# 1. Einführung in die Mikroprozessorarchitektur

Computer sind Geräte, welche eine Liste oder auch Folge von Befehlen abarbeiten. Eine solche Folge von Befehlen heißt *Programm*. Diese Programme befinden sich im *Speicher*. Ausgeführt werden diese Programme von einem *Prozessor*. Prozessor und Speicher sind durch *Busse* miteinander verbunden.

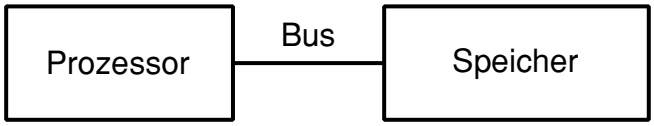


Illustration 1.1: Einfachste Darstellung eines Computers

Man kann sich den Speicher wie eine lange Liste mit Zeilennummern vorstellen. In jeder Zeile steht ein Befehl.

Veranschaulicht sieht das in etwa so aus:

| **Zeile** | **Befehl** |
|---|---|
| 1 | Gehe einkaufen! |
| 2 | Putze das Fahrrad! |
| 3 | Staubsauge das Wohnzimmer! |
| 4 | Lese die GAG-News! |

Tabelle 1.1

Grundsätzlich funktioniert das so: Der Prozessor nennt dem Speicher die Zeilennummer, deren Befehl er wissen möchte. Der Speicher nennt dem Prozessor diesen Befehl, und der Prozessor führt ihn aus.

Statt Zeile sagt man beim Computer jedoch *Adresse*. Technisch umgesetzt wurde dieser Ablauf mit Hilfe eines *Programmzählers* (englisch *programm counter*, kurz *PC*), einem *Adress-* und einem *Datenbus*.

Im Programmzähler steht die Adresse, welche der Prozessor wissen möchte. Der Programmzähler fängt nach dem Anschalten bei Null an und wird nach jeder Ausführung eines Befehls automatisch erhöht. Der Programmzähler ist über den *Adressbus* mit dem Speicher verbunden. Der Speicher gibt den Inhalt der Adresse, welcher im Programmzähler steht und über den Adressbus an den Speicher gemeldet wird, auf den Datenbus aus. Der Datenbus ist ebenfalls mit dem Prozessor verbunden. Der Prozessor "sieht" so, was an der Adresse steht, dessen Adresse er über den Adressbus an den Speicher meldet. Dies ist eine ganz starke Vereinfachung des tatsächlichen Vorgangs. Aber im Prinzip funktioniert es so.

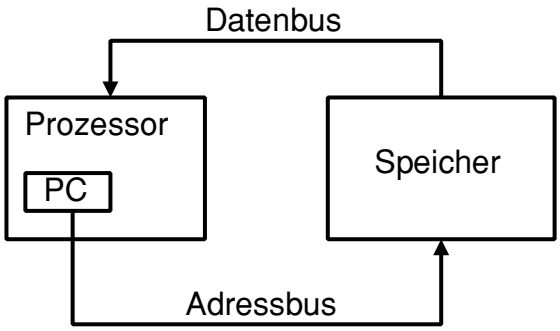


Illustration 1.2: Computer mit Programmzähler (PC), Adress- und Datenbus

Während der Mensch viele verschiedene Zeichen kennt, z. B. die Ziffern 0 bis 9 oder die 26 Buchstaben des Alphabets, arbeitet der Computer für gewöhnlich nur mit zwei verschiedenen "Zeichen". Diese können z. B. mit verschiedenen Spannungswerten verwirklicht werden (es liegt Spannung an oder nicht). Man kann sich das wie einen Schalter vorstellen, der zwei verschiedene Positionen einnehmen kann: an oder aus. Man schreibt dafür auch 0 oder 1. Jeder Schalter, z. B. ein Lichtschalter, ist damit auch automatisch ein Speicher. Er kann sich nämlich die Stellung "merken".

Illustration 1.3: Schalter offen (oben) und Schalter geschlossen (unten). Für einen offenen Schalter schreibt man auch 0 und für einen geschlossenen 1.

Würde nun jede Adresse des Speichers nur über einen einzigen Schalter verfügen, also nur die Zustände an oder aus annehmen können, könnte man nicht allzuviel machen. Denn dann könnte der Computer im besten Fall nur über zwei verschiedene Befehle verfügen. Er könnte kaum etwas unterscheiden.

Das gleiche betrifft auch den Programmzähler (PC): würde dieser nur über einen einzigen Schalter verfügen, könnte er nur die zwei Adressen 0 und 1 vom Speicher unterscheiden. Also nicht weiter zählen als bis eins. Er würde abwechselnd hin- und herschalten (an und aus, bzw. 0 und 1). Das wäre alles.

Genauso wie man die verschiedenen Ziffern 0, 1, 2, 3, 4, 5, 6, 7, 8 9 zu ganzen Zahlen miteinander kombiniert (z. B. 10, 11, 12 usw.), arbeitet man hier deshalb mit einer Vielzahl von Schaltern, mit einer Gruppe von Schaltern an jeder Adresse.

Illustration 1.4: Gruppe von acht Schaltern. Für das hier dargestellte Beispiel schreibt man auch 0100 1101 (0: offen, 1: geschlossen). Das Leerzeichen dient nur zur besseren Lesbarkeit.

Die aktuelle Adresse steht im Programmzähler. Diese Adresse wird über den Adressbus dem Speicher mitgeteilt. Die Stellungen all der Schalter an der betreffenden Adresse im Speicher werden dann vom Speicher über den Datenbus dem Prozessor rückgemeldet. Ein Bus besteht hier also aus einer Vielzahl von Leitungen und dient einem gemeinsamen Zweck, einem gleichzeitigen Vorgang.

Wieviele verschiedene Kombinationen können nun eine bestimmte Anzahl von Schalter haben? Bei einem Schalter wissen wir: er kann die Stellungen an oder aus (wir schreiben hierfür auch 0 oder 1) haben. Kombiniert man mehrere Schalter, schreibt man die möglichen Kombinationen am besten unter Verwendung eines Übertrages einfach hin. Ein Übertrag bedeutet, daß man einen weiteren Schalter hinzufügt, wenn man alle möglichen Kombinationen der bisherigen Schalter durch hat. Man merkt sich so diesen Übertrag. Dann fängt man bei den hinteren Schaltern wieder von vorne an. Hier ein Beispiel mit vier Schaltern:

| | | | |
|---|---|---|---|
| 0000 | | 0 | |
| 0001 | | 1 | |
| 0010 | (1. Übertrag) | 2 | |
| 0011 | | 3 | |
| 0100 | (2. Übertrag) | 4 | |
| 0101 | | 5 | |
| 0110 | (3. Übertrag) | 6 | |
| 0111 | | 7 | |
| 1000 | (4. Übertrag) | 8 | |
| 1001 | | 9 | |
| 1010 | (5. Übertrag) | 10 | (1. Übertrag) |
| 1011 | | 11 | |
| 1100 | (6. Übertrag) | 12 | |
| 1101 | | 13 | |
| 1110 | (7. Übertrag) | 14 | |
| 1111 | | 15 | |

Tabelle 1.2

Man hat eine gewisse Anzahl von verschiedenen Zeichen, die man durchzählt. Sind alle Zeichen durch, fügt man eine Stelle hinzu (Übertrag) und fängt wieder von vorne an. Das funktioniert ungeachtet der Anzahl von Zeichen immer gleich. Hat man weniger Zeichen zur Verfügung, findet ein Übertrag früher statt. Der Übertrag steht normalerweise immer davor. Die linke Stelle ist damit die Höchstwertige, die rechte die niedrigwertigste.

Wir sehen also: Mit einem Schalter sind zwei verschiedene Kombinationen möglich. Mit zwei Schaltern vier. Mit drei Schaltern acht. Mit vier Schaltern sind es bereits sechszehn. Die Anzahl der möglichen Kombinationen ist damit $2^n$ ($2^n$ bedeutet, die Zahl 2 wird n mal mit sich selbst malgenommen, also z. B. für $n = 3$: $2^3 = 2 * 2 * 2 = 8$ ). Mit acht Schaltern hat man bereits $2^8 = 256$ mögliche Kombinationen. Auf diese Anzahl hat man sich anfangs für sehr viele Computer geeinigt wie für den Commodore 64 oder Schneider CPC.

Weil viele Schalter schnell sehr unübersichtlich werden, kann man diese anders darstellen. In Tabelle 1.2 sehen wir rechts die Folge im Zehnersystem. Dieses System verwendet zehn verschiedene Zeichen (0, 1, 2, 3, 4, 5, 6, 7, 8 und 9) und lernen wir in der Schule. Wir können statt 1111, wie in der ersten Spalte angegeben, also auch 15 schreiben (das hat dann die Bedeutung: vier Schalter sind gesetzt). Es gibt hier eine *eindeutige* Zuordnung.

Der Mensch ist in der Regel gewohnt, mit dem Zehnersystem zu arbeiten. Doch das muß nicht sein. Man kann mit jedem beliebigen anderen Zahlensystem rechnen.

Weil es sich besser aufgeht, hat man dem Zehnersystem die sechs weiteren Zeichen A, B, C, D, E und F hinzugefügt. Es handelt sich dabei um das Hexdezimal- oder Sechszehnersystem. Ergänzen wir die Tabelle 1.2 damit und schreiben wir das ganze noch einmal hin:

| | | Dezimalsystem (Zehnersystem) | | Hexdezimalsystem (Sechszehnersystem) | |
|---|---|---|---|---|---|
| 0 0000 | | 0 | | 0 | |
| 0 0001 | | 1 | | 1 | |
| 0 0010 | (1. Übertrag) | 2 | | 2 | |
| 0 0011 | | 3 | | 3 | |
| 0 0100 | (2. Übertrag) | 4 | | 4 | |
| 0 0101 | | 5 | | 5 | |
| 0 0110 | (3. Übertrag) | 6 | | 6 | |
| 0 0111 | | 7 | | 7 | |
| 0 1000 | (4. Übertrag) | 8 | | 8 | |
| 0 1001 | | 9 | | 9 | |
| 0 1010 | (5. Übertrag) | 10 | (1. Übertrag) | A | |
| 0 1011 | | 11 | | B | |
| 0 1100 | (6. Übertrag) | 12 | | C | |
| 0 1101 | | 13 | | D | |
| 0 1110 | (7. Übertrag) | 14 | | E | |
| 0 1111 | | 15 | | F | |
| 1 0000 | (8. Übertrag) | 16 | | 10 | (1. Übertrag) |

Tabelle 1.3

Wir haben dem ganzen noch eine weitere Zeile hinzugefügt. Wie man in Tabelle 1.3 sieht, findet im Hexdezimalsystem der erste Übertrag erst an 17-ter Stelle oder Zeile statt. Damit entsprechen vier Schalter mit den möglichen Kombinationen an (0) und aus (1) immer *einer* Stelle dem System in der dritten Spalte. Wenn man sechszehn verschiedene Zeichen verwendet, läßt sich mit nur einem dieser Zeichen immer eindeutig eine mögliche Kombination von vier Schaltern darstellen. Das ist von Vorteil und geht mit dem Zehnersystem *so* nicht.

Natürlich lassen sich diese Systeme leicht verwechseln, da man ja mitunter die gleichen Zeichen

hinschreibt. Ohne weitere Angabe ist unklar, was 10 sein soll. Man muß deshalb immer angeben,was gemeint ist!

In der ersten Spalte von Tabelle 1.3 verwenden wir *zwei* verschiedene Zeichen. Deshalb heißt dieses System Binärsystem (lat. aus zwei Einheiten bestehend).

In der zweiten Spalte von Tabelle 1.3 verwenden wir *zehn* verschiedene Zeichen. Deshalb heißt dieses System Dezimal- oder Zehnersystem. Zur Markierung verwendet man oft ein # vor die Zahl, also etwa #43.

In der dritten Spalte von Tabelle 1.3 verwenden wir *sechszehn* verschiedene Zeichen. Deshalb heißt dieses System Sechszehner oder Hexdezimalsystem. Zur Markierung schreibt man in der Welt von RISC OS ein & vor die Zahl. Also z. B. &1F. In der Welt von Unix schreibt man aber ein 0x, also z. B. 0x1f.

Statt von Schaltern spricht man in der Fachsprache jedoch von *Bits* (engl. *binary digit*). Spricht man von Bitbreite, ist damit gemeint, mit wievielen Bits das System gleichzeitig arbeitet. Typisch sind 8 Bit, 16 Bit, 32 Bit oder mittlweile 64 Bit und mehr.

Ein *Byte* sind acht Bit. Viele frühe Computer wie der Commodore 64 oder Amstrad CPC arbeiten mit einem Byte bzw. acht Bit. Dazu zählt aber auch der relativ neue Mega65 von Trentz Elektronik, welchen man durchaus als Nachfolger des nie auf den Markt gekommenen Commodore 65 sehen kann. Der Acorn Archimedes arbeitet mit 32 Bit oder 4 Byte. Diesen 32 Bits gibt man auch die Einheit *Word* (englisch für Wort). Ein Word sind also 4 Byte oder 32 Bit.

32 Bit = 4 Byte = 1 Word

Dies gilt jedoch nicht immer. Auf anderen Systemen kann ein Word auch 2 Bytes oder 16 Bit umfassen! Bei der ARM (Acorn Archimedes) sind 16 Bit jedoch wieder ein *halbes* Word.

$2^{10}$ *Bytes* = 1024 *Bytes* entsprechen einem Kilobyte [Kb]. $2^{10}$ *Kilobytes* = 1024 *Kilobytes* entsprechen einem Megabyte [MB]. $2^{10}$ *Megabytes* = 1024 *Megabytes* entsprechen einem Gigabyte. $2^{10}$ *Gigabytes* = 1024 *Gigabytes* entsprechen einem Terrabyte [TB]. Man sieht also, daß die Einheiten *nicht* um den Faktor Tausend, sondern um den Faktor Tausendvierundzwanzig steigen.

Nun enthält der Speicher nicht nur *Befehle* für den Prozessor, sondern auch *Daten*. Und der Prozessor kann den Inhalt einer Adresse nicht nur *lesen*, sondern auch *schreiben*. Mit Schreiben ist gemeint, daß er die Schalter umstellen, die Bits ändern kann. Über einen dritten, nämlich dem *Steuerbus*, teilt der Prozessor dem Speicher mit, ob er die Adresse lesen oder schreiben, also den Inhalt ändern möchte.

Ob der Inhalt einer Adresse als Befehl oder als Daten verstanden werden muß, hängt von der Logik der Befehle und vom Programm ab. Hierbei können Fehler passieren. Diese führen dazu,

daß ein Programm nicht richtig, nicht wie gedacht funktioniert. Denn der Prozessor kann das Programm, welches er gerade abarbeitet, auch überschreiben und damit die Befehle verändern oder löschen und damit die Logik zerstören.

Über den Speicher kommuniziert der Prozessor aber auch mit anderer Elektronik. Man kann sich das so vorstellen: Wird auf der Tastatur[1] eine Taste gedrückt, ändert sich an einer ganz bestimmten Adresse des Speichers der Inhalt. Von der Tastatur wird an dieser Adresse ein ganz bestimmtes Bitmuster, ein ganz bestimmter Wert gesetzt, welcher eindeutig einer Taste zugeordnet werden kann. Die Schalter werden nach einer Tabelle der gedrückten Taste entsprechend umgelegt. Der Prozessor kann diesen Wert an dieser Adresse auslesen und weiß damit, welche Taste gerade eben gedrückt worden ist. Der Wert an dieser Adresse darf in diesem Fall vom Prozessor eben nicht als Befehl verstanden werden!

Auf der Tastatur findet man die 26 Buchstaben des Alphabets, die Ziffern 0 bis 9 sowie diverse Sonderzeichen. Die Zuordnung von Zeichen und Wert hat man in der ASCII[2]-Tabelle festgelegt. Der ursprüngliche ASCII-Code ist sieben Bit ›lang‹, das heißt er besteht aus einer Folge von sieben Bit. Ein um Sonderzeichen wie den deutschen Umlauten oder dem scharfen S erweiteter ASCII-Code ist acht Bit lang. Dies dürfte der Grund für die gleichzeitige Verarbeitung von acht Bits früherer Maschinen sein oder warum man sich auf acht Bit festgelegt hatte. Sieben Bits wären auch unpraktisch gewesen wegen der Umrechnung der Zahlensysteme. Sieben ist eben kein Vielfaches von zwei. Und mit weniger Bits hätte man einfach nicht genug Zeichen von der Tastatur abbilden, unterscheiden können.

Das Gesagte gilt auch für andere Geräte wie dem Monitor, dem Lautsprecher oder irgendwelche Ein- oder Ausgänge. So kann der Prozessor an einer ganz bestimmten Adresse einen ganz bestimmten Schalter umlegen und so einen Ausgang plötzlich auf Spannung umschalten. In diesem Fall wirkt so ein Bit tatsächlich wie ein Schalter.

Es ist natürlich eine ganz blöde Idee, an einer solchen Stelle, also an einer solchen Adresse Information oder einen Befehl hinterlegen zu wollen. Information oder Befehl wären schließlich weg, sobald eine Taste gedrückt würde. Oder man bekommt an einem Ausgang etwas, was man gar nicht haben wollte! Deshalb ist es so wichtig, den Speicher des Systems, das man programmieren will, also die Hardware ganz genau zu kennen!

Bei Mikrokontrollern ist es oft so, daß dort nur Befehle ablaufen, welche man selbst für dieses System eingegeben hat. Man muß dort also keine Rücksicht auf andere schon bereits im Speicher vorhandenen Programme oder Befehle nehmen. Das macht es etwas leichter. Allerdings heißt das auch, daß sich solche Mikrokontroller nur von anderen Computern aus programmieren lassen. Sie selbst sind ohne Programm ja nicht arbeitsfähig. Ohne Programm können sie nicht programmiert werden.

Die Computer mit Tastatur und Bildschirm sind üblicherweise kurz nach dem Einschalten arbeitsfähig. Das heißt, man kann irgendwas mit ihnen machen. Man kann auf der Tastatur Tasten drücken und sieht irgendwas auf dem Bildschirm. Damit das so funktioniert, müssen diese Computer nach dem Einschalten bereits ein oder mehrere Programme gestartet haben.

---

[1] Die Tastatur dürfte so ziemlich das erste Eingabegerät gewesen sein.
[2] ASCII: American Standard Code for Information Interchange (1968)

Diese Programme befinden sich dann bereits im Speicher. Es handelt sich dabei meist um das Betriebssystem. Es ist auch eine schlechte Idee, diese Programme im Speicher zu überschreiben. Denn dann funktioniert ja irgendwann der Computer nicht mehr.

Man muß als Programmierer neben der Speicherbelegung durch die Hardware auch noch die Speicherbelegung durch das Betriebssystem kennen und wissen, wie man seine Programme so für das Betriebssystem gestaltet, daß es sich reibungslos in das System einfügt. Allerdings kann man von seinem Programm aus dann auch auf schon vorhandene Folgen von Befehlen (*Betriebssystemroutinen*) des Betriebssystems zurückgreifen.

Wenn wir bisher vom Speicher gesprochen haben, so war immer der *Hauptspeicher* des Computers, (*engl. random access memory*, Kurzbezeichnung *RAM*), gemeint. Dieser ist direkt mit dem Prozessor verbunden. Dieser Speicher behält seinen Inhalt in der Regel nur, wenn das System angeschaltet ist und unter Spannung steht.

Es gibt jedoch noch viele weitere Arten von Speichern beim Computer. Weil der Hauptspeicher in der Regel erst nach dem Einschalten aktiv wird, enthält er zu diesem Zeitpunkt freilich noch kein Programm. Er kann Programme auch nicht behalten, wenn er abgeschaltet wird. Diese Programme werden dann gelöscht.

Der Prozessor braucht aber ein Programm, damit er arbeiten kann. Dieses Startprogramm steht in der Regel in einem Festwertspeicher geschrieben (*engl. read only memory*, Kurzbezeichnung *ROM*), welches seinen Inhalt auch dann bewahrt, wenn der Computer abgeschaltet ist. Beim Einschalten wird der Inhalt von diesem ROM ins RAM eingeblendet. Damit bekommt der Prozessor ein Programm zur Verfügung, das er abarbeiten kann. Dieses Programm erst macht das System lauffähig.

Dieses Startprogramm kann dann von anderen Speichern wie z. B. einer Festplatte, weitere Daten und Programme in den Hauptspeicher nachladen lassen. Diesen Vorgang nennt man auch *booten*.

Auch der Prozessor selbst verfügt über eigene, sehr kleine Speicher. Diese heißen *Register* und können meist nur sehr wenige Daten aufnehmen. Ein Register entspricht von der Struktur her ungefähr dem Inhalt einer Zeile oder Adresse im Speicher (siehe auch Illustration 1.4). Dort wird Information hinterlegt. Erst damit arbeitet und rechnet der Prozessor. Der Programmzähler ist ein spezielles Register. In diesem wird mit jedem Takt des Systems der Inhalt um eins erhöht, also um eins weitergezählt oder die Schalter bzw. Bits entsprechend umgeschaltet wie in Tabelle 1.3 aufgeführt.

Man kann auch den Inhalt in diesem Register namens Programmzähler verändern. Dafür gibt es einen Befehl. Der Prozessor holt sich den nächsten Befehl dann vom Speicher, dessen Addresse im Programmzähler steht. Damit können Sprünge im Speicher oder im Programm realisiert werden.

# 2. Werkzeuge unter RISC OS

## 2.1 Dateityp Absolute in Editoren wie !StrongED und !Zap

Die ARM, das ist ein bestimmter Prozessor, eine CPU. RISC OS ist ein Betriebssystem, welches auf diesem Prozessor läuft.

Dateien, welche ausführbare Maschinenprogramme enthalten, haben unter RISC OS den Dateityp *Absolute*. Diese Programme liegen im Befehlsatz der ARM vor. Die ARM versteht diese Befehle *direkt*.

In dem Programmverzeichnis !StrongED findet man eine Datei namens !RunImage. Diese hat den Dateityp *Absolute* und enthält Maschinencode. Das Programmverzeichnis läßt sich öffnen, indem man eine der Umschalttasten gedrückt hält und gleichzeitig einen Doppelklick mit der Maus darauf anwendet.

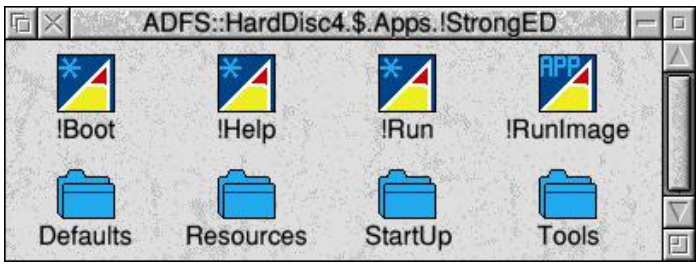


Illustration 2.1.1: Die Datei mit dem Namen !RunImage hat hier den Dateityp *Absolute*.

Diese Verzeichnisse und Dateien liegen auf einem *Festwertspeicher* wie einer Festplatte vor. Diese sind *nicht* gleichzusetzen mit dem Speicher, mit welchem der Prozessor arbeitet. Es handelt sich um eine andere Art von Speicher! Unter RISC OS sind diese links unten auf der Symbolleiste zu finden.

Den Inhalt einer solchen Datei kann man sich sinnvollerweise mit einem der mächtigen Editoren !StrongED oder !Zap anzeigen lassen. In !StrongED schaltet man am besten im Dump-Modus auf die Darstellung ASM um. ASM steht für Assembler.

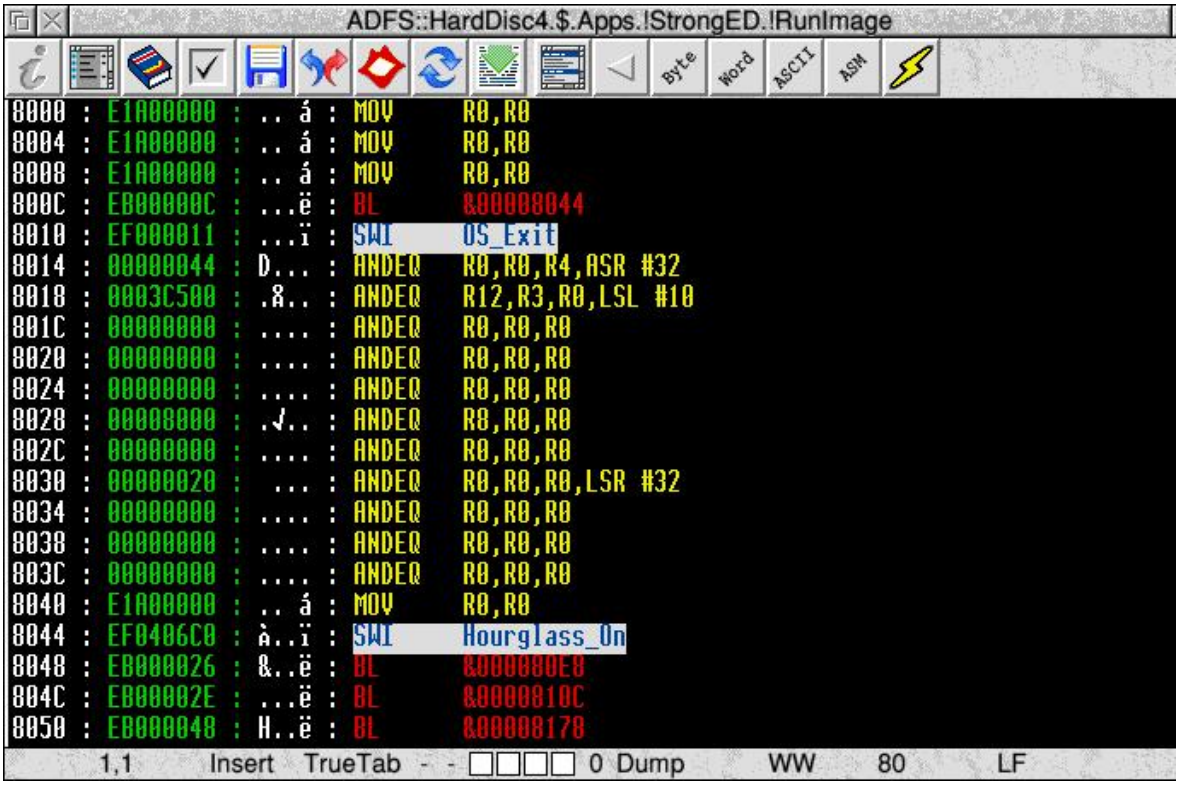


Bild 2.1.2: Die Datei !RunImage aus dem Verzeichnis !StrongED in !StrongED angezeigt.

In der ersten Spalte sieht man weiß die Speicheradressen. In der zweiten Spalte sieht man grün den Maschinencode in hexdezimaler Form. In der dritten Spalte, wieder weiß, sieht man die Werte aus Spalte zwei als ASCII-Zeichen gedeutet und dargestellt. Bei den letzten beiden Spalten handelt es sich noch einmal um eine andere Darstellung der Werte aus Spalte zwei. Hier wird der Speicherinhalt als *Mnemonics* dargestellt. Mnemonics sind nur eine andere Darstellung von Maschinencode. Und zwar in einer Art und Weise, die der Mensch besser lesen kann. Es handelt sich um *Assemblerbefehle*. Diese hat uns !StrongED aus den Werten in Spalte zwei errechnet. Spalte zwei und vier bedeuten genau dassselbe.

Üblicherweise werden beim Programmieren diese Mnemonics von einem Assembler, das ist eine ganz bestimmte Art von Programm, in Maschinencode umgerechnet. Im vorliegenden Fall wurde jedoch *rückwärts* gerechnet, wurden also aus dem Maschinencode die Mnemonics bestimmt. Die Mnemonics versteht der Prozessor nicht direkt.

Startet man die Datei namens !RunImage mit dem Dateityp *Absolute* durch einen Doppelklick mit der Maus, so lädt RISC OS diese Datei und hinterlegt sie im Speicher ab der Adresse mit dem hexdezimalen Wert `&8000`. Anschließend wird der Programmzähler des Prozessors auf diese Adresse gesetzt. Der Prozessor holt sich jetzt von dort den ersten Befehl und arbeitet das gerade eben in den Arbeitsspeicher geladene und gestartete Programm ab.

Wenn man sich jetzt viele verschiedene solcher Dateien mit dem Dateityp Absolute ansieht, wird man feststellen, daß *jedes* dieser Programme bei der hexdezimalen Adresse `&8000`

beginnt.

Das ist insofern verwunderlich, weil unter RISC OS mehrere Programme gleichzeitig laufen können. Denn das hieße ja, daß jedes Programm im Speicher das andere überschreiben würde.

Daß dem nicht so ist, nicht sein kann, sollte klar sein. In früherer Zeit befand sich zwischen dem Prozessor und dem Speicher noch ein weiterer Chip namens MEMC, welcher den Speicher verwaltete. Dieser Chip wies dem Programm dann den tatsächlichen Speicherort zu. Die verschiedenen Programme können über eine Tabelle eingeblendet werden. Für den Prozessor sieht es immer so aus, wie wenn sich nur ein einziges Programm im Speicher befände. Inzwischen wurde diese Funktion vom MEMC in den Prozessor selbst integriert.

Die tatsächlichen Speicheradressen befinden sich also woanders, werden aber für den Prozessor ab der Adresse `&8000` eingeblendet.

Wir können das überprüfen, indem wir zwei Aufgabenfenster (engl. *task windows*) starten. Das geht über das Pop-up-Menü des Task-Symbols ganz rechts auf der Symbolleiste (*engl. icon bar*) oder über die Tastenkombination STRG (*engl. CTRL*) + F12. In beiden Fenstern sollte nun der Stern der Befehlszeile zu sehen sein.

Im ersten Aufgabenfenster tippen wir einfach `memory &8000` ein. Den Befehl müssen wir, wie jeden Befehl, mit einem Druck auf die Eingabetaste bestätigen.

Im zweiten Aufgabenfenster tippen wir `BASIC` ein. Anschließend geben wir `*memory &8000` ein. Der Stern muß hier mit eingegeben werden!

In beiden Fenstern wird uns jetzt der Speicherbereich ab der Adresse `&8000` angezeigt. Wir werden feststellen, daß uns in beiden Fenstern etwas anderes angezeigt wird. In dem Aufgabenfenster, wo wir BBC BASIC gestartet haben, sehen wir genau dieses Pogramm im Speicher liegen. Wir sehen links die Werte in hexdezimaler Darstellung. Rechts sehen wir das entsprechende ASCII-Zeichen.

So ein Programm im Maschinencode oder in Maschinensprache läßt sich mittels !Zap oder !StrongED besser anzeigen und analysieren. Es startet also immer mit der Adresse &8000. Mittels dem Menü `Create -> Dump` von !StrongEd auf der Symbolleiste können wir uns unter anderem so auch Inhalte des Arbeitsspeichers (*engl. read only memory* oder *RAM*) anzeigen lassen.

In Abbildung 2.1.2 fällt auf, daß jeder Befehl mit jeder *vierten* Adresse des Speichers anfängt. Das liegt daran, daß der Speicher *byte*adressiert ist (ein Byte entspricht acht Bits), ein Befehl jedoch vier Bytes (oder 32 Bits) umfaßt.

Früher konnte der Programmzähler nur auf jede vierte Adresse gesetzt werden, da Bit 0 und 1 immer 0 waren. Die letzten zwei Bits konnten damals nicht überschrieben werden.

Seit der vierten Version der ARM ist das jedoch anders. Hier können Bit 0 und 1 ebenfalls gesetzt werden. Man sollte das tunlichst vermeiden! Es ist sonst unvorhersehbar, was der Prozessor machen wird. Der Wert im Programmzähler sollte also normalerweise immer durch vier (ohne Rest) teilbar sein.

Bevor der Prozessor das Programm anspringt, schreibt er noch den aktuellen Wert des Programmzählers in Register 14. Will man nun sein Programm beenden, muß man nur den Programmzähler auf die Adresse setzen, welche im Register 14 hinterlegt wurde. Damit sind wir auch schon beim ersten notwendigen Befehl: `&E1A0 F00E` oder als Mnemonics geschrieben: `MOV PC, R14`. Wichtig dabei ist natürlich, daß wir den Wert im Register 14 nicht verändert haben, während unser Programm abläuft. Oder daß wir den ursprünglichen Wert im Register 14 wieder dorthin geschrieben haben, bevor dieser Befehl, `MOV PC, R14`, zum Einsatz kommt.

Wir können nun in !StrongED ein neues (leeres) Dokument erzeugen, indem wir auf das Symbold von !StrongED auf der Symbolleiste klicken. Dieses leere Dokument speichern wir unter einem Dateinamen ab. Dabei geben wir ihm gleichzeitig den Dateityp *Absolute*. Alternativ können wir den Dateityp auch später über das Dateisystem ändern. Auf jeden Fall sollten wir dann die Datei schließen und wieder neu laden, indem wir sie auf das Symbol von !StrongED auf der Symbolleiste fallen lassen. Alternativ können wir sie laden, indem wir einen Doppelklick mit der Maus bei gleichzeitig gedrückter Umschalttaste anwenden.

Jetzt brauchen wir den BaseMode *Dump*. Wir finden ihn im Menü von !StrongED (zum Öffnen des Menüs mittlere Maustaste oder Rollrad drücken) unter dem Eintrag `BaseMode -> Change mode -> Dump`. Dann klicken wir in der Werkzeugleiste auf `ASM`.

Nun geben wir dort in der zweiten Spalte den Befehl

```
E1A0 F00E
```

ein und speichern das ganze. (Das Leerzeichen dient nur zur besseren Lesbarkeit und darf nicht mit eingegeben werden.) Die Eingabe mag am Anfang etwas fremd auf einen wirken. Statt die Schriftmarke von links nach rechts zu verschieben und die einzelnen Zeichen nacheinander anzuhängen, werden die Zeichen von rechts nach links geschoben.

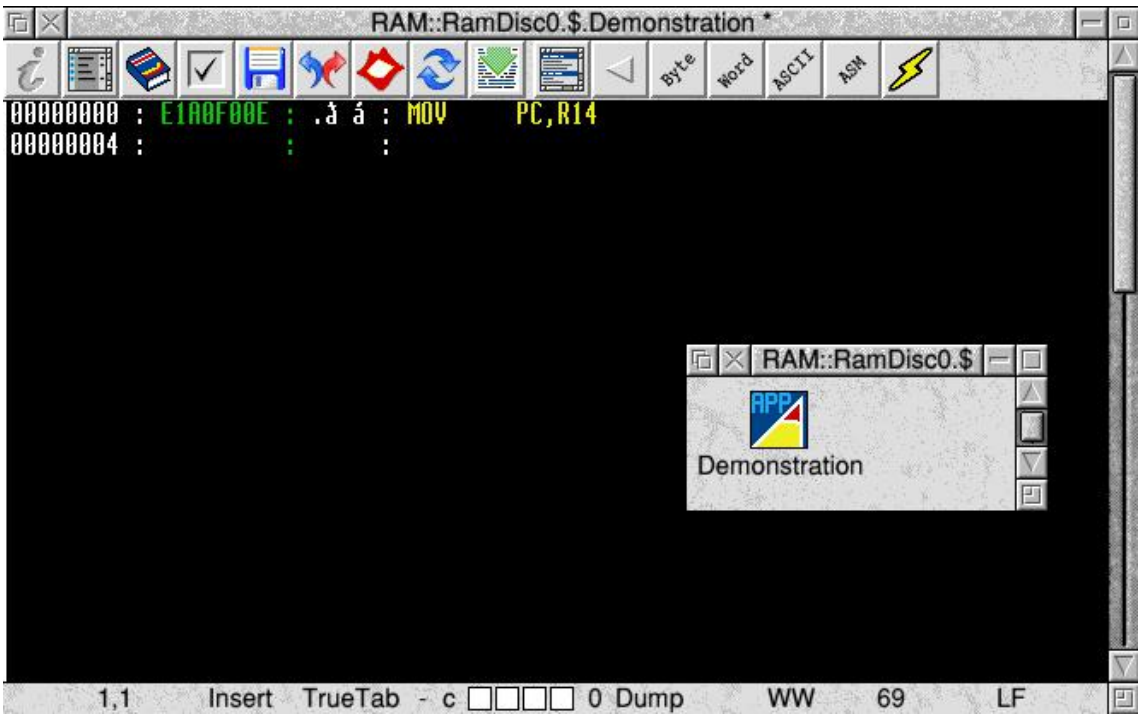


Bild 2.1.3: Das erste lauffähige Programm in Maschinensprache im Editor !StrongED

Wir können selbstverständlich auch den Editor !Zap verwenden. Wir positionieren dazu den Mauszeiger über dem Symbol von !Zap unten auf der Symbolleiste. Dann drücken wir auf die mittlere Maustaste bzw. das Drehrad. Nun erscheint ein Menü. Wir schieben die Maus nacheinander über die Einträge `Create -> New file -> Other` und klicken im letzten Menü in der Liste auf auf den Eintrag `Absolute`. (Um die Untermenüs angezeigt zu bekommen, müssen wir den Mauszeiger rechts über die jeweiligen Pfeile schieben.)

In !Zap funktioniert die Eingabe anders als in !StrongED. In !Zap müssen wir die Mnemonics eingeben. Also `MOV PC, Rl4`. Das Programm rechnet nach einem Druck auf die Eingabetaste diesen Befehl sogleich in Maschinencode um. Diese Datei können wir jetzt mittels einem Druck auf die Funktionstaste F3 oder über das Menü wieder in einem Verzeichnisfenster ablegen (speichern).

Mittels einem Doppelklick auf das Symbol der Datei können wir das Programm starten.

Es scheint sich nichts zu tun. Es tut aber doch etwas: Es gibt die Kontrolle sofort wieder ans Betriebssystem zurück. Der Computer stürzt nämlich *nicht* ab!

Wir können den Befehl im Editor ändern in

```
ElA0 E00F
```

Wie aus der Darstellung in der rechten Spalte ersichtlich ist, sind jetzt Quell- und Zielregister vertauscht. Bei `&E` und `&F` handelt es sich also um die beiden Register 14 und

15. Das Register 15 heißt auch PC. Das ist der Programmzähler (engl. *programm counter*, kurz PC). Dieser Befehl schreibt den aktuellen Wert von Register 15 ins Register 14. Diesesn Befehl sollten wir an dieser Stelle jedoch tunlichst *nicht* starten, denn sonst hängt sich nach dem Programmstart womöglich noch der Rechner auf!

Wir können dem Befehl `E1A0 F00E` einen weiteren Befehl voranstellen und so unser Programm erweitern:

```
E1A0 0000
E1A0 F00E
```

Der hinzugefügte Befehl `&E1A0 0000` schreibt den aktuellen Wert von Register 0 ins Register 0. Das ist Unsinn. Denn dort steht ja bereits dieser Wert! Sollte hier aber zur Übung dienen.

## 2.2 BBC BASIC

Das BBC BASIC kann unter der Befehls- oder Kommandozeile mit dem Befehl `*BASIC` gestartet werden. Zu beachten ist, daß unter BASIC alle Befehle groß geschrieben sein müssen.

In BBC BASIC können wir mit Hilfe der Anweisung `!` Werte in den Speicher klopfen. Auf dem Commodore 64 entspräche das dem Befehl `POKE`. Folgendes Programm schreibt die Werte `E1A0 F00E` ab der hexdezimalen Adresse `&9000` in den Arbeitsspeicher:

```
10 !&9000=&0E
20 !&9001=&F0
30 !&9002=&A0
40 !&9003=&E1
```

```
Listing 2.2.1
```

Wir können hier den Speicher nicht ab der Adresse `&8000` nutzen, weil sich dort das gestartete BBC BASIC befindet. Sonst würden wir dieses überschreiben und damit kaputtmachen.

Das BASIC-Programm wird mit `RUN` gestartet. Damit läuft aber noch nicht unser Maschinencode. Das BASIC-Programm klopft erst einmal nur diese Werte in den Speicher.

Wenn wir jetzt `*memory &9000` eingeben, sehen wir an der Adresse `&9000` die Werte `&E1A0F00E`.

Gestartet werden kann der Maschinencode mit dem Befehl `CALL &9000`.

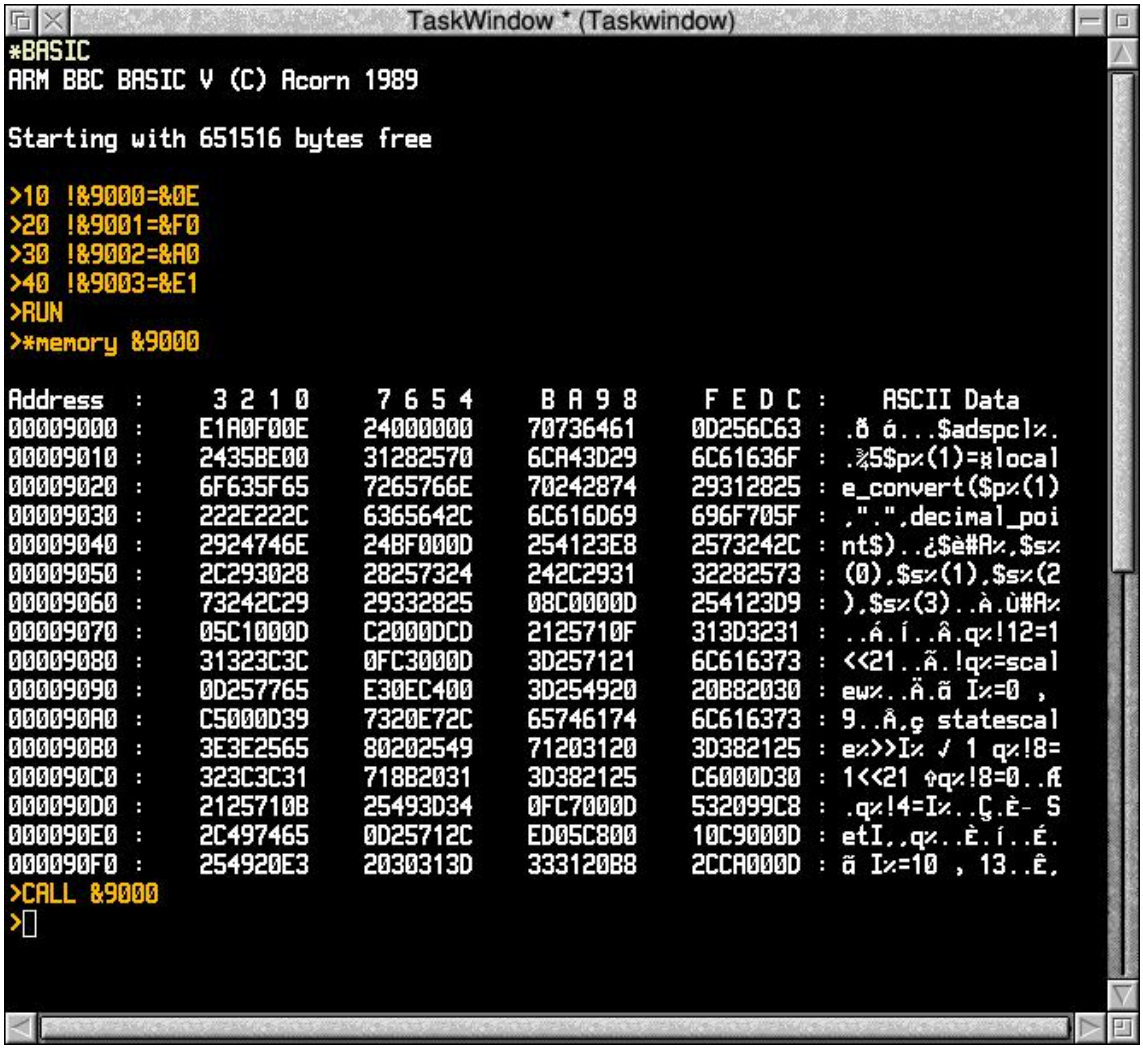


Bild 2.2.1: Unser Maschinenprogramm in BASIC

In Bild 2.2.1 fällt auf, daß der Speicher tatsächlich byteorientiert ist, d. h. daß eine Adresse immer acht Bit umfaßt. In der hexdezimalen Schreibweise sind das zwei Stellen. Außerdem fällt auf, daß ein Befehl immer vier Bytes umfaßt, also 32 Bit. Auf der ARM ist das ein Word. Damit umfaßt ein Befehl vier Adressen. Die niedrigste Adresse eines Befehls steht in der Darstellung von Bild 2.2.1 aber ganz rechts, die höchste ganz links. Das liegt vermutlich daran, weil die höchste Adresse auf der ARM immer den eigentlichen Befehl umfaßt. Das ist reine Festlegungssache und hätte man wohl auch anders machen können.

Wir können das Listing 2.2.1 natürlich auch mit einer Schleife machen. Das sähe dann so aus:

```
10 FOR a = 0 TO 3
20 READ b
30 !(&9000 + a) = b
40 NEXT a
50 DATA &0E, &F0, &A0, &E1
```

Listing 2.2.2

Die Werte können wir ruhig hexdezimal eingeben. Wir können auch hexdezimale und dezimale Werte zusammenzählen lassen. Daran sieht man, wie leistungsfähig das mächtige BBC BASIC ist!

Mit Hilfe von BBC BASIC können wir hexdezimale Werte ins dezimale umrechnen lassen und umgekehrt.

Der Befehl

```
PRINT &F4
```

z. B. schreibt z. B. den dezimalen Wert 244 auf den Bildschirm.

Mit Hilfe der Befehle

```
a=34; PRINT &a
```

kann man den dezimalen Wert a ins hexdezimale umrechnen lassen. Der hexdezimale Wert beträgt &10.

Damit aber noch nicht genug! Das mächtige BBC-BASIC des Archimedes beinhaltet auch einen Assembler. Wir können damit ebenfalls unser erstes Beispielprogramm aus Abschnitt 2.1 erzeugen. BBC BASIC kann den Assemblerbefehl `MOV PC, R14` direkt in den Maschinencode `E1A0 F00E` umrechnen. Wir müssen die hexdezimalen Werte damit nicht mehr direkt eingeben. Weil es aber einige nette Seiteneffekte mit BBC BASIC gibt, sollten wir uns das unbedingt näher ansehen.

Listing 2.2.3 erzeugt den Maschinencode `E1A0 F00E` aus dem Assemblerbefehl `MOV PC, R14`:

```
10 DIM code% (100)
20 FOR pass = 0 TO 3 STEP 3
30 P% = code%
40 [
50  OPT pass
60  .start
70  MOV PC, R14
80  ]
90 NEXT pass
100 PRINT "Startadresse &:" ~code%
110 PRINT "Programmgröße ist &"; P%-start; "Bytes lang"
120 END
```

`Listing 2.2.3`

Man kann das Programm in einen Editor eingeben, als Datei mit dem Dateityp BASIC speichern und per Doppelklick starten. Oder man startet BBC BASIC in einem Kommandozeilenfenster (Tastekombination STRG + F12) oder auf der Kommandozeile (Funktionstaste F12) und gibt das Programm *mit Zeilennummern* ein.

Nun ist es hier nicht besonders sinnvoll, das Programm mit einem Doppelklick zu starten. Denn das Programm 2.2.3 übersetzt wieder nur den Assemblerbefehl `MOV PC, R14` in Maschinencode. Der Maschinencode selbst wird aber nicht ausgeführt.

Wir sollten bei den folgenden Untersuchungen daher wieder auf die Kommandozeile von BASIC zurückgreifen. Das Programm kann mittels `LOAD "Dateiname"` von einer Datei ins BASIC geholt werden. Damit das funktioniert, ist aber vor jedem Befehl ein Leerzeichen zu setzen! Wichtig ist auch, daß sich die Datei im aktuellen Arbeitsverzeichnis befindet. Der Inhalt des Arbeitsverzeichnisses kann mit dem Befehl `*CAT` abgerufen werden. Bei neueren Versionen von RISC OS kann das aktuelle Arbeitsverzeichnis mittels dem Menüeintrag `Set Directory` gesetzt werden.

CAT steht vermutlich als Abkürzung für englisch *catalog*, zu deutsch Katalog. Unter Unix werden die Verzeichnisse auch als Kataloge bezeichnet.

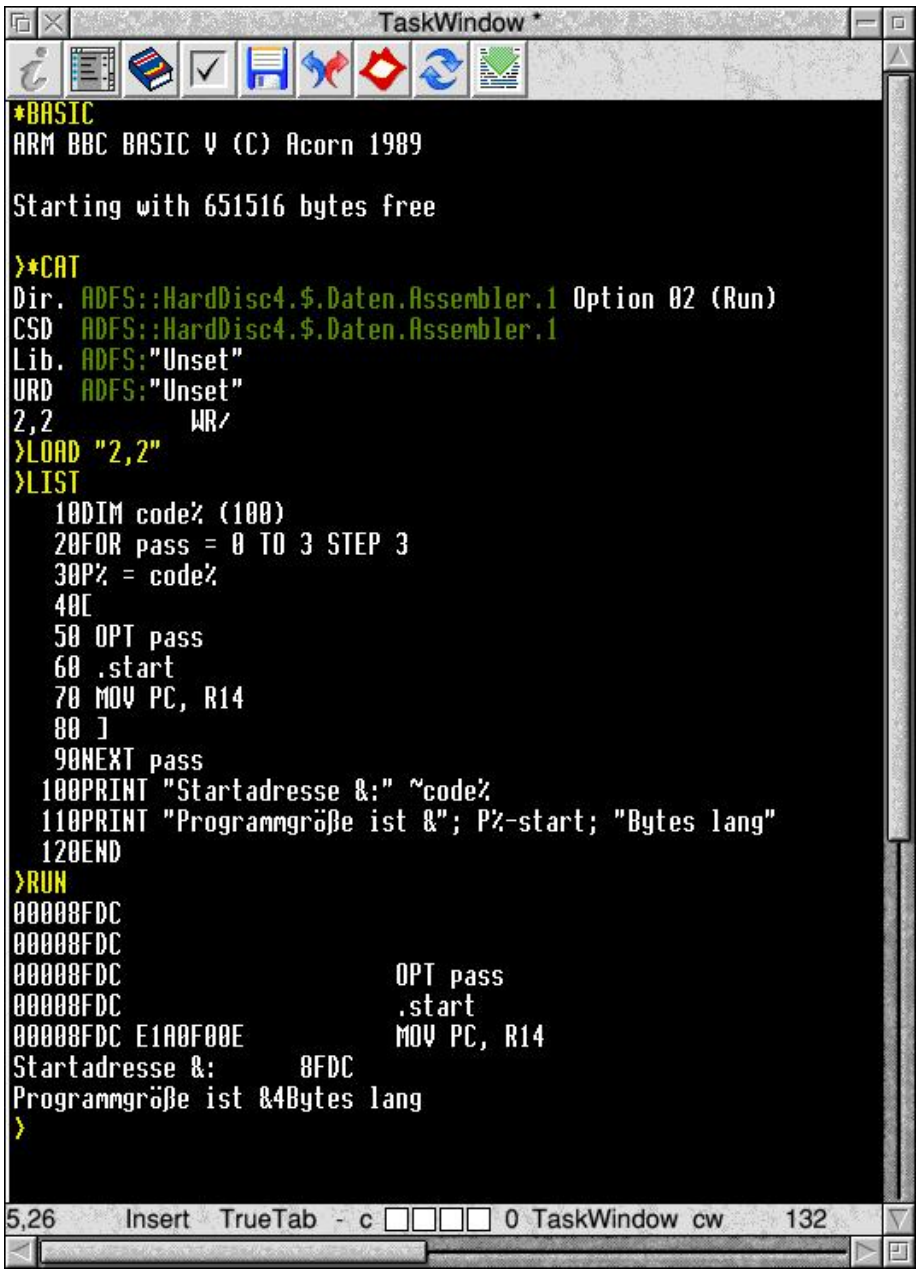


Bild 2.2.2: Unser erstes Assembler-Programm in BBC BASIC

Wir sehen, daß hier das Programm nicht an der Adresse `&8000`, sondern an der Adresse `&8FDC` beginnt. Das liegt daran, daß wir bereits ein Programm gestartet haben, welches an der Adresse `&8000` beginnt: nämlich BBC BASIC. BBC BASIC weist dann unserem Programm innerhalb des Bereiches, den BBC-BASIC selbst verwendet, einen freien Speicherplatz zu. Dieser Speicherplatz und damit auch die Startadresse kann ein jedes Mal anders sein.

Das von BBC-BASIC zusammengebaute (*engl. to assembly*) Programm können wir dann mittels dem Befehl `CALL <startadresse>` starten. Im Bild 2.2.2 wäre das `CALL &8FDC`. Es verhält sich hier also so ziemlich anders als wie unter dem Abschnitt 2.1

beschrieben, wo wir das Programm unter RISC OS direkt eingegeben und gestartet hatten. Mittels dem Befehl `*memory <Adresse>` können wir uns wieder den Speicher näher ansehen.

Geben wir `*memory &8000` ein, so sehen wir, daß sich dort bereits ein Programm befindet. Es handelt sich um das bereits erwähnte BBC-BASIC, welches wir in Bild 2.2.2 gestartet haben.

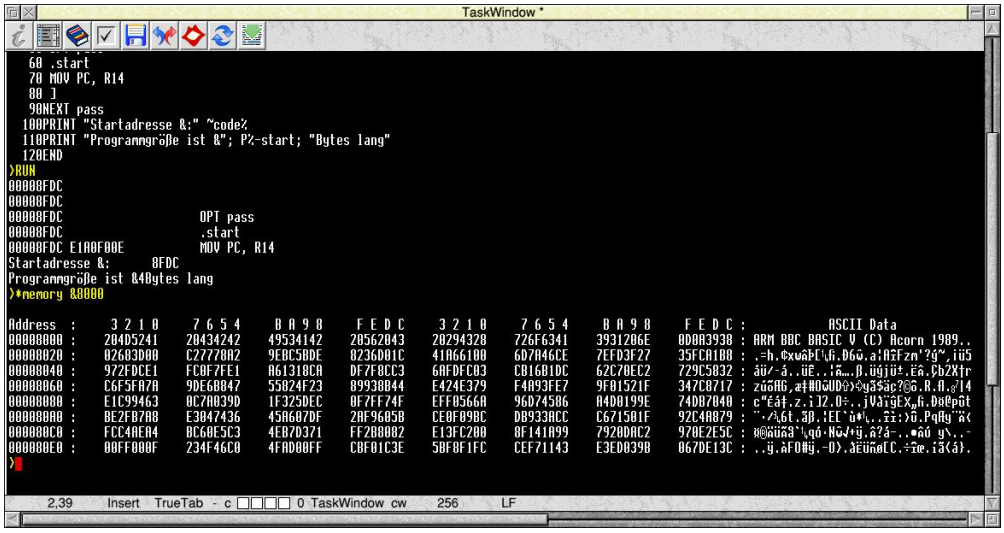


Bild 2.2.3: BBC-BASIC im Speicher ab Adresse &8000

Geben wir hingegen `*memory &8FDC` ein, so sehen wir unser kleines Programm `MOV PC, R14` oder die Werte `E1A0 F0E0` an der Speicheradresse `&8FDC`:


Bild 2.2.4: Unser kleines Assembler-Programm `MOV PC, R14` liegt hier im Speicher ab Adresse `&8FDC`.

Wie bringen wir jetzt dieses kleine Programm `E1A0 F0E0` in eine Datei? Das können wir mittels dem Befehl

```
*SAVE <Dateiname><Startadresse>+<Länge>
```

machen.

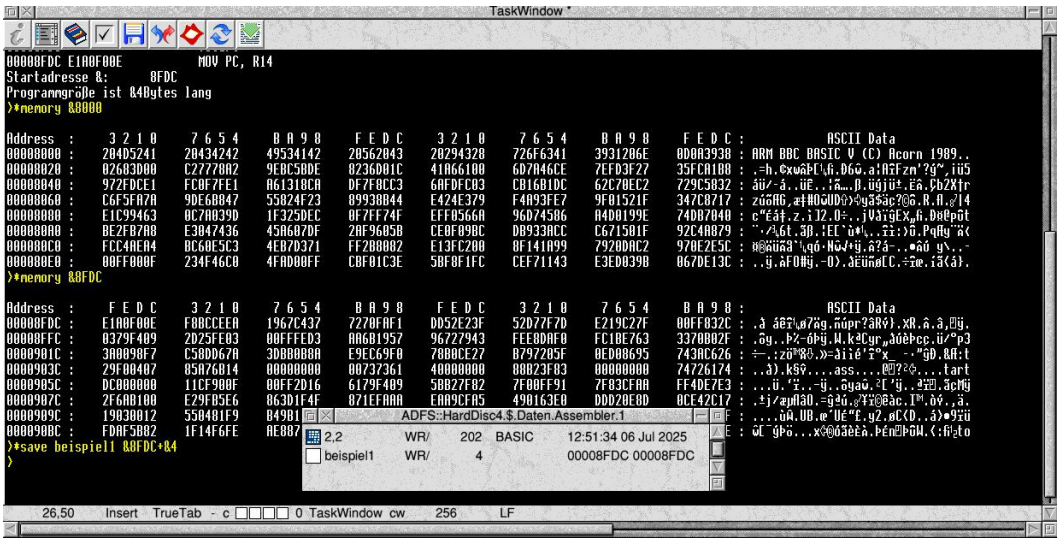


Bild 2.2.5: Unser kleines Maschinenprogramm in eine Datei gespeichert.

In unserem Beispiel lautet der Befehl

```
*save beispiell &8FDC+&4
```

Man wird feststellen, daß diese Datei keinen Dateityp hat. Die Startadresse wird mit abgespeichert und angezeigt, wenn man die Darstellung im Verzeichnisfenster (englisch *filer*) auf *Full Info* umschaltet. RISC OS weiß bei einem Doppelklick also schon, wo es die Datei anspringen muß. Das erkennt auch !StrongED und zeigt das Programm ab der richtigen Startadresse an. Es bleibt jedoch zu vermuten, daß bei dieser Vorgehensweise der Speicher ab `&8000` bis zur Startadresse unseres Programms unbenutzt bleibt, was schade ist.

Auch eine solche Datei kann also mittels Doppelklick gestartet werden. Lädt man diese Datei in !StrongED, so werden einem im Dump-Modus wieder die hexdezimalen Werte `E1A0 F00E` begegegnen.

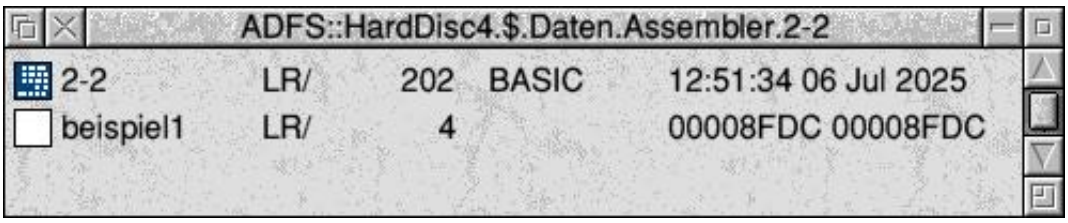


Bild 2.2.5: Unser kleines Beispielprogramm `beispiell` mittels dem BBC BASIC Assembler erstellt und in eine Datei gespeichert.

Interessant ist auch, daß man in BBC-BASIC BASIC-Befehle und den Assembler miteinander kombinieren kann.

## 2.3 Acorn Assembler (DDE)

Der *Acorn Assembler* ist Bestandteil des *Desktop Development Environment* (Oberflächenentwicklungsumgebung, kurz *DDE*) von Acorn, obwohl man damit natürlich keine Programme schreiben muß, welche unter der graphischen Oberfläche von RISC OS laufen. Das DDE bekommt man heutzutage von RISC OS Open Limited oder über den Plingstore.

Auch der Acorn Assembler kann den Befehl `MOV PC, Rl4` zur Beendigung eines Programmes in Maschinensprache umrechnen. Unser erstes kleines Beispiel schaut hier so aus:

```
AREA     |main|, CODE
ENTRY
MOV pc, rl4
END
```

`Listing 2.3`

Ganz wichtig ist hier, am Anfang eines jeden Befehls immer ein Leerzeichen zu setzen! Der Acorn Assembler versteht sonst nicht, was man will.

Wir erstellen einen Ordner oder ein Verzeichnis mit dem Namen `s`. Dies ist eine Krücke, weil es unter RISC OS keine Dateiendungen, sondern nur Dateitypen gibt. Daher hat man die Dateiendungen in die Ordner- oder Verzeichnisnamen verschoben. Was normalerweise hinten steht, steht unter RISC OS so davor! Unter RISC OS kann man keine Dateien gleich benennen, auch wenn sie unterschiedliche Dateitypen aufweisen.

Dann erstellen wir z. B. mit !StrongEd eine Text-Datei mit obigen Inhalt und speichern sie unter irgendeinem beliebigen Namen in das vorher erstellte Verzeichnis mit dem Namen 's' ab. Im übrigen bietet uns !StrongEd den Mode !ObjAsm an. Damit erkennt !StrongEd die Sprachelemente eines jeden Assembler-Quellprogramms und kann die einzelnen Befehle, Variablen usw. farblich besser hervorheben. Befindet sich eine solche Textdatei in einem

Verzeichnis mit dem Namen `s`, so erkennt StrongEd die Struktur dieser Datei von selbst und schaltet beim Laden oder Öffnen dieser Datei auf den Mode ObjAsm um.

Eine solche Datei enthält einen *Quellcode*. Diesen Quellcode versteht ein Prozessor jedoch nicht. Daher lassen wir nun diesen Quellcode von einem oder mehreren Programmen auf mehreren Schritten nacheinander in Maschinensprache umrechnen.

Im Verzeichnis von `DDE.Apps.DDE` benötigen wir jetzt die beiden Programme !ObjAsm und !Link. Da beide im Multitasking laufen, können wir ein jedes einfach per Doppelklick mit der Maus starten.

Die Bedienung beider Programme findet wie unter RISC OS gewöhnt per *Drag & Drop* (ziehen und fallen lassen) mit der Maus statt.

Nun nehmen wir die erstellte Textdatei und lassen sie auf ObjAsm auf der Symbolleiste fallen. Es öffnet sich ein Fenster. Wir könnten dort weitere Einstellungen vornehmen. Klicken aber einfach auf den Knopf *Run*.

Es wird uns eine Speicherdialogbox angeboten. Die Datei ziehen wir einfach irgendwohin, am besten aber in ein Verzeichnis mit dem Namen `o`. Wir können uns die neue Datei im Texteditor ansehen, werden aber nicht recht schlau daraus werden. Es handelt sich noch nicht um Maschinensprache.

Nun ziehen wir die neue Datei auf das Symbol mit dem Namen Link auf der Symbolleiste. Die neue Datei speichern wir wieder irgendwohin. Es handelt sich jetzt um eine Datei mit dem Dateityp Absolute und dem ausführbaren Maschinencode. Diese können wir nun mit einem Doppelklick starten.

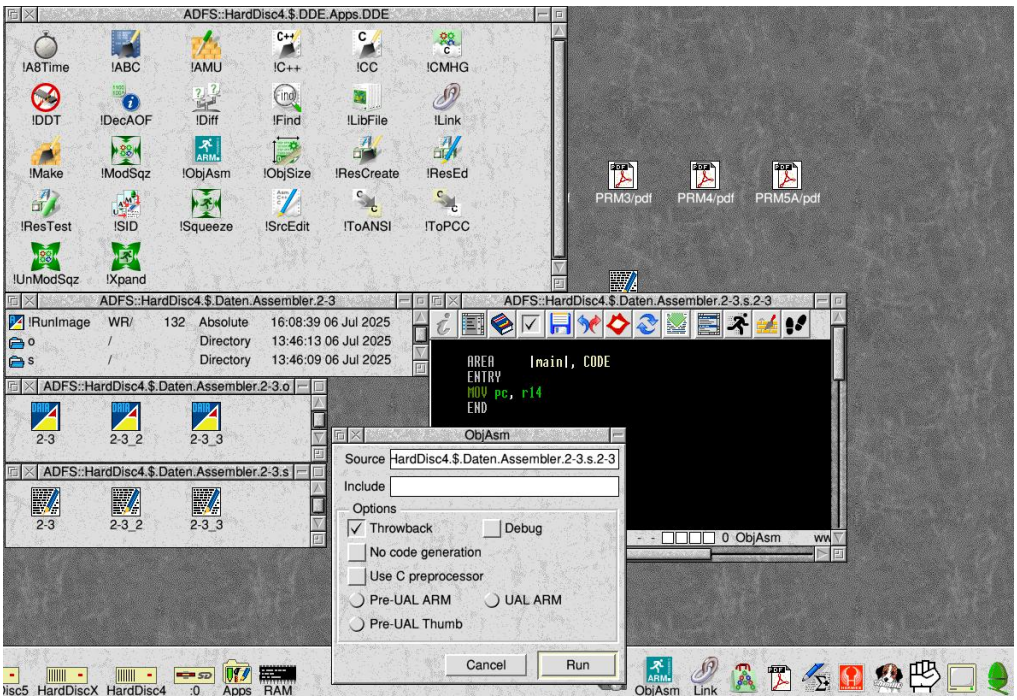


Bild 2.3.1: Der Acorn Assembler im Einsatz.

Das Programm gibt die Kontrolle wie gewohnt sofort wieder ans Betriebssystem zurück. Allerdings fällt auf, daß die Datei des Programms mit 132 Bytes ungewöhnlich groß ist. Sehen wir uns den Inhalt dieser Datei einmal in !StrongEd an.

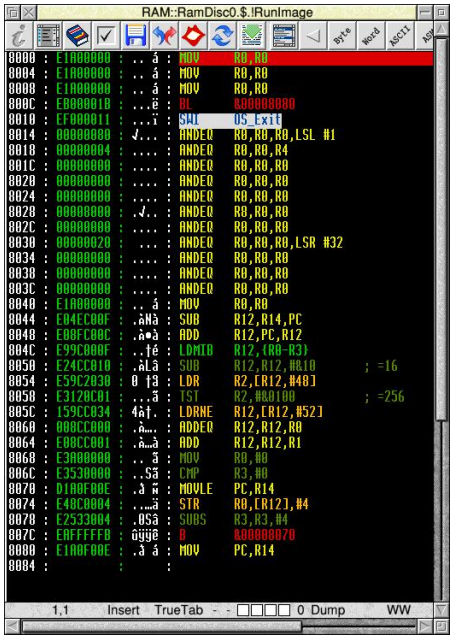


Bild 2.3.2: Das ungewohnt lange Programm

Wir sehen, daß das Programm wieder mit der hexdezimalen Adresse `&8000` beginnt.

Allerdings kommt dann drei Takte lang der bereits erwähnte, recht sinnlose Befehl `E1A0 0000` oder `MOV R0, R0`, der eigentlich gar nichts tut. Anschließend kommt dann plötzlich der Befehl `&EB` (in Maschinensprache) oder `BL` (in Assemblersprache). Diesen haben wir jedoch gar nicht im Quellcode eingegeben! Was soll denn das? Sowas hatte ich nicht erwartet! Wenn notwendig, dann hätte man die anderen Befehle auch im Assembler eingeben und in Maschinensprache umrechnen lassen können. Das wäre der bessere, weil durchsichtigere Weg gewesen!

Unseren eigenen Befehl finden wir dann viel weiter unten an der Stelle &8080.

Wenn man den Acorn Assembler verwendet, gibt es also ebenfalls mitunter unerwünschte Seiteneffekte.

## 2.4 GCC

Die GCC, das ist die GNU Compiler Collection. Diese Programmsammlung stammt aus der Welt von Unix und wurde nach RISC OS portiert. Sie ist sehr mächtig - hat allerdings das Problem, daß es von einer anderen Plattform stammt.

Im Gegensatz zum Acorn Assembler arbeitet die GNU Compiler Collection noch wie früher auf der Kommandozeilenebene. Drag & Drop oder eine graphische Oberfläche ist dieser Programmsammlung fremd. Das verwundert nicht. So revolutionär es damals auch war: Unix stammt aus den sechziger Jahren des letzten Jahrhunderts und ist schon uralt.

Die GCC sollte man z. B. über den Plingstore bekommen.

Wie bereits auch schon für den Acorn Assembler, legen wir hier wieder ein Verzeichnis mit dem Namen `s` an.

Nun erzeugen wir mit !Zap oder !StrongEd oder auch !Edit eine Textdatei. In dieser tippen wir unser kleines Beispielprogramm in der Assemblersprache des GCCs nieder:

```
    .global _start

_start:
     MOV PV, R14
```

Listing 2.4

Wie man merkt, unterscheidet sich dieser Code vom Listing 2.3. Assembler ist leider nicht gleich Assembler.

Programm 2.4 speichern wir in das Fenster des vorher angelegten Verzeichnisses, welches den Namen `s` erhalten hat. Nun klicken wir mit der rechten Maustaste auf das Schließkreuz des Fensters, um in der Verzeichnisstruktur eine Ebene tiefer zu gelangen. Dort klicken wir

im Menü den Eintrag `Set Directory` an. Damit setzen wir das aktuelle Arbeitsverzeichnis auf diesen Pfad.

Nun klicken wir auf den Task Manager auf der Symbolleiste. Das ist normalerweise das Symbol ganz rechts. Dort ziehen wir den Balken neben dem Eintrag *Next* auf mehrere Megabytes auf. Der GCC braucht wirklich viel Speicher!

Als nächstes starten wir ein Task Window. Das geht über das Menü vom Task Manager auf der Symbolleiste oder durch die Tastenkombination STRG + F12 (auf englischen Tastaturen ist STRG CTRL) und geben den Befehl `*CAT` gefolgt von einem Druck auf die Eingabetaste (die große Taste mit dem Pfeil) ein. Wenn wir alles richtig gemacht haben, erscheint nun der Inhalt des Verzeichnisses mit dem angelegten Unterverzeichnis `s`.

Dann geben wir nacheinander die Befehle

```
*as -o <Dateiname>.o <Dateiname>.s
*ld -o <Dateiname> <Dateiname>.o
```

ein. Natürlich ist statt `<Dateiname>` der vergebene Dateiname einzugeben! In unserem Beispiel in Bild 2.4.1 ist das 2-4.

Man kann diese zwei Zeilen auch in eine Datei vom Dateityp Obey packen und diese Datei mit einem Doppelklick starten.

Und schon wird unser kleines Assemblerprogramm vom GNU in ausführbaren Maschinencode umgerechnet.

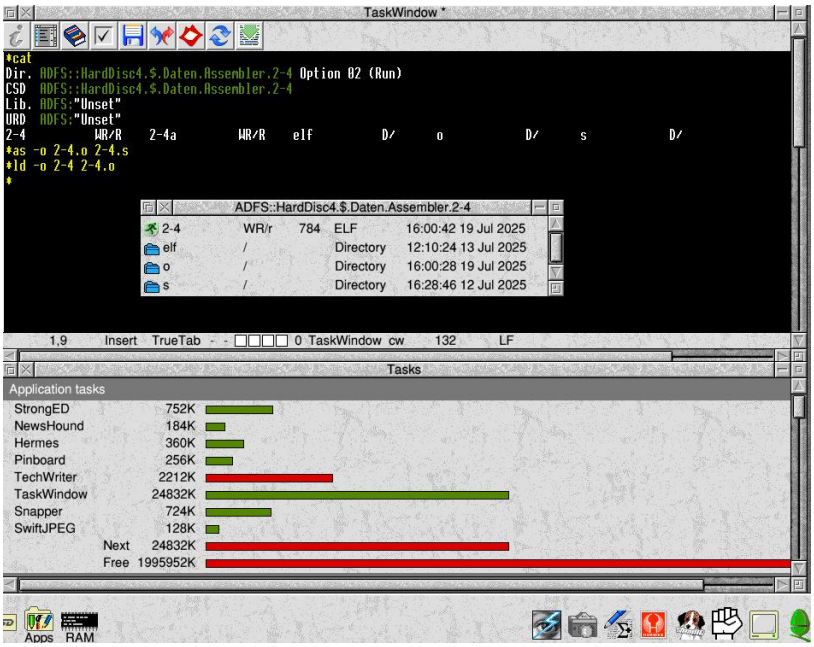


Bild 2.4.1: Der GCC in Aktion

Das Programm funktioniert zwar auch hier. Die erzeugte Datei läßt sich mittels einem Doppelklick starten. Aber man sehe und staune! Das übersetzte Programm ist hier sagenhafte 784 Bytes groß! Und noch etwas fällt auf: Bei der erzeugten Datei handelt es sich ja gar nicht um eine Datei vom Dateityp Absolute. Sondern sie hat den Dateityp ELF!

Dieser Dateityp stammt aus der Unixwelt. Und bringt ein Haufen Ballast mit, den unser Programm so gar nicht bräuchte. Wenn man sich das Programm im Dump-Modus von StrongEd ansieht, so sieht man eine lange Liste von Befehlen. Irgendwo darunter findet sich dann in einer Zeile unser Befehl: `E1A0 F0E0` oder `MOV PC, R14`.

Man sieht also: auch bei der Verwendung der GCC gibt es merkwürdige Effekte. Assembler ist nicht gleich Assembler. Und damit fängt der ganze Ärger an! Man kann einen Quellcode, der für Assembler A geschrieben wurde, nicht einfach durch Assembler B in Maschinensprache umrechnen lassen. Das funktioniert nicht.

## 2.5 Die Kommandozeilenebene

Außerhalb von BBC-BASIC hatten wir die Programme bisher immer nur von der Oberfläche aus mit einem Doppelklick mit der Maus gestartet.

Nun gibt es auch die Möglichkeit, ein Programm nur in den Speicher zu laden, ohne es sofort auszuführen. Das kann man auf der Kommandozeile (Druck auf Funktionstaste F12 oder Kommandofenster mit der Tastenkombination STRG + F12 oder über das POP-UP-Menü des Aufgabenmanagers) mittels dem Befehl

```
*load <Dateiname>
```

machen. Starten kann man das Programm dann mit

```
*go <Startadresse>
```

Gibt man keine `<Startadresse>` ein, so geht RISC OS davon aus, daß die Startadresse `&8000` ist.

Dem Mausklick gleichzusetzen ist der Befehl

```
*run <Dateiname>
```

Dieser Befehl lädt eine Datei in den Arbeitsspeicher und führt sie sofort aus.

Mittels dem Befehl `help <Befehl>` kann man mehr über einen Befehl erfahren, z. B.

```
*help run
```

```
==> Help on keyword Run
*Run loads and executes the named file, passing optional
parameters to it.
Syntax: *Run <filename> [<parameters>]
*
```

Mit dem Befehl `*memory <Startadresse> <Endadresse>` läßt sich ein Speicherbereich ansehen.

Mit dem Befehl `*memoryi <Startadresse> <Endadresse>` läßt sich ein Speicherbereich diassemblieren, d. h. die Inhalte werden als Mnemonics verstanden angezeigt.

Selbst ein Debugger, mit dem man ein Programm an jeder Stelle unterbrechen und sich die Register ausgeben lassen kann, ist bei RISC OS schon mit eingebaut. Man muß das Programm zuvor jedoch in den Arbeitsspeicher geladen haben, ohne es auszuführen. Deshalb ist es so wichtig zu wissen, wie man unter RISC OS ein Programm *nur* in den Arbeitsspeicher holen kann, ohne es sofort auszuführen. Man muß dazwischen nämlich noch etwas machen.

Haben wir also ein Programm im Maschinencode vom Dateityp Absolute mittels dem Befehl `*load <Dateiname>` in den Arbeitsspeicher geholt, können wir mit Hilfe des Befehls

```
*breakset <adresse>
```

das Programm an jeder beliebigen Stelle dazu zwingen, abzubrechen und eine Übersicht über den Zustand des Prozessors, das heißt, eine Liste aller Werte der Register, auszugeben. Wir können den Befehl auch mehrmals hintereinander anwenden und so auch mehrere Adressen angeben. Eine Liste aller angegebenenAdressen spukt der Befehl `*breaklist` aus.

Nachdem wir das Programm mit Hilfe des Befehls `*go <adresse>` gestartet haben, bricht das Programm an genau jenen Stellen ab, welche wir zuvor mit dem Befehl `*breakset <adresse>` angegeben haben.

Mit dem Befehl `*continue` wird das Programm weiter fortgesetzt, unter Umständen nur bis zum nächsten mittels dem Befehl `*breakset` eingegebenen Adresse.

Mit dem Befehl `*breakclr` können die gesetzten Adressen wieder gelöscht werden.

Diese Befehle sind natürlich auch von BBC BASIC aus erreichbar, indem wir den Stern * vor dem Befehl mit eingeben. Befinden wir uns in der Kommandozeile, so kann der Stern * vor dem Befehl weggelassen werden. Er schadet allerdings auch nicht. So kann man auch innerhalb von BBC BASIC aus debuggen.

# 3 Software Interrupts

Um als Programmierer das Rad nicht ständig neu erfinden zu müssen, stellt uns das Betriebssystem *Programme* zur Verfügung, welche wir von unserem Programm aus anspringen können. Diese werden auch *Unterprogramme* oder *Betriebssystemroutinen* genannt. Das kann uns eine Menge Arbeit ersparen. Die ARM hat dafür einen eigenen Befehl: `&EF`, gefolgt von der Nummer der Routine oder des Unterprogramms. Der Befehl muß in dieser hexdezimalen Schreibweise jedoch wie immer *achtstellig* sein!

In Assembler schreibt man dafür auch `SWI`. `SWI` steht für *Software Interrupt*. Wir werden bei dieser speziellen Art von Unterprogrammen daher auch von SWIs reden.

Bei diesem Befehl, `&EF`, schaut der Prozessor an Hand der folgenden Nummer in einer langen Liste nach, wo er im Speicher das Unterprogramm findet. Dieses springt er dann an und arbeitet er ab. Ist das Unterprogramm beendet, setzt er das ursprüngliche Programm wieder fort. Dazu wird, wie bereits erwähnt, der Wert im Programmzähler verändert.

Solche Betriebssystemroutinen oder Unterprogramme bzw. SWIs können ohne oder mit Paramter aufgerufen, d. h. angesprungen werden. Für RISC OS werden solche Betriebssystemroutinen unter der Bezeichnung SWI im Programmer's Reference Manual beschrieben.

## 3.1 SWIs ohne Parameter

Wir schauen uns der Einfachheit halber zuerst zwei SWIs an, welche ohne Parameter auskommen. Sie haben die hexdezimalen Nummern `&406C0` und `&406Cl`. `&406C0` schaltet die Sanduhr ein, `&406Cl` schaltet sie wieder aus. Probieren wir das also einmal in der Praxis aus und geben zwei kleine Maschinenprogramme in !StrongEd oder in !Zap wie unter Abschnitt 2.1 beschrieben ein:

```
EF04 06C0
ElA0 F0E0
```

Listing 3.1.1

```
EF04 06Cl
ElA0 F0E0
```

Listing 3.1.2

Nicht vergessen: Die Befehle müssen in der hexdezimalen Darstellung immer acht Zeichen lang sein. Der eigentliche Befehl `EF` steht ganz links. Die fehlenden Stellen zwischen dem Befehl und der Nummer des SWIs müssen wir deshalb immer mit Nullen auffüllen! Die kleinste Nummer fängt nämlich rechts an, nicht links. Die Zahlen bauen sich dann wie

gewohnt von rechts nach links auf, d. h. der Übertrag wird immer links an der vorherigen Stelle angefügt.

Diese Programme müssen wir wieder in zwei Dateien speichern, bevor wir sie jeweils mittels einem Doppelklick starten können. Siehe hierzu auch den Abschnitt 2.1.

Starten wir Listing 3.1.1 so verwandelt sich der Mauszeiger in eine Sanduhr. Starten wir Listing 3.1.2, so verwandelt sich die Sanduhr wieder in den Mauszeiger zurück.

Dies sind die ersten zwei lauffähigen Programme, welche irgendwas bewirken, was der Anwender auch sieht. Beide Programme sind jeweils 8 Bytes groß. So einfach ist das!

In Assembler können wir ebenfalls die Nummer des SWIs verwenden. Allerdings soll man laut diversen Lehrbüchern nicht die Nummer, sondern den Namen des SWIs angeben. Dieser Name wird vom Assembler dann in die Nummer des SWIs umgerechnet und ist im Programmer's Reference Manual festgelegt. Der Prozessor selbst kann mit dem Text nichts anfangen. Er braucht die Nummer.

Listing 3.1.2 sähe im Assembler des BBC-BASICs so aus:

```
DIM code% (100)
FOR pass = 0 TO 3 STEP 3
P% = code%
[
 OPT pass
 .start
 SWI "Hourglass_On"
 MOV PC, R14
 ]
NEXT pass
PRINT "Startadresse &:" ~code%
PRINT "Programmgröße ist &"; P%-start; "Bytes lang"
END
```

Listing 3.1.3

Listing 3.1.3 macht nur Sinn, wenn auch der Mauszeiger zu sehen ist. Falls der Mauszeiger abgeschaltet ist, z. B. weil man durch Druck auf die Funktionstaste F12 von der WIMP auf die Befehlszeile gewechselt hat, kann man mittels dem Befehl `*pointer 1` zuvor den Mauszeiger aktivieren.

Man kann in Listing 3.1.3 nun noch die Nummern `&406C0` und `&406C1` statt den Texten "Hourglass_On" und "Hourglass_Off" ausprobieren und auch versuchen, die Programme 3.1.1 bzw. 3.1.2 mit dem Acorn Assembler oder der GCC zu erstellen. Als Grundlage hierfür mögen die Assemblerbefehle aus Listing 3.1.3 hilfreich sein. Das &-Zeichen ist dabei stets mit einzugeben.

Beim Acorn Assembler (DDE) müssen wir berücksichtigen, daß wir dem Assembler erst die Namen der SWIs bekannt machen müssen (falls wir diese verwenden wollen). Diese Namen sind in der Datei

```
DDE.Sources.DDE-Examples.ObjAsm.AsmHdrs.h.SWINames
```

zu finden. Am besten, man kopiert diese Datei mit den übergeordneten Verzeichnissen `AsmHdrs.h.SWINames` ins aktuelle Verzeichnis und bindet diese Datei mit dem Befehl `GET AsmHdrs.h.SWINames ins` Quellprogramm mit ein. Leider sind diese Art von Listen nicht im Programmverzeichnis vom Acorn Assembler enthalten. Diese unschöne Art wurde wohl von Unix übernommen.

Beim Befehl `GET` handelt es sich *nicht* um ein Mnemonic! Also um keinen Befehl, welcher der Prozessor in irgend einer Art und Weise kann. Dieser Befehl wird daher auch nicht in Maschinensprache übersetzt. Er macht nur dem Assembler etwas bekannt, damit er an Stelle des Namens die entsprechende Nummer verwenden kann. Sonst kann er mit dem Namen nichts anfangen.

Außerdem ist zu berücksichtigen, daß der Name des SWIs im Acorn Assembler im Vergleich zum BASIC Assmbler *ohne* Anführungszeichen eingegeben werden muß! Groß- und Kleinschreibung sind ebenfalls zu beachten!

```
      AREA     |main|, CODE

      GET AsmHdrs.h.SWINames

      ENTRY
      SWI Hourglass_On
      MOV pc, rl4
      END
```

`Listing 3.1.5`

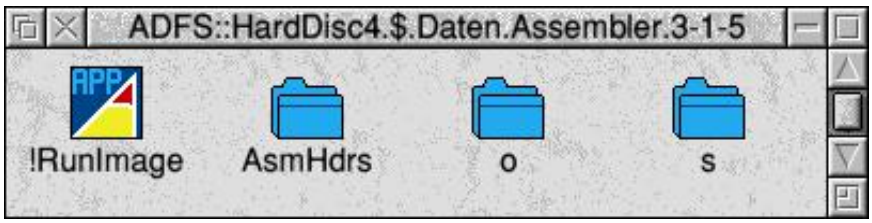


Bild 3.1: Das Arbeitsverzeichnis von Listing 3.1.1.5 für den Acorn Assembler mit dem bereits in Maschinensprache übersetzten Programm

Bei Verwendung der GCC müssen die SWIs noch einmal anders behandelt werden. Am

einfachsten ist es, man schreibt einfach die Nummer des SWIs in hexdezimaler Form hinter den Befehl. Allerdings muß diese Nummer hier statt mit einem & mit einem 0x beginnen.

```
   .global _start

_start:
    SWI 0x406C0
    MOV PC, R14
```

Listing 3.1.6

Übersetzt (compiliert) wird dieser Quellcode wie unter Abschnitt 2.4 für die GCC beschrieben.

Will man bei der GCC die Namen der SWIs verwenden - ich habe dazu nichts gefunden. Der Befehl `GET` und die Datei `AsmHdrs.h.SWINames` für den Acorn Assembler aus Listing 3.1.5 scheint die GCC jedenfalls nicht zu verstehen.

### 3.1.2 SWIs mit einem einzigen Parameter

Parameter werden üblicherweise mit den Registern des Prozessors übergeben. Das heißt, man füllt zuerst die Register des Prozessors mit den entsprechenden Daten. Anschließend ruft man den entsprechenden SWI auf.

Der erste SWI mit der Nummer `&0` gibt - wen sollte es wundern! - ein Zeichen auf den Bildschirm aus. Klar, man möchte ja etwas auf dem Bildschirm sehen. Deshalb ist dieser SWI so wichtig. Das Zeichen, welches man ausgeben möchte, muß ASCII-codiert im Register R0 stehen.

Das kleine 'a' hat den ASCII-Wert 65 oder hexdezimal 41. Das geht dann so:

*Maschinensprache:* *Assembler:*

```
E3A0 0041                  MOV R0, #&41
EF00 0000                  SWI OS_WriteC
E1A0 F00E                  MOV PC, R14
```

Listing 3.2.1 und Listing 3.2.2

Wie man beim Vergleich von Listing 3.2.1 und Listing 3.2.2 erkennen kann, gibt es in Maschinensprache zwei verschiedene MOV-Befehle. Der eine Befehl (E1) überträgt den Wert von einem Register in ein anderes. Der andere Befehl (E3) aber schreibt einen festen Wert in ein Register. Dieser Wert ist Bestandteil von dem Befehl, der im Speicher steht und *vier* Bytes umfaßt.

Nun gibt es hierbei das Problem, daß dieser Wert nur ein Byte (8 Bit) umfassen kann. Denn die anderen drei Bytes werden für den Befehl selbst gebraucht. Ein Register ist aber 32 Bit breit! Wir können mit diesem Befehl (E3) allein also kein Register füllen.

Im Falle vom `SWI 0` oder `OS_WriteC` ist das nicht schlimm, weil diese Routine nur den (erweiterten) ASCII-Satz verarbeiten kann und dieser (erweiterte) ASCII-Satz nur 1 Byte (oder acht Bit) umfaßt.

## Befehlsübersicht

| Maschinensprache: | Assembler: |
|---|---|
| `E0A0 <ZR>00<QR>` | `MOV <ZR>, <QR>` |

mit

```
<ZR>: Zielregister
<QR>: Quellregister
```

Kopiert den Inhalt vom Quellregister ins Zielregister.

Wichtige Befehle und Beispiele:

| Maschinensprache: | Assembler: |
|---|---|
| `E0A0 F00E` | `MOV PC, R14` |

Befehl muß ganz am Schluß eines Programmes stehen, damit RISC OS ordnungsgemäß weiterarbeiten kann. Er wird auch zur Beendigung eines Unterprogramms verwendet.

| Maschinensprache: | Assembler: |
|---|---|
| EF &SWI | SWI |