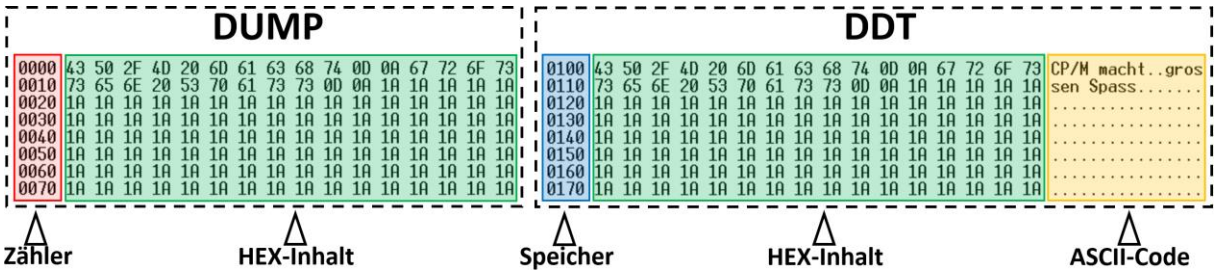


Abbildung 22 - Die Gegenüberstellung von DUMP und DDT

Bei DUMP ist zu sehen, dass es sich in der ersten Spalte lediglich um eine Auflistung beziehungsweise Durchnummerierung der einzelnen Bytes beginnend bei ***0000h*** handelt und bei DDT eben um die genannte Speicheradresse ***0100h***. Um DDT zu verlassen, muss entweder die Tastenkombination ***Strg-C*** gedrückt werden oder die Eingabe ***g0*** hinter dem Prompt mit der ***RETURN***-Taste bestätigt werden. Mit DDT lassen sich nicht nur Daten anzeigen, sondern auch manipulieren, was jedoch für uns im Moment nicht weiter von Bedeutung ist. Nähere Informationen dazu sind unter der folgenden Internetadresse zu finden.

http://www.gaby.de/cpm/manuals/archive/cpm22htm/ch4.htm

2.8.2 Eine kleine Einführung in verschiedene Zahlensysteme

Nun geht es wirklich an's Eingemachte, denn wir beginnen mit der Programmierung von Z80-Maschinensprache. Nun, ganz so einfach ist das nicht und doch möchte ich einen geeigneten Einstieg vorstellen. Was ist überhaupt Maschinensprache und wie schaut sie aus? Eine Maschinensprache, die genau genommen eigentlich keine Programmiersprache. Es handelt sich um einen Maschinencode, die ein Prozessor direkt ausführen kann. Wenn das so ist, dann muss es sich ja lediglich um Einsen und Nullen handeln, die eine Maschinensprache ausmacht. Ja und Nein! Wenn zum Beispiel die folgende Bitfolge vorliegt, dann handelt es sich beim Z80 schon um ein kleines Programm. Wir werden gleich im Detail sehen, was dieser Code so bewirkt.

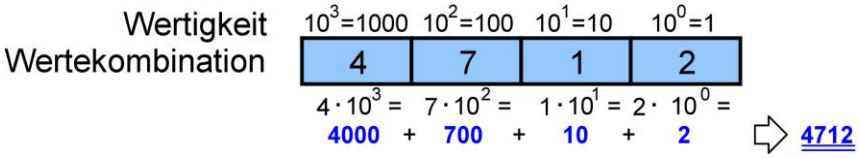
00111110 00000100

00111100

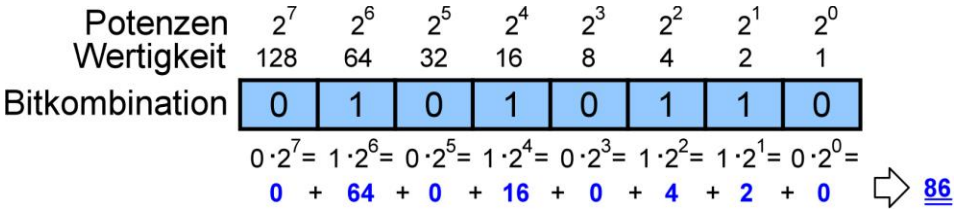
11001001

Nun ist es aber recht mühsam, wenn man sich als Programmierer mit diesen Einsen und Nullen herumschlagen muss, denn irgendwie ist das genauso aussagekräftig, wie ägyptische Hieroglyphen. Jedenfalls für einen Nicht-Sprachwissenschaftler. Doch fangen wir von vorne an!

Unser bekanntes Dezimalsystem ist wie folgt aufgebaut, wobei jede einzelne Stelle die uns bekannten Wertigkeiten - von rechts nach links - Einer, Zehner, Hunderter, Tausender, usw. besitzt. Es arbeitet also in einem Stellenwertsystem zur Basis 10.



Das Ergebnis ist für uns Normalsterbliche natürlich sofort ablesbar, da wird dieses Zahlensystem in Form des Dezimalsystems täglich im Gebrauch haben. Eine Zahl hingegen, die lediglich aus den genannten Einsen und Nullen besteht, wird *Binärzahl* genannt. Hier schauen die Stellenwertigkeiten ein wenig anders aus und arbeitet in einem Stellenwertsystem zur Basis 2.

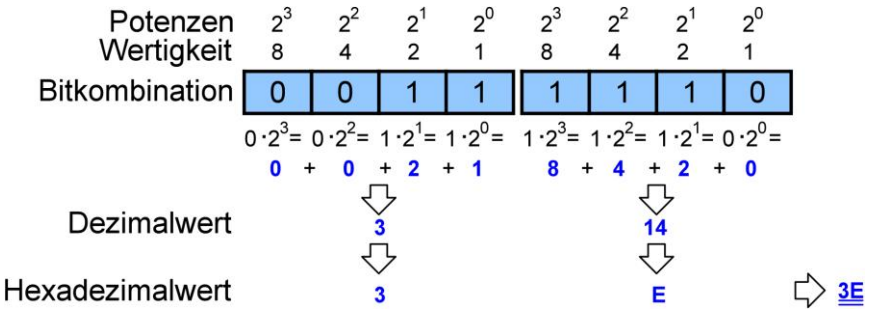


Um diese Bytefolgen, die ja aus vier 8-Bit-Gruppen besteht, etwas übersichtlicher zu gestalten, hat man die hexadezimalen Zahlen ins Leben gerufen. Diese beziehen sich immer auf 4-Bits - auch *Nibble* genannt. Im Hexadezimalsystem werden die einzelnen Zahlen in einem Stellenwertsystem zur Basis 16 abgebildet. Es gibt also statt 10 Ziffern (0 bis 9) des Dezimalsystems im Hexadezimalsystem 16 unterschiedliche Ziffern. Nun, das stimmt wiederum nicht ganz, denn es handelt sich dabei nicht nur um Ziffern, die ja auf 10 unterschiedliche Werte begrenzt sind. Es fehlen also noch 6 weitere und man hat sich dabei den Buchstaben *A* bis *F* bedient. In der folgenden Tabelle sind die dezimalen Werte den Hexadezimalen und Binärzahlen gegenübergestellt.

Dezimalzahl	Hexadezimalzahl	Binärzahl
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Tabelle 6 - Verschiedene Zahlensysteme

Hinsichtlich der ersten Bitkombination schaut das dann wie folgt aus.



Um nun die eben gezeigten Bitfolgen in Gänze in hexadezimale Werte (kurz: HEX-Werte) zu wandeln, gestaltet sich das Ergebnis wie folgt.

3	E	0	4
0011	1110	0000	0100

3	C
0011	1100

C	9
1100	1001

Der Maschinencode würde dementsprechend wie folgt lauten.

3E 04 3C C9

Ok, das ist schon ein wenig besser zu lesen, obwohl es immer noch einiges an Vorstellungskraft abverlangt. Auf diese Weise zu Programmieren ist sicherlich keine Freude, denn er handelt sich ja eigentlich um Befehle für die Z80-CPU, die sich dahinter verbergen. Gibt es keinen einfacheren Weg, der den Code irgendwie sprechender gestaltet? Den gibt es! Eine kleine Stufe über der Maschinensprache ist die sogenannte *Assemblersprache* angesiedelt. Die Assemblersprache - kurz auch *Assembler* genannt -, ist eine Programmiersprache, die auf den bestimmten Befehlsvorrat einer bestimmten CPU ausgerichtet ist. Hier natürlich die Z80-CPU. Diese Sprache bedeutet auf jeden Fall einen Fortschritt im Gegensatz zur Maschinensprache. Assembler ist leichter zu lesen und zu verstehen. Sehen wir uns doch einmal die ersten beiden Hexadezimalzahlen genauer an, denn sie gehören zusammen.

3E 04

Sie *sagen* der Z80-CPU, dass der Wert 04 in eine interne Speicherstelle geladen werden soll. Der Z80 besitzt einige unterschiedliche interne Speicherstellen, wobei eine spezielle Speicherstelle eine ganz besondere Stellung einnimmt. Es handelt sich um den sogenannten *Akkumulator* mit der Abkürzung *A*. In diesem Register, so werden die internen Speicherstellen auch genannt, werden alle Rechenoperationen durchgeführt. In Prosa ausgedrückt, würde die Anweisung wie folgt lauten.

„Lade den Wert 4 in den Akkumulator!"

Kommen wir noch einmal zum Maschinencode zurück, denn da haben die HEX-Werte die folgenden Bedeutungen. Der erste HEX-Wert *3E* ist der sogenannte OP-Code

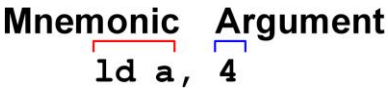
(Abkürzung für Operation-Code), der über die angegebene Zahl einen Maschinenbefehl repräsentiert, wobei die Summe aller OP-Codes den Befehlssatz des entsprechenden Prozessors bilden.

OP-Code	Befehl
3E	ld a

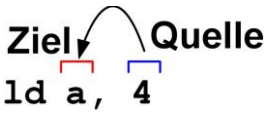
Der nachfolgende Wert *04* stellt das Argument für diesen Befehl dar. In Assembler lautet das dann wie folgt.

ld a, 4

Die beiden Buchstaben *ld* sind die Abkürzung für das englische Wort *Load*, was übersetzt „laden" bedeutet. In Assembler wird jede Anweisung an den Computer durch ein sogenanntes mnemonisches Symbol dargestellt, was ich gerade etwas lapidar *Befehl* genannt habe. Diese *Mnemonics* sind sehr kurze Wörter, die so ähnlich klingen, wie di zu repräsentierende Anweisung. Sollten wir uns diese Befehlszeile ein wenig genauer anschauen. Da befindet sich auf der linken Seite das Mnemonic und auf der rechten Seite das Argument.



Nun ist es in beim Assembler für den Z80 immer so, dass sich Ziel auf der linken und die Quelle auf der rechten Seite befindet. Das schaut dann wie folgt aus.



Die nächste Anweisung lautet binär

00111100

In der hexadezimalen Schreibweise wäre das der Maschinencode *3C*. Ok, keinen Schimmer, was das nun wieder zu bedeuten hat! Einen Blick in die Liste der OP-Codes für den Z80 zeigt, dass dieser Wert für *inc a* steht. Es handelt sich natürlich wieder um einen Mnemonic, wobei die Abkürzung *inc* für *Increment* steht, was übersetzt „Erhöhung" bedeutet. Der Inhalt des Akkumulators soll also um den Wert 1 erhöht werden. Im Assembler wird das wie folgt geschrieben.

inc a

Die letzte Anweisung lautet binär

11001001

Das wird in HEX mit *C9* kodiert. Ein Blick in die Liste der OP-Codes zeigt, dass dies *ret* bedeutet und die Abkürzung für *return* ist. Return bedeutet übersetzt „zurück". Im Assembler wird das wie folgt geschrieben.

ret

Eine Liste aller OP-Codes ist unter der folgenden Internetadresse zu finden.

http://map.grauw.nl/resources/z80instr.php

Ich zeige an dieser Stelle das Ergebnis noch einmal in einer Tabelle, damit es etwas übersichtlich erscheint.

Binärkombination(en)	HEX-Wert	Mnemonic
00111110 00000100	3E 04	ld a, 4
00111100	3C	inc a
11001001	C9	ret

Somit wäre das kleine Programm fertig. Doch wir haben das Pferd von hinten aufgezäumt und das hatte natürlich seinen Sinn. Ich wollte den Weg von den Einsen und Nullen über die HEX-Zahlen hin zu den Mnemonics des Assemblers aufzeigen. Ein Programmierer beschreitet genau den anderen Weg. Er programmiert in Assembler mithilfe der Mnemonics und als Endergebnis fällt quasi hinten der Maschinencode raus. Genau diesen Weg werden wir jetzt gehen. Wenn es um die Programmierung des Z80-Prozessors geht, sollte man sich aufgrund der Fülle an Informationen auf keinen Fall verrückt machen lassen. Das Ganze wirkt - wie fast alles in der Elektronik oder Programmierung - auf den ersten Blick erschlagend. Doch keine Sorge, denn es ist wirklich ein normaler Vorgang, wie alles im Leben. Klein anfangen und einfach dranbleiben. Man muss nicht zu Beginn alle Z80-Befehle kennen, um von heute auf morgen ein Spezialist zu sein. Das Standardwerk zur Z80-Programmierung ist in meine Augen von *Rodnay Zaks* verfasst worden und wer ein bisschen nach Literatur im Internet sucht, der wird schnell dieses Buch in Form einer PDF-Datei finden. Der Titel lautet *Programmierung des Z80* mit einem Umfang von 640 Seiten. Da kann einem schon Angst und Bange werden - Quatsch!

2.9 Die Programmierung über den Assembler

Die Programmierung in Assembler kann über zwei unterschiedliche Ansätze erfolgen, die ich beide vorstellen möchte. Zum einen - und damit werde ich beginnen - kann die Programmierung mit Bordmitteln, wie man das so schön sagt, erfolgen, also mit Programmen und Tools, die direkt für CP/M entwickelt wurden. Zum anderen gibt es noch die Möglichkeit, auf einem PC die Entwicklung zu starten und dann von dort aus mithilfe eines sogenannten *Cross-Assemblers* das Maschinenprogramm zu generieren. Bei einem Cross-Assembler handelt es sich um ist eine Spezialform eines Assemblers, der auf einer speziellen Computerplattform - also zum Beispiel auf einem PC - läuft und den Maschinencode für eine andere Computerplattform - dem RunCPM - generiert. Auch diesen Ansatz werde ich verfolgen. Zuerst sollten wir jedoch einen Blick auf die internen Speicherstellen des Z80 werfen, die ja bekanntermaßen *Register* genannt werden.

2.9.1 Die Z80-Register

Auf der folgenden Abbildung sind die Z80-Register zu sehen. Es handelt sich um interne Speicherstellen, die sehr schnell zu adressieren sind.

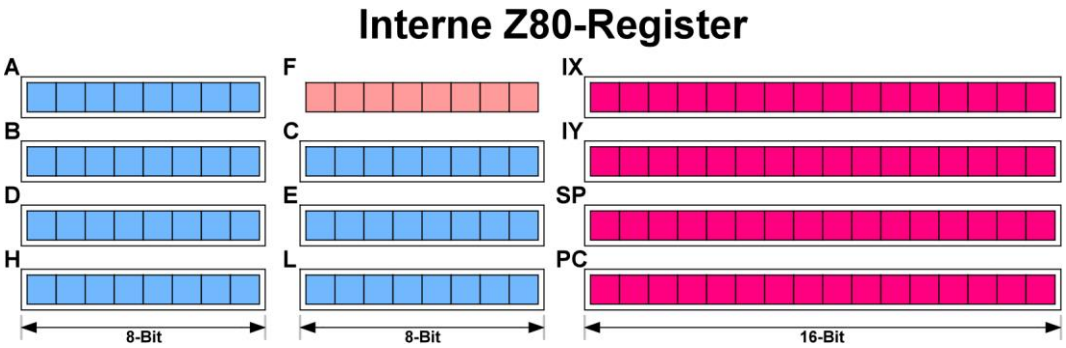


Abbildung 23 - Die Z80-Register

Diese Register haben unterschiedliche Datenbreiten, die 8 und 16 Bit breit sein können. Ein besonderes Register ist das *A*-Register, was für den *Akkumulator* steht, in dem alle Rechenoperationen durchgeführt werden und eine Breite von 8 Bits einnimmt. Das *PC*-Register, das die Abkürzung für *Program-Counter* ist, zeigt immer auf den nächsten Befehl, den die CPU abzuarbeiten hat und besitzt eine Datenbreite von 16-Bits. Die Register können zum Teil paarweise zu sogenannten Doppelregistern zusammengefasst werden.

- AF (A und F)
- BC (B und C)
- DE (D und E)
- HL (H und L)

Dann gibt es noch die sogenannten Flags - hier mit *F* abgekürzt - in dem die Ergebnisse von verschiedenen Rechenoperationen in Form von Status-Bits abgebildet sind. Wir kommen natürlich noch im Detail darauf zu sprechen.

2.9.2 Der externe Speicher – ROM und RAM

Natürlich können nicht nur in den internen Registern des Z80 Daten gespeichert werden, es gibt ja auch noch den externen Speicher in Form der ROM und des RAM. Aber Vorsicht, denn nur das RAM kann Daten speichern. Das ROM besitzt bekannter Weise auch Daten, die jedoch nicht über den Z80 aktualisiert werden können.

2.10 Die Z80-Bordmittel verwenden

Bevor wir jedoch über den Assembler mithilfe der Mnemonics ein Programm erstellen, sind noch einige Details zu erwähnen. Das ursprüngliche CP/M war für den Prozessor 8080 entwickelt worden. Da wir es nun jedoch mit einem Z80 zu tun haben, schaut die Sache - trotz der Abwärtskompatibilität vom Z80 zum 8080 - etwas anders aus. Der ursprüngliche Assembler *asm* für den 8080, besitzt andere Mnemonics als der Z80, was dazu führt, dass dieser Assembler für unsere Belange nicht genutzt werden kann, es sei denn, es werden die für den 8080 erforderlichen Mnemonics verwendet. Das möchte ich jedoch nicht, denn es geht hier um den Z80! Die Mnemonics für den Z80 wurden im Hinblick auf den 8080 vereinfacht. Es besteht jedoch keine Änderung der Technik des Z80 zum 8080, denn nur die „Befehle" wurden vereinfacht! Diese generieren jedoch hinsichtlich der Funktionalität den gleichen Maschinencode wie die 8080-

Assembler-Mnemonics mithilfe der „alten" Befehle. Kein Grund zur Sorge also. Es ist lediglich notwendig, einen anderen Assembler zur Programmierung zu verwenden. Alle erforderlichen Programme sind unter der folgenden Internetadresse zu finden. Notfalls auch auf meiner Internetseite.

https://github.com/MiguelVis/zsm

Doch bevor es losgeht, sollte ich einige Details besprechen. Ein ausführbares Programm unter dem Betriebssystem CP/M besitzt laut Konvention die Dateiendung *.com*. Na, hat sich da das Betriebssystem *DOS* etwas abgeschaut?! Ein Schelm, wer wieder Böses denkt! Es besteht also die Aufgabe, mithilfe eines Assemblers eine ausführbare Datei zu generieren, die die Dateiendung *.com* besitzt. Das geht jedoch nicht, denn ein Assembler generiert immer nur eine sogenannte *Intel-HEX-Datei* mit der Dateiendung *.hex*. Diese Datei ist quasi ein Zwischenprodukt auf dem Weg zur eigentlichen COM-Datei, die ja von uns beabsichtigt ist und die letztendlich einfach ausgeführt werden kann, was man von der Intel-HEX-Datei nicht behaupten kann. Das Intel HEX-Format wird zur Speicherung und Übertragung von binären Daten verwendet und wird auch heute noch dazu verwendet, um Programmierdaten für Mikrocontroller beziehungsweise Mikroprozessoren zu speichern. Um nun eine Datei im Intel-HEX-Format in eine COM-Datei zu wandeln, wird ein kleines Tool - also Hilfsprogramm - benötigt, das für den 8080 ***load*** heißt. Doch sehen wir uns das in einem Arbeitsablauf - dem sogenannten *Workflow* - genauer an, der von links nach rechts abgearbeitet wird. Ganz links ist der Text-Editor ***te*** zu sehen, mit dessen Hilfe der Quellcode in Form einer einfachen Textdatei eingegeben wird. Die erstellte Text-Datei muss die Endung ***.asm*** vorweisen, denn nur eine derartige Datei wird vom Assembler *asm* erkannt und beim Aufruf von ***asm*** darf die Dateierweiterung nicht mit angegeben werden. Das Ergebnis des Assembleraufrufes sind zwei Dateien, die die Endungen *.hex* und *.prn* besitzen. Die Datei .hex beinhaltet den schon erwähnten HEX-Code, der im nächsten Schritt mithilfe des Programms *load* dazu verwendet wird, eine ausführbare COM-Datei im Binärformat zu generieren. Die Datei mit der Endung *.prn* besitzt das ursprüngliche Quellprogramm, jedoch ergänzt um zusätzliche Assembler-Informationen in den äußersten 16 Spalten, die zum Beispiel die Programmadressen und den hexadezimalen Maschinencode enthalten. Diese Datei kann als Backup für die originale Quelldatei dienen. Wenn die Quelldatei versehentlich entfernt oder zerstört wird, kann die PRN-Datei durch Entfernen der äußersten linken 16 Zeichen jeder Zeile bearbeitet werden.

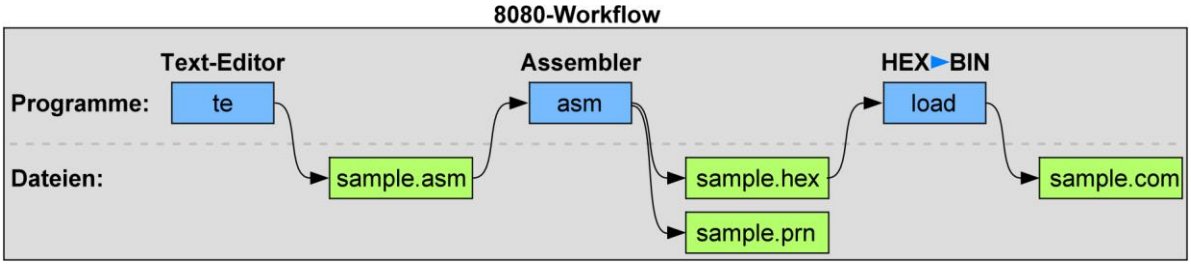


Abbildung 24 - Der Assembler-Workflow für den 8080

Der Assembler *asm* verarbeitet alle mnemonischen 8080-Befehle. Doch Stopp! Wir wollen ja nicht für den 8080, sondern für den Z80 programmieren. Zu diesem Zweck müssen ein paar zusätzliche Programme installiert beziehungsweise einfach auf das entsprechende Laufwerk kopiert werden, die unter der gerade

genannten Internetadresse zu finden sind. Es handelt sich dabei um die Programme

- ZSM.COM
- HEXTOCOM.COM
- DUMP.COM

Diese Programme sind jedoch bei der Verwendung von RunCPM schon vorhanden und befinden sich auf Laufwerk ***C1:***

Der Workflow schaut dabei ähnlich aus und verwendet anstatt *asm* das Programm *zsm* und anstatt *load* das Programm *hextocom*, wobei der Workflow ganz ähnlich ausschaut.

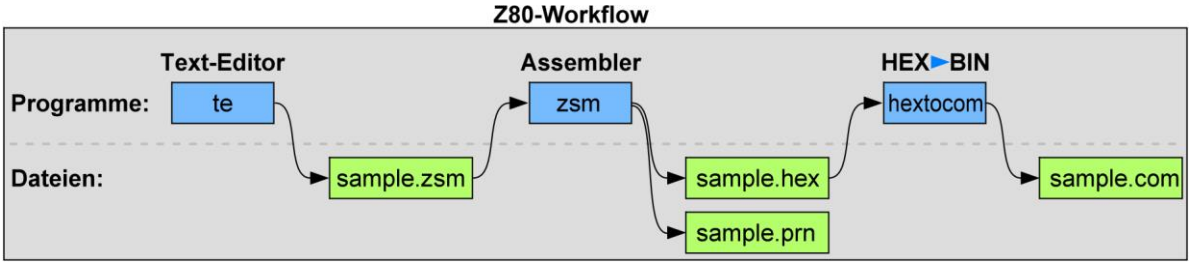


Abbildung 25 - Der Assembler-Workflow für den Z80

Ich denke, dass wir eine derartige Workflow-Session einmal komplett durchspielen sollten.

2.10.1 Die Quellcode-Eingabe über den Text-Editor TE

Ich möchte meine Z80-Programme auf dem Laufwerk ***C1:*** speichern und verwalten und wechsle über die folgenden Eingaben dieses Laufwerk.

```
A0>c:
C0>user 1
C1>dir
C: ALLOC    H   | ATEXIT   H   | BSEARCH  H   | BUILD    SUB
C: CC       COM | CCOPT    C   | CCOPT    COM | CCOPT    H
C: CCOPT    RUL | CLOCK    H   | CONIO    H   | COPYING  TXT
C: CPM      H   | CTYPE    H   | C_ASM    C   | C_BUF    C
C: C_CPP    C   | C_DEFS   C   | C_ERROR  C   | C_EXPR   C
C: C_IOCON  C   | C_IOFILE C   | C_MAIN   C   | C_PARSER C
C: C_STRING C   | FILEIO   H   | FPRINTF  H   | HELLO    C
C: HELLO    COM | HEXTOCOM C   | HEXTOCOM COM | INFO     TXT
C: MAKE_CC  C   | MEM      H   | MESCC    H   | MESCC    TXT
C: PRINTF   H   | QSORT    H   | RAND     H   | REDIR    H
C: SETJMP   H   | SPRINTF  H   | STDBOOL  H   | STRING   H
C: TEMPLATE C   | XPRINTF  H   | Z80      H   | ZSM      COM
C: ZSM      TXT
C1>
```

Abbildung 26 - Das Wechseln in das Verzeichnis C1:

Es ist zu sehen, dass sich dort schon die genannten und erforderlichen Tools ***zsm*** und ***hextocom*** befinden. Der Text-Editor ***te*** befindet jedoch auf Laufwerk ***A:***, was jedoch kein Problem darstellt, da CP/M zuerst immer auf Laufwerk ***A:*** nachschaut, ob sich das aufgerufene Programm dort befindet. Nach der Eingabe von ***te*** öffnet sich also der Text-Editor.

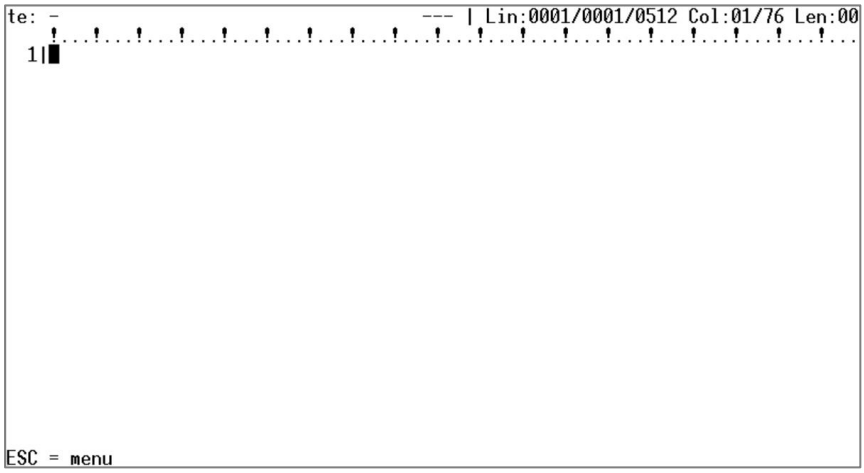


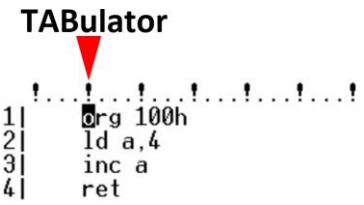
Abbildung 27 - Der Text-Editor TE

Es wird die momentan einzige vorhandene Zeilennummer 1 mit dem Cursor angezeigt und nun kann der Quellcode eingegeben werden. Die Handhabung des Editors habe ich zuvor schon kurz besprochen und ich denke, dass der Umgang recht schnell in Fleisch und Blut übergeht, je länger man damit hantiert!

Bevor es nun wirklich losgehen kann, ist es wichtig zu wissen, wie die Struktur von Assemblerprogrammen respektive einer einzelnen Zeile denn so ausschaut. Eine Unterscheidung zwischen Groß- beziehungsweise Kleinschreibung findet hier übrigens nicht statt. Es ist also nicht *Case-Sensitiv*, wie man Neu-Deutsch so sagt. In Assembler werden die einzelnen Befehle also durch Textzeilen formuliert, wobei jeder Befehl einer einzigen Zeile entspricht. Eine derartige Zeile besitzt die folgende Struktur.

[<Label>[:]]<Space/Tab><Befehl>[[<Space/Tab><Argument1>],<Argument2>]

Da es wichtig ist, an welcher Stelle im Speicher sich das Programm später befinden soll, ist eine zusätzliche Angabe erforderlich. Sie wird über die eine sogenannte Assembler-Direktive - auch *Pseudo-Anweisung* genannt - angegeben, die sich *org* nennt. Wir nutzen dazu - ich erwähnte es schon - den freien Speicher ab ***0100h***. Doch nun zum eigentlichen Programm, dass ich in den Text-Editor eingegeben und unter dem Namen *sample.zsm* abgespeichert habe. Da bei diesem Programm noch keine Labels - also Sprungmarken - zur Anwendung kommen, habe ich jede einzelne Zeile mit einem Tabulator nach rechts eingerückt. Wenn das nicht gemacht wird, liefert der Assembler *zsm* Fehler.



Um zu sehen, wie es auf dem Laufwerk ***C1:*** nun ausschaut, habe ich das Kommando *stat* abgesetzt.

```
C1>stat sample.zsm

 Recs  Bytes  Ext Acc
    1     4k    1 R/W C:SAMPLE.ZSM
Bytes Remaining On C: 7228k

C1>
```

Die Datei *sample.zsm* ist also vorhanden.

2.10.2 Der Aufruf des ZSM-Assemblers

Um nun diese Quelldatei zu assemblieren, wird das Programm *zsm* mit dem Namen der Datei ohne den Zusatz *.zsm* aufgerufen, was dann das folgende Ergebnis liefert.

```
C1>zsm sample
Zilog/Mostek Z80 Assembler Version 3.4 (Z80 CPU)

Pass 1
Pass 2

Errors    0

Finished


RunCPM Version 6.0 (CP/M 60K)

C1>
```

Der Assembler hat die Quelldatei also ohne einen Fehler (Errors 0) übersetzt. Dann wollen wir einmal sehen, wie es jetzt auf dem Laufwerk ***C1:*** ausschaut und ich habe zusätzlich nach dem eigentlichen Dateinamen *sample* gefiltert.

```
C1>stat sample.*

 Recs  Bytes  Ext Acc
    2     4k    1 R/W C:SAMPLE.HEX
    3     4k    1 R/W C:SAMPLE.PRN
    1     4k    1 R/W C:SAMPLE.ZSM
Bytes Remaining On C: 7240k

C1>
```

Es ist zu sehen, dass zwei neue Dateien erstellt wurden, die *sample.hex* und *sample.prn* lauten. Die für uns im Moment wichtige Datei ist die mit der Endung *.hex*, denn diese wird benötigt, um eine ausführbare COM-Datei zu generieren.

	Wo finde ich weitere Informationen zum ZSM-Z80-Assembler?
Das *ZSM-Manual* Manual befindet sich im gleichen Verzeichnis, wie das Programm selbst und kann über *zsm.txt* angesehen werden.	

2.10.3 Eine COM-Datei generieren

Um die COM-Datei zu generieren, muss das Programm *hextocom* mit dem Namen der Datei ohne den Datei-Zusatz *.hex* aufgerufen werden. Das schaut dann wie folgt aus.

```
C1>hextocom sample
HexToCom v1.05 / 10 Jan 2016

(c) 2007-2016 FloppySoftware

First address: 0100
Last address:  0103
Size of code:  0004 (4 dec) bytes


RunCPM Version 6.0 (CP/M 60K)

C1>
```

Es ist zu sehen, dass das Programm *sample.com* nach der Ausführung ab der Speicherstelle ***0100h*** im Speicher liegt, wobei das letzte Byte die Adresse ***0103h*** belegt und das Programm in Summe eine Länge von 4 Bytes besitzt. Ein erneuter Blick auf das Laufwerk ***C1:*** zeigt, dass nun eine Datei mit dem Namen *sample.com* generiert wurde.

```
C1>stat sample.*

 Recs  Bytes  Ext Acc
    1     4k    1 R/W C:SAMPLE.COM
    2     4k    1 R/W C:SAMPLE.HEX
    3     4k    1 R/W C:SAMPLE.PRN
    1     4k    1 R/W C:SAMPLE.ZSM
Bytes Remaining On C: 7228k

C1>
```

2.10.4 Ein Blick in die COM-Datei - DUMP

Kann man denn nun überhaupt einen Blick in die generierte binäre und ausführbare Datei mit der Endung *.com* werfen? Das ist möglich, denn für diese Zwecke gibt es die schon erwähnten Programme ***dump*** und ***ddt*** und fange einmal mit *dump* an. Dieses Programm zeigt - wir wissen es schon - den Inhalt einer Datei mit ihren binären Werten in Form von einzelnen Byte-Blöcken an. Die Ausgabe gestaltet sich wie folgt.

```
C1>dump sample.com

0000 3E 04 3C C9 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A
0010 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A
0020 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A
0030 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A
0040 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A
0050 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A
0060 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A
0070 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A

C1>
```

Die ersten vier Bytes zeigen genau die zuvor generierten Bytes in Form von HEX-Zahlen, die dem Maschinencode unseres Programms entsprechen. Jede, der hier gezeigten Zeilen besitzt 16 Bytes und eine führende Nummerierung, die ebenfalls eine HEX-Zahl ist. Die führende Nummerierung repräsentiert jedoch nicht die eigentlichen Speicherstellen, an denen sich das Programm befindet, das ja ab der Speicherstelle ***0100h*** abgelegt werden soll. Befindet sich das Programm *sample.com* denn überhaupt schon in diesem Bereich ab ***0100h***? Das

Programm *dump* zeigt nur den Inhalt einer Datei an und nicht die Inhalte von Speicherstellen im ROM beziehungsweise RAM. Ist das auch möglich? Natürlich, denn auch das haben wir schon besprochen!

2.10.5 Die Speicherbereiche anzeigen - DDT

Starten wir also das Programm *ddt* mit der Angabe des Dateinamens *sample*.com. Nach dem Strich, was das schon bekannte Prompt beziehungsweise Bereitschaftszeichen von *ddt* ist, muss dann ***d*** eingegeben, um einen Dump zu erzeugen.

```
C1>ddt sample.com
DDT VERS 1.4
NEXT  PC
0180 0100
-d
0100 3E 04 3C C9 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A >.<.............
0110 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A ................
0120 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A ................
0130 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A ................
0140 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A ................
0150 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A ................
0160 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A ................
0170 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A ................
0180 1A 84 12 13 C3 69 01 D1 2E 00 E9 2A 7C 1D EB 0E .....i.....*|...
0190 1A CD 67 1B C9 3E 0C D3 01 3E 08 D3 01 DB 01 07 ..g..>...>......
01A0 07 07 1F DA A9 08 C3 9D 08 DB 03 E6 7F C9 21 83 ..............!.
01B0 1D 70 2B 71 2A 82 1D 44 4D CD A1 07 0E 3A CD 86 .p+q*..DM....:..
-
```

Der Inhalt sieht ähnlich dem aus, der mit *dump* erzielt wurde und doch beinhalten die Informationen weitere Details. Ganz zu Beginn werden zwei wichtige Hinweise gegeben.

- ***NEXT***: Gibt die Adresse an, die als nächste belegt würde - hier ***0180h***
- ***PC***: Steht für *Programm-Counter* und ist der Programmzähler, der zeigt, an welcher Speicherstelle das Programm sich nach der Ausführung im Speicher befindet

Es ist über *ddt* auch möglich, sich den Inhalt in Form der Assembler-Mnemonics anzuschauen. Es muss dazu nur der Befehl *l* (kleines L) abgesetzt werden Hier wird jedoch der Code so angezeigt, als wäre er für den 8080 geschrieben worden. ***MVI A*** entspricht ***LD A*** und ***INR A*** entspricht ***INC A***.

```
C1>ddt sample.com
DDT VERS 1.4
NEXT  PC
0180 0100
-l
  0100  MVI  A,04
  0102  INR  A
  0103  RET
  0104  LDAX D
  0105  LDAX D
  0106  LDAX D
  0107  LDAX D
  0108  LDAX D
  0109  LDAX D
  010A  LDAX D
  010B  LDAX D
-
```

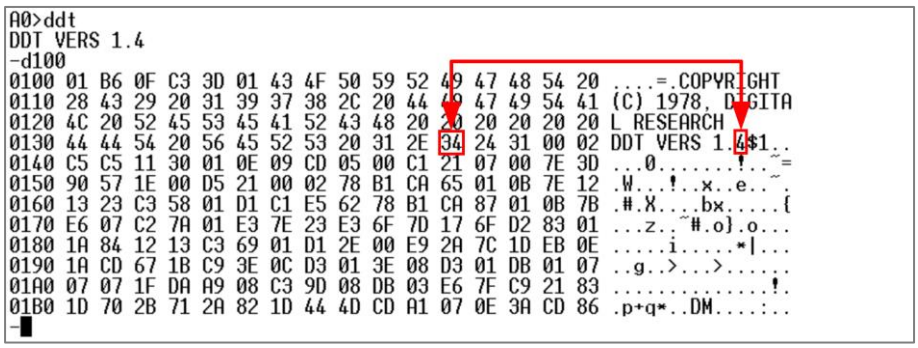
Das interessante an *ddt* ist, dass das geladene Programm schritt- beziehungsweise zeilenweise abgearbeitet werden kann und die Möglichkeit besteht, verschiedene Parameter wie ***PC*** (Program-Counter) und ***A*** (Akkumulator) zu inspizieren. Bei beiden handelt es sich um interne Register in im Z80 und es gibt noch einige andere mehr, auf die ich noch zu sprechen komme, die im

Moment jedoch nicht von Bedeutung sind. Vorab zeige ich hier schon einmal eine verkürzte Liste mit einigen Tastenkürzeln, um *ddt* zu bedienen.

Kommando	Bedeutung
A - Assembly	Ermöglicht die Eingabe von Assembler-Mnemonics
D - Display	Zeigt den Speicher in Hexadezimal und ASCII an
G - Go	Ein Programm wird ausgeführt und kann mit bis zu zwei optionalen Haltepunktadressen versehen werden
L - List	Listet Speicher mit Assembler-Mnemonics auf
R - Run	Startet ein Programm schrittweise zum anschließenden Testen
S - Set	Es können Speicherplätze untersucht und gegebenenfalls geändert werden
T - Trace	Verfolgt die Programmausführung mit der Anzeige der Registerinhalte
X - Examine	Die selektive Anzeige und Änderung des aktuellen CPU-Status (Register) für das zu testende Programm

2.10.6 Einen Programmcode modifizieren

Ich möchte nachfolgend zeigen, wie es mit dem Tastenkürzel ***S*** für *Set*, möglich ist, den Inhalt des Speichers zu modifizieren. Starten wir doch einfach das *ddt*-Programm und modifizieren eine einzige Speicherstelle, um dann zu sehen, welche Auswirkungen dies hat. Natürlich werde ich nicht den eigentlichen Programmcode anpassen, was garantiert zum Absturz des aufzurufenden Programms führen würde. Ich werde eine Textstelle ändern, was eine eher unkritische Modifikation ist. Nach dem Start wird ***d100*** eingegeben, was dazu führt, dass die Speicherstellen ab Adresse *100h* - das ist die Startadresse aller Programme - angezeigt werden und sich wie folgt gestaltet. Ziel ist es, den Text, den das ddt-Programm nach dem Starten ausgibt, anzupassen. Ich möchte die Version 1.4 auf die Version 1.5 anpassen, wobei ich lediglich den ASCII-Wert 34h, der für die Ziffer 4 steht, gegen 35h austauschen will, damit eine 5 erscheint. Ich habe den betreffenden Inhalt der Speicheradresse schon markiert. Nun müssen wir noch die eigentliche Speicheradresse bestimmen, was recht leichtfällt, denn der erste Wert in dieser Zeile steht an *0130h*. Wir zählen also 11 Positionen weiter nach rechts und landen dann bei Adresse *013Bh*.



Im nächsten Schritt wird diese Speicherstelle über die Eingabe von ***S013B*** aufgerufen. Es ist zu sehen, dass sich der Wert 34 zeigt. Der Cursor blinkt

und wartet auf eine Eingabe. Wird hier einfach die *RETURN*-Taste gedrückt, kommt es zu keiner Änderung des Inhaltes.

```
-s013B
013B 34 
```

Wir wollen aber dort den Wert 35 hinterlegen und geben diesen dort ein und bestätigen die Eingabe mit der *RETURN*-Taste. Daraufhin wird die nächste Speicherstelle angezeigt. Eine weitere Modifikation wollen wir nicht und geben zum Verlassen des *Set*-Modus einen Punkt ein und bestätigen diesen mit der *RETURN*-Taste und landen wieder beim *ddt*-Prompt. Hier darf nicht die *Strg-C*-Tastenkombination gedrückt werden, denn dann käme es zum Verlassen des *ddt*-Programms.

```
013B 34 35
013C 24 .
-
```

Um jetzt das ddt-Programm aus dem Speicher zu starten, muss lediglich der Befehl ***g100*** eingegeben werden, was dafür sorgt, dass die Ausführung ab Speicherstelle *0100h* erfolgt. Wir sehen die folgende Anzeige.

```
-g100
DDT VERS 1.5
-
```

Wow, so schnell kann man eine neue Version erzeugen. Kleiner Spaß am Rande! Da diese Modifikation lediglich innerhalb des Speichers erfolgte und sich nicht auf die Daten des Datenträgers auswirkte, erscheint das Programm nach einem erneuten Start vom CP/M-Prompt aus, natürlich in der Version 1.4.

2.10.7 Die Schrittweise Ausführung - DDT

Im nächsten Schritt wollen wir das geschriebene Programm etwas erweitern, damit der Inhalt des Akkumulators erhöht und erniedrigt wird. Dann lassen wir uns den Inhalt anzeigen. Bevor das jedoch geschieht, sollten wir einen Blick auf den sogenannten ***PC*** (Program-Counter) werfen, denn das ist ein essentieller Zähler innerhalb einer CPU. Der Z80 muss quasi im Auge behalten, an welcher Stelle im Speicher er den Code denn gerade ausführt und was als nächstes auszuführen ist. Er speichert diese Adresse in dem 16-Bit breiten PC-Register, wobei es diese Breite besitzen muss, um den ebenfalls 16-Bit breiten Adressbus komplett adressieren zu können. Der *PC* zeigt immer auf die Adresse, die als nächstes auszuführen ist. Das schauen wir uns am besten am schon erwähnten und erweiterten Quellcode an, der wie folgt ausschaut.

```
te: sample2.zsm                          --- | Lin:0006/0006/0512 Col:08/76 Len:07
    !...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...
  1|    org 100h
  2|    ld a, 4
  3|    inc a
  4|    inc a
  5|    dec a
  6|    ret
```

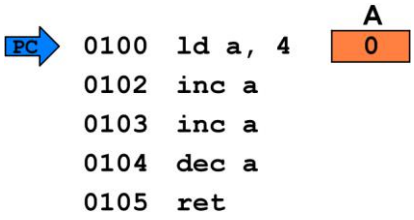
Es ist zu sehen, dass 2x ein ***inc a*** erfolgt und dann 1x ein ***dec a***, was für *Decrement* steht und Erniedrigung um den Wert 1 bedeutet. Der Inhalt des Akkumulators wird also schrittweise die Werte 4, 5, 6 und letztendlich 5 durchlaufen. Das hört sich sehr trivial an, was es ja auch ist, doch das Zusammenspiel von ***A*** und ***PC*** sollte genauer untersucht werden. Dabei hilft uns

natürlich wieder das Programm *ddt*. Zuerst muss also der gezeigte Quellcode nach gezeigter Abfolge eingegeben, assembliert und in eine COM-Datei überführt werden, was ich hier nicht erneut zeigen werden. Im Anschluss erfolgt der Aufruf von *ddt*, wobei ich die neue Datei ***sample2.zsm*** genannt habe. Um die Mnemonics anzuzeigen, wurde in *ddt* wieder *l* (keine Eins!) eingegeben, was zur folgenden Ausgabe führte.

```
C1>ddt sample2.com
DDT VERS 1.4
NEXT  PC
0180 0100
-l
  0100  MVI  A,04
  0102  INR  A
  0103  INR  A
  0104  DCR  A
  0105  RET
  0106  LDAX D
  0107  LDAX D
  0108  LDAX D
  0109  LDAX D
  010A  LDAX D
  010B  LDAX D
-
```

Natürlich werden hier wieder die 8080-Mnemonics zur Anzeige gebracht, was jedoch nicht weiter stören soll, denn es geht primär um den Maschinencode und dessen Ausführung, der ja - wie schon erwähnt - bei 8080 und Z80 identisch ist. Nun wollen wir das Programm schrittweise ausführen, um dann die Register A und PC zu inspizieren. Bei der nachfolgen Erläuterung verwende ich jedoch wieder die Z80-Mnemonics, um die Verwirrung nicht allzu groß werden zu lassen. Wie wir sehen, zeigt der ***PC*** zu Beginn auf die Speicherstelle ***0100h***, wobei diese Ausführung jedoch noch nicht stattgefunden hat. Der Inhalt des Akkumulators ist noch 0 oder besitzt einen anderen Wert, der uns jedoch nicht weiter zu interessieren hat.

Schritt 1: Ausgangssituation



Um den Zustand mithilfe von *ddt* nun abzufragen, muss ***t*** für *trace* eingegeben werden. Es ist zu sehen, dass *A=00* und *P(PC)=0100* sind. Rechts vom PC ist das Mnemonic *MVI A, 04* für den 8080 zu sehen, das im nächsten Schritt zur Ausführung kommt.

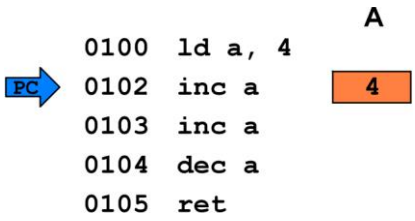
```
-t
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0000 S=0100 P=0100 MVI  A,04*0102
-
```

Schritt 2: Ausführung 1. Befehl: ld a,4

Um den 1. Befehl jetzt zur Ausführung zu bringen, wird mit ***r*** für *run* diese Zeile abgearbeitet und der PC auf die nächste Zeile positioniert. Im Anschluss können mit ***t*** wieder die Registerinhalte angezeigt werden. Doch eigentlich kann die Eingabe von ***r*** entfallen und nur mit ***t*** die Programmschritte abgearbeitet werden. Einfach mal ausprobieren!

```
-r
NEXT  PC
0180 0102
-t
C0Z0M0E0I0 A=04 B=0000 D=0000 H=0000 S=0100 P=0102 INR  A=0103
-
```

Es ist zu sehen, dass der Akkumulator jetzt den Wert *04* besitzt und der PC auf die nächste Zeile *0102h* positioniert wurde. Es ist dabei zu beachten, dass der PC nicht immer um den Wert 1 erhöht wird, denn bei *ld a, 04* handelt es sich um einen 2-Byte-Befehl. Auf der folgenden Abbildung ist das noch einmal übersichtlicher dargestellt.

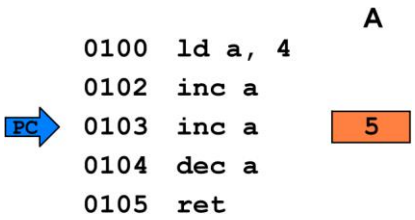


Schritt 3: Ausführung 2. Befehl: inc a

Um den 2. Befehl zur Ausführung zu bringen, wird wiederum mit ***r*** diese Zeile abgearbeitet und der PC auf die nächste Zeile positioniert. Im Anschluss können mit ***t*** wieder die Registerinhalte angezeigt werden.

```
-r
NEXT  PC
0180 0103
-t
C0Z0M0E0I0 A=05 B=0000 D=0000 H=0000 S=0100 P=0103 INR  A=0104
-
```

Es ist zu sehen, dass der Akkumulator jetzt den Wert *05* besitzt, da er über ***inc a*** um den Wert 1 erhöht wurde und der PC auf die nächste Zeile ***0103h*** positioniert wurde. Auf der folgenden Abbildung ist das erneut übersichtlicher dargestellt.



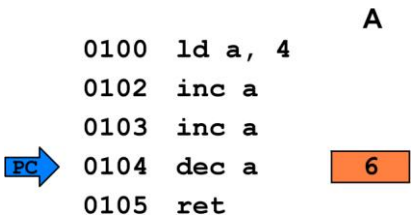
Schritt 4: Ausführung 3. Befehl: inc a

Um den 3. Befehl zur Ausführung zu bringen, wird wiederum mit ***r*** diese Zeile abgearbeitet und der PC auf die nächste Zeile positioniert. Im Anschluss können mit ***t*** wieder die Registerinhalte angezeigt werden.

```
-r
NEXT  PC
0180 0104
-t
C0Z0M0E0I0 A=06 B=0000 D=0000 H=0000 S=0100 P=0104 DCR  A=0105
-
```

Es ist zu sehen, dass der Akkumulator jetzt den Wert *06* besitzt, da er über *inc a* wieder um den Wert 1 erhöht wurde und der PC auf die nächste Zeile

0104h positioniert wurde. Auf der folgenden Abbildung ist das erneut übersichtlicher dargestellt.

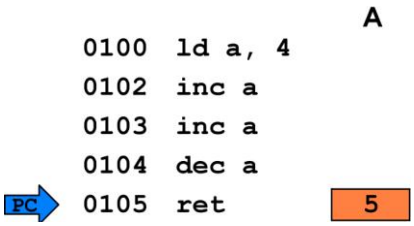


Schritt 5: Ausführung 4. Befehl: dec a

Um den 4. Befehl zur Ausführung zu bringen, wird wiederum mit ***r*** diese Zeile abgearbeitet und der PC auf die nächste Zeile positioniert. Im Anschluss können mit ***t*** wieder die Registerinhalte angezeigt werden.

```
-r
NEXT  PC
0180 0105
-t
C0Z0M0E0I0 A=05 B=0000 D=0000 H=0000 S=0100 P=0105 RET  *043E
-
```

Es ist zu sehen, dass der Akkumulator jetzt den Wert *05* besitzt, da er diesmal über ***dec a*** um den Wert 1 erniedrigt wurde und der PC auf die nächste Zeile ***0105h*** positioniert wurde. Auf der folgenden Abbildung ist das erneut übersichtlicher dargestellt.



Schritt 6: Ausführung 5. Befehl: ret

Über den nächsten Befehl ***ret*** würde die Kontrolle zurück an den Aufrufer gegeben und das Programm wäre abgearbeitet. Das hier vorgestellte Programm ist natürlich sehr einfach, doch es bietet sicherlich einen guten Einstieg in die Handhabung von Programmen, um aus einer Quelldatei eine COM-Datei über einen Assembler zu erstellen.

2.10.7.1 Das Programm ZSID

Kommen wir nun zur versprochenen Alternative zu *ddt*, die sich ***zsid*** nennt. Das Programm und das Manual sind unter den folgenden Internetadressen zu bekommen.

- http://www.cpm.z80.de/binary.html
- http://www.cpm.z80.de/manuals/zsid-m.pdf
- http://ucw.datatraveler.co.uk/manuals/zsid-m.pdf

Das Programm befindet sich bei *RunCPM* schon in Laufwerk ***A0:*** und braucht demnach nicht aus dem Internet heruntergeladen zu werden. Nach dem Aufruf von zsid mit der Angabe der Datei *sample2.com* und der Eingabe von ***d*** schaut die Ausgabe wie folgt aus.

```
C1>zsid sample2.com
ZSID VERS 1.5
NEXT  PC  END
0180 0100 DAFF
#l
  0100  LD   A,04
  0102  INC  A
  0103  INC  A
  0104  DEC  A
  0105  RET
  0106  LD   A,(DE)
  0107  LD   A,(DE)
  0108  LD   A,(DE)
  0109  LD   A,(DE)
  010A  LD   A,(DE)
  010B  LD   A,(DE)
#
```

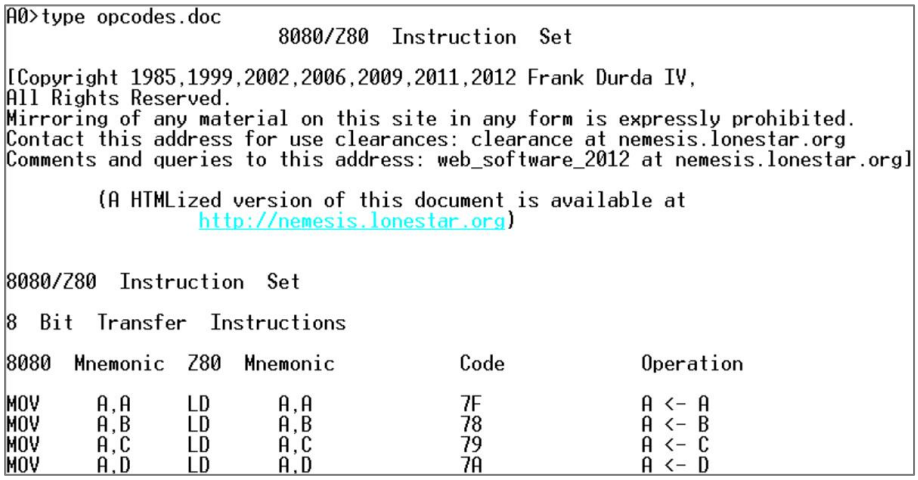
Und sieh da, es sind die Mnemonics zu sehen, die diesmal dem Z80 entsprechen. Fantastisch! Auch hier können mit der Taste ***t*** für *Trace* die einzelnen Programmschritte angesprungen werden. Es gut zu sehen, wie sich der Inhalt des Akkumulators schrittweise ändert.

```
#t
 ----- A=00 B=0000 D=0000 H=0000 S=0100 P=0100
 ----- A'00 B'0000 D'0000 H'0000 X=0000 Y=0000 LD   A,04
*0102
#t
 ----- A=04 B=0000 D=0000 H=0000 S=0100 P=0102
 ----- A'00 B'0000 D'0000 H'0000 X=0000 Y=0000 INC  A
*0103
#t
 ----- A=05 B=0000 D=0000 H=0000 S=0100 P=0103
 ----- A'00 B'0000 D'0000 H'0000 X=0000 Y=0000 INC  A
*0104
#t
 ----- A=06 B=0000 D=0000 H=0000 S=0100 P=0104
 ----- A'00 B'0000 D'0000 H'0000 X=0000 Y=0000 DEC  A
*0105
#t
 ----- A=05 B=0000 D=0000 H=0000 S=0100 P=0105
 ----- A'00 B'0000 D'0000 H'0000 X=0000 Y=0000 RET
*043E
#
```

Abschließend kann mit der Eingabe von ***Strg-C*** oder ***g0*** das Programm *zsid* verlassen werden. Perfekt!

2.10.8 Eine Liste der OP-Codes

Auf Laufwerk ***A0:*** befindet sich eine Datei mit Namen ***OPCODES.DOC***, die alle OPCodes der *8080*- und *Z80*-CPU enthält. Über den Befehl ***type opcodes.doc*** kann ein Blick in die Datei geworfen werden und liefert interessante Informationen.



```
A0>type opcodes.doc
                        8080/Z80  Instruction  Set

[Copyright 1985,1999,2002,2006,2009,2011,2012 Frank Durda IV,
All Rights Reserved.
Mirroring of any material on this site in any form is expressly prohibited.
Contact this address for use clearances: clearance at nemesis.lonestar.org
Comments and queries to this address: web_software_2012 at nemesis.lonestar.org]

        (A HTMLized version of this document is available at
                http://nemesis.lonestar.org)


8080/Z80  Instruction  Set

8  Bit  Transfer  Instructions

8080  Mnemonic  Z80  Mnemonic          Code            Operation

MOV     A,A     LD      A,A             7F              A <- A
MOV     A,B     LD      A,B             78              A <- B
MOV     A,C     LD      A,C             79              A <- C
MOV     A,D     LD      A,D             7A              A <- D
```

Abbildung 28 - Die OPCodes für 8080 und Z80

Mit einem Tastendruck wird weitergeblättert.

2.11 Den Z80-Cross-Assembler verwenden

Kommen wir nun zur angekündigten zweiten Variante, mit Assembler zu programmieren. Es handelt sich um den schon erwähnten Cross-Assembler. Ich stelle hier den *Pasmo* vor, der unter der folgenden Internetadresse zu finden ist.

http://pasmo.speccy.org/

Der Vorteil dieser Vorgehensweise besteht in der viel einfacheren Handhabung des Quellcodes über aktuelle Betriebssysteme wie zum Beispiel Windows. Wenn es um Texteditoren geht, die sehr gut mit Quellcodes umgehen können und zudem das *Syntax-Highlighting* beherrschen, dann ist *Notepad++* sicherlich eine sehr gute Wahl! Unter Syntax-Highlighting wird die Fähigkeit eines Texteditors verstanden, bestimmte Wörter abhängig von ihrer Bedeutung in unterschiedlichen Farben darzustellen. Notepad++ ist unter der folgenden Internetadresse zu finden.

https://notepad-plus-plus.org/

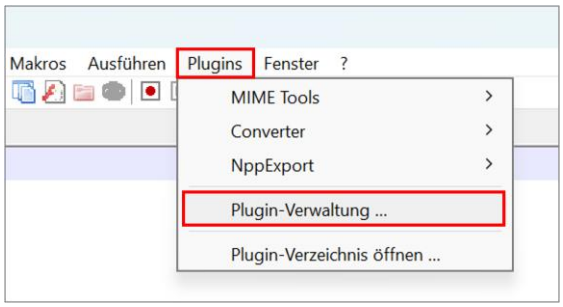
Im Folgenden stelle ich die erforderlichen Schritte vor, eine Entwicklungsumgebung für Z80 auf einem Windows-Rechner einzurichten. Sowohl Pasmo, als auch Notepad++ sollten jetzt heruntergeladen beziehungsweise installiert worden sein. Ich verwende Pasmo in der Version 0.5.4, wobei nach dem Entpacken der Zip-Datei die Datei ***pasmo.exe*** zur Verfügung steht und nicht installiert werden muss. Der Ordner von *Pasmo* sollte an einer bestimmten Stelle - am besten nicht im Download-Ordner - innerhalb des Dateisystems verschoben werden. Ich habe den Ordner einfach auf meine *D*-Partition verschoben und ihn *Pasmo* genannt. Der Aufruf von Pasmo gestaltet sich recht einfach, obwohl es Unmengen an weiteren Parametern gibt, mit denen Pasmo während des Aufrufs quasi konfiguriert werden kann.

pasmo.exe <Quelldatei.asm> <Zieldatei.com>

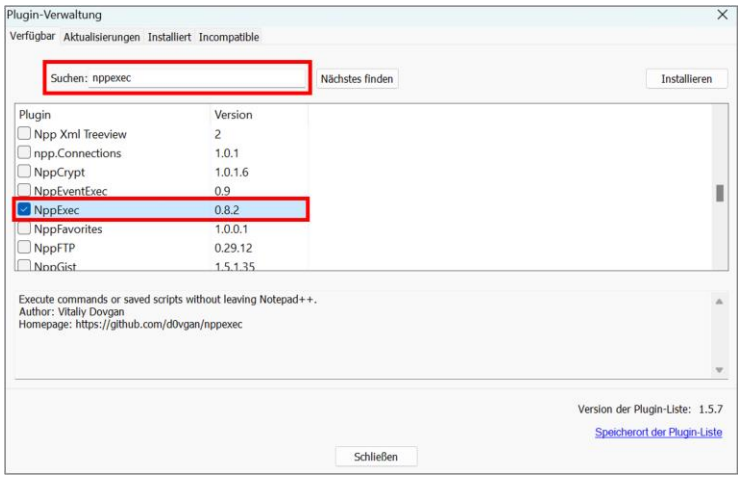
Nähere Informationen zu den diversen Parametern sind in der umfangreichen Dokumentation zu finden, die auf der genannten Internetseite einzusehen ist.

http://pasmo.speccy.org/pasmodoc.html

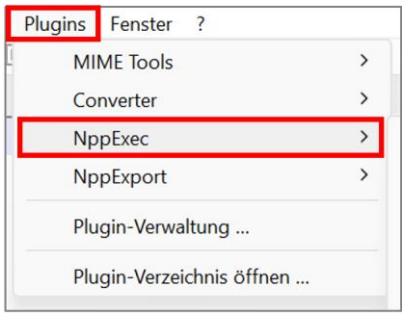
Nun geht es daran, Notepad++ so zu konfigurieren, dass aus dem Editor heraus ein bestimmter Quellcode kompiliert beziehungsweise assembliert werden kann. Dazu ist eine Erweiterung - auch *Plugin* genannt - erforderlich, die die ganze Sache sehr vereinfacht. Über den Menüpunkt *Plugins* muss dafür die *Plugin-Verwaltung* aufgerufen werden.



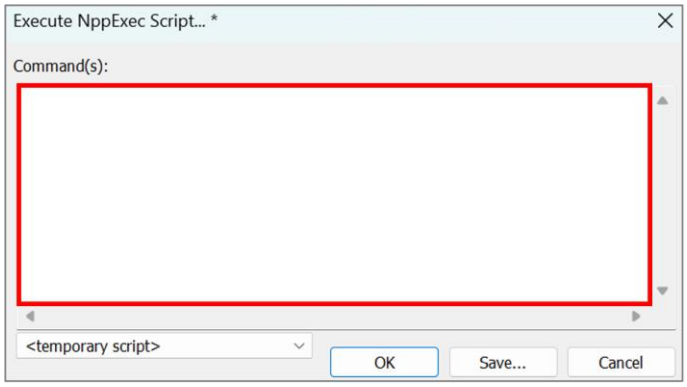
Im nächsten Schritt muss das Plugin mit dem Namen *NppExec* ausgewählt und über die Schaltfläche *Installieren* installiert werden. Um das Plugin schnellstmöglich zu finden, gibt man einfach im Suchfeld den entsprechenden Namen ein.



Wird im Anschluss erneut der Menüpunkt *Plugins* aufgerufen, ist dort unter anderem der Punkt *NppExec* zu sehen, der vorher aufgrund des fehlenden Plugins nicht dort war.



Entscheidend ist nun das Untermenü *Execute NppExec Script* auf der rechten Seite, was jetzt ausgewählt werden muss, wonach sich ein kleines Dialog-Fenster öffnet, das wie folgt ausschaut.



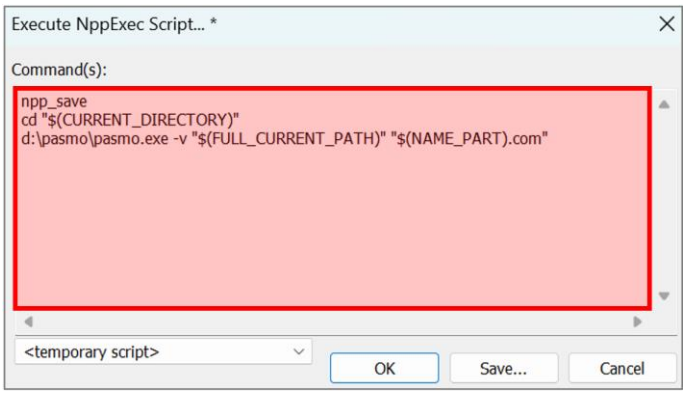
In diesen kleinen Editor müssen nun die entsprechenden Befehle eingetragen werden, um die Quelldatei mithilfe des Cross-Assemblers *Pasmo* in eine COM-Datei zu assemblieren. Woher sollte Notepad++ auch ohne diese Informationen wissen, wie das zu machen ist. Dazu sind einige Befehle erforderlich, die wie folgt ausschauen.

```
npp_save
cd "$(CURRENT_DIRECTORY)"
d:\pasmo\pasmo.exe -v "$(FULL_CURRENT_PATH)" "$(NAME_PART).com"
```

Was bedeuten die einzelnen Zeilen und was bewirken sie? Die folgende Tabelle gibt Aufschluss.

Kommando	Bedeutung
npp_save	Speichern der aktuellen Datei
cd "$(CURRENT_DIRECTORY)"	In das aktuelle Verzeichnis wechseln
d:\pasmo\pasmo.exe -v "$(FULL_CURRENT_PATH)" "$(NAME_PART).com"	Pasmo mit den erforderlichen Parametern aufrufen. Der Schalter -v (verbose=ausführlich) gibt Aufschluss über den ablaufenden Prozess und liefert eine Ausgabe in der Notepad-Konsole

Eingetragen in das Dialog-Fenster schaut das nun wie folgt aus, wobei ich das Skript unter dem Namen *pasmo* mithilfe der Schaltfläche *Save* gespeichert habe.



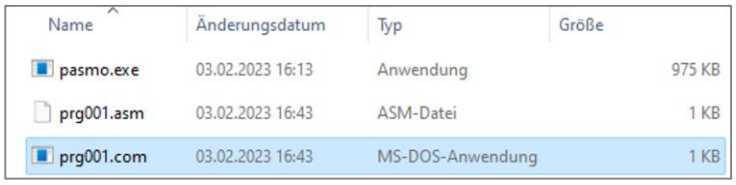
In *Notepad++* habe ich nun den schon bekannten und sehr einfachen Quellcode hereingeladen, unter *prg001.asm* abgespeichert, der sich dann wie folgt gestaltet.

```
prg001.asm
1   org #0100
2   ld a, 4
3   inc a
4   ret
```

Wenn nun über die Funktionstaste *F6* das Menü *Execute* des zuvor installierten Plugins aufgerufen wird und abschließend die Schaltfläche *OK* angeklickt wird, startet der Pasmo-Prozess. In Notepad++ wird ein Konsolen-Fenster unterhalb des Quellcodes angezeigt, der die Ausgaben von Pasmo aufgrund des Schalters *-v* anzeigt.

```
NppExec Console
Finished loading file: D:\Pasmo\prg001.asm in 5
Entering pass 1
Pass 1 finished
Entering pass 2
Pass 2 finished
<<< Process finished (PID=11268). (Exit code 0)
================ READY ================
```

Nun werfe ich einen Blick in mein Verzeichnis, in dem ich die Quelldatei abgespeichert habe.



Es ist zu sehen, dass sich dort nun eine COM-Datei befindet, die das Ergebnis des Pasmo-Cross-Compilers ist. Es steht dort zwar als Typ MS-DOS-Anwendung, ist aber in Wahrheit eine Z80-COM-Datei. Nun kann diese ausführbare Datei auf den USB-Stick des CP/M-Systems kopiert und ausgeführt werden. Auf diese Art und Weise kann eine Entwicklung eines Z80-Maschinenprogramms etwas komfortabler erfolgen. Aber das muss jeder wieder für sich selbst beurteilen, ob er sich nun für den traditionellen und eigentlichen Weg der Z80-Programmierung entscheidet oder den etwas moderner und nicht ganz so authentischen Weg beschreitet. Beide sind in meinen Augen einen Blick wert!

2.12 Tiefergehende Programmierung

Nachfolgend möchte ich ein paar Z80-Programme zeigen, die etwas mehr Hintergrundwissen voraussetzen. Natürlich ist das hier kein Z80-Maschinensprachekurs, denn das würde den Rahmen etwas sprengen. Aber vielleicht ist es dennoch interessant für einen Einsteiger und macht Lust auf mehr. Wenn es zum Beispiel darum geht, ein einzelnes Zeichen oder eine ganze Zeichenkette auf der Konsole anzuzeigen, dann stellt das *BDOS* einige nützliche Funktionen zur Verfügung, die über Funktionsnummern aufzurufen sind. Alle BDOS-Funktionen werden durch einen sogenannten *CALL*-Befehl an der Speicherstelle ***0005h*** aufgerufen. Um dem BDOS mitzuteilen, was es tun soll, muss dafür gesorgt werden, dass die internen Register des Z80 die erforderlichen Informationen enthalten, bevor die CALL-Anweisung ausgeführt wird. Die erwähnte Funktionsnummer des spezifischen Funktionsaufrufs, der aufgerufen werden soll, muss sich im *C-Register* befinden. Wenn zum Beispiel ein Ein-Byte-Wert an das BDOS übergeben muss, wie zum Beispiel ein einzelnes Zeichen, das an die Konsole gesendet werden soll - das werden wir gleich machen -, dann muss sichergestellt werden, dass sich dieser Wert im *E-Register* befindet. Wenn der Wert, der an das BDOS übergeben werden muss, ein 16-Bit-Wert ist, wie zum Beispiel die Adresse eines Puffers oder eines Dateisteuerblocks, dann muss dieser Wert im Registerpaar *DE* (Doppelregister) stehen. Das kommt dann beim Anzeigen einer Zeichenkette zum Tragen, was wir ebenfalls gleich sehen werden. Wenn BDOS einen 8-Bit-Wert zurückgibt, zum Beispiel ein Tastaturzeichen oder einen Rückgabecode, der den Erfolg oder Misserfolg der angeforderten Funktion anzeigt, wird dieser im *A-Register* zurückgegeben. Wenn das BDOS einen 16-Bit-Wert zurückgibt, steht dieser im Registerpaar *HL*. Unter der folgenden Internetadresse sind die BDOS-Funktionen aufgelistet.

https://www.seasip.info/Cpm/bdosfunc.html

Es ist natürlich wichtig zu erwähnen, dass das BDOS gibt keine Garantie über den Inhalt der anderen Register gibt. Die zuvor genannten Register werden bei einem BDOS-Aufruf gnadenlos überschrieben und wenn sich ein Wert in einem bestimmten Register befindet, der erhalten bleiben soll, so muss er vor dem Aufruf einer BDOS-Funktion in irgendeiner Art und Weise gesichert werden, indem er entweder im Speicher abgelegt oder auf den sogenannten *Stack* geschoben wird. Der Stack, was übersetzt *Stapel* bedeutet, ist eine der wichtigsten Grundstrukturen in der Programmierung. Er funktioniert nach dem Prinzip, dass man bildlich gesprochen mehrere Elemente aufeinanderstapelt und immer nur auf das oberste Element dieses Stapels zugreifen kann um es dann zu lesen und/oder zu entfernen. Das Verhalten eines Stapels wird im Englischen auch mit den vier Buchstaben *LIFO* abgekürzt, was übersetzt *Last-In-First-Out* bedeutet. Was also zuletzt auf den Stapel gepackt wurde, wird zuerst wieder von ihm entnommen.

2.12.1 Ein Zeichen auf der Konsole ausgeben

Um ein einziges Zeichen auf der Konsole - sprich im Terminal-Fenster - anzuzeigen, sind in Assembler so einige Dinge zu beachten. Das *BDOS* stellt - ich erwähnte es gerade - einige Funktionen zur Verfügung, die die Handhabung mit dem Betriebssystem CP/M sehr vereinfachen. Die nachfolgende Liste zeigt

einige Funktionen, wobei die diejenigen, die gleich genutzt werden, schon markiert sind.

0000 =	B$SYSRESET	EQU	0	;System Reset
0001 =	B$CONIN	EQU	1	;Read Console Byte
0002 =	B$CONOUT	EQU	2	;Write Console Byte
0003 =	B$READIN	EQU	3	;Read "Reader" Byte
0004 =	B$OUNOUT	EQU	4	;Write "Punch" Byte
0005 =	B$LISTOUT	EQU	5	;Write Printer Byte
0006 =	B$DIRCONIO	EQU	6	;Direct Console I/O
0007 =	B$GETIO	EQU	7	;Get IOByte
0008 =	B$SETIO	EQU	8	;Set IOByte
0009 =	B$PRINTS	EQU	9	;Print Console String

Tabelle 7 - BDOS-Funktionen (Auszug)

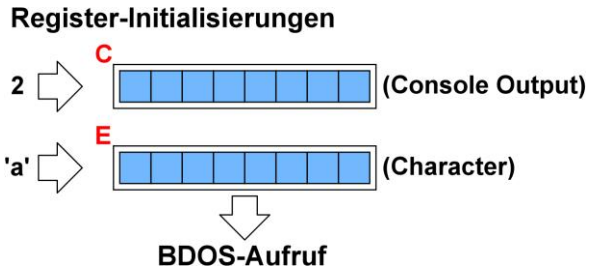
Wenn also zum Beispiel ein einzelnes Zeichen auf der Konsole angezeigt werden soll, dann besitzt der sogenannte Function-Code den Wert 2 (Console Output). Dieser Wert muss in das *C-Register* geladen werden. Das anzuzeigende Zeichen muss dann im *E-Register* abgelegt werden. Abschließend erfolgt dann der Aufruf von BDOS über die Adresse ***0005h***. Die folgende Auflistung zeigt die erforderlichen Parameter.

- **Function Code:** 2
- **Bezeichnung:** Console Output / Konsolenausgabe
- **Input:** C = 02h, E = ASCII-Wert
- **Output:** -

Das entsprechende Programm mit Namen *out.zsm* schaut wie folgt aus, wobei das Zeichen *‚a'* auf der Konsole ausgegeben wird. ***LD*** (load) in den Zeilen 3 und 4 sorgt für das Laden der angegebenen Register mit dem nachfolgend aufgeführten Wert.

```
te: OUT.ZSM                          --- | Lin:0006/0006/0512 Col:01/76 Len:31
   !...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...
  1|     org 0100h  ; free memory
  2|BDOS equ 0005h  ; BDOS entry point
  3|     ld c,2     ; BDOS function print char
  4|     ld e,'a'   ; load char
  5|     call BDOS  ; CALL BDOS
  6|     ret        ; return to CCP
```

Hier noch einmal die Details zum Programm.



Der Aufruf der BDOS-Adresse ***0005h*** führt letztendlich zur Ausführung des Konsolen-Outputs mit dem Zeichen *‚a'*. Wird das gezeigte Programm assembliert und ausgeführt, zeigt sich die folgende Ausgabe im Terminal-Fenster.

```
C1>out
a
C1>
```

In diesem Programm ist eine weitere Assembler-Direktive zur Anwendung gekommen, die sich ***EQU*** nennt und einen symbolischen Namen darstellt, dem ein Wert zugewiesen werden soll. In Hochsprachen wie zum Beispiel *C* werden sogenannte *Variablen* benannt, die einen bestimmten Wert repräsentieren, denn der Wert ***0005h*** ist wohl eine sogenannte magische Konstante, wo keiner weiß, was sich dahinter verbirgt. Jetzt kommt *EQU* ins Spiel. Das schaut doch schon viel „sprechender" aus und macht mehr Sinn. Es wird quasi eine Variable mit dem Namen *BDOS* geschaffen, die dann im weiteren Verlauf des Quellcodes immer wieder verwendet werden kann. Über ***CALL*** in Zeile 5 wird zur angegebenen Adresse gesprungen, was dem Aufruf eines Unterprogramms gleichkommt. Über ***RET*** (return) wird das Programm endgültig verlassen und die Kontrolle an die Konsole zurückgegeben.

2.12.2 Mehrere Zeichen auf der Konsole ausgeben

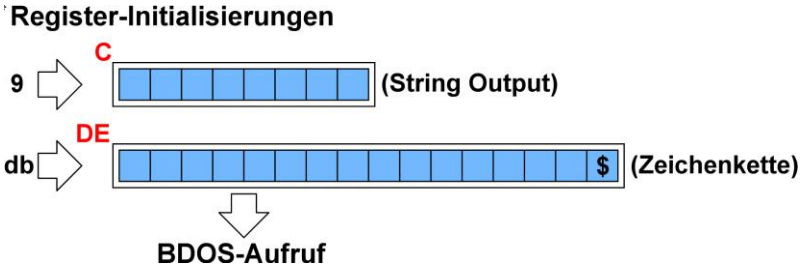
Ein einzelnes Zeichen auf der Konsole ist natürlich recht spärlich, denn in den meisten Fällen müssen ganze Texte zur Anzeige von Statusinformationen angezeigt werden. Ist das überhaupt möglich, ohne, dass ein mehrfacher Aufruf der Anzeige einzelner Zeichen erfolgen müsste? Auch dafür gibt es eine BDOS-Funktion, die diese Aufgabe übernimmt. Anstatt des Function-Codes 2 muss jetzt der Code 9 verwendet werden. Das ist aber nur die halbe Wahrheit, denn das zuvor verwendete *E-Register*, das zur Anzeige eines einzelnen Zeichens verwendet wurde, kann nicht mehrere Zeichen aufnehmen beziehungsweise speichern. Wie können also mehrere Zeichen zur Anzeige gebracht werden? Dafür reicht ein einzelnes Register natürlich nicht aus! Es muss also eine Technik zur Anwendung kommen, die quasi auf eine definierte Zeichenkette verweist. Dies wird mit einem sogenannten *Zeiger*, *Vektor* oder *Pointer* realisiert, der auf eine Startadresse innerhalb des Speichers verweist, wo die anzuzeigende Zeichenkette abgelegt ist. Da dieser Zeiger jedoch einen Speicherbereich von 64K abdecken muss, kann es sich also nicht um einen 8-Bit-Zeiger handeln, der nur die Adressen von 0 bis 255 erreichen kann. Ein 16-Bit-Zeiger schafft dagegen schon mehr und kann die vorhandenen 64K voll abdecken. Deswegen kommt jetzt das *DE-Register* zum Tragen. Die folgende Auflistung zeigt die erforderlichen Parameter.

- **Function Code:** 9
- **Bezeichnung:** Print String / Zeichenkette ausgeben
- **Input:** DE => Zeichenkette
- **Output:** -

Das Registerpaar *DE* zeigt auf das erste Byte der Zeichenkette. Die anzuzeigende Zeichenkette muss dabei mit dem Zeichen *$* abgeschlossen werden. Es ist darauf zu achten, dass das letzte Zeichen der Zeichenkette wirklich das Zeichen *$* ist, was BDOS verwendet, um dieses Zeichen als Markierung für das Ende der Zeichenkette zu akzeptieren. Das Zeichen $ selbst wird nicht auf der Konsole ausgegeben. Einer der größten Nachteile dieser Funktion ist die Verwendung des Zeichens *$* als Beendigungszeichen, dass sonst innerhalb der Zeichenkette nicht verwendet werden darf. Sehen wir uns zunächst das Programm an, das ich *out2.zsm* genannt habe.

```
te: OUT2.ZSM                              --- | Lin:0007/0007/0512 Col:37/76 Len:36
    !...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...
  1|     org 0100h  ; free memory
  2|BDOS equ 0005h  ; BDOS entry point
  3|     ld c,9     ; BDOS function print string
  4|     ld de,MSG  ; load address of string
  5|     call BDOS  ; CALL BDOS
  6|     ret        ; return to CCP
  7|MSG  db 'Hallo, hier spricht CP/M!$'
```

Das C-Register wird nun mit dem Wert 9 geladen, was der *Print-String* Funktion des BDOS entspricht. Im nächsten Schritt wird das DE-Register (Doppelregister) mit einer Adresse geladen, die auf die anzuzeigende Zeichenkette verweist, die über das Label *MSG* zu erreichen ist. Die Assembler-Direktive ***db*** spiel hier eine entscheidende Rolle. Die Buchstaben *db* steht für *define byte*. Es erlaubt die Definition von einem oder mehreren literalen Bytes, sowie Zeichenketten von Bytes. Literal bedeutet übersetzt *Buchstabe*. Hier noch einmal die Details zum Programm.



Der Aufruf der BDOS-Adresse ***0005h*** führt letztendlich wiederum zur Ausführung des Konsolen-Outputs, doch diesmal mit der Ausgabe der Zeichenkette. Wird das gezeigte Programm assembliert und ausgeführt, zeigt sich die folgende Ausgabe im Terminal-Fenster.

```
C1>out2
Hallo, hier spricht CP/M!
C1>
```

Ein Blick in die COM-Datei zeigt die folgenden Informationen.

```
C1>ddt out2.com
DDT VERS 1.4
NEXT  PC
0180 0100
-d
0100 0E 09 11 09 01 CD 05 00 C9 48 61 6C 6C 6F 2C 20 .........Hallo,
0110 68 69 65 72 20 73 70 72 69 63 68 74 20 43 50 2F hier spricht CP/
0120 4D 21 24 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A M!$.............
0130 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A ...............
0140 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A ...............
0150 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A ...............
0160 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A ...............
0170 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A ...............
0180 1A 84 12 13 C3 69 01 D1 2E 00 E9 2A 7C 1D EB 0E .....i.....*|...
0190 1A CD 67 1B C9 3E 0C D3 01 3E 08 D3 01 DB 01 07 ..g..>...>......
01A0 07 07 1F DA A9 08 C3 9D 08 DB 03 E6 7F C9 21 83 ..............!.
01B0 1D 70 2B 71 2A 82 1D 44 4D CD A1 07 0E 3A CD 86 .p+q*..DM....:..
-
```

Dort ist die anzuzeigende Zeichenkette *„Hallo, hier spricht CP/M!“* sehr gut zu erkennen. Hinter dem Ausrufezeichen *!* befindet sich das erwähnte Zeichenketten-Terminierungszeichen *$*. Wer sich einen Überblick über die komplette ASCII-Tabelle verschaffen möchte, kann auf der folgenden Internetseite nachsehen.

https://www.torsten-horn.de/techdocs/ascii.htm

2.12.3 Mehrere gleiche Zeichen auf der Konsole ausgeben

Wir haben schon gesehen, dass es möglich ist, eine Zeichenkette auf der Konsole auszugeben. Wie ist es aber möglich, zum Beispiel ein bestimmtes einzelnes Zeichen mehrfach auszugeben und das in Abhängigkeit eines bestimmten Wertes? Eine Fortschrittsanzeige wäre da ein repräsentatives Beispiel. Jeder einzelne Punkt auf der Konsole würde 10% Fortschritt von Irgendetwas darstellen und wenn 10 Punkte sichtbar sind, ist der Fortschritt beendet. Mit dem folgenden Programm können zum Beispiel zwei Punkte auf der Konsole angezeigt werden. Das Programm habe ich *out3.zsm* genannt.

```
te: OUT3.ZSM                          --- | Lin:0009/0009/0512 Col:32/76 Len:31
    !...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...
  1|     org 0100h  ; free memory
  2|BDOS equ 0005h  ; BDOS entry point
  3|     ld c,2     ; BDOS function print char
  4|     ld e,'.'   ; load char
  5|     call BDOS  ; CALL BDOS
  6|     ld c,2     ; BDOS function print char
  7|     ld e,'.'   ; load char
  8|     call BDOS  ; CALL BDOS
  9|     ret        ; return to CCP
```

Es ist zu sehen, dass der Punkt in das *E-Register* geladen wurde, und dann zweimal hintereinander die BDOS-Funktion für das Anzeigen eines Zeichens auf der Konsole aufgerufen wurde. Die Anzeige schaut auf der Konsole dementsprechend aus.

```
C1>out3
..
C1>
```

Es sind zwei Punkte zu sehen. Manch einer mag jetzt vielleicht denken, dass nach der Initialisierung der beiden Register ***c*** und ***e*** lediglich der BDOS-Aufruf ***call BDOS*** zwei Mal hintereinander erforderlich wäre. Warum müssen also vorher beim erneuten BDOS-Aufruf wiederum die Register ***c*** und ***e*** einer Initialisierung unterzogen werden? Falls das nicht erfolgt, kann es je nach BDOS-Version Probleme geben. Das liegt darin begründet, dass beim internen Aufruf von BDOS-Funktionen diese wiederum zahlreiche Register nutzen und es kann vorkommen, dass die vor dem BDOS-Aufruf initialisierten Register, durch den eigentlichen Aufruf mit anderen Werten überschrieben werden. Ein weiterer BDOS-Aufruf könnte scheitern oder das komplette System einfrieren.

Wie kann das Programm jedoch ein wenig flexibler gestaltet werden, dass der Aufruf der BDOS-Funktion von einer bestimmten Variablen abhängig ist? Die Lösung liegt in den sogenannten *Kontrollstrukturen*, die durch *Schleifen*, die auch *Loops* genannt werden, realisiert werden. Vielleicht sind dem einen oder anderen die *FOR-NEXT*-Schleifen aus den Zeiten von Basic noch ein Begriff. Auf ähnliche Art und Weise arbeitet auch das Z80-Konstrukt zur Realisierung derartiger Wiederholungen. Sehen wir uns das gleich genauer an. Vorher möchte ich jedoch etwas sehr Wichtiges einschieben, denn es ist von essentieller Bedeutung in der Programmierung von Maschinensprache. Es geht dabei um den schon kurz erwähnten *Stack*.

2.12.4 Der Stack

Ich sagte gerade eben, dass beim Aufruf von BDOS-Funktionen diverse Register genutzt werden, und wurden vor dem Aufruf bestimmte Register initialisiert, bedeutet das nicht, dass diese nach einem BDOS-Aufruf immer noch dieselben Werte besitzen. Es muss also eine Technik gefunden werden, diese Werte in irgendeiner Art und Weise zu sichern. Jetzt kommt der *Stack* ins Spiel. Beim Stack handelt es sich um einen sogenannten *Stapelspeicher*, der nach einem bestimmten Prinzip aufgebaut ist.

Ein Stack besitzt die grundlegende Aufgabe der temporären Speicherung von Werten. Da die internen Register in der Anzahl recht knapp bemessen sind, ist der Stack in einem bestimmten Speicherbereich des RAMs anzutreffen. Nicht selten kommt es in der Programmierung vor, dass wichtige Daten (Registerinhalte oder Speicheradressen) gespeichert werden müssen, weil kurzzeitig ein noch wichtigeres Thema bearbeitet werden muss. Nach dessen Abarbeitung beziehungsweise Beendigung kann es dann an vorheriger Stelle mit den gesicherten Daten weitergehen. Was könnte das für ein Szenario sein, auf das ich hier abziele? Ganz einfach! Es geht um den Aufruf einer Unterroutine aus dem Hauptprogramm heraus. Wurde dieses Unterprogramm abgearbeitet, muss die Ausführung des Hauptprogramms natürlich an geeigneter Stelle fortgesetzt werden, als wenn das Unterprogramm überhaupt nicht existiert hätte. Genau das ist beim BDOS-Aufruf der Fall. Man kann sich das wie folgt vorstellen wie es auf der folgenden Abbildung zu erkennen ist.

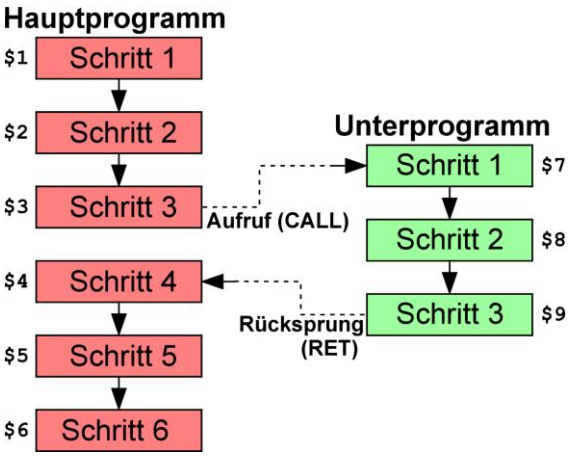


Abbildung 29 - Das Hauptprogramm ruft ein Unterprogramm

Auf der linken Seite ist ein Hauptprogramm mit 6 Schritten zu sehen, dass sich im fiktiven Adressbereich von $1 bis $6 befindet. Auf der rechten Seite ist ein Unterprogramm mit 3 Schritten zu sehen, das sich im fiktiven Adressbereich von $7 bis $9 befindet. Würde das Hauptprogramm keinen Unterprogrammaufruf besitzen, wäre die Sache mit dem Programmzähler recht einfach. Der Aufruf der Adressen $1 bis $6 bestünde im simplen Hochzählen der Adresse $1 bis zum Ende des Programms an Adresse $6. Nun gibt es aber das Unterprogramm, das vom Hauptprogramm an Adresse $3 aufgerufen wird. Es erfolgt ein Sprung zur Adresse $7 und der Programmzähler (*PC*: Program Counter) arbeitet in bekannter Weise dort wieder die Adressen ab. Nach Beendigung des Unterprogramms an Adresse $9 soll es jetzt zurück zum Hauptprogramm gehen. Doch hoppla! An welcher Stelle geht es denn jetzt weiter??? Klar, wir sehen

das hier auf der Abbildung recht eindeutig, doch die CPU hat keine Ahnung, an welcher Stelle denn nun der Aufruf zum Unterprogramm erfolgte und wo es dann später weitergehen soll! Die Lösung des Problems besteht in der Sicherung der sogenannten Rücksprungadresse vor der Abarbeitung des Unterprogramms. Diese Rücksprungadresse ist hier $4, die bei Schritt 4 des Hauptprogramms angesiedelt ist.

Kommen wir noch mal zurück zum Stapel, der nach seiner eigentlichen Funktion die sogenannte *LIFO*-Struktur (last-in-first-out) vorweist. Übersetzt bedeutet das: Das zuletzt abgelegte Element wird auch wieder zuerst ausgegeben. Um sich das bildlich vorzustellen, kann ein Aktenstapel bemüht werden, auf dem nach und nach neue Hefter oben abgelegt werden. Will man eine Akte bearbeiten, wird diese wieder von oben heruntergenommen. Die zuletzt abgelegte kommt also zuerst in Betracht. Das erste Stapelelement – hier das dunkelblaue Element - befindet sich naturgemäß auf dem Boden des Stapels, das letzte Stapelelement – hier das Gelbe - auf dessen Spitze. Um also an die zuvor abgelegten Elemente des Stapels zu gelangen, muss dieser von oben nach unten quasi abgetragen werden. Die Struktur des Stacks schaut wie folgt aus.

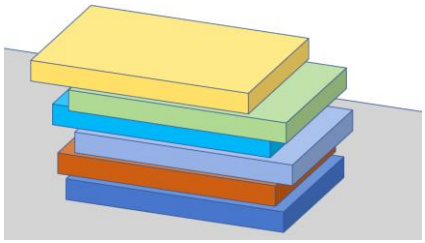


Abbildung 30 - Irgendein Stapel

Sehen wir uns dazu einmal ein Beispiel an, bei dem zwei Befehle genutzt werden, um einerseits etwas auf den Stapel abzulegen, was *Push* genannt wird und andererseits etwas vom Stapel zu nehmen, was *Pull* oder *Pop* genannt wird.

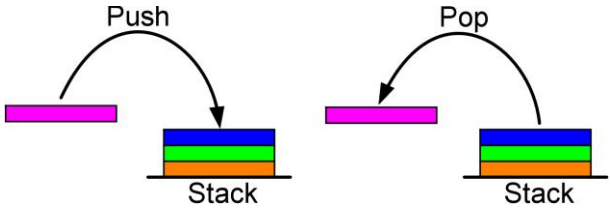


Abbildung 31 - Die beiden Stack-Aktionen

Der Stack beim Z80 ist in einem beliebigen Speicherbereich untergebracht. Dieser Stapel wächst von oben nach unten, also von den höheren zu den niedrigeren Speicheradressen wächst, was bedeutet, dass das zuletzt auf den Stack gelegte Element immer die kleinste Adresse aller Stapelelemente besitzt. Damit jedoch auf den Stack zugegriffen werden kann, ist ein Zeiger erforderlich, der auf das oberste - zuletzt hinzugefügte - Element des Stacks zeigt. Dieser Zeiger ist in einem speziellen Register, dem Stack-Pointer, untergebracht und wird mit *SP* abgekürzt. Da der gesamte Speicherbereich von 64 KByte nicht mit einer einzigen 8-Bit-Adresse adressiert werden kann, muss dieser Stack-Pointer eine Breite von 16-Bit besitzen. Auf der nachfolgenden Abbildung habe ich den SP als Register farblich hervorgehoben.

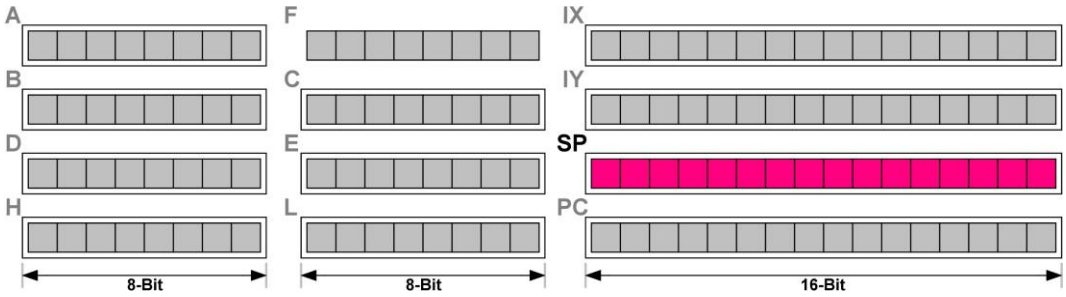


Abbildung 32 - Der Stack-Pointer (16-Bit)

Der Stack zeigt stets auf das oberste Element des Stapels, was mit *LSB* (Least-Significant-Byte) bezeichnet wird. Ich denke, wir sollten uns dazu ein kleines Programm ansehen, welches Werte auf den Stack legt und diese auch wieder entnimmt. Wir erinnern uns noch einmal: Der *push*-Befehl legt etwas oben auf den Stack und der *pop*-Befehl entnimmt das oberste Element. Die Vorgänge laufen wie folgt ab.

Das Ablegen eines Byte-Wertes auf den Stack über push:

- Der Stackpointer SP wird um den Wert 1 vermindert
- Der Byte-Wert wird an dieser Stelle gespeichert

Das Holen eines Byte-Wertes vom Stack über pop:

- Der Byte-Wert, auf den der Stackpointer im Speicher zeigt, wird vom Stack geholt
- Der Stackpointer wird um den Wert 1 erhöht

Da der Stack - wie erwähnt - im Speicher von oben nach unten wächst, drehe ich die Speicheraufteilung nur optisch um und lege den höheren Wert an das untere Ende des anzuzeigenden Speicherblocks. Angenommen, wir haben es mit dem folgenden Programm zu tun.

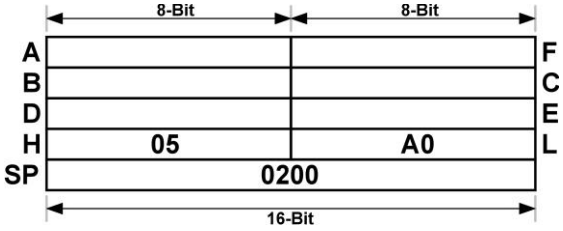
```
te: STACK1.ZSM                          --- | Lin:0011/0011/0512 Col:33/76 Len:32
    !...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...
  1|    org 0100h    ; free memory
  2|    ld sp, 0200h ; stack address
  3|    ld hl, 05a0h ; HL-Register
  4|    ld bc, 17f2h ; BC-Register
  5|    push bc      ; push BC on stack
  6|    push hl      ; push HL on stack
  7|    ld hl, 0000h ; clear HL-Register
  8|    ld bc, 0000h ; clear BC-Register
  9|    pop hl       ; pop HL from stack
 10|    pop bc       ; pop BC from stack
 11|    ret          ; return to ccp
```

Gehen wir das schrittweise durch. Damit der Stack auch über ein Programm richtig genutzt werden kann, muss dieser initialisiert werden. Das geschieht über den Befehl ***ld sp, 0200h*** in Zeile 2. Das SP-Register wird mit dem Wert ***0200h*** versehen und teilt dem System mit, dass der Stack ab Adresse ***0200h*** zu finden ist. Es ist zu bemerken, dass diese Adresse lediglich den Ausgangspunkt für den Stack markiert und der erste abzulegende Wert erst ab Adresse ***0200h - 1*** zu finden ist!

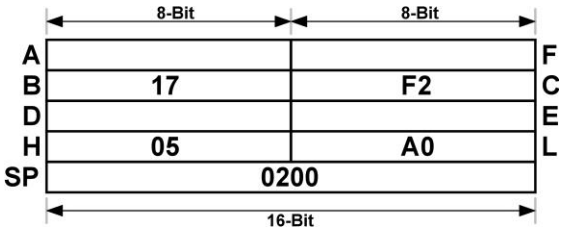
Um im Programm weiter fortzufahren, werden in den Zeilen 3 und 4 einige Register mit Testwerten versehen, die später auf den Stack geschoben werden sollen. Das sehen wir uns jetzt im Detail an.

In Zeile 2 wird der Stackpointer SP mit dem Wert 0200h initialisiert und zeigt auf diese Speicheradresse.

In Zeile 3 wird das Registerpaar HL mit den angegebenen Werten ***05a0h*** initialisiert.



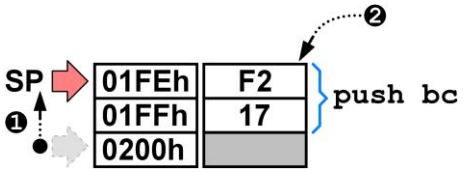
In Zeile 4 wird das Registerpaar BC mit den angegebenen Werten ***17f2h*** initialisiert.



Kommen wir nun zu den Stack-Aktionen. Zuerst werden die Registerpaare auf dem Stack abgelegt. Da es sich um je 16-Bitwerte handelt, beträgt der Wert für das Inkrementieren und Dekrementieren des Stackpointers 2.

In Zeile 5 wird über ***push bc*** das Registerpaar auf dem Stack abgelegt. Die Reihenfolge der beiden Aktionen ist.

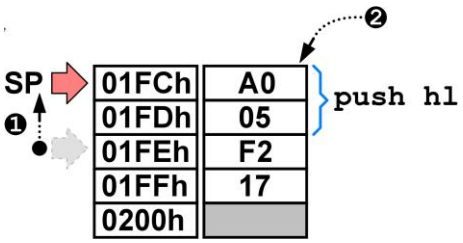
1. Den Stackpointer SP um den Wert 2 vermindern
2. Das Ablegen des Wertepaares ***17f2h*** über *push* auf dem Stack



Da es sich um einen 16-Bit-Wert handelt, zeigt der Stackpointer auf das *LSB* (Least-Significant-Byte), was in diesem Fall der Wert ***F2h*** ist.

In Zeile 6 wird über ***push hl*** das Registerpaar auf dem Stack abgelegt. Die Reihenfolge der beiden Aktionen ist.

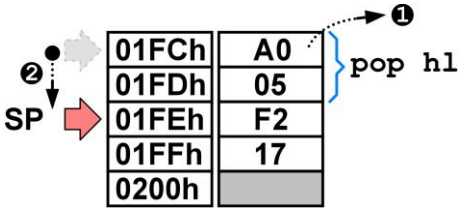
1. Den Stackpointer SP um den Wert 2 vermindern
2. Das Ablegen des Wertepaares ***05a0h*** über *push* auf dem Stack



Um die Werte, die auf dem Stack abgelegt worden sind, zurückzuholen, muss das natürlich in einer bestimmten Reihenfolge erfolgen. Der zuletzt abgelegte Wert beziehungsweise das zuletzt abgelegte Wertepaar - in unserem Fall das Registerpaar *hl* - muss zuerst über *pop* geholt werden und dann das davor und so weiter und so fort. Doch bevor es weitergeht, möchte ich zur besseren Veranschaulichung den Inhalt der beiden Registerpaare *hl* und *bc* löschen, was in den Zeile 7 und 8 über ***ld hl, 0000h*** und ***ld bc, 0000h*** erfolgt.

In Zeile 9 wird über ***pop hl*** das Registerpaar wieder vom Stack geholt und in die Register geschrieben. Die Reihenfolge der beiden Aktionen ist.

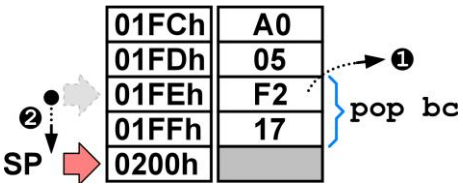
1. Das Holen des Wertepaares ***05a0h*** über *pop* vom Stack
2. Den Stackpointer SP um den Wert 2 erhöhen



Nun geht es daran, dass Registerpaar *bc* wieder vom Stack zu holen und in die Register zu schreiben.

In Zeile 10 wird über ***pop bc*** das Registerpaar wieder vom Stack geholt und in die Register geschrieben. Die Reihenfolge der beiden Aktionen ist.

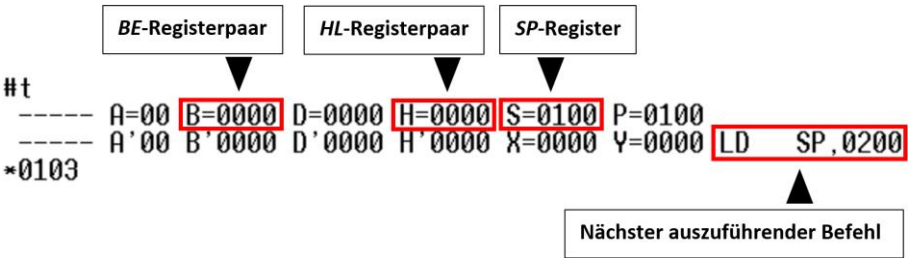
1. Das Holen des Wertepaares ***17f2h*** über *pop* vom Stack
2. Den Stackpointer SP um den Wert 2 erhöhen



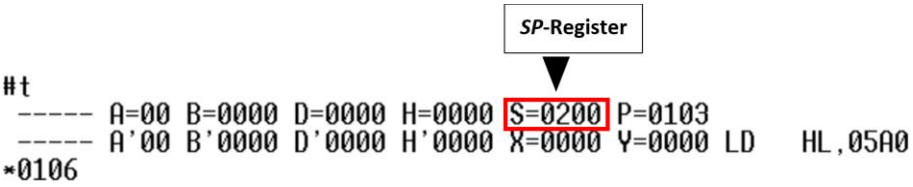
Nun ist die Theorie erst einmal abgehandelt, doch erfolgt dies auch in dieser Weise in der Praxis? Wir wollen das mit dem *zsid*-Programm überprüfen. Der Aufruf erfolgt natürlich wieder mit der Angabe der *stack1.com*-Datei. Zuerst zeige ich im Anschluss über den Befehl ***L*** für *List* den Quellcode an. Ich nutze dazu die Speichergrenze ***0100h*** und ***010dh***, wobei die nachfolgende Adresse ***010eh*** dennoch zur Anzeige gebracht wird.

```
C1>zsid stack1.com
ZSID VERS 1.5
NEXT  PC  END
0180 0100 DAFF
#l
  0100  LD   SP,0200
  0103  LD   HL,05A0
  0106  LD   BC,17F2
  0109  PUSH BC
  010A  PUSH HL
  010B  LD   HL,0000
  010E  LD   BC,0000
  0111  POP  HL
  0112  POP  BC
  0113  RET
  0114  LD   A,(DE)
#
```

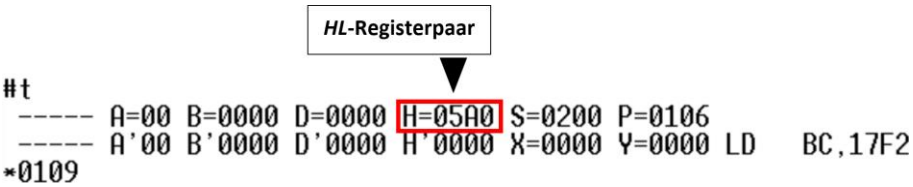
Das Debugging führe ich mit dem Kommando ***t*** für *trace* durch, das nach und nach eingegeben wird, um darüber die Befehle und die Registerinhalte anzuzeigen. Das erste ***t*** fördert folgendes zutage.



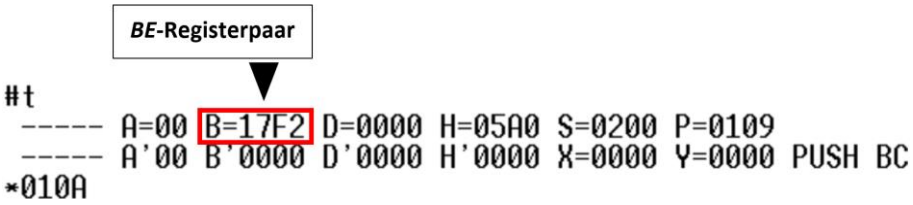
Die Anzeige ***LD SP, 0200*** auf der rechten Seite, zeigt immer den nächsten auszuführenden Befehl an. Starten wir also die schrittweise Ausführung über weitere Eingaben von ***t***.



Es ist zu erkennen, dass der zuvor angekündigte Befehl ausgeführt wurde und der Stackpointer *SP*, der in der Anzeige lediglich mit *S* gekennzeichnet ist, korrekt mit dem Wert ***0200h*** initialisiert wurde. Es steht der nächste Befehl an, der über ***LD HL, 05A0*** das Registerpaar *hl* initialisieren soll. Geben wir wieder ein *t* ein.

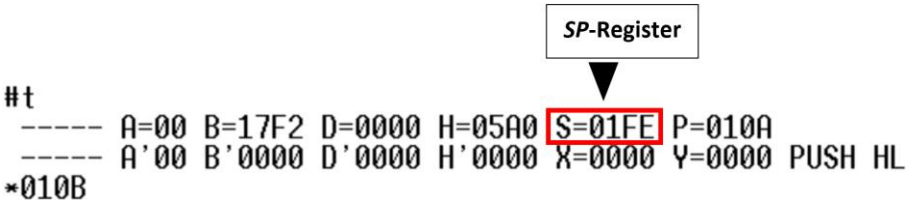


Wir sehen, dass der Befehl ***LD HL, 05A0*** das Registerpaar *hl* korrekt initialisiert hat. Ähnliches soll mit dem Registerpaar *bc* passieren, was über den Befehl ***LD BC, 17F2*** angekündigt wird. Fahren wir mit der Eingabe von *t* fort.

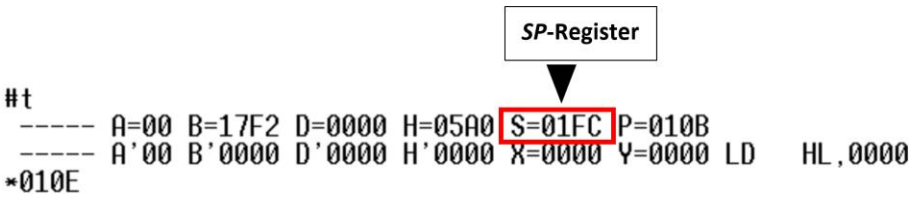


Wir können erkennen, dass der Befehl ***LD BC, 17F2*** das Registerpaar *bc* korrekt initialisiert hat.

Nun geht kommen wir zu den Stack-Operationen. Zuerst soll das Registerpaar *bc* über den Befehl ***PUSH BC*** auf den Stack geschoben werden. Wir geben wieder den Befehl *t* ein, um diese Aktion auszuführen. Wir erinnern uns, dass der Stackpointer bei einem *push* für 16-Bit um den Wert 2 vermindert wird. Sehen wir nach.



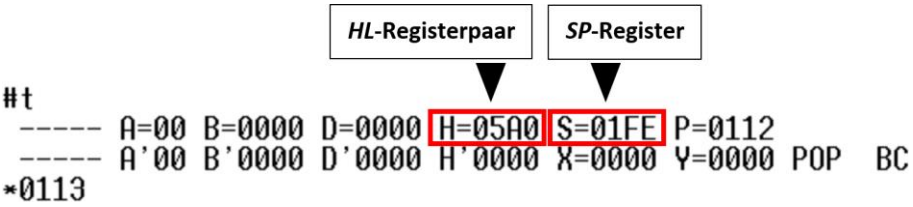
Ok, der Wert des Stackpointers ist von vormals ***0200h*** auf ***01FEh*** gewechselt, was *0200h - 2* entspricht. Der nächste anstehende Befehl lautet ***PUSH HL***, um darüber das Registerpaar *hl* auf den Stack zu schieben. Wir geben wieder den Befehl *t* ein, um diese Aktion auszuführen.



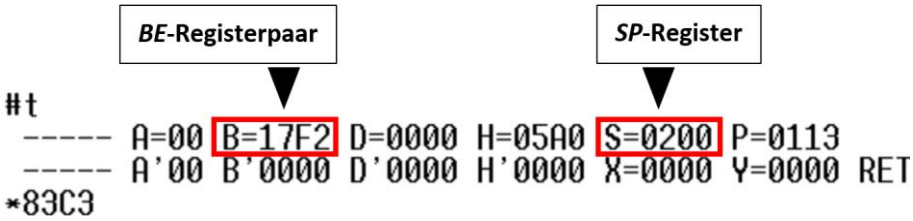
Der Wert des Stackpointers ist von vormals ***01FEh*** auf ***01FCh*** gewechselt, was *01FEh - 2* entspricht. Um zu sehen, ob das Zurückholen vom Stack auch funktioniert, werden die beiden Registerpaare *hl* und *bc* gelöscht und mit den Werten *0000h* versehen. Die Befehle dafür lauten ***LD HL, 0000h*** und ***LD BC, 0000h***.

```
#t
----- A=00 B=17F2 D=0000 H=0000 S=01FC P=010E
----- A'00 B'0000 D'0000 H'0000 X=0000 Y=0000 LD   BC,0000
*0111
#t
----- A=00 B=0000 D=0000 H=0000 S=01FC P=0111
----- A'00 B'0000 D'0000 H'0000 X=0000 Y=0000 POP  HL
*0112
```

Jetzt gehen wir den umgekehrten Weg und holen die beiden Registerpaar wieder vom Stack herunter. Der nächste anstehende Befehl lautet ***POP HL*** und holt das Registerpaar *hl* vom Stack und initialisiert darüber wieder das Registerpaar mit den abgelegten Werten auf dem Stack. Wir geben wieder den Befehl *t* ein, um diese Aktion auszuführen. Wir erinnern uns, dass der Stackpointer bei einem *pop* für 16-Bit um den Wert 2 erhöht wird. Sehen wir nach.



Der Wert des Stackpointers ist von vormals ***01FCh*** auf ***01FEh*** gewechselt, was *01FCh + 2* entspricht. Der Inhalt des Registerpaares wurde mit dem Wert ***05A0h*** vom Stack geladen. Der nächste anstehende Befehl lautet ***POP BC*** und holt das Registerpaar *bc* vom Stack und initialisiert darüber wieder das Registerpaar mit den abgelegten Werten vom Stack. Wir geben wieder den Befehl *t* ein, um diese Aktion auszuführen.



Der Wert des Stackpointers ist von vormals ***01FEh*** auf ***0200h*** gewechselt, was *01FEh + 2* und dem Ausgangswert entspricht. Das Registerpaar *bc* hat wieder den Wert ***17F2h***. Der nächste auszuführende Befehl wäre ***RET***, was der *Return*-Anweisung entspricht und die Kontrolle an die Konsole zurückgibt.

2.12.5 Eine Schleife programmieren

Eine Schleife zu programmieren, ist in Assembler nicht weiter schwer. Ich nutze zu diesem Zweck ein spezielles 8-Bit Register, was den Schleifendurchlauf auf 255 begrenzt, was aber im Moment nicht weiter interessieren soll. Der Z80 hat eine spezielle Anweisung zur Implementierung von Schleifenzählungen. Die *DJNZ*-Anweisung (*Decrement B-Register and Jump if Not Zero*) steht für

„B-Register dekrementieren und springen, wenn nicht Null“

Das geht in diesem Fall nur mit dem *B-Register*, was also das Register unserer Wahl ist, um eine Schleife zu implementieren. Hier ein entsprechendes MBASIC-Programm.

```
10 FOR I=8 TO 1 STEP -1
20 PRINT ".";
30 NEXT I
Ok
run
........
Ok
```

Eine in Basic programmierte *FOR-NEXT*-Schleife muss also in diesem Fall rückwärts implementiert werden, da das Register bis zum Wert 0 herunterzählt. Wenn man das in Flussdiagramm überträgt, dann würde sich das wie folgt gestalten.

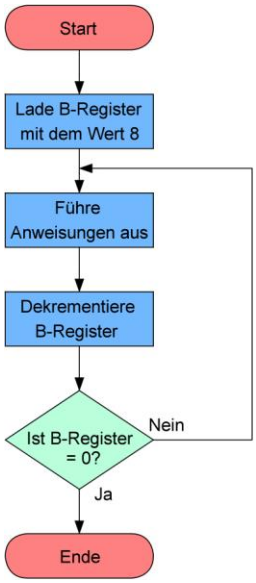


Abbildung 33 - Das Flussdiagramm für eine Schleife

Werden wir nun konkret und schauen uns den Quellcode des Programms *out4.zsm* an, wobei ein *Label* eingesetzt und das als Sprungmarkierung verwendet wird.

```
te: OUT4.ZSM                          --- | Lin:0010/0010/0512 Col:33/76 Len:32
   !...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...
  1|      org 0100h  ; free memory
  2|BDOS  equ 0005h  ; BDOS entry point
  3|      ld b,8     ; loop 8x
  4|LOOP: push bc    ; bc to stack
  5|      ld c,2     ; BIOS function print char
  6|      ld e,'.'   ; load char
  7|      call BDOS  ; CALL BDOS
  8|      pop bc     ; bc from stack
  9|      djnz LOOP  ; decrease B and loop
 10|      ret        ; return to CCP
```

Es ist hier zu sehen, dass in der Zeile 4 das Registerpaar ***BC*** vor dem BDOS-Aufruf über die ***push***-Anweisung auf den Stack gelegt und damit gesichert wird. In Zeile 7 erfolgt dann der BDOS-Aufruf und in Zeile 8 wird durch die ***pop***-Anweisung das Registerpaar ***BC*** wieder von Stack geholt und steht für einen erneuten BDOS-Aufruf mit den ursprünglichen Daten zur Verfügung.

Wird das Programm assembliert, in eine COM-Datei überführt und ausgeführt, zeigt sich, ob das auch wirklich funktioniert hat mit dem 8-fachen Schleifendurchlauf. Es müssten also 8 Punkte zu sehen sein.

```
C1>out4
........
C1>
```

Das müssten 8 Punkte sein und das Programm funktioniert also mit dem 8-fachen Aufruf der BDOS-Funktion zur Anzeige eines einzelnen Zeichens. Wenn der Schleifenwert auf den Maximalwert von 255 gesetzt wird, kann man sehen, wie schnell die Ausgabe erfolgt.

2.12.6 Eine einfache Addition

Im nächsten Beispiel soll etwas gerechnet werden, denn dazu ist ja der schon erwähnte Akkumulator ***A*** eigentlich da. Ich sollte noch etwas detaillierter auf die Funktion eines Akkumulators eingehen. Ein Akkumulator ist ein Register innerhalb einer CPU - hier natürlich der Z80 -, in die Ergebnisse der Recheneinheit gespeichert werden. Ohne einen Akkumulator wäre es notwendig, jedes Ergebnis einer Berechnung, sei es Addition, Multiplikation oder auch logische Verknüpfung, im Hauptspeicher, also RAM, zu speichern und später von dort wieder in den Z80 zu laden. Der Zugriff auf das interne Register - genannt *Akkumulator* - ist aber wesentlich schneller als ein Zugriff auf das RAM, da dieses Register mit der ***ALU*** (Englisch: Arithmetic Logic Unit) in der CPU integriert ist. Auf der folgenden Abbildung ist der vereinfachte Aufbau des Akkumulators im Zusammenspiel mit dem Datenbus und der ***ALU*** (*Arithmetic Logic Unit*) zu sehen.

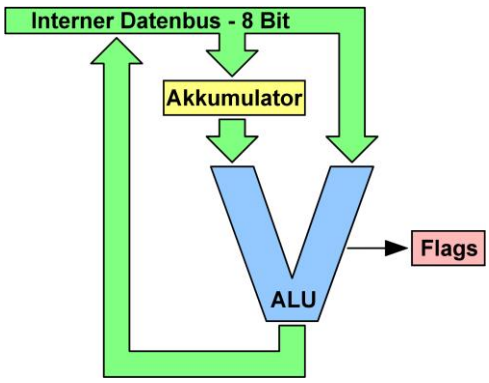
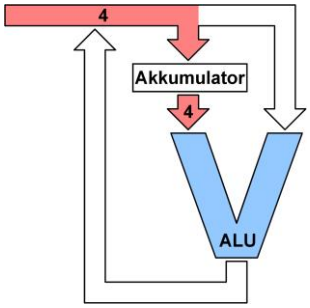


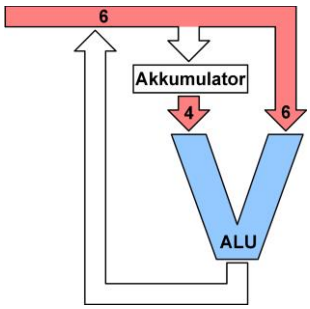
Abbildung 34 - Der Akkumulator im Zusammenspiel mit der ALU

Es ist zu sehen, dass nicht im Akkumulator die verschiedenen Rechen- beziehungsweise Logik-Operationen durchgeführt werden, sondern in der *ALU*. Der Akkumulator arbeitet als spezielles Register, das der ALU den ersten Wert für eine Operation zuliefert. Der zweite Wert wird über den internen Datenbus bereitgestellt. Der Akkumulator kann bei einem Befehl sowohl Ein- als auch Ausgabe einer Operation sein. Die meisten Befehle des Z80 beeinflussen ein oder mehrere Flags, die Aufschluss über bestimmte Ergebnisse liefern. Ich komme noch darauf zu sprechen. Spielen wir den Ablauf einer anstehenden Addition der Werte 4 und 6 einmal durch.

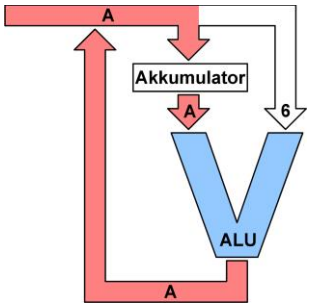
Im ersten Schritt wird der Wert 4 über den internen Datenbus mittels eines *Akkumulator-Lade-Befehls* in den Akkumulator geladen und versorgt darüber den linken Arm der ALU mit dem ersten Operanden.



Im zweiten Schritt wird der Wert 6 in den rechten Arm der ALU geladen.



Nun stehen die beiden Operanden ***4*** und ***6*** in der ALU bereit, so dass die eigentliche Addition durchgeführt werden kann. Das Ergebnis ***A*** in hexadezimaler Schreibweise wird auf den internen Bus gelegt und steht wieder zum Abruf oder für weitere Operationen im Akkumulator zur Verführung.



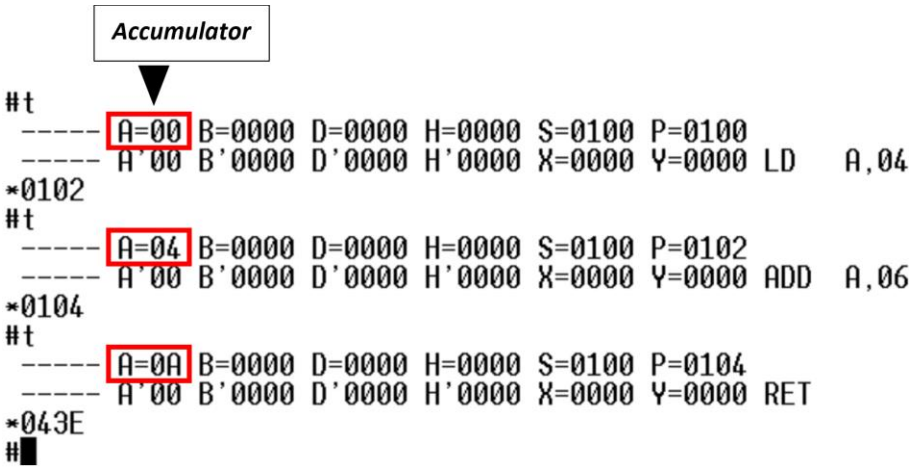
Sehen wir uns die entsprechende Umsetzung in Assembler an. Die Angabe der Werte bezieht sich auf reine Dezimalzahlen. Sollen Hexadezimalzahlen verwendet werden, muss eine führende Null und das *h* - ähnlich wie in Zeile 1 - angehängt werden. Für den hexadezimalen Wert *a3* müsste man also die Schreibweise *0a3h* verwenden. Der Quellcode von *add1.zsm* dazu schaut wie folgt aus.

```
te: ADD1.ZSM                          --- | Lin:0004/0004/0512 Col:32/76 Len:31
    !...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...
  1|    org 0100h    ; Free memeory
  2|    ld a, 4      ; Load Accumulator
  3|    add 6        ; Add #6
  4|    ret          ; return to CCP
```

Um einen Wert mit dem Akkumulator zu addieren, wird der befehl ***ADD*** und der Angabe des entsprechenden Wertes verwendet. Um zu sehen, wie sich das Ergebnis beziehungsweise die einzelnen Schritte gestalten, wird wieder *zsid* mit der Angabe der COM-Datei bemüht, die bei mir natürlich *add1.com* lautet.

```
C1>zsid add1.com
ZSID VERS 1.5
NEXT  PC  END
0180 0100 DAFF
#l100,104
  0100  LD   A,04
  0102  ADD  A,06
  0104  RET
  0105
#
```

Sehen wir uns wieder die schrittweise Ausführung des Codes an.



Es ist zu sehen, dass im ersten Schritt der Inhalt des Akkumulators noch den Wert 0 enthält. Nach der Ausführung des Ladebefehls ***LD A, 04*** wurde der Wert *04* in den Akkumulator geladen. Der nächste Schritt führt dann eine Addition über ***ADD A, 06*** mit dem Wert *06* durch, was dazu führt, dass der Akkumulator den Wert dieser Addition beinhaltet und ***0A*** ist. Somit ist die Addition erfolgreich durchgeführt worden, denn 4 + 6 = 10 und der Wert 10 ist in hexadezimaler Schreibweise eben *0A*. Bevor ich mit den sogenannten Flags weitermache, sollte ich ein paar Beispiele mit unterschiedlichen Literalen, also möglichen Zeichenfolgen, zeigen, die im Assembler möglich sind.

```
te: LITERAL.ZSM                          --- | Lin:0006/0006/0512 Col:36/76 Len:35
    !...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...
  1|    org 0100h        ; free memory
  2|    ld a, 65         ; load a with 41h
  3|    ld a, 10011100b  ; load a with 9Ch
  4|    ld a, 0f3h       ; load a with f3h
  5|    ld a, 'a'        ; load a with 61h
  6|    ret              ; return to CCP
```

Gehen wir das Ganze Zeile für Zeile durch und überprüfen es am Ende mit *zsid*.

Zeile 2

Der Ladebefehl ***ld*** für den Accumulator wird mit dem dezimalen Wert *65* ausgeführt. Wird dieser Wert in eine hexadezimale Zahl gewandelt, so ergibt dies *41h*.

Zeile 3

Der Ladebefehl ***ld*** für den Accumulator wird jetzt mit der Binärzahl *10011100* durchgeführt. Um dies dem Assembler mitzuteilen, muss am Ende ein *b* angehängt

werden. Die Umwandlung des binären Wertes in eine hexadezimale Zahl lautet *9Ch*.

Zeile 4

Der Ladebefehl ***ld*** für den Accumulator wird mit einer Hexadezimalzahl durchgeführt, der *f3* lautet. Um dies dem Assembler mitzuteilen, muss vor dem Wert eine *0* und am Ende ein *h* gesetzt werden.

Zeile 5

Der Ladebefehl ***ld*** für den Accumulator wird mit einem einzelnen Zeichen (Character) durchgeführt, das *a* lautet. Um dies dem Assembler mitzuteilen, muss das Zeichen in einfache Hochkommata eingeschlossen werden. Der ASCII-Wert von 'a' ist laut ASCII-Tabelle der hexadezimale Wert 61h.

Sehen wir uns nun zur Überprüfung das Ergebnis mit *zsid* an.

```
C1>zsid literal.com
ZSID VERS 1.5
NEXT  PC  END
0180 0100 DAFF
#l100,108
  0100  LD   A,41
  0102  LD   A,9C
  0104  LD   A,F3
  0106  LD   A,61
  0108  RET
  0109
#
```

Das Ganze ist jedoch nicht nur für 8 Bit-Register möglich, sondern auch für Doppelregister. Sehen wir uns das am Beispiel des Doppelregisters HL an.

```
te: LITERALS.ZSM                        --- | Lin:0006/0006/0512 Col:08/76 Len:07
    !...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...
  1|    org 0100h                ; free memory
  2|    ld hl, 100011110001b     ; load hl with 8F1h
  3|    ld hl, 111100001b        ; load hl with 1E1h
  4|    ld hl, 65535             ; load hl with FFFFh
  5|    ld hl, 0B1Eh             ; load hl with 0B1Eh
  6|    ret
```

Und nun wieder die Überprüfung mit *zsid*.

```
C1>zsid literals.com
ZSID VERS 1.5
NEXT  PC  END
0180 0100 DAFF
#t
 ----- A=00 B=0000 D=0000 H=0000 S=0100 P=0100
 ----- A'00 B'0000 D'0000 H'0000 X=0000 Y=0000 LD   HL,08F1
*0103
#t
 ----- A=00 B=0000 D=0000 H=08F1 S=0100 P=0103
 ----- A'00 B'0000 D'0000 H'0000 X=0000 Y=0000 LD   HL,01E1
*0106
#t
 ----- A=00 B=0000 D=0000 H=01E1 S=0100 P=0106
 ----- A'00 B'0000 D'0000 H'0000 X=0000 Y=0000 LD   HL,FFFF
*0109
#t
 ----- A=00 B=0000 D=0000 H=FFFF S=0100 P=0109
 ----- A'00 B'0000 D'0000 H'0000 X=0000 Y=0000 LD   HL,0B1E
*010C
#t
 ----- A=00 B=0000 D=0000 H=0B1E S=0100 P=010C
 ----- A'00 B'0000 D'0000 H'0000 X=0000 Y=0000 RET
*F121
```

Doch nun zu den angekündigten Flags. Diese Flags sind in einem speziellen Statusregister innerhalb des Rechenwerks eines Mikroprozessors gespeichert. Da diese einzelnen Bits auch als Flags bezeichnet werden, wird das Statusregister auch *Flag-Register* genannt. Es enthält eine Reihe von Bits, die von der arithmetisch-logischen Einheit (ALU) in Abhängigkeit von der zuletzt durchgeführten Rechenoperation gesetzt werden. Das können zum Beispiel ein Überlauf oder ein negatives Ergebnis sein. Doch lenken wir unser Augenmerk zuerst auf das sogenannte *Carry-Bit*. Das Carry-Bit (auch *Übertrags-Bit* genannt) ist ein spezieller Begriff aus der Informatik. Er bezeichnet ein Bit, welches den Übertrag einer Addition oder Subtraktion von Bits auf das nächst höherwertige Bit repräsentiert. Das hört sich sehr geschwollen an und doch ist es recht logisch! Der Akkumulator besitzt eine Breite von 8 Bits, was bedeutet, dass der höchste Wert, der dort gespeichert werden kann, 255 beträgt. Was passiert aber, wenn zum dem Wert 255 der Wert 1 addiert wird? Das Ergebnis wäre rein rechnerisch natürlich der Wert 256, der jedoch nicht innerhalb von 8-Bits gespeichert werden kann. Es wäre ein weiteres Bit notwendig, das aber nicht zur Verfügung steht! Um diesen Umstand zu signalisieren, ist das *Carry-Bit* verantwortlich! Dieser sogenannte *Überlauf* darf bei einer durchgeführten Rechnung nicht unberücksichtigt bleiben. Auf der folgenden Abbildung ist der Umstand zu sehen.

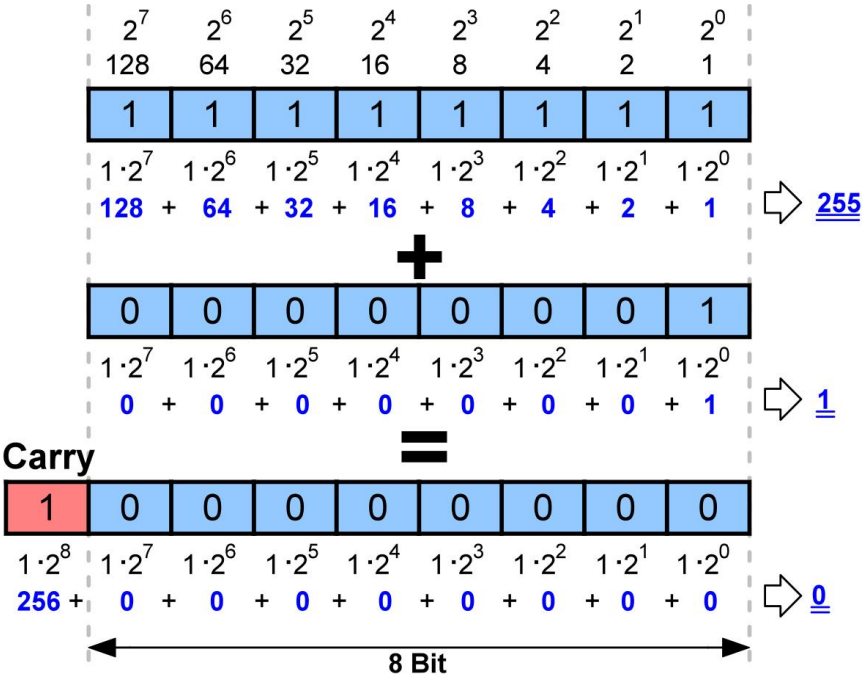


Abbildung 35 - Addition mit Überlauf

Sehen wir uns das an einem konkreten Beispiel an, bei dem der Wert 255 um den Wert 1 erhöht werden soll. Der Quellcode von *add2.zsm* schaut dabei wie folgt aus.

```
te: ADD2.ZSM                          --- | Lin:0004/0004/0512 Col:32/76 Len:31
    !...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!
  1|    org 0100h    ; Free memeory
  2|    ld a, 255    ; Load Accumulator #255
  3|    add 1        ; Add #1
  4|    ret          ; return to CCP
```

Das Ergebnis der durchgeführten Addition würde also 256 lauten, was aber nicht innerhalb von 8 Bits abgespeichert werden kann, denn 8-Bits können maximal einen Wert von 255 speichern. Da kommt das Carry-Bit ins Spiel, was signalisiert, dass hier bei der durchgeführten Rechenoperation etwas hinsichtlich der 8-Bits nicht stimmt! In der folgenden Abbildung sind die Flags zu sehen, wo natürlich auch das Carry-Bit zu finden ist.

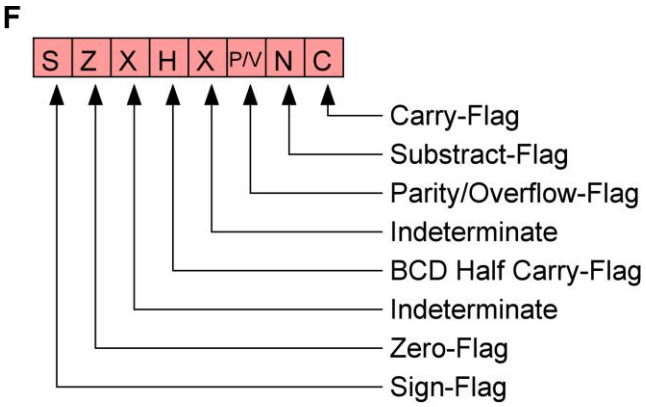
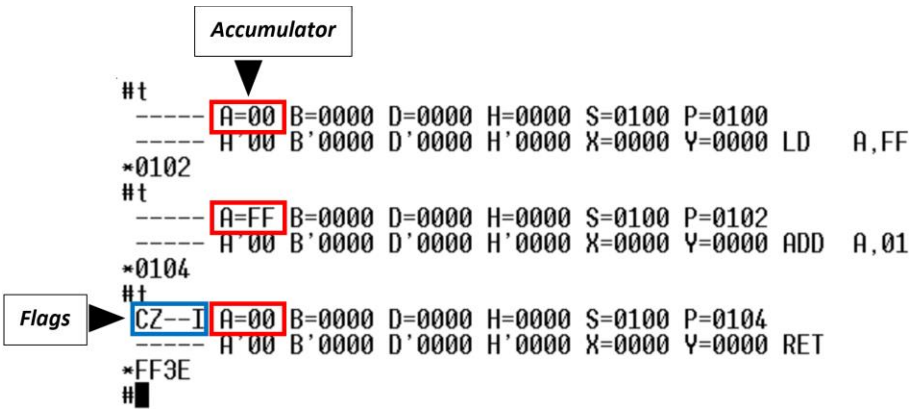


Abbildung 36 - Die Z80-Flags

Um zu sehen, wie sich das Ergebnis beziehungsweise die einzelnen Schritte gestalten, werde ich zu Beginn wieder *zsid* mit der Angabe der COM-Datei bemühen, die bei mir *add2.com* lautet.

```
C1>zsid add2.com
ZSID VERS 1.5
NEXT  PC  END
0180 0100 DAFF
#1100,104
  0100  LD   A,FF
  0102  ADD  A,01
  0104  RET
  0105
#
```

Wenn die Ausführung wieder schrittweise durchgeführt wird, ist Folgendes zu sehen.



Der Inhalt des Accumulators beträgt zu Beginn den Wert *00h*. Dann wird er mit seinem Maximalwert ***FFh*** initialisiert und im Anschluss eine Addition mit dem Wert ***01h*** durchgeführt. Als Ergebnis ist der Accumulator wieder mit dem

Wert ***00h*** versehen, denn wenn es zu einem Überlauf kommt, beginnt die Zählung wieder von vorne bei ***00h***.

Nun sollten wir uns die Ausgabezeilen für *trace* einmal hinsichtlich der möglichen Buchstaben etwas genauer anschauen. Die Angaben haben dabei die folgenden Bedeutungen.

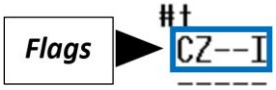
Buchstabe	Bedeutung
C	Carry - Übertragsbit
Z	Zero - Nullbit
M	Minus - Vorzeichenbit
E	Even-Parity - Paritätsbit
I	Interdigit Carry - Hilfsübertragsbit

Tabelle 8 - Die Bedeutung der Statusbits

Es ist nach der durchgeführten Addition sehr gut zu sehen, dass das Carry-Bit gesetzt wurde. Der Inhalt des Akkumulators A änderte sich von ***FFh*** nach ***00h***, was aus den genannten Gründen ja verständlich ist.

Die vier Bindestriche am Anfang der geben dabei den Status der Flags an. Wenn sie zu sehen sind und kein weiterer Buchstabe in Erscheinung tritt, bedeutet das, dass keines der Flags gesetzt ist. Das ist in der ersten und zweiten Zeile der Ausgaben von *zsid* zu sehen.

In der dritten Zeile von *zsid*, schaut es anders aus.



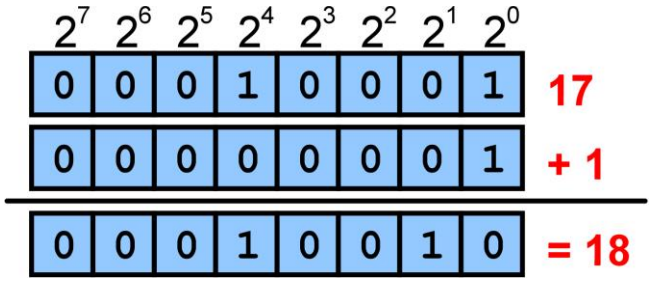
Hier ist zu sehen, dass einige Flags gesetzt wurden, was an den erscheinenden Buchstaben zu erkennen ist. Das für uns wichtige Flag ist das ***C*** ganz links außen, was darauf hindeutet, dass ein Übertrag stattgefunden hat.

2.12.7 Flag-Details

Die grundlegende Funktion beziehungsweise Aufgabe der Flags sollte in groben Zügen klar sein. Dennoch möchte ich mithilfe einiger Beispiele zeigen, wie diese Flags beeinflusst werden. Sehen wir uns dazu den folgenden Code an.

```
te: FLAGS.ZSM                          --- | Lin:0008/0008/0512 Col:32/76 Len:31
   !...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...
  1|    org 0100h    ; free memory
  2|    ld a, 17     ; load a with #17
  3|    add a, 1     ; add #1
  4|    ld a, 127    ; load a with #127
  5|    add a, 1     ; add #1
  6|    ld a, 255    ; load a with #255
  7|    add a, 1     ; add #1
  8|    ret          ; return to CCP
```

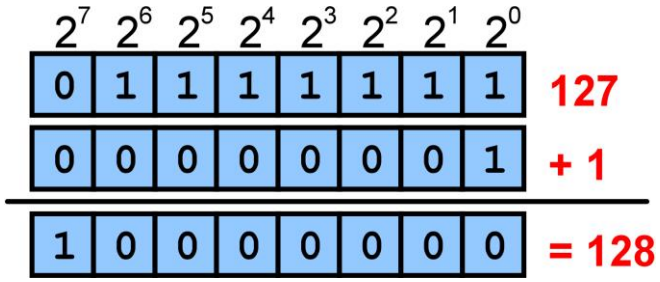
Wir fangen mit den Zeilen 2 und 3 an und sehen, welche Auswirkungen diese auf die Flags haben. Dazu werden wir uns die ganze Sache auf Bit-Ebene anschauen.



Das Ergebnis der Addition von 17 und 1 ist recht unspektakulär 18. Wie schaut es dann bei der Ausführung dieser beiden Programmschritte hinsichtlich der Flags aus? Sehen wir mit *zsid* nach. Es ist zu sehen, dass sich nach dieser Addition bei den Flags nichts getan hat und sie alle gelöscht sind.

```
#t
 ----- A=00 B=0000 D=0000 H=0000 S=0100 P=0100
 ----- A'00 B'0000 D'0000 H'0000 X=0000 Y=0000 LD   A,11
*0102
#t
 ----- A=11 B=0000 D=0000 H=0000 S=0100 P=0102
 ----- A'00 B'0000 D'0000 H'0000 X=0000 Y=0000 ADD  A,01
*0104
#t
 ----- A=12 B=0000 D=0000 H=0000 S=0100 P=0104
 ----- A'00 B'0000 D'0000 H'0000 X=0000 Y=0000 LD   A,7F
*0106
#
```

Kommen wir zur zweiten Addition, die wie folgt ausschaut.



Es ist nichts Ungewöhnliches hierbei zu entdecken, doch nach der Ausführung der beiden Zeilen 4 und 5 hat sich bei den Flags etwas getan.

```
 ----- A=7F B=0000 D=0000 H=0000 S=0100 P=0106
 ----- A'00 B'0000 D'0000 H'0000 X=0000 Y=0000 ADD  A,01
*0108
#t
 --MEI A=80 B=0000 D=0000 H=0000 S=0100 P=0108
 ----- A'00 B'0000 D'0000 H'0000 X=0000 Y=0000 LD   A,FF
*010A
#
```

Es ist zu sehen, dass im markierten Bereich die folgenden Flags gesetzt sind:

- *M*: Minus
- *E*: Even Parity
- *I*: Hilfs-Carry

Das für mich wichtige Flag ist das *Minus*-Flag, was besagt, dass 128 ein negativer Wert sein soll. Kann doch irgendwie nicht wahr sein, oder?! Und doch ist das natürlich korrekt. Ich sollte aber bei diesem Aspekt etwas weiter vorne anfangen, denn ansonsten fehlen zum Verständnis wichtige Grundlagen.

Da mit Binärzahlen - und ich bleibe bei einer Datenbreite von 8 Bits - Werte von 0 bis 255 dargestellt werden können, handelt es sich dabei naturgemäß um positive Werte. Nun gibt es aber auch negative Werte, die in irgendeiner Weise gekennzeichnet werden müssen. Da wir es aber lediglich mit 8 Bits zu tun haben, die uns zur Verfügung stehen und kein weiteres Bit quasi als Vorzeichenbit vorhanden ist, müssen wir einen anderen Weg einschlagen. Man hat sich dafür entschieden, ein bestimmtes Bit als Vorzeichenbit per Definition zu verwenden. Diese Bit ist das höchstwertige Bit, was MSB (Most-Significant-Bit) bezeichnet wird.

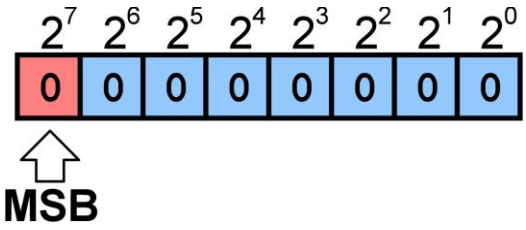


Abbildung 37 - Das MSB als Vorzeichenbit

Dabei ist folgendes zu beachten:

- MSB = 0: positiver Wert
- MSB = 1: negativer Wert

Es liegt in der Natur der Sache, dass für die Abbildung des eigentlichen Wertes ein Bit weniger zur Speicherung zur Verfügung steht. Wie groß ist dann aber der zur Verfügung stehende Bereich, wenn zu den positiven Werten auch noch negative hinzukommen? Bei 8-Bits schaut das dann wie folgt aus.

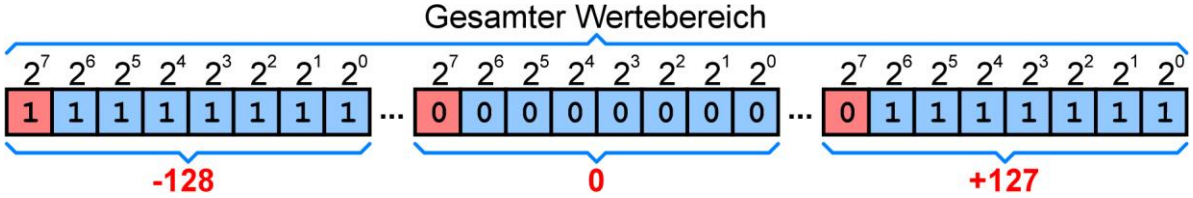


Abbildung 38 - Gesamter Wertebereich mit negativen Werten

Ich muss noch einmal betonen, dass es eine Interpretationssache ist, wie eine Bitkombination zu werten ist. Wird lediglich mit positiven Werten gearbeitet, dass ist der Wert 128 in der gezeigten Bitkombination 10000000 vollkommen richtig. Das *Minus*-Flag wird dann einfach ignoriert. Doch wenn es darum geht, zusätzlich noch mit negativen Werten zu arbeiten, schaut die Welt vollkommen anders aus.

Es sollte jetzt verständlich sein, warum bei dieser Bitkombination das Minus-Flag zur Markierung als negativer Wert gesetzt wurde. Sehen wir us da am Beispiel einer anderen Bitkombination an die 10000110 lautet und dem dezimalen Wert 134 entspricht. Die Interpretation als negativer Wert ist vollkommen anders, als man das vielleicht vermuten mag und entspricht ***nicht***

– wie man das vielleicht meinen möchte – dem Wert -6! Warum ist das so? Und warum entspricht aber die Bitkombination links außen 11111111 dem dezimalen Wert -128 und wie kommt man dahin?

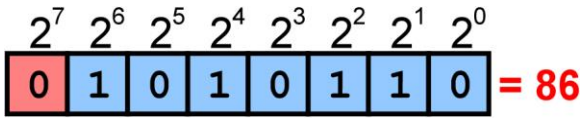
Es gibt zur Darstellung von negativen Werten das sogenannte *Einerkomplement* und das *Zweierkomplement*. Hört sich geschwollen an, ist aber recht einfach!

	Was ist das Einerkomplement?
Das Einerkomplement ist eine arithmetische Operation im Binärsystem. Bei dieser Vorgehensweise werden alle Bits einer Binärzahl invertiert. Aus 0 wird 1 und aus 1 wird 0.	

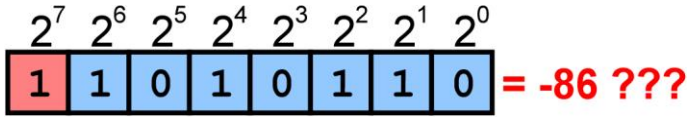
Und dann...

	Was ist das Zweierkomplement?
Das Zweierkomplement stellt eine Möglichkeit dar, negative Zahlen im Binärsystem darzustellen, ohne die Angabe von zusätzlichen Zeichen wie + und – zu verwenden, da sich diese Information innerhalb der zur Verfügung stehenden Bitkombination verbirgt. Das Zweierkomplement wird aus dem Einerkomplement + 1 gebildet.	

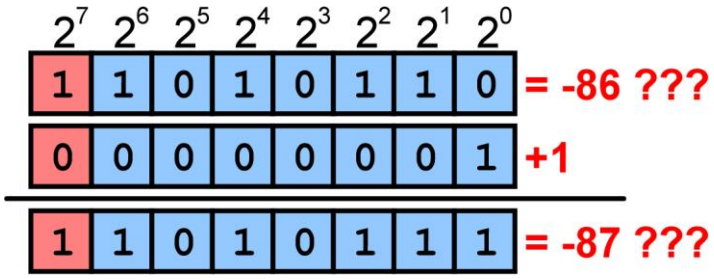
Warum muss das aber auf diese Weise geschehen? Dazu muss ich wieder ein wenig ausholen. Angenommen, wir haben die folgende Bitkombination vorliegen.



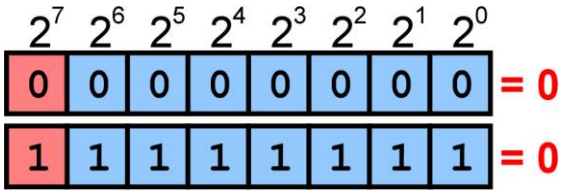
Diese positive Zahl (das MSB ist 0) stellt einen dezimalen Wert von +86 dar. OK, dann wollen wir daraus eine negative Zahl machen, indem wir lediglich das Vorzeichen-Bit von 0 auf 1 ändern. Das vermeintlich richtige Ergebnis wäre das folgende.



Wenn wir an dieser Stelle die Festlegung treffen würden, dass das der negative Wert der eben gezeigten positiven Zahl ist, wäre zunächst alles OK. Damit könnten wir leben. Doch in der Datenverarbeitung werden nicht nur Werte gespeichert und angezeigt. Es wird auch mit ihnen gerechnet. Und da laufen wir in ein Problem hinein. Wir nehmen einmal an, wir wollten einen Wert, sagen wir +1, addieren, was bedeutet, dass das Ergebnis um den Wert 1 größer wird als der Ursprungswert. Sehen wir uns das wieder auf Bit-Ebene an.



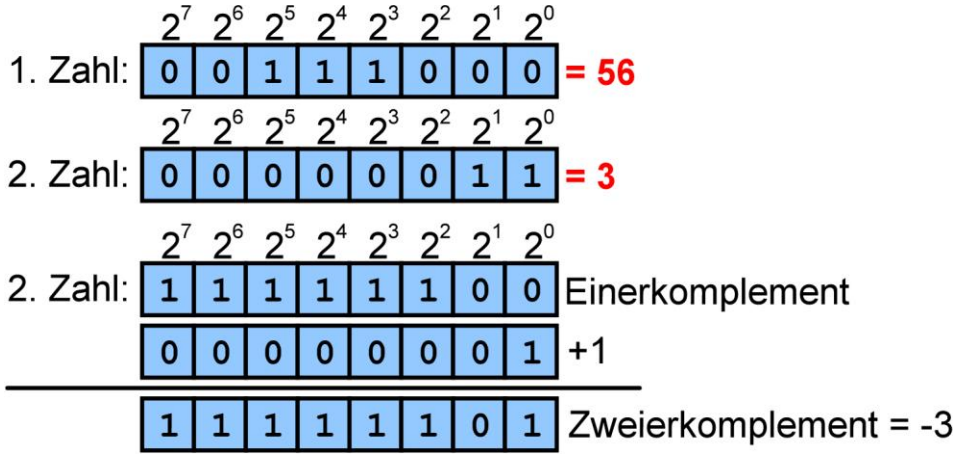
Na, fällt hier etwas auf? Trotz Addition eines positiven Wertes ist das Ergebnis um den Wert 1 kleiner geworden. -86 + 1 = -87 ??? Auf diese Art und Weise kommen wir also nicht zum Ziel. Jetzt wenden wir das eben angepriesene Einerkomplement auf den Ursprungswert an. Ich werde dabei auch direkt auf das nächste Problem hinweisen, das sich bei einer ganz besonderen Zahl ergibt. Von jedem Wert kann ich das negative Pendant bilden, in dem ich ein negatives Vorzeichen davorsetze, so auch bei der Zahl 0. Aber 0 und -0 sind absolut identisch und es besteht kein arithmetischer Unterschied.



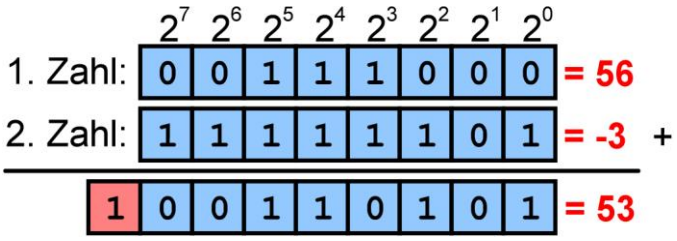
Das kann so nicht akzeptiert werden, da die Eindeutigkeit nicht gewährleistet ist. Aus diesem Grund wird der Wert 1 addiert, was in Summe die Zweierkomplementbildung ergibt. Dann wollen wir doch einmal eine einfache Subtraktion zuerst händisch und dann über die CPU durchführen, um zu sehen, ob da auch das gleiche Ergebnis herauskommt. Ich möchte die Differenz von 56 und 3 bilden. Es gilt:

Ergebnis = *Minuend* minus *Subtrahend* gleich Wert der Differenz

Um das händisch durchzuführen, kann man die Summe von beiden Werten bilden, doch zuvor muss vom Subtrahenden das Zweierkomplement gebildet werden. Das würde dann wie folgt ausschauen. Der Binärwert des Minuenden 56 steht ganz oben und das Zweierkomplement von 3 wird darunter gebildet.



Also muss nun lediglich die Summe von 00111000 (56) und 11111101 (-3) gebildet werden. Der entstandene Übertrag wird dabei nicht berücksichtigt!



Das Ergebnis lautet binär korrekterweise 00110101, was dezimal +53 entspricht. Abschließend zum Zweierkomplement zeige ich einmal den Zahlenkreis für 4-Bit-Werte, in dem sowohl die positiven, als auch die negativen Werte aufgetragen sind.

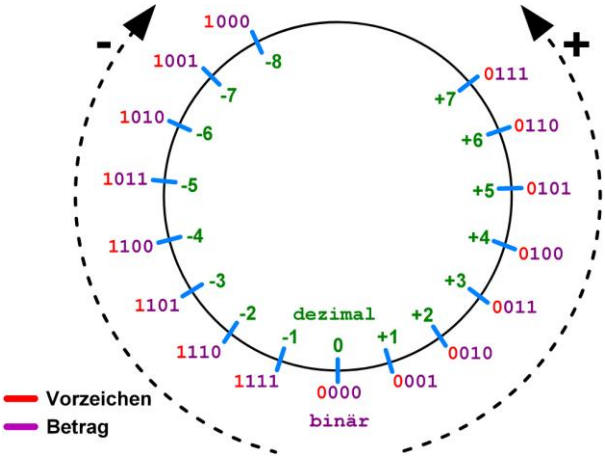
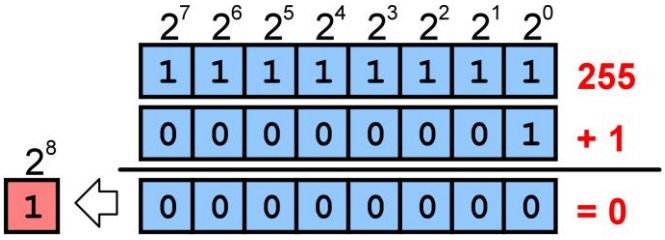


Abbildung 39 - Der Zahlenkreis für positive und negative Werte

Gehen wir weiter im Programm vor und führen die nächsten beiden Zeilen 6 und 7 aus. Nach dem Laden des Accumulators mit dem Wert 255 und der Addition mit 1 schaut es wie folgt aus.



Es ist zu erkennen, dass es zu einem Übertrag (Carry) gekommen ist und der Wert im Accumulator demnach wieder mit 0 versehen ist. Führen wir die schrittweise Programmausführung fort.

```
#t
 --MEI A=80 B=0000 D=0000 H=0000 S=0100 P=0108
 ----- A'00 B'0000 D'0000 H'0000 X=0000 Y=0000 LD   A,FF
*010A
#t
 --MEI A=FF B=0000 D=0000 H=0000 S=0100 P=010A
 ----- A'00 B'0000 D'0000 H'0000 X=0000 Y=0000 ADD  A,01
*010C
#t
 CZ--I A=00 B=0000 D=0000 H=0000 S=0100 P=010C
 ----- A'00 B'0000 D'0000 H'0000 X=0000 Y=0000 RET
*113E
#
```

Wir sehen, dass im markierten Bereich die folgenden Flags gesetzt sind:

- *C*: Übertrag
- *Z*: Zero
- *I*: Hilfs-Carry

Die beiden ersten Flags sind wieder wichtig, denn *C* wurde wegen des auftretenden Übertrags gesetzt und *Z* zeigt an, dass der Wert im Accumulator 0 ist.

2.12.8 Einfache Subtraktionen

Kommen wir zur entgegengesetzten Rechenoperation zur Addition. Es handelt sich um die Subtraktion. Natürlich beherrscht der Z80 auch das *Abziehen* von Werten. Sehen wir uns dazu das folgende recht einfache Beispiel an und ich habe damit gewartet, bis wir uns ein bisschen Hintergrundwissen bezüglich der Flags angeeignet haben.

```
te: SUB1.ZSM                              --- | Lin:0004/0004/0512 Col:32/76 Len:31
    !...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...
  1|    org 100h     ; Free memory
  2|    ld a, 3      ; Load Accumulator
  3|    sub 3        ; Sub #3
  4|    ret          ; Return to CPP
```

Natürlich läuft auch bei der Subtraktion wieder alles über den Akkumulator. Um einen Wert vom Akkumulator zu subtrahieren, wird der Befehl ***SUB*** und der Angabe des entsprechenden Wertes verwendet. Um zu sehen, wie sich das Ergebnis beziehungsweise die einzelnen Schritte gestalten, wird wieder *zsid* mit der Angabe der COM-Datei bemüht, die bei mir *sub1.com* lautet.

```
C1>zsid sub1.com
ZSID VERS 1.5
NEXT  PC  END
0180 0100 DAFF
#t
 ----- A=00 B=0000 D=0000 H=0000 S=0100 P=0100
 ----- A'00 B'0000 D'0000 H'0000 X=0000 Y=0000 LD   A,03
*0102
#t
 ----- A=03 B=0000 D=0000 H=0000 S=0100 P=0102
 ----- A'00 B'0000 D'0000 H'0000 X=0000 Y=0000 SUB  03
*0104
#t
Flags ▶ -Z--- A=00 B=0000 D=0000 H=0000 S=0100 P=0104
 ----- A'00 B'0000 D'0000 H'0000 X=0000 Y=0000 RET
*033E
#
```

Es ist hier nach der Ausführung des Befehls ***SUB 03*** zu sehen, dass das *Z-Flag* gesetzt wurde. Es deutet darauf hin, dass das Ergebnis der Subtraktion den Wert 0 ergeben hat, was auch im Akkumulator zu sehen ist. Die Abkürzung *Z* steht für *Zero*, was bekanntermaßen Null bedeutet.

Sehen wir uns ein weiteres Beispiel für eine Subtraktion an. Hier wird nun ein größerer Wert von einem kleineren abgezogen, was zu einem negativen Ergebnis führt.

```
te: SUB2.ZSM                              --- | Lin:0004/0004/0512 Col:32/76 Len:31
    !...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...
  1|    org 100h     ; Free memory
  2|    ld a, 3      ; Load Accumulator
  3|    sub 4        ; Sub #4
  4|    ret          ; Return to CPP
```

Um zu sehen, wie sich das Ergebnis beziehungsweise die einzelnen Schritte gestalten, wird wieder *zsid* mit der Angabe der COM-Datei bemüht, die bei mir *sub2.com* lautet.

```
C1>zsid sub2.com
ZSID VERS 1.5
NEXT  PC  END
0180 0100 DAFF
#t
 ----- A=00 B=0000 D=0000 H=0000 S=0100 P=0100
 ----- A'00 B'0000 D'0000 H'0000 X=0000 Y=0000 LD   A,03
*0102
#t
 ----- A=03 B=0000 D=0000 H=0000 S=0100 P=0102
 ----- A'00 B'0000 D'0000 H'0000 X=0000 Y=0000 SUB  04
*0104
#t
 C-M-I A=FF B=0000 D=0000 H=0000 S=0100 P=0104
 ----- A'00 B'0000 D'0000 H'0000 X=0000 Y=0000 RET
*033E
#
```

Flags

Es ist zu sehen, das unter anderem das M-Flag gesetzt wurde, was für Minus steht und auf ein negatives Ergebnis hindeutet. Interessant ist auch die Betrachtung des Akkumulatorinhaltes, der jetzt mit einem ***FF*** aufwartet. Warum ist das so? Nun, da der Akkumulator lediglich 8 Bits im Bereich von *00* bis *FF* verarbeiten kann, beginnt die rückwärtige Zählung wieder mit dem Maximalwert FF.

2.12.9 Verschiedene Adressierungsarten

Wie die meisten Prozessoren besitzt auch die Z80-CPU über unterschiedliche Möglichkeiten, wie auf Daten hinsichtlich einer Operation zugegriffen wird. Bisher haben wir bei durchgeführten Additionen oder Subtraktionen immer mit konstanten Werten gearbeitet. Zum Akkumulatorinhalt wird der konstante Wert 3 addiert oder vom Akkumulator wird der konstante Wert *4* abgezogen. Nun ist es aber sicherlich auch wichtig, dass nicht mit konstanten Werten gerechnet wird, sondern mit Werten, die sich in Speicherstellen befinden, deren Inhalten variabel sind. Der beiden genannten Varianten liegen unterschiedliche Adressierungsarten zugrunde. Wird mit einer Konstanten gearbeitet, dann handelt es sich um eine ***unmittelbare Adressierung***. Soll mit dem Inhalt einer Speicherstelle gearbeitet werden, so haben wir es mit einer ***absoluten Adressierung*** zu tun. Sehen wir uns da wieder im Detail an. Im nachfolgenden Programm kommen beide Adressierungsarten zur Anwendung. In Zeile 2 wird der konstante Wert 4 über die unmittelbare Adressierung in den Akkumulator geladen, wogegen in Zeile 3 über die absolute Adressierung der Inhalt aus Speicherstelle 4 in den Akkumulator geladen wird. Es ist zu sehen, dass der Wert 4 in runden Klammern geschrieben wird.

```
te: ADDRESS.ZSM                          --- | Lin:0004/0004/0512 Col:08/76 Len:07
   !...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...
 1|    org 100h  ; Free memory
 2|    ld a, 4   ; Unmittelbar (Immediate)
 3|    ld a, (4) ; Absolut     (Absolute)
 4|    ret
```

Schauen wir uns das Ergebnis wieder mit *zsid* an.

```
C1>zsid address.com
ZSID VERS 1.5
NEXT  PC  END
0180 0100 DAFF
#t
 ----- A=00 B=0000 D=0000 H=0000 S=0100 P=0100
 ----- A'00 B'0000 D'0000 H'0000 X=0000 Y=0000 LD   A,04
*0102
#t
 ----- A=04 B=0000 D=0000 H=0000 S=0100 P=0102
 ----- A'00 B'0000 D'0000 H'0000 X=0000 Y=0000 LD   A,(0004)
*0105
#t
 ----- A=12 B=0000 D=0000 H=0000 S=0100 P=0105
 ----- A'00 B'0000 D'0000 H'0000 X=0000 Y=0000 RET
*043E
#
```

Bei der unmittelbaren Adressierung wird der Wert 12h in den Akkumulator geladen und wir können diesen Wert über einen Dump mit der Eingabe von ***d3*** aus zsid heraus überprüfen.

```
#d0
0000: C3 03 FE 3D 12 C3 00 DB 00 00 00 00 00 00 00 00 ...=............
0010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

Ich denke, wir sollten diesbezüglich noch ein paar mehr Worte über den Unterschied der beiden Adressierungsarten verlieren. Werfen wir doch einmal einen Blick in die ***PRN***-Datei, die vom Assembler mit generiert wird.

```
C1>type address.prn

ZSM-3.4    Source file: ADDRESS    Page No:    1

0100 =              org 100h   ; Free memory
0100 3E04           ld a, 4    ; Unmittelbar (Immediate)
0102 3A0400         ld a, (4)  ; Absolut     (Absolute)
0105 C9             ret
0100 =

Errors    0

Next address: 0106H
```

Für die unmittelbare Adressierung schauen die Bytes wie folgt aus.

3E 04

Der Op-Code ist demnach *3E* und das Literal *04* und es handelt sich um einen 2-Byte-Befehl. Hinsichtlich der absoluten Adressierung sind die folgenden Bytes zu sehen.

3A 04 00

Der Op-Code ist hier *3A* und die Adresse *04 00*, wobei das LSB zuerst genannt wird. Wir können sehen, dass es sich jetzt um einen 3-Byte-Befehl handelt, denn die Adresse muss sich über den gesamten 16-Bit Adressbereich erstrecken können. Auf der folgenden Abbildung sind die beiden Adressierungsarten im Vergleich zu sehen.

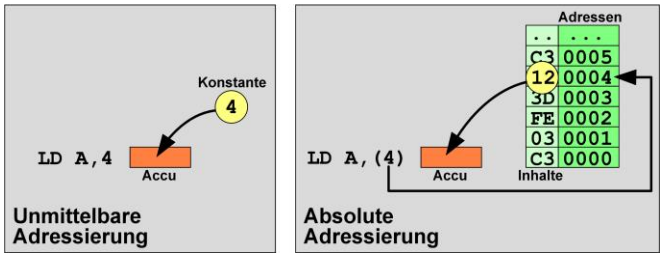


Abbildung 40 - Die unmittelbare und absolute Adressierung

Natürlich gibt es noch weitere Adressierungsmöglichkeiten, doch ich verweise auf die Fachliteratur. Die nachfolgende Liste zeigt alle verfügbaren Adressierungsarten kurz auf.

- Implizite Adressierung
- Unmittelbare Adressierung
- Absolute Adressierung
- Zero-Page Adressierung
- Relative Adressierung
- Indizierte Adressierung
- Direkte und indirekte Adressierung

2.12.10 Noch mehr über Register

Die internen Register - klingt wie ein weißer Schimmel -, denn Register sind immer intern, haben folgende Struktur, auf die ich eigentlich noch überhaupt nicht eingegangen bin.

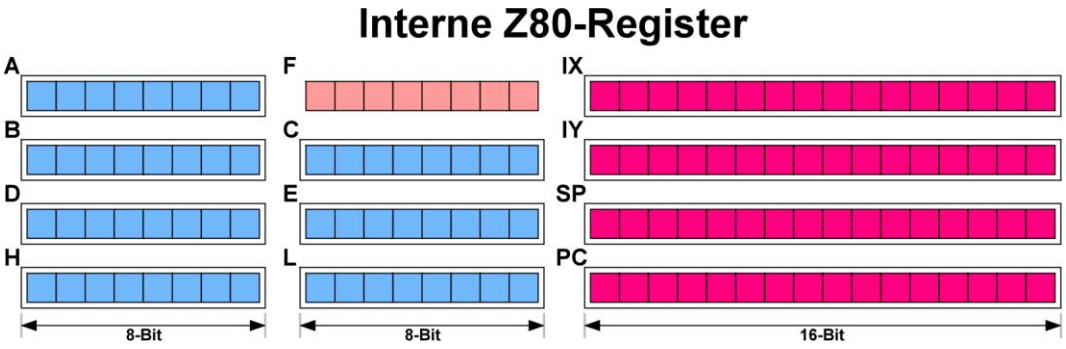
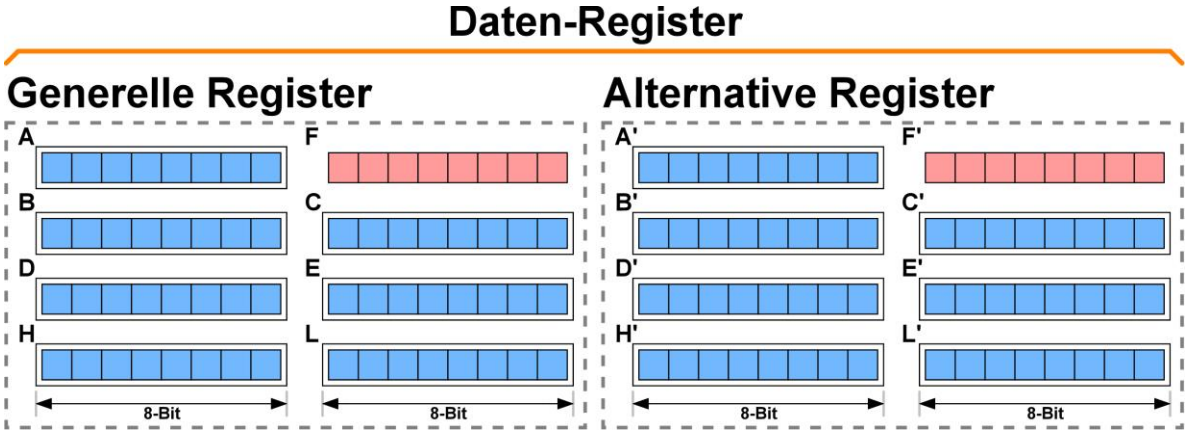


Abbildung 41 - Die Z-80-Register

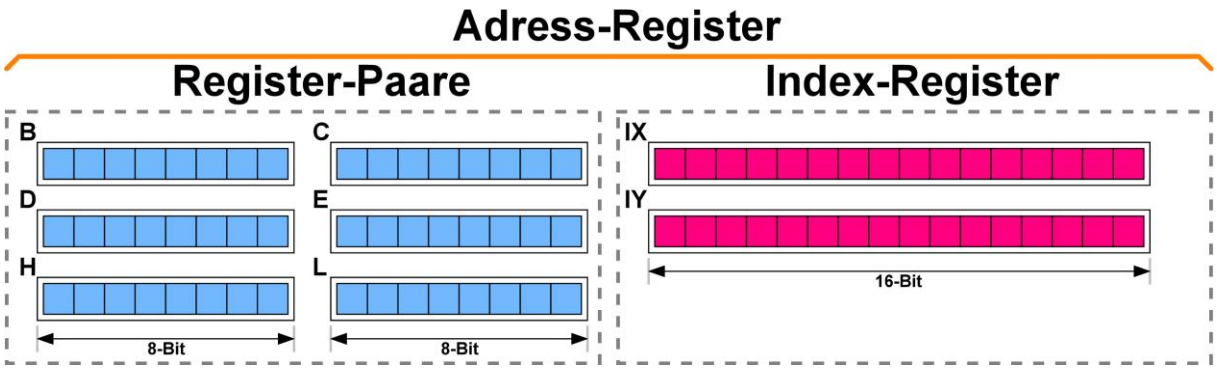
Das Wichtigste, was wir verstehen müssen, ist, was ein Register ist und wie man es benutzt. Ein Register ist ein 16-Bit-Speicherplatz, der aus 2 Bytes besteht und sich in der CPU selbst befindet. In der Z80 haben wir die folgenden wichtigen Register beziehungsweise Registerpaare:

- AF
- BC
- DE
- HL
- IX
- IY

Nachfolgend ist die Aufteilung der Daten-Register zu sehen, die je 8 Bit breit sind. Die gezeigten Register sind in der Z80-CPU als Doppelregister angelegt und besitzen in den Bezeichnungen einen Strich neben dem Namen. Diese zusätzlichen Register werden *alternative Register* genannt. Die meisten Operationen der CPU funktionieren nur auf dem ersten Satz, und die alternativen Register bieten nur vorübergehenden Platz, um den Inhalt von *A* bis *L* zu speichern.



Der Z80 enthält zwei Arten von 16-Bit-Adressierungsregistern. *Registerpaare* und *Indexregister*. Um die Adressierung hinsichtlich des 16 Bit-Busses zu gewährleisten, können Register zu Registerpaaren (2x8-Bit) zusammengefasst oder die beiden Index-Register ***IX*** beziehungsweise ***IY*** genutzt werden, die eine Breite von 16 Bits besitzen.



Von diesen 8-Bit-Registern ist ***A*** (der Accumulator) das wichtigste, wie wir das schon gesehen haben. Das Register *A* ist mit dem Register *F* gruppiert, wobei F steht für Flags steht, was wir ebenfalls schon kennengelernt haben.

Beim Z80 sind die Register 1 Byte breit (8 Bits), können jedoch zu Paaren zusammengefasst werden, wie ich das gerade gezeigt habe, so dass sie dann zwei Bytes oder ein *Wort* (Word) enthalten. Diese Register werden als Registerpaare bezeichnet, da es sich um zwei Register handelt, die zusammengefügt werden. Diese Paare können nur in gezeigter Kombination gebildet werden. Also zum Beispiel *AF*, *BC*, *DE* oder *HL*. Kombinationen wie *AB*, *BH* sind nicht möglich.

Neben dem Akkumulator *A*, der am häufigsten verwendet wird, kommt *HL* als Adressregisterpaar hauptsächlich für die Referenzierung von Speicheradressen zur Anwendung. Sehen wir uns da direkt an einem kleinen Beispiel an. Das

Registerpaar *HL* soll also auf eine Stelle im Speicher weisen, um dort einen Wert abzulegen, der sich im Accumulator befindet.

```
te: HL.ZSM                              --- | Lin:0005/0005/0512 Col:36/76 Len:35
    !...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...
  1|    org 0100h       ; free memory
  2|    ld a, 0abh      ; load a with ab
  3|    ld hl, 20a1h    ; load hl with 20a1
  4|    ld (hl), a      ; ld 20a1 with ab
  5|    ret             ; return to CCP
```

In Zeile 2 wird der Accumulator mit dem hexadezimalen Wert ***ab*** geladen. Damit das HL-Registerpaar auf die genannte Speicherstelle *20a1* zeigen soll, wird dieses in Zeile 3 mit diesem Wert geladen. Zeile 4 ist jetzt interessant, denn das Registerpaar *HL* steht dort in runden Klammern, was bedeutet, dass nicht das Registerpaar mit dem Wert des Accumulators geladen wird, sondern die Speicherstelle, auf die das Registerpaar *HL* weist, und das ist hier die Adresse *20a1*. Sehen wir uns das genauer mit *zsid* an.

```
C1>zsid hl.com
ZSID VERS 1.5
NEXT  PC  END
0180 0100 DAFF
#l100,106
  0100  LD   A,AB
  0102  LD   HL,20A1
  0105  LD   (HL),A
  0106  RET
  0107
#
```

Führen wir jetzt die einzelnen Schritte über Trace *t* aus und sehen uns die Auswirkungen an.

```
C1>zsid hl.com
ZSID VERS 1.5
NEXT  PC  END
0180 0100 DAFF
#t
 ----- A=00 B=0000 D=0000 H=0000 S=0100 P=0100
 ----- A'00 B'0000 D'0000 H'0000 X=0000 Y=0000 LD   A,AB
*0102
#t
 -----[A=AB] B=0000 D=0000 H=0000 S=0100 P=0102
 ----- A'00 B'0000 D'0000 H'0000 X=0000 Y=0000 LD   HL,20A1
*0105
#t
 ----- A=AB B=0000 D=0000[H=20A1] S=0100 P=0105
 ----- A'00 B'0000 D'0000 H'0000 X=0000 Y=0000 LD   [(HL),A]
*0106
#t
 ----- A=AB B=0000 D=0000 H=20A1 S=0100 P=0106
 ----- A'00 B'0000 D'0000 H'0000 X=0000 Y=0000 RET
*AB3E
#
```

Es ist zu sehen, wie die Register A und HL mit den Werten geladen werden. Wie können wir aber überprüfen, ob sich an Speicherstelle *20a1* der Wert *ab* befindet? Natürlich über einen Dump mit der Eingabe von *d* und der Angabe des Speicherbereiches. Kennen wir diese Adressierungsart schon?

```
#d20a1,20a1
20A1: AB .
#
```

Und siehe da, es ist der Wert ab zu sehen. Man sollte natürlich sehr vorsichtig mit der Speicherung eines Wertes sein, um nicht in einen Bereich zu gelangen, der das Betriebssystem zerstört!

2.13 Die Pinansteuerung am Board

Kommen wir zu einem weiteren interessanten Thema, um auf verschiedene Pins am verwendeten Board zuzugreifen. Das kann natürlich zum Beispiel das Arduino Due oder das Teensy 3.6-Board sein, aber auch jedes andere, dass unterstützt wird. Ich zeige es anhand meines Teensy 3.6-Boards. Doch zuvor sollte ich auf ein paar Grundlagen eingehen, die zum Verständnis der Ansteuerung beitragen. Da wir es mit der Programmierung der Boards über die Arduino-Entwicklungsumgebung zu tun haben, werde ich die Abfrage beziehungsweise Ansteuerung der Pins auf dieser Plattform etwas beschreiben. Auf der folgenden Abbildung ist das Standardmodell *Arduino-Uno* zu sehen, an dem ich die vom Mikrocontroller nach außen geführten Pins zeigen möchte.

Abbildung 42 - Der Arduino-Uno

Auf dieser kleinen Platine befinden sich in dieser Ausrichtung am oberen und unteren Rand Buchsenleisten, die *Header* genannt werden. Diese ermöglichen einem, direkt auf die Anschlüsse des Mikrocontrollers Zugriff zu nehmen. Es gibt dabei zwei grundsätzlich unterschiedliche Arten von Pins.

- Digitale Pins
- Analoge Pins

Die digitalen Pins können sowohl als Eingänge, als auch als Ausgänge konfiguriert werden, wobei die analogen Pins nur als Eingänge zur Verfügung stehen. Ich möchte mit den digitalen Pins beginnen.

2.13.1 Digitale Pins

Ein digitaler Pin muss vor seiner Nutzung konfiguriert werden, damit diese entweder als Ein- oder als Ausgang zur Verfügung steht. Sehen wir uns einen entsprechenden *Sketch*, also den Code zur Programmierung innerhalb der Arduino-Entwicklungsumgebung an.

```
01 int led = 13; // LED-Pin
02
03 void setup() {
04   pinMode(led, OUTPUT);     // Konfiguration als Ausgang
05   digitalWrite(led, HIGH); // LED an
06 }
07
08 void loop() {/* ... */}
```

Listing 1 - Arduino-Sketch zur LED-Ansteuerung

In Zeile 1 wird eine Variable definiert, die auf den anzusteuernden digitalen Pin weist. Nun kommt es innerhalb der *setup*-Funktion, die beim Arduino einmalig ausgeführt wird, einerseits zur Konfiguration des Pins mittels der *pinMode*-Funktion in Zeile 04. Die erforderlichen Argumente sind die Pin-Nummer und die Datenflussrichtung, was in unserem Fall *OUTPUT* ist, denn der Arduino soll die LED ansteuern. Andererseits wird in Zeile 05 die *digitalWrite*-Funktion aufgerufen, die ebenfalls zwei Argumente erwartet. Zum einen ist das natürlich wieder die Nummer des Pins und zum anderen der Pegel. Ein *HIGH*-Pegel lässt die LED leuchten, wogegen ein *LOW*-Pegel die LED verlöschen lässt.

Die Umsetzung über *RunCPM* erfolgt nach einem ähnlichen Schema. Die Unterstützung für digitales und analoges Lesen/Schreiben auf der Arduino-Plattform wurde von *Krzysztof Kliś* durch zusätzliche, nicht standardisierte BDOS-Aufrufe hinzugefügt. Wer sich die näheren Informationen anschauen möchte, der sollte sich das RunCPM-Projekt in die Arduino-Entwicklungsumgebung laden, den Tab-Reiter ***cpm.h*** öffnen und ganz nach unten scrollen. Dort befinden sich die entsprechenden Definitionen der BDSOS-Aufrufe.

Die Umsetzung von pinMode

```
01 ld c, 220
02 ld d, <pin>  ; Arduino-Pin-Nummer
03 ld e, <mode> ; 0 = INPUT / 1 = OUTPUT
04 call BDOS    ; Aufruf von BDOS
```

Die Umsetzung von digitalWrite

```
01 ld c, 222
02 ld d, <pin>   ; Arduino-Pin-Nummer
03 ld e, <value> ; 0 = LOW / 1 = HIGH
04 call BDOS     ; Aufruf von BDOS
```

Um jetzt einen digitalen Pin anzusteuern, müssen wir eines noch bedenken, was schon beim Stack und der Programmierung einer Schleife zur Sprache gekommen ist. Während des BDOS-Aufrufs können Registerinhalte verändert werden und so ist es sicherlich ratsam, diese vorher auf den Stack zu sichern. Das folgende Programm lässt die LED an Pin 12 aufleuchten.

```
te: pin1.zsm                          --- | Lin:0011/0011/0512 Col:32/76 Len:31
   !...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...
 1|     org 0100h  ; free memory
 2|BDOS equ 0005h  ; BDOS entry point
 3|     ld c, 220  ; pinMode
 4|     ld d, 12   ; digital pin number
 5|     ld e, 1    ; mode (0 = INPUT, 1 = OUTPUT)
 6|     push de    ; Save register
 7|     call BDOS  ; CALL BDOS
 8|     pop de     ; Restore register
 9|     ld c, 222  ; digitalWrite
10|     call BDOS  ; BDOS entry point
11|     ret        ; return to CCP
```

Hier ist zu sehen, dass hinter dem Aufruf von *digitalWrite* in Zeile 9 nicht noch einmal das *d*- beziehungsweise *e*-Register geladen werden müssen. Diese

Informationen wurden über die Zeile 6 mit ***PUSH DE*** auf den Stack gesichert und in der Zeile 8 nach dem BDOS-Aufruf über ***POP DE*** wieder zurückgeschrieben. Auf den folgenden Abbildungen ist der recht einfache Schaltplan und der entsprechende Schaltungsaufbau mit dem Teensy-Board und einer LED zu sehen.

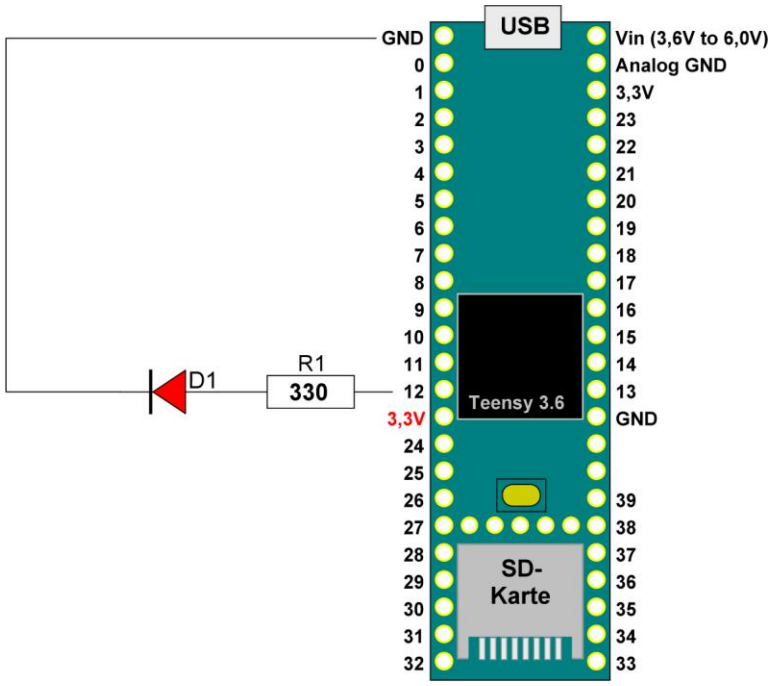


Abbildung 43 - Der Schaltplan zur Ansteuerung einer LED

Und hier der Schaltungsaufbau. Es sei zu erwähnen, dass sich das Teensy-Board hier im Gegensatz zur Ausrichtung auf dem Schaltplan um 180 Grad gedreht auf dem Breadboard aufgesteckt ist.

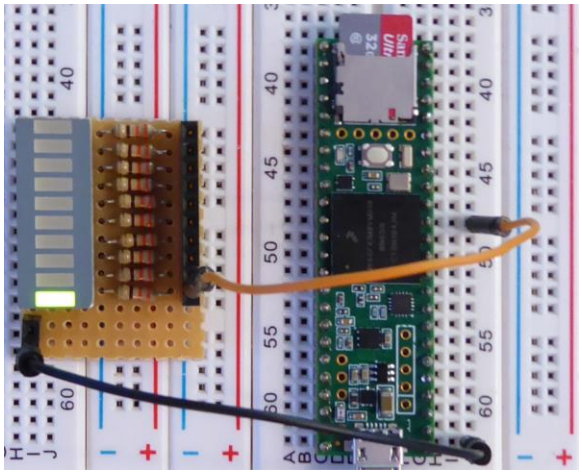


Abbildung 44 - Der Schaltungsaufbau zur Ansteuerung einer LED

Um die richtigen Pins ausfindig zu machen, sollte man natürlich die Pinbelegung (das sogenannte *Pinout*) des verwendeten Boards kennen. Unter der folgenden Adresse ist unter anderem die Pinbelegung des *Teensy 3.6* zu finden.

https://www.pjrc.com/teensy/pinout.html

Falls das *Arduino-Due*-Board zum Einsatz kommt, ist hier der entsprechende Link zu nutzen.

https://docs.arduino.cc/hardware/due

Im nächsten Schritt wollen wir einen digitalen Pin in Abhängigkeit des Status eines anderen Pins ansteuern. Pin 12 bleibt dabei weiterhin unser Ausgangspin und Pin 11 wird der Eingangspin. Sehen wir uns dazu die erforderliche Schaltung an.

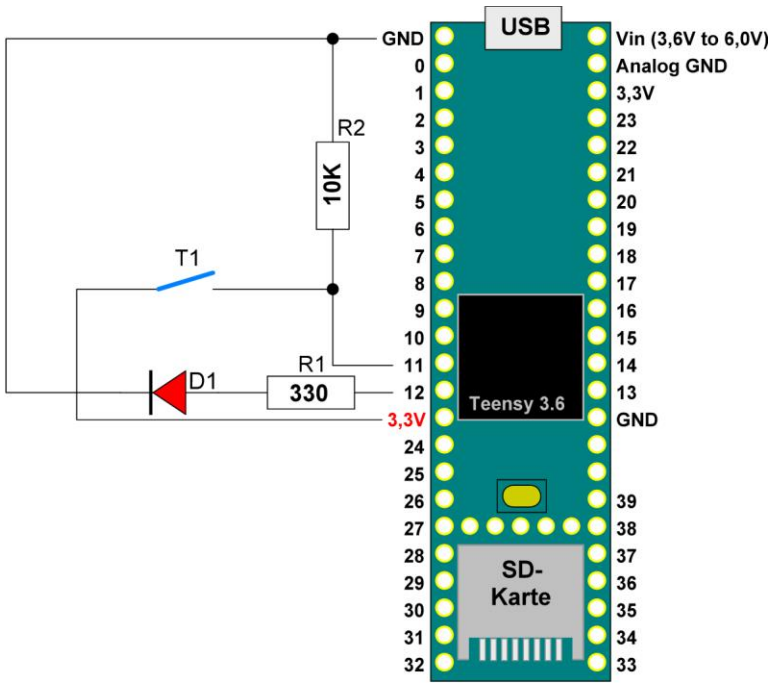


Abbildung 45 - Der Schaltplan zur Abfrage des Tasters und Ansteuerung der LED

Kommen wir nun zum Programmcode, in dem alle drei Elemente zur Anwendung kommen.

- pinMode
- digitalWrite
- digitalRead

Die erforderliche Kodierung für *pinMode* und *digitalWrite* haben wir schon gesehen. Fehlt nur noch die für das *digitalRead*.

Die Umsetzung von digitalRead

```
01 ld c, 221
02 ld d, <pin> ; Arduino-Pin-Nummer
03 call BDOS   ; Aufruf von BDOS -> Ergebnis im A (0 = LOW / 1 = HIGH)
```

Wir sehen, dass sich das Ergebnis des BDOS-Aufrufs im Accumulator befindet. Gehen wir wieder schrittweise vor.

```
te: PIN3.ZSM                          --- | Lin:0015/0028/0512 Col:01/76 Len:05
  1|     org 0100h  ; free memory
  2|BDOS equ 0005h  ; BDOS entry point
  3|PINO equ 12     ; Output-Pin
  4|PINI equ 11     ; Input-Pin
  5|
  6|     ld c, 220  ; pinMode
  7|     ld d, PINO ; digital pin number (OUTPUT-PIN)
  8|     ld e, 1    ; mode (1 = OUTPUT)
  9|     call BDOS  ; call BDOS
 10|
 11|     ld c, 220  ; pinMode
 12|     ld d, PINI ; digital pin number (INPUT-PIN)
 13|     ld e, 0    ; mode (0 = INPUT)
 14|     call BDOS  ; CALL BDOS
 15|
```

Zur besseren Übersicht habe ich die beiden Pins zur Ein- und Ausgabe in den Zeile 3 und 4 über ***EQU*** mit Namen versehen. Der Output-Pin heißt jetzt ***PINO*** und der Input-Pin ***PINI***. Der Output-Pin ***PINO*** wird in den Zeilen 6 bis 9 - wie wir das vorher schon einmal gemacht haben -, entsprechend konfiguriert. Der Input-Pin ***PINI*** erfährt in den Zeilen 11 bis 14 seine Konfiguration. Diese Konfigurationen unterscheiden sich lediglich im Wert, der dem *E*-Register (0 = INPUT, 1 = OUTPUT) zugewiesen wird.

Damit der Taster, der sich am Input-Pin befindet, auch kontinuierlich abgefragt wird, ist eine Schleife erforderlich, die mit einem sogenannten Jump-Befehl (Sprung-Anweisung) realisiert wird. In der Z80-Programmierung gibt es eine Reihe unterschiedlicher Jump-Befehle, die die unterschiedlichsten Funktionen haben. Alle hier aufzuzählen würde den Rahmen etwas sprengen. Als wir eine Schleife programmiert haben, kam zum ersten Mal der JUMP-Befehl ***DJNZ*** zum Einsatz. Wir erinnern uns: Durch diesen einen Befehl wird das Register *B* verringert und ein bedingter Sprung durchgeführt, wenn das Ergebnis nicht Null ist. Für unser Vorhaben benötigen wir einen Sprungbefehl, der das *Z-Flag* - also das Zero-Flag - auswertet, denn wenn den Taster an Pin 11 nicht gedrückt ist, befindet sich der Wert 0 im Accumulator, was bedeutet, dass wir mit einem Trick diese Flag auswerten können. Warum sage ich Trick? Nun, das Flag-Register wird nicht immer aktualisiert, sondern nur bei bestimmten Aktionen, wie zum Beispiel bei arithmetischen Operationen wie *ADC* (Add with Carry). Addieren wir also den Wert 0 zum Inhalt der Accumulators, bleibt sein Inhalt unverändert, doch die Flags werden auf den neuesten Stand gebracht. Sehen wir uns also jetzt den nächsten Teil des Codes an.

```
 16|LOOP: ld c, 221  ; digitalRead, result in A(0 = LOW, 1 = HIGH)
 17|     ld d, PINI ; digital pin number (INPUT-PIN)
 18|     call BDOS  ; BDOS entry point
 19|     push af    ; Save Accumulator
 20|
 21|     ld c, 222  ; digitalWrite
 22|     ld d, PINO ; digital pin number (OUTPUT-Pin)
 23|     ld e, a    ; load e with accu
 24|     call BDOS  ; BDOS entry point
 25|     pop af     ; Restore accu
 26|     adc a, 0   ; Set flags
 27|     jp z,LOOP  ; Jump to loop if Z=0
 28|     ret        ; Return to ccp
```

Um eine Schleife zu realisieren, wird normalerweise ein Label verwendet, was wir in der Vergangenheit ebenfalls schon verwendet haben und in Zeile 16 zu sehen ist. In den Zeilen 16 bis 19 wird ein *digitalRead* ausgeführt, denn

es soll eine kontinuierliche Abfrage des betreffenden Pins erfolgen. Wichtig hierbei ist wieder das Sichern des Accumulator-Inhaltes in Zeile 19, denn es erfolgt später in den Zeilen 21 bis 24 die Ausführung von *digitalWrite* mit einem weiteren BDOS-Aufruf, der den Accumulator beeinflusst.

Um jetzt den Accumulator hinsichtlich des Z-Flags abzufragen, wird besagter Trick angewendet, und in Zeile 26 ein ***ADC A, 0*** durchgeführt, was den Inhalt des Accumulators nicht beeinflusst, aber die Flags entsprechend setzt. Solange das Z-Flag nicht gesetzt ist und den Wert 0 vorweist (der Taster wurde noch nicht gedrückt), erfolgt ein Sprung über ***JP Z, LOOP*** (Jump if Zero) zum Label ***LOOP:*** und der Prozess beginnt von vorne. Erst, wenn der Taster gedrückt ist, bekommt der Inhalt von Z den Wert 1, ist somit gesetzt und die Bedingung für den Sprung nicht mehr erfüllt. Es wird die Zeile 28 nach dem Sprung zur Ausführung gebracht und die ***RET***-Anweisung ausgeführt, was die Kontrolle an die Konsole zurückgibt.

Es ist zu bemerken, dass das Programm wirklich solange alles blockiert, bis der Taster gedrückt wird und damit einen HIGH-Pegel an Pin 11 leitet. Das laufende Programmkann nicht durch eine Tastenkombination von ***Strg-C*** oder ähnlichen Abbruchsversuchen wie zum Beispiel über die ***ESC***-Taste unterbrochen werden. Wenn also ein Fehler in der Programmierung vorhanden ist und sich die Programmausführung an irgendeiner Stelle „festgefressen" hat, so kann man die Kontrolle über CP/M erst durch einen Kaltstart (Aus/An) wiedererlangen.

2.13.2 Analoge Pins

Was mit den digitalen Pins geht, ist auch mit analogen Pins machbar. Die folgenden Initialisierungen sind dafür erforderlich.

Die Umsetzung von analogRead

```
01 ld c, 223
02 ld d, <pin> ; Arduino-Pin-Nummer
03 call BDOS   ; Aufruf von BDOS -> Ergebnis in HL-Reg. (0 - 1023)
```

Die Umsetzung von analogWrite mittels PWM

```
01 ld c, 224
02 ld d, <pin> ; Arduino-Pin-Nummer
03 ld e, value ; PWM von 0 bis 255
04 call BDOS   ; Aufruf von BDOS
```

Wenn es um das Lesen eines analogen Wertes an einem Eingang des Boards geht, so rate ich, einen Sprung in das Kapitel über die Programmiersprache *Turbo Pascal* zu wagen. Dort wird beschrieben wie ein sehr einfaches Abfragen eines analogen Pins zu realisieren ist. Zudem wird dort die *PWM*-Ansteuerung eines digitalen Pins besprochen. Weitere Informationen sind unter der folgenden Internetadresse zu finden.

 https://learn.adafruit.com/z80-cpm-emulator-for-the-samd51-grand-central/io-in-z80-assembly

3 Die Programmiersprache C

Wenn es um eine Hochsprache geht, die auch heute noch sehr stark Verwendung findet, dann ist das die Programmiersprache *C*. Es handelt sich um eine imperative und prozedurale Programmiersprache, die der Informatiker *Dennis Ritchie* in den 1970er Jahren an den Bell Laboratories entwickelt hat. Sie bietet eine hohe Performance, arbeitet maschinennah und funktioniert mit minimalen Ressourcen auf den unterschiedlichsten Plattformen und war beziehungsweise ist der Grund für die weite Verbreitung. Natürlich ist sie auch für den Z80 unter CP/M verfügbar. Unter der folgenden Internetadresse ist Mike's *Enhanced Small C Compiler for Z80 and CP/M* zu finden.

https://github.com/MiguelVis/mescc

Am Ende dieses Kapitels werde ich kurz auf den ***Aztec-C***-Compiler eingehen. Es handelt sich um einen C-Compiler für *MS-DOS*, *Apple II*, *DOS 3.3*, *ProDOS*, *Commodore C64*, Macintosh, CP/M-80, *Commodore Amiga* und *Atari ST*.

Im Internet finden sich eine Unmenge an freier Literatur und Tutorials. Der hier genannte Compiler stellt eine abgespeckte Version eines C-Compilers dar, was das Wort *Small* sicherlich erkennen lässt. Dennoch lässt sich hiermit ein sehr guter Einstieg in die C-Programmierung finden, denn die zahlreichen Bibliotheken, die zur Verfügung, erleichtern das Erstellen eines Programms. Das typische *Hello-World!* Programm schaut dann wie folgt aus, was ich unter *hello.c* gespeichert habe.

```
te: HELLO.C                              --- | Lin:0007/0007/0512 Col:02/76 Len:01
   !...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...
  1|#include "mescc.h"
  2|#include "conio.h"
  3|
  4|main()
  5|{
  6|    puts("Hello world!");
  7|}
```

Auch hier möchte ich einen geeigneten Einstieg für jemanden finden, der sich noch nicht mit dieser Programmiersprache auseinandergesetzt hat. Programme in der Programmiersprache *C* werden - wie bei Maschinensprache - in einer Textdatei gespeichert, deren Dateiendung *.c* ist. Auch hier kann natürlich der schon zum Einsatz gekommene Text-Editor *te* verwendet werden. Wie ist aber ein C-Programm im Grunde genommen strukturiert? Sehen wir uns das einmal genauer an.

Jedes C-Programm enthält die Definition einer Funktion mit dem Namen *main*, die den definierten Start des Programms darstellt. Das *int* vor der Funktion ist der Datentyp des Rückgabewertes dieser Funktion. Im gezeigten Beispiel wird dieser Datentyp jedoch nicht explizit angegeben, was vollkommen ok ist, denn im Hintergrund ist er trotzdem vorhanden.

```
int main()

{

}
```

Vorsicht, denn die Programmiersprache *C* unterscheidet zwischen Klein- und Großschreibung, ist also Case-Sensitiv! Eine Funktion ist in diversen höheren Programmiersprachen die Bezeichnung eines Programmkonstrukts, mit dem der Quellcode strukturiert wird, so dass Teile der Funktionalität des Programms wiederverwendbar sind. Eine Funktion kapselt sozusagen mehrere Einzelanweisungen unter einem bestimmten Namen. Wenn dieser Name dann programmtechnisch aufgerufen wird, kommt es zur Ausführung der einzelnen Anweisungen, die diese Funktion beinhaltet. Beim Start eines C-Programms wird sofort nach der *main*-Funktion gesucht, die zwingend vorhanden sein muss, und diese dann aufgerufen.

Vor der Definition der *main*-Funktion befinden sich einige Konstrukte, die sehr hilfreich beim Programmieren sind. Mit der Befehlszeile *#include* wird eine Bibliothek eingebunden, welche einen mehr oder weniger großen Satz an Befehlen zur Verfügung stellt, die für das komfortable Programmieren benötigt werden. Auf diese Weise muss das Rad also nicht immer wieder neu erfunden werden. Bei der genannten #include-Anweisung handelt es sich genaugenommen um keine Anweisung, sondern um eine sogenannte Präprozessordirektive, die mit einem # eingeleitet werden. Sie werden vom Präprozessor ausgewertet. Die Direktive #include bedeutet, dass die nachfolgende Headerdatei einzufügen ist. Headerdateien besitzen die Dateinamenendung (*Suffix.h*) und sind in der Programmierung Textdateien, die bestimmte Deklarationen und andere Bestandteile des Quelltextes beinhalten.

Die beiden einzufügenden Header-Dateien

- mescc.h
- conio.h

besitzen also Informationen, die im späteren Verlauf des Programms benötigt werden. Die Datei *mescc.h* beinhaltet Details, die der *Small C Compiler for Z80* unter CP/M benötigt und muss immer an erster Stelle aufgeführt werden. Die Datei *conio.h* - Der Name *conio* kommt von *CONsole Input/Output* - enthält Funktionen für die Konsolen Ein- beziehungsweise Ausgabe. Einige der an den häufigsten verwendeten Funktionen sind *clrscr*, *getch*, *getche*, usw. Sie können dazu verwendet werden, um den Bildschirm zu löschen, die Farbe von Text und Hintergrund zu ändern, Text zu verschieben, zu prüfen, ob eine Taste gedrückt ist und um andere Aufgaben auszuführen.

Der eigentliche Start des Programms ist dann der Aufruf der *main*-Funktion, wo die eigentlichen Anweisungen beginnen, was das Programm denn überhaupt machen soll. Die *puts*-Funktion (Bibliotheksfunktion) innerhalb der *main*-

Funktion bewirkt, dass eine angegebene Zeichenkette bis einschließlich des Nullzeichens nach *stdout*. Ein Befehl in *C* wird immer mit einem Semikolon (;) abgeschlossen. An die Ausgabe wird ein Zeilenumbruchzeichen angehängt. Was bedeutet denn um Himmels Willen *stdout*? Die Datenströme (englisch: *standard streams*) sind Datenströme für die Ein- und Ausgabe in einem Betriebssystem und werden von der C-Standard-Bibliothek unterstützt. Wird *stdout* aufgerufen, wird ein Datenstrom - zum Beispiel eine Zeichenkette - an die Konsole weitergeleitet. Das soll zur Einleitung eines C-Programms erst einmal genügen. Sehen wir uns den Ablauf an, der zur Erstellung einer COM-Datei aus der C-Quelldatei notwendig ist, einmal genauer an.

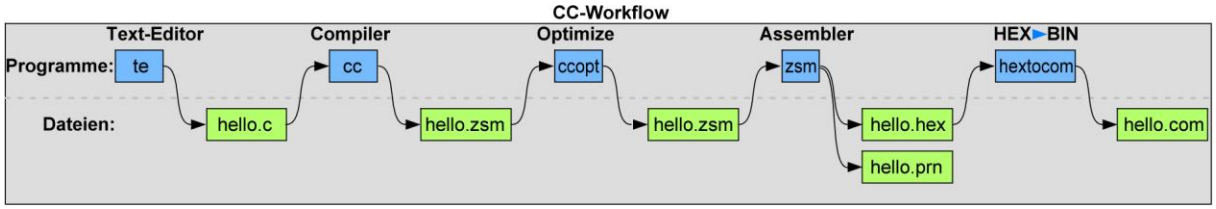


Abbildung 46 - Der CC-Workflow

Sagen wir, dass die Datei *hello.c* existiert, die über den Text-Editor *te* eingegeben wurde. Die folgenden Schritte sind erforderlich, wobei die jeweiligen Dateiendungen nicht mit angegeben werden. Um detaillierte Informationen zu bekommen, ist ein Blick in die folgende Datei ratsam.

https://github.com/MiguelVis/mescc/blob/master/mescc.txt

3.1 Die Quelldatei kompilieren

Um die Quelldatei zu kompilieren, muss die folgende Befehlszeile eingegeben werden.

cc <Dateiname>

Die Angabe der Dateiendung *.c* ist hier nicht erforderlich.

```
C1>cc hello
Mike's Enhanced Small C Compiler v1.21 - 03 May 2018

(c) 1999-2018 FloppySoftware

#include "mescc.h"
#end include
#include "conio.h"
  Function: getchar
  Function: putchar
  Function: putstr
  Function: puts
#end include
Function: main

Writing strings...
Writing globals...

Table    Used  Free
-------- ----- -----
Strings     13  9203
Globals    126  7974
Macros     121  3879

Errors:     0
Next label: a4


RunCPM Version 6.0 (CP/M 60K)

C1>
```

Als Ergebnis liegt nun die Assembler-Datei *hello.zsm* vor.

3.2 Die Kompilierung optimieren

Um die erstellte Datei *hello.zsm* zu optimieren, wird die folgende Kommandozeile eingegeben

ccopt <Dateiname>

Die Angabe der Dateiendung *.zsm* ist hier ebenfalls nicht erforderlich.

```
C1>cc hello
Mike's Enhanced Small C Compiler v1.21 - 03 May 2018

(c) 1999-2018 FloppySoftware

#include "mescc.h"
#end include
#include "conio.h"
  Function: getchar
  Function: putchar
  Function: putstr
  Function: puts
#end include
Function: main

Writing strings...
Writing globals...

Table    Used  Free
-------- ----- -----
Strings     13  9203
Globals    126  7974
Macros     121  3879

Errors:     0
Next label: a4


RunCPM Version 6.0 (CP/M 60K)

C1>
```

Es ist zu erkennen, dass beim Optimierungsprozess 2 Bytes eingespart wurden. Die zuvor vorhandene *hello.zsm*-Datei wurde aktualisiert und neu generiert.

3.3 Die Assemblierung durchführen

Im nächsten Schritt wird die erstelle und optimierte *hello.zsm*-Datei assembliert. Um diese Datei zu assemblieren, muss die folgende Befehlszeile eingegeben werden.

zsm <Dateiname>

Die Angabe der Dateiendung *.zsm* ist hier wiederum nicht erforderlich.

```
C1>zsm hello
Zilog/Mostek Z80 Assembler Version 3.4 (Z80 CPU)

Pass 1
Pass 2

Errors    0

Sorting symbols

Finished


RunCPM Version 6.0 (CP/M 60K)

C1>
```

Das Ergebnis dieses Prozesses besteht in der Generierung der beiden Dateien *hello.hex* und *hello.prn*. Letztendlich muss aus der HEX-Datei - wie schon vormals erwähnt - eine COM-Datei erzeugt werden, was mit dem Programm *hextocom* erfolgt.

3.4 Eine COM-Datei generieren

Im nächsten und letzten Schritt erfolgt die Generierung einer ausführbaren COM-Datei. Es muss die folgende Befehlszeile eingegeben werden.

hextocom <Dateiname>

Die Angabe der Dateiendung *.hex* ist hier - wie sollte es anders sein - nicht erforderlich.

```
C1>hextocom hello
HexToCom v1.05 / 10 Jan 2016

(c) 2007-2016 FloppySoftware

First address: 0100
Last address:  04AD
Size of code:  03AE (942 dec) bytes


RunCPM Version 6.0 (CP/M 60K)

C1>
```

Letztendlich liegt nun die *hello.com* Datei vor, die einfach ausgeführt werden kann.

3.5 Die COM-Datei ausführen

Jetzt kommt die Zeit der Wahrheit und es kann überprüft werden, ob die Quelldatei denn auch richtig programmiert wurde. Dazu muss die folgende Befehlszeile eingegeben werden.

<Dateiname>

Die Angabe der Dateiendung *.com* ist hier - ich muss es eigentlich ja wohl nicht mehr erwähnen - nicht erforderlich.

```
C1>hello
Hello world!


RunCPM Version 6.0 (CP/M 60K)

C1>
```

Und siehe da, die Nachricht wurde korrekt angezeigt. *Puhh!!!* Natürlich ist das hier kein kompletter Programmierkurs in C, denn im Internet gibt es Tonnen an Tutorials.

3.6 Ein weiteres kleines C-Programm

Im nächsten kleinen C-Programm wollen wir eine Eingabe machen und dann diesen Text mit der Anzahl der einzelnen Zeichen wieder ausgeben. Das Programm schaut wie folgt aus, wobei ich es *inout.c* genannt habe.

```
te: INOUT.C                              --- | Lin:0017/0017/0512 Col:02/76 Len:01
    !...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...
  1|#include "mescc.h"
  2|#include "conio.h"
  3|#include "printf.h"
  4|
  5|int main()
  6|{
  7|    char c, sentence[80];
  8|    int i;
  9|    i = 0;
 10|    puts("Enter text:");
 11|    while((c=getchar())!='\r')
 12|        sentence[i++]=c;
 13|    sentence[i]='\0';
 14|    puts("\nText was:");
 15|    printf("with %d letters,\n", i);
 16|    return 0;
 17|}
```

Hier kommen einige Programmkonstrukte der Programmiersprache C zum Einsatz.

- Datentypen
- Variablen
- Array
- Schleife
- Funktion zur Eingabe
- Funktion zur formatierten Ausgabe
- Escape-Steuerzeichen

Das sind natürlich einige Aspekte, die für einen Einsteiger in der Programmiersprache *C* absolut neu sind. Dies soll aber kein Grundkurs in C sein und darum bringe ich nur ein paar essentielle Dinge. Die verwendete Funktion zur Anzeige einer Zeichenkette oder einzelner Zeichen *printf* ist etwas flexibler, als *puts*, denn mit ihr kann eine formatierte Ausgabe erzielt werden. Um diese zu nutzen, muss jedoch eine neue #include-Zeile mit dem Namen der entsprechenden Header-Datei eingefügt werden. Über die *while*-Schleife wird die Eingabe der eingegebenen Zeichen mithilfe der *getchar*-Funktion solange fortgeführt, bis die *RETURN*-Taste gedrückt wird, was dem Zeichen *\r* entspricht, wobei das einzelne Zeichen in der Variablen *c* gespeichert wird und im Endeffekt alle Zeichen im Array *sentence* abgelegt sind. Die Funktion *puts* zeigt die eingegebenen Zeichen an und über die *printf*-Funktion kommt es dann letztendlich zur Anzeige der Variablen *i*, in der ein Anzahl der eingegebenen Zeichen gespeichert sind. Das Zeichen *\n* bewirkt nach der Ausgabe einen Zeilenvorschub. Eine mögliche Ein und Ausgabe schaut dann wie folgt aus.

```
C1>inout
Enter text:
Ich liebe CP/M!
Text was:
with 15 letters,

RunCPM Version 6.0 (CP/M 60K)

C1>
```

3.7 Verschiedene Farben auf der Konsole ausgeben

Wenn es um die Ansteuerung eines Terminals geht, dann ist das *VT100* sehr verbreitet. Es handelt sich um ein ASCII-Computer-Terminal, das von der Firma *Digital Equipment Corporation* (DEC) in den Jahren 1978 bis 1983 hergestellt wurde. Wenn es zum Beispiel um die Steuerung des Cursors oder um die Darstellung verschiedener Farben geht, dann werden sogenannte *ANSI-Escape-Sequenzen* verwendet. Es werden dabei anstelle von darzustellenden Buchstaben und Zahlen definierte Zeichenfolgen als Steueranweisungen an das Terminal gesendet, die mit dem Escape-Zeichen (ASCII: dezimaler Wert = 27, oktal = 033) beginnen.

Um also zum Beispiel die Vordergrundfarbe eines anzuzeigenden Textes zu ändern, werden die folgenden Escape-Sequenzen verwendet.

- Black: \033[0;30m
- Red: \033[0;31m
- Green: \033[0;32m
- Yellow: \033[0;33m
- Blue: \033[0;34m
- Purple: \033[0;35m
- Cyan: \033[0;36m
- White: \033[0;37m

Für Fettschrift (BOLD) muss einfach die anfängliche 0 vor dem Semikolon durch eine 1 ersetzt werden. BOLD Yello wäre dann *\033[1;33m*. Ein entsprechendes C-Programm kann wie folgt ausschauen.

```
te: COLOR.C                              --- | Lin:0016/0016/0512 Col:02/76 Len:01
    !...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...
  1|#include "mescc.h"
  2|#include "conio.h"
  3|#include "printf.h"
  4|
  5|int main()
  6|{
  7|    printf("\033[0;31m");
  8|    printf("Red\n");
  9|    printf("\033[0;32m");
 10|    printf("Green\n");
 11|    printf("\033[1;33m");
 12|    printf("Yello\n");
 13|    printf("\033[0;30m");
 14|    printf("Black\n");
 15|    return 0;
 16|}
```

Die Ausgabe schaut dann wie folgt aus.

```
C1>color
Red
Green
Yello
Black


RunCPM Version 6.0 (CP/M 60K)

C1>
```

Es ist kaum zu sehen, dass die Textausgabe für Gelb (Yello), fett dargestellt ist. Es zeigt sich ein abweichendes Verhalten auf unterschiedlichen Systemen. Gerade bei einem weißen Hintergrund ist die Farbe Gelb auch sehr schwer zu erkennen und da ist ein schwarzer Hintergrund schon besser geeignet.

3.8 Ein eigenes DUMP-Programm

Wir haben ja jetzt schon das DUMP-Programm des Öfteren genutzt, um darüber den Inhalt einer Datei sowohl im HEX-Format anzeigen zu lassen. Ich möchte dieses Programm mit der Programmiersprache C selbst programmieren und eine ASCII-Anzeige hinzufügen. Ich habe zur späteren Anzeige dazu die folgende Text-Datei erstellt.

```
te: INFO.TXT                             --- | Lin:0003/0003/0512 Col:26/76 Len:25
    !...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...
  1|Das ist eine Text-Datei
  2|mit einem Info-Text,
  3|der absolut sinnfrei ist!
```

Über das bisherige DUMP-Programm schaut die Anzeige wie folgt aus und es sind lediglich die HEX-Werte zu sehen.

```
C1>dump info.txt

0000 44 61 73 20 69 73 74 20 65 69 6E 65 20 54 65 78
0010 74 2D 44 61 74 65 69 0D 0A 6D 69 74 20 65 69 6E
0020 65 6D 20 49 6E 66 6F 2D 54 65 78 74 2C 0D 0A 64
0030 65 72 20 61 62 73 6F 6C 75 74 20 73 69 6E 6E 66
0040 72 65 69 20 69 73 74 21 0D 0A 1A 1A 1A 1A 1A 1A
0050 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A
0060 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A
0070 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A

C1>
```

Mit dem Programm, das wir gleich entwickeln, schaut es wie folgt aus.

```
C1>mydump info.txt
Requested file: INFO.TXT
0000 - 44 61 73 20 69 73 74 20 65 69 6E 65 20 54 65 78 Das ist eine Tex
0010 - 74 2D 44 61 74 65 69 0D 0A 6D 69 74 20 65 69 6E t-Datei..mit ein
0020 - 65 6D 20 49 6E 66 6F 2D 54 65 78 74 2C 0D 0A 64 em Info-Text,..d
0030 - 65 72 20 61 62 73 6F 6C 75 74 20 73 69 6E 6E 66 er absolut sinnf
0040 - 72 65 69 20 69 73 74 21 0D 0A 1A 1A 1A 1A 1A 1A rei ist!........
0050 - 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A ................
0060 - 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A ................
0070 - 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A ................


RunCPM Version 6.0 (CP/M 60K)

C1>
```

Auf der rechten Seite werden, ganz wie man das vom DDT-Programm gewohnt ist, die interpretierten ASCII-Zeichen zur Anzeige gebracht. Im *MYDUMP*-Programm werden jedoch nur die druckbaren ASCII-Zeichen dargestellt und der Rest mit Punkten abgebildet. Es soll nicht verwundern, dass zu erkennen ist, dass das Programm nach seiner Ausführung einen sogenannten *Warmstart* an den Tag legt. Ein Warmstart wird verwendet, um das System schnell zurückzusetzen, was in Situationen hilfreich sein kann, in denen ein System neu initialisiert werden muss, aber nicht komplett heruntergefahren und neu gestartet werden muss, wie bei einem Kaltstart. Fangen wir also an.

Ich habe den Quellcode aufgrund seiner Länge in einzelne bereich unterteilt, so dass ich die Bedeutung der einzelnen Zeilen besser dokumentieren kann.

Erster Teil

In den Zeilen 1 bis 5 werden die benötigten Bibliotheken durch ***#include***-Anweisungen eingebunden, wobei in Zeile 4 eine ***#define***-Anweisung steht, um darüber die ***fread***-Funktion im Code nutzen zu können. Wenn ein C-Programm aufgerufen wird, dann werden der Dateiname und eventuell vorhandenen Argumente in einem Array mit der Bezeichnung ***argv*** gespeichert und deren Anzahl in ***argc***. Der eigentliche Programmname ist dabei in *argv[0]* und die Argumente in *argv[1]*, *argv[2]*, usw. zu finden. Diesen Umstand machen wir uns zunutze, um eine Datei mit ihrem Namen dem eigentlichen Programmaufruf zu übergeben, die untersucht und angezeigt werden soll, ganz so, wie man das auch von *DUMP*, *DDT* oder anderen Programmen gewohnt ist.

```
te: MYDUMP.C                              --- | Lin:0001/0063/0512 Col:01/76 Len:18
   !...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...
  1|#include "mescc.h"
  2|#include "conio.h"
  3|#include "printf.h"
  4|#define  CC_FREAD
  5|#include "fileio.h"
  6|
  7|int main(argc, argv)
  8|int argc;
  9|int argv[];
 10|{
 11|    int i;           /* Counter for bytes per line */
 12|    int line;        /* Adress-Line */
 13|    FILE *fp;        /* pointer for file */
 14|    char buffer[16]; /* buffer for file */
 15|
 16|    if(argc !=2)
 17|    {
 18|        printf("\nUsage: mydump... <filename>");
 19|        return 1;
 20|    }
```

In den Zeile 7 bis 9 sind sowohl die *main*-Funktion, als auch die angesprochenen Variablen *argc* und *argv* zu finden. In den Zeilen 11 bis 14 werden einige Variablen deklariert, die später zum Einsatz kommen. Die Variable *i* wird dafür genutzt, um die einzelnen Bytes - 16 an der Zahl - aus der gelesenen Datei anzuwählen. Wir sehen das gleich innerhalb der beiden *for*-Schleifen. Die Variable *line* wird genutzt, um die Startadresse - beginnend bei 0 - je Zeile um den Wert 16 zu erhöhen, weil in jeder Zeile 16 HEX-Werte gelistet werden. Um die darzustellende Datei zu lesen, wird der Zeiger **fp* vom Datentyp *FILE* deklariert und die 16 HEX-Werte finden im *buffer*-Array ihren Platz.

In den Zeilen 16 bis 20 wird über eine *if*-Abfrage überprüft, um dem Programm überhaupt ein Argument mitgegeben wurde. Ist der Wert von *argc* kleiner 2, bedeutet dies, dass das erforderliche Argument in Form eines Dateinamens fehlt und das Programm wird mit einem entsprechenden Hinweis zur Nutzung und einem Fehler über *return 1* verlassen.

Zweiter Teil

In Zeile 22 wird über die *printf*-Funktion der angeforderte Dateiname auf der Konsole ausgegeben, was jedoch nicht unbedingt erforderlich ist. Die Zeilen 25 bis 29 sorgen dafür, dass die angeforderte Datei auch wirklich im Dateisystem vorhanden ist. Ist das nicht der Fall, wird das Programm wieder über *return 1* mit einer vorherigen Fehlermeldung verlassen.

```
te: MYDUMP.C                              --- | Lin:0042/0063/0512 Col:01/76 Len:09
   !...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...
 22|    printf("Requested file: %s\n", argv[1]);
 23|
 24|
 25|    if((fp = fopen(argv[1],"rb")) == NULL)
 26|    {
 27|        printf("No file. Error!");
 28|        return 1;
 29|    }
```

Ist die Datei jedoch vorhanden, kommt es zur Ausführung der Programmzeilen hinter Zeile 29.

Dritter Teil

In Zeile 31 wird die Zeilennummer über die Variable *line* auf 0 gesetzt und nach jedem Durchlauf der *while*-Schleife in Zeile 43 um den Wert 16 erhöht. Die while-Schleife sorgt dafür, dass aus der zu untersuchenden Datei die

Werte in Häppchen von 16 Bytes in den *buffer* über die *fread*-Funktion geladen werden. Die führenden Zeilennummern werden über die *printf*-Funktion in Zeile 34 zur Anzeige gebracht. Innerhalb der ersten for-Schleife kommt es zum mehrmaligen Aufruf der *showHEX*-Funktion, die zur Anzeige der HEX-Werte verantwortlich ist und wird später unter im Code definiert. Die zweite für-Schleife ruft die *showASCII*-Funktion auf, die zur Anzeige der ASCII-Werte genutzt wird und ebenfalls später im Code definiert wird. Es darf nicht vergessen werden, ein sogenanntes Datei-Handle über den Zeiger *fp* wieder zu schließen, was über die *fclose*-Funktion in Zeile 47 erfolgt und das Programmüber *return 0* in Zeile 48 verlassen wird.

```
te: MYDUMP.C                              --- | Lin:0051/0063/0512 Col:01/76 Len:10
   !...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...
 31|    line = 0;
 32|    while(fread(buffer, 16, 1, fp)==1)
 33|    {
 34|        printf("%04x - ", line);
 35|        for(i=0; i < 16; ++i)
 36|        {
 37|            showHEX(buffer[i]);
 38|        }
 39|        for(i=0; i < 16; ++i)
 40|        {
 41|            showASCII(buffer[i]);
 42|        }
 43|        line += 16; /* Jump to next line */
 44|        printf("\n");
 45|    }
 46|    fclose(fp);
 47|    return 0;
 48|}
```

Vierter Teil

Nachfolgend sind die Definitionen der beiden erwähnten Funktionen *showHEX* und *showASCII* zu sehen.

```
 50|showHEX(byte)
 51|char byte;
 52|{
 53|    printf("%02x ", (byte & 0xFF));
 54|}
 55|
 56|showASCII(byte)
 57|char byte;
 58|{
 59|    if(((byte & 0xFF) < 0x20) || ((byte & 0xFF) > 0x7E))
 60|        printf(".");
 61|    else
 62|        printf("%c", (byte & 0xFF));
 63|}
```

Über die Eingrenzung der darstellbaren ASCII-Werte im Bereich *0x20* und *0x7E* wird in Zeile 59 erreicht, dass wirklich nur anzuzeigende ASCII-Zeichen in der Anzeige auftauchen. Nicht darstellbare Zeichen werden über einen Punkt gekennzeichnet.

3.9 Die Anzeige der CP/M-Versionsnummer

Ich möchte am Beispiel der Anzeige der CP/M-Versionsnummer verdeutlichen, wie man innerhalb von C-Quellcode auch Assemblercode unterbringen kann, um diesen dann recht einfach aufzurufen. Ich gehe später im *Turbo Pascal*-Kapitel noch einmal darauf ein und zeige dort, wie ein entsprechender BDOS-Aufruf zu realisieren ist. Um die CP/M-Versionsnummer zu ermitteln, muss vor dem BDOS-Aufruf das C-Register mit dem Wert 12 geladen werden.

- **Function Code:** 12
- **Bezeichnung:** Return Version Number / CP/M Versionsnummer
- **Input:** C = 0Ch
- **Output:** HL = 0022h (nur bei CP/M 2.2)

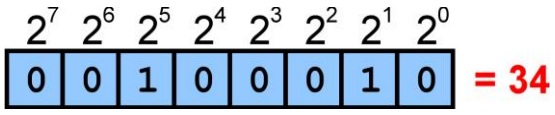
Sehen wir uns dazu den C-Quellcode an.

```
te: CV.C                                  --- | Lin:0017/0017/0512 Col:08/76 Len:07
   !...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...
  1|#include "mescc.h"
  2|#include "conio.h"
  3|#include "printf.h"
  4|
  5|main()
  6|{
  7|    unsigned int ver;
  8|    ver = cpmver();
  9|    printf("Wert: %d\n", ver);
 10|    printf("CP/M-Version: %d.%d",(ver>>4) & 0x0F, ver & 0x0F);
 11|}
 12|
 13|#asm
 14|cpmver:
 15|    ld  c, 12
 16|    jp 5
 17|#endasm
```

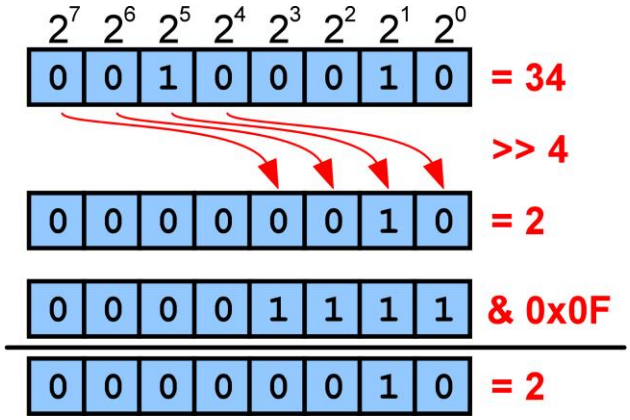
In Zeile 7 wird die Variable ***ver*** von Datentyp ***unsigned int*** deklariert, was bedeutet, dass es sich um einen vorzeichenlosen Ganzzahltyp handelt. In der darauffolgenden Zeile 8 erfolgt die Initialisierung dieser Variablen über einen Funktionsaufruf mit dem Namen ***cpmver***. Jedoch ist im C-Quellcode keine derartige Funktion zu sehen. Stattdessen ist aber in den Zeilen 13 bis 17 ein Assemblercode zu sehen, der über ***cpmver:*** eingeleitet, mit ***#asm*** gekennzeichnet und mit ***#endasm*** abgeschlossen ist. Der darin enthaltene Code wird ähnlich wir purer Assemblercode verarbeitet und das Ergebnis an den Aufrufer zurückgeliefert. Wir sehen, dass in Zeile 15 das C-Register mit dem Wert 12 geladen wurde und mittels ***jp 5*** der BDOS-Aufruf über den Sprung zur genannten Adresse erfolgt. Nach dem Start des Programms ist die folgende Ausgabe zu sehen.

```
C1>cv
Wert: 34
CP/M-Version: 2.2

RunCPM Version 6.0 (CP/M 60K)

C1>
```

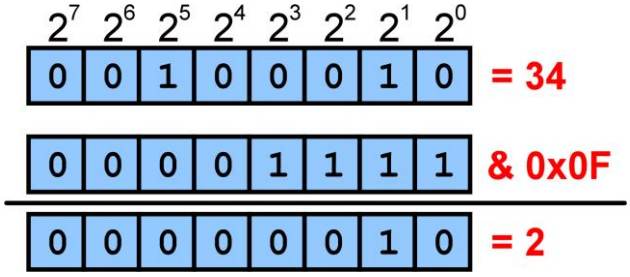
Es ist zu erkennen, dass der Rückgabewert *34* beträgt und es ist sicherlich eine berechtigte Frage, wie man von *34* auf den Wert *2.2* kommt. Sehen wir uns dazu den Wert 34 auf Bit-Ebene an.



Um jetzt an die linke 2 (Major-Version) zu kommen, müssen alle Bits um 4 Stellen nach rechts verschoben und mit dem Wert ***0x0F*** logisch UND (**&**) verknüpft werden. Das nachts rechts Schieben erfolgt hier über den ***Shift-Right-Operator >>***.



Um an die rechte 2 (Minor-Version) zu gelangen, muss der Inhalt der Variablen lediglich mit dem Wert ***0x0F*** UND verknüpft werden.



3.10 Der Aztec-C-Compiler

Wie am Anfang erwähnt, hier ein paar einleitende Worte zum *Aztec-C*-Compiler. Dieser Compiler befindet sich auf Laufwerk ***C3:***. Wie schaut es mit einem entsprechenden Manual aus?

	**Wo finde ich das Aztec-C Manual?**
Das *Aztec-C-Manual* Manual befindet sich im gleichen Verzeichnis, wie das Programm selbst und kann über *manual.pdf* angesehen werden.	

Bevor wir mit diesem Compiler beginnen, sollten wir uns den Aztec-C-Compiler-Workflow ansehen.

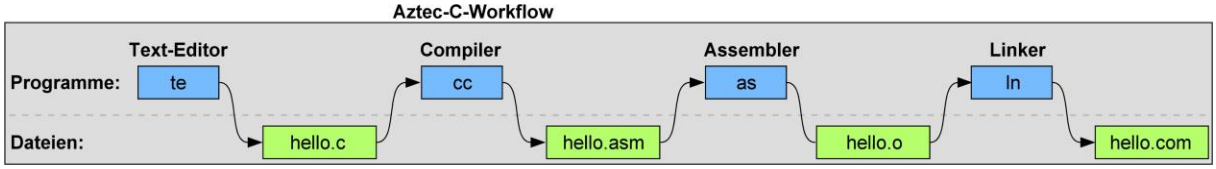


Abbildung 47 - Der Assembler-Workflow für den Aztec-C-Compiler

Gehen wie die einzelnen Schritte einmal durch, die erforderlich sind, um vom Quellcode zu einer ausführbaren COM-Datei zu gelangen.

3.10.1 Die Eingabe des Quellcodes

Wie wir das schon kennen, ist es zu Beginn natürlich erforderlich, einen Quellcode zu besitzen, der vorgibt, wie das Programm abzulaufen hat. Das machen wir wieder mit unserem *TE*-Texteditor. Als Beispiel nehme ich einen Algorithmus, der die Primzahlen von 1 bis zu einem maximal vorgegebenen Wert ausgibt. In der äußeren *for*-Schleife werden die zu prüfenden Zahlen von 1 bis 100 durchlaufen. In der inneren *for*-Schleife wird für die aktuell zu überprüfende Zahl ermittelt, ob für einen Divisor im Bereich von 2 bis aktueller Wert -1 eine ganzzahlige Division ohne Rest vorliegt. Wenn dies für keine Zahl möglich ist, dann handelt es sich um eine Primzahl, die dann auf der Konsole ausgegeben wird.

```
te: PRIM.C                              --- | Lin:0020/0020/0512 Col:02/76 Len:01
   !...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...
  1|#define MAX_ZAHL 100 /* Maximal zu testende Zahl */
  2|
  3|main()
  4|{
  5|    int aktuell;    /* aktuell zu testender Wert */
  6|    int divisor;    /* aktueller Divisor         */
  7|    int primzahl;   /* Primzahl                  */
  8|    printf("Ermittlung aller Primzahlen zwischen 1 und %d\n", MAX_ZAHL);
  9|    for(aktuell=1;aktuell<MAX_ZAHL;aktuell++)
 10|    {
 11|        primzahl = 1; /* Startwert fuer Primzahl */
 12|        for(divisor=2;divisor<aktuell;divisor++)
 13|        {
 14|            /* Falls kein Rest -> Keine Primzahl */
 15|            if(aktuell%divisor==0) primzahl = 0;
 16|        }
 17|        if(primzahl) printf("%8d", aktuell);
 18|    }
 19|    printf("\n"); /* Zeilenvorschub */
 20|}
```

Der Quellcode wird unter ***prim.c*** abgespeichert.

3.10.2 Der Aufruf des Compilers

Im nächsten Schritt wird der *Aztec-C-Compiler* über das Kommando ***cc <Quellcode>*** aufgerufen, wobei die Dateinamenerweiterung ***.c*** nicht mit angegeben wird.

```
C3>cc prim
C Vers. 1.06D 8080  (C) 1982 1983 1984 by Manx Software Systems


RunCPM Version 6.0 (CP/M 60K)

C3>
```

Als Ergebnis dieser Aktion liegt nun eine Datei mit Namen ***prim.asm*** vor.

```
C3>dir prim.*
C: PRIM     C    |  PRIM     ASM
C3>
```

Es handelt sich also um eine Assembler-Datei, die wie folgt ausschaut, wobei das nur der Anfang ist und sich dieser Code über sehr viele Zeilen erstreckt. Dieser Inhalt hat uns im Grunde genommen jedoch nicht zu interessieren, denn er ist die Arbeitsdatei für den nun aufzurufenden Assembler.

```
C3>type prim.asm
        extrn   .begin,.chl,.swt
        extrn   csave,cret,.move
        PUBLIC main_
main_:  lxi d,.2
        call csave
        LXI H,100
        PUSH H
        LXI H,.1+0
        PUSH H
        CALL printf_
        POP D
        POP D
```

3.10.3 Der Aufruf des Assemblers

Der Assembler wird mit dem Kommando ***as <Quellcode>*** aufgerufen, wobei die Dateinamenerweiterung ***.asm*** nicht mit angegeben wird.

```
C3>as prim
8080 Assembler Vers. 1.06D


RunCPM Version 6.0 (CP/M 60K)

C3>
```

Als Ergebnis dieser Aktion liegt nun eine Datei mit Namen ***prim.o*** vor.

```
C3>dir prim.*
C: PRIM     C    |  PRIM     ASM  |  PRIM     0
C3>
```

Was hat es aber nun mit dieser Datei auf sich und was bedeutet die Dateiendung ***.o***? Nach dieser Aktion wird über den Assembleraufruf eine sogenannte ***Objekt***-Datei generiert, die auf dem Weg hin zur COM-Datei erforderlich ist. Bei einer Objektdatei handelt es sich um eine Struktur, die Objektcode enthält und, die von einem Assembler generiert wurde. Diese Datei ist nicht direkt ausführbar und stellt eine Zwischendatei dar. Es gibt Fälle, bei denen mehrere Objekt-Dateien generiert werden. Um aus diesen Dateien eine ausführbare Datei zu generieren, muss ein weiteres Programm hinzugezogen werden. Es handelt sich um den sogenannten ***Linker***.

3.10.4 Der Aufruf des Linkers

Unter einem **Linker** wird ein Programm, das einzelne Programmmodule zu einem ausführbaren Programm zusammenfügt. Auf der folgenden Abbildung ist die Aufgabe eines Linkers vereinfacht dargestellt. Es werden erforderliche Bibliotheken (*lib*) und Objektdateien vom Linker zu einer ausführbaren Datei (*com*) zusammengefügt.

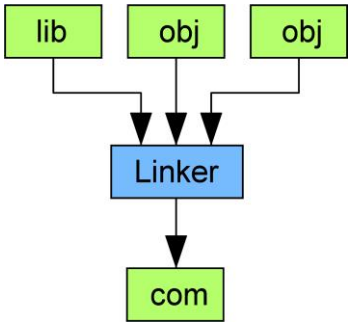


Abbildung 48 - Die Aufgabe eines Linkers

Der Aufruf des Linkers erfolgt über das Kommando ***ln prim.o c.lib***, wobei die schon erwähnte Bibliothek in Form der Datei ***lib.c*** mit eingebunden werden muss. Im Quellcode wird zum Beispiel die *printf*-Funktion verwendet, die in dieser Library unter anderem enthalten ist.

```
C3>ln prim.o c.lib
C Linker Vers. 1.06D
Base: 0100   Code: 1c35  Data: 01ef  Udata: 0250  Total: 002078


RunCPM Version 6.0 (CP/M 60K)

C3>
```

Als Endergebnis liegt jetzt die ausführbare COM-Datei vor.

```
C3>dir prim.*
C: PRIM     C    |  PRIM     ASM  |  PRIM     O    |  PRIM     COM
C3>
```

3.10.5 Der Aufruf der COM-Datei

Letztendlich kann also unser Programm über den Namen der Quelldatei ***prim*** in der Kommandozeile aufgerufen werden und die Primzahlen werden angezeigt.

```
C3>prim
Ermittlung aller Primzahlen zwischen 1 und 100
       1       2       3       5       7      11      13      17      19      23
      29      31      37      41      43      47      53      59      61      67
      71      73      79      83      89      97


RunCPM Version 6.0 (CP/M 60K)

C3>
```

3.10.6 Ein BDOS-Aufruf

Ich möchte dieses Kapitel mit einem BDOS-Aufruf beenden. Diesmal möchte ich den sogenannten ***Direct Console I/O*** mit über den Wert 6 nutzen. Wir müssen also das C-Register mit dem Wert 6 und das E-Register mit dem Wert FFh versehen. Die erkannte Taste, die auf der Konsole gedrückt wurde, wird im Akkumulator vorgehalten.

- **Function Code:** 6
- **Bezeichnung:** Direct Console I/O / Direkte Konsolen Ein-/Ausgabe
- **Input:** C = 06Ch

    - E = FFh: Konsoleneingabe
    - E = FEh: Konsolenstatus prüfen
    - E = ASCII-Zeichen zur Konsolenausgabe
- **Output:**
    - A = Zurückgegebenes Zeichen (falls E = 0FFh)
    - A = Konsolenstatus (falls E = FEh)

Der Code dazu schaut wie folgt aus. In einer Endlosschleife, die durch das Konstrukt ***for(;;)*** realisiert wurde, wird eine endlose Abarbeitung des Codes bewirkt. Erst, wenn die Tastenkombination ***Strg-C*** gedrückt wird, was einem Code von 3 entspricht, kommt es zur Ausführung der ***break***-Anweisung, was ein Verlassen der *for*-Schleife nach sich zieht und in der Beendigung des Programms mündet.

```
te: KEY.C                              --- | Lin:0016/0016/0512 Col:02/76 Len:01
    !...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...
  1|main()
  2|{
  3|    int a, bc, de;
  4|    a = 0, bc = 6, de = 0xFF;
  5|    printf("Start des Programms. Warte auf Tastendruck\n");
  6|    printf("Abbruch mit Strg-C\n\n");
  7|    for(;;)
  8|    {
  9|        a = bdos(bc, de);
 10|        if(a != 0)
 11|          printf("Pressed key is: '%c' - ASCII is %3d (%2xh)\n", a, a, a);
 12|        if(a == 3) /* Strg-C */
 13|            break;
 14|    }
 15|    printf("Ende des Programms.");
 16|}
```

Die Ausgabe gestaltet sich wie folgt.

```
C3>key
Start des Programms. Warte auf Tastendruck
Abbruch mit Strg-C

Pressed key is: 'a' - ASCII is  97 (61h)
Pressed key is: 'k' - ASCII is 107 (6bh)
Pressed key is: '3' - ASCII is  51 (33h)
Pressed key is: 'P' - ASCII is  80 (50h)
Pressed key is: 'c' - ASCII is  99 (63h)
Pressed key is: '' - ASCII is   3 ( 3h)
Ende des Programms.

RunCPM Version 6.0 (CP/M 60K)

C3>
```

4 Die Programmiersprache Basic

```
10 PRINT "Hallo, hier ist Basic"
20 PRINT "Und es macht Spass, damit zu programmieren"
```

Kommen wir nun zu einer Programmiersprache, die eine entscheidende Rolle in der Programmierung von Computern gespielt hat und von sehr vielen als „Spielsprache" degradiert wurde. Das sind Individuen, die in einem Café sitzen und sich die verbeilaufenden Menschen anschauen und sich anmaßen zu urteilen, wie der- oder diejenige denn ausschaut. Der hat aber eine komische Hose an und die trägt aber eine merkwürdige Frisur. Ist der nicht zu dick und hat die nicht ein zu aufdringliches Makeup aufgetragen? Sie wissen nicht, dass sie fernab dessen sind, worauf es ankommt und haben keine Ahnung, wie Abgrund tief sie gesunken sind. Doch das ist nur meine bescheidende Meinung. Zurück zum Thema. Bei der Programmiersprache Basic handelt es sich um eine sogenannte imperative Programmiersprache, die 1964 von *John G. Kemeny*, *Thomas E. Kurtz* und *Mary Kenneth Keller* am *Dartmouth College* als bildungsorientierte Programmiersprache entwickelt wurde. Es gibt eine Vielzahl verschiedener *Basic-Dialekte*, von denen einige der jüngeren alle Elemente höherer Programmiersprachen aufweisen, so etwa die sogenannte *Objektorientierte Programmierung*. Das Akronym *Basic* steht für *Beginner's All-purpose Symbolic Instruction Code*, was so viel bedeutet wie „symbolische Allzweck-Programmiersprache für Anfänger". Diese Sprache ist in meinen Augen ein sehr guter Einstieg in die Programmierung von Computern. Die Hochsprachen wie C oder C++ stellen natürlich eine Weiterentwicklung von Programmiersprachen dar, doch sie bauen alle auf den Anfängen von Basic auf und das darf nicht vergessen werden, auch, wenn das viele einfach ignorieren! In den meisten IT-Unternehmen kommen moderne Programmiersprachen zum Einsatz. Der Blödsinn, der dort zeitweise propagiert wird, dient nicht der Eleganz der Programmierung, sondern alleine dem Profit von Wenigen! Jede Woche gibt es neue Updates diverser Programmpakete und nur die wenigsten wissen, was sich im Hintergrund wirklich abspielt. Jeder will sich - um immer auf dem neuesten Stand zu sein - mit neuen Features vertraut machen und niemand merkt, wie er in eine Ecke gedrängt wir, die absolut unnötig sind und ihn von dem abbringt, was das eigentlich Ziel ist: *Der Spaß am Programmieren* und nicht den von oben vorgegebenem Wahnsinn zur Profit-Maximierung! Zu Zeiten von Basic in den 1980er Jahren war es noch spannend, Programme zu entwickeln, die einfach tolle Dinge vollbrachten. Heutzutage geht es nur darum, mit irrwitzigen Programmiersprachen, die sicherlich hier und da ihre Berechtigung haben, die Reichen noch reicher zu machen und die genialen Programmierer zu knechten. Kommt das einem irgendwie bekannt vor? Wer sich also mit der Programmiersprache *MBASIC* unter CP/M Release 5.0 und später etwas vertraut machen will, der kann unter der folgenden Internetadresse interessante Hinweise finden.

http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/sources/mbasic.pdf

Unter RunCPM ist auf dem Laufwerk ***A:*** die Programmiersprache *MBASIC* zu finden, so dass man sofort loslegen kann.

```
A0>mbasic
BASIC-85 Rev. 5.29
[CP/M Version]
Copyright 1985-1986  $  by Microsoft
Created: 28-Jul-85
39224 Bytes free
Ok
```

Es handelt sich um die Version 5.29 von *Basic-85* und wurde von *Microsoft* entwickelt. Nach dem Start von MBASIC ist ein *Ok-Prompt* zu sehen. *„Basic ist doch keine Programmiersprache"* raunten damals schon die selbst ernannten Spezialisten. Na und? Erstens ist diese Aussage absoluter Quatsch und zweitens ist es mir vollkommen egal! Um MBASIC wieder zu verlassen und zu Kommandozeile von CP/M zurückzukehren, muss einfach der Befehl ***system*** eingegeben werden.

Wenn ich meinen Computer dazu bringe, meine Eingaben und Wünsche zu verarbeiten und auszugebe, dann programmiere ich. Basic ist eine sehr einfache, aber dafür auch leicht erlernbare Programmiersprache. Moderne Hochsprachen wie *C* oder *C++* benötigen einen Compiler, der den Quellcode übersetzt, so dass die CPU ihn ausführen kann. Bei Basic hingegen handelt es sich um eine sogenannte *Interpretersprache*, was bedeutet, dass ein Programm unmittelbar ausgeführt werden kann. Das geht natürlich zu Lasten der Ausführungsgeschwindigkeit, denn Basic-Programm sind im Vergleich zu Programmen, die in *C*, *C++* oder sogar in *Assembler* geschrieben wurden, viel langsamer.

Bei den frühen Computern wie dem *VC-20*, *C64*, *Apple II* oder *CPC464* - um nur einige zu nennen - war es so, dass nach dem Einschalten der Benutzer sofort mit der Programmierung von Basic beginnen konnte, denn der Basic-Interpreter war Teil des ROMs. Leider gab es keine einheitlichen Standards von der Programmiersprache Basic und jeder kochte quasi sein eigenes Süppchen, was natürlich dazu führte, dass Basic-Programme untereinander nicht kompatibel waren. Zwar gab es einen Grundwortschatz, der aber - wie gerade schon erwähnt -, je nach Firma, einige Erweiterungen erfahren hatte.

MBASIC macht da natürlich keine Ausnahme und doch ist es möglich, mit MBASIC so einiges anzustellen. Wie schaut die Programmierung in der Sprache Basic denn überhaupt aus? Nun, da wird jeder einzelnen Befehlszeile mit einer sogenannten Zeilennummer versehen. Die dadurch entstandene Befehlssequenz wird anhand der vorherrschenden Zeilennummern von der niedrigsten hin zur höchsten abgearbeitet. Da es jedoch Kontrollstrukturen gibt, kann mit geeigneten Abfragen der flache Ablauf von oben nach unten durchbrochen werden. Über Sprunganweisungen (mit *GOTO*) oder Aufrufen von Unterprogrammen (mit *GOSUB*) kann es zu mehr oder weniger intelligenten Verzweigungen im Programmablauf kommen.

Sehen wir uns einfach einmal ein paar grundlegende Strukturen an, wobei ich wieder lediglich an der Oberfläche kratzen werde. Wer sich intensiv mit MBASIC auseinandersetzen möchte, der findet im Internet massenweise Tutorial und auch freie Bücher. Man muss nur etwas suchen!

4.1 Der interaktive Modus

Nachdem MBASIC wie gezeigt mit der Eingabe von *mbasic* gestartet wurde, kann man unmittelbar den interaktiven Modus nutzen. Da es sich bei MBASIC um eine Interpretersprache handelt, muss zur Ausführung der Befehle nichts kompiliert werden. Alles erfolgt unmittelbar nach der Eingabe des Befehls beziehungsweise der Befehlszeile nach der Bestätigung mit der *RETURN*-Taste.

```
A0>mbasic
BASIC-85 Rev. 5.29
[CP/M Version]
Copyright 1985-1986  $  by Microsoft
Created: 28-Jul-85
39224 Bytes free
Ok
print "Hallo, hier spricht MBASIC!"
Hallo, hier spricht MBASIC!
Ok
? 17 + 4
 21
Ok
a = 23 : ? a - 30
-7
Ok
```

Es ist zu sehen, dass man mit dem *print*-Befehl etwas auf der Konsole anzeigen lassen kann. Soll es sich um einen Text handeln, so wird dieser in doppelte Anführungszeichen gesetzt. Die Kurzform für den *print*-Befehl ist das Fragezeichen. Über die Eingabe von ***? 17 + 4*** kann direkt eine gewünschte Berechnung durchgeführt werden und das Ergebnis wird sofort sichtbar. Sollen mehrere Befehle in einer einzigen Zeile platziert werden, müssen diese durch den Doppelpunkt voneinander getrennt angegeben werden.

4.2 Der Programm Modus

Zeilennummern werden im BASIC-Quellcode der oder den Anweisung(en) vorangestellt und dienen dem Interpreter als Markierungen oder Ziele für Sprunganweisungen. Über diese Nummerierung wird ein Programm erstellt und es wird dem System mitgeteilt, dass es sich nicht mehr um den interaktiven Modus handelt, in dem die Befehle unmittelbar zur Ausführung kommen. Die Reihenfolge der Ausführung des Programms beziehungsweise der Programmzeilen erfolgt von niedrigen zu höheren Zeilennummern. Zudem helfen die Zeilennummern dem Programmierer bei der Identifikation von Syntax-Fehlern.

Nach dem Start von MBASIC liegt natürlich noch kein Programm vor, welches ausgeführt werden könnte, so dass lediglich der interaktive Modus zur Verfügung steht. Um ein Programm einzugeben, wird wie beschrieben verfahren, indem zuerst eine Zeilennummer und ein Befehl durch ein Leerzeichen getrennt eingegeben werden. Der Abschluss der Zeile wird wieder mit der *RETURN*-Taste bestätigt. Nachfolgend ist ein einfaches Programm zu sehen. Nach der Eingabe der Programmzeilen kann es über *run* gestartet werden. Es ist hier zu sehen, dass zwischen dem *input*-Befehl und der anzuzeigenden Zeichenkette in Zeile 10 kein Leerzeichen vorhanden ist, jedoch in Zeile 20 zwischen dem *print*-Befehl und der nachfolgenden Zeichenkette keines vorhanden ist. Das führt zu keinem Programmfehler, ist jedoch optisch etwas unschön und macht es in meinen Augen schwieriger, denn Quellcode zu lesen.

```
10 INPUT "Wie lautet dein Name: ", N$
20 PRINT"Hallo: "; N$
run
Wie lautet dein Name: Erik Bartmann
Hallo: Erik Bartmann
Ok
```

Um sich das Programm noch einmal anzusehen, muss der Befehl *list* eingegeben werden.

```
list
10 INPUT "Wie lautet dein Name: ", N$
20 PRINT"Hallo: "; N$
Ok
```

Die meisten BASIC-Programme starten bei mit der Zeilennummer 10 und folgende werden immer um den Wert 10 erhöht. Also *10*, *20*, *30*, *40*, usw. Das hat den Vorteil, dass zwischen schon vorhandenen Programmzeilen bei Bedarf weitere eingefügt werden können.

Eine Zeilennummer, die schon vorhanden ist, wird bei erneuter Verwendung durch die neue Eingabe überschrieben. In MBASIC gibt es eine sehr einfache Möglichkeit, die Zeilennummern automatisch generieren zu lassen, wofür der ***auto***-Befehl verwendet wird.

- ***auto***: Automatische Zeilennummerierung 10, 20, 30, 40, usw.
- ***auto 100,5***: Automatische Zeilennummerierung 100, 105, 110, 115, usw.

Um die automatische Generierung von Zeilennummern abzubrechen, muss die Tastenkombination ***Strg-C*** gedrückt werden.

4.3 Sichern und Laden

Um ein Programm nach einem Neustart des Systems noch verfügbar zu haben, muss es im Dateisystem gespeichert werden, was über den *save*-Befehl erfolgt. Der Dateiname inklusive der Erweiterung wird in doppelte Anführungszeichen gesetzt, wie das nachfolgend zu sehen ist.

```
10 INPUT "Wie lautet dein Name: ", N$
20 PRINT"Hallo: "; N$
save "name.bas"
Ok
```

Das spätere Laden des Programms erfolgt über den *load*-Befehl in gleicher Weise.

```
A0>mbasic
BASIC-85 Rev. 5.29
[CP/M Version]
Copyright 1985-1986  $  by Microsoft
Created: 28-Jul-85
39224 Bytes free
Ok
load "name.bas"
Ok
list
10 INPUT "Wie lautet dein Name: ", N$
20 PRINT"Hallo: "; N$
Ok
```

Sehen wir doch einmal im Dateisystem mit *dir* nach und lassen uns den Inhalt der Datei über *type* anzeigen.

```
A0>dir *.bas
A: ERIK1    BAS  |  IO       BAS  |  NAME     BAS
A0>type name.bas
ma
   "Wie lautet dein Name: ", N$b"Hallo: "; N$
A0>
```

Das schaut irgendwie nicht ganz so aus, wie es vorher eigegeben wurde. Zwar ist der Programmtest der beiden Zeilen zu sehen, doch die Zeilennummern fehlen und zu Beginn der Ausgabe sind zwei ominöse Buchstaben *ma* zu erkennen. Das entspricht also nicht der Eingabe des Basic-Quellcodes. Damit das jedoch auch über *type* oder einem anderen Text-Editor wie zum Beispiel ***TE*** möglich ist, muss die Quelldatei, als ASCII-Datei in diesem Format gespeichert werden, was beim Speichern über den *save*-Befehl über den Zusatz *,a* erfolgt.

```
10 INPUT "Wie lautet dein Name: ", N$
20 PRINT"Hallo: "; N$
Ok
save "name.bas",a
Ok
system


RunCPM Version 6.0 (CP/M 60K)

A0>type name.bas
10 INPUT "Wie lautet dein Name: ", N$
20 PRINT"Hallo: "; N$

A0>
```

Nun wird alles korrekt angezeigt. Es ist zudem möglich, schon beim Aufruf von MBASIC eine zu ladende Quelldatei anzugeben, was jedoch ohne die doppelten Anführungszeichen erfolgen muss. Es ist zu sehen, dass das angegebene Programm geladen und zudem ausgeführt wird.

```
A0>mbasic name.bas
BASIC-85 Rev. 5.29
[CP/M Version]
Copyright 1985-1986  $  by Microsoft
Created: 28-Jul-85
39224 Bytes free
Wie lautet dein Name: Erik Bartmann
Hallo: Erik Bartmann
Ok
```

4.4 Die Variablen und Datentypen

Ich erzähle sicherlich nichts Neues, wenn ich sage, dass Variablen verwendet, um Daten im Speicher zu abzulegen. MBASIC unterscheidet dabei drei Datentypen:

- Integer-Zahlen (Ganzzahlen)
- Real-Zahlen (Gleitkommazahlen)
- Strings (Zeichenketten)

Variablennamen bestehen aus alphanumerischen Zeichen und gegebenenfalls einem Suffix. Das Suffix wird verwendet, um den Variablentyp zu bestimmen.

Typ	Suffix	Speicherplatz	Beispiel
Integer	%	2 Bytes	i% = 5

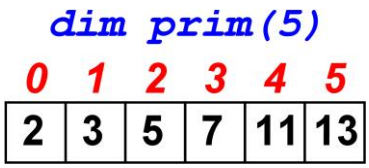
Real	kein Suffix	4 Bytes	pi = 3.1415
Real	! (einfache Genauigkeit)	4 Bytes	w! = 234.656
Real	# (doppelte Genauigkeit)	8 Bytes	w# = 23.7234082354
String	$	bis 255 Bytes	a$="Hallo"

Tabelle 9 - Die einfachen Datentypen in MBASIC

Bei den Variablen für einfache Datentypen kann jeweils nur ein Wert gespeichert werden. In MBASIC stehen jedoch als Datenstrukturen sogenannte *Arrays* beziehungsweise *Tabellen* zur Verfügung, die zeitweise auch unter den Namen Felder, Bereiche oder Listen anzutreffen sind. In MBASIC sind dabei bis zu 255 Dimensionen möglich, wobei zur Deklaration der Dimension die ***DIM***-Anweisung genutzt wird. Hierzu einige Beispiele. Nachfolgend ist die Variable prim über die *dim*-Anweisung als 1-dimensionales Array deklariert worden, wobei auf die einzelnen Elemente über den in runden Klammern stehenden Index zugegriffen werden kann. Bei einem Wert 5, den ich hier verwendet habe, werden 6 Elemente (0 bis 5) im Speicher reserviert.

```
dim prim(5)
Ok
prim(0)=2:prim(1)=3:prim(2)=5:prim(3)=7:prim(4)=11:prim(5)=13
Ok
? prim(0)
 2
Ok
?prim(3)
 7
Ok
?prim(6)
Subscript out of range
Ok
```

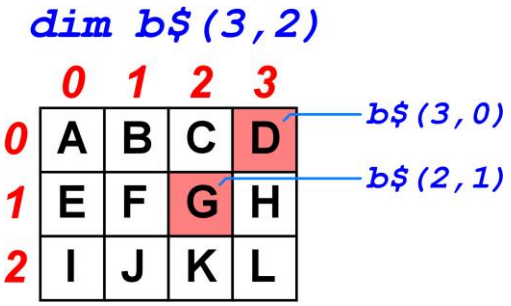
Über ***dim prim(5)*** wird das 1-dimensionale Array mit den nachfolgenden Initialisierungen wie folgt angelegt.



Wird versucht, auf ein Element außerhalb der erlaubten Dimension zuzugreifen, erscheint eine entsprechende Fehlermeldung mit dem Hinweis *Out-Of-Range*. Nachfolgend ist ein 2-dimensionales Array deklariert worden.

```
dim b$(3,2)
Ok
b$(0,0)="A":b$(1,0)="B":b$(2,0)="C":b$(3,0)="D"
Ok
b$(0,1)="E":b$(1,1)="F":b$(2,1)="G":b$(3,1)="H"
Ok
b$(0,2)="I":b$(1,2)="J":b$(2,2)="K":b$(3,2)="L"
Ok
?b$(2,1)
G
Ok
?b$(3,0)
D
Ok
```

Über ***dim b$(3,2)*** wird das 2-dimensionale Array mit den nachfolgenden Initialisierungen wie folgt angelegt.



In gleicher Weise kann auch ein 3-dimensionales Array angelegt werden, wobei ich mir die Erläuterungen hierbei erspare.

4.5 Kommentare

Natürlich ist es auch in MBASIC möglich, Kommentare zu hinterlegen, die über das Schlüsselwort ***REM*** für *Remark* eingeleitet werden und sich auf die ganze Zeile erstrecken. In den Zeile 10 und 20 sind Kommentare hinterlegt und werden bei der Ausführung des Programms nicht berücksichtigt, so dass nur die Zeilen 30 und 40 das eigentliche Programm repräsentieren.

```
10 REM Das ist eine Kommentarzeile
20 REM und diese auch noch!
30 A = 17
40 PRINT A
Ok
```

4.6 Schleifen

Wir wissen mittlerweile, was Schleifen sind, und wozu sie verwendet werden. Programme, die Schleifen enthalten bilden sogenannte *Kontrollstrukturen*. Diese Kontrollstrukturen werden für die Ablaufsteuerung in Programmen benötigt und sind ein essentieller Bestandteil der strukturierten beziehungsweise imperativen Programmierung.

Die Ablaufsteuerung basiert auf Schleifen (z.B. FOR-NEXT) und Verzweigungen (z.B. IF-THEN), die über logische Ausdrücke der booleschen Algebra gesteuert werden.

4.6.1 Die FOR-NEXT-Schleife

Bleiben wir beim ersten Punkt, den Schleifen und sehen uns dazu die ***FOR-NEXT***-Schleife an. Es wird über die Schleife die sogenannte *Laufvariable i* je Schleifendurchlauf (Iteration) um den Wert 1 erhöht und das Programm gibt diesen Wert inklusive der Verdopplung in einer Zeile aus.

```
10 FOR I=0 TO 10
20 PRINT I, 2*I
30 NEXT I
Ok
run
 0            0
 1            2
 2            4
 3            6
 4            8
 5            10
 6            12
 7            14
 8            16
 9            18
 10           20
Ok
```

Wird der Wert von *i* um den Wert 1 erhöht und überschreitet dieser die festgelegte Grenze 10, wird die Schleife verlassen. Gibt man den Befehl ***print i*** nach der Beendigung des Programms ein, wird man sehen, dass diese den Wert 11 besitzt. Möchte man, dass der Inhalt der Laufvariablen i um einen anderen Wert als 1 erhöht wird, muss diese Schrittweite über den Zusatz *step* angegeben werden, wobei auch Runterzählen möglich ist wie das nachfolgend zu sehen ist. In diesem Fall wird von 10 runter nach 0 gezählt, wobei der Startwert natürlich größer als der Endwert und die Schrittweite ein negativer Wert sein muss.

```
10 FOR I=10 TO 0 STEP -2
20 PRINT I, I*I
30 NEXT I
Ok
run
 10           100
 8            64
 6            36
 4            16
 2            4
 0            0
Ok
```

Bei einer FOR-NEXT-Schleife sehen wir, dass die Anzahl der Iterationen im Schleifenkopf fest definiert wurde. Es steht also vorher schon fest, wie oft die einzelnen Befehle im Anweisungsblock zwischen dem Schleifenkopf *FOR* und dem Schleifenende *NEXT* ausgeführt werden. Das ist jedoch nicht immer erwünscht, denn wenn es zum Beispiel über Berechnungen eines Näherungswertes geht, steht die Anzahl der Berechnungen nicht fest. Dazu kann die sogenannte *WHILE-WEND*-Schleife verwendet werden.

4.6.2 Die WHILE-WEND-Schleife

Die nächste Schleife, mit der wir uns also befassen, ist die *WHILE-WEND*-Schleife. Der Anweisungsblock dieser Schleife wird solange durchlaufen und ausgeführt, solange (*while* bedeutet übersetzt *solange*) die aufgeführte Bedingung erfüllt, also wahr ist. Man kann eine *FOR-NEXT*-Schleife natürlich auch mit einer *WHILE-WEND*-Schleife realisieren, wie wir das am folgenden Beispiel sehen können.

```
10 WHILE I <= 10
20 PRINT I
30 I = I + 1
40 WEND
Ok
run
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
 10
Ok
```

Es wird die Laufvariable *i* innerhalb des Schleifenkörpers modifiziert und nach jedem Durchlauf durch die im Schleifenkopf definierte Bedingung auf Wahrheitsgehalt hin überprüft. Würde die Berechnung in Zeile 30 irgendwie fehlerhaft sein und der Wert durch Manipulation zu keinem Zeitpunkt größer 10 werden, hätten wir es mit einer Endlosschleife zu tun, die niemals je verlassen würde, weil niemals eine Abbruchbedingung eintreten würde.

4.7 Freien Speicherplatz ermitteln

In MBASIC gibt es zur Überprüfung des freien Speicherplatzes die Funktion ***FRE(0)***, die als Ergebnis eine Zahl zurückliefert, die für die Anzahl der noch zur Verfügung stehenden Bytes steht.

```
A0>mbasic
BASIC-85 Rev. 5.29
[CP/M Version]
Copyright 1985-1986  $  by Microsoft
Created: 28-Jul-85
39224 Bytes free
Ok
?fre(0)
 39224
Ok
```

Es ist zu sehen, dass der angezeigte Wert von 39224 genau dem Wert entspricht, den auch MBASIC unmittelbar nach seinem Start über die Zeile ***39224 Bytes free*** ausgibt. Wird jetzt zum Beispiel ein Program geladen, das natürlich einen gewissen Speicherbereich belegt, steht im Anschluss weniger zur Verfügung. Angenommen, es werden die beiden Programmzeilen eingegeben, so ist nach der gezeigten Differenzbildung zu sehen, dass das Programm 19 Bytes beansprucht.

```
10 A = 17
20 PRINT A
Ok
? 39224 - fre(0)
 19
Ok
```

4.8 Auf Speicherinhalte Zugreifen

Wer die anfängliche Zeit der Heimcomputer mitgemacht hat und zum Beispiel einen *Apple II* oder *Commodore C64* sein Eigen nannte, kennt sicherlich die *Peeks* und *Pokes* noch, mit denen man so einiges an den Rechnern anstellen

konnte. Es handelt sich dabei um Befehle, die es einem Erlauben, direkt einen Speicheradresse auszulesen oder sie zu manipulieren.

- ***Peek***: Es bedeutet übersetzt *gucken* und gibt den Inhalt einer Speicherstelle an.
- ***Poke***: Es bedeutet übersetzt *stecken* und erlaubt es, eine Speicherstelle mit einem Wert zu versehen

Die allgemeine Schreibweise lautet wie folgt lautet.

- wert = ***peek***(adresse)
- ***poke*** adresse, wert

Wollen wir doch einmal sehen, wie wir mit *peek* und *poke* einen Wert lesen und auch ändern können. Mit der folgenden Zeile wird eine Ganzzahl-Variable mit einem Wert versehen.

```
a% = 17
Ok
```

Nun werde ich über den *peek*-Befehl und einer bestimmten Speicheradresse (ich sage noch nicht, wie ich darauf gekommen bin), einen Wert anzeigen lassen. Und siehe da, an Speicherstelle 25048 befindet sich der Wert 17! Zufall?

```
? peek(25048)
 17
Ok
```

Wir wollen jetzt diesen Wert einmal über *poke* verändern und sehen, was dann passiert. Mit ***poke 25048, 18*** wird die Speicherstelle mit dem Wert ***18*** überschieben, was im Anschluss auch mit *peek* überprüft werden kann. Doch wenn wir den Inhalt der Variablen *a%* über ihren Namen anzeigen lassen, ist dort ebenfalls der Wert 18 zu sehen. Wir haben also wirklich den Inhalt der Variablen a% über die Verwendung von *poke* verändert.

```
poke 25048, 18
Ok
? peek(25048)
 18
Ok
? a%
 18
Ok
```

Wie aber habe ich den Wert der betreffenden Speicherstelle ermitteln können, an der die Variable a% den Wert abgelegt hat? Kommen wir in diesem Zusammenhang zum nächsten Thema.

4.9 Zeiger auf Variablen

Wenn Werte im Speicher abgelegt werden, müssen diese natürlich in irgendeiner Weise wieder zugänglich gemacht werden, denn ansonsten macht es absolut keinen Sinn, wenn diese nicht mehr wiedergefunden werden können. Bleiben wir diesbezüglich bei den Variablen. Eine Variable wird über ihren vergebenen Namen referenziert und bei der Angabe des Namens erfolgt der Lese-

und Schreibzugriff auf diese Variable. Ist es zum Beispiel erforderlich, über Maschinensprache darauf zuzugreifen, wird natürlich die Adresse der Variablen benötigt, unter der sie einen Wert gespeichert hat. Nun müssen wir dabei eines beachten. Wir haben es hier mit einen 8-Bit-System zu tun und mit 8 Bits können lediglich Werte im Bereich von 0 is 255 gespeichert werden. Das reicht also für größere Werte nicht aus und bedarf einer Erweiterung auf weitere Bytes. Wir haben ja schon gesehen, dass unterschiedliche Datentypen eine gewisse Anzahl von Bytes in Anspruch nehmen und so ist es natürlich auch bei ganzzahligen Werten. Gleich dazu mehr.

Kommen wir auf unsere Ganzzahl-Variable *a%* zurück, die irgendwo im Speicher abgelegt wurde. Nun gibt es eine Funktion, die die Startadresse im Speicher zurückliefert, an der dieser Wert gespeichert wurde. Sie lautet ***varptr*** und ist die Abkürzung für Variable-Pointer, also ein Zeiger (englisch: *pointer*) auf eine Variable. Mal sehen, was diese Funktion für unsere Variable *a%* ausgibt.

```
? varptr(a%)
 25048
Ok
```

Und siehe da, es handelt sich genau um die von mir verwendete Speicheradresse. Ich möchte die Sache etwas vertiefen und wir rekapitulieren noch einmal die Grundlagen. Wir wissen, dass zur Speicherung einer Variablen des Datentyps *Integer* 2 Bytes erforderlich sind. Mit 2 Bytes kann der Wertebereich von 0 bis 65.535 abgedeckt werden. Doch stopp, denn irgendetwas scheint hier nicht ganz zu stimmen, wenn es darum geht, zum Beispiel den obersten Wert zu speichern. Sehen wir uns das an.

```
a% = 65535
Overflow
Ok
```

Es wird hier ein sogenannter Overflow, also ein Überlauf angezeigt, denn es ist nicht möglich, diesen Wert in 2 Bytes unterzubringen. Komisch, denn es sollte doch möglich sein. Die Ursache des Problems liegt darin begründet, dass es nicht nur positive, sondern auch negative Ganzzahlen gibt und die müssen ebenfalls in die Datenbreite von 2 Bytes passen. Wir kennen das Verfahren zu Kennzeichnung negativer Werte bei der Maschinensprache schon. Ich sage nur *MSB* (Most-Significant-Bit). Das bedeutet also, dass zur Speicherung von Ganzzahl-Werten der Wertebereich von -32.768 bis +32.767 zur Verfügung steht.

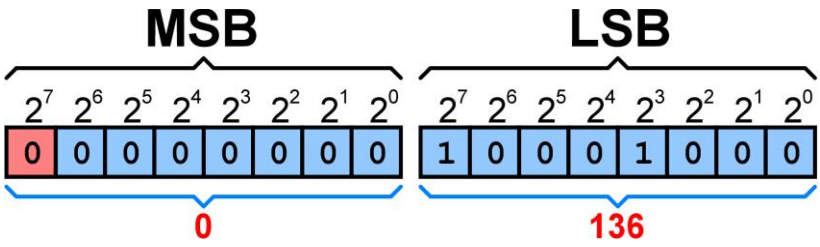
Sehen wir uns in diesem Zusammenhang das folgende kleine Programm an. Es wird eine eingegebene Zahl in der Ganzzahl-Variablen *a%* gespeichert und im Anschluss in Zeile 30 die Startadresse über die ***VARPTR***-Funktion (Variable-Pointer) im Speicher ermittelt. Im Anschluss erfolgt in Zeile 40 die Ausgabe dieser Adresse. Da es sich um einen Zeiger handelt, der auf das *LSB* (Least-Significant-Byte) weist, wird in Zeile 50 zur Ermittlung von *LSB* und *MSB*, wobei sich das MSB unmittelbar hinter dem LSB befindet. In den Zeile 60 und 70 werden die Werte, die im LSB und MSB gespeichert sind, über *peek* ermittelt und zur Anzeige gebracht. Die Berechnung des eingegebenen Wertes über LBS und MSB ist in Zeile 80 zu sehen.

```
10 INPUT "Wert: ", A%
20 PRINT "Eingegebene Ganzzahl: "; A%
30 ADR% = VARPTR(A%)
40 PRINT "Die Zahl: "; A% ; " belegt 2 Bytes ab Adresse: "; ADR%
50 LSB = PEEK(ADR%) : MSB = PEEK(ADR% + 1)
60 PRINT "Unter LSB ist:    "; LSB ; " gespeichert"
70 PRINT "Unter MSB ist:    "; MSB ; " gespeichert"
80 PRINT "Der Wert lautet: "; LSB + MSB * 256
Ok
```

Wir sehen uns jetzt zwei Beispiele dazu an. Bei einem Wert von *136* schaut die Ausgabe wie folgt aus.

```
Eingegebene Ganzzahl:  136
Die Zahl:  136  belegt 2 Bytes ab Adresse:  25352
Unter LSB ist:    136  gespeichert
Unter MSB ist:    0  gespeichert
Der Wert lautet:  136
Ok
```

Sehen wir uns das mit den beiden Bytes *LSB* und *MSB* im Detail an. Im Speicher schaut es wie folgt aus, wobei die Adresse erst einmal keine Rolle spielt.



Die Berechnung des eigentlichen Wertes erfolgt wie schon erwähnt über

Wert = LSB + MSB x 256

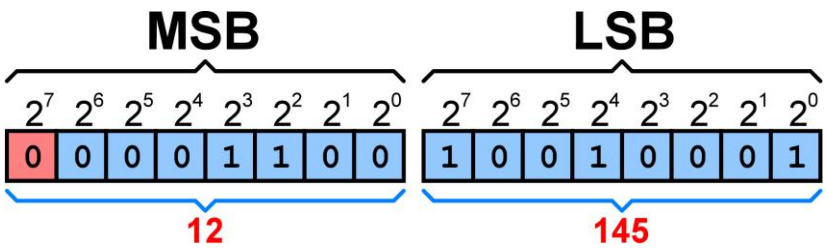
Konkret für unser Beispiel ist das

Wert = 136 + 0 x 256 = 136

Sehen wir uns einen Wert größer 255 an, denn dann muss das MSB mit hinzugezogen werden.

```
Wert: 3217
Eingegebene Ganzzahl:  3217
Die Zahl:  3217  belegt 2 Bytes ab Adresse:  25352
Unter LSB ist:    145  gespeichert
Unter MSB ist:    12  gespeichert
Der Wert lautet:  3217
Ok
```

Wie schaut das im Speicher aus.



Die Berechnung des eigentlichen Wertes erfolgt über

Wert = 145 + 12 x 256 = 3217

Wie schaut es aber mit negativen Werten aus? Ich gebe dazu einmal den Wert -84 ein.

```
Wert: -84
Eingegebene Ganzzahl: -84
Die Zahl: -84  belegt 2 Bytes ab Adresse:  25352
Unter LSB ist:    172  gespeichert
Unter MSB ist:    255  gespeichert
Der Wert lautet:  65452
Ok
```

Oh, das ist wohl nicht korrekt! -84 stimmt nun wirklich nicht mit +65452 überein. Warum ist das so? Nun, wir kennen ja schon das Vorzeichenbit, das an höchster Stelle des MSB zu finden ist. Wir sollte das einmal unter die Lupe nehmen und dazu unser Programm etwas modifizieren.

```
10 INPUT "Wert: ", A%
20 PRINT "Eingegebene Ganzzahl: "; A%
30 ADR% = VARPTR(A%)
40 PRINT "Die Zahl: "; A% ; " belegt 2 Bytes ab Adresse: "; ADR%
50 LSB = PEEK(ADR%) : MSB = PEEK(ADR% + 1)
60 PRINT "Unter LSB ist:    "; LSB ; " gespeichert"
70 PRINT "Unter MSB ist:    "; MSB ; " gespeichert"
80 IF (MSB AND 128) > 0 THEN M = 1
90 PRINT "Vorzeichenbit:    "; M
100 IF M = 0 THEN PRINT "Der Wert lautet: "; LSB + MSB * 256 : END
110 IF M = 1 THEN PRINT "Zweier-Kompl.:    ";(LSB XOR 255) + (MSB XOR 255)*256+1
Ok
```

In Zeile 80 wird über die *AND*-Verknüpfung mit dem Wert 128 (binär: 10000000) ermittelt, ob das höchstwertige Bit gesetzt ist. Ist das der Fall, so wird die Variable *M* mit dem Wert *1* versehen. Falls *M* gleich 0 sein sollte, wird in Zeile 100 der Wert wie bisher angezeigt. Ist er jedoch gleich 1, was auf einen negativen Wert hindeutet, so wird in Zeile 110 das Zweierkomplement gebildet, was über die Invertierung der Bits von *LSB* und *MSB* über die *XOR*-Funktion mit dem Wert 255 und der abschließenden Addition von 1 erfolgt. Wer sich darüber nicht mehr im Klaren sein sollte, muss noch einmal einen Blick in das *Flag-Details*-Kapitel werfen.

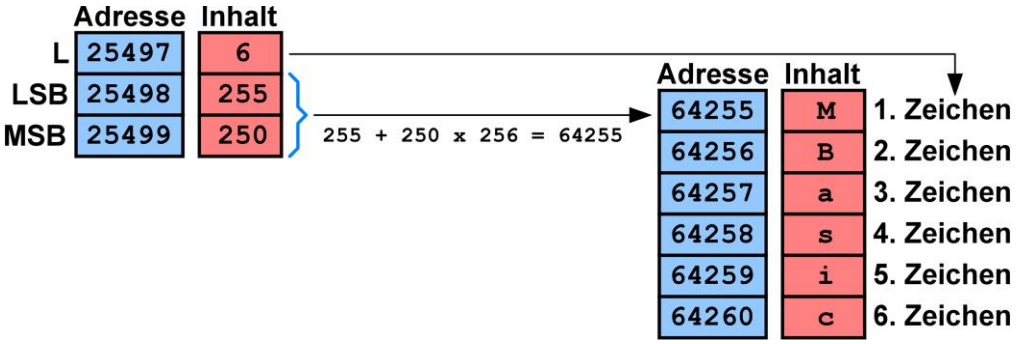
Wir können das ganze Spiel natürlich auch mit einer Zeichenkette ausprobieren und nachsehen, wie diese im Speicher abgelegt wurde. Das folgende Programm wird beim Verständnis helfen.

```
10 INPUT "Text: "; T$
20 PRINT "Eingegebener Text: "; T$
30 ADR% = VARPTR(T$)
40 PRINT "Der Text: "; T$ ; " belegt 3 Bytes ab Adresse: "; ADR%
50 L% = PEEK(ADR%): LSB = PEEK(ADR% + 1): MSB = PEEK(ADR% + 2)
60 PRINT "Unter L ist:    "; L% ; " gespeichert"
70 PRINT "Unter LSB ist:  "; LSB ; " gespeichert"
80 PRINT "Unter MSB ist:  "; MSB ; " gespeichert"
90 FOR I = 0 TO L% - 1
100 ADRSTR = LSB + MSB * 256
110 PRINT "Unter Adr.:" ; ADRSTR + I ; "befindet sich: "; CHR$(PEEK(ADRSTR+I))
120 NEXT I
Ok
```

Sehen wir uns jetzt die Ausgaben des Programms an, wenn ich den Text ***MBasic*** eingebe.

```
Text: ? Mbasic
Eingegebener Text: Mbasic
Der Text: Mbasic belegt 3 Bytes ab Adresse:  25497
Unter L ist:     6  gespeichert
Unter LSB ist:   255  gespeichert
Unter MSB ist:   250  gespeichert
Unter Adr.: 64255 befindet sich: M
Unter Adr.: 64256 befindet sich: b
Unter Adr.: 64257 befindet sich: a
Unter Adr.: 64258 befindet sich: s
Unter Adr.: 64259 befindet sich: i
Unter Adr.: 64260 befindet sich: c
Ok
```

Das sollten wir uns wieder genauer ansehen. Zur Speicherung beziehungsweise Organisation werden diesmal 3 Bytes verwendet, wobei das erste Byte die Länge der Zeichenkette enthält, und die darauffolgenden zwei Bytes wieder den Zeiger über das LSB/MSB-Konstrukt, was diesmal jedoch nicht auf nur einen Wert zeigt, sondern wiederum auf eine Startadresse, die auf den ersten Buchstaben der Zeichenkette weist.



Mit diesen Informationen ist es recht einfach, die einzelnen Buchstaben abzurufen, die natürlich in Form von ASCII-Werten an den betreffenden Speicherstellen zu finden sind. Wir wollen jedoch nicht den ASCII-Wert zur Anzeige bringen, sondern das entsprechende ASCII-Zeichen. Zur Umwandlung wird die ***CHR$***-String-Funktion verwendet, die in Zeile 110 zu sehen ist. Die allgemeine Syntax lautet:

CHR$(code)

Zur Anzeige der Zeichenkette aus dem Speicher liegen alle Informationen vor. Die Startadresse des ersten Zeichens der Zeichenkette und deren Länge. Zur Anzeige wird die FOR-NEXT-Schleife verwendet, die mit der Laufvariablen *i* als *Offset*-Wert in Verbindung mit der Startadresse der Zeichenkette alle einzelnen Buchstaben auswählt, was in den Zeilen 90 bis 120 erfolgt. Da sich die Länge der Zeichenkette in einem einzigen Byte befindet, sollte klar sein, dass die Länge begrenzt und auf 255 einzelne Zeichen beschränkt ist.

4.10 Der Maschinencode

Wir habe im entsprechenden Kapitel gesehen, wie man in Assembler programmiert. Nun ist es in MBASIC nicht so einfach, auch aus dieser Basic-Umgebung heraus in Maschinensprache zu programmieren und doch gibt es da einen Weg. Ich möchte zu diesem Zweck das Beispiel aufgreifen, wo es darum ging, ein einzelnes Zeichen über den BDOS-Aufruf auf der Konsole anzeigen zu lassen. Sehen wir uns dazu noch einmal den Assemblercode an.

```
te: OUT.ZSM                                --- | Lin:0006/0006/0512 Col:01/76 Len:31
   !...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...
  1|     org 0100h  ; free memory
  2|BDOS equ 0005h  ; BDOS entry point
  3|     ld c,2     ; BDOS function print char
  4|     ld e,'a'   ; load char
  5|     call BDOS  ; CALL BDOS
  6|     ret        ; return to CCP
```

Hier können wir den Assemblercode sehen, doch wenn es darum geht, den Maschinencode in MBASIC zu übertragen, sehen wir hier nicht die entsprechenden Werte. Wir können uns aber die generierte *out.prn*-Datei einmal genauer ansehen, wobei ich den für uns wichtigen Bereich Rot umrandet habe.

```
C1>type out.prn

ZSM-3.4    Source file: OUT       Page No:    1

0100 =              org 0100h  ; free memory
0005 =         BDOS equ 0005h  ; BDOS entry point
0100 0E02           ld c,2     ; BDOS function print char
0102 1E61           ld e,'a'   ; load char
0104 CD0500         call BDOS  ; CALL BDOS
0107 C9             ret        ; return to CCP
0005 =
```

Um es etwas übersichtlicher zu machen, extrahiere ich die gewünschten Informationen in eine Tabelle.

Maschinencode (HEX)	**Maschinencode (DEC)**	**Mnemonics**
0E 02	14 02	LD c, 2
1E 61	30 97	LD e, 'a'
CD 05 00	205 05 00	CALL 5
C9	201	RET

Tabelle 10 - Assemblerprogramm out.prn

Wenn es jetzt eine Möglichkeit geben könnte, die Informationen aus der Maschinencode (DEC)-Spalte irgendwie speichern und nutzen zu können. Hier muss ich wieder ein wenig ausholen, denn in MBASIC können Zahlenreihen hintereinander abgelegt und dann später wieder abgerufen werden. Die beiden Befehle, die das ermöglichen, lauten *data* und *read*. In der Zeile 10 werden hinter dem Schlüsselwort ***DATA*** 6 Werte durch Komma getrennt aufgelistet. Diese Werte werden **innerhalb** der *FOR-NEXT*-Schleife in der Zeile 30 durch das Schlüsselwort *READ* gelesen und über die *PRINT*-Anweisung auf der Konsole angezeigt.

```
10 DATA 45, 2, -12, 34, 102, 9
20 FOR I = 1 TO 6
30 READ A : PRINT A
40 NEXT I
Ok
run
 45
 2
-12
 34
 102
 9
Ok
```

Es ist dabei zu beachten, dass niemals mehr READ-Anweisungen erfolgen, wie Daten hinterlegt wurden, denn das würde unweigerlich zu einem Fehler führen. Angenommen wir würden in Zeile 30 die *FOR-NEXT*-Schleife hinsichtlich des Endwertes der Laufvariablen auf 7 ändern, so würde es zu besagtem Fehler

führen, der da lautet ***Out of DATA in 30***, weil keine zu lesenden Daten mehr vorhanden sind. Die Speicherung von Daten auf diese Weise ist recht einfach, birgt jedoch einen entscheidenden Nachteil in sich, denn DATA-Anweisungen sind ohne weitere Informationen zu deren Hintergrund schwierig zu entschlüsseln oder zu verstehen, da sie wie eine zufällige Zahlengruppe aussehen.

DATA-Anweisungen können an beliebiger Stelle in einem Programm stehen. Sie werden ignoriert, bis eine entsprechende *READ*-Anweisung ausgeführt wird. Der Computer merkt sich über einen Zeiger in Form einer Markierung wo das zuletzt ausgeführte Lesen stattgefunden hat, so dass er immer genau informiert ist, welches Datenelement zuletzt eingelesen wurde und weiß daraufhin genau, auf welches Element sich die nächste *READ*-Anweisung beziehen wird.

Kommen wir zurück zu unserem Vorhaben der Umsetzung des Maschinenprogramms in MBASIC. Wir habe also gesehen, dass das Programm *out.zsm* zur Anzeige eines einzelnen Zeichens auf der Konsole aus der folgenden Zahlenfolge zusammengesetzt ist.

14, 02, 30, 97, 205, 05, 00, 201

Nun brauchen wir uns in diesem Fall keine Gedanken darüber machen, dass dieser Code auch an der richtigen Stelle im Speicher abgelegt wird, um die Funktion zu gewährleisten. Wir wissen, dass ein Maschinenprogramm im herkömmlichen Sinne ab Adresse ***0100h*** zu liegen hat, was hier nicht zutrifft. Warum das so ist, werden wir in Kürze sehen, denn MBASIC bietet dafür eine komfortable Funktion an, die wir nutzen können.

Um aber die einzelnen Werte in Form einer Datenkette im Speicher abzulegen, nutzt man am besten eine Zeichenkette, die bei jedem Durchlauf des Lesens der Werte immer um einen Buchstaben erweitert wird. Damit das machbar ist, kommt die ***CHR$***-Funktion zum Einsatz, die einen Integerwert in ein entsprechendes ASCII-Zeichen wandelt und die wir schon kennengelernt hatten. Da wir aber unterschiedliche Maschinenprogramme sehr flexibel handhaben wollen, sollten wir einen Mechanismus etablieren, der uns signalisiert, dass das Ende des Codes erreicht ist, was am besten über einen negativen Integerwert erfolgt, denn dieser kommt bei Maschinencodes nicht vor. Ein weiterer Punkt ist natürlich die Speicheradresse (Anfangsadresse), an der die Daten der Zeichenkette zu finden sind. Auch das haben wir schon gesehen und mittels der *VARPTR*-Funktion realisiert.

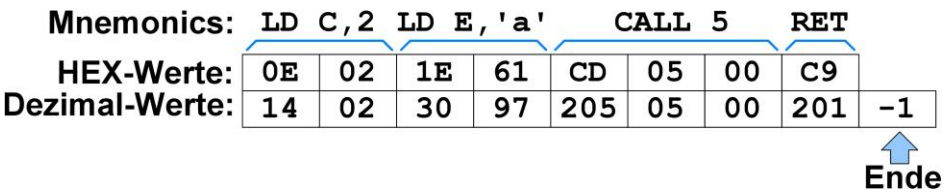
Wenn sich also jetzt ein Maschinenprogramm irgendwo im Speicher befindet, das von MBASIC aus aufgerufen werden soll, so ist noch ein weiterer Schritt erforderlich. Es muss über eine USR-Funktion dem System mitgeteilt werden, dass an besagter Stelle die Startadresse des Maschinenprogramms zu finden ist. Das erfolgt mit der Definition einer User-Funktion in Form von

```
DEF USRx = ADR
```

Das *x* steht dabei für Funktionen von 0 bis 9, die definiert werden können. Das klingt alles sehr theoretisch, so dass ich am besten mit unserem konkreten Beispiel beginne und es schrittweise aufrolle.

4.10.1 Die Definition des Codes

Wie schon gesagt, werden die Daten, die dem Maschinencode entsprechen, über die *DATA*-Anweisung hinterlegt. Nachfolgend sind die Assembler-Befehle mit ihrem entsprechenden Code zu sehen. Die erwähnte Markierung über den negativen Wert -1 gehört natürlich nicht mit zum eigentlichen Maschinenprogramm.



Die Umsetzung in MBASIC schaut wie folgt aus, wobei ich den eigentlichen Code noch einmal als Kommentarzeilen hinzugefügt habe.

```
10 REM 0E 02     ld c,2      ; BDOS function print char
20 REM 1E 61     ld e,'a'    ; load char
30 REM CD 05 00  call 5      ; CALL BDOS
40 REM C9        ret         ; return to CCP
50 DATA 14, 02, 30, 97, 205, 05, 00, 201, -1
60 REM * load maschinecode to memory *
70 C$ = "": READ A: WHILE A>-1 : C$ = C$ + CHR$(A) : READ A : WEND
```

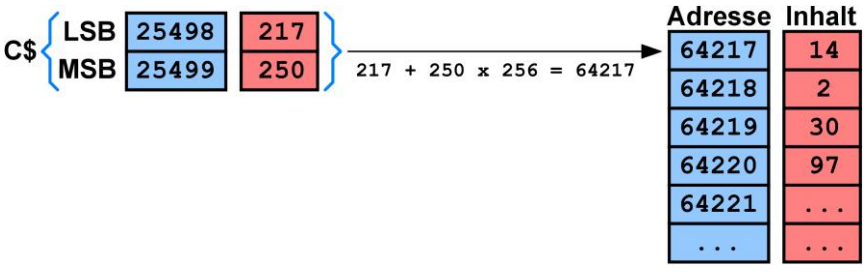
Das Einlesen der Daten in die Variable ***C$*** erfolgt in Zeile 70 über das einmalige Lesen vor der *WHILE-WEND*-Schleife und dann innerhalb der Schleife solange, bis der Wert -1 gelesen wurde.

4.10.2 Die Stelle im Speicher

Nun ist es wichtig zu wissen, an welcher Stelle im Speicher sich die Startadresse der Zeichenkette befindet. Wir rekapitulieren noch einmal, dass zur Speicherung von Zeichenketten 3 Bytes erforderlich sind. Das erste Byte gibt die Länge der Zeichenkette zurück und die beiden folgenden bilden einen Zeiger in Form von *LSB* und *MSB*, so dass die Startadresse von ***C$*** wie folgt ermittelt wird.

```
80 ADR = PEEK(VARPTR(C$) + 1) + PEEK(VARPTR(C$) + 2) * 256
```

Das schaut dann wie folgt aus.



Man kann sich jetzt die abgelegten Daten im Speicher ansehen, was ich über die folgenden Zeilen exemplarisch für die ersten vier Werte gemacht habe. Die Startadresse von *C$* für die einzelnen Zeichen ist also 64217 und dort befindet sich auch der erste Wert *14*.

```
? adr
 64217
Ok
? peek(64217):? peek(64217+1):? peek(64217+2):? peek(64217+3)
 14
 2
 30
 97
Ok
```

4.10.3 Die Definition der Maschinenfunktion

Nun kommt es zur Definition der *USR*-Funktion in Zeile 100 und deren späterer Aufruf über Zeile 110.

```
100 DEF USR0 = ADR
110 X = USR0(0)
```

Über ***DEF USR0*** wird die Funktion mit der Startadresse des Maschinenprogramms initialisiert, damit der Aufruf in Zeile 110 erfolgreich durchgeführt werden kann. Das Argument 0 beim Aufruf von ***USR(0)*** dient lediglich der notwendigen Angabe eines Wertes, der jedoch nicht ausgewertet wird.

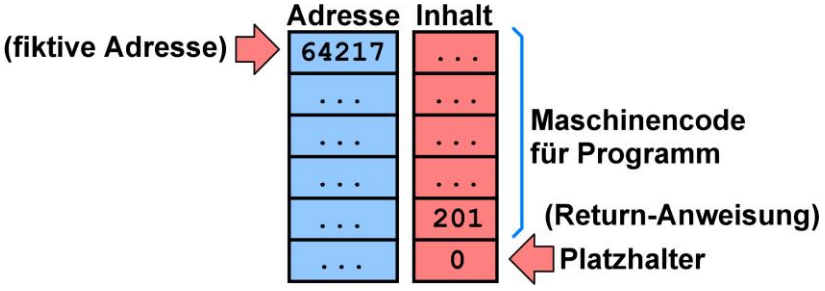
Nach dem Starten des MBASIC-Programms erscheint auf der Konsole das kleine *a*, gefolgt vom *OK*-Prompt von MBASIC.

```
run
aOk
```

4.10.4 Eine Parameterübergabe

Bisher haben wir mit dem festen Wert für den Buchstaben a gearbeitet, doch es wäre sicherlich schöner, könnte man das etwas flexibler handhaben. Wir konnten Sehen, dass der Aufruf der *USR0*-Funktion mit dem Dummy-Wert 0 erfolgte, weil dieser nicht ausgewertet wurde. Nun können wir aber eine Abfrage über *INPUT* in MBASIC realisieren, wo ein Wert in einer Variablen gespeichert wird, der wiederum an einer bestimmten Stelle im Speicher abgelegt wird. Der Inhalt dieser Speicherstelle wird jetzt beim Aufruf der *USR0*-Funktion genutzt, wobei lediglich die Speicheradresse übergeben wird. Sehen wir uns das genauer an.

Um den eingegebenen Wert zu speichern, nutzen wir einfach die Speicherstelle, die sich hinter dem letzten Maschinenbefehl befindet, der ja durch die *RET*-Anweisung gebildet wird. Wir fügen dem Maschinencode einfach den Wert 0 hinzu, der quasi als Platzhalter für den später einzugebenden Wert dient. Das schaut dann ungefähr wie folgt aus.



Da wir wissen, wie viele Bytes das eigentlich Maschinenprogramm benötigt, können wir die Speicheradresse für den Platzhalter sehr einfach bestimmen. Das wäre die erste Hürde, die es zu überwinden galt. Die zweite besteht darin, eine Technik zu finden wie der eigegebene Wert denn in das E-Register kommt, das ja das anzuzeigende Zeichen vor dem BDOS-Aufruf enthalten soll. Jetzt kommt das *HL*-Register ins Spiel, das beim ***USR0***-Aufruf den Inhalt des Argumentes an dieses Register übermittelt. Im Maschinencode kann dann dieses Registerpaar ausgelesen werden, um die Informationen an das E-Register zu übertragen. Nachfolgen ist das entsprechende Assemblerprogramm zu sehen, das wir gleich mit MBASIC realisieren wollen.

```
ZSM-3.4    Source file: OUTC       Page No:    1

0100 =              org 0100h  ; free memory
0005 =         BDOS equ 0005h  ; BDOS entry point
0100 E5             push hl    ; save HL-Register
0101 0E02           ld c,2     ; BDOS function print char
0103 5E             ld e,(hl)  ; load char
0104 CD0500         call BDOS  ; CALL BDOS
0107 E1             pop hl     ; restore HL-Register
0108 C9             ret        ; return to CCP
0005 =
```

Um es etwas übersichtlicher zu machen, extrahiere ich wieder die gewünschten Informationen in eine Tabelle.

Maschinencode (HEX)	**Maschinencode (DEC)**	**Mnemonics**
E5	229	PUSH HL
0E 02	14 02	LD c, 2
5E	94	LD E, (HL)
CD 05 00	205 05 00	CALL 5
E1	225	POP HL
C9	201	RET

Tabelle 11 - Assemblerprogramm outc.prn

Wie wir das schon gelernt haben, sollte man vor der Nutzung von Registern über einen BDOS-Aufruf, diese über *PUSH* und *POP* sichern und wiederherstellen. Das geschieht über die Zeilen, in denen der Code *E5* und *E1* steht. Wir sollten uns jedoch jetzt ein wenig die Zeile mit dem Befehl ***LD E, (HL)*** ansehen. Es handelt sich um einen Ladebefehl, der das E-Register mit Informationen über das HL-Registerpaar versorgt. Beim E-Register handelt es sich um ein 8-Bit-Register und das HL-Registerpaar besitzt eine Breite von 16 Bit. Es ist also nicht möglich, den Inhalt von HL in E zu laden. Was sollte das aber bedeutet? Die runden Klammern deuten auf etwas hin, das mit einer speziellen Adressierungsart des Z80-Prozessors zu tun hat. Es handelt sich um die sogenannte *indirekte Adressierung*. Es bedeutet, dass nicht der Inhalt des HL-Registerpaares an das E-Register übertragen wird, was aus genannten Gründen ja nicht geht, sondern es wird der Inhalt der Speicherstelle, die durch das HL-Registerpaar adressiert wird, in das E-Register geladen.

Sehen wir uns jetzt das entsprechende MBASIC-Programm an.

```
10 REM E5        push hl     ; save HL-Register
20 REM 0E 02     ld c,2      ; BDOS function print char
30 REM 05        ld e, (hl) ; load char
40 REM CD 05 00  call 5      ; CALL BDOS
50 REM E1        pop hl      ; restore HL-Register
60 REM C9        ret         ; return to CCP
70 DATA 229, 14, 02, 94, 205, 05, 00, 225, 201, -1
80 INPUT "ASCII-Wert: ", V
90 REM * load maschinecode to memory *
100 C$ = "": READ A: WHILE A>-1 : C$ = C$ + CHR$(A) : READ A : WEND
110 ADR = PEEK(VARPTR(C$) + 1) + PEEK(VARPTR(C$) + 2) * 256
120 POKE ADR + 9, V
130 REM * USR0 = BDOS-CALL *
140 DEF USR0 = ADR
150 X = USR0(PEEK(ADR + 9))
Ok
```

Es ist hier zu sehen, dass die Variable *V* in Zeile 80 den Wert für das darzustellende ASCII-Zeichen aufnimmt. Dieser Wer wird über ***POKE ADR + 9, A*** an die Adresse hinter der *RET*-Anweisung geschrieben. Beim Aufruf der *USR0*-Funktion in Zeile 150 wird der zu übergebende Wert aus dieser Speicherstelle mittels PEEK(ADR + 9) abgerufen und anstelle der zuvor dort vorhandenen 0 eingesetzt. Ich gebe nachfolgend den ASCII-Wert *82* für den Buchstaben *P* ein, der nach dem Start auch vor dem *Ok*-Prompt zu erkennen ist.

```
run
ASCII-Wert: 82
ROk
```

5 Die Programmiersprache Turbo Pascal

Wenn es um eine Programmiersprache geht, an die ich persönlich aus meinem Studium auf der Technikerschule noch sehr gut erinnere, dann ist es *Turbo Pascal*. Diese Sprache hatte es mir angetan und ich war hin und weg, denn Programmieren hatte mich immer schon fasziniert und hier war so einiges Möglich, was mit Basic nicht machbar war. Im Jahre 1983 erschien Turbo Pascal 1.0 auf der Bühne und zu der Zeit waren natürlich die Betriebssysteme MS-DOS und CP/M Rechner weit verbreitet. Es handelt sich bei Turbo Pascal nicht um eine Interpretersprache, sondern eine Compilersprache. Es gibt sehr viele Aspekte, die diese Programmiersprache damals so attraktiv machte.

- Schnelles Compiling
- Generierter Maschinencode, der Programmen, die direkt in Assembler geschrieben waren, Konkurrenz machte
- Die Entwicklungsumgebung mit Editor, Linker und Compiler war schlank

Wir wollen in diesem Kapitel ein paar kleine Programme entwickeln, um ein Gefühl für Turbo Pascal zu bekommen. Unter RunCPM ist auf dem Laufwerk ***H:*** für ***User 3*** die Programmiersprache *Turbo Pascal* zu finden, so dass man nach einem Wechsel dorthin sofort loslegen kann. Nach der Eingabe von ***turbo*** in der Konsole, meldet sich Turbo Pascal in der Version *3.01A* wie folgt.

```
---------------------------------------
TURBO Pascal system       Version 3.01A
                           CP/M-80, Z80

Copyright (C) 1983,84,85   BORLAND Inc.
---------------------------------------

Terminal: ANSI


Include error messages (Y/N)? 
```

Wird hier als Antwort auf die Frage *Y* eingeben, so wird die Datei mit den aussagekräftigen Fehlermeldungen in den Speicher eingelesen. Um etwas Platz zu sparen, kann hier auch *N* eingegeben werden, was ca. 1,5 KByte Speicherplatz sparen würde. Im Anschluss meldet sich das Hauptmenü von Turbo Pascal und präsentiert sich wie folgt.

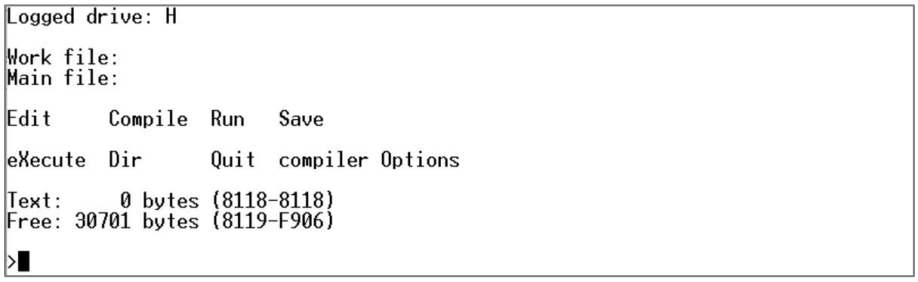


Abbildung 49 - Das Turbo Pascal Menü

Es werden alle zur Verfügung stehenden Kommandos angezeigt, wobei auf die zu sehenden Großbuchstaben in den einzelnen Befehlen zu achten ist. Es ist nicht erforderlich, die Auswahl mit der ***RETURN***-Taste zu bestätigen!

- **E**dit - Den Editor aufrufen
- **C**ompile - Das Compiling starten
- **R**un - Das Starten der Anwendung
- **S**ave - Die aktuelle Datei speichern
- e**X**ecute - Starten eines beliebigen Programms
- **D**ir - Inhaltsverzeichnis anzeigen
- **Q**uit - Anwendung verlassen
- compiler **O**ptions - Compiler Optionen

In der ersten Zeile ist zu sehen, auf welchem Laufwerk man quasi eingeloggt ist (hier *H:*) und im unteren Teil sind Informationen über den verbrauchten Speicher über den Text-Editor beziehungsweise des noch zur Verfügung stehenden Speicherplatzes zu finden. Um jetzt ein Programm in Form eines Quellcodes in den Editor zu laden, gehen wir wie folgt vor.

5.1 Den Quellcode laden

Über das Drücken der Taste ***W*** erscheint der folgende Dialog, wo wir hinter *Work file name* einfach *hello* eingeben. Darüber wird die zu bearbeitende Datei ausgewählt.

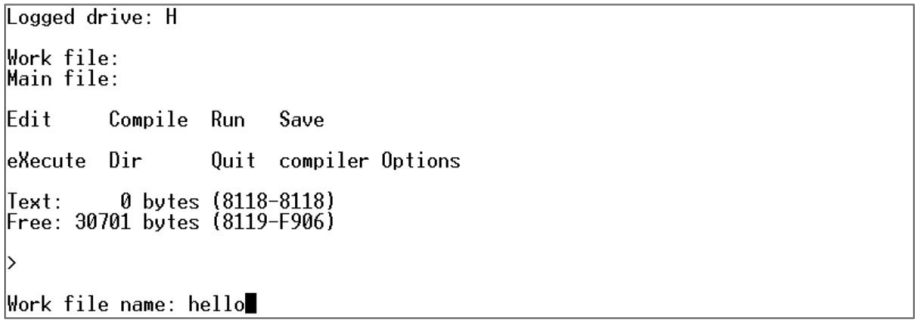


Abbildung 50 - Das Laden eines Quellcodes in den Editor

Die im Dateisystem vorhandene Datei mit dem Namen *hello.pas* besitzt für Turbo Pascal Quelldateien den Dateinamenzusatz *pas*, der jedoch nicht mit

angegeben werden muss. Nachfolgend erscheint die folgende Meldung und es ist zu sehen, dass die Erweiterung pas berücksichtigt wurde.

```
Work file name: hello

Loading H:HELLO.PAS
>
```

Wird jetzt die Taste ***E*** für Edit gedrückt, erscheint die Arbeitsdatei im Editor, was wie folgt ausschaut. Und *hey*, es gibt überhaupt keine Zeilennummern wie bei *MBASIC*. Es verhält sich hierbei ähnlich wie bei der Programmiersprache *C*, wo alles von oben nach unten abgearbeitet wird.

```
     Line 1    Col 1    Insert    Indent  H:HELLO.PAS
program HelloWorld;

begin
     writeln('Hello World!');
end.
```

Abbildung 51 - Der geladene Quellcode des Programms hello.pas

Wer jetzt versucht, mit den normalen Cursortasten zu navigieren, erlebt wieder böse Überraschungen, denn das ist hier nicht möglich. Um sich in horizontaler beziehungsweise vertikaler Richtung mit dem Cursor zu bewegen, muss über die nachfolgend gezeigten Tastenkombinationen gehen.

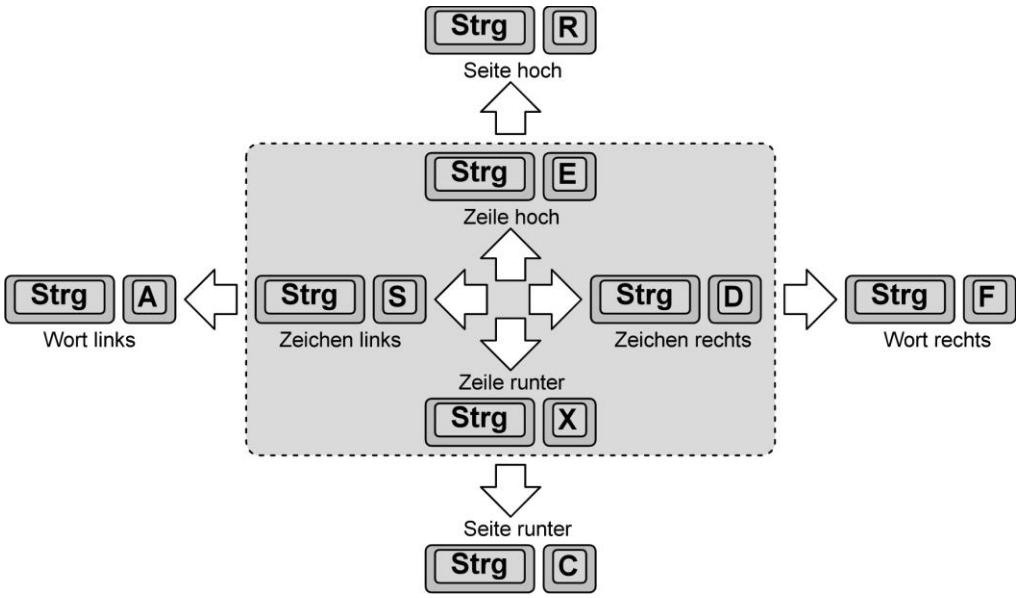


Abbildung 52 - Die Navigation im Editor

5.2 Das Kompilieren des Quellcodes

Wurde eine Quelldatei geladen oder über den Editor neu eingegeben, muss diese vor der Ausführung einer Kompilierung unterzogen werden, denn es handelt sich hier nicht wie bei der Programmiersprache *Basic* um eine Interpretersprache. Dieser Vorgang wird durch das Drücken der Taste ***C*** angestoßen. Doch zuvor muss natürlich der Editor über die folgende Tastenkombination verlassen werden.

Es erscheint am unteren Rand der Konsole das **>**-Prompt, was signalisiert, dass wir uns wieder im Hauptmenü befinden, wobei die einzelnen Menübefehle nicht sichtbar sind. Wer möchte, kann hier einfach die *RETURN*-Taste drücken und es erscheint alles wie gewohnt. Jetzt sind auch die Speicher-Informationen hinsichtlich der geladenen Datei und des Restspeichers zu sehen.

```
Logged drive: H

Work file: H:HELLO.PAS
Main file:

Edit     Compile  Run   Save

eXecute  Dir      Quit  compiler Options

Text:    67 bytes (8118-815B)
Free: 30634 bytes (815C-F906)

>
```

Abbildung 53 - Die geladene Quelldatei

Jetzt drücken wir die besagte Taste ***C*** und es erscheinen die folgenden Hinweise.

```
Compiling
  5 lines

Code:    53 bytes (815D-8192)
Free: 31595 bytes (8193-FCFE)
Data:     7 bytes (FCFF-FD06)

>
```

Die Quelldatei wurde ohne Probleme kompiliert und steht zur Ausführung bereit. Das Starten der COM-Datei kann dabei direkt über die Entwicklungsumgebung beziehungsweise über das Menü erfolgen. In diesem Fall war die COM-Datei als Beispiel schon vorhanden. Ich komme gleich noch einmal auf das Thema zu sprechen, wenn es darum geht, eine eigene Quelldatei zu kompilieren.

5.3 Die Ausführung des Programms

Über den Menüpunkt ***Run*** mittels Taste ***R*** (es geht auch mit dem kleinen ***r***) kann das zuvor kompilierte Programm ausgeführt werden und es erscheint die folgende Anzeige.

```
Running
Hello World!

>
```

Es handelt sich dabei genau um den Text, der sich in einfachen Anführungszeichen innerhalb der *writeln*-Funktion befindet. Doch dazu später mehr. Ich denke, dass es jetzt an der Zeit ist, dass wir uns ein wenig die grundlegende Struktur eines Turbo Pascal Programms ansehen sollten.

5.4 Die Struktur eines Turbo Pascal Programms

Ich denke, wir sollten uns noch einmal den gerade gesehenen Quelltext ansehen, doch zuvor werfen wir einen Blick auf die grundlegende Struktur eins Turbo Pascal Programms. Vorweg kann gesagt werden: In Turbo Pascal ist es vollkommen egal, ob Anweisungen groß oder klein geschrieben. Daher besteht keine Case-Sensitivität, also eine Unterscheidung zwischen Groß- und Kleinschreibung.

Der grundlegende Aufbau des Programms muss wie folgt aussehen, wobei die erste Zeile mit der Angabe des Programmnamens optional ist.

```
program <Name>;

begin



end.
```

Das eigentliche Programm wird dann zwischen ***begin*** und ***end.*** Aufgeführt, wobei der Punkt hinter dem end wichtig ist! Nun ist das Programm im Grunde genommen in zwei Bereiche unterteilt.

- Der Deklarationsteil
- Der Anweisungsteil

Im *Deklarationsteil* werden die die im Anweisungsteil verwendeten Variablen deklariert. Jede einzelne Anweisung wird mit einem Semikolon abgeschlossen. Dieser Abschnitt beginnt mit dem reservierten Bezeichner ***var***. Das schaut zum Beispiel wie folgt aus:

```
program Test;
var i   : integer;
    s   : string[10];
    r   : real;
    a   : array[1..5] of integer;
begin
     writeln('Test-Programm');
end.
```

Es ist zu sehen, dass in den vier Zeilen unterschiedliche Variablendeklarationen erfolgen. In der ersten Zeile hinter var kommt es zur Deklaration der Variablen ***i*** des Datentyps ***integer***, also einem Ganzzahltyp. In der zweiten Zeile wird die Variable ***s*** des Datentyps ***string*** mit 10 Zeichen deklariert. Die dritte Zeile zeigt die Deklaration der Variablen ***r*** vom Datentyp ***real***, also einem Fließkommatyp. In der letzten Deklarationszeile erfolgt die Deklaration der Variablen ***a*** als Array vom Datentyp ***integer*** mit 5 Elementen.

Wir wollen nun zum Anweisungsteil wechseln, der hier recht kurzgehalten ist und lediglich die ***writeln***-Anweisung zur Anzeige eines Textes auf der Konsole beinhaltet.

```
program Test;
var i   : integer;
    s   : string[10];
    r   : real;
    a   : array[1..5] of integer;
begin
     writeln('Test-Programm');
end.
```

Wir wollen das Programm jetzt so erweitern, dass die Variablen auch mit konkreten Werten initialisiert und zur Anzeige gebracht werden. Das Ganze schaut wie folgt aus.

```
program Test;
var i   : integer;
    s   : string[10];
    r   : real;
    a   : array[1..5] of integer;
begin
     i := 17;
     s := 'Hallo';
     r := 3.14;
     a[1] := 39;
     writeln('i     = ', i);
     writeln('s     = ', s);
     writeln('r     = ', r);
     writeln('a[1]) = ', a[1]);
end.
```

Es ist zu sehen, dass die Initialisierung der Variablen über die Zeichenfolge ***:=*** ermöglicht wird und nicht nur alleine über das Gleichheitszeichen, wie man das vielleicht von *C* oder *MBASIC* gewohnt ist. Nach dem Verlassen des Editors erfolgt das Compiling über die Taste ***C***, die folgende Ausgaben zeigt.

```
Compiling
  15 lines

Code:   234 bytes (8256-8340)
Free: 31136 bytes (8341-FCE1)
Data:    36 bytes (FCE2-FD06)

>
```

Über die Taste ***R*** wird das Programm gestartet und liefert die folgende Anzeige.

```
Running
i     = 17
s     = Hallo
r     =  3.1400000000E+00
a[1]) = 39

>
```

Ich greife jetzt den zuvor angesprochenen Punkt noch einmal auf, wo es um die Generierung einer COM-Datei ging.

5.5 Eine COM-Datei generieren

Ich bleibe zur Verdeutlichung der Thematik beim zuletzt gezeigten Quellcode und wie sehen uns an, wie eine COM-Datei generiert werden kann. Wir haben gesehen, dass für das Starten des Compiling die Taste ***C*** gedrückt werden muss. Als Ausgabe waren folgende Meldungen zu sehen.

```
Compiling
  8 lines

Code:    54 bytes (81B5-81EB)
Free: 31477 bytes (81EC-FCE1)
Data:    36 bytes (FCE2-FD06)

>
```

Wir können jetzt über Turbo Pascal einen Blick ins Dateisystem werfen, um nachzusehen, ob sich dort eine ausführbare COM-Datei befindet, denn wir haben doch ein Compiling durchgeführt. Wir drücken aus dem Hauptmenü heraus die Taste ***D*** für *Dir* und geben die Maske *test.** ein.

```
Dir mask: test.*
TEST     PAS : TEST     BAK

Bytes Remaining On H: 7248k
>
```

Es ist zu sehen, dass sich dort lediglich zwei Dateien befinden. Zum einen natürlich die Quelldatei mit der Endung *.PAS* und eine Datei mit der Endung *.BAK*, was eine Backupdatei des Projektes ist. Also ist hier keine COM-Datei zu sehen. Was ist passiert oder besser gesagt, was ist hier warum nicht passiert? Zur Beantwortung dieser Frage müssen wir einen Blick in ein weiteres Menü werfen, das mit der Taste ***O*** aufgerufen wird und die *Optionen* anzeigt. Folgendes ist zu sehen.

```
compile -> Memory
           Com-file
           cHn-file

Find run-time error  Quit

>
```

Wir erkennen, dass sich das Compiling lediglich auf den Speicher bezieht, da der Pfeil auf ***Memory*** weist, was die Standardeinstellung ist. Eine kompilierte Datei liegt lediglich im Speicher vor und kann dann über die Taste ***R*** für *Run* ausgeführt werden. Um eine COM-Datei zu generieren, muss in diesem Menü die Taste ***C*** gedrückt werden, was zur folgenden Anzeige führt.

```
           Memory
compile -> Com-file
           cHn-file

Start address: 20E3 (min 20E3)
End   address: FA42 (max FD06)

Find run-time error  Quit

>
```

Ist diese Option aktiv, wird der Code in eine Datei geschrieben, die den gleichen Namen wie die Arbeitsdatei besitzt jedoch jetzt den Dateityp .COM vorweist. Diese Datei enthält sowohl den Programmcode, als auch die erforderliche Pascal-Laufzeitbibliothek, und kann durch Eingabe ihres Namens auf der Konsole gestartet werden. Die auf diese Weise kompilierten Programme können größer sein als Programme, die lediglich im Speicher kompiliert werden, da der Programmcode Zusatzinformationen beinhaltet, wie die schon angesprochenen Laufzeitbibliothek. Sie ist eine spezielle Programmbibliothek beziehungsweise eine Sammlung von Softwarefunktionen, die innerhalb eines Programms die eingebauten Funktionen zur Zeit der Ausführung des Programms - also zur Laufzeit - nutzt. Um das Optionen-Menü zu verlassen, muss die Taste ***Q*** für *Quit* gedrückt werden. Jetzt starten wir über das Drücken der Taste ***C***

für *Compile* erneut das Compiling. Es ist in der ersten Zeile zu sehen, dass dieser Vorgang die Datei ***TEST.COM*** auf dem aktuellen Laufwerk *H:* erzeugt hat.

```
Compiling  --> H:TEST.COM
  8 lines

Code:    54 bytes (20E3-2119)
Free: 55427 bytes (211A-F99D)
Data:   164 bytes (F99E-FA42)

>
```

Wir können über die Taste ***D*** unter Zuhilfenahme der Maske *test.** wieder einen Blick in das Dateisystem werfen und sehen, dass sich dort wirklich die Datei *test.com* befindet.

```
Dir mask: test.*
TEST     PAS : TEST     BAK : TEST     COM

Bytes Remaining On H: 7232k
>
```

Über die Taste ***X*** für *eXecute* kann diese Datei auch von Turbo Pascal heraus ausgeführt werden, ohne, dass dazu die Entwicklungsumgebung verlassen werden muss.

```
Program: test.com

Test-Programm


Loading H:TURBO.MSG
Loading H:TEST.PAS
>
```

Wir sollten jedoch einmal einen genaueren Blick auf die Ausgaben nach der Umstellung von *Memory* auf *COM-File* werfen. Diese Angaben lauteten für unser Projekt wie folgt, wobei ich die für uns wichtigen Stellen Rot umrandet habe.

```
            Memory
compile -> Com-file
            cHn-file

Start address: 20E3 (min 20E3)
End   address: FA42 (max FD06)

Find run-time error  Quit

>
```

Was bedeuten in diesem Zusammenhang *Start-* beziehungsweise *End*-Adresse?

Start-Address:

Die Startadresse gibt die Adresse in hexadezimaler Form des ersten Bytes des Codes an. Dies ist normalerweise die Endadresse der Pascal-Bibliothek plus eins, kann aber auch auf eine höhere Adresse geändert werden, wenn zum Beispiel Platz für absolute Variablen geschafft werden muss, die von einer Reihe von verketteten Programmen gemeinsam genutzt werden sollen.

End-Address:

Die Endadresse gibt die höchste Adresse in hexadezimaler Form an, die dem Programm zur Verfügung steht. Der Wert in Klammern gibt den höchsten Wert des *TPA* auf Ihrem Computer. Die Standardeinstellung ist 700 bis 1000 Bytes

weniger, um Platz für den *Loader* zu schaffen, der sich bei der Ausführung von Programmen direkt unterhalb des BDOS befindet.

Nähere Informationen dazu befinden sich im *Turbo Pascal Reference Manual für CP/M der Version 3 vom Dezember 1988* auf der Seite 145. Der Name der PDF-Datei lautet: ***TURBO_Pascal_Reference_Manual_CPM_Version_3_Dec88.pdf***

5.6 Die FOR-TO-DO-Schleife

Natürlich gibt es in Turbo Pascal auch Kontrollstrukturen in Form von Schleifen. Die einfachste wäre wieder die ***FOR-TO-DO***-Schleife, die der *FOR-NEXT*-Schleife aus *MBASIC* ähnelt. Sehen wir uns dazu ein Beispiel an. Es ist zu sehen, dass die *FOR-TO-DO*-Schleife - ganz wie der Anweisungsteil - durch eine Blockbildung über die Schlüsselwörter *begin* und *end* die gewünschten Anweisungen umschließt, wobei das end jedoch hier keinen Punkt am Ende vorweist. Über die geschweiften Klammern

{ Das ist eine Kommentarzeile }

können Kommentare in einer Zeile platziert werden, was auch über mehrere Zeilen erfolgen kann.

{

Das ist ein Kommentar,

der sich über zwei Zeilen erstreckt.

}

Eine weitere Möglichkeit ist die Verwendung der Syntax

(* Das ist ebenfalls eine Kommentarzeile *)

was eine ältere Variante darstellt.

```
     Line 9    Col 1   Insert    Indent  H:LOOP1.PAS
program Test;
var i   : integer;
begin
    for i := 1 to 10 do     { Startwert des Index = 1 }
    begin                   { Zaehlung von 1 bis 10   }
         writeln('i= ', i); { Ausgabe des Wertes      }
    end                     { Schleifenende           }
end.
```

Nach dem Compiling und dem Start schaut die Ausgabe wie folgt aus.

```
Running
i= 1
i= 2
i= 3
i= 4
i= 5
i= 6
i= 7
i= 8
i= 9
i= 10

>
```

5.7 Ein und Ausgabe

Nun haben wir schon einiges über Turbo Pascal gelernt, doch beginnen wir einmal mit den grundlegenden Strukturen der Ein- beziehungsweise Ausgabe von Daten. Die Ausgabe haben wir schon gesehen, was über das Schlüsselwort ***writeln*** möglich ist. Dieses *write-line* bewirkt ja ein Anzeigen mit gleichzeitigem Zeilenvorschub, was bedeutet, dass die nächste Ausgabe in der nächsten Zeile erfolgt. Um diesen Zeilenvorschub zu unterbinden, muss lediglich das *ln* am Ende von *write* entfallen. Sehen wir uns das genauer an.

```
     Line 10   Col 5   Insert    Indent  H:WRITE.PAS
program write;
begin
     write('Das ');    { Ohne Zeilenvorschub }
     write('steht ');  { Ohne Zeilenvorschub }
     write('alles ');  { Ohne Zeilenvorschub }
     write('in ');     { Ohne Zeilenvorschub }
     write('einer ');  { Ohne Zeilenvorschub }
     writeln('Zeile!');{ Mit Zeilenvorschub }
     writeln('Wobei das auch einfacher geht.');
end.
```

Die entsprechende Ausgabe schaut wie folgt aus.

```
Running
Das steht alles in einer Zeile!
Wobei das auch einfacher geht.

>
```

Kommen wir zur Eingabe von Informationen, die dann innerhalb des Programms verarbeitet werden. Sehen wir uns dazu das folgende Programm an.

```
     Line 10   Col 5   Insert    Indent  H:READ.PAS
program read;
var i, anzahl : integer;
begin
     write('Anzahl der Werte: '); { Ausgabe }
     readln(anzahl);              { Eingabe }
     for i := 1 to anzahl do
     begin
          write(i:4);             { Ausgabe }
     end
end.
```

Über das Schlüsselwort ***readln*** können Tastatureingaben in das Programm eingelesen und gespeichert werden. Die Speicherung der Eingabe erfolgt in der Variablen, die in runden Klammern angegeben wird. Das gezeigte Programm liest also die Eingabe und speichert sie in der Variablen *anzahl* ab, um darüber in der *FOR-TO-DO*-Schleife die Werte, die Werte der Laufvariablen *i* über *write* anzuzeigen. Doch was bedeutet die merkwürdige Schreibweise *write(i:4)* überhaupt? Es handelt sich um eine Ausgabe-Formatierungsanweisung, die besagt, dass der anzuzeigende Wert mit einer Breite des angegebenen Wertes hinter dem Doppelpunkt auf der Konsole erscheint. Hier ist das der Wert 4, was dazu genutzt werden kann, mehrere Wert in Form einer Tabelle akkurat formatiert anzeigen zu lassen. Doch sehen wir uns zunächst diese Ausgabe an.

```
Running
Anzahl der Werte: 10
   1   2   3   4   5   6   7   8   9  10
>
```

Es ist zu sehen, wie die einzelnen Werte mit einer festen Breite von 4 Zeichen ausgegeben werden, was aber erst richtig bei der Ausgabe von Zeilen und Spalten sichtbar wird. Bevor wir das umsetzen, müssen wir auf eine weitere

Kontrollstruktur eingehen, die in Form einer *IF*-Abfrage in Erscheinung tritt, was mich zum nächsten Punkt bringt.

5.8 Die IF-THEN-Abfrage

Im folgenden Programm möchte ich nach jedem 10. Wert einen Zeilenumbruch erwirken, damit die anzuzeigenden Werte in Form einer formatierten Tabelle zur Anzeige gebracht werden. Dazu ist natürlich eine Kontrollstruktur erforderlich, die diese Ausgabe überwacht. Es kommt die erwähnte ***IF-THEN***-Abfrage zum Einsatz. Zudem müssen wir uns eines weiteren Befehls bedienen, der sich ***Modulo***-Operator nennt. Das reservierte Wort ***mod*** steht wird dafür verwendet und liefert den Rest einer Division. Ist das Ergebnis einer Modulo-Division gleich Null, ist die angegebene Zahl durch diesen Wert teilbar, was wir uns für einen erforderlichen Zeilenumbruch für alle 10 Werte in der Anzeige zunutze machen.

```
     Line 11   Col 5   Insert    Indent  H:IF.PAS
program format;
var i, anzahl : integer;
begin
     write('Anzahl der Werte: '); { Ausgabe }
     readln(anzahl);
     for i := 1 to anzahl do
     begin
          write(i:4);              { Ausgabe }
          if i mod 10 = 0 then writeln;
     end
end.
```

Die folgende Ausgabe erscheint bei der Eingabe zur Anzeige für 100 Werte. Es ist hier wunderbar die formatierte Tabellenstruktur zu erkennen.

```
Running
Anzahl der Werte: 100
   1   2   3   4   5   6   7   8   9  10
  11  12  13  14  15  16  17  18  19  20
  21  22  23  24  25  26  27  28  29  30
  31  32  33  34  35  36  37  38  39  40
  41  42  43  44  45  46  47  48  49  50
  51  52  53  54  55  56  57  58  59  60
  61  62  63  64  65  66  67  68  69  70
  71  72  73  74  75  76  77  78  79  80
  81  82  83  84  85  86  87  88  89  90
  91  92  93  94  95  96  97  98  99 100

>
```

Natürlich gibt es auch bei der *IF*-Kontrollstruktur die Möglichkeit einer *IF-THEN-**ELSE***-Erweiterung. Wir wissen ja, dass die *IF-THEN-ELSE*-Anweisung eine angegebene Bedingung auswertet. Wenn die angegebene Bedingung wahr ist, kommt es zur Ausführung des *THEN*-Zweiges. Ansonsten werden die im *ELSE*-Zweig aufgeführten Anweisungen ausgeführt, wobei der *ELSE*-Zweig optional ist.

5.9 Eine Datei lesen

Im nachfolgenden Beispiel wollen wir den Inhalt einer Textdatei lesen. Dazu habe ich mit dem Text-Editor *TE* eine Datei mit dem Namen *info.txt* erstellt, die lediglich eine einzige Zeile Text enthält. Die mit einem Texteditor erstellten Textdateien sind Dateien, deren einzelne Elemente vom Typ *char* sind. Der Inhalt dieser Dateien vom Typ *text* kann mit einem gewöhnlichen Editor wie den genannten *TE* erstellt werden. Hier der Inhalt der genannten Textdatei.

```
te: TEXT.TXT                              --- | Lin:0001/0001/0512 Col:17/76 Len:16
    !...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...
  1|Das ist ein Test
```

Sehen wir uns das folgende Programm an. Ich habe in den betreffenden Zeilen Kommentare hinterlegt, die deren Funktion erläutern.

```
     Line 13   Col 5   Insert    Indent  H:TPTEXT.PAS
program readFile;
var     f : text; { text-Datentyp }
        b : char; { char-Datentyp }
begin
        assign(f,'text.txt'); { Zurodnung Variable zu Datei }
        reset(f); { Dateil lesend oeffnen }
        while not eof(f) do { Solange nicht EOF erreicht wird }
        begin
             read(f, b); { liest ein Zeichen aus der Datei }
             write(b); { Zeigt das Zeichen auf der Konsole }
        end;
        close(f); { Schliesst die Datei }
end.
```

Die Variable ***f*** vom Datentyp ***text*** nimmt den Inhalt der zu öffnenden Datei auf und die Variable ***b*** vom Datentyp ***char*** den Inhalt des einzelnen Zeichens. Nun muss im ersten Schritt die Dateivariable ***f*** mit der zu lesenden Textdatei verknüpft werden, was über den Befehl ***assign*** erfolgt. Nach diesem Schritt ist die Datei für das Lesen des Inhaltes jedoch noch nicht geöffnet worden. Das erfolgt im nächsten Schritt über den Befehl ***reset*** mit der Angabe der Variablen, die zuvor mit der Datei verknüpft wurde.

Damit die einzelnen Zeichen aus der Datei gelesen werden können, wird natürlich eine Schleife benötigt, die solange durchlaufen wird, wie das Ende der Datei noch nicht erreicht wurde. Mit ***EOF*** (*End Of File*) wird das Ende einer Datei signalisiert. Es handelt sich um einen booleschen Wert, der den Wahrheitswert ***true***, also *wahr*, zurückliefert, wenn das Ende der Datei erreicht wurde. Wir setzen hinter der *while*-Schleife den ***not***-Operator, der die Logik negiert, also umkehrt. Wurde das Ende also noch nicht erreicht, wird die Schleife erneut durchlaufen. Über die *read*-Funktion wird das aktuelle Zeichen aus dem Datenstrom gelesen und über die nachfolgende *write*-Funktion ohne Zeilenumbruch ausgegeben. Es darf niemals vergessen werden, eine zuvor geöffnete Datei über den ***close***-Befehl wieder zu schließen. Die Anzeige schaut wie folgt aus und entspricht dem gezeigten Inhalt des zuvor gezeigten Texteditors.

```
Running
Das ist ein Test

>
```

Sehen wir und jetzt eine zweite Variante für das Lesen einer Textdatei an, wo es darum geht, den Inhalt Zeilenweise zu lesen. Der Datentyp der variablen, der den Inhalt aufnimmt, wurde jetzt von ***char*** nach ***string*** geändert, wobei die Anzahl der einzelnen Zeichen in eckigen Klammern angegeben werden muss. Der Rest des Codes bleibt fast unverändert. Lediglich die *write*-Funktion wurde gegen die *writeln*-Variante getauscht, um darüber einen Zeilenvorschub nach jedem Anzeigen zu bewirken.

```
     Line 13   Col 5   Insert    Indent  H:TPTEXT2.PAS
program readFile;
var     f : text;
        b : string[80];
begin
     assign(f,'text2.txt');
     reset(f);
     while not eof(f) do
     begin
          readln(f, b);
          writeln(b);
     end;
     close(f);
end.
```

Der Inhalt der Konsole schaut nach dem Start des Programms wie folgt aus. Die drei Zeilen aus der Textdatei ***text2.txt*** werden angezeigt.

```
Running
Das ist die erste Zeile.
Das ist die zweite Zeile.
Das ist die dritte Zeile.

>
```

5.9.1 Die farbliche Darstellung von Text

Es wäre bestimmt eine gute Übung, die farbliche Darstellung von Text aus dem C-Kapitel hier noch einmal aufzugreifen. Damit nicht zu viel geblättert werden muss, hier noch einmal die Details. Es werden die folgenden Escape-Sequenzen verwendet.

- Black: \033[0;30m
- Red: \033[0;31m
- Green: \033[0;32m
- Yellow: \033[0;33m
- Blue: \033[0;34m
- Purple: \033[0;35m
- Cyan: \033[0;36m
- White: \033[0;37m

Für Fettschrift (BOLD) muss einfach die anfängliche 0 vor dem Semikolon durch eine 1 ersetzt werden. BOLD Yello wäre dann *\033[1;33m*. Zur Vorbereitung habe ich eine neue Textdatei mit Namen ***color.txt*** erstellt, die den folgenden Inhalt hat. Nachfolgend ist der erste Teil zu sehen, wo es einerseits um die und Typ- und Variablen-Deklaration und andererseits um die Programmierung der Prozedur ***setColor*** zu Farbeinstellung geht. Für die Einleitung der *Escape*-Sequenz wird hier der Wert ***#27*** verwendet.

```
     Line 14   Col 5   Insert    Indent  H:TPTEXT3.PAS
program readFile;
type    s80 = string[80];
var     f : text;
        b : s80;

procedure setColor(b: s80);
var v : string[6];
begin
     if b = 'rot'     then v := '[0;31m';
     if b = 'blau'    then v := '[0;34m';
     if b = 'gruen'   then v := '[0;32m';
     if b = 'schwarz' then v := '[0;30m';
     write(#27); write(v);
end;
```

Im zweiten Teil sehen wir das eigentliche Hauptprogramm und den hinzugefügten Aufruf der Prozedur *setColor* mit der Übergabe des gelesenen Textes aus der Textdatei.

```
begin
     assign(f,'color.txt');
     reset(f);
     while not eof(f) do
     begin
          readln(f, b);
          setColor(b);
          writeln(b);
     end;
     close(f);
end.
```

Nach der Ausführung des Programms schaut die Ausgabe auf der Konsole wie folgt aus.

```
Running
rot
blau
gruen
schwarz

>
```

5.9.2 Eine Binärdatei lesen

Wie eine Textdatei zu lesen ist, habe wir schon gesehen. Wie schaut es denn mit einer Binärdatei aus? Kann diese auf ähnliche Weise gelesen werden? Das ist möglich und wir wollen einmal sehen, wie die schon genutzte Binärdatei *sample.com* gelesen werden kann. Der Inhalt schaute ja wie folgt aus.

```
H3>dump sample.com

0000 3E 04 3C C9 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A
0010 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A
0020 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A
0030 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A
0040 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A
0050 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A
0060 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A
0070 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A

H3>
```

Die vier Bytes kamen dadurch zustande, dass das folgende Assemblerprogramm kompiliert wurde.

```
3E 04 ld a, 04

3C    inc a
```

```
C9    ret
```

Sehen wir uns nun das entsprechende Turbo Pascal Programm für das Leser dieser COM-Datei an. Wir brauchen hier nicht auf einen sogenannten Binärmodus umschalten, sondern können den bisherigen Quellcode nutzen und ein wenig anpassen. Die Funktion *int2hex* zur Umwandlung einer Dezimalzahl in einen hexadezimalen Wert haben wir schon gesehen und sie wird auch hier zum Einsatz kommen. Wir sollten unser Augenmerk auf die Zeile lenken, in der die write-Funktion zu sehen ist. Dort wird die ***ord***-Funktion genutzt, die das gelesene Zeichen vom Datentyp *char* in einen sorgenannten *ordinalen* Typ konvertiert. In der Programmierung ist ein ordinaler Typ ein Datentyp mit der Eigenschaft, dass seine Werte gezählt werden können.

```
     Line 21   Col 5   Insert    Indent  H:TPBIN.PAS
program readBin;
var  f   : text;
     b   : char;
type l2 = string[2];

function int2hex(dec : integer) : l2;
const hex : array[0..15] of char = '0123456789ABCDEF';
begin
     int2hex := hex[dec shr 4 and $0F] + hex[dec and $0F];
end;

begin
     assign(f, 'sample.com');
     reset(f);
     while not eof(f) do
     begin
          read(f, b);
          write(int2hex(ord(b)) + ' ');
     end;
     close(f);
end.
```

Nach dem Starten des Programms ist die folgende Ausgabe in der Konsole zu sehen. Die Werte entsprechen genau denen in der Binärdatei enthaltenen.

```
Running
3E 04 3C C9
>
```

5.10 Ein kleines Ratespiel

Lassen wir uns ein kleines Ratespiel programmieren, wo es darum geht, dass der Spieler zu Beginn einen Maximalwert eingibt, bis zu diesem Wert eine Zufallszahl generiert wird. Also wird zum Beispiel 100 eingegeben, so wird eine Zufallszahl im Bereich von 0 bis 100 generiert und es muss versucht werden, durch spätere Eingaben, diesen Wert mit möglichst wenigen Versuchen zu ermitteln. Die Ausgaben des Programms sind.

- Der eingegebene Wert ist zu klein
- Der eingegebene Wert ist zu gross
- Der Wert wurde erraten

Sehen wir uns das Programm an.

```
     Line 21   Col 5   Insert    Indent  H:ZUFALL.PAS
program zufall;
var max, z, e, i : integer;
begin
     write('Maximaler Zufallswert: ');
     readln(max);
     z := random(max + 1); { Zufallszahl generieren }
     i := 0; { Anzahl der Versuche }
     { writeln(z); }
     while e <> z do
     begin
          write('Dein Wert: ');
          readln(e);  { geratenen Wert eingeben }
          i := i + 1; { Anzahl Versuche hochzaehlen }
          if e < z then
             writeln('Wert ist zu klein!');
          if e > z then
            writeln('Wert ist zu gross!');
     end;
     writeln('Errarten!');
     writeln('Anzahl der Versuche: ', i);
end.
```

In der Variablen *max* wird über die *readln*-Funktion der größtmögliche Wert für Zufallszahlen über die Konsole eingegeben. In der darauffolgenden Zeile kommt es zum Aufruf der *random*-Funktion, die einen Zufallswert zwischen 0 und dem übergebenen Wert generiert, wobei dieser Wert exklusive ist und deswegen eine 1 addiert wird. Dieser Wert wird in der Variablen z gespeichert. Die Variable *i* wird bei jedem Versuch um den Wert 1 erhöht, um am Ende die Anzahl der benötigten Versuche auszugeben. Die Eingaben zu Ermittlung des Wertes werden in der Variablen *e* gespeichert und die *while-do*-Schleife solange durchlaufen, wie *e* und *z* ungleich sind. Über die *if-then*-Abfragen kommt es zu Bewertung der Eingabe und entsprechenden Anzeigen, ob der Wert zu klein oder zu groß ist. Nachfolgend ist ein kurzer Testlauf des Programms mit den Anzeigen zu sehen.

```
Running
Maximaler Zufallswert: 10
Dein Wert: 5
Wert ist zu klein!
Dein Wert: 8
Wert ist zu klein!
Dein Wert: 9
Errarten!
Anzahl der Versuche: 3

>
```

5.11 Der BDOS-Aufruf

Natürlich kann man unter Turbo Pascal auf einem CP/M-System auch BDOS-Aufrufe starten. Diese werden über die ***BDOS***-Funktion aufgerufen und sie besitzt die folgende Syntax.

```
BDOS(Func [,Param]);
```

Die Parameter *Func* und *Param* sind vom Datentyp *integer*, wobei die Angabe von *Param* optional ist. Func bezeichnet die Nummer der aufgerufenen Routine und wird ins *C*-Register geladen. Param bezeichnet einen Parameter, der in das Registerpaar *DE* geladen wird. BDOS wird - wie wir das schon wissen - auf Adresse 5 aufgerufen. Die Funktion BDOS wird wie die Prozedur aufgerufen und gibt einen integer Wert zurück, der dem Wert entspricht, der durch das BDOS in das *A*-Register zurückgegeben wird. Nachfolgend sind ein paar Beispiel zur Nutzung zu sehen, wobei auch die ***BDOSHL***-Funktion verwendet wird, um auf das Registerpaar *HL* zuzugreifen.

- ***BDOS***(c, de);
- hl:= ***BDOSHL***(c, de);
- a:= ***BDOS***(c, de)

5.11.1 Ein Zeichen auf der Konsole

Ich schlage vor, dass wir das bekannte Beispiel zur Ausgabe eines Buchstaben auf der Konsole einmal in Turbo Pascal umsetzen. Der erste Parameter der BDOS-Funktion wird hier mit *2* angegeben, der dann in das C-Register wandert. Der zweite Parameter in Form des ASCII-Codes für den Buchstaben *'a'* wird in das Registerpaar *DE* übertragen, wobei hier nur das LSB im *E*-Register zur Anzeige des ASCII-Zeichens herangezogen wird.

```
     Line 7    Col 5   Insert    Indent  H:BDOS.PAS
program bdos;
var ascii, c : integer;
begin
     ascii := 97;    { entspricht 'a' }
     c     := 2;     { Wert fuer das C-Register }
     bdos(c, ascii); { Aufruf der BDOS-Funktion }
end.
```

Als Ausgabe erscheint der Buchstabe *a* auf der Konsole.

```
Running
a
>
```

5.11.2 Eine Zeichenkette auf der Konsole ausgeben

Nun ist es natürlich auch unter Turbo Pascal möglich, mit dem geeigneten BDOS-Aufruf mehrere Zeichen auf der Konsole auszugeben. Sehen wir uns dazu das folgende Programm an. Hier geht es darum, dass bei einer Zeichenfolge über den BDOS-Aufruf die Startadresse des ersten Zeichens anzugeben ist. Es ist zu sehen, dass die Variable ***s*** eine Zeichenkettenvariable mit 50 Elementen ist. Dem späteren BDOS-Aufruf muss jetzt die Startadresse als zweites Argument mitgegeben werden. Wie wird das ermöglicht? Nun, es gibt dazu die ***addr***-Funktion, die die Startadresse einer Variablen zurückliefert. Die allgemeine Syntax lautet.

addr(<variable>);

Doch nun zum eigentlichen Programm. Das *C*-Register wird mit dem Wert ***9*** initialisiert und das Registerpaar *DE* mit der Startadresse der Variablen.

```
     Line 9    Col 5   Insert    Indent  H:BDOS2.PAS
program bdos2;
var s : string[50]; { String-Variable }
    c : integer;

begin
     s := 'Hallo, hier spricht CP/M!$';
     c := 9;          { Wert fuer das C-Register }
     bdos(c, addr(s)); { Aufruf der BDOS-Funktion }
end.
```

Nach dem Start des Programms schaut die Anzeige wie folgt aus. Es darf auch hier natürlich nicht das Terminierungszeichen ***$*** für die Kennzeichnung des Zeichenkettenendes vergessen werden.

```
Running
Hallo, hier spricht CP/M!
>
```

5.11.3 Auf einen Tastendruck warten

Es ist mit einem BDOS-Aufruf möglich, den Konsolen-Status abzufragen. Diese Funktion kann verwendet werden, um das Programm solange zu unterbrechen, bis eine Taste gedrückt wurde. Der Aufruf erfolgt mit der Übergabe des Wertes ***11*** an das *C*-Register. Der Inhalt des Registerpaares *DE* spiel hierbei keine Rolle. Wurde keine Taste gedrückt, liefert der BDOS-Funktionsaufruf den Wert ***0*** zurück. Wurde jedoch eine Taste gedrückt ändert sich der Wert auf ***255***. Sehen wir uns dazu das folgende Programm an. Über die *while*-Schleife wird ein kontinuierlicher Durchlauf erzwungen, solange die BDOS-Funktion den Wert *0* an die Variable ***a*** zurückliefert. Erst nach dem Drücken einer Taste ändert sich der Wert auf *255*, so dass die Bedingung für den Durchlauf der *while*-Schleife nicht mehr erfüllt wird. Die Schleife wird verlassen und das Programm mit einer entsprechenden Meldung verlassen.

```
     Line 11   Col 5   Insert    Indent  H:BDOS4.PAS
program bdos4;
var a : byte;
begin
     a := 0;
     writeln('Bitte eine Taste duecken!');
     while a = 0 do
     begin
          a := bdos(11, 0);
     end;
     writeln('Eine Taste wurde gedrueckt!');
end.
```

Solange keine Taste gedrückt wurde, ist lediglich die folgende Meldung zu sehen und das Programm läuft weiterhin, was am fehlenden Prompt-Zeichen zu erkennen ist.

```
>

Running
Bitte eine Taste duecken!
```

Jetzt wird eine Taste gedrückt und das Programm wird mit der entsprechenden Meldung verlassen. Das Prompt-Zeichen ist wieder sichtbar.

```
Running
Bitte eine Taste duecken!
Eine Taste wurde gedrueckt!

>
```

5.11.4 Die Anzeige der CP/M-Versionsnummer

Sehen wir uns ein weiteres kleines Programm an, das die CP/M-Versionsnummer zur Anzeige bringt. Wir nutzen dazu den BDOS-Aufruf mit dem Wert ***12***. Das Programm schaut wie folgt aus, wobei ich ein paar Dinge untergebracht habe, die neu sind. Da der BDOS-Aufruf in der Form eine Dezimalzahl als Ergebnis zurückliefert, wäre es sicherlich schön, könnten wir diesen in eine hexadezimale Zahl umwandeln. Dazu habe ich eine sogenannte *Funktion* geschrieben, die diese Aufgabe übernimmt. In Turbo Pascal wird zwischen *Prozeduren* und *Funktionen* unterschieden. Bei einer Prozedur handelt es sich

um ein Unterprogramm, das mit ihrem Namen aufgerufen wird und über eine optionale Parameterleiste verfügt. Nach deren Abarbeitung erfolgt geht es im Hauptprogramm weiter. Eine Funktion hingegen stellt eine Erweiterung einer Prozedur dar, die ebenfalls eine optionale Parameterliste hat, kann jedoch nach dessen Aufruf einen Rückgabewert an den Aufrufer liefern. Diese Möglichkeit machen wir uns zunutze, wenn es darum geht, einen Ganzzahlwert in eine hexadezimale Zahl zu wandeln. Diese Funktion wird im Quellcode ***int2hex*** genannt und verfügt über einen Parameter dec vom Datentyp *integer*, in dem der Übergabewert zur späteren Berechnung gespeichert wird. Ist die Funktion abgearbeitet worden, liefert sie einen Rückgabewert vom Datentyp ***12*** zurück, der benutzerdefiniert ist und über das Schlüsselwort ***type*** auf zwei Zeichen beschränkt ist. Sehen wir uns zunächst den Quellcode an und dann die Details zur Umwandlung der beiden genannten Datentypen.

```
     Line 14   Col 5   Insert    Indent  H:VER.PAS
program ver;
var v : integer;
type 12 = string[2];

function int2hex(dec : integer) : 12;
const hex : array[0..15] of char = '0123456789ABCDEF';
begin
  int2hex := hex[dec shr 4 and $0F] + hex[dec and $0F];
end;

begin
     v := bdos(12);
     writeln(int2hex(v));
end.
```

Da sich die hexadezimalen Werte der bekannten Ziffern beziehungsweise Zeichen *0* bis *9* und *A* bis *F* bedienen, wurde dazu ein Array von Datentyp *char* angelegt, dass diese Werte beinhaltet und über den Indes ausgerufen werden kann. Da aber ein Byte-Wert einen Wertebereich von 0 bis 255 besitzt, kann eine hexadezimale Zahl auch zweistellig sein. Wir müssen dazu den binären Wert über geeignete Manipulationen filtern, um an die gewünschten Informationen zu gelangen. Sehen wir uns das genauer an.

Nachfolgend ist eine aus 8 Bits bestehende Binärzahl zu sehen, die in einen Hexadezimalen Wert - kurz *HEX*-Wert - konvertiert wird.

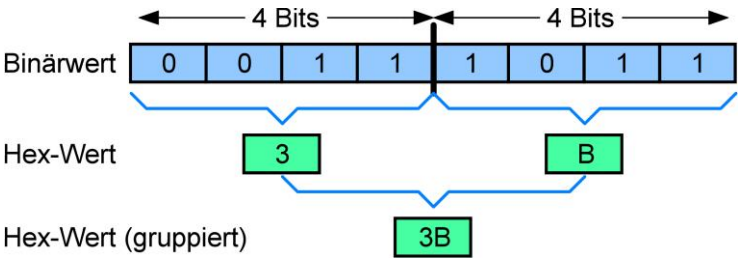
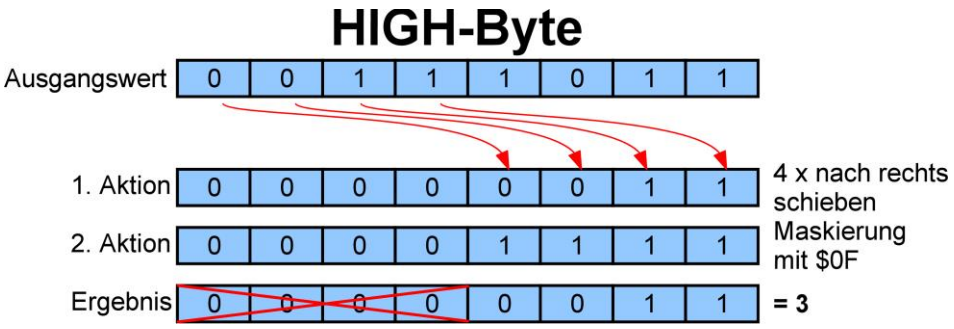


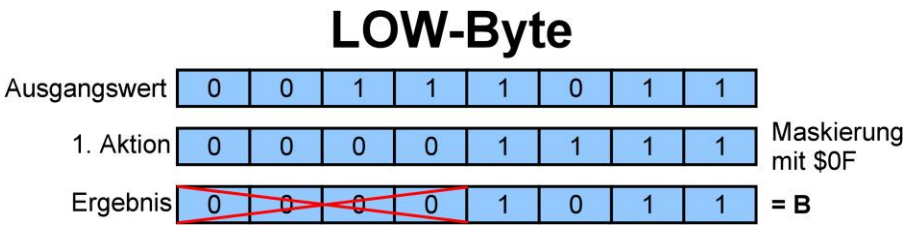
Abbildung 54 - Binär zu HEX

Es werden hier immer 4 Bits - also ein *Nibble* - einer Binärzahl durch ein hexadezimales Zeichen ersetzt. Betrachten wir den an die Funktion übergebenen Wert aus binärer Sicht und beschränken uns hier auf 8 Bits. Aus dem eigentlichen binären Wert *00111011* müssen wir die beiden Nibbles irgendwie extrahieren. Hier gibt es mehrere Ansätze, wobei ich den folgenden gewählt habe. Ich fange mit dem *HIGH*-Byte an. Zuerst wird über den ***shr***-Befehl (Shift-Right) die Bitkombination um die angegebenen Stellen nach rechts geschoben wobei auf der linken Seite pro Shift der binäre Wert 0 eingeschoben wird.

Dann erfolgt eine binäre *UND*-Verknüpfung über den ***and***-Operator mit dem Wert *$0F*, was bedeutet, dass nur bei den untersten 4 Bits die Werte berücksichtigt werden, denn die obersten 4 Bits werden mit 0-Werten versehen.



Beim *LOW*-Byte ist dagegen nur eine Aktion erforderlich, denn es müssen lediglich die die untersten 4 Bits berücksichtigt werden, was wieder über den *and*-Operator mit dem Wert *$0F* erfolgt.



Nach dem Compiling und Start des Programms schaut die Ausgabe wie folgt aus. Es ist zu sehen, dass der Wert *22* ausgegeben wird, was für die CP/M-Version 2.2 steht.

```
Compiling
  15 lines

Code:   178 bytes (8238-82EA)
Free: 31236 bytes (82EB-FCEF)
Data:    22 bytes (FCF0-FD06)

Running
22

>
```

5.11.5 Die Pinansteuerung am Board

Wir haben im Kapitel über die Maschinensprache schon gesehen, wie es möglich ist, die digitalen Pins auf dem Board zu beeinflussen. Da wir das über geeignete BDOS-Aufrufe gemacht haben, ist das natürlich auch bei Turbo Pascal möglich, was wir uns jetzt ansehen werden. Auf der folgenden Abbildung ist das Zusammenspiel der Register bei den BDOS-Aufrufen für *pinMode* und *digitalWrite* noch einmal zu sehen.

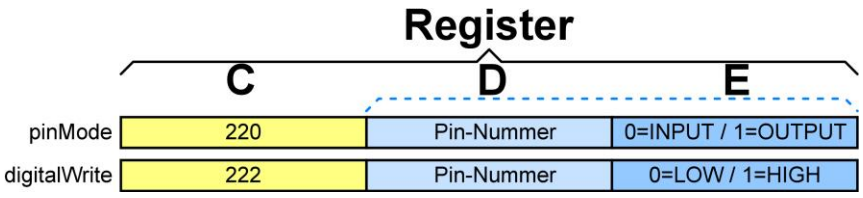


Abbildung 55 - Die Register für pinMode und digitalWrite

Sehen wir uns zuerst das komplette Programm in Turbo Pascal an, was wie folgt ausschaut.

```
     Line 14   Col 5   Insert   Indent  H:LED.PAS
program led;
const
     input  = 0;
     output = 1;
var
     pin, value : integer;
begin
     write('Pin-Number: ');
     readln(pin);
     write('Value: ');
     readln(value);
     bdos(220, (pin shl 8) + output);
     bdos(222, (pin shl 8) + value);
end.
```

Die Pin-Nummer befindet sich also in der Variablen *pin* und der Wert zur Ansteuerung in der Variablen *value*. Damit das Ganze funktioniert, muss die Pin-Nummer in das *D*-Register und der Wert in das *E*-Register übertragen werden. Da es sich beim Datentyp *integer* um 16-Bit Werte handelt, müssen die Bits zur Übertragung etwas verschoben werden. Wir erinnern uns, dass der BDOS-Aufruf sich wie folgt gestaltet.

```
BDOS(Func [,Param]);
```

Die Parameter *Func* und *Param* sind beide vom Datentyp *integer*, wobei die Angabe von *Param* jetzt entscheidend ist, denn das Argument, das übergeben wird, landet im *DE*-Registerpaar. Damit die Pin-Nummer im D-Register gespeichert wird, muss der eingegebene Wert um 8 Bitpositionen nach links geschoben werden, was über den ***shl***-Befehl (Shift-Left) mit der Angabe der Positionen erfolgt. Es funktioniert in ähnlicher Weise, wie wir das schon beim *shr*-Befehl für das Rechtsschieben gesehen haben.

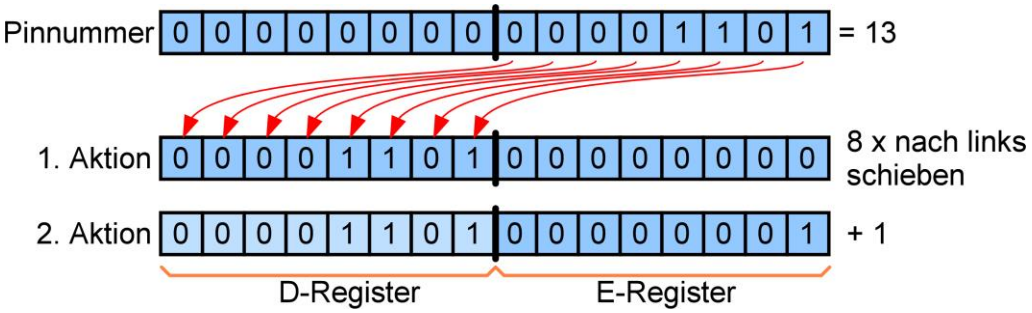


Abbildung 56 - Die Verschiebung der Bits und die Addition von 1

Es ist zu sehen, dass das DE-Registerpaar derart vorbereitet wurde, um den entsprechenden Pin über *pinMode* zu konfigurieren und über *digitalWrite* zu manipulieren, was in beiden Fällen die gleiche Aktion erfordert. Lediglich

die spätere Addition entscheidet über *INPUT* oder *OUTPUT* beziehungsweise *LOW* oder *HIGH*.

5.11.6 Die Ansteuerung wie beim Arduino

Nun kommen wir zu einer Erweiterung der Programmierung, die es uns ermöglicht, das Ganze so zu gestalten, wie es aus dem Arduino-Umfeld in einem *Sketch* ausschaut. Wir erinnern uns, dass der Quellcode beim Arduino *Sketch* genannt wird. Doch bevor es losgeht, müssen wir uns etwas tiefer mit Turbo Pascal befassen. Wer sich schon etwas mit der Programmiersprache C auskennt, dem ist sicherlich bekannt, dass es dort die Möglichkeit der Einbindung von weiteren Dateien über die Präprozessor-Direktive *#include* besteht. Include-Dateien in C/C++ enthalten typischerweise Informationen, die im Rahmen der Kompilierung verschiedener Projekte mehrfach benötigt werden. Die Verwendung von Include-Dateien führt in der Regel zu besser strukturierten und wartbaren Programmen. Es liegt in der Natur der Sache, dass auf diese Weise Informationen beziehungsweise Programmcode nur ein einziges Mal abgelegt und Änderungen einmalig an zentraler Stelle vorgenommen werden müssen.

Dieses Konstrukt ist auch bei Turbo Pascal möglich und wird über die Direktive ***{$I filename}*** eingefügt. Darüber wird der Compiler angewiesen, den Code aus der genannten Datei zu lesen. Die dort vorhandenen Anweisungen werden so eingefügt, als ob sie in der aktuellen Datei vorhanden werden. Doch sehen wir uns zunächst die Include-Datei an, die in der Regel die Dateiendung ***.inc*** besitzt. Ich starte der Einfachheit halber mit zwei Prozeduren, die dort programmiert wurden. Ich bin auf den Unterschied zwischen Prozeduren und Funktionen kurz eingegangen. Hier die allgemeine Syntax einer sogenannten ***Procedure*** in Turbo Pascal.

```
procedure <Name> (Parameter)

begin

   Anweisung 1;

   Anweisung 1;

   Anweisung 1;

   ...

end;
```

Die Definition einer Prozedur muss oberhalb des Anweisungsblocks stehen, aus dem sie aufgerufen wird. Das bedeutet beim eigentlichen Programm noch vor dem ***begin***. Wir nutzen die genannte Möglichkeit der Auslagerung des Codes in eine externe Datei, die wie folgt ausschaut. In der ersten Zeile ist lediglich der Kommentar zu sehen, dass es sich um eine Include-Datei handelt. Über die Konstanten werden die Werte für *input*, *output*, *low* und *high* definiert. Die beiden Prozeduren besitzen den gleichen Namen, die auch in der Arduino-Entwicklungsumgebung genutzt werden. Die Übergabeparameter sind vom Datentyp *byte* und reichen die Werte an die Prozedur zu Bearbeitung weiter. Hier sind die gleichen BDOS-Aufrufe zu sehen, die vorher schon zum Einsatz kamen.

```
     Line 17   Col 32  Insert    Indent  H:ARDUINO.INC
{ arduino.inc }

const
     INPUT  = 0;
     OUTPUT = 1;
     LOW    = 0;
     HIGH   = 1;

procedure pinMode(pin, mode: byte);
begin
     bdos(220, (pin shl 8) + mode);
end; { procedure pinMode }

procedure digitalWrite(pin, value: byte);
begin
     bdos(222, (pin shl 8) + value);
end; { procedure digitalWrite }
```

Nach der Speicherung dieser Datei können wir uns der eigentlichen Datei widmen, die als Hauptprogramm in Erscheinung tritt. Der Inhalt schaut wie folgt aus. Zu Beginn muss natürlich die aus der Include-Datei eingebunden werden, was über die Direktive ***{$I arduino.inc}*** erfolgt. Mit der Konstanten *pin* wird die zu nutzende Pin-Nummer festgelegt und ich habe wieder *Pin 13* genommen, da sich diese direkt als LED auf dem Board befindet und es keinen Anschluss einer externen LED bedarf.

Innerhalb von ***begin end*** wird einmalig die *pinMode*-Prozedur zu Konfiguration mit den übergebenen Werten aufgerufen, wobei *OUTPUT* die Konstante aus der Include-Datei ist. Als Schleife wird hier das ***repeat-until***-Konstrukt verwendet. Diese Schleife wird solange durchlaufen, bis die Bedingung hinter *until* zutrifft. Das genannte ***keypressed*** durchsucht den Tastaturpuffer und prüft, ob eine Taste gedrückt wurde. Ist dies der Fall, wird der Wert *true* (wahr) zurückgegeben. Falls nicht, wird *false* (falsch) zurückgegeben. Die Sondertasten *Shift*, *Alt*, *Ctrl* werden über keypressed nicht berücksichtigt. Die digitalWrite-Prozedur wird wie aus der Arduino-Entwicklungsumgebung gewohnt, benutzt. Eine Pause erfolgt über die Turbo Pascal eigene *delay*-Prozedur mit der Angabe in Millisekunden, was jedoch zeitlich nicht so funktioniert, wie das gewünscht ist, da die Ausführungsgeschwindigkeit beziehungsweise Taktrate von *RunCPM* erhöht ist. Ich würde den Wert auf 10000 erhöhen oder ein bisschen experimentieren.

```
     Line 12   Col 5   Insert    Indent  H:PIN2.PAS
{$I arduino.inc}
const
     pin = 13; { Pin-Nummer }
begin
     pinMode(pin, OUTPUT);
     repeat
          digitalWrite(pin, HIGH); { LED an                    }
          delay(1000);             { Pause                     }
          digitalWrite(pin, LOW);  { LED aus                   }
          delay(1000);             { Pause                     }
     until keypressed;             { Fortfahren bis Tastendruck }
end.
```

Ich schlage vor, die Include-Datei selbst einmal um die *digitalRead*-Funktion zu erweitern! Ist hier eine weitere Prozedur erforderlich oder funktioniert das nicht mit diesem Konstrukt?

5.11.7 Einen analogen Eingang abfragen

Ich hatte es im Kapitel über die Maschinensprache schon geschrieben, dass die Abfrage eines analogen Eingangs hier im Kapitel über die Programmiersprache Turbo Pascal zu finden ist. Nun ist es soweit. Um diese

Möglichkeit nutzen zu können, müssen wir unsere *Include*-Datei um eine Funktion erweitern. Wir erinnern uns, dass hier eine Funktion und keine Prozedur zu verwenden ist, denn eine Prozedur ist nicht in der Lage, einen Rückgabewert an den Aufrufer zu liefern! Hier die allgemeine Syntax einer sogenannten ***Function*** in Turbo Pascal.

```
function <Name> (Parameter) : Rückgabetyp

begin

   Anweisung 1;

   Anweisung 1;

   Anweisung 1;

   <Name> := Wert;

end;
```

Damit die Funktion nach deren Aufruf einen Wert zurückliefert, muss dieser Wert dem Namen der Funktion zugewiesen werden. Hier die Funktion *analogRead*, um die die Include-Datei erweitert werden muss. Da der Rückgabewert sich im Bereich von 0 bis 1023 befindet, muss die *BDOSHL*-Funktion mit dem Wert 223 aufgerufen werden!

```
function analogRead(pin: byte) : integer;
begin
     analogRead := bdosHL(223, (pin shl 8));
end;
```

Sehen wir uns jetzt das Hauptprogramm an. Es wird hier der analoge Pin ***A0*** verwendet, was dem Pin 14 entspricht. Wir sehen uns gleich dazu das Schaltbild an, so dass dies klar sein wird. Der ermittelte Messwert wird in der Variablen *value* gespeichert, was über den Aufruf der *analogRead*-Funktion erfolgt. Der nachfolgende Aufruf der *writeln*-Funktion zeigt diesen Wert auf der Konsole an.

```
      Line 12   Col 5   Insert    Indent  H:ANALOG.PAS
{$I arduino.inc}
const
     pin = 0; { Analoge Pin-Nummer }
var
     value : integer; { Messwert }
begin
     repeat
     value :=analogRead(pin); { analogen Wert ermitteln           }
     writeln(value);          { Messwert anzeigen                 }
     delay(2000);             { Kurze Pause                       }
     until keypressed;        { Fortfahren bis Tastendruck erfolgt }
end.
```

Kommen wir nun zum Schaltplan, der mit lediglich einem Potentiometer von 10KΩ versehen ist.

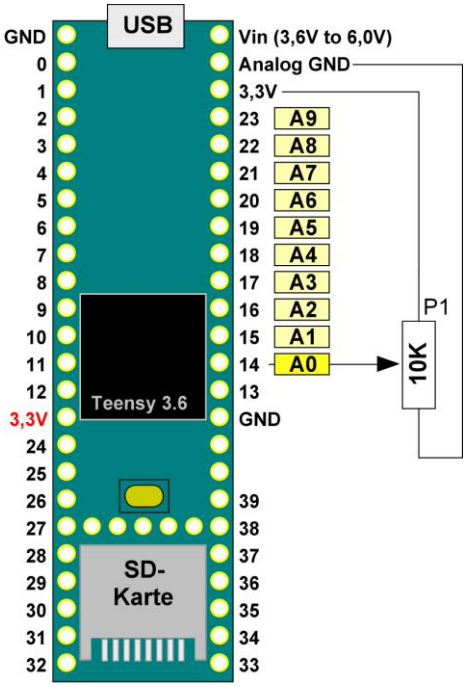


Abbildung 57 - Schaltplan zur Abfrage des analogen Eingangs A0 an Pin 14

Die Ausgabe gestaltet sich wie folgt, wobei ich natürlich an meinem Potentiometer hin und her gedreht habe.

```
Running
200
156
83
21
17
45
152
165
25
3
47
198
248
125
```

Der Schaltungsaufbau ist schnell hergestellt.

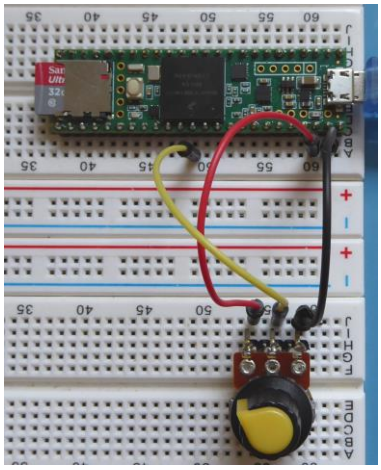
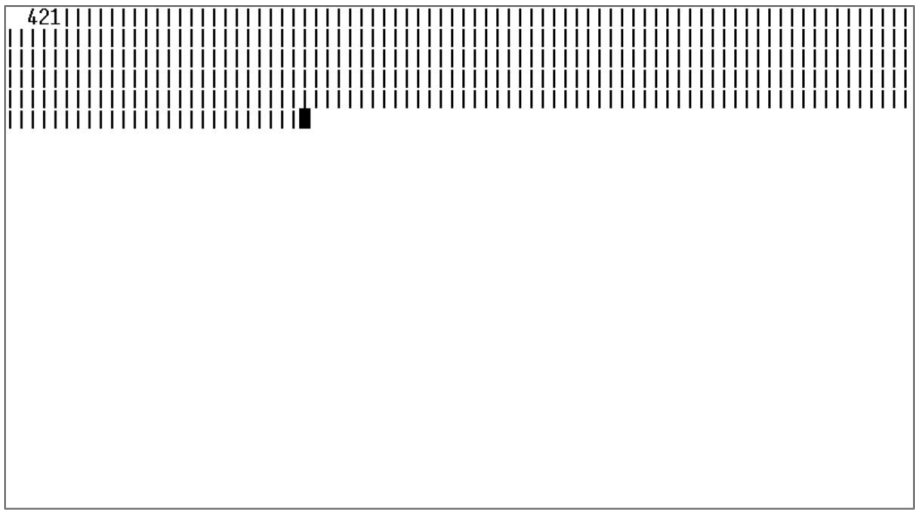


Abbildung 58 - Der Schaltungsaufbau zur Abfrage eines Potentiometers

Ich möchte diesen Code etwas erweitern und das Ergebnis der analogen Messung nicht nur als einen Messwert in der Konsole anzeigen, sondern etwas grafisch aufpeppen. Da es in dieser Form einer Konsole jedoch nicht möglich ist, Grafiken darzustellen, müssen wir uns der zur Verfügung stehenden Zeichen bemühen. Also am besten einen senkrechten Strick, der auch als Pipe-Symbol (|) bekannt ist. Je größer der Messwert ist, desto mehr Zeichen sollen angezeigt werden. Was aber mit den momentanen Kenntnissen nicht so einfach ist, denn jede neue Ausgabe über *write* beziehungsweise *writeln* bewirkt die Anzeige in einer neuen Zeile. Jetzt gibt es mehrere Möglichkeiten, um unser Vorhaben zu realisieren. Es gibt in Turbo Pascal zum Beispiel den Befehl ***clrscr*** (Clear-Screen), also den Bildschirm löschen. Eine nachfolgende Ausgabe erfolgt startend immer in der linken oberen Ecke. Sehen wir uns den folgenden Code an. Es wurde eine Prozedur mit dem Namen ***printValue***, in der bei jedem Aufruf, der ja kontinuierlich erfolgt, zuerst immer der Bildschirm mit genanntem Befehl gelöscht wird. Im Anschluss erfolgt die numerische Anzeige des Messwertes und über die nachfolgende *FOR-TO-DO*-Schleife die Anzeige der senkrechten Striche, deren Anzahl von dem übergebenden Messwert abhängt.

```
     Line 1    Col 17  Insert    Indent  H:ANALOG2.PAS
{$I arduino.inc}
const
     pin = 0; { Analoge Pin-Nummer }
var
     i, value : integer; { Messwert }

procedure printValue(v : integer);
begin
     clrscr;
     write(v : 5); { Wert anzeigen }
     for i := 1 to v do
     begin
          write('|'); { Balken anzeigen }
     end;
end;

begin
     repeat
     value :=analogRead(pin); { analogen Wert ermitteln            }
     printValue(value);       { Balken anzeigen                    }
     delay(2000);             { Kurze Pause                        }
     until keypressed;        { Fortfahren bis Tastendruck erfolgt }
end.
```

Sehen wir uns dazu die Ausgabe an. In der linken oberen Ecke ist der Messwert zu sehen und in Abhängigkeit dieses Wertes wird die Anzahl der Striche beeinflusst. Der Abbruch des Programms erfolgt bei einem Tastendruck.



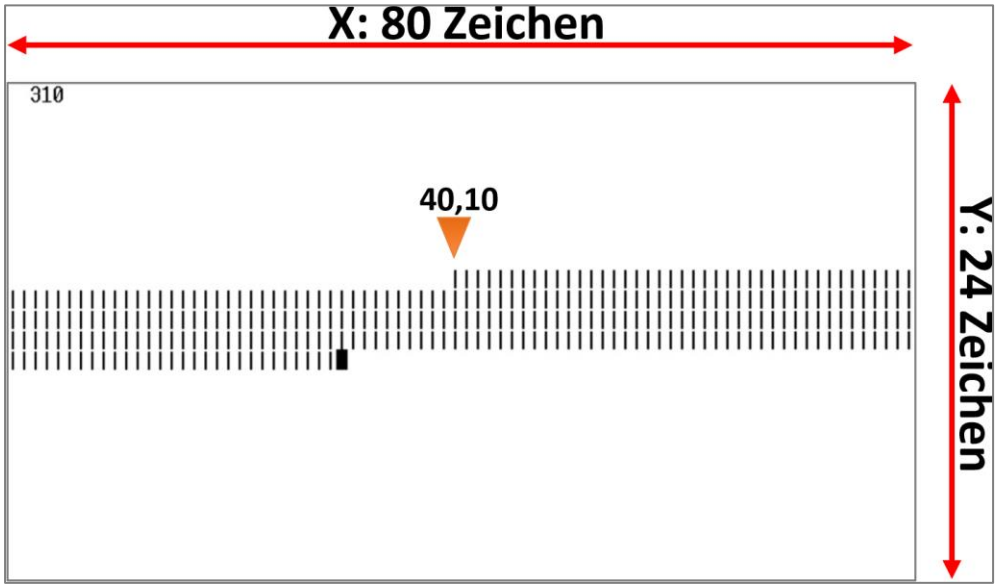
Eine weitere Möglichkeit der absoluten Positionierung des Cursors besteht in der Nutzung des ***gotoxy***-Befehls. Die allgemeine Syntax lautet.

```
gotoxy(x, y);
```

Der Parameter ***X*** steht für die Spaltenposition und ***Y*** für die Zeilenposition. Nachfolgend ist der Code zu sehen, der den Messwert in der linken oberen Ecke anzeigt und die Balken startend an der Position *40,10* positioniert.

```
     Line 1    Col 17  Insert    Indent  H:ANALOG2.PAS
{$I arduino.inc}
const
     pin = 0; { Analoge Pin-Nummer }
var
     i, value : integer; { Messwert }

procedure printValue(v : integer);
begin
     clrscr;
     gotoxy(0,0);
     write(v : 5); { Wert anzeigen }
     gotoxy(40,10);
     for i := 1 to v do
     begin
          write('|'); { Balken anzeigen }
     end;
end;

begin
     repeat
     value :=analogRead(pin); { analogen Wert ermitteln           }
     printValue(value);       { Balken anzeigen                   }
     delay(2000);             { Kurze Pause                       }
     until keypressed;        { Fortfahren bis Tastendruck erfolgt }
end.
```

Die Anzeige gestaltet sich wie folgt.



5.11.8 Einen analogen Wert schreiben

Kommen wir zu einem speziellen Thema, was sich mit *PWM* befasst. Doch beginnen wir ganz von vorne. Die meisten Mikrocontroller besitzen – wenn es um die Ansteuerung von Verbrauchern geht – von Hause aus lediglich digitale Ausgänge. Damit kann man, wenn es um Themen der Digitaltechnik geht, einiges bewirken, um die Verbraucher an- bzw. auszuschalten, also zum Beispiel Leuchtdioden, Lampen, Relais oder auch Motoren, um nur einige wenige zu nennen. Wenn wir aber zum Beispiel eine LED oder einen Motor nicht ausschließlich in ihren Grenzzuständen - an oder aus - betreiben möchten, wird es mit digitalen Ausgängen schwierig, das Vorhaben umzusetzen. Wir können damit entweder 0V oder die Betriebsspannung von 3,3V an den Verbraucher schicken. Es gibt keinerlei Abstufungen oder Zwischenwerte, damit die LED nur halb so hell leuchtet oder sich der Motor nur halb so schnell dreht. Wir benötigen also einen analogen Ausgang, der z.B. den Verbraucher mit lediglich 1,5V versorgt. Das ist ungefähr die Hälfte des Wertes der Betriebsspannung und somit steht auch weniger Energie zum Betreiben eines Verbrauchers zur Verfügung. Doch es hilft alles nichts, ein derartiger analoger Ausgang steht uns nicht zur Verfügung. Und dennoch gibt es die Möglichkeit über einen Trick, einen Verbraucher mit regelbarer Energie zu versorgen. Mir fällt ein Beispiel aus der Praxis ein, das zum Verständnis dessen, worauf ich hinaus möchte, sicher helfen wird. Zur Behandlung von Hautwucherungen werden teilweise Laser eingesetzt, die das Gewebe auf der Haut - laienhaft ausgedrückt - wegbrennen. Es aber keinesfalls wünschenswert, den Laserstrahl mit voller Leistung auf die betroffene Wucherung zu lenken, weil darunterliegende oder benachbarte nicht erkrankte Hautregionen in Mitleidenschaft gezogen würden. Um das zu verhindern, lässt man den Laserstrahl in verschiedenen Frequenzen pulsieren. Auf diese Weise wird der Energietransfer auf das Zielgebiet reguliert und kontrolliert. In vergleichbarer Weise werden wir jetzt eine LED ansteuern. Da sind wir schon beim Thema *PWM*.

Was bedeutet PWM?

PWM ist die Abkürzung für *Pulse Width Modulation*, zu Deutsch: Puls-Weiten-Modulation und beschreibt das Verhältnis zwischen der Einschaltzeit und Periodendauer eines Rechtecksignals bei einer konstanten Grundfrequenz, wobei das Verhältnis zwischen der Einschaltzeit und der Periodendauer als das *Tastverhältnis* bezeichnet wird. Darüber kann die mittlere Spannung gesteuert werden.

Wie aber funktioniert PWM im Detail? Das gerade erwähnte Tastverhältnis - auch *Duty-Cycle* genannt - ist das Verhältnis von Impulsdauer zur Periodendauer. Sehen wir uns das anhand des folgenden Diagramms genauer an.

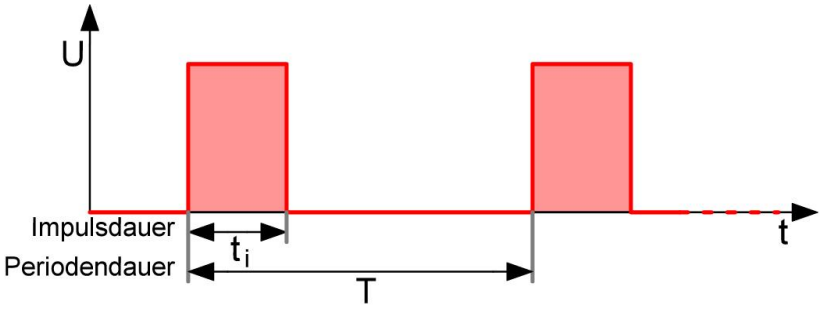


Abbildung 59 - Impuls- und Periodendauer

Die Formel zur Berechnung des Tastverhältnisses lautet wie folgt.

$$Tastverhältnis = \frac{t_i}{T} \cdot 100\ [\%]$$

Wenn zum Beispiel die Impulszeit gleich der Pausenzeit ist, schaut das Signal wie folgt aus.

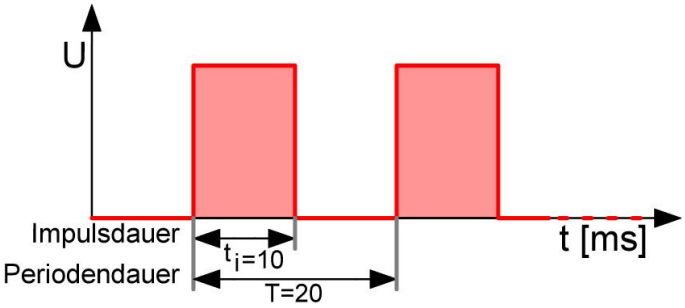


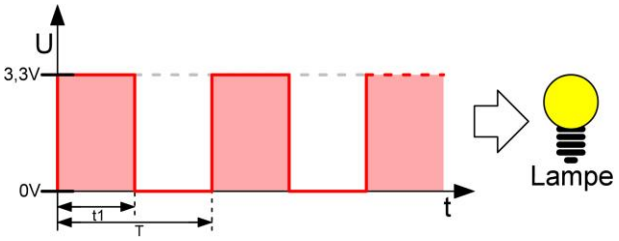
Abbildung 60 - Impuls- und Pausenzeit sind gleich

Mit den angegebenen Werten werfen wir einen Blick auf die Formel.

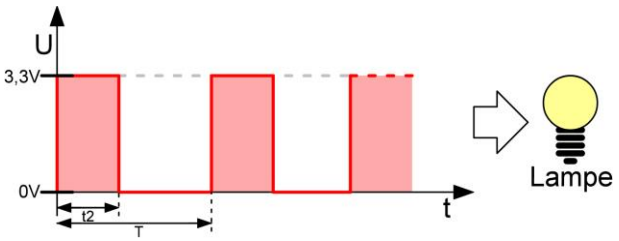
$$Tastverhältnis = \frac{10ms}{20ms} \cdot 100\ = 50\%$$

Das macht auch Sinn, denn die Impulsdauer ist im Verhältnis zur Periodendauer nur halb so breit und das entspricht eben 50%. Angenommen, wir steuern eine LED über einen Rechtecksignal in einer bestimmten Frequenz an, wobei die Frequenz jedoch so hoch ist, dass die Trägheit unseres Auges diese stetigen An-Aus-Phasen nicht mehr wahrnimmt und ein durchgehendes Leuchten zu erkennen ist. Sehen wir uns dazu die folgenden Abbildungen an, in der ein Rechtecksignal zu sehen ist, die unterschiedliche Impulsdauern (*t1 bis t3)* und eine konstante Periodendauer *T* vorweisen. Je kürzer die Impulsdauer ist, desto geringer ist die Leuchtstärke der Lampe.

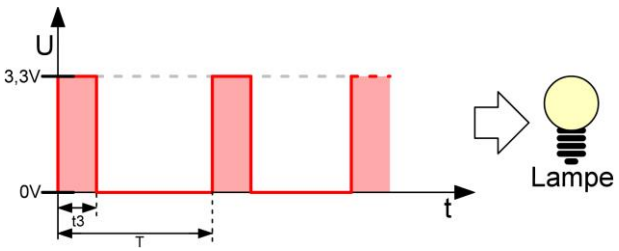
Puls-Weiten-Modulation mit 50%



Puls-Weiten-Modulation mit 35%



Puls-Weiten-Modulation mit 25%



Wie aber kann man sich das mit der Energie vorstellen, die dem angeschlossenen Verbraucher zur Verfügung gestellt wird? Indem die Stromversorgung eines Verbrauchers mit einer hohen Frequenz ein- und ausgeschaltet wird, wird der durchschnittliche Strom, der durch das Gerät fließt, reguliert. Durch schnelles Hin- und Herschalten zwischen den beiden Extremwerten 0V und Versorgungsspannung ergibt sich der Mittelwert als Funktion der Zeit, die in jedem der beiden Zustände verbracht wird. Man kann sagen: *Je länger An, desto mehr Leistung steht in einem zeitlichen Bereich zur Verfügung.*

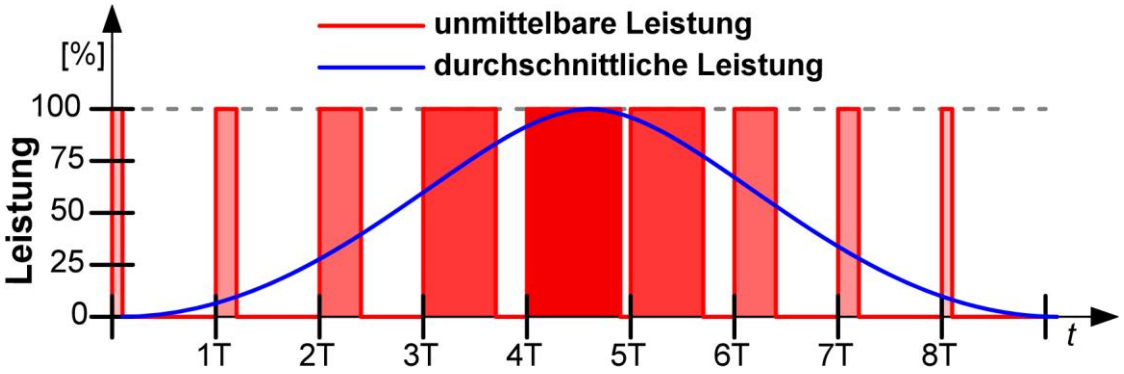


Abbildung 61 - An- und Ausschalten der Versorgungsspannung

Zur Helligkeitssteuerung einer angeschlossenen LED wird also ein digitaler Ausgang genutzt. Jedoch können zur Umsetzung nicht alle digitalen Ausgänge verwendet werden, da nicht alle eine PWM-Funktionalität vorweisen. Man sollte sich also beim verwendeten Mikrocontroller-Board auf der Herstellerseite kundig machen, um zu sehen, welche Pins dafür infrage kommen. Beim *Teensy-Board 3.6* habe ich zu diesem Zweck den Pin 23 verwendet. Da wir im vorangegangenen Kapitel die Abfrage eines analogen Eingangs gesehen haben, können wir diesen Messwert zur Ansteuerung eines PWM-Ausgangs nutzen. Es ist jedoch dabei zu beachten, dass ein analoger Eingang Messwerte im Bereich von 0 bis 1023 zur Verfügung stellt, die PWM-Ansteuerung jedoch Werte im Bereich von 0 bis 255 erwartet. Bei größeren Werten wird einfach wieder bei 0 begonnen. Im Arduino-Umfeld gibt es eine *map*-Funktion, die Eingangswerte (z.B. 0 bis 1023) auf Ausgangswerte (z.B. 0 bis 255) mappen kann.

Bevor es jedoch losgehen kann, müssen wir wieder die Include-Datei um eine Prozedur erweitern, damit die PWM-Funktionalität genutzt werden kann. Die Prozedur lautet *analogWrite*, ruft BDOS mit dem Wert 224 auf und erwartet eine Pin-Nummer und den PWM-Wert.

```
procedure analogWrite(pin, value: byte);
begin
     bdos(224, (pin shl 8) + value);
end; { procedure analogWrite }
```

Sehen wir uns das Hauptprogramm an. Es läuft solange, bis eine Taste gedrückt wird.

```
      Line 14   Col 5   Insert    Indent  H:ANALOGW.PAS
{$I arduino.inc}
const
     pinA = 0;  { Analoge Pin-Nummer }
     pinD = 23; { Digitale Pin-Nummer }
var
     value : integer; { Messwert }
begin
     repeat
     value :=analogRead(pinA); { analogen Wert ermitteln           }
     writeln(value);           { Messwert anzeigen                 }
     delay(500);               { Kurze Pause                       }
     analogWrite(pinD, value); { analogen Wert schreiben           }
     until keypressed;         { Fortfahren bis Tastendruck erfolgt }
end.
```

Kommen wir nun zum Schaltplan, der mit dem schon verwendeten Potentiometer von 10KΩ versehen ist und um eine LED mit Vorwiderstand von 330Ω an Pin 23 erweitert wurde.

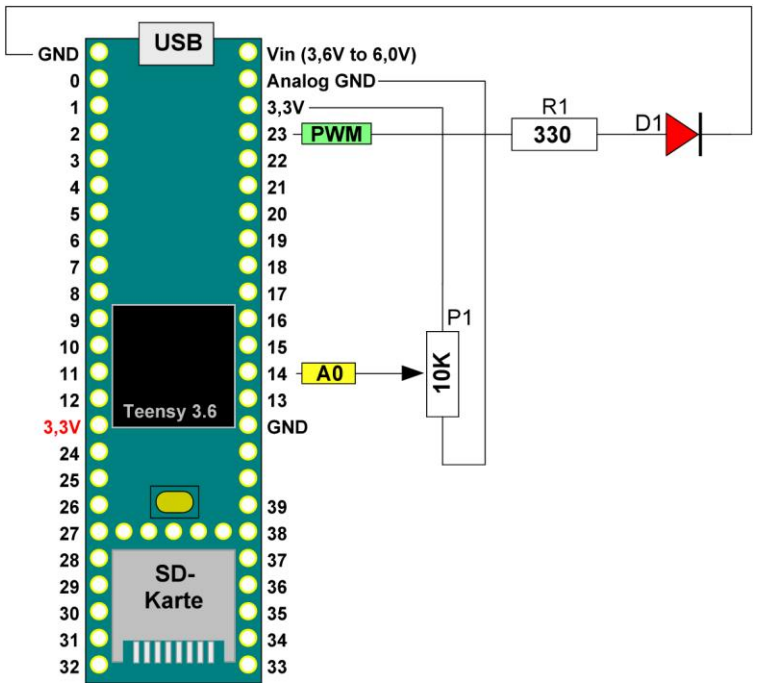


Abbildung 62 - Schaltplan zur Abfrage PWM-Ansteuerung an Pin 23

Bevor ich nun mir der Programmierung in diesem Buch weitermache, werde ich das an dieser ein kleines Kapitel über sehr bekannte Programme unter CP/M einfügen.

6 Die Programmiersprache FORTRAN

Die Programmiersprache ***FORTRAN*** ist auch nach knapp sechzig Jahren heutzutage noch an vielen Stellen anzutreffen. Wenn man Konstrukte wie ein arithmetisches *if*, *goto*-Anweisungen und merkwürdige Schleifen wie *do-continue* sieht, dann wirkt das auf heutige Programmierer wie Kauderwelsch, also ein Gemisch aus mehreren Sprachen und widerspricht den Paradigmen aktueller Programmiersprachen mit ihren strukturierten oder objektorientierten Inhalten. Der Name Fortran kommt von ***For****mula* ***Tran****slator* und war die erste höhere Programmiersprache. Sie wurde von *John Backus* entwickelt und die erste Version erschien bereits im Jahre 1957. Die Absicht dieser Sprache war es, Programmierern, die Formeln implementieren wollten, eine einfache Schnittstelle zur Verfügung zu stellen, als das zum Beispiel mit kryptischen Assembler-Anweisungen möglich war. Die Version ***FORTRAN 80***, die unter CP/M in Laufwerk ***F1:*** zu finden ist, stellt eine besondere Variante dar. Sie ist von Microsoft und ist eine begrenzte Untermenge von ***FORTRAN IV*** und kommt hinsichtlich seines Funktionsumfangs nicht einmal annähernd an ***FORTRAN 7*** heran. Manche Stimmen behaupten, dass es eine Implementierung von FORTRAN 66 ist. Näheres dazu ist im Internet zu finden. Hier ist ein Link zu FORTRAN, wo unter anderem natürlich die verschiedenen Versionen beschrieben werden.

https://fortranwiki.org/fortran/show/HomePage

Aber all das soll uns nicht weiter stören, denn wir wollen das Gefühl dafür zum Leben erwecken, unter CP/M in FORTRAN programmieren zu können und erste Versuche damit lassen erahnen, was es mit dieser Programmiersprache auf sich hat. Über Sinn und Unsinn hatte ich schon zu Beginn referiert und es ist - ich wiederhole mich hier gerne noch einmal - gelinde gesagt egal, was andere zu diesem Thema beitragen mögen.

Lassen wir uns einfach einmal einsteigen. Wie schaut es mit einem entsprechenden Manual aus?

Wo finde ich das *FORTRAN 80* Manual?

Das *FORTRAN 80* Manual befindet sich im gleichen Verzeichnis, wie das Programm selbst und kann über *manual.pdf* angesehen werden. Zusätzlich ist ein *User's Manual* unter dem Namen *Addendum.pdf* vorhanden.

Wenn es um das Kodieren von FOTRAN 80 Code geht, dann spiel die Formatierung eine entscheidende Rolle, die bei Nichteinhaltung unweigerlich zu Fehlern führt. FORTRAN in dieser Version ist keine Sprache mit einer freien Formatierung, sondern hat eine Reihe strenger Regeln für die Formatierung des Quellcodes. Die wichtigsten Regeln für die Spaltenposition sind:

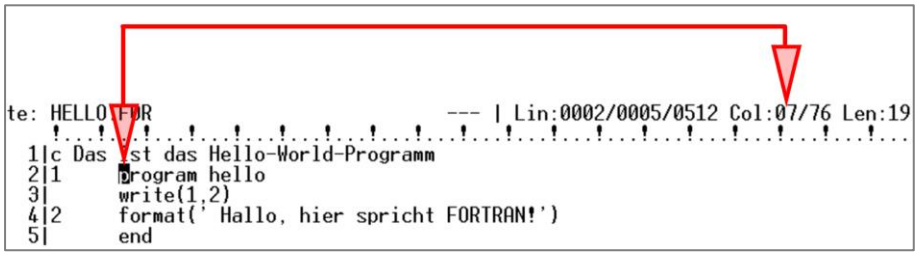
Spalte	Bedeutung
1	Leer, oder ein "c" oder "*" für Kommentare
2 bis 5	Anweisungs-Label (optional)
6	Fortsetzung der vorherigen Zeile (optional)
7 bis 72	Anweisungen
73 bis 80	Laufende Nummer (optional, wird heute selten verwendet)

Tabelle 12 - Spaltenformatierung in FORTRAN

Die meisten Zeilen in einem FORTRAN beginnen mit 6 Leerzeichen und enden vor Spalte 72, d.h. es wird nur das Anweisungsfeld verwendet. Ab FORTRAN 90 ist ein freies Format erlaubt.

6.1 Das Hello-World-Programm

Sehen wir uns also das *Hello-World*-Programm in FORTRAN an. Ich habe den Cursor für den eigentlichen Code einmal an die richtige Stelle gesetzt und wie zu sehen ist, handelt es sich um die Spalte 7 (Col: 07).



Sehen wir uns den sehr einfachen Code genauer an. In Zeile 1 ist eine Kommentarzeile zu erkennen, die über den Buchstaben ***c*** eingeleitet wird. Da es sich nicht um eine Anweisung handelt, kann dieser Kommentar in Spalte 1 beginnen. In Zeile 2 befindet sich in Spalte 1 ein Label und ab Spalte 7 (zuvor demnach 6 Leerzeichen) dann der erste Befehl, der mit dem Schlüsselwort ***program*** den Programmnamen vergibt. In Zeile 3 steht der ***write***-Befehl, der zwei Argumente erwartet. Die erste Zahl gibt das Ziel der Ausgabe an, was bei uns die 1 ist und auf die Konsole weist, was jedoch systemabhängig ist. Das zweite Argument deutet auf eine vorhandene Formatanweisung hin, die in diesem Fall die 2 ist. Diese Formatanweisung ***format*** ist in Zeile 4 zu finden, in der in Spalte 1 als Label besagte 2 zu sehen ist. Diese Formatanweisung beinhaltet lediglich eine Zeichenkette, die jedoch mit einem Leerzeichen beginnt. Das Programm endet mit dem Schlüsselwort ***end*** in Zeile 5. Kommen wir zum Ablauf der Erstellung einer ausführbaren COM-Datei. Auf dem genannten Laufwerk sind die beiden Programme ***f80*** (Compiler) und ***l80*** (Linker) vorhanden. Auf der folgenden Abbildung ist der vereinfachte Zusammenhang zu sehen.

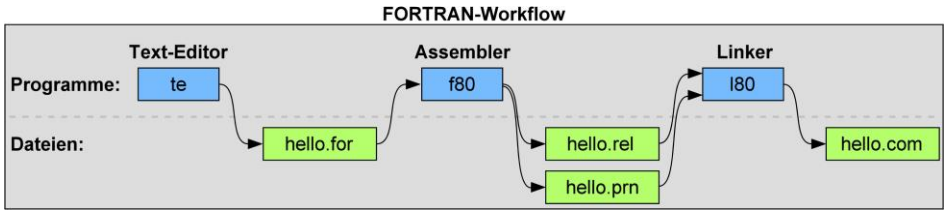


Abbildung 63 - Der Fortran-Workflow

Sehen wir uns das einmal konkret an. Nachdem der Quellcode unter dem Namen ***hello.for*** abgespeichert wurde, erfolgt zuerst der Aufruf des FORTRAN-Compilers über die Eingabe von ***f80***. Als Prompt wird der Stern * zurückgegeben.

```
F1>f80
*
```

Im nächsten Schritt wird laut User's-Manual die Befehlszeile ***hello,hello=hello*** eingegeben. Dies bedeutet im Detail: Compile *HELLO.FOR*, schreibe den Objectcode in *HELLO.REL* und das Listing in *HELLO.PRN*. Nähere Informationen mit weiteren Optionen sind im genannten Dokument auf der Seite 5 und nachfolgenden Seiten zu finden. Nach der Eingabe erscheint die Ausgabe des Programmnamens.

```
*hello,hello=hello
HELLO

*
```

Im Anschluss wird der Compiler über das Drücken der Tastenkombination ***Strg-C*** verlassen. Der Linker wird über die Eingabe von ***l80*** aufgerufen. Auch hier meldet sich das Programm lediglich mit dem Prompt in Form des Sterns.

```
F1>l80

Link-80  3.44  09-Dec-81  Copyright (c) 1981 Microsoft

*
```

Nachfolgend muss zuerst das zuvor kompilierte Programm mit dem Namen *hello* aufgerufen werden. Es werden die nachfolgenden Informationen zurückgeliefert, die für uns jedoch nebensächlich sind.

```
*hello
Data    0103    013E    <   59>

 FORLIB RQUEST
-$EX    0138    -$INIT  0129    -$ND    0135
-$W2    0132
    4 Undefined Global(s)
53460 Bytes Free

*
```

Das Programm könnte jetzt ohne die Generierung einer COM-Datei über die Eingabe von ***/g*** für *GO* gestartet werden. Im Anschluss erfolgt ein Rücksprung zum Betriebssystem.

```
*/g

Data    0103    1A24    < 6433>

45882 Bytes Free
[0125   1A24       26]
[Begin execution]

Hallo, hier spricht FORTRAN!


RunCPM Version 6.0 (CP/M 60K)

F1>
```

Zu Generierung einer COM-Datei wird die nachfolgende Befehlszeile mit dem Kommando ***hello/n/e*** abgesetzt. Nähere Informationen mit weiteren Optionen sind im genannten Dokument auf der Seite 11 und nachfolgenden Seiten zu finden. Im Anschluss wird der Linker automatisch verlassen (***/e*** bewirkt: Exit Linker, Rückkehr zum Betriebssystem) und die gewünschte *COM*-Datei wurde generiert, wie wir das mit dem *DIR*-Kommando überprüft haben.

```
*hello/n/e

Data    0103    1A24    < 6433>

45882 Bytes Free
[0125   1A24       26]


RunCPM Version 6.0 (CP/M 60K)

F1>dir hello.com
F: HELLO    COM
F1>
```

Jetzt kann da Programm wie gewohnt über den Dateinamen aufgerufen werden.

```
F1>hello

Hallo, hier spricht FORTRAN!


RunCPM Version 6.0 (CP/M 60K)

F1>
```

6.2 Ein weiteres FORTRAN-Programm

Ein weiteres FORTRAN-Programm soll eine wiederholte Ausführung von Programmzeilen anhand einer Schleife verdeutlichen.

```
te: LOOP.FOR                          --- | Lin:0013/0013/0512 Col:10/76 Len:09
    !...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...
  1|c Ein weiteres FORTRAN-Programm
  2|      program loop
  3|      integer i
  4|      write(1,1)
  5|      write(1,2)
  6|    1 format(' Ein Programm zur Umsetzung')
  7|    2 format(' einer Schleife')
  8|    3 i = i + 1
  9|      write(1,10) i
 10|      if(i.eq.10) stop
 11|      goto 3
 12|   10 format(i3)
 13|      end
```

Sehen wir uns den sehr einfachen Code genauer an. In Zeile 3 wird die Variable ***i*** von Datentyp *Integer* deklariert. Die Zeile 4 und 5 bewirken eine Ausgabe von Text, der in den Zeilen 6 und 7 über *format*-Anweisungen definiert

wurde. Die eigentliche Schleife wird in den Zeilen 8 bis 11 realisiert, wobei in Zeile 8 eine Inkrementierung der Variablen *i* erfolgt und diese Zeile gleichzeitig mit dem Label 3 versehen ist. Darüber wird erreicht, dass der *goto*-Befehl aus Zeile 11 immer wieder nach Label 3 verzweigen kann. In Zeile 9 kommt es über den *write*-Befehl zu einer Ausgabe der Variablen *i* auf der Konsole, die mithilfe der *format*-Anweisung aus Zeile 10 erreicht wird. In der format-Anweisung steht ***i3***. Dieser Formatcode besagt, dass es sich um den Datentyp Integer mit einer Feldweite von 3 handelt, also der Ausgabe drei Druckpositionen zur Verfügung stehen. Damit es jedoch zu keiner Endlosschleife kommt, wird in Zeile 10 über eine *if*-Anweisung der Inhalt der Variablen *i* überwacht und wenn dieser größer 10 ist, erfolgt die Ausführung des *stop*-Befehls. Über ***eq***, was die Abkürzung für *equal* (gleich) ist, wird auf Gleichheit getestet. Weitere Möglichkeiten sind ***lt*** was *less than* (kleiner gleich) bedeutet und ***gt*** für *greater than* (größer gleich). Die Ausgabe schaut wie folgt aus.

```
F1>loop

Ein Programm zur Umsetzung
einer Schleife
 1
 2
 3
 4
 5
 6
 7
 8
 9
10 STOP


RunCPM Version 6.0 (CP/M 60K)

F1>
```

6.3 Die Bildung des Mittelwertes

Kommen wir zu einem weiteren Programm, das zwei Werte über die Konsole einliest, um dann den Mittelwert zu berechnen. In der Zeile 3 werden die drei Variablen *a*, *b* und *c* vom Datentyp ***real*** deklariert. Es handelt sich dabei um Fließkommavariablen. In der darauffolgenden Zeile 4 werden über die ***read***-Anweisung die Variablen *a* und *b* mit Werten über die Tastatur eingelesen und durch ein Komma getrennt eingegeben. Es kommt hier die *format*-Anweisung aus Zeile 7 zum Tragen, die 4 Felder berücksichtigt, für die Ausgabe 20 Spalten zur Verfügung stellt und 6 Ziffern angezeigt werden. In Zeile 6 wird das Ergebnis der Mittelwertbildung über die ***write***-Anweisung zur Anzeige gebracht, wobei wiederum die *format*-Anweisung aus Zeile 7 berücksichtigt wird.

```
te: MITTEL.FOR                          --- | Lin:0008/0008/0512 Col:10/76 Len:09
    !...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...!...
  1|c Das ist das Hello-World-Programm
  2|      program mittel
  3|      real a,b,c
  4|      read(1,100)a,b
  5|      c=(a+b)/2
  6|      write(1,100)c
  7|  100 format(4e20.6)
  8|      end
```

Hier die Ein- und Ausgabe auf der Konsole.

```
F1>mittel
2.1,4.5

        .210000E+01         .450000E+01         .330000E+01


RunCPM Version 6.0 (CP/M 60K)

F1>
```

Es ist zu sehen, dass die Ergebnisse in einem wissenschaftlichen Format über die sogenannte *exponentielle Notation* angezeigt werden. Das Ergebnis *0.210000E+01* bedeutet, dass der angezeigte Wert mit 10 in der 1.Potenz multipliziert werden muss, was 2.1 ergibt und der Eingabe des ersten Wertes entspricht.

7 Namhafte CP/M-Programme

Natürlich gibt es unter CP/M sehr namhafte Programme, die später unter dem Betriebssystem *Windows* als neue und eigene Entwicklungen präsentiert wurden. Ich kann und will es nicht beurteilen, ob hier viele Ideen einfach übernommen wurden, denn das muss jeder selbst recherchieren. Wer an der Wahrheit interessiert ist, wird sie auch finden.

7.1 WordStar

Das Programm *WordStar* war eines der ersten Textverarbeitungsprogramme für einen Computer und die Version 1.0 war eine Weiterentwicklung des Programms *WordMaster*, das im September 1978 für das Betriebssystem CP/M entwickelt beziehungsweise veröffentlicht wurde. Natürlich ist *WordStar* in der Version *Release 4* auch standardmäßig auf *RunCPM* installiert und auf dem Laufwerk ***E0:*** zu finden.

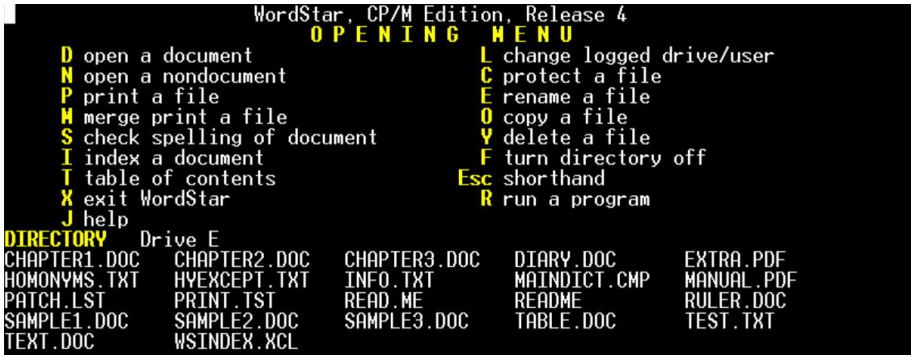


Abbildung 64 - Wordstar

Wie schaut es mit einem entsprechenden Manual aus?

	**Wo finde ich das Wordstar 4 Manual?**
Das *Wordstar 4* Manual befindet sich im gleichen Verzeichnis, wie das Programm selbst und kann über *manual.pdf* angesehen werden.	

Wordstar kann auch als ganz normaler Texteditor genutzt werden, um Quelldateien für diverse Programmiersprachen zu erstellen oder zu bearbeiten. Ich zeige das hier kurz an einem Beispiel für die Programmierung unter *C*. Nach dem Start von Wordstar ist das sogenannte *Logged Drive* auf *E:* eingestellt. Die von uns zuvor erstellen Quellcodes liegen auf dem Laufwerk *C1:* und wir müssen unter Wordstar auf dieses Laufwerk umschalten, so dass die dort vorhandenen Dateien sichtbar werden. Es muss also, wie das im oberen Menü zu sehen ist, die Taste ***L*** für *change logged drive/user* ausgewählt werden.

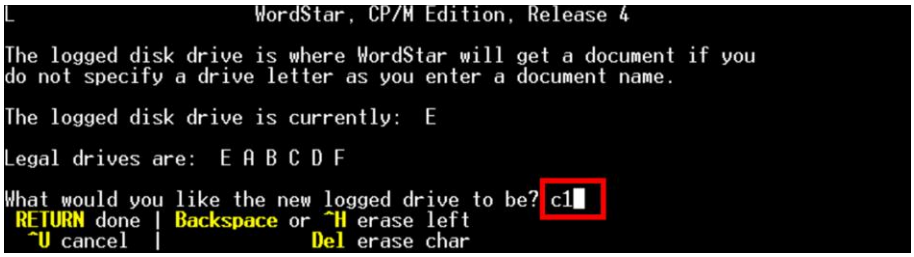


Abbildung 65 - Die Auswahl des aktuellen Laufwerkes

Im Anschluss werden alle auf dem ausgewählten Laufwerk vorhandenen Dateien angezeigt. Wenn es sehr viele Dateien sind, kann über die Tastenkombinationen ***Strg-W*** (Scroll-Up) und ***Strg-Z*** (Scroll-Down) geblättert werden.

Abbildung 66 - Die Dateien auf Laufwerk C1 werden angezeigt

Zur Auswahl der gewünschten Datei muss jetzt die Taste ***D*** gedrückt werden, um dann den gewünschten Dateinamen einzugeben und ich wähle hier *hello.c* aus.

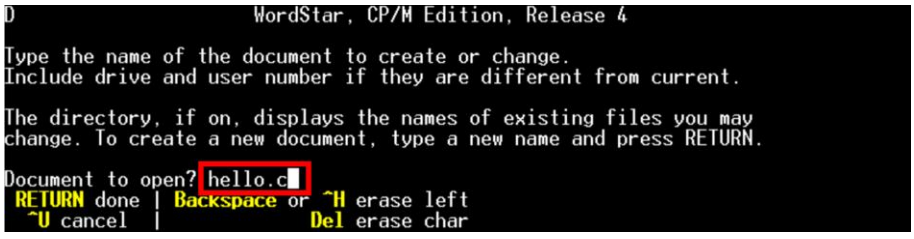


Abbildung 67 - Die Auswahl der Datei hello.c

Nach der Bestätigung über die *RETURN*-Taste wird die Datei geladen und kann editiert werden.

Abbildung 68 - Das Editieren der geladenen Datei

Im Editor-Menü ist zu sehen, wie die Navigation des Cursors zu erreichen ist. Es handelt sich um die schon bekannten Steuerungsfunktionen, die ich schon bei *Turbo Pascal* gezeigt hatte und hier noch einmal zu sehen sind.

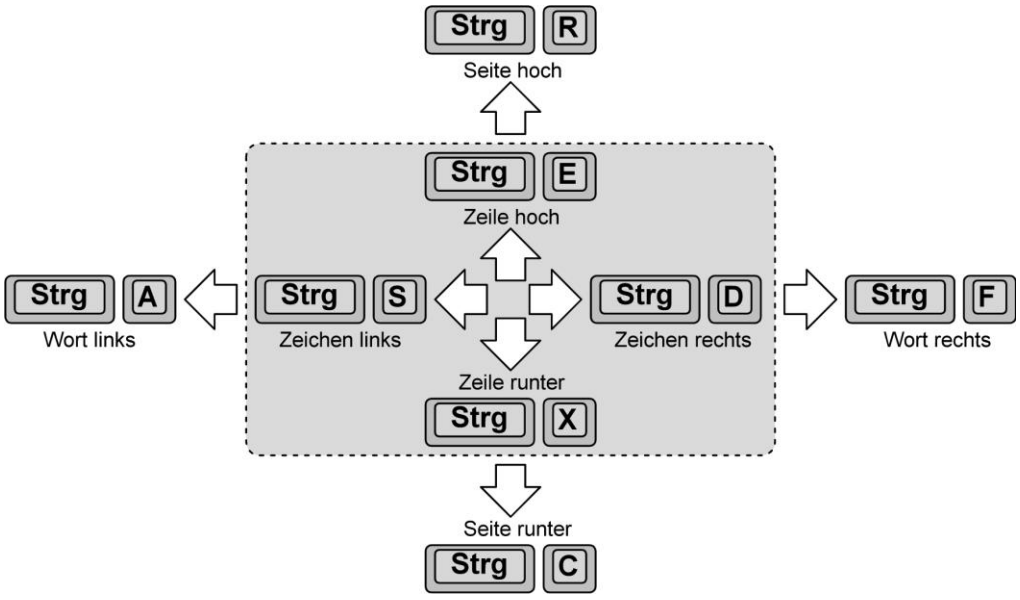
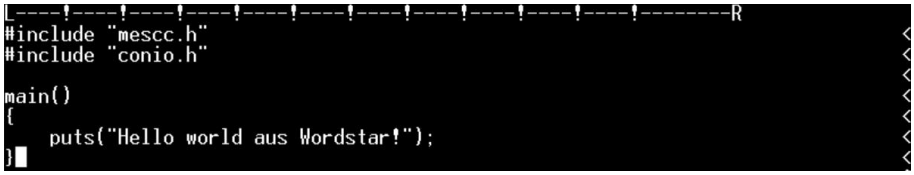
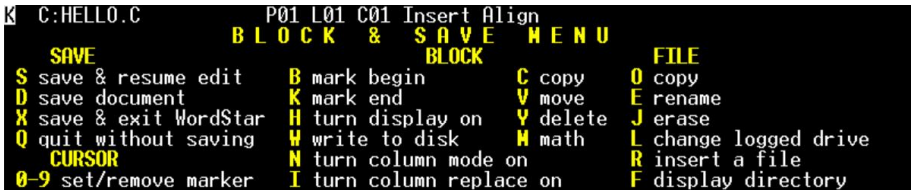


Abbildung 69 - Die Navigation im Editor

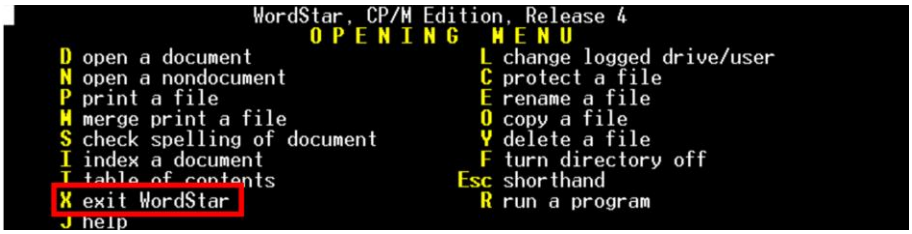
Natürlich gibt es noch viele weitere Steuerfunktionen, die ich jedoch nicht alle benennen werde. Ich passe den Quelltext etwas an und speichere ihn unter dem gleichen Namen wieder ab.



Um den Editiermodus zu verlassen, muss die Tastenkombination ***Strg-K*** gedrückt werden. Es erscheint ein weiteres Menü, das die möglichen Optionen anzeigt.



Über das schon aus *Turbo Pascal* bekannte Drücken von ***D*** wird die Datei gespeichert und der Editor verlassen. Die anderen Optionen sollten selbst erforscht werden. Im Anschluss kann Wordstar über das Drücken von ***X*** (exit Wordstar) verlassen werden.



Zur Sicherheit sehen wir einmal auf dem Laufwerk ***C1:*** nach und lassen uns den Inhalt der Datei *hello.c* anzeigen. So wie es ausschaut, hat das Editieren funktioniert!

```
E0>c:
C0>user 1
C1>type hello.c
#include "mescc.h"
#include "conio.h"

main()
{
    puts("Hello world aus Wordstar!");
}

C1>
```

7.2 Multiplan

Das Programm *Multiplan* ist ein Tabellenkalkulationsprogramm der Firma *Microsoft*. Es wurde im Jahre 1982 veröffentlicht und zunächst für CP/M und dann MS-DOS, später auch für den Rechner *Apple II* sowie den *Apple Macintosh* umgesetzt. Es gab sogar eine Version für den *Commodore C64*. Die entsprechenden COM-Dateien sind im Netz zu finden. Multiplan ist auf dem Laufwerk ***M0:*** in der *Version 1.06* zu finden.

Abbildung 70 - Multiplan

Wie schaut es mit einem entsprechenden Manual aus?

	**Wo finde ich das Multiplan Manual?**
Das *Multiplan* Manual befindet sich im gleichen Verzeichnis, wie das Programm selbst und kann über *manual.pdf* angesehen werden.	

Lassen wir uns auch hier ein kleines Beispiel ausprobieren, wie man das heutzutage auch aus der Tabellenkalkulation *Excel* kennt. Die Navigation zur Auswahl der einzelnen Zellen kann über die Cursortasten erfolgen, oder, falls das nicht funktionieren sollte, über die schon bekannten Tastenkombinationen der Steuerungstaste *Strg*. Ich gebe jetzt in Zelle ***R1C1*** (Row=1, Column=1) den Wert ***17*** ein und in der darunterliegenden Zelle ***R2C1*** (Row=2, Column=1) den Wert ***23*** ein. Die Eingabe des Wertes erfolgt unterhalb und nicht innerhalb der markierten Zelle hinter dem Schriftzug ***VALUE*** im unteren Bereich und muss mit der *RETURN*-Taste bestätigt werden.

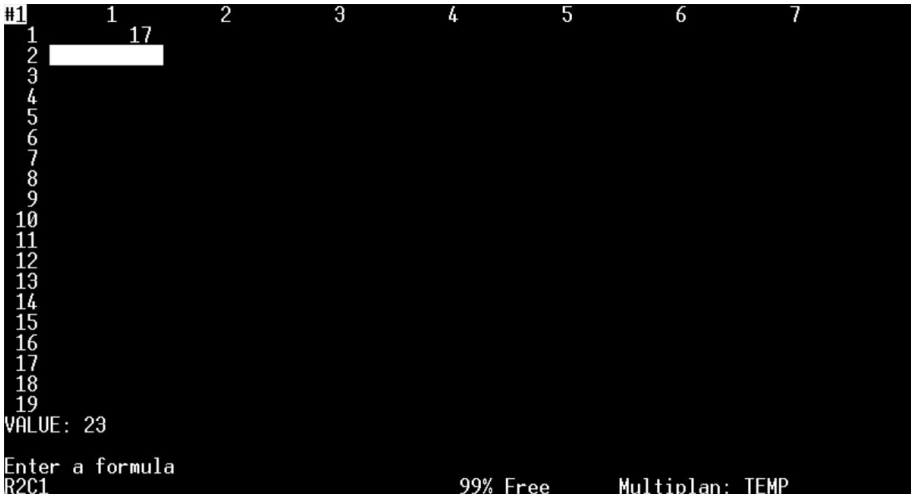
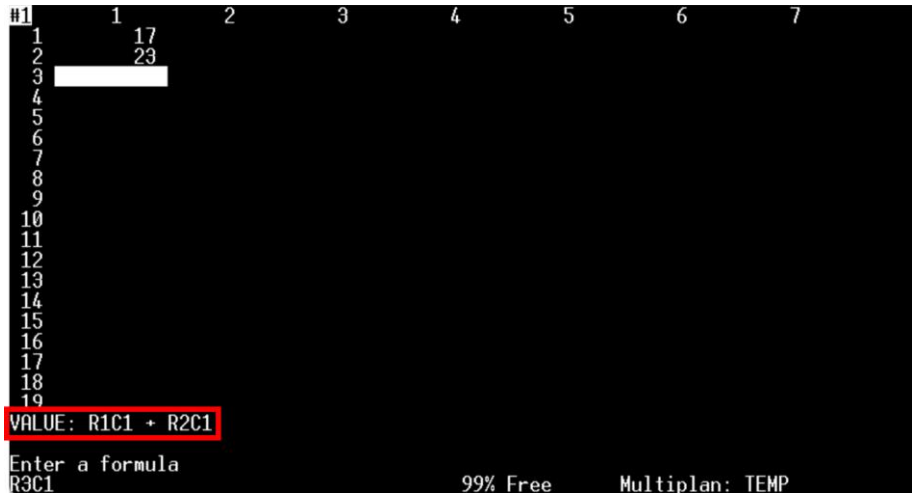


Abbildung 71 - Die Eingabe von Werten innerhalb von Zellen

Im Endeffekt schaut es dann wie folgt aus.

Nun wollen wir in der darunterliegenden Zelle *R3C1* eine Formel hinterlegen, die die Summe von *R1C1* und *R2C1* bildet. Zuerst muss natürlich die Zelle *R3C1* in bekannter Weise markiert werden, um dann zuerst das Gleichheitszeichen = einzutippen. Im unteren Bereich erscheint der Schriftzug VALUE, hinter dem die gezeigte Summenformel eingetippt wird.



Nach der Bestätigung über die *RETURN*-Taste erscheint sofort das Ergebnis.

Das interessante an derartigen Tabellenkalkulationsprogrammen ist ja, dass eine unmittelbare Neuberechnung aufgrund hinterlegter Formeln erfolgt, wenn sich Änderungen von Inhalten berücksichtigter Zellen ergeben. Wird zum Beispiel der Wert aus Zelle *R1C1* geändert, erscheint nach einer Bestätigung sofort das Ergebnis der Summenbildung in Zelle *R3C1*. Um Multiplan zu verlassen, muss die Taste ***Q*** für *Quit* gedrückt und nachfolgend mit ***Y*** für *Yes* bestätigt werden. Das soll als kleine Einführung genügen.

7.3 dBase II

Das Programm *dBase* ist ein Datenbankmanagementprogramm, das von der Firma *Ashton-Tate* ursprünglich für CP/M entwickelt wurde. Später kam es dann zu Portierungen für *DOS* auf einem IBM-PC. Das Programm ***dBase II*** ist in der *Version 2.43* auf dem Laufwerk ***D1:*** zu finden.

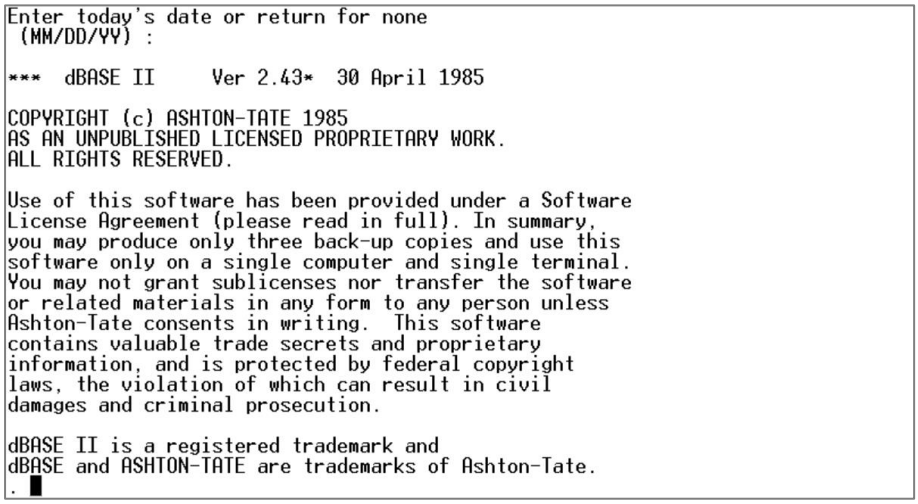


Abbildung 72 - dBase II

Wir haben es hier nicht mit einem Fullscreen-Editor zu tun und müssen alles an Befehlen hinter dem dBase-Prompt, dem *Punkt*, eingeben. Ich möchte dBase an einem sehr einfachen Beispiel präsentieren. Zuerst wollen wir eine Arbeitsdatei erstellen, was mit dem Befehl *create* erfolgt. Im Anschluss muss ein Dateiname eingeben werden. Ich möchte hier verschiedene Familiennamen hinterlegen und gebe *familie* ein.

Die nächste Eingabeaufforderung fragt nach den Merkmalen der einzelnen Datenfelder, einschließlich des Namens, Datentyp der Daten, Breite (Anzahl der Zeichen) und Anzahl der Dezimalstellen. Für folgende Felder habe ich mich entschieden.

- Nachname
- Vorname
- Geburtsdatum
- Kinder

Im nächsten Schritt müssen wir uns über die Datentypen der einzelnen Felder im Klaren sein. Es stehen drei Möglichkeiten zur Auswahl.

- ***C***: Character - max. 254 Zeichen
- ***N***: Numeric - Nur Ziffern, Plus- und Minuszeichen sowie das Dezimalkomma sind gültige Werte. Numerische Felder sind auf zehn Stellen genau, einschließlich des Vorzeichens und des Dezimalkomma
- ***L***: Logic - Logische Felder enthalten die Werte *wahr* oder *falsch*. Gültige Werte sind ***Y***, ***y***, ***T*** und ***t*** für wahr und ***F***, ***f***, ***N***, und ***n*** für falsch

Die folgende Datenbank-Struktur habe ich eingegeben.

```
. create familie
Enter record structure as follows:
Field   Name, Type, Width, Decimal places
 001     Nachname,c,20
 002     Vorname,c,10
 003     Geburt,c,10
 004     Kinder,n,2
 005
Input data now?Y
```

Will man keine weiteren Datensätze eingeben, muss lediglich die ***RETURN***-Taste gedrückt werden und es wird dann im Anschluss gefragt, ob die Daten eingegeben werden sollen. Ich habe das mit ***Y*** bestätigt. Nachfolgend ist die Eingabe des ersten Datensatzes (*Record 00001*) zu sehen.

```
RECORD # 00001
NACHNAME  :Hagen              :
VORNAME   :Joe       :
GEBURT    :1983/09/01:
KINDER    :3:
```

Dann der zweite...

```
RECORD # 00002
NACHNAME  :Schmitt            :
VORNAME   :Benno     :
GEBURT    :2001/12/01:
KINDER    :0:
```

Auf diese Weise habe ich mehrere Datensätze der Datenbank *familie* hinzugefügt. Soll die Eingabe beendet werden, muss wieder lediglich die *RETURN*-Taste gedrückt werden. Um sich den Inhalt der Datenbank anzusehen, muss zum einem die Datenbank - zum Beispiel nach einem Neustart der Anwendung - mit dem Befehl ***use <Datenbankname>*** ausgewählt werden. Im Anschluss können mit dem Befehl ***list*** alle Datensätze zur Anzeige gebracht werden.

```
. use familie
. list
00001  Hagen               Joe        1983/09/01   3
00002  Schmitt             Benno      2001/12/01   0
00003  Mueller             Ralf       1962/03/17   0
00004  Held                Monika     1974/12/28   1
00005  Ullmer              Ingrid     1945/11/08   3
00006  Goertz              Harald     1967/11/11   0
00007  Goertz              Marlene    1957/01/01   0
.
```

Über den Befehl ***structure*** ist es möglich, sich die Struktur der Datenbank wieder anzusehen.

```
. list structure
Structure for file: D:FAMILIE .DBF
Number of records: 00007
Date of last update: 00/00/00
Primary use database
Fld     Name        Type Width  Dec
001     NACHNAME    C    020
002     VORNAME     C    010
003     GEBURT      C    010
004     KINDER      N    002
** Total **            00043
.
```

Kommen wir zu einem Punkt, was eine Datenbank so interessant macht. Ich möchte einen Befehl absetzen, der die komplette Datenbank mit all ihren Datensätzen nach Menschen durchsucht, die keine Kinder haben. Folgende Eingabe ist dazu erforderlich.

```
. list for Kinder=0
00002  Schmitt              Benno      2001/12/01   0
00003  Mueller              Ralf       1962/03/17   0
00006  Goertz               Harald     1967/11/11   0
00007  Goertz               Marlene    1957/01/01   0
. 
```

Es wird also eine Liste erstellt, bei der das Feld *Kinder* den Wert 0 enthält. Welche Menschen haben mehr wie zwei Kinder? Das kann mit der folgenden Abfrage realisiert werden.

```
. list for Kinder>2
00001  Hagen                Joe        1983/09/01   3
00005  Ullmer               Ingrid     1945/11/08   3
. 
```

Um *dBase* zu verlassen, muss auf dem Prompt lediglich *quit* eingegeben werden.

```
. quit
*** End run   dBASE II   ***

Remember to back-up your data.

RunCPM Version 6.0 (CP/M 60K)

D1>
```

Das sollte als kleines Beispiel genügen und macht hoffentlich Spaß auf mehr.

Wo finde ich das dBase II Manual?

Das *dBase II* Manual befindet sich im gleichen Verzeichnis, wie das Programm selbst und kann über *manual.pdf* angesehen werden.

8 Spiele

Wer sollte es glauben, es gibt auch unter CP/M Spiele. Natürlich können hier keine Spiele mit hochauflösender Grafik zum Einsatz kommen. Nein, es können überhaupt keine Grafiken anzeigt werden, doch das macht überhaupt nichts! In den frühen Zeiten, wo es lediglich möglich war, Texte auf der Konsole anzuzeigen, waren sogenannte *Text-Adventures* sehr beliebt. Dabei handelt es sich um Spiele, die nicht über eine Maus - so wie das heutzutage üblich ist - zu steuern, sondern über eine Dialogsteuerung über die Tastatur. Das sind dann zum Beispiel Eingaben wie ***look*** für schauen, was in der näheren Umgebung an interessanten Dingen zu sehen ist oder ***take shovel***, um die auf dem Boden liegende Schaufel in ein Inventar, also den Rucksack zu packen. Über die Eingaben der Himmelsrichtungen wie ***N***, ***S***, ***E***, ***W*** kann in die entsprechenden Richtungen geschaut beziehungsweise über ***go*** und die genannten Himmelsrichtungen kann in diese Richtungen gegangen werden. Es handelt sich hier natürlich nur ein paar rudimentäre Beispiele für mögliche Aktionen und hängen vom jeweiligen Spiel ab.

8.1 Zork

Die hier genannten Befehle beziehen sich zum Beispiel auf das weltbekannte Text-Adventure ***Zork***, was auch heute noch immer gerne gespielt wird. Das Spiel befindet sich auf Laufwerk ***A5:*** und wird einfach über die Eingabe von *zork* gestartet. Zuvor sollte jedoch klar sein, über welche Befehle man das Spiel beeinflussen und wie man sich bewegen kann.

Kommando	**Funktion**
take	nehmen
n, go n, north, go north	nach Norden gehen
s, go s, south, go south	nach Süden gehen
e, go e, east, go east	nach Osten gehen
w, g w, west, go west	nach Westen gehen
up	nach oben gehen
down	nach unten gehen
climb	Klettern
drop	Gegenstand fallen lassen
open	öffnen
move	bewegen
enter	Zutritt verschaffen
i	Inventar anzeigen

Tabelle 13 - Zork-Kommandos

Hat man das Spiel also gestartet, befindet man sich in einer Kommunikation mit der Konsole und es werden Informationen zum jetzigen Standpunkt beziehungsweise Ort geliefert, an dem man sich gerade befindet. Da sich alles natürlich in englischer Sprache abspielt, sollte man schon etwas Englisch können.

```
A5>zork1
ZORK I: The Great Underground Empire
Copyright 1982 by Infocom, Inc.
All rights reserved.
ZORK is a trademark of Infocom, Inc.
Release 25 / Serial number 820515

West of House
You are standing in an open field west of a white house, with
a boarded front door.
There is a small mailbox here.

>
```

Abbildung 73 - Der Startpunkt von Zork1

Folgendes wird mitgeteilt: ***Sie stehen auf einem offenen Feld westlich eines weißen Hauses mit einer vernagelten Eingangstür. Hier befindet sich ein kleiner Briefkasten.***

Nun kann man natürlich versuchen, sich in die vier möglichen Himmelsrichtungen zu bewegen und ***n***, ***s***, ***e*** oder ***w*** eingeben. Doch da ist ein Hinweis, dass ein kleiner Briefkasten in Reichweite ist. Man sollte also nachsehen, ob sich darin etwas befindet. Ich führe nachfolgend mehrere Aktionen durch.

- Über ***open mailbox*** öffne ich den Briefkasten
- Über ***take leaflet*** nehme ich den Brief
- Über ***i*** schaue ich nach, was sich im Inventar befindet

```
>open mailbox
Opening the mailbox reveals a leaflet.

>take leaflet
Taken.

>i
You are carrying:
  A leaflet

>
```

Nun kann man natürlich den Brief lesen und ich gebe ***read leaflet*** ein. Es wird ein netter Hinweis zum Zork-Spiel angezeigt, der aber nicht Spiel relevant ist.

```
>read leaflet
WELCOME TO ZORK
     ZORK is a game of adventure, danger, and low cunning. In
it you will explore some of the most amazing territory ever
seen by mortals.

    No computer should be without one!

    Copyright 1982 by Infocom, Inc.
          All rights reserved.
  ZORK is a trademark of Infocom, Inc.


>
```

Übersetzt bedeutet das: ***WILLKOMMEN BEI ZORK. ZORK ist ein Spiel voller Abenteuer, Gefahren und kleiner Tricks. Im Spiel werden Sie einige der erstaunlichsten Gebiete erkunden, die je von Sterblichen gesehen wurden.***

Damit man nicht den Überblick verliert, sollte man sich hinsichtlich der vorhandenen Orte, die man besuchen kann, eine Karte erstellen. Natürlich gibt es schon fertige Karten im Internet, denn das Spiel ist ja schon einige Jahrzehnte alt. Man muss lediglich den Suchbegriff ***zork map*** eingeben. Ich rate jedoch, nicht immer auf vorgekaute und fertige Dinge zurückzugreifen, wenn man das auch selbst machen kann. Das macht auf jeden Fall viel mehr Spaß. Wenn man bei Spielen an irgendwelchen Stellen nicht mehr weiterkommt, dann kann man sich natürlich mal einen Hinweis aus dem Netz ziehen. Das habe ich auch schon oft gemacht, denn manchmal sind die Rätsel so abstrus, dass es einfach kein Weiterkommen ohne Hilfe gibt.

Sehen wir mal, wie es im Spiel so weitergeht. Ich versuche mal um das Hausherumzugehen. Zuerst nach Süden und dann nach Osten. Im Süden ist nichts Besonderes zu sehen, doch auf der Ostseite ist wohl ein kleines Fenster, was wohl offen ist. Die Übersetzung lautet: ***Sie befinden sich hinter dem weißen Haus. Ein Pfad führt in den Wald nach Osten. In einer Ecke des Hauses befindet sich ein kleines Fenster, das leicht angelehnt ist.***

```
>s
South of House
You are facing the south side of a white house. There is no
door here, and all the windows are boarded.

>e
Behind House
You are behind the white house. A path leads into the forest
to the east. In one corner of the house there is a small
window which is slightly ajar.

>
```

Wir versuchen einmal, über das Fenster ins Haus zu gelangen und geben ***open window*** ein. Das sieht vielversprechend aus. Die Übersetzung lautet: ***Mit großer Mühe öffnen Sie das Fenster weit genug, um den Eintritt zu ermöglichen.***

```
>open window
With great effort, you open the window far enough to allow
entry.

>
```

Über den Befehl ***enter house*** können wir versuchen, ins Haus zu gelangen. Das schaut ebenfalls gut aus. Die Übersetzung dazu lautet: ***Sie befinden sich in der Küche des Weißen Hauses. Ein Tisch scheint vor kurzem für die Zubereitung von Speisen benutzt worden zu sein. Ein Durchgang führt nach Westen, und man sieht eine dunkle Treppe, die nach oben führt. Ein dunkler Schornstein führt nach unten und im Osten befindet sich ein kleines Fenster, das geöffnet ist. Auf dem Tisch liegt ein länglicher brauner Sack, der nach scharfen Paprika riecht. Eine Flasche steht auf dem Tisch. Die Glasflasche enthält eine Menge Wasser.***

```
>enter house
Kitchen
You are in the kitchen of the white house. A table seems to
have been used recently for the preparation of food. A passage
leads to the west and a dark staircase can be seen leading
upward. A dark chimney leads down and to the east is a small
window which is open.
On the table is an elongated brown sack, smelling of hot
peppers.
A bottle is sitting on the table.
The glass bottle contains:
  A quantity of water

>
```

Ok, dann versuche ich einfach mal mit dem Befehl ***take all*** alles zu nehmen, was sich das so gezeigt hat. Die Übersetzung dazu lautet: ***Küchentisch: Das kann doch nicht Ihr Ernst sein. Brauner Sack: Genommen. Glasflasche: Genommen. Wassermenge: Kann ich nicht erreichen.***

```
>take all
kitchen table: You can't be serious.
brown sack: Taken.
glass bottle: Taken.
quantity of water: I can't reach that.

>
```

Nun möchte ich das Spiel hier nicht weiter erläutern, denn das würde sicherlich mehrere Seiten in Anspruch nehmen. Jedenfalls kann man aufgrund der bisherigen Recherchen eine Karte erstellen, die mit den Dingen rund um das Haus und auch Bereiche innerhalb des Hauses angereichert ist, wie das auf der folgenden Abbildung zu erkennen ist. Ich rate, eine derartige Karte anzufertigen.

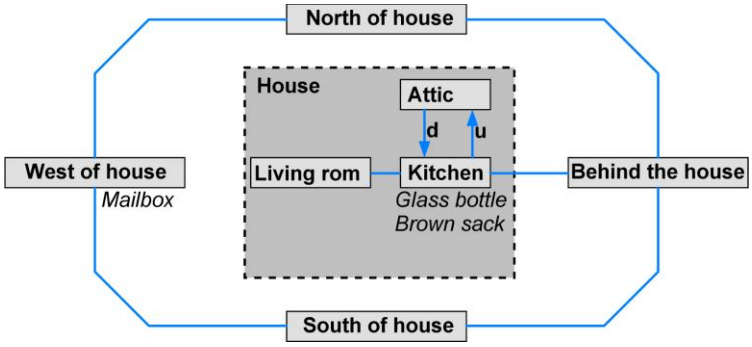


Abbildung 74 - Zork1-Karte (rund um's Haus)

Das soll es zum Spiel Zork erst einmal sein. Wer einfach keine Lust hat, eine derartige Karte selbst zu erstellen - welch eine Schande! - der kann sich über den folgenden Link schlau machen, was es alles für Orte in Zork 1 gibt.

https://mocagh.org/infocom/zork-map-front.pdf

Wer sich für ältere Spiele interessiert und auch ein paar interessante Bücher als PDF-Dateien sucht, für den ist diese Seite ein Eldorado!

https://mocagh.org/

Viel Spaß beim Erkunden der Umgebung!

8.2 Schach

Ein weiters sehr interessantes Spiel ist natürlich *Schach* und zu jener Zeit war ***Sargon*** sehr populär. Sargon wurde vom amerikanischen Ehepaar Dan und Kathe Spracklen in den späten 1970er und den 1980er Jahren programmiert, das damals als Maßstab für große Spielstärke und Komfort galt. Es wurde in Assembler programmiert und hatte als Motivation ein unfertiges Schachprogramm, das jedoch in Basic entwickelt wurde. Das Programm ist auf der folgenden Internetseite in der Version 2.1 zu finden.

http://www.retroarchive.org/cpm/games/games.htm

Nach dem Start sind erste einmal einige Fragen zu beantworten.

1. Die Frage, ob ein Piep bei einem Fehler zu hören sein soll.

DO YOU WANT THE ERROR BELL?(Y OR N):

2. Die Frage, ob ein neues Spiel begonnen (G), die Farbe gewechselt (C) oder das Spiel verlassen (X) werden soll.

SELECT OPTIONS

NEW GAME, CHANGE BOARD OR EXIT? (G,C,X):

3. Die Frage, ob der Drucker genutzt werden soll.

PRINTER?(Y,N)

4. Die Wahl der Farbe. (Weiß beginnt immer!)

YOUR COLOR? (B,W):

5. Die Wahl der Spielstärke.

LEVEL OF PLAY? (1-6):

6. Die Frage, ob nach jedem Zug, der eingegeben wurde, das Board aktualisiert werden soll.

DISPLAY BOARD EVERY MOVE?(Y OR N)

Im Anschluss wird das Spiel begonnen und es ist die folgende Anzeige zu sehen.

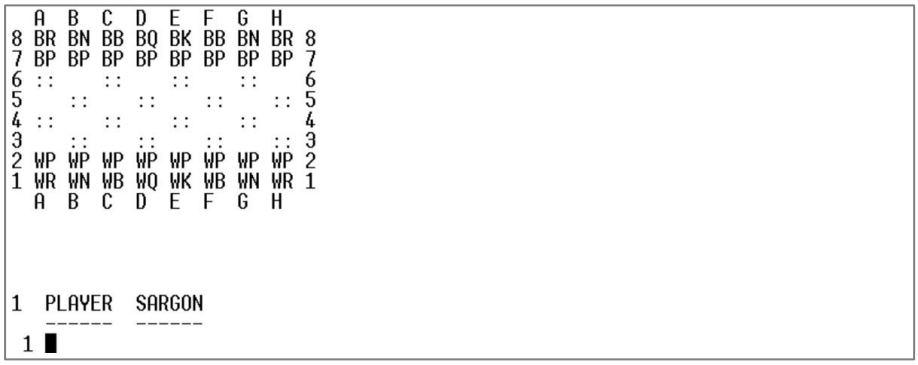


Abbildung 75 - Das Schachspiel Sargon - Ausgangsposition

Dabei haben die Buchstabenpaare der einzelnen Figuren die folgenden Bedeutungen, wobei ein führendes ***W*** immer *White*, also eine weiße Figure und ein ***B*** immer *Black*, also eine schwarze Figure (darf man das heute noch sagen oder ist das rassistisch?). Jedenfalls eine Figur mit dunkel pigmentiertem Holz.

- P: Pawn - Bauer
- R: Rook - Turm
- N: Knight - Springer
- B: Bishop - Läufer
- Q: Queen - Dame
- K: King - König

Um einen Zug durchzuführen, sagen wir Bauer von *D2* nach *D4*, muss lediglich diese Eingabe gemacht werden, ohne, dass diese mit *RETURN* bestätigt werden muss. Es ist nach der Aktualisierung des Brettes zu sehen, dass der weiße Bauer wie gewünscht bewegt wurde und Schwarz mit dem Gegenzug Springer *G8* nach *F6* geantwortet hat.

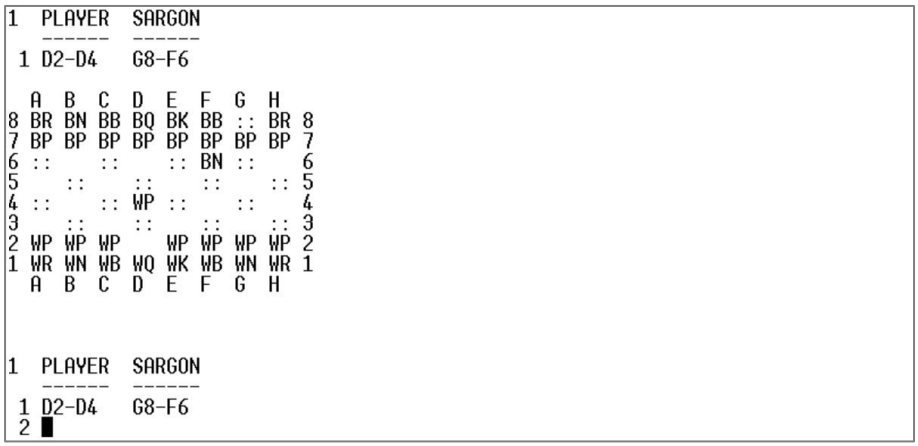


Abbildung 76 - Das Schachspiel Sargon - Zug und Gegenzug

Die Bedienungsanleitung ist unter der folgenden Internetadresse zu finden.

https://www.classic-computers.org.nz/system-80/software-manuals/manuals-Sargon-II.pdf

9 CP/M über reine Software-Emulationen

Nun ist es ebenfalls möglich, CP/M auch ohne spezielle Hardware zu nutzen. Dazu stehen verschiedene Emulatoren bereit. Ich möchte nachfolgend auf einige wenige eingehen, denn auch hier ist das Angebot sehr vielfältig. Ich kann jedenfalls sagen, dass ich in den 1980er Jahren, als ich mich mit dem Apple //e befasste, auch eine sogenannte ***Z-80 Softcard*** hatte und natürlich immer noch besitze. Diese Erweiterungskarte für einen der Apple II-Slots ist eine von der Firma *Microsoft* entwickelte *Plug-in-Prozessorkarte*, die den Computer in ein CP/M-System verwandelt. Bevor ich auf weitere Softwarelösungen hinsichtlich CP/M eingehen, möchte ich mit der Realisierung eines Apple II-Computers beginnen, die sich ***AppleWin*** nennt. AppleWin ist ein //e Open-Source-Software-Emulator für Windows und wurde ursprünglich von *Mike O'Brien* geschrieben.

9.1 Der AppleWin-Emulator

Die Internetseite von AppleWin ist unter der folgenden Adresse zu finden.

https://www.berlios.de/software/applewin/

Die Informationen auf dieser Seite besagen, dass unter *AppleWin* wohl die meisten Programme, die für den *Apple II+* oder den *Apple IIe* programmiert wurden, lauffähig sind. Das Interessante daran ist, dass auch die genannte Z-80 Softcard unterstützt wird. Auf der folgenden Abbildung ist AppleWin mit CP/M zu sehen.

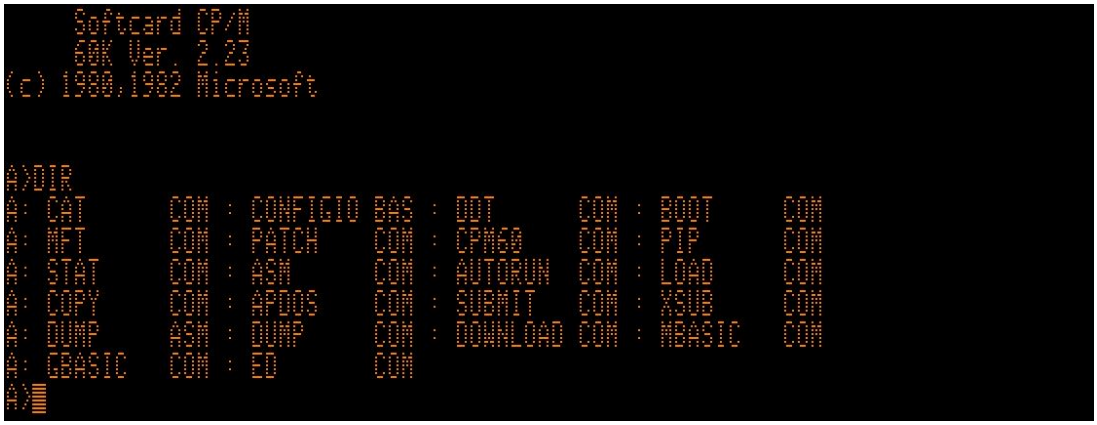


Abbildung 77 - AppleWin mit CP/M

Nach der Installation von AppleWin muss die Konfiguration über die untere Schaltfläche erfolgen.

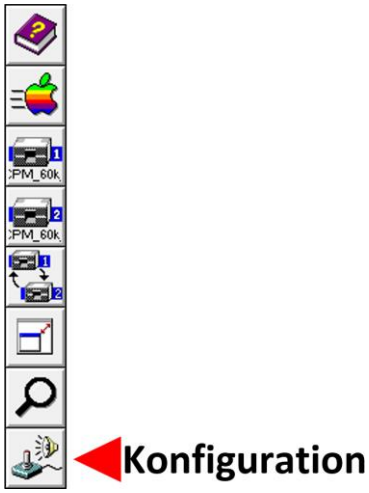


Abbildung 78 - Der Aufruf der AppleWin-Konfiguration

Im *Configuration-Reiter* habe ich den *Enhanced Apple //e* ausgewählt.

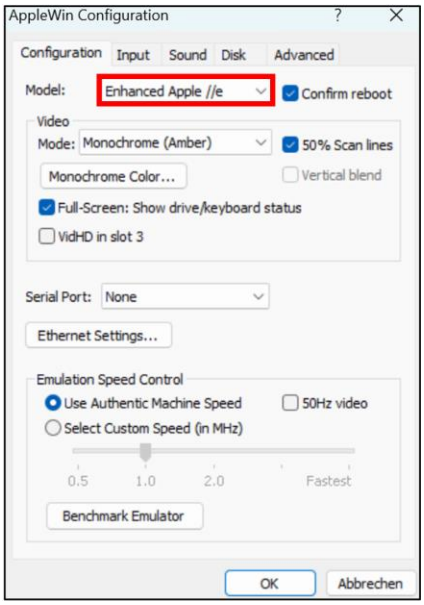


Abbildung 79 - Die Konfiguration von AppleWin - Configuration

Um die *Z-80-Softcard* einzubinden, muss in den *Input-Reiter* gewechselt werden. Ich habe bei mir Slot 4 ausgewählt.

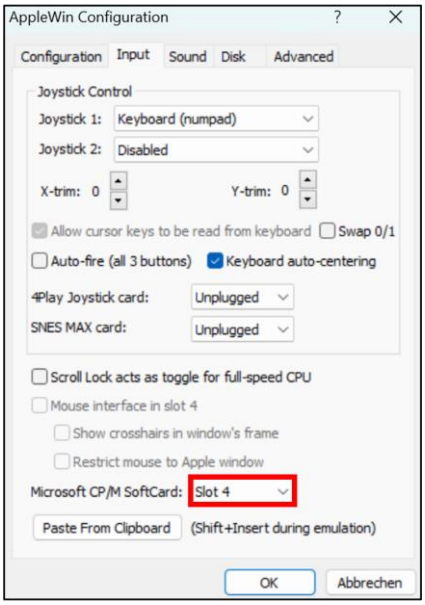


Abbildung 80 - Das Einbinden der Z-80-Softcard

Nun sind die grundlegenden Voraussetzungen für das Starten der Emulation getroffen. Was jetzt noch fehlt, ist natürlich die CP/M-Software, die hier erforderlich ist. Ich werde aber keinen konkreten Link anbieten, denn ich bin mir über die Rechtmäßigkeit über das Anbieten der Software auf verschiedenen Internetseiten nicht im Klaren und rate, dort selbst etwas zu suchen. Suchbegriffe wie *Apple II*, *Software*, *CP/M* führen sehr schnell zu Ergebnissen, aber ich werde keine Verantwortung oder Haftung über illegal heruntergeladene und genutzte Software übernehmen. Das liegt in der

Verantwortung jedes Einzelnen, sich darüber zu informieren. Ich persönlich verfüge über damals offiziell erworbene CP/M-Software für den Apple //e.

Zur Anschauung habe ich die beiden Disketten-Images ***CPM_60k_a.dsk*** und ***CPM_60k_b.dsk*** in den Emulator geladen, was über die markierten Schaltflächen erfolgt.

9.2 Der Z80-Emulator - EMUZ80

Ein CP/M-Emulator, der unter dem Windows-Betriebssystem lauffähig ist, ist der ***EMUZ80*** von *Ronald Daleske*. Er simuliert die CP/M-Version 2.2 und stellt zwei Laufwerke zur Verfügung. Das Programm kann ohne eine vorherige Installation CP/M Programme nutzen. Für Entwickler bietet der Emulator interessante Merkmale.

- Einzelschrittbetrieb (Single step)
- Unterbrechungspunkte (Breakpoints)
- Die Anzeige der CPU-Register
- Die Anzeige des Z80-Arbeitsspeichers (64K RAM)
- Die Anzeige des ausführenden Befehls (Trace)

Auf der nachfolgenden Abbildung ist der Emulator mit einer Oberfläche zu sehen.

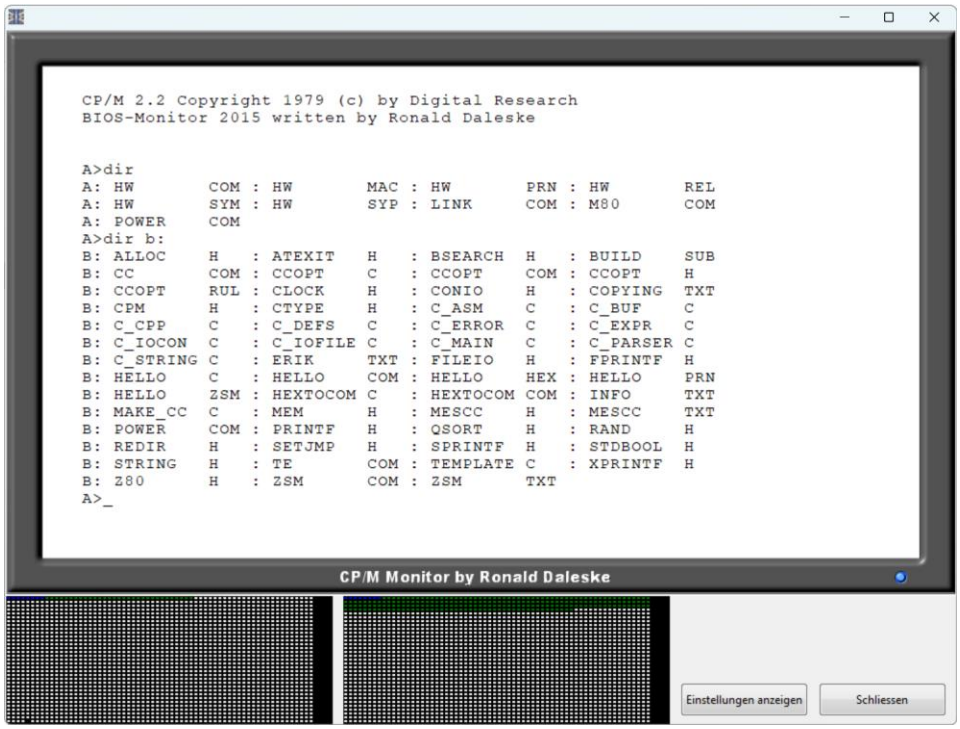


Abbildung 81 - Der CP/M-Emulator EMUZ80

Im Dateisystem stehen die beiden zu sehenden Ordner zur Verfügung, über die CP/M-Programme verwaltet werden können.

10 CP/M auf realer Hardware

Kommen wir jetzt zum Thema, bei dem es darum geht, das Betriebssystem CP/M auf einer Hardware lauffähig zu bekommen. Wie wir schon gesehen haben, gibt es über verschiedene Mikrocontroller-Boards die Möglichkeit, das Betriebssystem zu simulieren. Die angesprochenen Boards wie *Teensy*, *Arduino Due* oder das *ESP32*-Board sind hier natürlich erste Wahl, um einen geeigneten und kostengünstigen Einstieg zu finden. Später werde ich dann auf Boards eingehen, die mit einer richtigen Z80-CPU bestückt sind.

10.1 Das Arduino-Due-Board

Das Arduino Due-Board schaut aus der Nähe wie folgt aus.

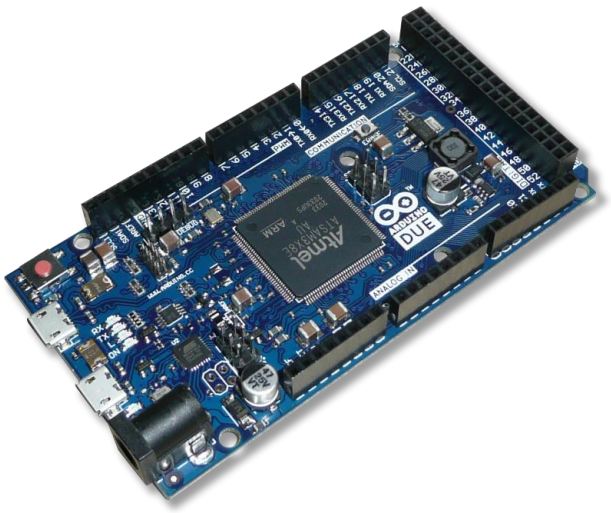


Abbildung 82 - Das Arduino-Due-Board

Da das Board von Hause aus über keinen SD-Karten-Adapter verfügt, ist es erforderlich, eine externe Lösung zu bemühen. Es ist wichtig zu erkennen, dass das Board mit einer Versorgungsspannung von 3,3V betrieben wird. Aus diesem Grund muss hinsichtlich eines SD-Karten-Adapters sichergestellt sein, dass dieser ebenfalls mit 3,3V arbeitet und keinen Level-Shifter zur Anpassung von 5V auf 3,3V besitzt.

Ein derartiger Adapter wird über die SPI-Schnittstelle des Boards angeschlossen.

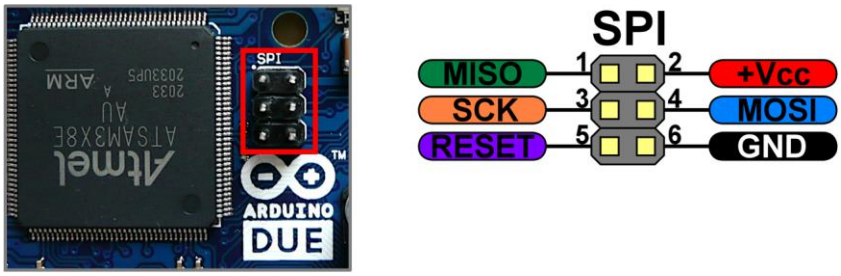


Abbildung 83 - Die SPI-Schnittstelle des Arduino-Due

Der von mir verwendete SD-Karten-Adapter schaut wie folgt aus.

Abbildung 84 - Der SD-Karten-Adapter

Er besitzt die für die SPI-Schnittstelle üblichen Pins, die auf der Rückseite der kleinen Platine beschriftet sind.

- CS (Chip Select)
- SCK (Clock)
- MOSI
- MISO
- Vcc
- GND

Der Adapter muss nun in der folgenden Weise mit dem Arduino-Due-Board verbunden werden.

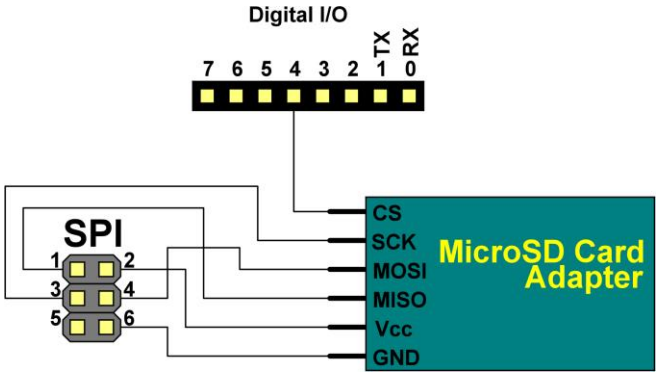


Abbildung 85 - Die elektrischen Verbindung des Adapters mit dem Arduino-Due

Im Endeffekt schaut es dann wie folgt aus.

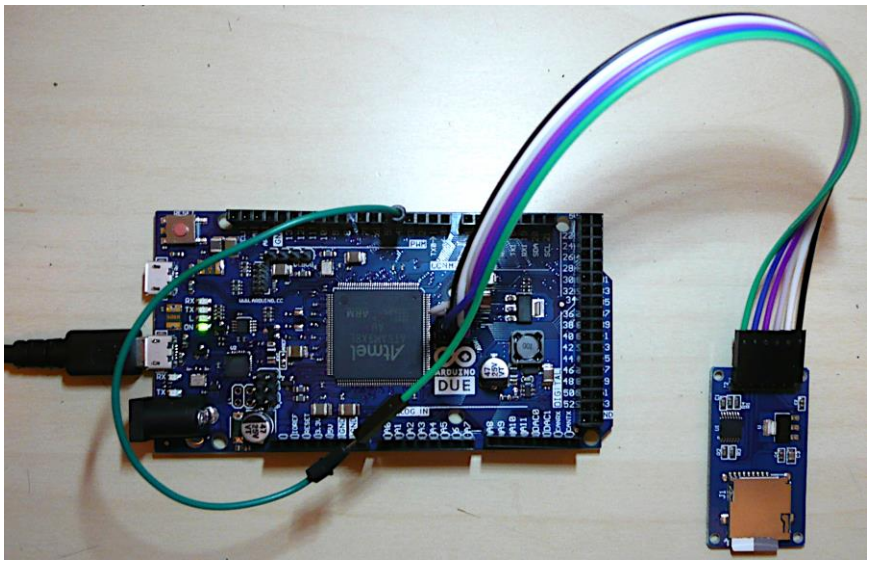


Abbildung 86 - Der Arduino-Due mit dem SD-Karten Adapter

10.1.1 Die Arduino-Software

Um das *RunCPM*-Projekt auf das Mikrocontroller-Board hochzuladen, ist die Arduino-Entwicklungsumgebung notwendig, die unter der folgenden Internetadresse zu finden ist.

https://www.arduino.cc/en/software

Um die Funktionalität der SD-Karte nutzen zu können, ist zusätzlich noch die Installation einer Library mit dem Namen SdFat von *Bill Greiman* erforderlich. Das Hinzufügen erfolgt über den Bibliotheksverwalter der Arduino Entwicklungsumgebung.

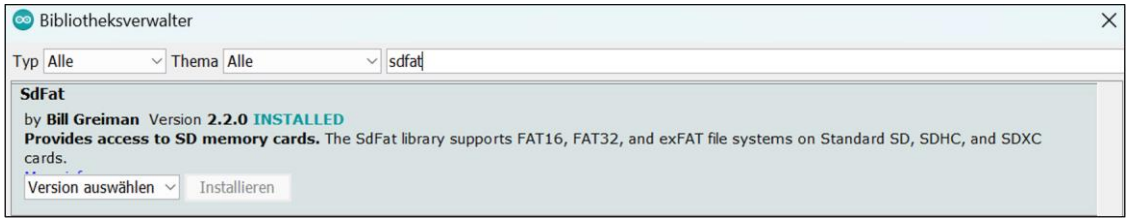


Abbildung 87 - Die benötigte SdFat-Bibliothek

Um nun das ganze Projekt beginnen zu können, ist es natürlich erforderlich, den Arduino-Quellcode (Sketch) in die Entwicklungsumgebung zu bekommen. Dazu lädt man sich am besten das gesamte Github-Repository herunter, das unter der folgenden Internetadresse zu finden ist.

https://github.com/MockbaTheBorg/RunCPM

Dazu wird einfach die grüne Code-Schaltfläche rechts oben angeklickt und dann der unterste Punkt ***Download ZIP*** ausgewählt.

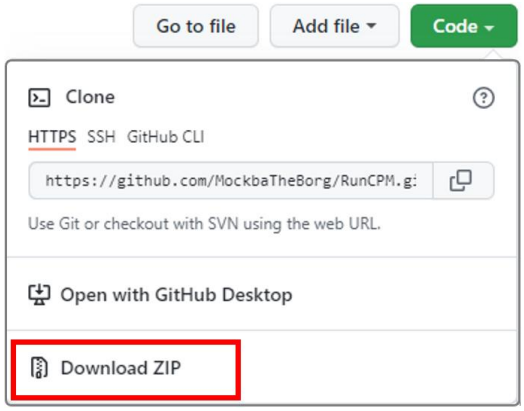


Abbildung 88 - Der Download des Repositories

Nach dem Entpacken befinden sich dort sehr viele Informationen und es sollte die Datei *readme.md* auf jeden Fall studiert werden. Für die Arduino-Entwicklungsumgebung ist das Verzeichnis ***RunCPM*** wichtig, das man sich in den Arduino-Sketch-Ordner kopiert. Nach dem Öffnen der Datei ***RunCPM.ino*** werden mehrere Dateien in die Entwicklungsumgebung geladen, was an den zahlreichen Reitern am oberen Rand zu erkennen ist.

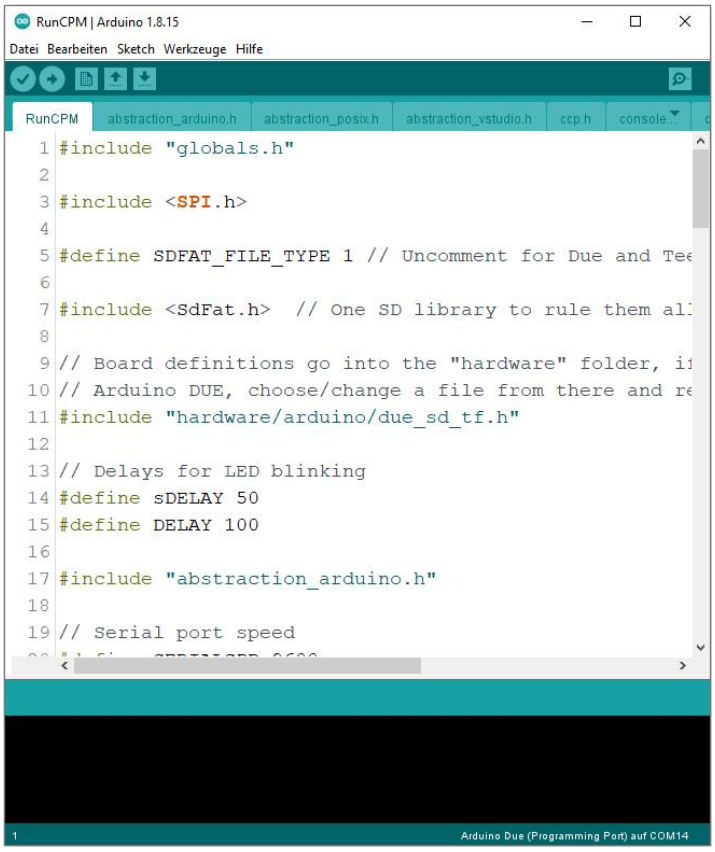


Abbildung 89 - Die Arduino-IDE mit den vielen benötigten Dateien

Ich schlage vor, an dieser Stelle noch nicht den Sketch hochzuladen, denn es ist ja noch das eigentliche CP/M-Betriebssystem erforderlich, das mit einigen Programmen und Tools auf eine SD-Karte in einer bestimmten Form abgelegt werden muss.

10.1.2 Die CP/M-Software

Damit der CP/M-Rechner auch richtig booten kann, ist natürlich das Betriebssystem erforderlich. Auf der SD-Karte müssen sich die folgenden Dateien im Root-Verzeichnis befinden.

CCP-CCPZ.60K	24.09.2016 06:59	60K-Datei
CCP-CCPZ.64K	24.09.2016 07:11	64K-Datei
CCP-DR.60K	23.09.2016 00:03	60K-Datei
CCP-DR.64K	23.09.2016 00:04	64K-Datei
CCP-Z80.60K	17.02.2021 14:35	60K-Datei
CCP-Z80.64K	17.02.2021 14:36	64K-Datei
CCP-ZCP2.60K	22.11.2018 14:28	60K-Datei
CCP-ZCP2.64K	23.09.2016 13:25	64K-Datei
CCP-ZCP3.60K	03.12.2018 22:12	60K-Datei
CCP-ZCP3.64K	03.12.2018 22:19	64K-Datei

Abbildung 90 - Die CPP-Files

Es ist zu sehen, dass hier die CCP-Binärdateien für 64K und 60K vorhanden sind. Wenn Sie eine SD-Karte verwenden, müssen sich RunCPM und seine CCPs im Stammverzeichnis der SD-Karte befinden. Die 64K-Version der CCPs stellt CP/M-Anwendungen die maximal mögliche Speichermenge zur Verfügung, aber ihre Adressierungsbereiche sind unrealistisch, wenn es um die Emulation eines echten CP/M 2.2-Computers geht. Die 60K-Version der CCPs bietet einen realistischeren Adressierungsbereich, da der CCP-Einstiegspunkt auf der gleichen Ladeadresse liegt wie auf einem physischen CP/M-Computer. Es können jedoch auch andere Speichergrößen verwendet werden, was aber die Neuerstellung der CCP-Binärdateien erforderlich macht (die Quellen befinden sich auf Diskette *A.ZIP*). Die CCP-Binärdateien sind so benannt, dass ihre Dateinamenerweiterungen mit der Speichergröße übereinstimmen, auf der sie laufen. So würde z.B. das CCP von DRI, das auf 60K Speicher läuft, CCP-DR.60K heißen. RunCPM sucht die Datei dementsprechend, je nachdem, welche Speichergröße beim Erstellen ausgewählt wurde.

RunCPM emuliert die CP/M-Disketten und -Benutzerbereiche mit Hilfe von Unterordnern unter der ausführbaren RunCPM-Datei. Um einen Ordner oder eine SD-Karte für die Ausführung von RunCPM vorzubereiten, sind folgende Schritte erforderlich

- Anlegen von mehreren Unterverzeichnissen im Root-Verzeichnis der SD-Karte mit den Namen "A", "B", "C", "D", usw. die jeweils ein physikalisches Laufwerk repräsentieren (Sowohl Ordnernamen, als auch Dateinamen sollten aus Kompatibilitätsgründen in Großbuchstaben gehalten werden!)
- Anlegen eines Unterordners unterhalb von *A* mit dem Namen "0". Er repräsentiert den Benutzerbereich 0 von Laufwerk A. Das sollte auch für alle anderen Laufwerke erfolgen.
- Wenn innerhalb von CP/M zu einem anderen Benutzerbereich gewechselt wird, werden automatisch die entsprechenden Unterordner für die Benutzerbereiche "1", "2", "3", usw. angelegt, sobald sie ausgewählt werden.
- Der Inhalt der entpackten Datei *A.ZIP* (befindet sich im heruntergeladenen Github-Reporitory unterhalb des *DISK*-Ordners) muss in den Unterordner "0" des Laufwerks A kopiert werden.
- CP/M unterstützte nur 16 Laufwerke von A: bis P:, so dass das Anlegen anderer Buchstaben oberhalb von P nicht funktioniert

Um die SD-Karte direkt mit vielen Programmen zu versehen, kann die Struktur verwendet werden, die unter dem folgenden Link zu finden ist.

https://drive.google.com/drive/folders/11WIu8rD_7pIDaET7dqTeA73CvX0jkxz2

10.1.3 Der Arduino-Sketch-Upload

Ist alles so weit vorbereitet, kann der RunCPM-Sketch auf den Arduino-Due hochgeladen werden. Nach Erfolgt zeigt sich das wie folgt im Status-Fenster der Arduino-Entwicklungsumgebung.

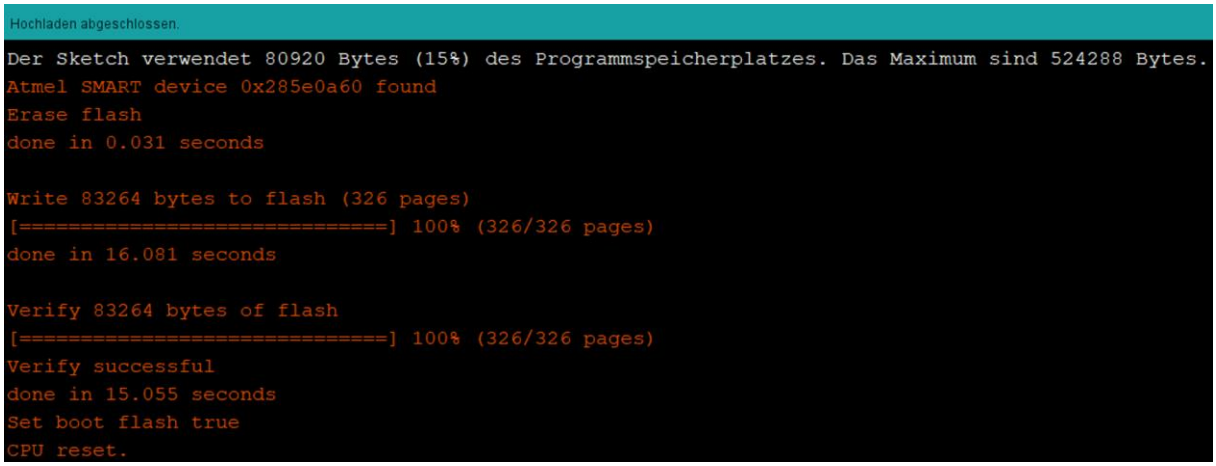


Abbildung 91 - Der RunCPM-Upload-Erfolg

Nun kann der CP/M-Rechner endlich in Erscheinung treten. Doch eine Kleinigkeit muss natürlich noch gemacht werden.

10.1.4 Der Terminal-Zugriff

Um letztendlich auf den CP/M-Rechner zugreifen zu können, ist ein Terminal-Programm erforderlich. Ich nutze dafür *Terra Term*, das kostenlos unter der folgenden Internetadresse zu finden ist.

https://tera-term.de.softonic.com/

Die folgenden Einstellungen hinsichtlich der seriellen Schnittstelle müssen vorgenommen werden.

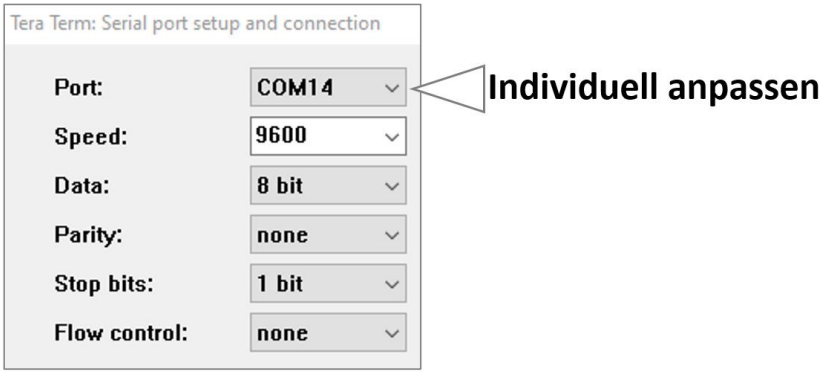


Abbildung 92 - Serial-Port-Setup

Für meine Belange habe ich noch Font-Größe und Farbe ein wenig angepasst. Nach der erfolgten Verbindungsaufnahme zeigt sich das Terminal-Fenster wie folgt.

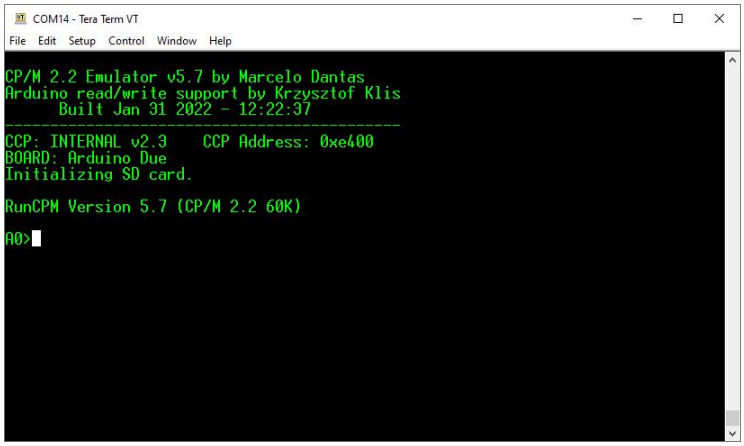


Abbildung 93 - Der CP/M-Rechner im Terminal-Fenster

10.2 Das Teensy-Board

Wer das Teensy 3.6-Board nutzt, der hat den Vorteil, dass sich schon ein Micro-SD-Adapter auf der kleinen Platine befindet. Es muss also nichts mehr mit Kabel verbunden oder gesteckt werden.

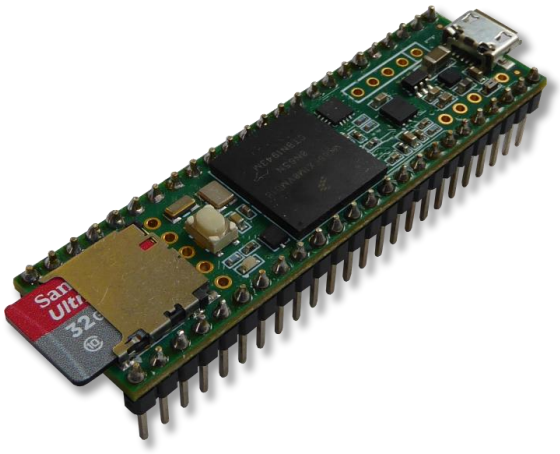


Abbildung 94 - Das Teensy 3.6-Board

Unter dem folgenden Link sind alle Informationen zu den verschiedenen Teensy-Boards zu finden.

https://www.pjrc.com/teensy/teensyduino.html

Wurde das zuvor genannte RunCPM-Arduino-Projekt geladen, muss im RunCPM-Tab die markierte Zeile mit der *#include*-Anweisung in Zeile 12 von den Kommentarzeichen befreit werden und die zuvor aktive für den Arduino-Due in Zeile 11 mit Kommentarzeichen versehen werden.

```
RunCPM | abstraction_arduino.h | abstraction_posix.h | abstraction_vstudio.h | ccp.h | console.h | cpm.h | cpu.h | disk.h | globals.h | host.h | lua.h | main.c | ram.h | resource.h
 7 #include <SdFat.h>  // One SD library to rule them all - Greinman SdFat from Library Manager
 8
 9 // Board definitions go into the "hardware" folder, if you use a board different than the
10 // Arduino DUE, choose/change a file from there and reference that file here
11 //#include "hardware/arduino/due_sd_tf.h"
12 #include "hardware/teensy/teensy36.h"
13 //#include "hardware/esp32/devkit.h"
```

Abbildung 95 - Die Konfiguration für das Teensy 3.6-Board

Hinsichtlich der verwendeten SD-Karte kann alles so bleiben, wie es schon beim Arduino-Due vorhanden war.

10.3 Das TTGO VGA32-Board

Kommen wir zu einer weiteren sehr interessanten Variante mit dem sogenannten ***TTGO VGA32-Board V1.4***, das mit einem *ESP32* versehen ist. Nähere Informationen sind auf der folgenden Internetseite des Herstellers zu finden.

http://www.lilygo.cn/prod_view.aspx?TypeId=50033&Id=1083

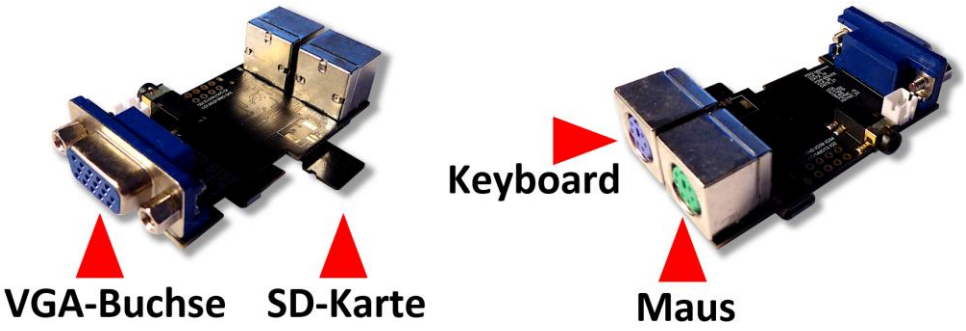


Abbildung 96 - Das TTGO-VGA32-Board

Es besitzt einen VGA-Anschluss und einer PS/2-Keyboard und PS/2-Maus-Buchse. Darüber ist es möglich, das Board ohne über den USB-Anschluss und Terminal-Programm vollkommen autark an einem Monitor zu betreiben. Der USB-Anschluss wird dabei lediglich zur Spannungsversorgung des Boards genutzt. Auf der folgenden Abbildung ist das Board am Monitor und einer PS/2-Tastatur zu sehen.

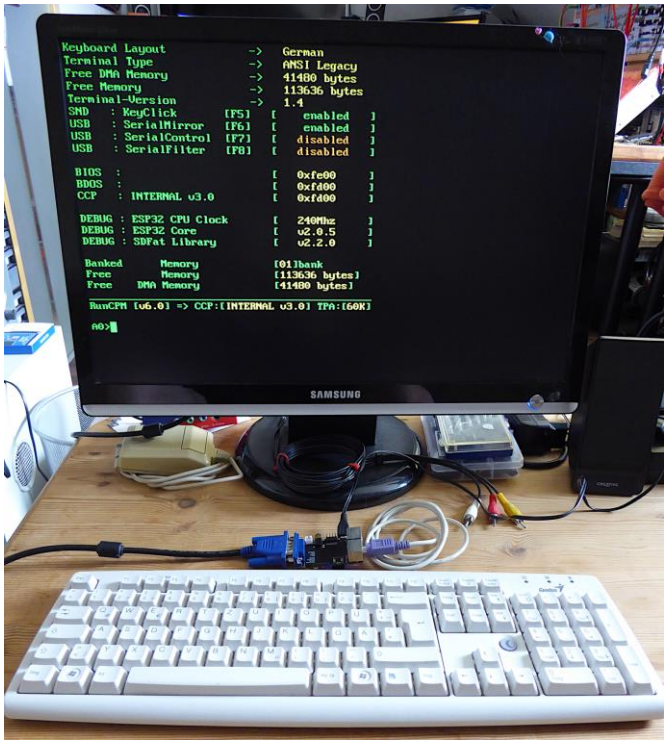


Abbildung 97 - Das TTGO-VGA32-Board am Monitor und mit PS/2-Tastatur

Über die Funktionstaste ***F12*** kann das Board hinsichtlich der Terminalfunktionen konfiguriert werden.

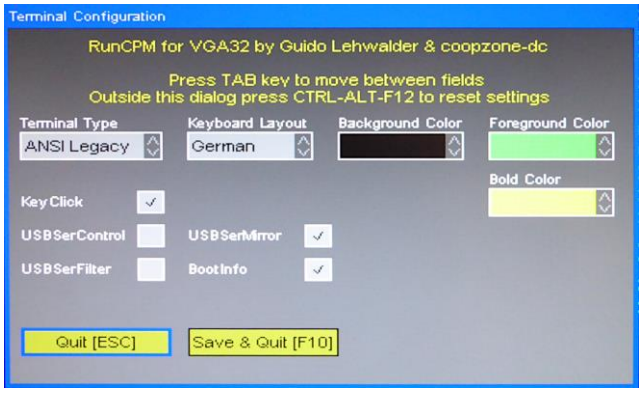


Abbildung 98 - Die Terminal-Konfiguration

Das Projekt für die Arduino-Entwicklungsumgebung ist unter der folgenden Internetadresse zu finden.

https://github.com/guidol70/RunCPM_VGA32/

Die momentan aktuellste Version ist die ***v6_0*** und es lohnt sich, dort nach eventuellen Updates zu schauen. Ich bin beim Kompilieren in ein Problem gelaufen und hätte besser die *Readme*-Datei besser gelesen. Der Fehler lautete.

Abbildung 99 - Die Fehlermeldung in der Arduino-Entwicklungsumgebung

Der entscheidende Hinweis zur Behebung des Problems lautete wie folgt.

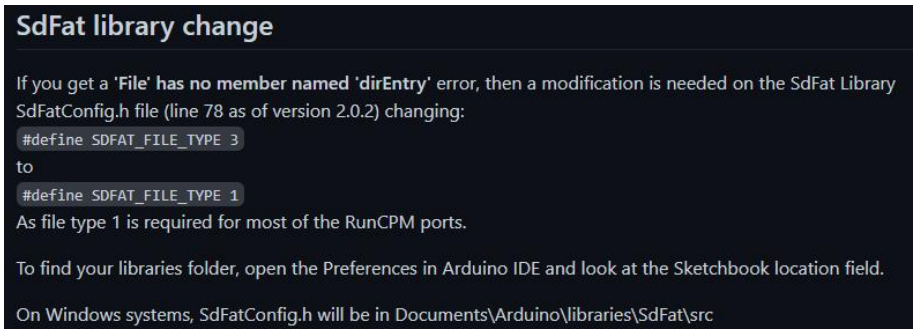


Abbildung 100 - Ein entscheidender Hinweis für die Nutzung der SdFat-Library

Nach dieser kleinen Modifikation funktionierte alles wunderbar!

10.4 Das Raspberry Pi-Pico-Board

Sehen wir uns nun das jetzt den Aufbau hinsichtlich des Raspberry Pi-Pico-Boards an. Es handelt sich um ein kostengünstiges, leistungsstarkes Mikrocontroller-Board mit flexiblen digitalen Schnittstellen. Eine Erweiterung stellt der *Raspberry Pi Pico W* dar, der ebenfalls auf dem *RP2040* Mikrocontroller-Chip basiert. Dieser wurde mit einer drahtlosen 2,4-GHz WLan-Schnittstelle versehen. Auf der folgenden Abbildung ist der Raspberry Pi-Pico zu sehen.

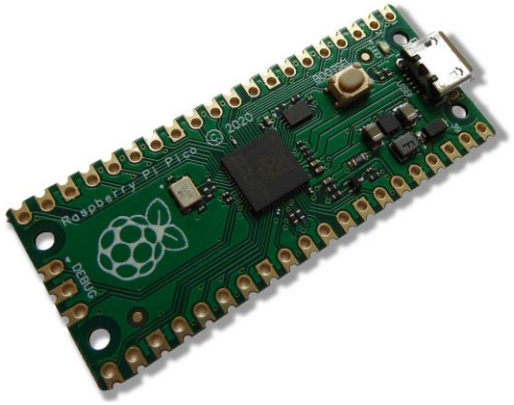


Abbildung 101 - Das Raspberry Pi-Pico-Board

Um einen SD-Karten-Adapter ohne Levelshifter anzuschließen, sehen wir uns die folgende Pinbelegung an.

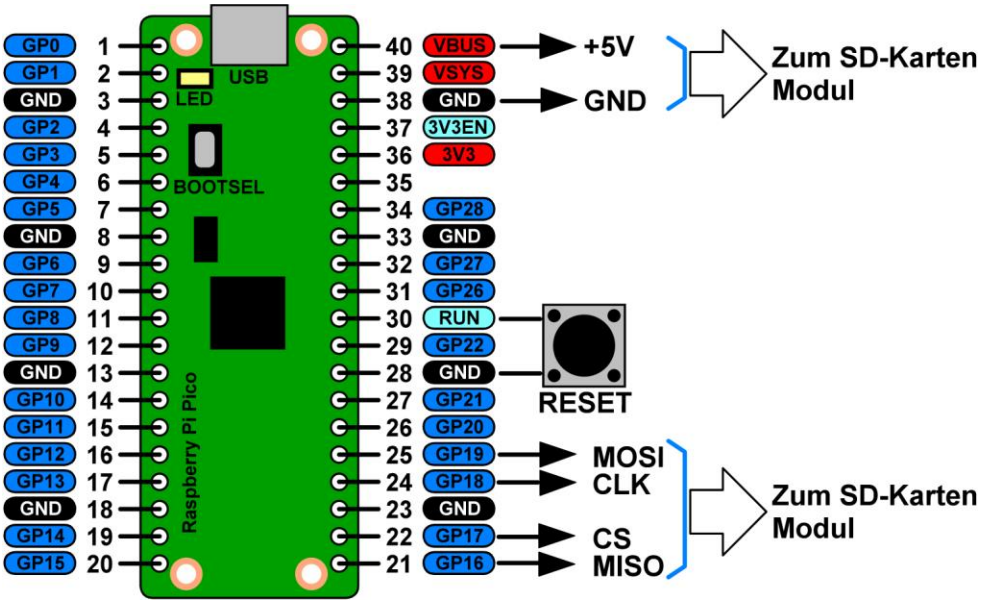


Abbildung 102 - Die Pinbelegung des Raspberry Pi-Pico-Boards

Um die Software auf das Pico-Board zu laden, muss beim verbinden des Boards mit der USB-Schnittstelle der kleine Taster mit der Beschriftung BOOTSEL gedrückt werden. Das hat zur Folge, dass sich das Board im Datei-Explorer als Laufwerk zur erkennen gibt. Bei mir ist dies das Laufwerk ***F:*** und zeigt zwei Dateien an.

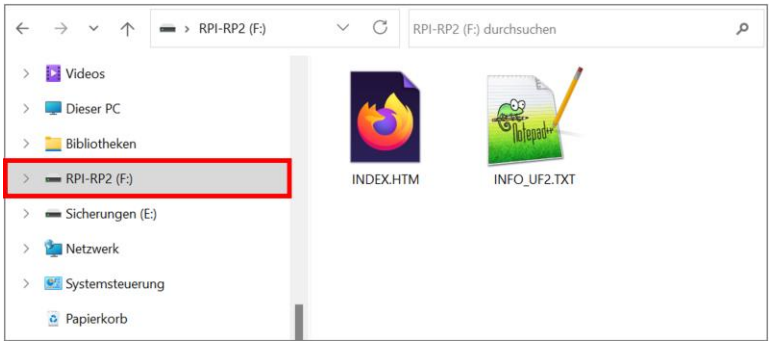


Abbildung 103 - Das Laufwerk des Raspberry Pi-Pico

Diese Dateien können wir ignorieren. Um das Board zu programmieren, muss die Projekt-Datei mit der Endung ***u2f*** einfach per *Drag&Drop* in den Ordner gezogen werden. Bei mir lautete die Datei ***RunCPM_Pico_02042022.uf2***. Nun geht es noch darum, die SD-Karte mit entsprechender Software zu versehen, die Teil des Projektes ist. Der entsprechende Ordner trägt den Namen ***SDCard_content*** und beinhaltet das Laufwerk ***A***. nach dem Einstecken der SD-Karte in das SD-Karten-Modul und der Verbindung des Pico-Boards über den USB-Anschluss mit dem Computer kann es losgehen. Der Raspberry Pi-Pico wird über ein Terminal-Programm mit der Baudrate von 9600 aufgerufen werden.

Abbildung 104 - Der Raspberry Pi-Pico meldet sich als CP/M-Rechner

Viel Spaß mit allem!

Herzlichen Gruß

Erik Bartmann