Das endliche Assemblerbuch

Christian Ullenboom
November 1993

Table of Contents

Vorwort

Assembler, ein Zauberwort. Es flößt einem doch schon etwas Respekt ein, wenn man auf die Frage „In welcher Sprache programmierst Du denn?“, die Antwort „BASIC, Pascal, ein bisschen C und, ach ja, noch Assembler“ erhält. Doch stellt sich gerade für den Anfänger immer wieder die elementare Frage, ob Assembler überhaupt notwendig ist. Heute sind wir schon in der vierten

Computergeneration, OOP (Objektorientiertes Programmieren) ist angesagt, Programmiersprachen mit riesigen Bibliotheken (Libraries) arbeiten in den Werkstätten der Hobbyprogrammierer, Programme existieren, die selbst Programme schreiben, Computer arbeiten mit Taktfrequenzen, die Bereiche erreichen, die wir gar nicht für möglich halten (Cray ist über 100 MHz getaktet, zum Vergleich: der C-64 ist annähernd auf 1 MHz), Rechner sind mit Giga-Speichern ausgerüstet, warum dann heute noch Assembler? Die heutigen C-, Modula-, Oberon-Compiler, und was sonst noch so alles existiert, generieren auch schnelle Programme, warum dann noch Assembler lernen, wo doch jeder sagt, es sei sowieso viel zu umständlich?

Doch halt, wir haben nur einen Rechner mit einer Taktfrequenz von 7,14 MHz und vielleicht 1 MB Speicher (ich spreche hier vom verbreiteten Modell). Und wenn wir auf diesem Rechner Demos sehen, in denen 1000 Sprites, 5 Laufschriften, 3D-Animation, Diashow und Musik gleichzeitig ablaufen, dann kann man sich eigentlich nicht vorstellen, dass dieses in einer anderen Sprache als Assembler programmiert wurde. Denn: Man wird höchstwahrscheinlich schon irgendwo gehört haben, dass Assembler wirklich das schnellste ist, und dass alle anderen Programmiersprachen den Rechner irgendwie nicht ausreizen.

Leider sehe ich in vielen Einführungsbüchern, die die Zaubersprache Assembler näher bringen wollen, immer noch ausschließlich Programme, die zeigen, wie ich eine Zeichenkette auf den Bildschirm ausgebe, oder ein Fensterchen öffne. Dass dies wichtig ist, ist einleuchtend, aber meines Erachtens ist man mit dem kurz eingeschobenen GFA-Interpreter sowieso schneller. Ein `PRINT`- oder ein `WINDOW`-Befehl, und, schwupp! das Resultat, ein ermutigendes „Ich bin das erste BASIC-Programm“, und ein Fenster, das man gleich hin und her schieben kann, erscheint. Soviel Erfolg für so wenig Zeilen. Und auch mit Compilern ist der Aufwand nicht wesentlich größer. Die Compilersprachen bieten zwar nicht die komplexen Befehle die BASIC bietet, und wir müssen uns mit den Betriebssystemstrukturen auseinandersetzen, doch sollten wir uns merken, das alles außer Assembler wirklich schneller zum Ziel verhilft, egal wie ungeliebt die Programmiersprache auch sein mag.

Um in Assembler ein Wort auf den Bildschirm (Screen) zu bekommen oder ein Screen überhaupt zu öffnen, benötigt man etwas mehr Zeit. Zudem erheblich mehr Hintergrundwissen über Computer und Hardware bzw. Betriebssystem. Für ein gutes C Programm benötigt man nicht weniger Wissen, aber es ist immerhin noch etwas anderes. 30 Zeilen sind einem bei einem Neuanfang, z. B. einer Textausgabe in ein Fenster, schon sicher. Im Gegensatz zu den 2 Zeilen in BASIC ist ein Assemblerprogramm wesentliche aufwändiger. (Wir merken schnell, dass man in Assembler tippgewand sein muss.)

Doch wir dürfen Assembler nicht so miesmachen, sonst könnte man das Buch mit der Begründung „Oh nein, viel zu viel Arbeit und viel zu kompliziert“, ja gleich weglegen. Wir müssen uns immer den Anwendungsbereich von Assembler vor Augen sehen. Er liegt da, wo 3D Koordinaten umgerechnet werden müssen, wo neue Fließkommaroutinen benötigt werden, wo ein neues Filesystem verlangt wird (das Fast-File-System ist nur deshalb dem normalen überlegen, da es nachträglich (für 1.3 und in erster Linie der Festplatte) in Assembler programmiert wurde), Sortierroutinen superschnell arbeiten müssen und wo natürlich Demos programmiert werden wollen. Somit versuche ich Assembler da einzusetzen, wo es gebraucht wird. Keine Sprache, in der man komplette Textverarbeitungen schreibt, allerdings Routinen zur schnellen Rechtschreibprüfung. Eine Sprache, die man nicht ausschließlich nutzt, um ein Fenster zu öffnen und darin einen Text auszugeben. Es ist eine Sprache, die jeden faszinieren kann, und wenn man einmal im Bann von Assembler gefangen wurde, wird man bestimmt mit der Stoppuhr am Rechner sitzen und sich über jede Sekunde freuen, die dass Programm zur Ausführung weniger benötigt. (Äußerungen wie „Meine Punktsetzroutine ist nun um 1/10 schneller geworden“, sind also normal und liegen in der Euphorie des Geschwindigkeitsrausches!).

Assembler ist eine Sprache für Tüftler und für Leute mit viel Zeit. Die Programmerstellung ist leider um ein vielfaches zeitaufwendiger und langwieriger im Vergleich zu anderen Sprachen, doch das Resultat rechtfertigt die lange Erstellungszeit. Somit wünsche ich jedem große Ausdauer und viel Geduld, denn Assembler hat so seine Schattenseiten. Es kann unter Umständen vorkommen, dass man sehr lange nach Fehlern sucht, und zudem dauert es meistens länger ein Programm zu testen, als es zu schreiben.

Aufbau des Buches

In dem Buch versuche ich natürlich in erste Linie Assembler vorzustellen. Da aber auch Hardwarewissen und Hintergrundwissen unabdingbar ist, wird auch auf diese „Randbereiche“ eingegangen. Wer Assembler programmiert sollte zudem auch ein allgemeines Computerwissen haben. Ich vergleiche und beschreibe daher auch den Amiga von anderer Seite, und setze in der Konkurrenz aus, die ja bekanntlich nicht schläft. So werden die Leser etwas über Atari und dessen Ansatz- Multitasking, Macintosh, und dessen Oberflächendesign, PC und dessen Windows-Aufsatz erfahren. Dieses Wissen ist sehr wichtig, und wesentlich für den Verstand und Einsatz von Computern. Der Amiga wird dann in einem ganz neuen Licht erscheinen, und nicht nur unter der Rubrik „Spielecomputer“ seinen Platz haben.

Vorwort (2013)

Mehr als 20 Jahre schlummerte auf dem Datenträger dieses unvollendete Amiga-Buch. Als ich 18-19 Jahre alt war habe ich mit dem Schreiben begonnen, und etwa 2 Jahre daran gearbeitet. Über 400 Seiten sind in der Zeit entstanden. Geschrieben habe ich es auf dem Amiga in einem einfachen Text-Editor, später habe ich große Teile in LaTeX konvertiert. Als das Studium kam, und Verlage an einem Buch nicht mehr interessiert waren (die große Amiga-Zeit war dann vorbei), wanderte der Buchentwurf auf die Platte. Ihn dort für immer zu belassen war eigentlich zu schade, und so habe ich den LaTeX- Text in AsciiDoc (http://www.methods.co.nz/asciidoc/) konvertiert und unter retrobude.de/dokumente/amiga-assembler-buch/ frei unter der Creative Commons (CC-by-nc-nd) online gesetzt.

Ganz klar ist das Buch aus dem Stand von 1993 und alles was nach 1993 geschah bildet das Buch nicht ab; weder der Untergang von Commodore/Amiga, noch die Weiterentwicklung des Betriebssystems. Damit das Buch von größerem Nutzen ist, müsste noch (etwas) Arbeit hineingesteckt werden:

- Es müsste komplett Korrektur gelesen werden, da noch satt Rechtschreibfehler im Text sind. Außerdem ist es der Schreibstil eines Teenagers, das sollte man glattbügeln.
- Ich habe zwar 10 Jahren intensiv Assembler programmiert, doch die Zeiten liegen lange hinter mir; seit 15 Jahren programmiere ich Java (und hierüber sind zwei neue Bücher entstanden). Das bringt mit sich, dass ich von meinem eigenen Buch nicht mehr alles checke. Vieles müsste ich mir wieder aneignen, doch dazu fehlt mir die Zeit/Lust. Es wäre cool, wenn einige Amiga/680x0er/Assembler-Spezis drüber schauen und mögliche fachliche Fehler und Ungenauigkeiten ausmerzen.

Das sind die Dinge, die gemacht werden müssen, als zweites kann man an Erweiterungen denken:

- Nicht immer nutze ich optimal AsciiDoc aus, das kann man noch verbessern, etwa Info-Blöcke setzen, Index-Einträge definieren. Es kann sein, dass +Markierungen+ fehlen, damit Code-Teile, Register, Ausdrücke Fixed-Font-Segmente werde. (Die AsciiDoc-Syntax wird z. B bei http://www.methods.co.nz/asciidoc/userguide.html [umfangreich], http://asciidoctor.org/docs/asciidoc-writers-guide/ oder http://powerman.name/doc/asciidoc erklärt. Konverter von AsciiDoc in z. B. HTML gibt es auch online, das ist prima zum Testen.)
- Vielleicht habe ich bei den Tabellen noch etwas übersehen und sie sind noch Fixed-Font-Blöcke. Natürlich wäre es besser, die "Tabellen" in echte AsciiDoc-Tabellen zu setzen.
- Ein paar Screenshots zu Devpac wären toll.

Wer hat Lust zu helfen? Unter http://retrobude.de/dokumente/amiga-assembler-buch/ ist der aktuelle Stand für jeden einsehbar. Interessenten würde ich dann Kapitel für Kapitel iterativ und inkrementell zukommen lassen (jeder sagt, was er gerne machen möchte, also Korrekturlesen, ins Format bringen, erweitern, ...), das dann mergen und weitergeben und so könnte am Ende ein umfangreiches und aktuelles Assembler-Buch für den Amiga stehen.

Danksagungen

Ich danke Wolfgang Hosemann und Ao Toprak für die Durchsicht.

Konventionen

In den Assembler-Listings schreibe ich die Mnemonics (`add`, `move`, ...) und Register klein, im Text konsequent groß (`ADD`, `D7`, ...).

Zahlen im Computer

Bevor wir in die Tiefen der Rechnerstruktur eindringen, geheimnisvolle Strukturverweise entdecken, und die CPU zum Qualmen bringen, muss ein etwas langweiliges Thema abgehakt werden. Die Zahlensysteme. Vorgestellt werden die Systeme, die der Computer verarbeitet, und die Zahlen mit denen er rechnet.

Stellenwertsysteme

Wenn man mit dem Computer arbeitet, wird man zwangsläufig mit zwei wichtigen Stellenwertsystemen konfrontiert. (Duden Informatik sagt zu Stellenwertsystem: “System zur Darstellung von Zahlen durch Ziffern, bei denen der Wert einer Ziffer von der Stelle abhängt, an welcher sie innerhalb einer Zahl geschrieben ist“.)

Diese beiden Systeme sind das Hexadezimalsystem (oder Sedezimalsystem) und das Dualsystem (oder Binärsystem). Unser allgegenwärtiges Stellenwertsystem, das Dezimalsystem oder Zehnersystem, bietet die Ziffern 0-9 zur Darstellung von Zahlen. Man sagt, wir haben die Basis von 10, da für jede Zahl von 0-9 eine verschiedene Ziffer existiert. Wir es gewöhnt sind mit dem Zehnersystem umzugehen, ist uns meist die Arbeitsweise, wie wir z. B. addieren oder multiplizieren, unbewusst. (Warum gerade 10 die Basis für zehn verschiedene Zahlen ist, und wir mit Dezimalzahlen arbeiten, ist wohl durch die 10 Finger erklärt, die ein Mensch normalerweise hat. Es vereinfacht auch das Fingerrechnen. Hintergründe zu dieser Theorie erfahren Leser in den Büchern „Ein Himmel voller Zahlen: Auf den Spuren mathematischer Wahrheit“ von John D. Barrow, „Number Words and Number Symbols: Cultural History of Numbers“ von Karl Menninger.)

Stellenwertsysteme

Wir können alle unsere Dezimalzahlen, welche aus Einern, Zehner, Hunderten, und weiteren 10^x-Termen aufgebaut werden. Die Dezimalzahl errechnet sich dann aus der Summe von Einzelpotenzen.

Die Zahl 123 ließe sich dann folgendermaßen in ihre Glieder zerlegen:

$123 = 1*10^2 + 2*10^1 + 3*10^0$

Der Mensch hat seine Finger und evtl. seine Zehen zum Zählen. (10+10=20? Oh, ein neues Stellenwert-System! PS.: Früher verwendete man auch die 12 als Basis).

Der Computer jedoch arbeitet nach einen anderen Prinzip: dem Vorhandensein von Strom. (Denn wie sollte der elektrische Strom die Finger ersetzen?) Der Rechner kennt zwei Zustände, ob Strom fließt, oder eben nicht. Er kennt daher nur zwei Werte (binäre Null und binäre Eins), oder vielleicht übertragen, nur einen Finger. Dies kennt jeder von der Schule, denn beim Aufzeigen, habe ich ja etwas zu sagen, eine Information also, die abgefragt werden kann. Melde ich mich nicht, habe ich diese Information auch nicht. (Der Computer kann natürlich nicht sagen, dass er keine Lust hat!). Wenn Strom vorhanden ist, ist das gleichzusetzen mit einem Signal, man bezeichnet daher den Zustand als "High" oder "True" (wahr). Entsprechend dem Nicht-Signal als "Low" oder "False" (falsch). Eine Einheit, die entweder Null oder Eins sein kann, nennt man Bit (“BInary digiT“) und ist die “Bezeichnung für die kleinste Darstellungseinheit für Daten in binärer Zahlendarstellung“ (Duden Informatik).

Mit den zwei Zuständen können wir analog zum Dezimalsystem natürlich auch Zahlen darstellen. Das Stellenwertsystem, das mit diesen zwei Zuständen arbeitet, heißt, wie eingangs zu diesem Kapitel schon erwähnt binäres Zahlensystem (oder Dualsystem).

Binäre und Hexadezimale Zahlen

Mit allen Zahlensystemen lassen sich Zahlen darstellen, allerdings ist der Anblick für uns ziemlich ungewohnt. Da nach 9 Zahlen unser Zeichenvorrat erschöpft ist, müssen wir nun zwangsläufig größere Zahlen anders zusammensetzen. Da wir bei dem Dezimalsystem 10 Ziffern haben, bietet es sich an, die Null als Nachfolger von der Neun anzusehen und die Zehnerstelle um den Wert eins zu erhöhen. Der Übergang von der höchstmöglichen Zahl (die 9) zurück zum Anfang (Null) nennt man Übertrag. Bei den Binären Zahlen kommt nach Null die Eins, und nach dem Übertrag von Eins nach Null geht’s wieder von vorne los. Wenn wir den Übertrag mit einbeziehen, können wir auch auf diese Weise Zahlfolgen generieren. Zur Veranschaulichung soll die Tabelle dienen.

Bei den Hexadezimalsystem ist nicht 2 die Basis, sondern 16. Da wir aber nur 10 numerische Zahlen haben, müssen wir Zahlen durch irgendwelche anderen Zeichen ergänzen. Wir nehmen daher Buchstaben aus dem Alphabet dazu. Es scheint sinnvoll die Buchstaben von A-F zu benutzen.

Es hat sich eingebürgert, das man zur Kennzeichnung von Hex-Zahlen ein `$` und bei Bin-Zahlen ein `%` vor den Wert schreibt. Besonders das Dollarzeichen für Hex-Zahlen kann von Computer zu Computer unterschiedlich sein. Der Schneider CPC und der Archimedes nutzen das `&`-Zeichen um Hex-Zahlen zu verwalten.

Table 1. Darstellung von Zahlen in den unterschiedlichen Systemen

Dezimal	Hexadezimal	Binär
0	0	0
1	1	1
2	2	10
3	3	11

4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
10	a	1010
11	b	1011
12	c	1100
13	d	1101
14	e	1110
15	f	1111
16	10	10000
17	11	10001
18	12	10010
19	13	10011
20	14	10100

Da diese Werte häufig benötigt werden, ist es sinnvoll, sich eine Tabelle zu besorgen oder ein kleines Programm zu schreiben, das die Werte in Binärer- und Hexadezimaler-Form ausdruckt. Ein Programm zur Umrechnung von diesen Werten wird in einem späteren Kapitel besprochen.

An dieser Stelle sei noch auf einen weiteren Zusammenhang zwischen Bit und Hexadezimalen Zahlen hingewiesen: Vier Bit bilden eine Hexadezimale Zahl. Das macht das Lesen ziemlich einfach. Vielleicht wurde das schon in der Tabelle bemerkt.

Wie wir gesehen haben, reiht man im Binärsystem die einzelnen Bit aneinander und erhält somit verschieden große Zahlen. Durch die Zunahme von einem Bit verdoppelt sich der Zahlenbereich. Haben wir also ein Bit zur Darstellung, so können wir auch nur zwei Zahl darstellen (wenn wir Null als Zahl mitrechnen). Haben wir schon zwei, so verdoppelt sich der darstellbare Bereich auf vier Zahlen. Mit drei Bit erreichen wir schon acht Zahlen.

Table 2. Mit wie vielen Bit man wie große Zahlen darstellen kann

Anzahl der Bit	Maximal Darstellbare Zahl
1	2-1
2	4-1
3	8-1
4	16-1
5	32-1
6	64-1
7	128-1
8	256-1
9	512-1
10	1024-1
12	2048-1
13	4096-1
14	8192-1

Rechnen mit den Zahlen ist natürlich auch möglich, man muss hierbei nur besonders auf den Übertrag achten.

```
1234+976        %01101011+%1111011      $345+$f4a

   1234            01101011                345
  + 976          +  1111011              + f4a
  -----           --------               -----
   2210            11100110               128f
```

Umrechnung von Dezimal- in Binärzahlen

Da die dezimale Zahl aus Zehnerpotenzen zusammengesetzt ist, müssen wir sie nun als Summe von Zweierpotenzen darstellen. Wir wollen dies am Beispiel der Zahl 100 nachvollziehen.

```
100 = 1*2^6 + 1*2^5 + 0*2^4 + 0*2^3 + 1*2^2 + 0*2^1 + 0*2^0
    =  1*64 +  1*32 +  0*16 +   0*8 +   1*4 +   0*2 +   0*1
    =    64 +    32 +     0 +     0 +     4 +     0 +     0
```

Im praktischen Leben sieht das dann so aus: Aus der zuletzt aufgelisteten Tabelle suche ich die höchst mögliche Zahl der Reihe nach aus, die in die gesuchte hineinpasst. Das setze ich bis zum Ende fort. Das Ergebnis, ob eine Zahl hineinpasst oder nicht quittiere ich mit einer Eins oder Null. Wählen wir die Zahl 1000.

Bit	Zahl	Rechenschritt	Resultat
10	1024	1024 passt nicht in 1000	0
9	512	Rest von 1000 ist 488	1
8	256	256 passt in 488, Rest 232	1
7	128	128 passt in 232, Rest 104	1
6	64	64 passt in 104, Rest 40	1
5	32	32 passt in 40, Rest 8	1
4	16	16 passt nicht mehr in 8	0
3	8	passt genau	1
2	4	muss null sein	0
1	2	muss ebenso null sein	0
0	1	ist gerade, daher auch Null	0

Die Binärzahl zum Äquivalent 1000 ist `%01111101000`.

Bessere Möglichkeit zur Berechnung

Die vorgestellte Methode ist zwar gut, doch bei großen Zahlen kostet sie einiges an Kopfspeicher, da alle Zweierpotenzen behalten werden müssen. Daher will ich noch eine weitere Möglichkeit zur Umrechnung vorstellen. Das Prinzip dabei ist folgendes: Die Zahl (wir wählen wieder 1000) wird immer durch 2 dividiert, und das Ergebnis wird wiederum als Ausgangszahl für die weitere Division benutzt. Ist die Zahl durch 2 teilbar, also Gerade, soll eine Null als Binärwert gelten. Zum Schluss werden die Zahlen in umgekehrter Reihenfolge hingeschrieben.

```
1000 : 2 = 500  (Ist durch 2 teilbar, daher 0)
 500 : 2 = 250  (0)
 250 : 2 = 125  (0)
 125 : 2 = 62   (nicht durch 2 teilbar, daher 1)
  62 : 2 = 31   (0)
  31 : 2 = 15   (1)
  15 : 2 = 7    (1)
   7 : 2 = 3    (1)
   3 : 2 = 1    (1)
   1 : 2 = uninteressant (1)
```

Umgekehrt hingeschrieben ergibt sich für die Dezimalzahl 1000 das Äquivalent `%1111101000`.

Dieses Prinzip eignet sich sehr gut für die Rechner-Maschine, denn dieser kann die Division durch zwei sehr schnell und einfach durchführen.

Einführung in Assembler

Nun soll es so richtig losgehen. Wir wollen zunächst einmal klären, welche Komponenten in einem Rechner stecken. In den folgenden Kapiteln werden wir auf den Mikroprozessor, das Herz des Computers, und auf den Speicher näher eingehen.

Was ist ein Mikroprozessor?

Der Mikroprozessor ist eine integrierte Schaltung (IS), oder auch Integrated Circuit (IC) genannt. Die integrierten Schaltungen erlauben es, mehrere tausend Bauteile, wie Transistoren, Kondensatoren, Widerstände und Spulen auf kleinsten Raum unterzubringen. Die erste Integrierte Schaltung, die 1958 von Jack Kilby gebaut wurde, konnte 4 Transistoren beherbergen. Durch die vielen Jahre hindurch ist die Chip-Technologie heute soweit, dass 1,2 Millionen (1200000) Transistoren (soviel hat der MC68040) auf einem Fingernagel untergebracht werden können.

Ein Mikroprozessor ist etwas besonderes, denn er ist ein IC, der programmierbar ist und Befehlsfolgen abarbeiten kann. So sind Mikroprozessoren nicht nur auf dem Bereich der Computer beschränkt, sondern auch in Taschenrechnern, Radios, Autos, Fernsehern, Videorecordern und Spielautomaten zu finden.

Die Präsentation der ersten Prozessoren gelang 1970 den Herstellern Intel (Internationale Elektronik oder auch Intelligente Elektronik) und Texas Instruments. Die damaligen Prozessoren gehörten der ersten Mikroprozessorgeneration an.

Motorola begab sich mit dem 6800 (der Vorgänger vom MC68000) in die zweite Generation. Die dritte Generation ist etwa 1976 anzusetzen. Sie brachte Prozessoren wie den 6502 (ist im VC 20, Vorgänger vom C-64, eingesetzt), der von der Firma MOS Technology stammt (eine Firma, die sich Commodore später unter den Nagel gerissen hat) und den Z80 von Zilog (eine von Intel abgespaltene Gruppe) hervor. Im Jahre 1982 waren der 6502 und der Z80 die am weitesten verbreiteten Mikroprozessoren.

Nun sind wir mit dem MC68000 in der vierten Generation. Dieses Buch beschäftigt sich mit der Programmierung des MC68000 auf dem Amiga Computer.

Was ist Assembler?

Assembler ist die Grundsprache jedes Computers. Doch war da nicht noch was von Maschinensprache? Sollte nicht sie die Grundsprache sein? Gibt es denn da einen Unterschied oder ist es vielleicht dasselbe mit einer anderen Bezeichnung? Genaugenommen gibt es da einen kleiner Unterschied, denn wenn wir von Assembler reden, meinen wir meist schon eine kleine Programmiersprache. Während in der Maschinensprache nur mit Binärzahlen programmiert wird, so bringt uns Assembler mit seinen Befehlen einen großen Schritt voran. Diese Befehle werden Mnemonics genannt, was im englischen für “Gedächtnisstütze“ steht und seinen Ursprung im griechischen Wort *mnēmoniká* “Gedächtnis“ hat. Man muss sich nicht mehr mit elendig langen Zahlenkolonnen herumärgern, sondern kann jeder dieser Zahlenreihe auch Namen geben und ich finde, das RTI leichter zu merken ist als $4ef9.

Assembler ist bei jedem Computer anders. Wer also schon mit dem C-64, den kleinen Ataris, den Schneider CPC, oder dem Sinclair Spectrum (Taschenrechner mit Tastatur) in Assembler programmiert hat, und nun diese Kenntnisse auf dem Amiga übernehmen will, der ist aufgeschmissen. Das heißt natürlich nicht, das gesammte Wissen wird unwichtig: wer die Denkweise einmal beherrscht, kann leichter auf andere Prozessoren umsatteln.

Übrigens, noch zum Thema C-64 Prozessor: Der Hauptprozessor, der im C-64 das Rennen macht, ist aber auch im Amiga eingebaut, jedoch da hat er auf der Platine die ehrenwerte Aufgabe als Tastaturprozessor. (Ganz schön runtergekommen, wenn man bedenkt, zu welchen Ruhm er dem C-64 verholfen hatte!)

Was bietet uns der MC68000

Eines der wichtigsten Fakten, die der MC68000 aufzuweisen hat, ist die Anzahl und Breite von Daten- und Adressregistern. Das bringt uns zum Begriff *Register* und das Wort wird noch häufiger fallen. Ein Register ist ein kleiner Speicherplatz auf dem Prozessor. Auf diesen Speicherplatz ist schnell zuzugreifen, da die Daten “nicht so weit entfernt sind“.

Da es praktisch und schneller ist, mit Registern als mit Speicher zu arbeiten, haben die Entwickler nach dem Motto „Darf es auch etwas mehr sein?“ beschlossen, mit Registern nicht zu geizen. Eine heraus stechende Eigenschaft ist nun die Anzahl der Register. Insgesamt sind es 16(!) an der Zahl.

Der 68000 hat acht Daten- und acht Adressregister mit der Breite von je 32 Bit. Dies soll in der Skizze verdeutlicht werden.

```
1. Datenregister 31 30 29 28 27 26 25 ..... 5 4 3 2 1 0
.
.

8. Datenregister 31 30 29 28 27 26 25 ..... 5 4 3 2 1 0
```

und

```
1. Adressregister 31 30 29 28 27 26 25 ..... 5 4 3 2 1 0
.
.
8. Adressregister 31 30 29 28 27 26 25 ..... 5 4 3 2 1 0
```

In jedes dieser Register lässt sich ein Wert reinschreiben. Natürlich kann der Inhalt des Registers wieder in an anderes Ziel übertragen werden. Auch kann der Programmierer diese Register untereinander verknüpfen, um Rechenoperationen möglich zu machen.

Die wichtigste Aufgabe eines Prozessors ist seine Fähigkeit Daten zu verarbeiten, und dies geschieht mit Hilfe von Registern. Das Wort Register wird uns daher im ganzen Buch begegnen.

Die Datenregister werden mit einem `D` und die Adressregister mit einem führenden `A` bezeichnet. Da wir 8 von jeder Sorte haben, und der Computer bei Null zu zählen beginnt, bezeichnet man die Datenregister mit `D0`, `D1`, `D2`, ..., `D7` und ebenso die Adressregister von `A0`, `A1`, `A2`,... , `A7`. Diese große Anzahl von Registern, die schnell verarbeitet werden können, muss man natürlich nutzen. So sind Compiler darauf spezialisiert mehr oder weniger diese 15 Register zu nutzen (das `A7`-Register ist schon für andere Zwecke besetzt, für den sogenannten Stack-Pointer). Falls Compiler dies nicht tun und nur einen schlechten Programmcode erzeugen, weil sie z. B. verschwenderisch mit Registern um sich werfen, gibt es noch Optimierer. (Wenn etwa ein Register ein Wert enthält, und dieser wird noch einmal in dieses Register geschrieben — zwei gleiche Zeilen --, kann eine Zeile verschwinden.) Ein Optimierer kann ein Assemblerprogramm scannen, und durch geschickte Algorithmen alle Register so gut wie eben möglich ausreizen. Je besser die Registerwahl ist, desto schneller läuft ein Programm ab, da lästiges Zwischenspeichern in den Hauptspeicher erspart bleibt. Speicherzugriffe sind immer langsam, das sollte bei der Programmoptimierung immer beachtet werden.

Prozessoren wie der Intel oder C-64 Chip haben den Vorteil der vielen Register nicht, die zudem alle gleichberechtigt verknüpft werden können. Bei den alten Prozessoren gibt es eine Rangordnung. So gibt es z. B. ein Register, den Akkumulator (AX-Register beim Intel), über den nur alle Rechnungen laufen können, oder ein Register, das nur zum Zählen (CX-Regsiter beim Intel) gedacht ist. (An unsere PC Übersteiger: Ich möchte gar nicht erst wissen, was für ein Aufwand es sein muss, schon 2 Schleifen zu schachteln.)

Doch um einem Computer mit Hilfe eines Prozessors zum Laufen zu bringen, benötigt man noch eine Zelle, in welchem der aktuelle Programmstand drinsteht. Es ist der schon erwähnte Program Counter, kurz PC (So hat eigentlich jeder, der einen Computer besitzt einen PC). Er wird bei der Programmausführung immer wieder der aktuellen Adresse angepasst.

Die Befehle des Prozessors

Um den Prozessor zu programmieren, d. h. dessen Register und Speicher zu verändern, benötigen wir die schon erwähnten Befehle bzw. Mnemonics. Der 68000 kennt 56 Grundbefehle. Diese unterscheiden sich durch ihre verschiedenen Aufgaben.

- *Datentransportbefehle*. Sie bewegen Daten zwischen Registern und Speicher, Speicher und Speicher und Speicher und Register. Der Aufbau des Speicher wird im nächsten Kapitel ausführlicher behandelt.
- *Arithmetische und Logische Befehle*. Durch eine eingebaute ALU (Arithmetic-Logic-Unit; Recheneinheit) ist der Prozessor in der Lage, Werte zu berechnen. Ihm stehen die vier Grundrechenarten und darüber hinaus die Verknüpfungsoperationen zur Verfügung.
- *Sprungbefehle*. Mit ihnen kann man Unterprogramme aufrufen oder anspringen.
- *Vergleichsbefehle*. Mit den Vergleichsbefehlen können Werte verglichen werden. So z. B. ob bei einem Vergleich von zwei Zahlen die erste größer, kleiner oder gleich war.
- *Rotations- und Schiebebefehle*. Mit ihnen lassen sich zum größten Teil die langsamen Multiplikations- und Divisionsroutinen ersetzen. Aber auch Scroll-Routinen benutzen zwangsläufig diese Befehle, denn mit der Hilfe der Befehle lassen sich Werte um bestimmte Positionen verschieben.
- *Bitmanipulationsbefehle*. Sie erlauben Veränderungen der Bit in den Registern oder im Speicher. Der MC68020 kennt auch erweiterte Bitmanipulationsbefehle, die Bitfeldoperationen, die eine neue Befehlsart bilden würden (dies ist für uns aber erst einmal Quizwissen!).
- *Spezialbefehle*. Unter die Spezialbefehle fallen z. B. diejenigen, die das Anlegen eines virtuellen Stacks (auch Quizwissen!) oder den Umgang mit Unterbrechungen (Interrupts), sinnvoll im Multitasking Betriebssystem, erlauben. Dies hört sich natürlich sehr kompliziert an, ist es aber eigentlich gar nicht! Die meisten dieser Befehle sind privilegiert, d. h. dass bei einem Aufruf der Rechner im Normalfall abstürzt. Man muss erst in einen speziellen Prozessormodus umschalten, um alle Befehle nutzen zu können.

  Normalerweise befinden wir uns im USER-Modus und müssen in den SUPERVISOR-Modus überwechseln, um in den Genuss aller Befehle zu kommen, doch das warum und wie kommt später bei der Einleitung in das Betriebssystem.

Der MC68020 erweitert diese Spezialbefehle insofern, dass Coprozessorsteuerungsbefehle hinzukommen, so z. B. Befehle, die den Matheprozessor ansprechen.

Die Zahlenbereiche

Natürlich kann der Amiga nicht unendliche große Zahlen verarbeiten. Ihm sind durch die Registerlänge von 32 Bit Grenzen gesetzt. Doch sollten wir nicht betrübt sein, der C-64 hat nur einen Zahlenbereich, der durch 8 Bit gegeben ist, also Zahlen bis 256. 32 Bit Zahlen dagegen können den Zahlenbereich von Null bis 2^{32} = $ffffffff = $4{,}9*10^9$ darstellen. Für viele Anwendungen mit Ganzzahlen ist das ausreichend.

Weil man nicht immer von 32 Bit Zahlen reden möchte, gab man ihnen einen anderen Namen: Long oder Longword. Der Name wird gleich noch etwas deutlicher.

Eine weitere Unterteilung umfasst 16 Bit, auch Word genannt. Daher auch der Name Long-Word, weil sie doppelt so groß sind. Wir sollten uns merken, wenn wir rechnen oder Werte benötigen, falls es geht, auf Longs zu verzichten. Das wird in den meisten Fällen gelingen, denn die 32 Bit Werte werden nur bei Adressen voll benötigt (und Adressen kann man sowieso nicht multiplizieren), und Words reichen zum Arbeiten voll aus. Der Bereich der Words umfasst 16 Bit. Es können also Zahlen bis 2^{16} = %1111111111111111 = 65536 = $ffff gebildet werden. Doch oft ist auch dieser Bereich zu groß und er kostet immerhin noch 16 Bit. Somit kommen wir nun zur ursprünglichsten Speichereinheit: Dem Byte.

Das Byte ist was Besonderes

Ein Bytge umfasst genau 8 Bit und stellt somit ein halbes Word da. Es umfasst einen Zahlenbereich von 0-255 und hat gegenüber dem Words den Vorteil, dass es weniger Speicherplatz braucht.

Doch haben Bytes auch einen historischen Kontext. Der ein oder andere wird wahrscheinlich schon von ASCII Zeichen oder vom ASCII Code gehört haben. ASCII ist die Abkürzung für „American Standard Code for Information Interchange“. Dieser Code ist aus der Notwendigkeit zur Übertragung von Texten zwischen verschiedenen Computersystemen entstanden. Dieser ASCII Zeichensatz enthält zentrale Zeichen, Umlaute, Satzzeichen sowie Sonderzeichen (z. B. für die DFÜ oder den Drucker). Da wir für Englisch oder Deutsch höchstens auf 50 Zeichen kommen, warum denn ein Long oder Word verschwenden? ASCII-Zeichen passen wunderbar in ein Byte herein. Da wir aber bei nur 50 Zeichen noch so unsere 205 übrig hätten, füllt man den Zeichensatz noch mit ein paar Sonderzeichen und Grafikzeichen aus, und schwupp, hat man seine 255. Dies heißt natürlich nicht, dass immer diese Anzahl von Zeichen vorliegen muss. Warum in einigen Fällen Platz verschwenden. So gibt es auch eine Spar-Version des Setmap/d, denn so spart man auch seine Sonderzeichen, die in der Deutschen Sprache nun mal wenig gebraucht werden. Wollen wir also Zeichenketten (Strings) bearbeiten, so tun wir dies am besten mit Bytes.

Leider ist die Bearbeitung von Bytes nicht schneller als die von Words. Wie kann das sein? Nun, bei einem Speicherzugriff werden immer 16 Bit gleichzeitig gelesen, unserer 8 Bit Zahlen kommen somit auch über diesen Bus. Die Verarbeitung ist also nicht schneller, sondern es fallen lediglich 8 unbenutzte Bit weg.

Table 3. Zusammenfassung der Datentypen

1 Byte	entspricht 8 Bit
1 Wort (Word)	entspricht 16 Bit, 2 Bytes
1 Langword (Long)	entspricht 32 Bit, 2 Words, 4 Bytes

Der Speicher

Was ist Speicher?

Man benötigt Speicher um Informationen und Daten aufzubewahren. Speichersysteme können Informationen auf Speichermedien Sichern, wobei man zwischen zwei grundlegenden Speichermöglichkeiten unterscheidet.

Externe Speicher

Externen Speicher sind zum Archivieren von Daten gedacht, die nach einem Stromausfall immer noch vorhanden sein müssen; Informationen, die z. B. die Bank bereithält, müssen auf externen Speichern gesichert werden. Externe Speicher stehen flüchtige internen Speicher gegenüber. Beispiel für externe Speicher sind das Diskettenlaufwerk, die Festplatte, die Datasette (man träumt von alten Brotkastenzeiten), dem Streamer (Kassetten in Kleinformat), die Lochstreifenkarten (man erinnert sich nur noch an FORTRAN Programme), die Opto-Disk (oder auch CD genannt, noch etwas lahm) und die Magneto-Opto-Disk (beschreibbare CD, die mittels Laser gelesen wird, und mit Hilfe der herkömmlichen Art beschrieben wird).

Interne Speicher

Auf der anderen Seite sind da die schnellen *Internen Speicher*. Es sind Integrierte Schaltungen (Halbleiter-Speicher), die logische Zustände (0 oder 1), unsere Bit, speichern können. Bei den internen Speichern unterscheidet man das Speicherverfahren ebenso, wie bei den Externen. Es gibt statischen RAM und dynamisches RAM. Beides hört sich nicht schlecht an, oder? Nun, die statischen RAMs werden durch die sogenannten Flip-Flops (auch FF oder bistabilen Multivibratoren in Elektronik-Freak-Kreisen genannt) realisiert. Flip-Flops sind einfache elektronische Schaltungen, bei denen das Eingangssignal gesichert wird. Diese Information, ein Bit, kann so lange gespeichert werden, bis der Strom futsch ist. In Normalgebrauch braucht er aber nicht viel Energie, nur 0,000001 Watt und mit einem Akku ist dieser Flip-Flop auch bei Stromausfall leicht zu versorgen.

Doch alles hört sich so gut an, wenn es da nicht noch einen kleinen Nachteil gäbe. Die Flippis brauchen auf dem Integrierten Chip sehr viel Platz (Es sind immerhin schon ein paar Transistoren drauf). Man entwickelte somit eine neue Art von Speicherchips, die dynamischen RAMs. Sie bestehen im Gegensatz zu den Flip-Flop-Speichern aus Kondensatoren, die geladen oder entladen unsere Bit darstellen.

Das ist ganz schön, wenn da nicht schon wieder ein Haken wäre. Die Kondensatoren verlieren durch sogenannte Leckströme einen Teil ihrer Energie, und müssen deshalb immer aufgefrischt werden. Diesen Auffrisch-Vorgang, der ca. alle 2 ms vonstatten geht, wird Refresh genannt. Der Computer führt sogenannte Refresh-Zyklen durch, um seine RAM-Bausteine mit Strom zu versorgen. Natürlich kann während dieser Zeit auch kein Speicherzugriff geschehen.

Trotz der höheren Energie (ca. 1000-mal), die für den dynamischem RAM aufgebracht werden muss, ist ohne ihn die Mega-Chip-Technologie unvorstellbar. Somit lassen sich auf gleichem Raum sehr viel mehr Informationen speichern.

RAM, ROM und WOM

Die internen Speicher werden ihrerseits untergliedert, ob sie Daten sichern können, oder nur Daten zur Verfügung stellen sollen. Die meisten internen Speicher können Daten sichern, und sie können wieder gelesen werden. Man bezeichnet diese Art von Speicher auch RAM (Random Access Memory) Speicher.

Betriebssysteme, wie unser Kickstart, ist in einem Nur-Lese-Speicher „eingebrannt“, das auch ROM (Read Only Memory) genannt wird. Aus dem ROM können Informationen nur gelesen werden, denn des Betriebssystem soll vor Überschreibungen sicher sein.

Im ersten Amiga, dem A1000, der am 23. Juli 1985 vorgestellt wurde, wollte man wegen seinem fehlerhaften Betriebssystem dies noch nicht ins ROM übertragen. Die bei dem Kauf des Computers beigelegte Diskette musste vor jedem Start in den Speicher des Computers geladen werden. Diese Zeit prägte die Begriffe Booten und Kickstart. Das dieses Laden ziemlich nervig ist, war klar, aber somit konnten immer neue Versionen auf Diskette herausgegeben werden, die dann nach und nach immer Fehlerfreier wurden. Man bediente sich hier eines Zwischenstückes zwischen RAM und ROM, um das Betriebssystem im Speicher zu sichern. Das WOM (Write Once Memory), das nur einmal, meist nach dem Einschalten und booten, beschrieben werden konnte, und das nach diesem Vorgang nichts mehr an sich heran ließ.

Nach wie vor sind im Amiga Betriebssystem Fehler, auch Bugs genannt, vorhanden. Um diese Fehler in den Systemroutinen zu korrigieren kann man den DOS-Befehl Patch benutzen. Wie dieser Befehl genau verwendet wird, werden wir im Exec-Kapitel sehen.

Doch in allen Fällen ist es sehr aufwändig RAM-Speicher als Dauerspeicher für Daten zu verwenden. Jedoch haben die externen Speicher den Nachteil, dass sie immer langsamer als Interne sind. Sie haben aber den Vorteil der Datensicherheit und Kapazität, der Menge der Informationen, die gesichert werden kann. Doch sind im Zuge der Technik schon Festplatten entstanden (erstmals für den Atari glaube ich), die mit 3 ms schon eine atemberaubende Geschwindigkeit haben, und mit Mega, Giga Speichern auch Filme und Musik CD aufnehmen könnten. Dies sind zwar für den RAM keine Konkurrenzzeiten, der mit 20 ns (Nanosekunden) fix dabei ist, jedoch ein Schritt zum schnellen Zugriff auf riesige Daten-Bänke.

Datenorganisation im Speicher

Die Daten bzw. Informationen müssen irgendwie im Speicher abgelegt werden. Durch eine Adresse können wir an alle beliebigen, zur Verfügung stehenden Speicherstellen gelangen. Da wir direkt über die Adresse an unser Ziel kommen, nennt man den Speicheraufbau auch durchgehend.

Da Adressregister vom MC68000 zwar 32 Bit breit ist, aber nur 24 Bit genutzt werden, können wir nicht 4 Giga Byte adressieren. (Der Nachfolgerprozessor MC68030 kann dagegen volle 32 Bit nutzen.) Die maximale Größe des Speichers wäre demnach also 2^{24} = 16777216 Bytes (16 MB). Der Speicher des Amigas fängt bei Null an und hört bei $ffffff (16777216 in Dezimal Darstellung) auf. Jedem Byte ist also eine Adresse zugeordnet. Wenn unser Programm ab dem Speicherinhalt 1000 bis 2222 zu finden ist, hat es eine Länge von 1222 Bytes.

Da unser 68000 aber auch Words und Longs verarbeiten kann, müssen sie ebenfalls aus Bytes zusammengesetzt sein. Der Vorteil eines 16 Bit Busses liegt darin, das 2 Bytes auf einmal gelesen werden können. Longs haben da einen Nachteil. Der Zugriff ist langsam. Woher kommt das? Es liegt daran, dass unser lieber Chip gar kein richtiger 32-Bitter ist, sondern einen Datenbus (Der Bus, durch den die Daten vom Speicher in die CPU [oder umgekehrt, ist gehüpft wie gesprungen] gelesen wird) von nur 16 Bit Breite hat. Wie schon oben beschrieben, ist der Motorola Chip in aller Beziehung ein Superchip. Denn um die Kosten möglichst gering zu halten (was jeder in der Produktion versucht), einigte man sich darauf, den Datenbus nur 16 Bit breit zu machen. Die Long-Daten, die dann in den Speicher geschrieben werden, oder aus dem Speicher geholt werden, werden aufgespalten in zwei 16 Bit Werte. Natürlich merkt man als Anwender nichts davon.

Speicheraufbau

Den allgemeinen Aufbau des Hauptspeichers sollen die folgenden Skizzen verdeutlichen:

Der Speicher fängt bei Byte $0 an, und endet bei Byte $ffffff.

```
Word $000000      Byte $000000, $000001
Word $000002      Byte $000002, $000003
.
.
Word $FFFFFC      Byte $FFFFFC, $FFFFFD
Word $FFFFFE      Byte $FFFFFE, $FFFFFF
```

Bei Long-Zahlen sieht die Sache noch etwas anders aus. Das Long Word wird in obere und untere Langwordhälfte unterteilt.

```
Longwort 0    obere Longworthälfte      Word $000000   Byte 0
                                                      Byte 1
              untere Longworthälfte                   Byte 2
                                                      Byte 3
Longwort 1    obere Longworthälfte      Word $000002   Byte 4
                                                      Byte 5
              untere Longworthälfte                   Byte 6
                                                      Byte 7
```

Aufgrund des dargelegten Speicheraufbaus sollten wir **nie** versuchen, Words oder Longs aus ungeraden Speicherzellen rauszuholen oder an ungeraden Adressen Word oder Longs schreiben. Andernfalls droht uns der rote Pirat (Guru). Er meldet sich mit einem `0000 0003`. Die Drei deutet an, dass wir versucht haben, auf ein ungerade Speicheradresse zuzugreifen. Wenn wir das wollen, müssen wir immer Bytes benutzen.

Die Konsequenz ist, dass auch die Assemblerbefehle nur auf geraden Adressen liegen dürfen. Jeder Assemblerbefehl ist also mindestens 2 Bytes lang, die Bezeichnung 2-Byte-Maschine hat sich eingebürgert, ganz im Gegensatz zu 1-Byte-Prozessoren wie der Intels 8088 und der 6510 des C-64.

Durch den geraden Aufbau des Speichers, darf ein Long oder ein Word **nur** auf geraden Adressen liegen.
Ein Nicht-Befolgen dieser Regel bringt einen Guru, also einen Absturz, mit sich.

Die Speicherkonfiguration

Die Speicherkonfiguration, die nach dem Einschalten existiert, ist in der folgenden Grafik dargestellt. Die Gesamtheit des Speichers wird auch als Adressraum bezeichnet. Unser Adressraum ist 16 MB groß, so dass mit dem Speicher nicht gegeizt werden muss. Die Speicherspiegelungen machen dies deutlich.

$000000	512 KB Chip RAM (mit Fat Agnus)
$080000	Spiegelung des Chip-RAMs (mit Fat Agnus)
$100000	Spiegelung des Chip-RAMs
$180000	Spiegelung des Chip-RAMs
$200000	8 MB Fast-RAM-Bereich
$A00000	Für CIA s reserviert
$B00000	Reserviert
$C00000	512 KB Erweiterungsraum Fast RAM für A500 und A1000
$C80000	Zusätzliches Fast RAM für den Amiga 2000A
$D00000	Reserviert
$DC0000	Echtzeituhr
$DF0000	Custom-Chips
$E00000	Reserviert
$E80000	Bereich für Expansionsslot (Konfigurations-Bereich)
$F00000	Cartridge (ROM-Module)
$F80000	Spiegelung des Kickstart ROMs und WOM für A1000
$FC0000	256 KB Kickstart ROMs

Der Assembler

Devpac und Co

Die Überschrift ist vielleicht etwas irreführend, denn man versteht unter Assembler noch mehr, als nur die Programmiersprache. Ein Assembler ist auch ein Programm, das den Assembler Quelltext, den wir geschrieben haben und der in Textform vorliegt, in ein ausführbares Maschinenspracheprogramm umwandelt. Wie ein Assembler arbeitet, welche Befehle er versteht und übersetzt, wird in den letzten Kapiteln weiter vertieft.

Der Assembler, den die meisten nutzen, ist der Devpac Assembler von Hisoft (Kostenpunkt für 2.1 Version 140 DM), denn Seka's Assembler-Zeiten sind out. (Ich glaube jetzt schon PD.) Devpac hat einen komfortablen Full-Screen Editor, eigentlich alles, was man so zum Programmieren braucht. Doch kommen auch neue Assembler wie der Maxon-Assembler von MAXON (149,-DM), der Code-X Assembler, ASM-One und der O.M.A. 2.0 Makro-Assembler (180 DM), der auch 68030 und 68882 Programmierung unterstützt, auf den Markt. Da diese aber noch nicht so sehr verbreitet sind und recht neu sind, beschränke ich mich auf den Devpac Assembler. Um aktuell zu sein, will ich nicht verschweigen, das auch Hisoft eine völlig neue Version herausbrachte, Devpac III. Wie seine Konkurrenten bietet er OS 2.0 Programmieren neue Include-Dateien (Dateien enthalten Informationen über das OS,

wie Konstanten, Felder und Strukturen). Auch die Geschwindigkeit wurde verbessert, denn Devpac war nicht gerade der Schnellste.

Wir werden also Devpac hier als Assembler einsetzten. Wer jedoch einen anderen Assembler gekauft hat, nicht schlimm. Die meisten der anderen Assembler verfügen über denselben Befehlssatz, so dass von dieser Seite her keine Schwierigkeiten zu erwarten sind. Lediglich im Umgang sind sie geringfügig anders. Hier ist etwas Mut zum Ausprobieren gefragt, dann jedoch kommt man schnell dahinter, was in welcher Form verändert werden muss.

Es geht los

Nachdem wir uns mit dem Prozessor auseinandergesetzt haben, ist es nun an der Zeit, mit ihm umzugehen. Eine Einführung in Devpac erlaubt es uns, das Erlernte sofort auszuprobieren. Ich finde es sinnvoll, erst so ziemlich alle Assemblerbefehle kennenzulernen und auszuprobieren, dazu kleine Programme zu schreiben, und dann erst komplexere Programme zu verwirklichen. Dann gehen wir an das Betriebssystem heran, um größere Projekte ins Auge zu nehmen.

Einleben in Devpac

Dieses erste Unterkapitel soll uns helfen, den Devpac Assembler anzuwenden und dessen Arbeitsweise zu verstehen.

Devpac und Programme auf Speichermedien

Da Assembler eine sehr Guru- bzw. Absturz-freudige Programmiersprache ist, und nach einem Absturz die Programme im Regelfall leider verloren gehen, ist es sinnvoll bei jeder größeren Änderung den Quellcode (Assemblerprogramm in Textform, engl. source code) abzuspeichern. Das heißt nicht, dass bei jeder neuen zugefügten Zeile und einem Ausführen des Programms dies geschehen soll, aber es ist schon sinnvoll hin und wieder mal eine Kopie zu machen. Leider bietet Devpac kein automatisches Speichern an, welches erlaubt unser liebes Programm immer in regelmäßigen Zeitabschnitten zu sichern. Bei den neuen Assemblern ist dies aber der Regelfall, das sie diese Option erlauben. Auch mir geht es häufig so, dass ich denke “Diese eine Zeile noch, dann sichere ich“, und nach einem Ausführen jammere ich “Sch..., warum hab ich Depp das nicht gesichert??“ (Reg auf, Brüll, Heul). Doch im Anfangssatz sagte ich was von “im Regelfall verloren“, heißt das, dass es noch eine Chance gibt? Ich würde sagen ja, eine Kleine. Da das Programm im Speicher steht, gibt es noch die Möglichkeit den gesamten Speicher zu durchforsten, und nach bekannten Wörtern (z. B. Move, SECTION oder Bemerkungen) zu suchen. Die Wahrscheinlichkeit das Programm wiederzufinden, steigt mit der Speicherkapazität. Wenn wir einen Amiga 500 ohne Speichererweiterung haben, dann ist die Wahrscheinlichkeit Mittel. Haben wir einen mit Erweiterung, dann steigt sie, denn Programme werden nicht so gequetscht im Speicher gehalten. Wie man den Speicher durchsucht kommt gleich. Wer in Kick Pascal programmiert, der wird vielleicht das Programm RESCUE (Engl.: Rettung) zu schätzen wissen, denn es durchsucht auch den Speicher, und gibt dem Programmierer die Chance, sein evtl. gefundenes Programm, oder Programmsegmente abzuspeichern.

RAM und RAD als RAT

Da jedoch das Speichern auf Diskette sehr mühselig und dauern kann, und wahrscheinlich wenige über eine Festplatte verfügen, bietet sich noch die RAM Disk an. Doch diese hat den Nachteil, dass nach einem Absturz oder Reset der Speicherinhalt verloren geht. Was kann man tun. Oh, ich höre Rattern im Gehirn? Da war doch das Word “Reset“. Natürlich, die resetfeste Version der RAM-Disk, die RAD (Resetfeste Ram Disk). Wir speichern einfach unsere Programme auf diese ab. Zudem noch unseren Devpac-Assembler, denn dieser kann durch die Startup-Sequenz bei einem Neustart sofort von der RAD geladen werden. Allerdings muss ich dazusagen, dass auch diese Methode nicht 100% sicher ist, aber 90%. Auch ich arbeite mit der RAD, und kopiere mit Diskcopy die Diskette in den Speicher, Voraussetzung ist natürlich genügend Speicher. Dies ist schneller als eine “COPY #?“ Version, falls die Diskette prall voll ist. Und auch mit dieser Methode hatte ich einst einen Absturz, und ich musste mit Erschrecken feststellen: Alles war Futsch. (Das erfreut besonders nach Stundenarbeit.) Und wenn einmal die Reset-Feste-Disk nicht mehr existiert, dann ist auch nichts mehr im Speicher drin, denn dieser wird nach dieser Art von Absturz gelöscht. (Zu den Absturzarten später mehr.) Doch war ich immer mit der RAD zufrieden, und im Normalfall kann man nicht mit dem Löschen des Speichers rechnen, also mein Rat lautet: resetfeste RAM-Disk (RAD) einrichten. Wie das geht steht im DOS Handbuch. Wer Kickstart 1.3. aufwärts hat, der kann von der RAD-Disk auch booten, ein Vorteil, den niemand missen wird, wenn er einmal damit gearbeitet hat.

Starten von Devpac

Wir wollen zunächst die Möglichkeiten unseres Assemblers kennenlernen, da er uns die Arbeit mit Maschinensprache sehr stark vereinfachen kann. Jeder wird wissen, wie man ihn aufruft. Mit “Devpac“. Und mit Trennung eines Leerzeichens kann noch ein Quellprogramm gleich mit geladen werden, das dann sofort im Editor steht. Interessiert mich der Editor nicht, kann ich auch nur den Assembler von Devpac nutzen. Er heißt “genim2“. Der Aufruf ist dem von Devpac gleich. Ich kann also auch mit andern Editoren Quelltexte erstellen, was aber wenig sinnvoll ist, da immer wieder getestet werden muss, und der Assembler springt gleich die fehlerhaften Zeilen an, was eine unschätzbare Erleichterung ist.

Doch bei Compilern, wo ein Quelltext erstellt wird, muss dieser ja auch mal zwangsläufig in was ausführbares übersetzt werden. Man kann so den recht langsamen A68K PD-Assembler durch den wesentlich schnelleren genam2 austauschen. Zwar bestehen noch einige Schranken, doch im Großen und Ganzen kann man zufrieden sein.

Die Arbeitsweise eines Assemblers

Die Arbeitsweise eines Assemblers unterscheidet sich nicht wesentlich von anderen Übersetzern. Vergleichen wir ihn doch zunächst mit anderen Compiler-Programmiersprachen, denn auch der Assembler ist ein Compiler. Das Wort wird häufig missverstanden, denn ein Compiler ist nicht nur einfach ein Programm das aus irgendeiner Sprache (C, Pascal, Modula,..) ein ausführbares Maschinenspracheprogramm erzeugt. Es kann auch ein Programm sein, das aus einem BASIC-Programm ein Pascal Programm macht. Zur deutlicheren Unterscheidung nennt man diese Programme jedoch Compiler-Compiler.

1 Pass- und 2 Pass-Compiler

Doch zurück zu den normalen Compilern, die ausführbare Programme erstellen. Man unterscheidet im Normalfall zwischen Ein- und Zwei-Pass Compilern. Der Unterschied liegt darin, dass ein Ein-Pass-Compiler nur einmal durch den Programmtext gehen muss, um dann das entsprechende Programm ausführbar bereit zu haben. Das bringt natürlich den Vorteil der Geschwindigkeit mit sich, ein wichtiger Aspekt vor allen dingen bei langsamen 7 MHz Rechnern. Alte Pascal-Compiler sind z. B. solche 1-Pass-Compiler. Man erkennt sie oft am einfachen Programmaufbau. Alle Unterprogramme oder Variablen, die benutzt werden, müssen am Anfang deklariert werden. Die einzige Ausnahme bildet der `FORWAD` Befehl in Pascal. Zwei-Pass-Compiler jedoch gehen zweimal durch das Programm, und benötigen daher eine höhere Compilezeit. Der Vorteil liegt eindeutig darin, das die Prozeduren nicht immer am Anfang stehen müssen, sondern auch am Ende stehen können. In Bezug auf die Prozeduren kann hier mit der Top-And-Down-Methode gearbeitet werden, wobei kleinere Unterprozeduren immer unter den Hauptprozeduren stehen. Ein C-Compiler könnte nie ein Quelltext mit einem Pass in ein ausführbares Programm umsetzen. Assembler ist genau so eine Sprache,

wo man nur mit Hilfe von 2-Pass-Durchgängen ans Ziel gelangt. Man hat sich darauf geeinigt, Unterprogramme an das Ende zu setzen, und danach die Variablen. Ein 1-Pass-Assembler wäre unmöglich, da Unterprogramme häufig aufgerufen werden, wenn sie noch nicht definiert sind, und Variablen immer unten definiert werden. Woher sollte der Assembler also wissen, wo das Unterprogramm anfängt, er kennt die Adresse ja nicht. Und somit hat man die 2-Pass-Methode eingeführt. Das Verfahren des Assemblers ist in groben Zügen folgendermaßen: Im ersten Pass wird das Programm auf syntaktische Korrektheit untersucht, d.h. ob alle Befehlswörter, die der Assembler kennt, richtig geschrieben sind, oder ein Fehler vorliegt. Es sind folgende Fehlerquellen denkbar: Werden Variable- oder Labelnamen geschrieben, die nicht zulässig sind (zu lang, nicht erlaubte Sonderzeichen); reicht evtl. der Speicher zum Assembliervorgang nicht aus; will man ein Programm nachladen, das nicht existiert; usw.? Im Zweiten Pass liegen nun alle Daten vor, und das Programm kann generiert werden. (Wie man einen Assembler selbst programmiert wird in einem der letzten Kapitel diskutiert.)

Findet der Assembler im Pass-Durchlauf z. B. einen Speichermanipulationsbefehl, so wird die zugehörige Bytefolge (früher in Maschinensprache lange Zahlenkolonnen, die man auswendig lernen durfte), die sogenannten Opcodes, in das Programm übertragen. Alle Befehle müssen gerade sein. Daher ist die kleinste Länge, die ein Befehl haben kann, 2 Byte.

Variablen

Ohne ein Programm zu schreiben, das Speicherzellen mit Assemblerbefehlen selbst ändert, ist es oft sinnvoll, schon mit vordefinierten Werten zu arbeiten. Stellen wir uns einmal vor, wir benötigten eine Konstante, dann ist sie nicht erst durch den Benutzer, vielleicht in einer Eingabe zu erfahren, sondern direkt verfügbar. Ein Beispiel wäre z. B. die maximale Länge der Eingabe, die Anzahl der zugelassenen Werte, die Versionsnummer usw.

Um nun Variablen einen Wert zuzuweisen, erlaubt Devpac dies mit dem Schlüsselwort DC. Ein Schlüsselwort ist ein Befehlswort aus dem Befehlsvorrat des Assemblers, mit dessen Hilfe er herausbekommt, was wir mit unserem Programm erreichen wollen (Profan ausgedrückt, wenn ich schreibe „Male Bildschirm rot“, färbt sich (nach der Compilierung) der Bildschirm rot). Der Programmteil, der die Eingaben erkennt und auswertet, nennt man Parser. Jedes Adventure mit Texteingabe muss einen solchen Parser besitzen. Je umfangreicher der Vokabelschatz, desto mehr Möglichkeiten sind gegeben. Der Devpac Assembler hat einen recht großen "Schatz", indem nicht nur alle Assemblerbefehle, sondern auch erweiterte Befehle verarbeitet. Unter ihnen zählen z. B. solche, die es dem Programmschreiber erlauben, Programme nachzuladen, Optimierungsmöglichkeiten einzuschalten u. v. m.

Dem Assembler ist es gleichgültig, ob wir die Schlüsselworte groß oder klein schreiben (er wandelt sie sowieso einheitlich um). Starten wir also den Assembliervorgang, so arbeitet Devpac das Programm von oben nach unten und von links nach rechts Schritt für Schritt ab und findet durch DC raus, dass wir nun einen Wert in das Programm schreiben wollen. Ein richtiges Programm ist das selbstverständlich noch nicht, denn das Datenwirrwarr, wird höchst vermutlich einen Absturz verursachen. Es sind ja nur Daten, nicht jedoch Programme. Es ist also darauf zu achten, das die Daten immer isoliert vom Programmteil stehen, da sonst das Programm Gefahr läuft, in die Daten hineinzulaufen, und ein Absturz ist höchst wahrscheinlich.

Schreibweise der Schlüsselwörter

Die Befehle, die der Assembler bereitstellt, sind einzurücken. Der DC Befehl zum Platz machen von Variablen gehört dazu. Der DC Befehl verlangt noch eine Erweiterungsangabe, nämlich von welchen Typ (Long, Word, Byte) die Variable sein soll. Diese wird durch Trennung mit dem Punkt direkt angehängt, und die Endungen lauten jeweils `l` bei Long, `w` bei Word und `b` bei Byte. Einfach zu merken. Auch bei allen anderen Befehlen, wird diese Typenkennung angehängt. Nach dem Variabelnamen und dem DC-Befehl plus Typ folgt der Wert, der wiederum eingerückt wird. Der Wert ist ein konstanter Ausdruck. Zum Glück unterstützt uns der Assembler im Umgang mit den verschiedenen Zahlenformaten wie Binär und Dezimal, und bietet zudem die Möglichkeit, Rechenoperationen durchzuführen und somit Werte zu verknüpfen, die so die Übersichtlichkeit fördern.

Beispiel:

```
        dc.b    12
        dc.w    234+345
        dc.l    $4564365-$34345+34
        dc.w    %1000101100110101
```

Da häufig mit Words gearbeitet wird, kann man die Endung `.w` bei den meisten Assemblern weglassen. Auch der Devpac erkennt dies, und meckert nicht. Er nimmt also immer den Typ Word an, wenn nichts hinter den Befehlen steht. Wiederum warne ich davor beim Zugriff auf ungerade Speicherstellen Words oder Longs zu benutzten.

Zeichenketten sind aneinandergereihte Bytes

Zeichenketten werden durch Bytes repräsentiert. Da sie häufig gebraucht werden, hat man, um nicht immer die ASCII-Codes aneinanderzureihen, den Anführungsstrich zur Umwandlung eingeführt. Möglich ist aber auch bei vielen Assemblern das Hochkomma. Wir wollen aber nur Anführungsstriche verwenden, dass dies normal ist.

Strings in Anführungsstriche stehen Zeichen für Zeichen, d.h. Byte für Byte im Speicher. Wenn wir ein Feld haben (eine Zeichenkette ist ein Char-Feld), können wir mit Komma getrennt noch weitere Werte anhängen.

Beispiele:

```
        dc.b    12
        dc.b    3
```

ist gleichbedeutend mit

```
        dc.b    12,3
```

oder ein anderes Beispiel:

```
        dc.b    "ULLI",0        ;ist gleichbedeutend mit

        dc.b    "U"
        dc.b    "L","L","I"
        dc.b    0               ;ist gleichbedeutend mit

        dc.b    $55,$4c,$4c,$49,0
```

```
        dc.l    $554c4c59
        dc.b    0
```

Eine interessante Frage wäre: Was ergeben

```
        dc.w    "Ulli"  ; oder
        dc.l    "Ulli"  ; ?
```

Nun, die erste Zeile würde einen Fehler ergeben. Da Ulli die Hexzahl $554c4c59 ergibt, die aber größer als ein Word ist, wäre höchstens

```
        dc.w    "Ul"
        dc.w    "li"
```

sinnvoll. Die zweite Möglichkeit ist jedoch sehr gut denkbar, denn man bekommt 4 Zeichen, oder 4 Bytes, in ein Long. Bei IFF-Bildern z. B. muss man die Kennung BODY finden. Dies geht sehr gut über ein Long, man spart sich viermaliges Suchen eines Bytes.

Labelnamen

Ein Labelname dient als Sprungziel (später mehr dazu) oder der Variablenspeicherung im Programm. Er kann aus alphanumerischen Zeichen (d.h. Buchstaben und Zahlen) und einigen Sonderzeichen zusammengesetzt werden, allerdings mit der Ausnahme, dass am Anfang eines Labelnamens immer ein Buchstabe stehen muss. Wenn die Variablen benutzt werden achtet der Assembler minutiös darauf, wie die Variablen geschrieben werden, jedoch kann man einstellen, ob er auf Groß/Kleinschrift achten soll. Ein Labelname wäre also `Hallo`, `Test1`, `_Meine_Variable` oder, vom Assembler, abhängig `$345`. Nicht erlaubt ist z. B. `24354` oder `*234`, da im ersten Beispiel eine Zahl und im zweiten Beispiel ein unerlaubtes Sonder-Zeichen am Anfang steht.

Es besteht auch die Möglichkeit hinter den Labelnamen ein Doppelpunkt zu setzen, um evtl. anderen Assemblern keine Chance zur Fehlermeldung zu geben, denn sie nutzen diesen Doppelpunkt zur Erkennung.

Durch den Druck auf die Tabulatortaste oder Leertaste unterscheidet der Assembler zwischen Labelnamen und Befehlen. Befehle werden immer eingerückt im Gegensatz zu den Labelnamen oder Variablen, die immer, ohne ein Trennzeichen, am Beginn einer Zeile stehen müssen.

Persönlich benutzte ich zum Einrücken immer die Tabulatortaste, die auf eine Sprung-Länge von 16 Zeichen gestellt ist. Alle Labelnamen sollen aussagekräftig sein, und daher nicht zu kurz sein. Die Beispielprogramme enthalten Label- und Variablennamen, die bis 16 - 1 Zeichen lang sind. Um die Optik zu verbessern, sollte diese Länge schon eingestellt werden. Da allderings im Buch so weit nach rechts eingerückte Programme problematisch sind, wird eine Einrückung von 8 Zeichen verwendet.

Ein Label-Programm

Wir sind jetzt in der Lage, unser erstes Pseudo-Programm zu schreiben. Es besteht aus Labelnamen.

Beispiel:

```
Hallo_Ich_bin_ein_Label
Wert1:
_Label3
```

Natürlich kann dies jetzt assembliert werden. Der Devpac-Assembler benutzt die Tastaturkürzel AmigaRechts+a. In dem aktuellen Window erscheinen nun Assemblier-Direktiven, die das Programm z. B. auf Diskette compilieren lassen. Der eigentliche Assembliervorgang wir durch das Textfeld Assemblieren oder Druck auf die Return Taste eingeleitet. Es erscheint ein neues Window, in dem Informationen über das erstellte Programm zu finden sind. Die Länge des Programms ist 0 Byte. Danach ist eine Taste zu drücken, und man befindet sich wieder zurück im Texteditor.

Da es unwichtig ist bei jedem Vorgang diese Optionen neu Einzustellen, und dann erst zu einem Window zu kommen, kann der Vorgang auch abgekürzt werden, indem man Amiga+Shift+a drückt. Es erscheint dann sofort das Informationsfenster mit der Länge und evtl. Fehlermeldungen.

Was geschieht beim Assemblieren?

Versuchen wir den Assembliervorgang zu verstehen. Devpac befindet sich an der ersten Spalte der Zeile. Ist dort ein Tabulator oder ein Leerzeichen? Nein. Daraus folgt, es kann sich nicht um einen Befehl handeln, da der immer eingerückt wird. Als zweites wird nun überprüft, ob der Labelname auch korrekt ist. Ist das erste Zeichen ungleich einer Zahl. Ja ist es. Nach der fertigen Überprüfung wird der Labelname in einer internen Tabelle eingetragen, damit anderen Befehlen bekannt ist, dass ein Label existiert.

Programm aus Labeln und Variablen

Wir sind nun in der Lange unser erstes Programm in Verbindung mit Variablen und Labelnamen zu schreiben. Es besteht aus einer Anreihung von Daten. Solche Dateien, die nur Informationen erhalten, gibt es häufig. Sie enthalten eingestellte Daten, wie z. B. die Fenstergröße, das Aussehen des Mauszeigers, die Farben, usw. Wenn wir das Workbench Programm Preferences aufrufen, können wir so eine Datei aus reinen Daten erstellen. Sie heisst „system-configuration“. So kann man einfach beim Booten des Computers den Bildschirm schwarz stellen, indem man die System-Datei in den Debugger lädt und sie nach den Bedürfnissen ändert und wieder sichert. Denn mit Preferences wird es kaum gelingen, alle Farbwerte auf Schwarz zu stellen. Auch Devpac arbeitet mit so einem Datenfile. Es heißt *genam2.inf* und enthält die Information, ob z. B. das Fenster beim Laden immer auf voller Größe entfaltet wird, oder ob auf Groß-Kleinschreibung geachtet wird, wie groß der Arbeitspuffer ist, ob Debug-Information beim Assemblieren erstellt werden soll.

Beispiel Programm 1:

```
Start   dc.b    "DOS",0                 ; Ich bin die Bootblock Kennung
Check   dc.l    $2134-$2+%1010101+3     ; Ich bin eine Checksumme
Größe   dc.b    10,20,200,100           ; Wir sind Fensterkoordinaten
```

```
        dc.b    "Blubbber, Blubber"     ; Wir sind Blubberer
Farben  dc      $fff                    ; Ich bin die Farbe weiß
        dc      $f00                    ; Ich bin Rot
```

Hinter den Befehlen, können durch Semikolon getrennt Bemerkungen stehen. Sie werden natürlich nicht übersetzt. Auch ist es nicht möglich mehrere Anweisungen in eine Zeile zu packen, nur zum Vorteil, sonst wäre noch mehr Lese-Schwierigkeit. Da der erste erkannte Befehl ausgewertet wird, ist es auch der Fall, dass das Semikolon vernachlässigt werden kann, man also direkt Bemerkungen anhängen kann (Trennzeichen nicht vergessen, am Besten ein Tab zur Übersichtlichkeit). Eine Zeile wie folgende wird nicht mit einem Fehler quittiert:

```
        dc.b    12 das geht
```

Dieser "Trick" ist durch den Assembliervorgang leicht zu verstehen. Wenn nach einem Datum (Daten im Singular) kein Komma folgt, werden vom Assembler keine weiteren Werte erwartet. Da er dann mit dieser Zeile fertig ist, kann er bis zum Return alles Überspringen.

Wenn wir das Beispiel-1-Programm assemblieren, merken wir schon an der Länge, dass wir etwas in den Speicher geschrieben haben. Um uns den Speicherinhalte einmal anzusehen, springen wir in den Debugger (AmigaRechts+d). Mit AmigaRechts+a können wir zu den Labelnamen springen. Wir probieren dies aus und geben <Start> ein. Befinden wir uns im aktiven Fenster links unten, so müssen wir noch durch Druck auf die Tabulatortaste in das Hex-Window rein, denn mit den Informationen, die in dem aktuellen Fester sind, können wir noch nichts anfangen. Wir drücken nun also noch einmal AmigaRechts+a um zum Label Start zu gelangen. Es wird dort eine Adresse sichtbar (es ist die gleiche wie links), die uns anzeigt, wo die Daten (oder das Programm) sich im Speicher befinden. Die Speicherorganisation übernimmt der Assembler, denn er weiß, wie lang das Programm wird, und organisiert dementsprechend freien Speicher. Die Adresse zeigt nun an, wo Devpac freien Speicher gefunden hat. Nun sind in dem Window die Daten in Hexadezimal- und ASCII-Form gegeben. Wollen wir den Inhalt der Variablen Farben erfahren, springen wir zum Label Farben, und der Speicherinhalt ist $0fff. Daneben finden wir $0f00. Das dieses Word nebenan steht, dürfe einleuchtend sein, denn in unserem Programm hätten wir statt

```
Farben  dc      $fff
        dc      $f00    auch

Farben  dc      $fff,$0f00
```

schreiben können.

Es gibt sogar noch eine weitere Möglichkeit diesen Sachverhalt zu formulieren. Da diese beiden Word zusammen ein Long ergeben, kann man auch

```
Farben  dc.l    $0fff0f00       oder
Farben  dc.l    $fff0f00        ; führende Nullen können weg
```

schreiben. Doch dann wäre die Lesbarkeit absolut mies, und wir könnten uns bei einem Programm mit 40 Variablen schon nicht mehr merken, dass das Long die zwei Farbwerte enthält.

Wichtig: Wir müssen uns merken, dass in Assembler Ordnung und gute Kommentierung ein Muss ist.

Der erste Befehl (MOVE)

Nach der Einleitung in verschiedene Zahlenformate, und einiger Allgemeinheiten sind wir in der Lage unsere ersten Programme zu schreiben. Es sind zunächst welche, die den Speicher auslesen. Der Befehl, der dies möglich macht, heißt MOVE. Er kommt aus dem Englischen (wie alle Befehle) und heißt übersetzt "bewege". Daten werden also mit diesem Befehl bewegt. Es ist der am häufigsten vorkommende Befehl, was eigentlich klar sein müsste. Der Befehl ist sehr komplex und leistungsstark. Wollen wir uns zunächst einmal damit beschäftigen Werte in Register zu schreiben.

Nach dem MOVE-Befehl folgt der bekannte Typ (falls Word dann kann es ja wegfallen), und nach dem gewohnten Trennungszeichen die Daten. Sie sind in einer besonderen Reihenfolge aufgebaut, und zwar so, dass zuerst die Quelle kommt (woher die Daten sind) und dann das Ziel (wohin mit den Daten).

Allgemein sieht der `move`-Befehl dann so aus:

```
move.(Typ) Quelle,Ziel
```

Unser erstes Programm soll einen Long Absolutwert, sagen wir 106060, in das vierte Datenregister (`D3`) schreiben.

Beispiel Programm1:

```
        move.l  #106060,d3
```

Der Zaun, auch Nummerzeichen genannt, sagt dem Assembler, das ein absoluter Wert die Quelle ist. In Erinnerung an die Rechenoperationen hätten wir auch

```
        move.l  #100000+6000+60,d3
```

schreiben können. Vielleicht wird an dieser Stelle schon deutlich, dass man mit Rechnungen den Term einfacher lesen kann, da man sieht, wie sich die Zahl zusammensetzt.

Wissen was in den Registern steht

Natürlich ist es auch möglich die Register untereinander zuzuweisen. Das ist in vielen Fällen auch sinnvoll, da die Zuweisung von Registern schneller ist als Zuweisung von Zahlen in Registern. Auch benötigt eine Register-zu-Register Zuweisung weniger Speicherplatz. Uns stellt sich einmal folgendes Problem: Die Datenregister `D0`, `D2`, `D5` und `D7` sollen auf null gesetzt werden.

Es bietet sich voreilig an, folgendes zu schreiben

```
        move    #0,d0
        move    #0,d2
        move    #0,d5
        move    #0,d7
```

Jetzt wollen wir einmal die Länge des Programms berechnen. Wie schon erwähnt hat jeder Befehl eine Mindestlänge von 2 Bytes. In diesen sind die Informationen über Art des Befehls, Quelle und Ziel enthalten. (Wie dies im Detail aussieht, werden wir später erfahren.) Der MOVE-Befehl kostet 2 Bytes und das darauffolgende Word enthält den Absolutwert, in unserem Falle 0. Die Länge eines Befehls zum Bewegen eines Wortes in ein Register kostet alles in allem 4 Bytes. Da wir 4 Zeilen haben, hat das Programm eine Länge von 4+4+4+4=16 Bytes. Dies lässt sich leicht durch den Assembliervorgang überprüfen. Doch wäre es nicht viel geschickter, die Inhalte der Register zu beachten? Da In dem `D0`-Register ja schon der Wert Null enthalten ist, warum dann nicht einfach das `D0` Register in die übrigen schreiben. Nun müssen nur noch 2 Bytes zur Identifikation her (MOVE-Befehl, der Werte zwischen Registern bewegt).

Das Programm lautete also:

```
        move    #0,d0
        move    d0,d2
        move    d0,d5
        move    d0,d7
```

Die Ersparnis ist offensichtlich. Die erste Zeile kostet immer noch 4 Bytes, doch die folgenden nur noch 2. Die Länge dieses Programms würde also nur noch 4+2+2+2=10 Bytes betragen. Wenn wir Longs benutzen würden, wäre der Unterschied natürlich noch größer, da dann der gesamte Befehl 6 Bytes bräuchte.

Wir sollten uns immer bewusst sein, was in den einzelnen Registern ist. Wenn es sein muss, sollte dies dokumentiert werden. Was sehr deutlich herauskam, ist, dass die Registerwahl sehr wichtig ist. Daher ist ein guter und sinnvoll gewählter Registersatz immer die Grundlage eines guten Programms. Nicht umsonst versuchen alle Compiler die volle Registeranzahl auszureizen, oder Notfalls noch einen Optimierer nach zu schalten, um nicht irgendwo noch ein Register-Rest ungeachtet zu lassen. Was aber nicht gemacht werden sollte, ist, evtl. Werte mit einzukalkulieren, nach dem Motto “Am Anfang habe ich schon mal eine Null in ein Register geladen, kann aber sein, dass das Unterprogramm den Inhalt schon bei seinem Aufruf überschrieben hat“. So nicht. Es muss immer eindeutig sein, was in jedem Register steht ist. Im späteren Verlauf wird dies bei der Betriebssystem-Programmierung noch sehr wichtig werden, und ist mitunter Quelle vieler Fehler und Abstürze.

Natürlich muss die Quelle nicht immer ein Register oder ein Absolutwert sein. Viel häufiger ist es, dass der Speicher ausgelesen werden muss, Variableninhalte also verlangt werden. Das folgende Programm schreibt den Inhalt der Variablen `MaxWert` in das `D2`-Register.

Programm:

```
        move.l  MaxWert,d2
        rts

MaxWert dc.l    2345345
```

Mit RTS kommen wir wieder ins aufrufende Programm zurück

Wir benutzten hier erstmalig einen neuen Befehl: RTS. Er soll aber erst an späterer Stelle beschrieben werden. Es sei nur kurz gesagt, dass RTS immer zum Schluss eines Programms steht. Findet der Prozessor den RTS Befehl wird wieder ins Hauptprogramm zurückgesprungen. Wenn wir den RTS Befehl vergessen, könnte ein Absturz kommen, da das Programm in die Variablen hineinläuft, also immer daran denken, Programm- und Variablenteil trennen.

Als nächstes wollen wir zwei Werte vertauschen. Quelle des MOVE-Befehls ist also einmal der Speicher, und einmal ein Register. Das Ziel ist auch einmal ein Register und einmal der Speicher.

Programm:

```
        move.l  Wert1,d0
        move.l  Wert2,d1
        move.l  d1,Wert1
        move.l  d0,Wert2
        rts

Wert1   dc.l    $3456
Wert2   dc.l    $523546
```

Wir können den Vorgang mit dem Debugger überprüfen. Nach der Compilierung (aber vor der Ausführung) können wir uns den Speicherinhalt, an der Adresse Wert1 ansehen. Falls der Monitor einen Fehler ausgibt, dass er den Labelnamen nicht kennen würde, sollte im Einstellfenster zur Assemblierung die Option Debug Information gewählt werden. Nach dem Sprung zur Adresse ist dort die Bytekette 0000 3467 0052 3546 zu finden. Nach dem Ausführen hat sich das Bild im Debug-Fenster geändert. `Wert1` und `Wert2` haben ihren Inhalt vertauscht, so wie es der Programmierer wollte.

Hi- und Lo-Byte vertauschen

Bei dem PC kommt es des Öfteren vor, das Hi- und Lo-Byte (wenn das Word die Zahl $1234 darstellt, ist das höhere (Hi) Byte $12 und das Tiefere (Lo) $34) eines Wordes vertauscht werden müssen. (Nachdem Digital Research erstmals für Z80 Rechnern ein Betriebssystem schrieb, und auch Atari von der Firma eines verlangte, übersetzten sie das Alte nur neu, fummelten ein paar veraltete Mac-Windows dazu, und haben sich nicht einmal die Mühe gemacht, es einem 32 Bit Rechner anzupassen. Eine Frechheit, und so kommt es noch vor, dass Hi- und Lo-Werte vertauscht werden müssen, damit die Routinen des Festspeichers, wie Diskette, einwandfrei funktionieren.) Bei diesem Vertauschen wenden wir einen kleinen Trick an.

Programm:

```
        move.b  Wert,d0
        move.b  Wert+1,d1
        move.b  d1,Lo
        move.b  d0,Hi
        move    VertauschtWert,d0
        rts

Wert    dc      $1234

VertauschtWert
Lo      dc.b    0
Hi      dc.b    0
```

Die erste Zeile ist noch offensichtlich. Ein Byte Wert, der an der Adresse von Wert zu finden ist, wird in das `D0`-Register geladen. Die zweite Zeile wird wahrscheinlich Kopfzerbrechen mit sich bringen, denn hier wird auf dem ersten Blick etwas Neues angewendet, was aber nach Überlegung nicht unbedingt neu ist. Wir sprachen schon einmal von mathematischen Operationen die Werte berechnen können. Diese werden hier angewendet, denn sie sind nicht nur in Verbindung mit der Berechnung der Quelle einzusetzen (wie die Absolute Zahl), sondern auch hier gibt es die Möglichkeit, noch nachträglich die Adresse zu ändern. Versetzen wir uns in die Lage des Assemblers. Nach dem Zweiten Pass kennt er die Adresse der Variablen Wert. Sie ist z. B. 234567. Das Programm holt sich aus diesem Speicherbereich ein Byte ($12) heraus. In der nächsten Zeile wird wiederum die Adresse 234567 als Grundwert genommen, doch durch Addition verschiebt sich der Bereich des Zugriffs um ein Byte. Die Adresse, die wir erhalten, beträgt 234568. Dort können wir das nächste Byte ($34) rausholen. In der fünften Zeile wird auf die vertauschte Reihenfolge mit einem Word zugegriffen. Das geht, denn der Speicherbereich ist gerade, und enthält die 2 Bytes, die wir kurz zuvor in den Speicher geschoben haben.

String umdrehen

Um dieses Verfahren noch zu erhärten, wollen wir einen vier Buchstaben langen Text umdrehen. Wir nutzen vier Register, in denen die vier Buchstaben abgelegt werden. Wir müssen hier das Byte als Registerlänge wählen, da sonst wieder ein Zugriff auf eine ungerade Speicherstelle geschehen würde. Das Programm hätte man auch anders hätte schreiben können. Eine Benutzung von nur zwei Registern hätte ebenso bei Registermangel gereicht, denn es bietet sich an, erst die äußersten Buchstaben, und dann die inneren zu vertauschen.

Programm:

```
        move.b  Text,d0     ; gleichbedeutend mit Text+0
        move.b  Text+1,d1   ; hier würde bei
                            ; Word ein Absturz
        move.b  Text+2,d2   ; sicher sein, denn Text+1
                            ; ist ungerade
        move.b  Text+3,d3
        move.b  d3,Text
        move.b  d2,Text+1
        move.b  d1,Text+2
        move.b  d0,Text+3
        rts

Text    dc.b    "Ulli"
```

Das erste Kopierprogramm

Nehmen wir einmal an, wir hätten einen Text, der 15 Buchstaben lang ist. Wir wollen den Text nach Puffer kopieren. Wie können wir vorgehen? Unser erster Gedanke bedeutet viel Tipparbeit, denn es besteht die Möglichkeit den Speicher Byte für Byte zu kopieren.

```
        move.b  Text,d0
        move.b  d0,Puffer
        move.b  Text+1,d0     ; d0 Brauchen wir nicht mehr
        move.b  d0,Puffer+1
        .
        .
        .
        move.b  Text+15,d0
        move.b  d0,Puffer+15
        rts

Text    dc.b    "Ich bin das Prg"
Puffer  dc.b    "Das Programm   , das verschiebt"
```

Doch warum sollten wir immer Bytes benutzen. Viel sinnvoller wäre es doch, die Länge eines Longs auszunutzen, denn er kann gleich 4 Bytes aufnehmen und somit auch verschieben. Da wir jedoch durch Longs nur durch vier teilbare Speicherblöcke bewegen können (ohne Rest), bleibt ein Rest von 3 Bytes. Nur 12 Bytes können verschoben werden (4 bewegte Bytes * 3 Long-Zeilen = 12 Bytes), bei verschieben von 4*4 Longs wären 16 Bytes bewegt, und das wäre falsch. Die restlichen drei Bytes können nun mit einem Word und einem Byte-Zugriff verschoben werden. Die verbesserte Kopier-Version lautet also:

```
        move.l  Text,d0      ; Byte eins bis 4 übertragen
        move.l  d0,Puffer
        move.l  Text+4,d0    ; Bytes 4 bis 8 übertragen
        move.l  d0,Puffer+4
        move.l  Text+8,d0    ; Bytes 8 bis 12 übertragen
        move.l  d0,Puffer+8
        move.w  Text+12,d0   ; Bytes 12 und 13 übertragen
```

```
        move.w  d0,Puffer+12
        move.b  Text+14,d0   ; Byte 14 übertragen
        move.b  d0,Puffer+15
        rts

Text    dc.b    "Ich bin das Prg"
        cnop    0,2    ; Grademachen der Speicheradressse

Puffer  dc.b    "Das Programm   , das verschiebt"
```

Ich kann es gar nicht oft genug sagen, dass man auf ungerade Adressen nicht mit Word oder Long zugreifen darf. Da bei Texten die Wahrscheinlichkeit 50 zu 50 ist, dass sie gerade oder ungerade sind, erscheint es sinnvoll, den darauf folgenden Puffer eine gerade Adresse zu geben. Dies geschieht mit dem Befehl CNOP. Mit ihm kann man also eine gerade Adresse erzwingen, und auf `Puffer` kann nun ein beliebiger Zugriff erfolgen.

Ab den Speicherbereich `Puffer` finden wir, mit dem Debugger zu überprüfen, den String “Ich bin das Prg, das verschiebt“.

Noch einige Worte zur Routine. Zwar haben wir einen Coprozessor, den Blitter, der Speicherbereiche schnell verschieben kann, aber dennoch ist der Prozessor mit ein paar weiteren Tricks schneller. Eine Betriebssystemroutine (unter `Exec` ist das `CopyMem()` oder `CopyMemQuick()`) nutzt ebenfalls den Vorteil, das ein Long schneller kopiert werden kann als vier einzelne Bytes. Für größere Speicherblöcke lohnt also auf jeden Fall der Aufruf dieser Funktion, da sie, wie wir gesehen haben, effektiv arbeitet.

Mathematische Operationen

Natürlich bietet der MC68000 nicht nur Kopierbefehle. Das Spektrum ist vielfältig, so werden auch alle Grundoperationen vom Prozessor realisiert. Dabei stellen die Addier- und Subtrahierbefehle den Mittelpunkt der mathematischen Operationen dar.

Addieren mit dem ADD-Befehl

Bei dem Addierbefehl benötigen wir eine Quelle, von der die Werte kommen, und ein Ziel, worin die Ergebnisse der Addition abgelegt werden. Es wird also immer die Quelle mit dem Ziel addiert und das Ergebnis steht im Ziel.

Allgemein kann man wie bei dem MOVE-Befehl schreiben:

```
add(.Typ) Quelle,Ziel
```

Wir haben schon 3 Quellen kennengelernt. Den Absolutwert, das Register und den Speicher. Ebenso zwei mögliche Ziele, das Register und der Speicher. Schreiben wir unser erstes Additionsprogramm, das den Umgang mit Ziel und Quelle ein wenig vertieft.

Programm:

```
        move.l  #2000,d0        ; Absolutwerte
        move.l  #1000,d1
        add.l   d1,d0           ; Ergebnis in d0
```

Zweite Möglichkeit:

```
        move.l  Wert1,d0        ; nun werden hier die Zahlen
        move.l  Wert2,d1        ; aus dem Speicher gelesen
        add.l   d1,d0           ; Ergebnis ebenso in d0
```

Dritte Möglichkeit:

```
        move.l  #2000,d0
        add.l   #1000,d0        ; Ergebnis auch in d0
```

Vierte Möglichkeit:

```
        move.l  Wert1,d0
        add.l   #1000,d0        ; Auch in d0
```

Fünfte Möglichkeit:

```
        move.l  Wert1,d0
        add.l   Wert2,d0        ; Ebenso in d0
```

Sechste Möglichkeit:

```
        move.l  #1000,d0
        add.l   d0,Wert2        ; Ergebnis in Wert2
```

Siebte Möglichkeit:

```
        move.l  Wert1,d0
```

```
        add.l   d0,Wert2        ; Ergebnis in Wert2
```

Achte Möglichkeit:

```
        add.l   #1000,Wert2     ; Ergebnis in Wert2

        rts

Wert1   dc.l    2000
Wert2   dc.l    1000
```

Bis zu der sechsten Möglichkeit wurden bewusst Register zur Verknüpfung gewählt.

Wer sich im Rechnen üben will, der kann ja einmal die Länge des Programms errechnen. Es sind vorweggenommen 108 Bytes.

Doch kann es oft sinnvoll sein, auch Variablenwerte heraufzusetzen. Ein Auszug könnte so aussehen:

```
        add.b   #1,WieOftSchonGespielt
```

oder

```
        add     #2,MausPositionXAchse
```

Buchstabenumwandlung

Im Amiga Zeichensatz trennt nur eine Länge von 30 Buchstaben die kleinen und großen Zeichen. Die Großbuchstaben kommen zuerst, und dann folgen anschließend die Kleinen. Wir können also ein Programm zur Umwandlung von Groß- in Kleinbuchstaben oder umgekehrt schreiben. Wiederum haben wir einen Speicherbereich, an dem der String, z. B. durch eine Texteingabe in dieser kopiert, steht. Dieser wird nun direkt in den gleichen Speicherbereich kopiert. Das spart einen Puffer, denn häufig interessiert die Eingabe sowieso keinen mehr.

```
        move.b  Text,d0
        add.b   #32,d0
        move.b  d0,Text
        move.b  Text+1,d0
        add.b   #32,d0
        move.b  d0,Text+1
        move.b  Text+2,d0
        add.b   #32,d0
        move.b  d0,Text+2
        rts

Text    dc.b    "ASS"
```

Wie beim MOVE-Befehl kann auch der ADD-Befehl optimierend eingesetzt werden. Man vergleiche dazu nur einmal das untere Programm.

Optimiertes Programm Version 1:

```
        move    #32,d1
        move.b  Text,d0
        add.b   d1,d0
        move.b  d0,Text
        move.b  Text+1,d0
        add.b   d1,d0
        move.b  d0,Text+1
        move.b  Text+2,d0
        add.b   d1,d0
        move.b  d0,Text+2
        rts

Text    dc.b    "ASS"
```

Der ADD-Befehl kann aber wegen seiner komplexen Adressierungsarten weiter verwendet werden.

Optimiertes Programm Version 2:

```
        move    #32,d1
        add.b   d1,Text
        add.b   d2,Text+1
        add.b   d3,Text+2
        rts

Text    dc.b    "ASS"
```

Subtrahieren mit dem SUB-Befehl

Ebenso wie der ADD-Befehl arbeitet der SUB-Befehl. Er subtrahiert das Ziel von der Quelle. Der Vorteil kann darin liegen, dass

Register leicht auf Null gesetzt werden können. Denn anstatt des schon bekannten

```
        move.l  #0,a0
```

lässt sich viel besser

```
        sub.l   a0,a0
```

schreiben. Dieser Trick mit den Adressregistern auf Null setzen ist merkenswert, denn auf Adressregister ist er schneller und kürzer.

NEG als Sonderfall von 0-Wert

Es kann schon einmal vorkommen, das von der Zahl Null eine weitere Zahl abgezogen werden muss. Um nicht umständlicher weise z. B. die Zahl 13 von 0 abzuziehen,

```
        moveq   #0,d0
        sub     #13,d0
```

bietet der 68000 einen weiteren Befehl an. Es ist der NEG Befehl, der, wie der Name schon sagt, das Argument, negiert. Das obige Beispiel würde mit dem NEG Befehl folgendermaßen viel einfacher zu gestalten sein.

```
        moveq   #13,d0
        neg     d0
```

Multiplizieren mit MULU

Mit dem MUL-Befehl können 2 Werte multipliziert werden. Es ist jedoch darauf zu achten, dass bei der Anwendung des Befehls nur zwei 16 Bit Register verarbeitet werden können. Dies erscheint logisch, denn der Maximalwert ist ein 32-Bit Wert. Um jedoch zwei Long Zahlen zu multiplizieren benötigen wir 64 Bit, das bietet der einfache 68000 nicht. (Seine Nachfolger sind aber dazu dennoch in der Lage).

Beispiel:

```
        move    #20,d0
        mulu    #50,d0
        rts
```

Das `D0`-Register enthält das Ergebnis: dezimal 100, bzw. hexdezimal $64.

Dividieren mit DIVU

Der DIVU- oder DIVS-Befehl ist sehr komplex und nimmt auf der CPU relativ viel Platz ein. Ebenso wie beim MUL-Befehl gibt es hier eine Einschränkung. Der Wert, der unter dem Nenner steht (Divisor in mathematischen Kreisen genannt) muss eine 16-Bit Zahl sein, und der Divisor muss eine 32-Bit Zahl sein. Das Ergebnis ist wieder ein 16 Bit Wert. Ging die Division nicht auf, so steht im Hi-Word des Long-Ziels der Rest, und im Lo-Word der Ganzteil der Division (Quotient).

Lautet die Division 20/6, so würde im Lo-Word 3 stehen (die Sechs passt dreimal in die Zwanzig) und der Rest von 2 würde im Hi-Byte stehen.

Beispiel:

```
        move.l  #1000,d0
        divu    #10,d0
        rts             ; 100 (als Word) steht in d0
```

Achtung! Ist der Divisor Null, so ist uns ein Guru mit der Meldung "0000 0005" gewiss, denn jeder weiß: eine Division durch Null ist nicht erlaubt.

Die Ausführzeiten

In den obigen Kapiteln beschrieb ich schon, was unter Taktzyklen zu verstehen ist. Ein Befehl, der von dem Prozessor verarbeitet wird, wird in seinen Phasen decodiert und ausgeführt. Die Ausführungszeiten sind verschieden lang und das ist einfach zu erklären da ein Befehl, der z. B. ein Wert aus dem Speicher holt und ihn etwa mit einem anderen Wert aus dem Speicher verknüpft, länger zur Ausführung braucht als z. B. ein Befehl, der nur interne Register verknüpft. Daher sind die Ausführzeiten bei Befehlen mit Registern noch erträglich. (Wiederum kann ich mich nur wiederholen und sagen: “Die Register ausreizen“.)

In den gleich folgenden Tabellen wird deutlich auffallen, dass Long-Words meistens eine höhere Ausführungszeit aufweisen als die Words. Über den Grund (16 Bit-Datenbus) haben wir schon gesprochen. Ich möchte die Zeiten in einer Tabelle anlegen, und daher ist es zwangsläufig, dass ich einige Abkürzungen einführe. Die Möglichkeiten einen Befehl anzusprechen, z. B. über Register oder Absolutwert, oder durch den Speicherinhalt fasst man unter den Begriff der Adressierungsart zusammen. Wenn ich also von der absoluten Adressierung spreche, so meine ich das die Quelle ein Absolutwert ist. Zur Abkürzung der Absoluten Adressierung wähle ich `#xxx`; ein Datenregister wird `Dx` genannt, falls das Datenregister eine Quelle ist; `Dy`, falls es Ziel sein soll; und `Dn`, wenn es allgemein in der Befehlscharakterisierung vorkommt. Das gleiche gilt für Adressregister. Variablen werden mit `xxxx` abgekürzt. Die Abk. `ea` kennzeichnet die effektive Adresse, die dann in der Tabelle angegeben ist.

Beispiel:

```
                Dx      Ax      xxxx    #xxxx
Befehl.w ea,Ay  12      12      16      20
```

Der Befehl `Befehl.w Dx,Ay` braucht also zur Ausführung 12 Taktzyklen.

Der Befehl `Befehl.w #xxxx,Ay` braucht 20 TZ um ein Absolutwort mit einem Adressregister zu verbinden.

Da aus den schon beschriebenen Gründen Bytes nicht schneller als Words ausgeführt werden können, werden nur Word und Long Zeiten in der Tabelle aufgelistet. Da die Tabelle jedoch noch mehr aussagt, ob nämlich eine bestimmte Adressierungsart erlaubt ist oder nicht, werde ich falls es notwendig ist, die Information, ob Byte Zugriff erlaubt ist, im Zweifelsfalle immer geben.

Wir wollen mit den Datentransferbefehl MOVE beginnen.

```
                Dx      Ax      xxxx    #xxxx
MOVE.w  ea,Dy   4       4       16      8
MOVE.l  ea,Dy   4       4       20      12
```

Die gleichen Zeiten gelten für MOVE-Befehle, bei denen das Ziel ein Adressregister (d.h. `move.w/l ea.Ay`) ist.

```
MOVE.w  ea,xxxx.l       12      12      28      20
MOVE.l  ea,xxxx.l       16      16      36      28
```

Die Addierbefehle

```
ADD.w   ea,Dy   4       4       16      8
ADD.l   ea,Dy   6       8       22      14
ADD.w   Dx,ea   4       -       20      -
ADD.l   Dx,ea   6       -       28      -
ADDA.w  ea,Ay   6       8       20      12
ADDA.l  ea,Ay   8       8       22      14
```

die Endung `A` an ADD müsste eigentlich auch an das MOVE dran, da dadurch deutlich wird, dass eine Adresse das Ziel ist. Doch benötigt Devpac dies nicht zur Erkennung, da er selbst an der Endung erkennt, dass dies ein Adressregister ist. Das gleiche gilt für Absolutzahlen, wo eigentlich ein `I` ans Ende gehört (z. B. SUBI).

```
ADDI.w  #xxxx,ea        8       -       24      -
ADD.l   #xxxx,ea        16      -       36      -
```

Die Subtraktionsbefehle benötigen die gleiche Taktzyklenzeit wie die Additionsbefehle, daher sind sie hier nicht erneut aufgelistet.

Die Multiplikations- und Divisionsbefehle

```
MULU.w  ea,Dy   70      -       82      74
DIVU.w  ea,Dy   158     -       170     16
```

Interessant ist, dass diese Befehle nur ein Datenregister als Ziel haben können.

Wenn wir uns diese Zeiten ansehen, und uns klarmachen, das Intel oder RISC Prozessoren nur so 2 Taktzyklen benötigen, so sind diese langen Zeiten blamierend. So ist es ein Muss für uns Programmierer, die kürzesten Befehle raus zu picken. In späteren Kapiteln, wo noch mehrere Quell- und Ziel-Arten vorkommen, wo wir also Bekanntschaft mit weiteren Adressierungsarten machen, ist dies wichtiger Bestandteil der Programmierung, daher ist es sinnvoll, sich die schon bekannten einmal einzuprägen.

Optimieren mit ADD

Die Optimierung soll zeigen, wie man kürzere und schnellere Programme generiert. Mit dem Addiere-Befehl lässt sich auch schon optimiert und zeitsparend arbeiten. Man versucht in Assembler die langsamen Multiplikationen durch die fast 70-mal schnelleren Additionen zu ersetzen, denn die Multiplikationsbefehle und die noch längeren Divisionsbefehle haben Zeiten, wo man sich schon einen Kaffee holen kann. (Für die, die sich jetzt benachteiligt fühlen, natürlich auch Kakao, Milch, Tee und Limo!). Die Ersetzung gelingt in den meisten Fällen.

Wir wollen nun ein Beispiel kennenlernen, wo wir die Multiplikation mit konstanten Werten durch Additionen ersetzen können. Wenn man eine Zahl mit Zwei multiplizieren will, verdoppelt sich der Wert der Zahl. Aber anstatt diese mit Zwei zu multiplizieren, können wir auch den Wert der Variablen zu sich selbst addieren. Das Ergebnis wäre das gleiche.

Programm:

```
        move    Wert,d0
        add     d0,d0   ; gleichbedeutend mit mulu #2,d0
        move    d0,Wert2
        rts

Wert    dc.l    123
Wert2   dc.l    0
```

Im Debugger kann man das Ergebnis untersuchen, und es lautet 246. Doch steht dort wirklich 246? Warum steht denn da eine kleinere Zahl? Unbemerkt haben wir den Wert der Variablen `Wert` als dezimalen gewählt. Im Monitor allerdings erscheinen nur Hex-Werte. Doch der unterstützt uns auch hier mit den Zahlenformaten. Mit AmigaRechts+o erscheint ein kleines Fenster, indem die Formate umgerechnet werden können. Bei der Eingabe von einem Hexwert (Kennzeichnung ist ein Dollerzeichen `$`) oder einer

Bin-Zahl (Kennzeichen das Prozentzeichen `%`) wird dies als Dezimalzahl und Hexzahl angezeigt. Dez-Werte oder Hex-Werte nach Bin zu wandeln geht leider nicht, und man vermisst diese Möglichkeit beim Speicherdurchschauen teilweise.

Unterprogramme mit BSR

Es fällt auf, dass wir bei dem vorherigen Programm immer einen Addierbefehl zur Multiplikation in der Zeile steht. Schöner ließe sich dies mit einem Unterprogramm lösen. Es soll nur die Addition beinhalten. In BASIC (der Assembler verwandtesten Sprache; nicht die am Assembler nächste Sprache!) geschieht dies mit dem Schlüsselwort `GOSUB` und endet mit `RETURN`. In Assembler heißt der Befehl zum Ansprung eines Unterprogramms BSR, vom Englischen „Branch to SubRoutine“ (verzweige zur Unterroutine).

Natürlich muss das Unterprogramm auch einmal beendet werden, und das geschieht mit dem schon uns bekannten RTS. Mit RTS (ReTurn from Subroutine (beende Unterroutine), nicht merken als Return To Subroutine) findet der Rechner wieder zurück ins Hauptprogramm (oder Programm, das eine Ebene höher liegt, denn auch Unterprogramme können Unterprogrammaufrufe enthalten. Der Prozessor fährt dann mit der Zeile die unter dem Aufruf steht weiter. (Näheres zur Arbeitsweise des Branch- und Return-Befehles später.) Nach dem BSR folgt ein Labelname, den wir jetzt erstmals als Ansprungsadresse einsetzen.

Unser Programm soll nun das Unterprogramm `MulMit2` enthalten.

```
          ; Hauptprogramm

          move    Wert1,d0
          bsr     MulMit2
          move    d0,Wert1Neu

          move    Wert2,d0
          bsr     MulMit4
          move    d0,Wert2Neu

          rts

; Unterprogramme zur schnellen Multiplikation

MulMit2   add     d0,d0
          rts

MulMit4   add     d0,d0
          add     d0,d0
          rts

; Variablenteil

Wert1     dc      10
Wert2     dc      20

Wert1Neu        dc      0
Wert2Neu        dc      0
```

Das Programm enthält nun ein weiteres Unterprogramm. Es multipliziert den Wert des `D0`-Registers mit Vier. Anstatt der Zeile, die ein `add d0,d0` enthält, hätten wir natürlich auch das Unterprogramm `MulMit2` aufrufen können. Denn zwei Mal mit Zwei multipliziert ergibt auch den Wert mit vier multipliziert! (Logisch nicht?)

```
MulMit4Neu
          bsr     MulMit2
          bsr     MulMit2
          rts
```

Mit Hilfe dieses Verfahrens lassen sich Zahlen über die Addiermethode immer um das doppelte erhöhen. Dieses Verfahren muss man sich merken, und wir werden später noch einmal darauf zurückkommen.

Doch um noch einmal auf die Unterprogramme zu sprechen zu kommen: Vielleicht wird man sich fragen, warum wir das Unterprogramm an das Ende gesetzt haben. Dies macht einfach der Übersicht halber. Und, oben steht das Hauptprogramm, unten das Unterprogramm. (Sonst würde man ja Oberprogramm sagen, oder?)

Die schnellen und kurzen Quickies

Es gibt Anweisungen und Befehle, die sehr häufig in Programmen auftauchen. Man einigte sich darauf, neue Befehle im Prozessor einzufügen, die diese Aufgaben schneller erledigen können, als die alten, langsamen. Die Ersparnis ist in allen Fällen gegeben, vor allen Dingen in der Long-Word Verarbeitung wird mit diesen Befehlen eine Lücke geschlossen.

Der Move Quick Befehl

Unter diesen Befehlen, die schneller verarbeitet werden können, zählen die MOVEQ-Befehle (sprich „Move quick“), die Werte zwischen 0 und 127 (Vorzeichenbehaftet, daher nicht 255) in ein Datenregister schreiben können. Die Quelle darf jedoch nur ein Absolutwert sein, und das Ziel ein Datenregister. Das hört sich nach einer sehr starken Einengung an, ist es aber im Normalgebrauch nicht. Die Ausführzeiten sprechen für die Einsetzung des Befehles und ich möchte in keinem Programm, welches Werte zwischen 0-127 (oder negative Werte zwischen -1 und -127) in Datenregister schreibt (4 TZ), einen normalen MOVE-Befehl mehr entdecken (8, bzw. 12 TZ). Doch außer der schnelleren Ausführung ist der Quicky auch noch Platzsparender. Während bei der normalen Move Anweisung 2 Bytes für die Kennung draufgehen, und dazu noch die 2 Bytes für benutzte Word, oder etwa 4 Bytes für Longs benötigt werden, so kommt der MOVEQ mit 2 Bytes aus. Das liegt daran, das der Absolutwert vom MOVEQ immer in den 2 Bytes untergebracht ist. Das ist toll. Eine Ersparnis bei Longs von 4 Bytes!

Ab jetzt schreiben wir also nicht mehr:

```
        move.l  #0,d0
```

sondern:

```
        moveq   #0,d0
```

Da der MOVEQ-Befehl keine Unterscheidung zwischen Longs und Words macht, sondern immer die Werte auf Longs ergänzt, ist dieser Befehl besonders schmackhaft. Wenn wir eine Long-Datenregister beschreiben müssten, würden wir 12 TZ verschwendet. Durch Benutzung eines MOVEQ sparen wir 12 - 4 = 8 TZ und 4 Bytes!

Doch Vorsicht: Der MOVE-Befehl bewegt nur so große Werte, wie sie ihm angegeben werden. Ist also das Long mit $ff000000 vordefiniert, und wir bewegen ein Byte ($ff) in das Register hinein, so hat es nach dem Bewegen den Wert $ff0000ff. Der MOVEQ allerdings löscht alles, was im Hi-Long steht. Da nur sehr selten Wörter oder Bytes bewusst ins Lo-Long-Datenregister geschrieben werden, und das Hi-Byte nicht zerstört werden darf, können wir meist getrost den Quicky benutzen.

Es sollte jetzt nicht mehr

```
        move.l  #16,Wert        ( ganze 28 TZ)
```

heißen, sondern nun

```
        moveq   #16,d0  ( 4 TZ)
        move.l  d0,Wert ( 20 TZ)
```

Eine Einsparung von immerhin 4 Bytes und nach dem alten Spruch, „Kleinvieh macht auch Mist“, können wir hier und da schon ein paar Bytes sparen. Ich bin sicher, das Compiler auf diese 4 Bytes unter keinen Umständen verzichten wollen.

Addiere Quick

Genauso wie es ein MOVEQ gibt, ist auch ein ADDQ, bzw. SUBQ vorhanden. Die Parallelen sind bei der Absolutquelle, der Unterschied jedoch bei dem Ziel. Das Ziel kann im Gegensatz zum MOVEQ, der nur Datenregister als Zieloperanden kennt, auch andere Adressierungsarten sein. Doch müssen wir hier mit einer (kleinen) Einschränkung leben. Unsere Quelle, also für den ADDQ/SUBQ der Absolutwert, darf nur zwischen 1 und 8 liegen. In den meisten Fällen ist damit aber auch zu leben.

So wollen wir auch nicht mehr

```
        add.l   #2,X_Achse      (36 TZ)
        add     #4,d0           (8 TZ)
        add.l   #5,d0           (16 TZ)
```

schreiben, sondern

```
        addq.l  #2,X_Achse      (28 TZ)
        addq    #4,d0           (4 TZ)
        addq.l  #5,d0           (8 TZ)
```

Sollten wir einmal in die Lage kommen, den Absolutwert 10 zu addieren, so könnten wir leicht in Versuchung kommen keinen Quicky anzuwenden. Doch halt! Neben den Ausführzeiten gibt es noch die Programm-Länge die es ebenfalls zu optimieren gilt. Bei einer Addition, die sich aus 2 Quickies zusammensetzen lässt, sollte der Quicky siegen, warum zeigt das Beispiel.

```
        add.l   #10,d0  (immerhin 16 TZ und 6 Bytes)
```

Aber es geht auch anders

```
        addq.l  #8,d0   (8 TZ und 2 Bytes)
        addq.l  #2,d0   (8 TZ und 2 Bytes)
```

Die Ausführungszeit für die beiden Quickies ist die gleiche wie bei der Ohne-Quicky-Methode (16 ZT). Doch benötigen wir nicht mehr 6 Bytes für die Addition, sondern nur noch 4. Eine kleine Optimierung, die auch zu kürzeren Programmen beitragen kann.

Alle Adressierungsarten

In den früheren Kapiteln habe ich schon angeschnitten, was Adressierungsarten sind. Wir wollen unter einer Adressierungsarten Möglichkeit verstehen, wohin der Prozessor Daten schreibt und wo er sie herholt. Das heißt, die Adressierungsart ist der Schlüssel zu unserem Speicherbereich.

Da wir schon etwas Erfahrung gesammelt haben, stellen die restlichen Adressierungsarten kein Problem mehr dar. Zur Übersicht seinen aber auch noch die bekannten aufgelistet.

Datenregister direkt

Bei dieser Adressierungsart wird der Operand entweder in das angegebene Register geschrieben, oder aus ihm gelesen. Es stehen

dazu alle Register von D0 bis D7 zur Verfügung.

Die allgemeine Syntax lautet: Dn

Adressregister direkt

Es stehen dazu alle Register von A0 bis A7 zur Verfügung.

Die allgemeine Syntax lautet: An

Adressregister indirekt

Da es ab hier neu wird, ist es wichtig die Funktionsweise dieser und folgender Adressierungsarten ein wenig zu erhellen.

Bei der indirekten Adressierungsart ist das Adressregister ein Zeiger.

Es stehen dazu alle Register von A0 bis A7 zur Verfügung.

Die allgemeine Syntax lautet: (An)

In der Erklärung tauchte das Wort Zeiger auf, und vielleicht fragen einige, was das denn sein soll. Programmierer, die mit C oder Pascal vertraut sind, dürften mehr oder weniger schon wissen, was man mit diesen Zeigern machen kann. Der Name kommt daher, dass sie auf eine Speicherzelle zeigen. Wenn z. B. das Adressregister A4 die Adresse $10000 enthält, können wir den Sachverhalt auch mit folgenden Worten beschreiben. Der Zeiger A4 zeigt auf die Adresse $10000. Na und? Was haben wir davon? Nun, mit Hilfe dieser Adressierungsart ist es möglich, Daten, die an der Adresse stehen, auszulesen, zu bearbeiten, oder natürlich auch zu schreiben. Wollen wir doch einmal den Inhalt des Speichers der Adresse $1000 in ein Datenregister (D2) schreiben. Es soll nur ein Byte gelesen werden. Mit unserem bisherigen Wissen machen wir das wie folgt:

```
        move.b  $10000,d2
```

Aber, was ist, wenn die Adresse nicht bekannt ist? Schon ein Beispiel, warum die absolute Adressierung in manchen Fällen einige Probleme bereitet. Wir wollen daher die Zeile erneut schreiben, und zwar jetzt mit der indirekten Adressierung.

```
        move.l  #$10000,a0
        move.b  (a0),d2
```

Anstatt ein move.l #xxxx,a0 kann man einen neuen Befehl verwenden, der Absolutwerte in Adressregister schreibt. Die neue Version sieht dann folgendermaßen aus:

```
        lea     $10000,a0
        move.b  (a0),d2
```

Mit dieser Möglichkeit können wir auch einfach Daten verschieben

```
        lea     Wert1,a0        ; gerade Adresse
        lea     Wert2,a1        ; gerade Adresse gefordert
        move    (a0),(a1)
```

Nicht jeder Assembler achtet auf die geraden Adressen, die bei dem LEA-Befehl geladen werden. Sollte eine Adresse ungerade sein, wird spätestens bei der Ausführung die Rache kommen. Bei der Bewegung der Word wird ein Absturz folgen.

Übergabe beim CLI-Aufruf

Doch die Anweisung ist vielfältiger Natur. Nach dem Aufruf eines Programms aus dem CLI-Fenster ist in A0 ein Zeiger auf den Eingabestring zu finden. In D0 befindet sich übrigens die Länge der Eingabe.

Haben wir ein Programm zur Directoryausgabe geschrieben, und wollen als Parameter die Laufwerksnummer haben, so zeigt A0 auf den Eingabestring. Wir kürzen bei der Parameterübergabe einmal ab, und schreiben anstatt DF0 nur einfach 0. Da ja nur die Nummer gefragt ist, und nicht das führende DF, muss die Länge des String, also auch der Inhalt von D0 Eins sein, andernfalls wäre die Übergabe fehlerhaft. Wir können nun auf die Laufwerknummer mit Hilfe der indirekten Adressierung zugreifen, indem wir das Byte auslesen.

```
    move.b (a0),d1 ; In d1 Steht die Ziffer
```

Es gäbe keine andere Möglichkeit an die Daten heranzukommen.

Adressregister indirekt mit Postinkrement

Bei der indirekten Adressierungsart mit Postinkrement ist das Adressregister ein Zeiger und dieser wird nach der Operation erhöht. Der Wert, um den das Adressregister erhöht wird, richtet sich nach der Operandenlänge. Wird ein Byte eingelesen, wird das Adressregister nur um den Wert eins erhöht, bei Word um zwei.

Es stehen dazu alle Register von A0 bis A7 zur Verfügung.

Die allgemeine Syntax lautet: (An) +

Die Features des Prozessors gehen noch weiter. In einigen Fällen wollen wir Daten, auf die die Adresse zeigt, auslesen, und die danach folgenden Informationen ebenfalls verwerten. Als Beispiel soll eine Tabelle angeführt werden, die Informationen über die Fenstergröße enthält. Dabei soll der erste Wert ins D0-Register geschrieben werden, und der nächste ins D1-Register. Die Tabelle enthält nur Words, ebenso sollen aus der Tabelle nur Words in die Register geladen werden. Der Zeiger auf Tabelle, nicht die Tabelle selber, soll in der Speicherstelle TabStart stehen.

```
    move.l  Tabstart,a2
    move    (a2),d0
    add.l   #2,a0
    move    (a2),d1
```

Diesem Programm erlaubt es, Tabellen auszulesen. Der Addiere-Befehl muss hier sein, denn wir müssen den Zeiger auf das nächste Element springen lassen, und das ist 2 Bytes (1 Word) weiter.

Jetzt wenden wir unsere neue Adressierungsart an:

```
    move.l  Tabstart,a2
    move    (a2)+,d0
    move    (a2),d1
```

Das ganze wirkt doch gleich viel übersichtlicher.

Indirektes kopieren

Doch auch Strings können über die indirekte Adressierungsart kopiert werden. `A0` soll ein Zeiger auf einen String, und `A1` soll der Zeiger auf den Puffer sein. Der String soll 7 Zeichen lang sein. Wir haben schon einmal eine Möglichkeit zur Stringverschiebung kennengelernt. Dies soll uns noch einmal bewusst werden.

```
AltesPrg:
        move.l (a0),d0
        move.l d0,(a1)
        addq.l #4,a0
        addq.l #4,a1
        move   (a0),d0
        move   d0,(a1)
        addq.l #2,a0
        addq.l #2,a1
        move.b (a0),d0
        move.b d0,(a1)
```

Mit diesem Programmteil können String mit einer Länge von 7 Bytes kopiert werden, die Voraussetzung ist jedoch, dass die Adressen gerade sind, denn sonst droht wieder ein toller, roter, blinkender Kasten mit einer 3 drin. Der Trick ist wieder folgender. Wir nutzen die 4 Bytes Aufnahmekapazität der Longs. Was uns auch hier wieder auffällt sind die Additionen, die je nach Länge der Übertragung natürlich verschieden sein müssen. (Einmal 4 Bytes und 2 Bytes). Zum Schluss brauchen wir nicht mehr einen Hinzuaddieren, da der String kopiert ist, und die Adresse nicht mehr angepasst werden muss. Da es sehr sehr häufig vorkommt, dass Programme oder Strings kopiert werden müssen, wenn es gerade nicht der Blitter macht, würde immer in Addierbefehl zum Zeigen auf das nächste Datum benötigt werden. Der Prozessor aber kennt aber die Möglichkeit der indirekten Adressierung mit Postinkrement, die wir gerade kennengelernt haben. Der Vorteil liegt darin, das der Additionsbefehl (2 Bytes und 4 TZ) gespart wird, und das immer die richtige Anpassung, d.h. Addition um auf das nächste Element zu kommen, jeweils um 4, 2 oder 1 ist.

Unsere verbesserte Programmversion sieht folgendermaßen aus:

```
        move.l  (a0)+,d0
        move.l  d0,(a1)+
        move.w  (a0)+,d0
        move    d0,(a1)+
        move.b  (a0),d0
        move.b  d0,(a1)
```

Nachdem der Prozessor also die Daten mit Hilfe des Zeigers in das `D0`-Register transportiert hat, wird die Adresse um 4 erhöht, da es sich bei dem Zugriff um ein Long handelte. Wichtig ist, das erst nach dem Bewegen der Daten die Adresse erhöht (inkrementiert wird). In der dritten Zeile wir die Adresse nur um Eins erhöht, da der Zugriff nur auf ein Word war.

Doch das unser Prozessor sehr mächtig ist, kennt er natürlich nicht nur die Möglichkeit über die indirekte Adressierung mit Inkrement in ein Datenregister zu schreiben, sondern auch als Ziel diesen Operanden zuzulassen. Das heißt, dass unser Programm wie folgt weiter verkürzt werden kann:

```
        move.l  (a0)+,(a1)+
        move    (a0)+,(a1)+
        move.b  (a0),(a1)
```

Toll nicht?

So sind auch einfache Zeichenumwandlungen möglich. Da wir die Schleifenbefehle noch nicht kennen, vorab ein Pseudocode. Es soll `A0` ein Zeiger auf eine Zeichenkette sein.

```
Schleife    move.b  (a0)+,d0
    WENN Ende der Zeichenkette erreicht
     DANN beende Programm
     SONST Weiter im Ablauf
    bsr WandleZeichenUm
    SPRINGE nach Schleife
```

Adressregister indirekt mit Prädekrement

Bei der indirekten Adressierung mit Prädekrement ist das Adressregister ein Zeiger, und dieser wird entsprechend der Länge der in den Prozessor eingelesenen Werte vor der Verarbeitung vermindert.

Es stehen dazu alle Register von `A0` bis `A7` zur Verfügung.

Die allgemeine Syntax lautet: `-(An)`

Ebenso wie es eine Adressierungsart mit einer anschließenden Erhöhung der Adresse gibt, so ist auch eine Gegenadressierungsart vorhanden. Jetzt wird zuerst die Adresse vermindert, und dann wird auf sie zugegriffen. Die Reihenfolge ist vielleicht so zu merken, indem man auf die Stellung des Plus- bzw. Minuszeichens achtet. Da das Minuszeichen vor der geklammerten Adresse steht, wird sie zuerst vermindert und dann benutzt, im Gegensatz zum Pluszeichen, das erst nach der geklammerten Adresse steht, und diese dann erhöht.

Adressregister indirekt mit Adressdistanz

Bei der indirekten Adressierungsart mit Adressdistanz bildet die Summe aus Adressregister und einem 16 Bit Absolutwert einen Zeiger auf den Operanden.

Es stehen dazu alle Register von `A0` bis `A7` zur Verfügung.

Die allgemeine Syntax lautet: `d16(An)`

Die Arbeitsweise wollen wir uns an einem Beispiel verdeutlichen. In dem Adressregister `A0` soll der Wert $10000 stehen. Er zeigt somit auf diese Adresse. Mit einem Absolutwert können wir diesen Bereich um Werte zwischen -32768 bis `32767 verschieben. Der Absolutwert soll einmal $1000 sein. Um nun aus der Summe $10000 + $1000 ein Byte in ein +D4`-Register zu bewegen schreiben wir folgendes.

```
        lea $10000,a0  ; oder move.l #$10000,a0
        move.b  $1000(a0),d5
```

Durch die Addition von $10000 und $1000 erhält der Rechner die benötigte Adresse von $11000, aus der er ein Byte in das `D5`-Register überträgt. Mit diesem Befehl lassen sich sehr leicht bestimmte Elemente einer Tabelle verarbeiten. Ein Beispiel. Das Amiga Betriebssystem ist sehr stark an Tabellen gebunden, die wir auch Strukturen nennen. (Näheres bei der Betriebssystemprogrammierung). Nehmen wir einmal an, wir erhalten nach Öffnen eines Fensters einen Zeiger auf dessen Struktur. An bestimmten Stellen der Struktur sind bestimmte Eigenschaften abgelegt. So auch die X und Y Koordinate des Mauszeigers. Sie befinden sich an der Position $c und $e. Wir wollen sie auslesen und in den Datenregistern `D0` und `D1` sichern. Der entsprechende Teil sieht so aus:

```
        ; Zeiger auf die Fenster Struktur in a0

        move    $c(a0),d0
        move    $e(a0),d1
```

Neben den Eigenschaften der Tabellenvereinfachung gibt es auch noch eine weitere geniale Anwendung des Befehls, und zwar auf LEA Befehle. Ich vermute, dass ich etwas Verwunderung stifte, doch soll sich diese lösen. Ich will die Anwendung einfach einmal hinschreiben. Ich wähle den Inhalt des Adressregister mit 22222.

```
        lea 100(a0),a0
```

Was passiert. Nun der Inhalt vom `A0`-Register, das ist 22222, wird mit 100 addiert und wieder in `A0` gesichert. Eine geniale Möglichkeit Adressregister zu erhöhen, die sonst über den `add.l#`-Befehl erhöht würden. Diese Möglichkeit wird man z. B. in IFF-Entpackern finden, die die Adresse bei dem Zeilendurchlauf um 10240 erhöhen müssen. Die Ausführungszeit spricht auch hier für die Anwendung des Befehls.

Adressregister indirekt mit Adressdistanz und Index

Bei der indirekten Adressierungsart mit Adressdistanz und Index bildet die Summe aus Adressregister und einem 8 Bit Absolutwert sowie einen 32 Bit Index den Zeiger auf den Operanden.

`D8` kann Werte zwischen -128 und 128 darstellen.

Es stehen dazu alle Register von `A0` bis `A7` zur Verfügung.

Die allgemeine Syntax lautet: `D8(An,Rx)`

Auch dieser Befehl eignet sich sehr gut für Tabellenarbeiten. Er erweitert die in 4.6.6 beschriebe Adressierungsart insofern, dass er ein Indexregister erlaubt. Über diesen sind z. B. Zählschleifen möglich. Nehmen wir einmal an, wir haben wiederum eine Tabelle, auf die das `A0`-Register zeigt. An der Stelle 100, die unser `D8` aufnimmt stehen 12 Werte, die benötigt werden. Diese müssen bedauerlicherweise noch addiert werden. Was soll man tun? Unsere neue Adressierungsart hilft uns weiter.

```
        add 100(a0,d0),d1
```

Mit dieser Zeile, die wir in eine Schleife packen können, sind wir in der Lage, bei Benutzung des `D0`-Registers als Zähler von 1 bis 12, unsere benötigten Werte zu addieren.

Absolut kurz

Als Adressangabe dient ein Wort, das den Speicherplatz angibt. Die Werte sind Words, die dann auf Longs umgewandelt werden.

Die allgemeine Syntax lautet: `#xxxx.w`

Diese Adressierungsart wird wenigsten genutzt. Außer um den Speicherbereich von 0-65535 ($ffff) anzusprechen taugt der Befehl nichts. Da wir sowieso in den meisten Fällen keinen Absolutspeicher gegeben haben, sondern nur den, den uns evtl. das

Betriebssystem zur Verfügung stellt, ist der Befehl fast überflüssig geworden. Wenn wir allerdings Betriebsystemfern programmieren, und uns die Speicherorganisation nicht mehr interessiert, können wir fleißig in den Bereich reinschreiben, denn der Zugriff ist schneller.

Absolut lang

Als Adressangabe dient ein Long, das den Speicherplatz angibt.

Die allgemeine Syntax lautet: `#xxxx.l`

PC Relativ mit Adressdistanz

Bei der indirekten Adressierungsart mit Adressdistanz bildet die Summe aus dem Programm-Counter und einem 16 Bit Absolutwert einen Zeiger auf den Operanden.

Die allgemeine Syntax lautet: `D16(PC)`

Die Anwendung der Adressierungsart ist hier und da zu finden. Sie bietet einige Vorteile bei der Variablenverarbeitung. Vorerst jedoch ein Zitat aus einem anderem Assemblerbuch: „Für die bedauernswerten Menschen, die trotzdem PC-Relativ programmieren möchten ...“. Programmiere ich also PC-Relativ, bin ich also bedauernswert, ach ja? Ich muss vorerst einmal einräumen, dass diese Adressierungsart durchaus nicht mit mehr Denkverarbeitung verbunden ist, denn bei normalen Programmen kommt hinter den Variablennamen nur noch `(PC)`. Dies sieht dann etwa so aus:

```
        move.l  Wert(PC),d0
```

Ich glaube, dass 4 Bytes Programmtext zu verkraften sind. Einen Nachteil dieses Befehls sehe ich nicht, und jetzt kommen wir zu den Vorteilen. Da “Wert“ einen Speicherstelle von 32 Bit darstellt, man aber bei der Relativen Programmierung meist keine großen Distanzen zwischen Programmzähler und Variablen auftauchen, wird zur genauen Speicherdefinition nur 16 Bit benötigt, eine Platzersparnis von 2 Bytes. Zudem ist PC relative Programmierung immer schneller als Absolute. Als Dritter Punkt wäre noch die interne Programmspeicherung zu nennen. Da das Amiga Betriebssystem ein Multitasking System ist, das mehrere Prozesse (Programme, Tasks) quasi gleichzeitig ablaufen lässt, müssen alle Programme an jeder Speicherstelle laufen können. Das OS (Operation System, Betriebssystem) stellt den Speicher zur Verfügung (er organisiert ihn), wir können nicht davon ausgehen, das sie immer an der selben Adresse laufen. Programme, die überall im Speicher stehen können, werden relokatibel genant. Benutzen wir daher im Programm Absolutwerte, so werden diese aus dem Programm herausgefiltert und in einem Hunk, der vor jedem Programm zu finden ist, für den Computer verständlich abgespeichert. Durch diese Hunk-Liste kann er dann an jeder anderen Speicherstelle das Programm zusammensetzen. Benutzten wir eine PC-Relative Programmierung, so kann dieser Eintag in die Hunk-Tabelle ausbleiben, das Programm wird also noch kürzer.

Ich wette, dass bei jedem guten Programm, das schnell ablaufen muss, PC-Relativ programmiert wird. (Man denke auch nur an den Cache!)

Für alle die, die sich nicht die Arbeit mit den „(PC)“ machen wollen, aber die Geschwindigkeit und kürze noch nachträglich in die Programme einbauen wollen, bietet der Devpac Assembler eine PC-Option `p+` an, auf die wir aber später noch einmal zu sprechen kommen, wenn es darum geht verschiebbare Programme zu schreiben.

PC Relativ mit Adressdistanz und Index

Bei der indirekten Adressierungsart mit Adressdistanz und Index bildet die Summe aus dem Programm-Counter und einem 16 Bit Absolutwert, sowie einem Index den Zeiger auf den Operanden.

Die allgemeine Syntax lautet: `D16(PC,Rx)`

Die Anwendung ist eine Mischung aus den obigen. Sie setzt sich zusammen aus der PC-Relativ Programmierung und der aus Kap. 4.6.7 (???). Die Anwendung ist hier dürftig, aber kann in manchen Fällen geschickt angewendet werden. Wiederum ist die Tabellenmöglichkeit, die als gutes Beispiel für die Anwendung vorausgeht.

Anstatt

```
        lea Tabelle,a0
        move.b  0(A0,d0)
        .
        .
Tabelle dc.b    1,2,3,4
```

lässt sich viel einfacher

```
        move.b  Tabelle(PC,d0)
```

schreiben. Dies ist kürzer und schneller. Wird jedoch der MOVE Befehl in einer Schleife verwendet, und der `A0` Zeiger nicht verändert, dann ist natürlich die erste Variante schneller, dass die Informationen über die Lage der Tabelle (Adresse) nicht erneut zur Verfügung stehen muss.

Konstanten-Adressierung

Man bezeichnet mit der Konstanten-Adressierung, oft auch unmittelbare Adressierung genannt, eine Möglichkeit, Absolutwerte (Konstantwerte) zu benutzen.

Es stehen dazu alle Register von `D0` bis `D7` bzw. `A0` bis `A7` zur Verfügung.

Die allgemeine Syntax lautet: `#Rx`

Wir haben in unseren früheren Programmen diese Adressierungsart schon kennengelernt. Sie muss bei Absolutwerten angewendet werden, wenn diese nicht im Speicher stehen. In vielen Fällen kann hier optimiert werden, zum Beispiel durch den MOVEQ Befehl.

Die Logikbefehle

Die Logikbefehle erweitern das Spektrum der mathematischen Möglichkeiten, die im 68000 stecken. Durch sie werden Verknüpfungen möglich, die uns bestimmte Werte miteinander verbinden lässt. Die Anwendungen sind nicht zu häufig, jedoch werden sie gerne zum Löschen von Hi-Long Words, zur Ausmaskierung gewisser Bit (einzelne Bit werden gesetzt oder gelöscht) und zum Setzen von ganzen Bitpaletten für die Coprozessoren verwendet.

Um die Funktionsweise zu erklären, müssen wir wieder auf die unterste Zahlenebene, der Binärebene herunter, denn bei allen logischen Befehlen wird die Arbeitsweise nur durch Betrachten der Binärzahlen sichtbar, jedoch nicht durch die Dez- bzw. Hex-Werte.

Die Und-Verknüpfung (AND)

Die Und-Verknüpfung (auch Konjunktion oder Logisches Produkt genannt) soll die erste sein, mit der wir in den Katalog der Verknüpfungen einsteigen. Als Grundbeispiel sei auf eine einfache logische Schaltung verwiesen. Wie haben eine Lampe, die durch zwei in Reihe geschalteten Schalter gesteuert werden kann. Das Ergebnis kann nur ein Brennen oder ein nicht Brennen sein, nicht jedoch ein halbes. Es werden also zwei geschlossene Schalter zum Brennen der Lampe benötigt. Wenn einer der beiden Schalter nicht geschlossen ist, so bleibt die Lampe dunkel. Wir können also folgende Tabelle aufstellen.

Schalter 1	Schalter 2	Resultat
Geschlossen	Geschlossen	Lampe brennt
Geschlossen	Nicht Geschlossen	Lampe brennt nicht
Nicht Geschlossen	Geschlossen	Lampe brennt nicht
Nicht Geschlossen	Nicht Geschlossen	Lampe brennt nicht

Die Prädikate geschlossen und nicht geschlossen, sowie Lampe brennt und Lampe brennt nicht, lassen sich auch wie folgt in einer Tabelle darstellen. Dabei soll eine Funktion UND definiert werden, mit dem Zeichen „&“.

Ein 1	Ein 2	Ein 1 & Ein 2 = Resultat
1	1	1
1	0	0
1	0	0
0	0	0

Da wir natürlich nicht nur ein Bit haben sondern 8, 16 oder 32, so werden diese immer zusammen betrachtet. Unsere erste Long-Zahl soll z. B. %1111100010101000 und die Zweite %0101010111001101 sein. Dann liefert die Verknüpfung durch ein Logisches Und das Resultat:

```
%1111100010101000
%0101010111001101
```

%0101000010001000

```
Im Programm wir dies so aussehen:
```

```
move    #%0101010111001101,d0
and     #%1111100010101000,d0
```

```
Die Oder-Verknüpfung (OR)
~~~~~~~~~~~~~~~~~~~~~~~~~

Die Oder-Verknüpfung (Disjunktion) lässt sich auch mit Hilfe von Schaltern vorstellen.
Im Gegensatz zum Und sind die Schalter jedoch parallel geschaltet, nicht
in Reihe. Sollte also ein Schalter geschlossen sein, ein anderer jedoch
nicht, so wird ein Signal vorhanden sein, da der Strom immer noch über
den geschlossenen Kreis fließen kann.

Wiederum soll die Verknüpfung über eine Funktion definiert werden, die
das Zeichen „+“erhält.

[cols="^,^,^", options="header"]
|=======================================================================
|*Ein 1* |*Ein 2* |*Ein 1 & Ein 2 = Resultat*
|1 |1 |1
|1 |0 |0
|1 |0 |0
|0 |0 |0
|=======================================================================

Unsere Bytes aus dem obigen Beispiel würden also mit der Oder-Funktion
```

```
verknüpft folgendermaßen aussehen:

  %1111100010101000
+ %0101010111001101
  ------------------
  %1111110111101101
```

```
Die Exklusiv-Oder-Verknüpfung (XOR)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Die Exklusiv-Oder-Verknüpfung, in Programmiersprachen auch unter XOR
bekannt, ist eine exotische Funktion. Sie löscht nur dann das Bit, wenn es
in dem ersten Ausdruck genauso ist wie im zweiten Ausdruck -- an
derselben Position versteht sich. Wenn dies nicht der Fall ist, so wird
das Bit gesetzt. Den genaueren Zusammenhang ist aus der Tabelle zu
entnehmen.

[cols="^,^,^",options="header"]
|=========================================================================
|Ein 1 |Ein 2 |Ein 1 XOR Ein 2 = Resultat
|1 |1 |0
|1 |0 |1
|0 |1 |1
|0 |0 |0
|=========================================================================

Diesen Sachverhalt sollten wir wiederum an einem Beispiel klarmachen.
```

```
     %1111100010101000
XOR  %0101010111001101
     ------------------
     %1010110101100101
```

```
Die Negierung (NOT)
~~~~~~~~~~~~~~~~~~~

Die Negierung (Nicht) ist ein interessanter Befehl, der doch ziemlich
simpel arbeitet. Er dreht einfach die einzelnen Bit um. Daraus folgt,
das aus jedem nicht gesetzten Bit ein gesetztes wird und umgekehrt.
Somit benötigen wir nur eine Quelle. Ein

Beispiel:
```

NOT %0101010111001101 ---------- %1010101000110010

```
Registervertausch und Registeraustausch
```

Nun wollen wir zwei Befehle kennenlernen, die die Handhabung mit den Registern ein wenig vereinfachen sollen.

Register wechsle dich! (EXG)

Jetzt haben wir schon so viele tolle Register, da wird es manchmal schwer, auch alle zu benutzen. Doch nach einiger Zeit gewöhnt man sich an diese Tatsache (arm dran sind die, die Intel-PC-Assembler versaut sind), das so viele Register zur freien Verfügung stehen, und wird sie alle benutzen. Und wenn einmal ein Programm steht, und alle 8 Register gnadenlos ausgenutzt werden, passiert es, das sich die Registerinhalte im Weg sind, und einfach in falschen Registern stehen. Anstatt nun Koordinaten im `D0` und `D1`-Reg. zu übergeben, kommt doch eine Routine so frech daher, und verlangt die Werte in umgekehrten Registern, also in `D1` und `D0`. Tja, was nun bleibt, ist ein typischer Vorgang, gerade in höheren Programmiersprachen, die Variablen zu vertauschen. Die Definition einer Hilfsvariablen, oder eines Austauschregisters ist angesagt. Ein Register-Austauschprogramm mit dem Registerpaar `D0`/`D1` und dem freien Register `D7` sieht dann so aus.

```
        move    d1,d2
        move    d0,d1
        move    d2,d0
```

oder etwa

```
        move    d0,d2
        move    d1,d0
        move    d2,d1
```

Um die Sache etwas schneller zu fabrizieren, deckten die Motorola Entwickler den Chip noch mit einem Register-Austauschbefehl ein. Er lautet EXG, und tauscht den Inhalt von zwei Registern 32-Bitig aus. Er kann in drei Kombinationen auftauchen:

```
        exg Dx,Dy
        exg Ax,Ay
        exg Dx,Ay
```

Doch wollen wir einmal richtig kritisch sein, und überlegen: “Wann brauchen wir denn einmal einen Registeraustauschbefehl?“ Um der Lösung auf die Spur zu kommen, generalisieren wir zuerst einmal die Fragestellung und fragen: „Wann werden überhaupt Werte vertauscht? “ Ein sinnvolles Beispiel sind Sortierroutinen, denn dort werden Elemente ausgetauscht, wenn z. B. ein großes Element am Anfang der Liste steht, und gegen ein Kleines, das sich am Ende befindet, ausgewechselt werden soll. Doch Halt, nichts übereilen, denn wenn wir auf die Elemente der Liste zugreifen, verwenden wir meist eine komplexe Adressierungsart, und keine Datenregister. Schon sind wir mit unserem EXG-Befehl an der falschen Adresse, denn er unterstützt nur den Austausch von Registern, nicht von Speicherzellen. Da urteile ich doch schnell: “Tja Motorola, leider daneben entworfen. Wenn ihr schon einen Schritt voraus sein wollt, dann erlaubt einen Befehl wie z. B. EXG <EA>, <EA>, das wäre etwas Innovatives!“

Warum gibt es den Befehl EXG?

Nun wirft sich die Frage auf, wenn der Befehl schon am Sortierprogramm vorbei implementiert wurde, wofür dann überhaupt. Meines Erachtens gibt es dafür drei Erklärungsansätze:

- So ein Befehl ist toll, und erhöht den Befehlssatz.
- Ein EXG besteht nur aus 2 Bytes, und eine Moverei schon aus 6.
- Dieser Befehl ist evtl. sinnvoll für Compiler.

Nun, wenn Motorroller nur den Befehlssatz in die Höhe treiben wollte, wären noch ganz andere Befehle implementiert worden, schauen wir uns doch näher Argument b und c an. Es ist richtig, das ein EXG nur 2 Bytes kostet. Das ist natürlich ein Argument, denn wir sind ja verwöhnte ich-will-kurzen-Code-Programmierer (IWKC-Programmierer). Doch neben der Länge gibt es ja auch noch unser Nebenprodukt, die Geschwindigkeit. Ein MOVE auf Datenregister kostet, wir erinnern uns, 4 TZ, d.h. für drei Zeilen sind wir schon 12 TZ quitt. Der EXG kostet uns nur 6TZ. Toll nicht! (Ich glaube, ich sollte diese Geschwindigkeits-Euphorie langsam abbauen, aber es ist wichtig, dass dies mal erwähnt wird. So!). Doch was haben wir denn davon, wenn wir mal das mit der Geschwindigkeit außer acht lassen. Wir sollten einmal eine andere Frage stellen, ein Problem ansprechen, das bis jetzt noch nicht angeschnitten wurde. Es ist die Frage nach der Busbenutzung und Prozessorbelastung. Noch einmal müssen wir unsere grauen Zellen weiß werden lassen, und dann leuchtet uns vielleicht ein, warum ein Befehl doch besser sein kann als mehrere einzelne. Wenn ein Befehl vom Prozessor bis zum Ende durchgezogen wird, durchläuft er einen Zyklus von 3 Stationen: Fetch, Decode und Execute. Sollten wir in unserem Fall des EXG-Ersatzes drei MOVE’s einsetzen, würde dreimal ein Fetch, Decode und Execute Prozess angeleiert. Bei Benutzung der Light-Version nur einen Zyklus. Doch wir haken weiter nach, warum es sinnvoll ist, einen Befehl zu benutzen und nicht drei, denn es erscheint ja gleichgültig, wie oft der Zyklus durchlaufen wird, denn ein Zyklus ist prozessorintern, und wir haben damit nicht viel am Hut. Ich greife hier schon ein bisschen vor, denn es ist ein Vorgang, den wir noch bei einem weiteren Befehl kennenlernen werden, der auch auf dem ersten Blick etwas sinnlos erscheint. Es handelt sich dabei um den Vorgang des Task-Switchings. (Später in `Exec` mehr dazu). Vorab sei erwähnt: Der Amiga unterbricht dauernd seine aktuell laufenden Programme (Prozesse), um noch einige Dienstaufgaben zu erledigen, die so anfallen (wie z. B. Mausabfrage). Auch ganze Programme werden quasi gleichzeitig ausgeführt, in dem sie schnell hin und her geschaltet werden. Der Vorgang ist also folgender: Unterbreche nach einer bestimmten Zeit das gerade laufende Programm, sichere alle Register und Status, und starte dann ein neues Programm. Wo nun das laufende Programm unterbrochen wird, ist willkürlich, und der Anwender erfährt nichts davon. Nun kann es ja sein, dass der Programmierer A soeben eine Variablen-Vertausch-Routine geschrieben hat. Leider kannte er EXG noch nicht, und er wählte die 3-MOVE-Methode. Doch nach dem zweiten MOVE funkt Anwender B’s Programm (Task) dazwischen, und stört seinen Vertausch. Schade eigentlich, denn nachdem die Zeit für den Task von Benutzer B abgelaufen ist, geht es noch mit dem dritten MOVE zum Vertausch weiter. Also, was lernen wir daraus? Wenn es geht, immer einen komplexen Befehl benutzen, sie sind in der Regel auch schneller. Es können auch nur abgeschlossene Befehle unterbrochen werden (d.h. 2 Unterbrechungen in 3 MOVE’s sind möglich), nicht allerdings in der Reihenfolge Fetch, Decode, Execute. Ein einzelner Befehl wir also immer abgearbeitet, und die Arbeit ist erledigt, währenddessen bei Aufblähung die Aufgabe durch Task-Switching (so nennt man das Umschalten der Tasks) immer noch nicht geregelt wurde.

Compiler bestimmen den Befehlssatz eines Prozessors

Reden wir noch kurz über den letzten Punkt. Was Compiler alles brauchen können, kann man so gar nicht abschätzen, denn viele brauchen die unmöglichsten Befehle, die normalerweise im normalen Programmierleben nur vom hören bekannt sind. In dieser Kategorie lässt sich auch unser EXG einordnen. Ein Compiler geht ja immer nach sturen Regeln vor, und so kann es einmal passieren, dass die Register nicht da sind wo sie eigentlich hingehören. Dies kann bei Funktionsaufrufen in Ausdrücken auftreten, jedoch ist dies wirklich eine Ausnahme und kein Regelfall.

Wenn wir noch einmal die Beweggründe der 68000 Entwickler zu unserem Chip in unser Ohr holen, so kommt ein besonderes Argument der Entwickler vor, das der “Höheren Programmiersprachenunterstützung“. Und so dachten sich die Entwickler, “Ach, wir basteln einfach mal so ’nen Befehl rein, der vielleicht mal so von ’nem Compiler gebraucht wird“. (Dies dachten sie natürlich in Englisch!) Na ja, was daraus geworden ist, kennen wir nun; einen Befehl der für seine Anwendung einfach genial ist, nur ist der Anwendungsbereich sehr klein! Gag bei Seite, die Designer haben sich natürlich etwas dabei gedacht. Der Befehl wird von Compilern wesentlich mehr genutzt als von Assemblerprogrammierern.

Die letzten werden die ersten sein (SWAP)

Es ist toll einen Rechner mit 32 Bit zu besitzen. Und dann noch 32-Bit Adressen, ho, ho, wer da noch aus alten Zeiten einen 64er programmierte, kann ein Lied von dem Speicherzugriff singen. Denn dort gab es noch nicht einmal 16-Bit-Register, die den Speichern adäquat ansprechen konnten. Damals musste man noch aus zwei 8-Bit Registern und indirektem Speicherzugriff auf flexible Adressen zugreifen. Ja, ja, das waren noch Zeiten. Aber auch ein PC mit veraltetem Prozessor, der ja nicht das gelbe vom Ei ist, hat so seine Schwierigkeiten in Bezug auf die Speicherverwaltung. Gerade dieses Mischen von Word, Byte, und neuerdings auch Long-Adressen (mit neuerdings meine ich unsere PC Freunde), hat man hin und wieder so seine Problemchen. So machte man sich in der Motorola Werkstatt ran, einen Befehl zum Handhaben beider im 68000 zu implementieren. Der Befehl lautet SWAP und er kann Register-Hälften vertauschen. Doch was wird vertauscht? Da unser Prozessor 32-Bit Register zum Standard gemacht hat, bietet sich ein Vertauschen der höheren 16 Bit mit den niedrigen 16 Bit (oder auch Umgekehrt, wie man ’s nimmt) an. Der SWAP Befehl ist also ein vergleichbarer EXG, nur, dass nicht verschiedene Register vertauscht werden, sondern in einem Register die Highs und Lows.

Wer verwendet SWAP?

Nun kommt natürlich die Frage nach der Anwendung, doch es sei gesagt, Beispiele zu SWAP zu suchen ist wirklich nicht schwierig. Wir kennen ja nun schon den MUL/DIV-Befehl, egal ob mit oder ohne Vorzeichen, der ja so das Manko hat, nur 16 Bit zu verarbeiten. Im späteren Verlauf, wenn Programmierbeispiele genannt werden, lernen wir noch ein paar Unterprogramme kennen, die dieses Manko beheben. Dann arbeiten wir auch mit SWAP, um an den Nachkommarest heranzukommen, der dann weiterverarbeitet werden kann. Doch wenn wir mit normalen Befehlen auf Word Basis arbeiten, wir auch nur der unterste Teil eines Longs genutzt. Dazu vielleicht noch eine kleine Skizze, die die Belegung von Bytes oder Words innerhalb des 32-Bit Registers

deutlich machen.

Befehl	Registerbelegung D0
move.l #$12345678,d0	$12345678
move #-1,d0	$1234ffff
move.b #$44,d0	$1234ff44
moveq #$12,d0	$00000012

Da das Register immer 32-Bitig ist, werden bei Operationen, die diese 32 Bit nicht ausfüllen, d.h. bei Word oder Byte Operationen, die restlichen 16 oder 24 Bit nicht gelöscht oder überschrieben. Im ersten MOVE, setzen wir die Long Zahl $12345678 ins D0-Register. Danach verwenden wir ein Word Zugriff (-1 in D0) auf dieses Register. Es werden jetzt nur die unteren 16 Bit erneuert, d.h. das Lo-Word, nicht aber das Hi, welches unverändert bleibt. Beim dritten Befehl ist dies ähnlich, nur, das hier ein Byte ($44) in das D0-Reg. geladen wird. Die obersten 24 Bit bleiben, wie bekannt, unangetastet. Was jetzt noch folgt ist ein Quicky, der demonstrieren soll, dass er auf 32 Bit Einfluss nimmt, und nicht auf seinen Zahlenbereich, der ja eigentlich nur durch 7 Bit plus Vorzeichen gegeben ist (also Byte), achtet.

Etwas Hardware

Doch was hat dies alles mit dem SWAP zu tun? Nehmen wir einmal ein Beispiel aus dem Hardware Bereich. Wir haben eine Adresse, die auf den Bitplaneanfang zeigt. (Wer dies nicht versteht, den vertröste ich auf die später folgenden Kapitel.) Die Adresse lautet $612345. Nun gibt es bestimmte Speicherbereiche, die die Hardware zur Darstellung der Grafik nutzt. Die Adresse, in der nun der Bitplaneanfang geschrieben werden kann, lautet $dff0e0, oder mit einem anderen Namen BPL1PTH. Nur kann einfach die Adresse $61234 in $dff0e0 geschrieben werden, natürlich als Long, und $dff0e0 und $dff0e2 werden dabei aktualisiert. Doch gibt es auch eine andere Möglichkeit, Adressen in diesen besonderen Speicherbereich zu schreiben. Neben dem Prozessor gibt es noch den Copper, der die Möglichkeit besitzt, eben diese Speicherbereiche zu verändern, und mit Werten zu laden. Doch wie der Schicksal es so will, können nur Words verschoben werden. Was uns nun übrigbleibt, ist ein Long in zwei Words aufzuspalten. Wenn $61234 als Long in $dff0e0 geschoben werden soll, so kann man äquivalent auch die 6 in die Speicherzelle $dff0e0 und $1234 in $dff0e2 schieben. Doch wie sage ich dem Copper, das der diese Words in den Speicher bewegen soll? Dazu dient eine sogenannte Copper-Liste, die dann nach dem starten automatisch abgearbeitet wird. Sie hat in unserem Beispiel folgenden Aufbau:

```
        dc  $0e0,$0006
        dc  $0e2,$1234
```

Was auffällt, ist das Fehlen von $dff vor dem $0e0 bzw. $0e2. Dies muss aber weggelassen werden, da der Bereich, auf dem der Copper zugreifen kann sowieso begrenzt ist (Leider). Gehen wir jetzt noch einen Schritt weiter. Die Adresse, an der die Bitplane liegt, ist oft variabel, so dass vorab noch gar keine feste Adresse vorliegt? Was gibt es also zu tun? Es ist ganz einfach, wir besorgen uns das Hi und Lo der Adresse, und schreiben diese Werte getrennt in die Copper-Liste. Die allgemeine Copper-Liste hat dann folgenden Aufbau.

```
        dc  $0e0
BPL1PTH dc  0
        dc  $0e2
BPL1PTL dc  0
```

BPL1PTH und BPL1PTH stehen dabei für das Lo und Hi. Die genauere Erklärung folgt im Kapitel über Hardware Programmierung. Nun gilt es also, eine Zahl, wie z. B. $123456 auf diese beiden Register zu verteilen. Von Hand ist dies einfach, denn es lässt sich leicht ersehen, das ein

```
        move    #$12,BPL1PTH    und
        move    #$3456,BPL1PTL
```

dies erledigt. Doch wie schon erwähnt, handelt es sich meist um Konstanten. Lassen wir doch einmal die Konstante $123456 im D0-Reg. stehen, und verfolgen den automatisierten Vorgang. Er lässt sich mittels SWAP wie folgt formulieren:

```
        move.l  #$123456,d0     ; ins d0-Reg.

        move    d0,BPL1PTL      ; Low in Register Lo
        swap    d0              ; Low -> Hi, Hi-> Low
        move    d0,BPL1PTH      ; Low von d0 in Reg. Hi
```

Das war 's. Vom Prinzip her ganz einfach. Es wird einfach das Low von D0 in BPL1PTL gesichert, denn es ist ja kein Problem, angesichts des Word Zugriffs auf die untersten 16 Bit. Da immer noch $12 im Hi und $3456 im Low stehen, ist es sinnvoll diese nun zu vertauschen. Das heißt, dass nach dem SWAP der Wert $3456 im Hi und umgekehrt $12 im Lo steht. Dies ist dann praktische, denn das Lo, das ja das Word darstellt, kann einfach in BPL1PTH gesichert werden. Auch geht der Wert $123456 nicht verloren, sonder kann durch ein weiteres SWAP wieder hergestellt werden.

Die Schiebebefehle

Wir erklären die Schiebebefehle genau so wie die Logik-Befehle. Zum besseren Verständnis wollen wir die einzelnen Bit betrachten, mit den Hex- oder Asc-Werten kommt man nicht so weit.

Bitweises, Logisches Linksschieben (LSL)

Mit dem ASL-Befehl kann ein Register oder eine Speicherstelle um eine oder mehrere Stellen verschoben werden. Der LSL Befehl führt einen Logic-Shift-Left durch. Daher die Abkürzung LSL. Das folgende Beispiel macht die Arbeitsweise deutlich

```
        move.l  #%00000111,d0
        lsl.b   #5,d0
```

Der Absolutwert %00001111 (Dezimal 7) wird um 5 Stellen nach links geschoben. Natürlich entsteht rechts ein Loch, welches aber mit Nullen aufgefüllt wird. Aus der 7 wird nun %11100000 (240). Wir können dies leicht beobachten, da es sich um Binärzahlen handelt, jedoch kann niemand einen Zusammenhang zwischen den Dezimalzahlen 7 und 240 vermuten, wenn er nicht die Bitmuster als Vergleichspunkt nähme.

Bitweises, Logisches Rechtsschieben (LSR)

Der LSR Befehl (Logical-Shift-Right) arbeitet ähnlich wie der LSL Befehl, jedoch hat dieser Links entstandene Lücken mit Null zu füllen.

Arithmetisches Linksschieben (ASL)

Dieser Befehl unterscheidet sich nicht vom LSL Befehl.

```
        move.l  #%10111011,d0
        asl.b   #3,d0
```

Aus %10111011 (187) wird %11011000 (216)

Arithmetisches Rechtsschieben (ASR)

Ich habe bei dem ASL Befehl stillschweigend gesagt, dass er derselbe sei wie der LSL Befehl. Aber geht denn das? Warum ist denn da kein Unterschied? Warum dann die Existenz dieses Befehls.

Ein Anstatt ist in der Historie und der Bedeutung der Abkürzung ASL bzw. ASR. Das „A“ am Anfang steht immer für „Arithmetic“, sodass die Abkürzung komplett für „Arithmetic Shift Left/Right“ steht. Vom LSL/LSR unterschieden sich die Mnemonics durch das „L“ der „Logic“.

Bei dem arithmetischen Schieben wird das 7. Bit als Füller genommen. Wenn beim logischen Verschiebevorgang von rechts nach links auf der rechten Seite Lücken entstehen, dann werden diese beim Lücken durch Nullen ersetzt. Nicht jedoch bei dem arithmetischen Verschieben nach Rechts. Das siebte Bit ist ja bei Vorzeichen-Zahlen immer Indikator, ob die Zahl negativ (7. Bit gesetzt) oder positiv (7. Bit gelöscht) ist.

Ist das siebte Bit nicht gesetzt, so verhält sich der ASR¬-Befehl wie ein LSR-Befehl. Die Lücke wird durch Nullen gefüllt.

```
        move.l  #%01010101,d0
        asr.b   #1,d0
```

Aus %01010101 (85) wird %00101010 (42); kein Unterschied zu LSR.

Ist jedoch das siebte Bit gesetzt, so wird mit diesem gefüllt, d.h. nicht mehr mit Null sondern mit eins.

```
        move.l  #%10010111,d0
        asr.b   #3,d0
```

Aus %10010111 (151) wird 11110010 (142); großer Unterschied

Da bei dem Verschieben nach Links das 7. Bit sowieso wegfällt kann dies nicht zur Betrachtung als Füllkörper herangezogen werden. Warum es ein ASL Befehl gibt kann ich mir nur so erklären, das man zu dem ASR Befehl ein Äquivalent wollte, den ASR Befehl.

Bitweises Linksrollen (ROL)

Bei den Rotationsbefehlen wird eine Seite nicht mehr mit Nullen aufgefüllt, sondern der herausfallende Teil kommt auf der anderen Seite wieder herein.

```
        move.l  #%10111011,d0
        rol.b   #3,d0
```

Aus %1110011 (230) wird %00110111 (55)

Bitweises Rechtsrollen (ROR)

Rollen, nur nach rechts.

```
        move.l  #%11110000,d0
        ror.b   #5,d0
```

Aus %11110000 (240) wird %10000111 (135)

Optimieren von mathematischen Ausdrücken durch Schieben

Mit Schiebebefehlen können wir in den meisten Fällen die Multiplikation mit Konstanten vermeiden. Wir wollen hier eine Reihe von Multiplikationen anführen.

Auf eine Reihe von Gegebenheiten muss allerdings geachtet werden:

- Man sollte immer den Definitionsbereich beachten!
- Welche Register sind frei, und welche belegt?

Ich habe bei der Erstellung der Optimierungen immer eine Abkürzung genommen, um allgemein bei den Registern zu bleiben, wie z. B. `Dx` und `Dx+1` (wie bei der MACRO-Definition), doch habe ich hier darauf verzichtet und verwende nur die Register `D0`, `D1` und `D2`.

Multiplizieren mit 2

ist einfach möglich durch einmaliges Schieben nach links.

```
        lsl d0
```

Anhand eines Bit-Beispiels ist dies vielleicht etwas deutlicher zu verstehen, warum denn ausgerechnet der Wert mit 2 multipliziert wird. Als Zahl soll die 13 dienen. Die Bitkombination sieht wie folgt aus: 13 = %0000001101. Nach Shiften ergibt sich: 26 = %0000011010

Da von Bit zu Bit immer um Zwei multipliziert wird, wird der Wert also verdoppelt. Apropos verdoppelt: wir erinnern uns, eine Multiplikation mit zwei kann auch als

```
        add d0,d0
```

geschrieben werden. Mit Hilfe der ADD- und LSL-Befehle will ich nun einmal einige Multiplikationen erklären.

Multiplikation mit 3:

Bei der Multiplikation mit 3 sichern wir die Zahl, und addieren sie zum Produkt mit 2.

```
        move d0,d1
        add d0,d0
        add d1,d0
```

Multiplikation mit 4:

Die Multiplikation ist durch zweimaliges Addieren, oder durch zweimaligen links schieben möglich.

```
        add d0,d0
        add d0,d0
```

Multiplikation mit 5:

Die Idee von der Multiplikation mit 3, dem sichern und aufaddieren finden wir in vielen ungeraden Zahlen wieder, so auch mit der Multiplikation mit 5, die die Zahl auf das Produkt mit 4 aufaddiert.

```
        move d0,d1
        add  d0,d0
        add  d0,d0
        add  d1,d0
```

Multiplikation mit 6:

Auch mit der Multiplikation mit 6 ist es wichtig den Wert zwischen zu sichern, allerdings nicht das Original, sondern die schon mit 2 multiplizierte Zahl. Der Trick liegt darin, die mit 2 multiplizierte Zahl auf die mit 4 multiplizierte aufzuaddieren.

```
        add  d0,d0
        move d0,d1
        add  d0,d0
        add  d1,d0
```

Multiplikation mit 7:

Warum sollte man immer Additionen verwenden? Hier zeigt die Anwendung einen viel besseren Trick. Er lässt sich immer dann anwenden, wenn Zahlen, die um eins kleiner sind als eine 2^x-Zahl, multipliziert werden müssen. So z. B. bei der Multiplikation mit 7 oder auch mit 15, werden wir die Grund-Zahl einfach vom Ergebnis abziehen.

```
        move d0,d1
        lsl  #3,d0
        sub  d1,d0
```

Multiplikation mit 8:

Dies ist einfaches Verschieben.

Multiplikation mit 9:

Als wir bei der Mul 7 Subtrahierten, werden wir hier natürlich addieren, auch ein Trick, der bei Zahlen 2^x+1 gilt.

```
move d0,d1
lsl #3,d0
add d1,d0
```

Multiplikation mit 10:

Die Multiplikation mit 10 würde wieder wie bei der 6 zwei getrennte Additionen erlauben (8 und 2), doch ist dies zu langsam, und wir gehen wie bei der Multiplikation mit 9 vor, doch Addieren wir noch einmal dazu.

```
move d0,d1
lsl #3,d0
add d1,d0
add d1,d0
```

Multiplikation mit 11:

Wie Multiplikation 10, nur noch ein ADD mehr, zeigt sich die elfte Routine.

```
move d0,d1
lsl #3,d0
add d1,d0
add d1,d0
add d1,d0
```

Multiplikation mit 12:

Hier nutzen wir wieder den Zwischenspeicherschritt nach der ersten Multiplikation mit 4, denn nach der erneuten Multiplikation mit 2 sind wir schon bei 8, und 8 plus 4 sind 12 (letzte Zeile).

```
lsl #2,d0
move d0,d1
add d0,d0
add d1,d0
```

Multiplikation mit 13:

Wie 12, nur mit der Addition der originalen Zahl, die in `D2` gesichert werden muss.

```
move d0,d2
lsl #2,d0
move d0,d1
add d0,d0
add d1,d0
add d2,d0
```

Multiplikation mit 14:

Wir multiplizieren erst die Zahl mit 2 (ADD) und sichern sie im `D0`-Register. Danach wird mit 8 Multipliziert (wir erhalten als Ergebnis die Multiplikation mit 16), und die Subtraktion 16-2 ist 14:

```
add d0,d0
move d0,d1
lsl #3,d0
sub d1,d0
```

Multiplikation mit 15:

Kennen wir schon, denn es ist die Strategie, die wir von Multiplikation mit 7 kennen.

```
move d0,d1
lsl #4,d0
sub d1,d0
```

Multiplikation mit 16:

Simpel durch Schieben

```
lsl #4,d0
```

Multiplikation mit 17:

Wie immer mit aufaddieren

```
move d0,d1
lsl #4,d0
add d1,d0
```

Multiplikation mit 18:

```
add d0,d0
move d0,d1
lsl #3,d0
add d1,d0
```

Multiplikation mit 19:

```
move d0,d2
add d0,d0
move d0,d1
lsl #3,d0
add d1,d0
add d2,d0
```

Multiplikation mit 20:

```
lsl #2,d0
move d0,d1
lsl #2,d0
add d1,d0
```

Multiplikation mit 40:

```
lsl #3,d0
move d0,d1
lsl #2,d0
add d1,d0
```

Multiplikation mit 640:

Die Multiplikation mit 640 kann sich wiederum in zwei Teile zerlegen lassen: Wir Nutzen die Tatsache aus, dass 640 = 512 + 128 ist.

```
move  d0,d1
lsl.l #8,d0   ; Mal 256
add.l d0,d0   ; dann 512

lsl.l #7,d1   ; Mal 128
add.l d1,d0   ; In D0 nun Wert * 640
```

Resúmee:

Es fällt auf, dass sich Primzahlen nur sehr schwer optimieren lassen. Vor allen Dingen bei größeren Zahlen, ist bei der Programmverschnellerung Köpfchen gefragt. Die Optimierung verlangt ein plastisches Vorstellungsvermögen, die Zahlen durch geschicktes Schieben und Addieren bzw. Subtrahieren zu bilden. Wer sich etwas Üben will, der kann sich dran machen, Multiplikationen mit Primzahlen größer 100 zu entwerfen.

Das Statusregister

Das Statusregister (SR) ist ein wichtiges internes Register, auf die ein Prozessor auf keinen Fall verzichten könnte. In vielen Fällen spricht man auch von Condition Codes, oder kurz cc.

Die einzelnen Bist des Statusregisters sind sogenannte Flags. Sie geben Auskunft über den Zustand bestimmter Werte und Vergleichsergebnisse an.

Das Statusregister ist ein Word lang, und teilt sich in Anwenderbyte und Systembyte auf. Die Aufteilung sieht folgendermaßen aus:

Bitnr.	Bedeutung	Engl. Übersetzung	Kürzel
Anwenderbyte			
Bit 0	Übertrags Flag	Carry Flag	C
Bit 1	Überlauf Flag	Overflow Flag	V
Bit 2	Null Flag	Zero	Z
Bit 3	Negativ Flag	Negativ Flag	N
Bit 4	Erweiterungsflag	Extension Flag	X
Bit 5	unbenutzt		
Bit 6	unbenutzt		
Bit 7	unbenutzt		
Systembyte			
Bit 8	Interrrupt Maske I0		

Bit 9	Interrrupt Maske I1		
Bit 10	Interrrupt Maske I2		
Bit 11	unbenutzt		
Bit 12	unbenutzt		
Bit 13	Supervisor Status		
Bit 14	unbenutzt		
Bit 15	Trace Modus		

Zur näheren Verdeutlichung seien die Flags noch einmal einzeln vorgestellt.

Die Flags und ihre Bedeutung

0 - Das Übertrags Flag

Das Übertragsbit ist ein Helfer bei Additionen und Subtraktionen. Es hilft dem Programmierer herauszufinden, ob durch eine arithmetische Operation ein Übertag stattfindet.

Beispiel:

```
  11010
+ 01110

   1111    Überträge
 ------
 101000
```

1 - Das Überlauf Flag

Reicht bei einer arithmetischen Operation der Zahlenbereich zur Darstellung der Zahl nicht aus, so wird das Überlaufsbit gesetzt.

Ein Beispiel, in dem wir zwei Long-Zahlen addieren wollen:

```
  $ffffffff
+ $11111111
  --------
  $00000000
```

Das Ergebnis ist zuviel für ein Long.

Doch nicht nur bei Bereichsüberschreitungen kann dieses Flag verändert werden. Es wird ebenso bei Divisionen gesetzt, wenn der Quotient größer als 16 Bit wird, oder der Divisor zu groß ist.

2 - Das Null Flag

Das Null Flag hat einen doppelten Einsatzbereich. Zum einen zeigt es bei Vergleichen an, dass die verglichenen Werte gleich sind, und es wird ebenso gesetzt, wenn ein Operand null ist.

Der Vorteil dieses Nullflags ist seine Anwendung bei Strings, da sie mit Null abgeschlossen werden sollten. Um nun das Ende des Strings zu suchen, vergleicht man, mit in einer Schleife erhöhenden Zähler, den Buchstaben, und ist dieser Null wird das Null-Flag gesetzt und man kann das Ende des Strings somit in Erfahrung bringen.

Programm:

```
        moveq   #0,d0       ; Null-Flag wird gesetzt
        add d0,d0           ; Null-Flag ist immer noch gesetzt
                            ; da das Ergebnis Null ist
        move.l  Wert,d0     ; Null-Flag gesetzt da Wert=0
        add #123,d1         ; Null-Flag gelöscht
        rts

Wert    dc.l    0
```

Die Null kennzeichnet in der Informatik eine Aussage. Null ist gleichwertig mit einer falschen Aussage, ein anderer Wert, meist 1 oder -1 ein Wahrheitswert. Diese Technik können wir z. B. in Unterprogrammen anwenden, wo wir an das Ende jedes Programms ein Register, wählen wir schon für die Zukunft das `D0`-Register, den Wert null enthält, wenn z. B. die Operation in dem Unterprogramm erfolgreich war.

Auch das Betriebssystem arbeitet mit diesen Wahrheitswerten. Öffnen wir im vierten Abschnitt des Buches einmal eine Datei, so wird das Resultat im `D0`-Register angeben. Ob die Datei erfolgreich zu Öffnen war erfahren wird durch den Inhalt von `D0`. Ist `D0` null, so war der Versuch fehlgeschlagen.

3 - Das Negativ Flag

Es wird dann gesetzt, wenn nach einer Operation das höherwertige Bit gesetzt ist.

Beispiel:

```
%10000000   ; Bit 7 ist gesetzt, d.h die Zahl wäre negativ
```

Das Negativ Flag eignet sich besonders als Dritte Möglichkeit eines Unterprogrammresultates. Denn ist der Wert Null, so kann die erste Bedingung gelten, ist die Zahl negativ, ist es die zweite Bedingung und ist sie ungleich null kann die dritte gelten.

Auch das Betriebssystem nutzt diese Möglichkeit des Rückgabeparameters z. B. bei den Mathematikroutinen. Ist bei der Berechnung des Sinus-Wertes das Ergebnis größer Null, ist das Negativ-Flag gesetzt ist. Ist das Resultat gleich null, so war die Zahl ebenso null war, und ist Überlauf Flag gesetzt, so war der Wert zu groß.

4 - Erweiterungs-Bit

Dieses Bit ist eine Besonderheit des MC68000 Prozessors. Es wird genauso wie das Übertrags-Bit gehandhabt, unterscheidet sich aber dadurch, das bei Rotationsbefehlen dieses evtl. nicht verändert wird.

8..10 - Interrupt Maske

Die Interrupt-Maske erlaubt ein gezieltes Auslösen eines der 7 Interrupts, die der 68000 erlaubt. In der Interrupt-Maske steht eine Zahl von 1 bis 7, und ein Interrupt ist nur dann zugelassen, wenn der Wert in der Maske kleiner ist als die Prioritätsebene des Interrupts. (Doch was ein Interrupt und eine Prioritätsebenen ist, und welche Typen es da gibt, wird im Abschnitt des Betriebssystems näher beschrieben.)

13 - Supervisor Status

Der Prozessor ist in der Lage zwischen zwei verschiedenen Prozessorebenen umher zu schalten. Mit dem Setzen des dreizehnten Bit gelangen wir in den Supervisor-Mode. Dieser benutzt nun ein zweites `A7` Register und ist auch nicht in der Lage, die User-Register zu modifizieren. In diesem Modus hat man einen kleinen Vorteil, dass er etwas mehr Befehle ohne einen Absturz ausführen kann. Es sind im Wesentlichen Befehle, die Programmabläufe steuern, sie gehören alle zur Kategorie der Sonderbefehle. In diesem Modus laufen Programme der obersten Ebene ab, Programme, die im Multitasking Betrieb absturzsicher sein müssen. Denn oft hat man einen Task-Finish, aber keinen Guru, eine Leistung des Betriebssystems, die anderen Hintergrundprogramme weiterlaufen zu lassen, nicht allerdings generell zu unterbrechen, so wie es Programme unter Windows 3.x tun. Ist ein Programm dort einmal „abgeschmiert“, dann ist von einem Weiterarbeiten nicht mehr die Rede.

Unter dem Betriebssystem kann aus einem Guru wieder ausgestiegen werden, die durch Programmierfehler wie ungerade Speicheradresse, nicht bekannter Befehl, verursacht wurden.

Der Intel 80286 aufwärts kann auch zwischen zwei Ebenen umschalten, man kann ihm im Real- und Protected-Mode fahren.

15 - Trace Modus

Durch Setzen des Trace-Bit wird nach dem Beenden eines Befehls eine Exception (für uns jetzt erst mal eine Unterbrechung) ausgeführt. Somit lassen sich einfach Debugger programmieren, die nach jedem Befehl den Status der Register anzeigen. Der 68000 ist einer der wenigen Prozessoren, die die Möglichkeit des Einzelschrittmodus erlauben, denn viele andere kennen diese Möglichkeit nur über Tricks, z. B. über regelmäßige Unterbrechungen den Programmstand anzuzeigen. Doch mit dem Motorola Chip kann dies einfach über die Software geschehen, ein leichtes Spiel für Programmierer von Debuggern.

Setzen und Abfragen der Flags

Zu bemerken ist, dass nach jedem Logischen, Mathematischen, Schiebe- oder Speichermanipulations-Befehl, der ausgeführt wurde, die Flags aktualisiert, d.h. gesetzt oder gelöscht werden. Diese Information ist sehr wichtig, denn sie erspart in vielen Fällen ein neues Überprüfen der Register oder Speicherzellen auf Ereignisse.

Bedingte Sprünge

Mit bedingen Sprüngen kann der Computer auf die Resultate der Flags mit bestimmten Absprüngen regieren. D.h.: Wenn ein Flag gesetzt ist, kann durch einen Sprungbefehl ein entsprechender Absprung getätigt werden. Andersrum: Wenn das Flag nicht gesetzt ist, bleibt ein Absprung aus.

Der Befehl, der die Sprünge einleitet, ist Bcc. „cc“ bedeutet hierbei „Condition Code“, und es bezeichnet die Statuswerte. Da jedes Anwenderbit abgefragt werden kann, existieren auch verschiedene cc's. Wenn wir überprüfen wollen, ob eine Zahl gleich Null ist, also das Nullbit im SR gesetzt ist, so können wir dies mit cc „eq“ machen. Der Befehl lautet dann BEQ. Nach diesem Bedingtem-Verzweige-Befehl folgt eine Sprungadresse, die, falls die Bedingung erfüllt ist, angesprungen wird. Ist die Bedingung nicht erfüllt, so erfolgt kein Sprung. Die cc's sind in der Tabelle einmal zusammengefasst.

cc	Mnemonik	Erläuterung
Carry=0 CC	Carry	Clear
Carry=1 CS	Carry	Set
Overflow=0	VC	oVerflow Clear
Overflow=1	VS	oVerflow Set
Zerro=0	EQ	Equal
Zerro=1	NE	Not Equal
Negativ=0	PL	PLus
Negativ=1	MI	MInus

Zu Beachten: Bei allen Sprüngen werden die einzelnen Flags nicht verändert.

Buchstaben von Groß nach Klein

Nehmen wir einmal an, wir hätten ein kleines Unterprogramm geschrieben, welches Überprüft, ob wir kleine oder große Buchstaben haben. Mit Hilfe dieser Information soll ein weiteres Unterprogramm angesprungen werden, welches die kleinen Buchstaben in Große umwandelt. Falls der Buchstabe, der in die Funktion als Argument einging, ein Kleiner ist, soll als Ergebnis eine -1 (Logisches Ja) im `D0`-Register übergeben werden. War der Buchstabe schon ein großer, so soll eine Null (Logisches Falsch) übergeben werden. Nun kann mit Hilfe der BEQ (sprich: „branch equal“) und BNE („branch not Equal“) Befehle das Ergebnis ausgewertet werden, und z. B. bei einem gelöschten Null-Bit eine Routine zum Umwandeln aufgerufen werden, währenddessen bei einem gesetzten Null-Bit (Großer Buchstabe) die Umwandlungsroutine übersprungen werden kann. Ein Teilprogramm könnte so aussehen.

```
        bsr Ist_Buchstabe_Groß
```

```
        ; nun ist entweder eine Null oder eine -1 im
        ; d0 Register d.h. das Zero Bit gesetzt oder
        ; gelöscht
        beq War_Schon_Groß ; Wird angesprungen, wenn
                           ; Ergebnis falsch, d.h. Null
                           ; Bit gesetzt.
        ; nun geht es hier normal weiter.
        bsr Wandle  ; Unterprogramm zum Umwandeln aufrufen
War_Schon_Groß
        ; nun weiter im Programm
```

Vergleichsbefehle (CMP)

Mit den Vergleichsbefehlen können Ziel und Quelle verglichen werden. Je nach Ausgang des Vergleiches werden die Flags gesetzt. Der allgemeine Aufruf lautet:

```
CMP.(Typ)    Ziel,Quelle
```

Nun mag man sich fragen: Wie macht der Prozessor das? Er bildet einfach die Differenz zwischen Quelle und Ziel, und enthält dementsprechend einen Wert. Ist dieser Null, so wird im Flagregister das Nullbit gesetzt. Wenn der Wert größer bzw. kleiner ist, werden das Carry Bit und das Nullbit gelöscht bzw. gesetzt, ein genaues Wissen, auch über die anderen Zustände ist unnötig, da wir nicht umständlicher weise jedes Bit abfragen müssen, um dann das Ergebnis auswerten zu können. Wiederum helfen uns die Condition-Codes weiter, da sie bei dem Vergleich gesetzt werden. Da sehr viele Vergleichsmöglichkeiten existierten (größer; gleich; kleiner gleich; Vorzeichen ja, nein; u.v.m.) müssen einige Flags gemeinsam einen Zustand anzeigen. Wie schon oben gesagt ist es unwichtig zu wissen, dass z. B. bei einem Vergleich zweier Werte, der Wert größer war, wenn das Carry Bit und Zero Bit gesetzt ist. Somit will ich nur die Mnemonics und die Vergleichsergebnisse vorstellen.

cc	Beschreibung	Erklärung
GE	größer oder gleich	Greater Equal
GT	größer als Null	Greater Then
HI	größer	Greater
LE	kleiner oder gleich	Less Equal
LS	nicht größer	Low or Same
LT	kleiner	Less Then

Zusammen gibt es 14 verschiedene Condition Codes (CC), davon kommen 6 von Vergleichen, die anderen 8 sind in den vorigen Tabelle genannt.

BRA und BSR sind Sonderfälle des Bcc

Vielleicht wird dem ein oder anderen schon der BRA oder BSR Befehl Aufgefallen sein. Es handelt sich hierbei um eine Sonderform des Branch (Verzweige) Befehls. Zudem kommen noch 2 cc's hinzu, dass „F" und das „T", welches immer war und falsch darstellen. Jedoch können diese nicht in Verbindung mit dem Bcc-Befehl gebracht werden.

Schleifenprogrammierung

Eine der Hauptanwendungen der Vergleichsbefehle sind die Schleifen. Mit dem folgenden Programm können wir mit Hilfe der bedingten Verzweigung ein Alphabet generieren. Da die Zahlen ASCII-Zeichen darstellen, beginnen wir bei ASCII-Zeichen „A", d.h. bei 65 und verlassen die Schleife, wenn das Zeichen „Z", Dezimal 90, erreicht ist.

```
        moveq   #"A",d0
Alphabet addq    #1,d0
        cmp.b   #"Z",d0
        blo.s   Alphabet
        rts
```

In der ersten Zeile schreiben wir mit Hilfe des Quickies das "A", unsere 65, in das `D0`-Register, welches auch im Weiteren das Zählregister darstellen soll. In der zweiten Zeile addieren wir zu der 64 eine Eins, wir erhalten also 65, d.h. ein "B". Nun wird in der dritten Zeile unser Buchstabe, beim ersten Durchlauf mit "Z" verglichen. Da wir aber eine Zahl kleiner als 90 (das ist das "Z") haben, verzweigt der BLO-Befehl, da die Zahl 65 nun mal kleiner als 90 ist. Dies wir so lange durchgeführt, bis der Buchstabe nicht mehr kleiner ist, also „Z" ist.

Nun wollen wir unser entstandenes Alphabet auch in einen Puffer sichern. Dazu verwenden wir einen Zeiger, der im `A0`-Register steht, und, damit es schneller geht, unseren Endbuchstaben, „Z", im `D1`-Register.

```
        moveq   #"A",d0
        moveq   #"Z",d1
        lea Puffer,a0
Alphabet move.b  d0,(a0)+
        addq    #1,d0
        cmp.b   d1,d0
        ble.s   Alphabet
        rts

Puffer  ds.b    "Z"-"A"+1
```

Erneut ist hier eine Schleife zu finden. Jedoch benötigen wir noch weitere Zeilen (die dritte z. B.) um das Adressregister mit dem Zeiger auf den Puffer zu laden und um unser Zeichen, welches immer ein Byte ist, indirekt abzuspeichern. Erst danach (fünfte Zeile) darf die Addition erfolgen, da unser „A“ mit abgespeichert werden soll. Auch in der sechsten Zeile hat sich etwas geändert. Wir vergleichen hier nicht mehr mit einem Absolutwert, sondern mit einem Register.

Der TST-Befehl (als Sonderfall von CMP #0,ea)

Immer wieder kommt es vor, das ein Nullbyte erkannt, und demnach auch verzweigt werden muss. Da Strings heutzutage immer mit Nullbyte abgeschlossen werden, wäre ein `cmp.b #0,ea` ziemlich langsam und lang. Der Befehl kostet 2 Bytes Kennung und weitere Bytes für die binäre Null. 2 Bytes für das Null-Byte bzw. -Word und 4 Bytes für das Long. Alles dafür, das etwas mit dem Nullbyte verglichen werden soll? Das Nullbyte ist ja ein Sonderfall unter allen vorkommenden Zahlen, es ist einfach Nichts. Und daher nimmt es eine Sonderstellung ein, und es wurde ein neuer Befehl zum Vergleich mit dem Nichts erschaffen. Der TST-Befehl. Er braucht im Gegensatz zu dem CMP-Befehle keine Quelle, spart also hier schon einmal 2 oder 4 Bytes. Intern wird einfach geprüft, ob irgendein Bit ungleich Null ist. Wenn das Resultat Null ist, so wird das Zerro-Flag gesetzt, ansonsten gelöscht.

Wie schon erfahren, werden die Flags nach fast jedem Befehl aktualisiert. Wollen wir jedoch nachträglich ein Flag gesetzt haben, so müssen wir dies mit dem TST-Befehl machen, durch diesen Befehl werden die Flags neu gesetzt. Das kann insofern günstig sein, als das wir ein Unterprogramm aufrufen, und von ihm einen Wahrheitswert zurückgekommen. Vielleicht wird danach mit einer kleinen Addition die Flaginhalte zerstört. Mit dem TST Befehl kann der Zustand wieder hergestellt werden, wenn wird das Ergebnis des Unterprogramms wieder benötigen.

Flags in der Betriebssystemprogrammierung

Ein Beispiel aus der Betriebssystemprogrammierung zeigt, wie die Flags hier eingesetzt werden, und abgefragt werden müssen.

Das folgende Beispiel öffnet eine Library. Wir können den kurzen Programmausschnitt durch Ausnutzen der Flags wunderbar optimieren.

```
Original:
        lea DosName,a1
        jsr OldOpenLibrary(a6)
        tst.l   d0
        beq.s   Fehler
        move.l  d0,a0

OptFlasch:
        lea DosName,a1
        jsr OldOpenLibrary(a6)
        beq.s   Fehler
        move.l  d0,a0
```

Leider darf so ein Programm nie optimiert werden. **Alle** OS-Unterroutinen übergeben neutrale Flags, die also nicht mit dem Übergabeparameter zu tun haben müssen. Der TST-Befehl müsste also rein, oder? Nein! Dass es noch anders geht, zeigt das folgende Programm:

```
OptRichtig:
        lea DosName,a1
        jsr OldOpenLibrary(a6)
        move.l  d0,a0
        beq.s   Fehler
```

Wenn `D0` nach `A0` bewegt wird, werden die Flags gesetzt.

Stringlänge

Ein häufiges Problem ist die Stringlänge. Doch mit Hilfe des TST-Befehls können wir ziemlich leicht diese ermitteln, da wir nur bei einer Zahl ungleich Null, die ja ein normales Zeichen repräsentiert, verzweigen, und einen Zähler um einen erhöhen.

```
StringLen   ;Stringpointer in a0. Resultat: Stringlänge in d0.

        moveq   #0,d0
StrgLoop addq.b  #1,d0
        tst.b   (a0)+
        bne.s   StrgLoop
        subq    #1,d0
        rts
```

Die Arbeitsweise ist einfach. Unser Zählregister ist das `D0`-Register. Danach betreten wir schon den Bereich der Schleife, die mit einem Addiere Befehl eingeleitet wird. Unsere Stringlänge ist also schon einmal mindestens eins, auch wenn der String keine Zeichen enthält. Nach der Erhöhung kommt der Test-Befehl, und die nachfolgende Zeile verzweigt, falls `D0` ungleich Null, dem Endezeichen war. Bei Ablauf und Finden des Null-Bytes muss natürlich noch der zu früh erhöhte Zähler, das Überbleibsel aus dem Schleifeneingang, um eins vermindert werden. Ist dies getan, so haben wir in `D0` unsere Stringlänge.

Optimierungen von Vergleichen

Für die Optimierung gilt das gleiche wie für die Optimierung von anderen Befehlen auch. Doch hier zieht ein Absolut-Lang-Word Vergleich (14 TZ) schon viel mehr rein, als z. B. ein Word Vergleich (8 TZ). Daher wenden wir schon wieder einmal an, was wir eigentlich schon Wissen. Wir Move-Quicken ein Long (leider dann begrenzt) in ein freies (!) D-Register (4 TZ) und vergleichen dies mit dem anderen Wert (6 TZ).

Nicht

```
        cmp.l   #100,d0 (14 ZT) (8 Byte)
```

sondern

```
        moveq   #100,d1 (4 TZ)   (2 Byte)
        cmp.l   d0,d1   (6 TZ)   (2 Byte)
            = 10 TZ, eine Ersparnis von 4 TZ
            = 4 Bytes, 4 Bytes Ersparnis
```

Und wenn auch nur irgend jemand einen `cmp.(Typ) #0,ea` verwendet, den sollte man ...

Der spezielle Schleifenbefehl DBcc

Der DBcc Befehl (cc steht für eine der 14 verschiedenen Bedingungen, Condition Codes wie bei den Bcc-Befehlen) ist in der Lage, komplexe Schleifen zu verwirklichen. Dieser mächtige Befehl überprüft ein Datenregister auf -1 oder auf die Bedingung cc hin. Bei Nichterfüllung der Bedingung, wird zu einem Label verzweigt. Hört sich schwieriger an als es ist. An diesem Befehl zeigt sich deutlich die Mächtigkeit des Prozessors. RISC-Prozessoren bieten im Regelfall keine solchen progressiven Befehle.

Neben den bekannten 14 cc's kommt noch eine Endung hinzu. Sie unterschiedet sich etwas in der Schreibweise von einem anderen cc's. Es ist die Endung "RA", der Schleifenbefehl lautet also "DBRA". Mit einer Schleife, die durch einen DBRA-Befehl geführt wird, wir also nur das D-Reg. auf seinen Wert überprüft.

Die Bedingung ist ja, dass immer gesprungen wird. Und da „RA" ja keine Bedingung ist, müssen wir uns also einen cc suchen, der immer unerfüllt ist, denn wir wissen ja, bei unerfüllten Bedingungen wird immer verzweigt. Ja, der einigste cc ist „F", für Falsch. Der Devpac-Degugger kennt auch nur die Schleife „DBF", jedoch finde ich persönlich DBRA schöner, da besonders gut der Immer-Sprung herauskommt.

String aus Chars

Am Beispiel der Stringgenerierung soll nun gezeigt werden, wie man nach einem gestellten Problem die optimale Prozedur schreibt, denn oft gibt es mehrere Möglichkeiten ein Programm zu schreiben, d.h. zu einer Lösung zu gelangen.

Die Prozedur soll einen String mit bestimmter Anzahl Zeichen (Chars) herstellen. Dabei benutzen wir unsere favorisierte DBRA-Schleife, da sie optimal als Zählschleife genutzt werden kann.

```
DublicateChar   ; Erzeugt String in a0 aus d0-Zeichen
                ; und Anzahl in d1

        subq    #1,d1
DubChar move.b  d0,(a0)+
        dbra    d1,DubChar
        clr.b   (a0)
        rts
```

Die DBRA Zählschleife zählt das `D1` Register bis -1. Um nicht ein String aus <Inhalt von `D1` Register> + 1 zu erhalten, muss `D0` um eins vermindert werden. Da das Zeichen in `D0` steht, kann es immer an die Stelle kopiert werden, und der Zeiger wird nach Schreiben des Zeichens um eins erhöht. Nach Beendigung der DBRA-Schleife folgt noch ein CLR.B für den Abschluss des Strings, der ja nicht so einfach in der Luft hängen soll, sondern noch zur weiteren Verarbeitung zur Verfügung stehen soll.

Ein typisches Programm, das diese Prozedur nutzt, könnte so aussehen.

```
        lea Puffer,a0
        moveq   #"-",d0
        moveq   #3,d1
        bsr DublicateChar
        rts

Puffer  ds.l    10 ; 40 Bytes
```

Doch vielleicht stört einen da noch etwas. So ein dummer SUB-Befehl. Er muss ja sein, um die Anzahl der Loops der entsprechenden Länge Anzupassen. Doch warum sollte er uns nicht alle Zeichen und diese Eine mehr darstellen? Da wir uns doch sowieso darauf geeinigt haben einen String mit Nullbyte abzuschließen können wir doch einfach den an die Letzte Stelle ein Nullbyte schreiben. Doch halt, `A0` ist ja schon ein Stelle weiter, wir müssten ihn also um eine Stelle zurücksetzen. Anstatt dies jedoch mit einem erneuten SUB-Befehl zu machen, ist es viel sinnvoller die Prädekrement-Adressierung anzuwenden. Denn wir kennen ihre Anwendung. Erst abziehen, dann mit diesem Wert rechenden. Im Gegensatz zum Postinkrement, wo erst die Adresse benutzt wird, und dann hochgezählt wird. So kann an die letzte Stelle, wo eigentlich schon das Zeichen stand, ein Nullbyte angefügt werden. Der alte Wert wird somit einfach überschrieben.

Programm die Zweite:

```
DublicateChar
    move.b  d0,(a0)+
    dbra    d1,DublicateChar
    clr.b   -1(a0)
    rts
```

Alphabet generieren

Zur näheren Arbeitsweise stelle ich noch folgendes Beispiel vor. Es soll wieder ein Alphabet erstellen. Diesmal jedoch mit Hilfe des DBRA-Befehls.

```
        lea Puffer,a0
        moveq   #"A",d0
        moveq   #"Z"-"A",d1
```

```
Alphabet move.b  d0,(a0)+
        addq    #1,d0
        dbra    d1,Alphabet
        rts

Puffer  ds.b    "Z"-"A"+1
```

Es müssen 26 Buchstaben generiert werden, d.h. der Zähler muss auch auf 26-1 gesetzt werden. In der dritten Zeile geschieht das, und wir brauchen nicht mehr einen Abzuziehen, da dies schon mit der Schreibweise „Z“ - „A“ verarbeitet ist, also dies 25 darstellt. Dementsprechend muss natürlich in der Pufferdefinition Eins hinzuaddiert werden. Andernfalls würden wir 25 Bytes anfordern (macht sich in der ungeraden Bytelänge des Programms bemerkbar) und das letzte Zeichen würde in eine undefinierte Speicherstelle geschrieben werden. Der Rest erklärt sich von selbst.

Die Bitmanipulationsbefehle

Die Bitmanipulationsbefehle erlauben den Programmierern auf der untersten Ebene der Daten zu hantieren.

Sie stellen eine wunderbare Ergänzung zu den logischen Befehlen dar, da mit ihnen einfacher einzelne Bit abgefragt oder gesetzt werden können. Es können aber nichtsdestoweniger alle Bitoperationen durch logische Operationen ausgetauscht werden. Der C-64 erkennt diese Bit-Direkt-Befehle nicht, und somit war es den Programmierern nicht erspart, durch herum rollen und or-en bzw. and-en ein Register zu setzen, löschen oder abzufragen.

Der Bit Set Befehl (BSET)

Kommen wir nun zu dem ersten Befehl unserer Serie. Der Aufruf lautet:

```
BSET.(Typ)  BitNummer,WoSollsGeschehn
```

Die Bitnummer kann ein Absolutwert sein. Falls man variabel sein will, kann dort auch in einem Register stehen.

Die Zeiten dieses Befehls haben es in sich, so kostet eine einfache Operation in der Grundausstattung schon 8 TZ (Natürlich mit einem Datenregister). Doch mit einem Absolut Long sind wir schon 16 TZ los. (Schluck!)

Nehmen wir einmal an, wir wollten Bit 2,3 und 5 in der Speicherstelle $1234 setzen. Der Aufruf lautet:

```
        bset    #2,$1234    ; 20 TZ
        bset    #3,$1234    ; 20 TZ
        bset    #5,$1234    ; 20 TZ
```

Dieser Abschnitt kostet uns 20*3=60 Taktzyklen. Es handelt sich bei $1234 um ein Word, also nutzen wir dies auch.

Bitset ist oft zu langsam

Da wir hier immer dieselbe Adresse haben, können wir auch verkürzter schreiben:

```
        lea $1234.w,a0  ;  8 TZ
        bset    #2,(a0) ; 12 TZ
        bset    #3,(a0) ; 12 TZ
        bset    #5,(a0) ; 12 TZ
```

Wir kommen immerhin schon auf 8+12*3=44 Taktzyklen runter. Doch kommt es bei guten Programmen sowieso nicht vor, das jemand mehrere Bit mit dem BitSet-Befehl setzt. Viel besser ist es, sie durch die schon oben genannte Methode, die der Verknüpfung, zu setzte. Kürzer lässt sich der Sachverhalt so formulieren:

```
        or.b  #00101100,$1234.w
```

Dies kostet nur noch 20 Taktzyklen, was aber immer noch eine Menge ist. (Zuviel würde ich sagen). Und auch eine Benutzung des Datenregisters bringt keine Vorteile, denn wir benutzen nur Words, nicht aber Longs zur Verknüpfung.

Der Bit Lösch Befehl (BCLR)

Der BitClr-Befehl ist äquivalent zu dem BitSet-Befehl, nur das er Bit löscht. Auch hier ist ein schnelleres Und-en schneller, da dieser Logische Befehl schnell mehrere Bit löschen kann.

Es sind die Bit 5 und 6 des `D0`-Registers zu löschen.

```
        bclr    #5,d0   ; 14 TZ
        bclr    #6,d0   ; 14 TZ, zusammen 28 TZ
```

Schneller wäre hier wiederum ein Logik-Befehl

```
        and #01100000,d0 ; 8 TZ
```

Der Bit-Umkehr Befehl (BCHG)

Der Bit Change Befehl ist ein recht seltener Befehl. Er besitzt die Fähigkeit, Bit umzudrehen. Ist ein Bit gesetzt, wird es gelöscht, und

ist es nicht gesetzt, wird es gesetzt. Natürlich gibt es hier wieder einen Ersatzbefehl, der die Sache wieder um ein vielfaches beschleunigt. Welcher ist das wohl?

Wollten wir das vierte Bit im `D4`-Register umdrehen, dann würden wir folgendes schreiben, und aus einem gesetzten wird ein gelöschtes, und umgekehrt.

```
        bchg    #4,d4
```

Der Bit Test Befehl (BTST)

Der Bit-Test Befehl ist am Schwierigsten zu umschreiben. Mit ihm kann man einzelne Bit abfragen.

Ist das abgefragte Bit nicht gesetzt, so ist das Null Flag gesetzt, einfach zu merken, denn ist im Bit nichts, so ist auch das Null-Bit gesetzt, da Null ja auch nichts ist.

Maustaste

Eine Anwendung des Befehls wäre z. B. die Abfrage von Hardware Registern. Das Bit 6 der Speicherstelle $bfe001 gibt etwa an, ob die linke Maustaste gedrückt wurde.

```
MausWait btst   #6,$bfe001
        bne.s  MausWait
```

Dieses Miniprogramm wird noch häufiger verwendet.

Nichts tun, und dafür noch Taktzyklen kriegen!

Der Wartebefehl ist schon ein seltsamer Befehl. Sein Mnemonik lautet NOP (No OPeration), und sein Dasein beschränkt sich auf Taktzyklenverbrauch. Er kostet 4 TZ und 2 Bytes. Die bekannte Frage ist: “Wann brauchen wir deeen?“. Oho, da gibt es eine Menge interessanter Erklärungen. Zuerst einmal müssen wir die Funktion des Wartens in einem Multitasking-Rechner klären. Der NOPi eignet sich, so wie es aussieht, fantastisch als Befehl zum realisieren von Warteschleifen, die etwa so aussehen könnten:

```
        moveq   #10,d0
loop1   moveq   #-1,d1
loop2   nop
        dbra    d1,loop2
        dbra    d0,loop1
```

Woran jetzt allerdings noch nicht gedacht wurde, ist das schnelle langsam werden anderer Programme, die noch so im Speicher herumlaufen. Denn eine Warteschleife ist ein recht gemeine, Takt-schluckende Routine, denn sie nimmt anderen laufenden Task einfach die Luft weg, um so „wertlose“ Aktionen durchzuführen. Ein Beispiel. Ein Programm müht sich mit dem Berechnen von Fraktalen ab, das andere wartet gerade so 10 Sekunden, damit es mit der derzeit laufenden Diashow nicht so schnell mit den neuen Bildern weitergeht. Das dies natürlich nicht geht, und das Warten recht unwichtig ist, dürfte klar sein, doch wie geht es anders? Mit Prozessorbefehlen ist dar nicht viel zu machen, und ich verschiebe die Aufgabe bis zu einem anderen Kapitel, in dem dann die `Delay()` Funktion bekannt wird.

Selbstmodifizierende Programme

Das mit dem Warten war wohl nicht so gut, und wir müssen und wieder ein neues Anwendungsgebiet suchen. Da er weder Register noch Statusbit verändert eignet er sich recht gut als Platzfüller! Ich spreche hier einmal ein recht heikles Thema der Assemblerprogrammierung an, die Selbstmodifizierung. Hört sich gut an, nicht? Unter Selbstmodifizierung versteht man das selbstständige Ändern (Modifizieren) von Programmen.

Da bekanntermaßen jeder Befehl als Op-Code dargestellt werden kann, lassen sich auch die dazugehörigen Zahlen in Speicherbereiche laden. Es kommt dann meistens soweit, dass die Befehle dann kurz danach zur Ausführung kommen. Nach soviel Erklärung erst einmal ein Paar Beispiele, die sich gerade auf unseren Befehl NOP beziehen. Er den Op-Code $4e71. Ein Programm kann also z. B. so aussehen.

```
        bsr Up1
        bsr Up2

        move    #$4e71,RtsWeg

        bsr Up1

        rts

Up1     .
        .
        .

RtsWeg  rts

Up1     .
        .
        .
        rts
```

Was durch die Automodifizierung erreicht wurde ist leicht ersichtlich. Die Routinen `Up1` und `Up2` existieren zuerst für sich alleine. Jedoch kann es manches mal sinnvoll sein, dass nach dem Aufruf von `Up1` direkt `Up2` aufgerufen wird. In den ersten beiden Zeilen wird diese Aufgabe getrennt durchgeführt, erst wird `Up1 aufgerufen, mit RTS beendet, dann +Up2` aufgerufen, welches wiederum mit RTS beendet wird, und dann ist die Aufgabe abgeschlossen.

In der dritten Zeile wird allerdings das RTS des ersten Unterprogramms durch ein NOP ersetzt. Die Konsequenz für den Aufruf ist folgender: Ist `Up1` abgearbeitet, wird nicht mehr mittels RTS in das Hauptprogramm zurückgesprungen, sondern das NOP ausgeführt. Danach wird `Up2` ausgeführt, und das RTS von `Up2` beendet das Unterprogramm. Die Ersparnis ist klar, der Gewinn von einem Unterprogrammaufruf, der ein paar Zyklen bringen kann.

Punkte setzen bzw. löschen sind elementare grafische Aufgaben, die bei keinem Malprogramm, oder grafischen Oberflächen fehlen dürfen. Nun sind sich die Punktroutinen ziemlich gleich. Wenn ein Punkt gesetzt wird, wird ein Bit auf der Bitplane gesetzt (der genaue Zusammenhang ist unter dem Kapitel Grafik nachzulesen), und wenn der Punkte wieder gelöscht wird, wird das entsprechende Bit eben gelöscht. Wir kennen die dazu verwandten Befehle BCLR und BSET, die eben ein Bit löschen, bzw. setzten. Da der Weg über die Koordinaten bis hin zu entsprechenden Speicherzelle schon so seine Zeilchen kosten könnte (es sind im kürzesten Fall 6!, im längsten mir bekannten ca. 20), kann man ein Grundprogramm benutzen, z. B. das des Setzens, und bei Bedarf, aus dem BSET ein BCLR machen. Da die Anwendung bei Punkten wenig Sinn machen würde, da bei jedem Punktsetz-Aufruf immer ein BSET eingesetzt werden müsste, denn ein voriger BCLR würde keine Punkte erscheinen lassen, ist es logischer, komplexere grafische Funktionen zu verwenden, z. B. ein Kreis, denn dort könnte einmal der BCLR beim Löschen durch ein BSET beim Zeichnen von Kreisen ausgetauscht werden, und dann durchwegs von der Punktsetzroutine des Kreises aufgerufen werden.

```
        moveq   #100,d0
        moveq   #100,d1 ; fiktiver Mittelpunkt
        moveq   #32,d2  ; fiktiver Radius
        bsr SetCircle   ; Kreis zeichnen

        moveq   #100,d0
        moveq   #100,d1 ; fiktiver Mittelpunkt
        moveq   #32,d2  ; fiktiver Radius
        bsr DelCircle   ; Kreis löschen

        rts     ; Ende des HP s

SetCircle move    <BSET_Befehl>,Adresse
        bra.s   MainCircle

DelCircle move    <BCLR_Befehl>,Adresse

MainCircle  ; es wird ein Kreis berechnet
        .
        .
Adresse <BSET bzw. BCLR_Befehl zum setzen/löschen der Punkte>
        .
        .
        rts
```

Dies soll nur eine Möglichkeit sein, wie Kreisfunktion selbst modifizierend geschrieben werden könnte. Wir verwenden zwei Prozeduren, `SetCirlce` und `DelCircle`, um die Kreise auf dem Bildschirm zu zaubern und zu löschen. Die beiden Unterprogramme bedienen sich ihrerseits der Prozedur `MainCircle`, die allgemein für das Zeichen von Kreisen zuständig ist. `SetCircle` und `DelCircle` verändern dabei das Unterprogramm `MainCircle` insofern, als sie den für sie benötigten Befehl, der hier ganz allgemein durch BCL/SET-Befehl angegeben ist, in das Unterprogramm an der passenden Adresse einsetzen.

Gefährlich! Das hört sich ja alles ganz gut an mit der Selbstmodifizierung, jedoch hat die Sache einen ganz entschiedenen Nachteil in Bezug auf die Kompatibilität mit anderen Prozessoren. Nehmen wir zum Beispiel einen aufgemotzten A4000 mit 68040 Prozessor. Die Prozessoren 68020 aufwärts zeichnen sich durch einen neuartigen Speicherraum aus, der Cache genannt wird.

In diesem Speicher werden schon Befehlszeilen hereingeholt, ohne das sie abgearbeitet wurden. In der Praxis heißt das: Wenn ein Programm 200 Zeilen lang ist, und davon erst 10 abgearbeitet sind, befinden sich die restlichen 190 schon im Cachespeicher. Nun ist der Cache Speicher bei Motorola Prozessoren im Chip selbst integriert, und belegt keinen externen Platz. (Einige PC Fritzen werben ja mit 64 KB Caches, die niemals auf einer CPU Platz hätten). Da nun die Übertragung von Daten aus dem Cache in den Motorola Chip weniger Zeit kostet als das Holen der Daten aus dem konventionellen Speicher dürfe klar wie Kloßbrühe sein. Doch dieses Vorhandensein der Programmzeilen bedeutet für selbst modifizierende Programme oft das Ende, denn es wird nicht daran gedacht, das der Programm zwar im RAM Speicher geändert wurde, aber diese Modifikation noch nicht im Cache steht. Und dies war's dann wohl.

Wer jedoch nicht auf selbstmodifizierende Programm verzichten möchte, muss mit einem Prozessorbefehl den Cache-Inhalt löschen.

Verschlüsseln durch Selbstmodifizierung

Da es wenig sinnvolle Programme gibt, möchte ich dennoch einen Aufgabebereich nicht übergehen. Die Verschlüsslung und Dekodierung. Wenn ein Programm sich selbes aufbaut ist es schwierig es zu verstehen.

Wir wollen eine Routine „verschlüsseln", die eine Addition von 11 Zahlen vornimmt.

```
        move    #10,T1+2
        move    #ADD_TOKEN,AddTok

        ; Cache löschen und neu holen
        ; das Programm ist jetzt fertig

T1      move    #0,d0
AddTok  or  d0,d0   ; täuschen ein OR vor
```

```
        dbra    d0,T1
```

Wer das Programm verstehen will, muss schon eine „entschlüsselte“ Version im Speicher haben. Ist es dort gut versteckt findet man es auch nicht (so leicht).

Mit diesen Worten legen den Gedanken an selbst modifizierende Programme (verändernde Viren wollen wir doch nicht etwa Programmieren, oder?! Das wäre Pfui!) erst einmal beiseite.

BS-Programming

Interrupts

Interrupts sind bei dem C-64 schon immer ein Schlagwort in der Demo- und Spiele-Programmierung gewesen. Doch warum sollten besonders sie so wichtig sein?

Interrupt ist das englische Wort für Unterbrechung, und, da haben wir schon das Wichtigste, es wird unterbrochen und zwar das laufende Programm. Ein Interrupt stört also ein ablaufendes Programm, und in der sogenannten Interrupt-Routine wird ein kleines anderes Programm ausgeführt. Die Signale, die Interrupts auslösen können, sind verschieden, und sind auch alle für verschiedene Zwecke einsetzbar. Nehmen wir einmal als Beispiel den Interrupt, der ausgelöst wird, wenn oben eine Rasterzeile anfängt den Bildschirm zu beschreiben. Das Interruptprogramm kann jetzt auf eine bestimmte Zeile warten, und dann z. B. die Farbe oder Auflösung ändern. Das Tolle dabei ist: Das Hauptprogramm weiß gar nichts von dem Interrupt, es arbeitet einfach weiter. Hier haben wir den ersten Vorteil: Abarbeitung von Programmen, die natürlich nicht zu lang sein dürfen, ohne dass das Hauptprogramm gestört wird. Noch ein anderer Vorteil ist mit der Ausführung verbunden: Die Gleichmäßigkeit der auslösenden Signale. Besonders bei Musikprogrammen ist es wichtig, dass die Unterbrechung gleichmäßig kommt, so dass bei der Musikabspielung kein Leiern entsteht.

Da der Benutzer viele Interrupts selber in das Programm integrieren kann, gibt es einen Interrupt-Server. Dieser übernimmt dann die Abarbeitung, und überprüft, dass kein Interrupt zu kurz kommt.

Um ein Interrupt in das System einzubinden, muss eine Interrupt-Struktur angelegt werden. Sie ist sehr kurz (gerade einmal 22 ($16) Bytes).

Table 4. Interrupt

$000	0	Node
$00e	14	Data
$012	18	Code

Diese Struktur muss nun einer `Exec`-Funktion übergeben werden, die diesen Interrupt dem System-Server hinzufügt. Die Routine heißt

```
AddIntServer(IntNr, Interrupt)
```

Wird vom System ein bestimmter Interrupt erzeugt (z. B. Rasterzeile oben), so wird eine zugehörige Routine abgearbeitet. Welcher Interrupt dies sein soll, wir können ja nicht bei jedem Interrupt unser Programm abarbeiten lassen, wird in `IntNr` (Register `D0`) übergeben. Löst das System nun einen Interrupt der Nummer `IntNr` aus, so wird unser Programm abgearbeitet. Natürlich gibt es auch hier Prioritäten, diese werden in der Interrupt-Struktur eingetragen.

Nach dem Aufruf ist der Interrupt aktiv, und unsere Routine wird immer beim Auslösen aufgerufen. Um den Interrupt wieder zu entfernen, muss er aus der Interrupt-Server-Liste wieder gelöscht werden. Dies übernimmt die `Exec`-Funktion `RemIntServer()`. Die Parameter sind analog denen von `AddIntServer()`.

Interessant ist, das bei `AddIntServer()` kein Handle mitgegeben wird, sondern für das Abmelden auch ein Zeiger auf die Original-Struktur vorhanden sein muss. Der Interrupt wird also nicht vom System kopiert! Das kann schwerwiegende Konsequenzen haben, wie wir im nächsten Kapitel erfahren können.

Das folgende Listing führt einen Farbwechsel durch. Das Programm wird im Rasterstahl-Interrupt, `IntNr`=5, ausgelöst.

```
* Interrupt Version 1 28.3.92 116 Bytes

************** VARIABLEN *********************
AddIntServer = -168
RemIntServer = -174

LN_PRI       =    9   ; Node Priorität
IS_CODE      =   18   ; Anfang des Interrupts

************** HAUPTPROGRAMM *****************

        bsr.s   IntStruktAufbauen
        bsr.s   InitInterrupt

MausPress
        btst    #6,$bfe001
        bne.s   MausPress
        bra.s   EndInterrupt

************** Interruptstruktur aufbauen ****
```

```
IntStruktAufbauen
        lea     IntStruct,a2
        move.l  #Interrupt,IS_CODE(a2)
        move.b  #-1,LN_PRI(a2)
        rts

************** Interrupt ins System einbinden *

InitInterrupt
        move.l $4.w,a6
        moveq   #5,d0 ; Interrupt Nr 5 =Raster-Interrupt
        lea     IntStruct,a1 ; Struktur Pointer in a1
        jmp     AddIntServer(a6)

************** Interrupt aus System entfernen *

EndInterrupt
        moveq   #5,d0 ; IntNumber 5
        lea     IntStruct,a1
        jmp     RemIntServer(a6); nun muss er wieder weg

************** Interrupt *********************

Interrupt
        movem.l D0-D7/A0-A6,-(SP)
        move    SR,-(SP)
        bsr.s   Farben ; Hauptteil des Interrupts
        move    (SP)+,SR
        movem.l (SP)+,D0-D7/A0-A6
        rts

Farben  move    $dff006,$dff180 ; Rasterzeile gibt Farbwert
        rts

************** Interrupt Struktur ************

IntStruct
        ds.b    22              ; Struktur ist 22 Bytes lang
```

Das Hauptprogramm besteht aus dem Aufrufen von `IntStruktAufbauen` sowie `InitInterrupt`, einer Warteschleife und aus dem Aufruf von `EndInterrupt`.

In dem ersten Unterprogramm wird eine Interrupt-Struktur aufgebaut. Freien Speicherplatz haben wir daher am Programmende mit der Zeile: “IntStruct ds.b 22“ belegt. In diesen Speicherbereich bauen wir jetzt eine Struktur auf. Nur wenige Einträge sind nötig. Den Zeiger auf den Interrupt-Code müssen wir eintragen, und vielleicht noch die Priorität, aber auch das könnte man sich eigentlich sparen. Jetzt steht die Struktur, und der Interrupt kann ins System eingebunden werden. Die `IntNr` ist 5, was für einen Rasterzeilen-Interrupt steht. Bei jedem Erreichen der ersten Bildschirmzeile durch den Rasterstahl wird ein Interrupt ausgelöst. Das Aufrufen von `AddIntServer()` bindet die Struktur in die Liste ein. Jetzt wird bei jedem Durchlauf unser Programm aufgerufen. Es ist ein einfaches Programm. Wir sichern Register und Statusflags, und können dann das Unterprogramm Farben aufrufen. Aus der Speicherzelle $dff006, die die aktuelle Rasterzeile angibt, übertragen wir den Wert in das Hintergrundfarbregister. Damit haben wir immer unterschiedliche Farben auf dem Schirm.

Dieses Farbgeflimmere läuft jetzt im Hintergrund, und das Programm wartet nun auf einen Druck auf die Maustaste. Ist dieser getätigt, wird das Unterprogramm `EndInterrupt` aufgerufen, damit wird der Interrupt aus dem System entfernt.

Tasks

Das Amiga OS ist ein Multitasking Betriebssystem. Multitasking heißt, mehrere Programme können quasi gleichzeitig abgearbeitet werden. Effektiv kann natürlich nur ein Programm real laufen, wir haben ja auch nur einen Prozessor. `Exec` schaltet in einer Interrupt-Routine die Programme um. Dazu müssen sie sich in einer speziellen Form befinden. Die Programme, die von `Exec` verwaltet werden, heißen Tasks (engl: Aufgabe). Tasks haben einen genormtes Bild, das es einfach macht, sie zu switchen (umzuschalten). Dieses Umschalten wird auch Task-Switching genannt.

Taskdemo 1

Einen Task zu erstellen ist ziemlich einfach. Wir werden daher ganz einfach anfangen, und durch Anforderungen an den Task immer mehr hinzulernen.

Beginnen wir mit dem ersten Task-Demo. Ein Task wird dabei von uns wie in Interrupt in einem Speicherbereich aufgebaut. Der Speicher für die Task-Struktur, die eine Länge von 92 Bytes einnimmt, wird wieder mit `ds.b` freigehalten. In diesem Speicherbereich schreiben wir nun ein paar Werte hinein. Da wir einfach beginnen wollen, und erst mal unseren Task bewundern wollen, ist ein einziger Eintrag ausreichend. Anzugeben ist auf alle Fälle (wer noch nie einen Absturz gesehen hat, kann den Eintrag natürlich auch auslassen!) der Zwischenspeicher für den Stack. In der Task-Struktur ab dem Offset 54 ist dieser also unbedingt anzugeben.

Nachdem die Struktur so weit vorbereitet ist, dass ein “minimal-Task“ laufen kann, muss dieser in das System eingebunden werden. Wie mit `AddIntServer()` ein Interrupt eingebunden wird, so wird mit `AddTask()` der Task aktiviert.

Der Task wird mit der Funktion `RemTask()` wieder aus dem Speicher entfernt. Wir benutzen diese Funktion allerdings nicht, da sich unsere Tasks immer selbstständig verflüchtigen, wenn sie ausgelaufen sind.

Das folgende Demo zeigt die Anwendung der OS-Funktionen.

```
* Taskdemo1 Version 2 19.3.92 148 Bytes

************** VARIABLEN *********************
```

```
AddTask           = -282 ; Exec

tc_SPReg          = 54 ; Task-Struct
sizeof_taskstruct = 92

************** HAUPTPROGRAMM ****************

        move.l  #SPZwischen,TaskStruct+tc_SPReg
        ; Zeiger auf Stack m u s s angegeben werden

        lea     TaskStruct,a1 ; Zeiger auf Task Struktur
        lea     TaskAnfang,a2 ; Startadresse des Tasks
        sub.l   a3,a3 ; keine eigene Rücksprungadr.
        move.l  4.w,a6
        jmp     AddTask(a6)

************** Task ************************

TaskAnfang
        moveq   #10,d0
        moveq   #-1,d1
Schleife move    d1,$dff180
        dbra    d1,Schleife
        dbra    d0,Schleife
        rts

************** Platz für die Task-Struct ****

TaskStruct
        ds.b    sizeof_taskstruct ; Hier kommt die Task-Struktur rein

SPZwischen
        dc.l     0     ; Zwischenspeicher für SP
```

Dies ist also das Task-Programm in Minimalkonfiguration. In einem wohl definierten Speicherplatz wird `sizeof_taskstruct = 92` Bytes Platz für die Struktur gelassen. Ein Struktureintrag wird mit einem Zeiger geladen, denn wir so quasi provisorisch setzen.

Bei dem Aufruf von `AddTask()` wird die Struktur als Pointer in `A1` übergeben. Doch die Funktion verlangt noch mehr Übergabeparameter, um im System einen neuen Task zum Laufen zu bringen. Das OS benötigt vielmehr den Zeiger auf die Startadresse des Task-Programms. Dieser Zeiger wird in `A2` verlangt, er wird auch `InitPC` genannt. Ein weiter Zeiger kann auf Wunsch übergeben werden, er ist aber im Normalfall null, `FinalPC`. Der Task ist ja oft, so wie unser Programm, einmal mit seiner Arbeit fertig. Dann muss er durch das letzte RTS aber irgendwo hin springen. Wenn wir `A3` auf ein Programm setzen, so wird dieses automatisch nach dem Ende angesprungen. Dies ist sinnvoll bei Tasks, die Speicher belegten, und ein anderes Programm diesen wieder freigeben. Wenn `A3=0` ist, so kümmert sich `Exec` um das Ende, und die System-Routine für `FinalPC` wird angesprungen.

Bei Ablauf des Programms wird zunächst einmal der Task eingerichtet, und dann ist das Hauptprogramm fertig. Das Programm springt durch den JMP zur nächst höheren Ebene zurück. Nach dem Init-Aufruf beginnt unser Task den Bildschirmhintergrund zu verändern.

Taskdemo, das nicht klappt

Unser Task wurde mit einer Minimalkonfiguration „ins Leben gerufen“. Es wurde lediglich ein Eintrag gesetzt. Doch.., heißt das etwa, wir können noch mehrere Werte in die Struktur schreiben? Ja freilich (sonst wäre sich nicht 92 Bytes groß), das müssen wir sogar. Warum? Dann einmal anschnallen uns testen. Setzen wir einmal den unteren Block anstatt des alten ein, und compilieren. Kein Fehler. Und ausführen. Ohhhh, der Computer stürzt gnadenlos ab. Der Übende kann dies ja einmal ausprobieren, nur, alles vorher sichern!

```
************** Task ************************

TaskStruct ds.b    92          ; Hier kommt die Task-Struktur rein

TaskAnfang
        moveq   #10,d0
        moveq   #-1,d1
Schleife
        bsr     Farben1
        dbra    d1,Schleife
        dbra    d0,Schleife
Farben1 bsr     Farben2
        rts
Farben2 move    d1,$dff180
        rts
```

Und, ausprobiert? Nein. OK! Ja? Dann neu laden, und mit den nicht-ausprobierenden F*i*l*n*e* nach der Ursache des Absturzes suchen.

Tja, wenn wir beide Task-Programme so vergleichen, eigentlich doch nur der Unterschied, das der zweite Programmblock zwei Unterprogramme ausruft. Was kann daran denn so schlimm sein?

Um das zu verstehen, ist es notwendig auf den JSR- oder BSR-Befehl zurückzukommen. Beide Befehle springen ein Unterprogramm auf, und, um wieder zurück zum alten aufrufenden Programmteil zu finden, speichern sie die Rücksprungadresse auf den Stack. Ja, auf den Stack. Kling, da müsste wieder ein Groschen gefallen sein, der Stack. Das Programm schreibt Werte auf

den Haupt-Stack, aber welcher soll das denn sein? Der Task benötigt einen eigenen Stack. Wir haben das Problem erkannt, nach der Werbung geht's weiter.

Taskdemo 1 verbessert

In der Task-Struktur gibt es zwei Einträge, die Zeiger auf einen Stack verwalten. `tc_SPLower` zweigt auf den unteren Bereich, und `tc_SPUpper` auf den oberen. Das der Stack von unten nach oben wächst sollte man mittlerweile wissen.

Mit diesem Hintergrundwissen, starten wie Folge zwei unserer Task-Serie: „Nur abstürzen ist leichter".

```
* Taskdemo2 Version 2 19.3.92 586 Bytes

************** VARIABLEN ********************

AddTask    = -282 ; Exec

tc_SPReg   = 54   ; Task-Struct
tc_SPLower = 58
tc_SPUpper = 62

sizeof_taskstruct = 92

************** HAUPTPROGRAMM ****************

        lea    TaskStruct,a1 ; Zeiger auf Task Struktur

        move.l #TaskStackEnd,tc_SPReg(a1); Zeiger auf Stack

        move.l #TaskStackAnf,tc_SPLower(a1) ; Untere Grenze des Stackspeichers

        move.l #TaskStackEnd,tc_SPUpper(a1) ; Obere Grenze des Stackspeichers+2

        lea    TaskAnfang,a2 ; Startadresse des Tasks
        sub.l  a3,a3 ; keine eigene Rücksprungadr.
        move.l 4.w,a6
        jmp    AddTask(a6)

************** Task ************************

TaskAnfang
        moveq  #1,d0
        moveq  #-1,d1
Schleife bsr    Farben1
        dbra   d1,Schleife
        dbra   d0,Schleife
Farben1 bsr    Farben2
        rts
Farben2 bsr    Farben3
        rts
Farben3 bsr    Farben4
        rts
Farben4 bsr    Farben5
        rts
Farben5 move   d1,$dff180
        rts

************** Task ************************

TaskStruct
        ds.b   sizeof_taskstruct  ; Hier kommt die Task-Struktur rein
TaskStackAnf
        ds.l   100       ; 400 Bytes Stackgröße
TaskStackEnd
```

Im Hauptprogramm richten wir den Stack in der Task-Struktur ein. Der Stack befindet sich von `TaskStackAnf` bis `TaskStackEnd`. Da er von unten wächst, werden die ersten Adressen natürlich näher bei `TaskStackEnd` stehen.

Empfehlung vom BundeSTACKminister: Den Stack mindestens 70 Byte groß machen.

Unabhängiger Task

An was haben wir nicht alles gedacht, Stack ist eingerichtet, wunderbar. Doch einen großen Fehler darf der Programmierer nie machen: er darf beim Programmablauf nicht compilieren. Denn wenn er das täte, wäre die Wahrscheinlichkeit, dass unsere Task, und seine Struktur, die verlassen von allen guten Programmen, im Speicher steht, durch den Compiliervorgang überschrieben wird, groß. Das ist der Tod. Demnächst lesen wird in WILD: Tod durch Absturz, Task kam gewaltsam ums Leben. Um das Problem zu lösen, muss der Task und die Task-Struktur in einem Speicherbereich stehen, der sicher vor dem Überschreiben ist. Dazu wird Speicher allokiert, und somit erreichen wir die Unabhängigkeit. Jetzt kann gnadenlos compiliert werden, der Task ist durch die Struktur sicher.

Um zum Ziel zu gelangen müssen folgende Schnitte unternommen werden:

- Speicher für den Stack, Task und dessen Struktur holen
- Struct einrichten
- den Task in den passenden Speicherbereich kopieren
- Task starten

Zum Schluss unserer Task-Folge die fehlerfreie Version.

```
* Taskdemo3 Version 2 8.2.91 236 Bytes

************** VARIABLEN ********************

AllocMem = -198 ; Exec
AddTask  = -282
FindTask = -294

STACKSIZE = 100
sizeof_taskstruct = 92

tc_Name     = 10 ; Task-Offsets
tc_SPReg    = 54
tc_SPLower  = 58
tc_SPUpper  = 62

NO_ERR_TO_DOS = 0 ; DOS Returnwert
INSUFFICENT_FREE_STORE = 103

************** HAUPTPROGRAMM *****************

        bsr     HoleMemFürTask
        move.l  d0,a1 ; Zeiger auf Speicher für Task
        beq.s   KeinSpeicher

        bsr     InitTaskStruct

        bsr     CopyTaskToMem

        bsr     StartTask

        bsr     InfoOfTask

        moveq   #NO_ERR_TO_DOS,d0 ; alles OK
        rts

KeinSpeicher
        moveq   #INSUFFICENT_FREE_STORE,d0 ; kein Mem mehr?
        rts

************** Speicher für den Task besorgen

HoleMemFürTask
        move.l  4.w,a6
        move.l  #TaskEnde-TaskAnfang+sizeof_taskstruct+STACKSIZE,d0

        move.l  #$10001,d1
        jmp     AllocMem(a6)

************** Task-Struktur einrichten ******

InitTaskStruct
        move.l  #TaskName,tc_Name(a1) ; Name des Tasks in Node

        move.l  a1,d0 ; Anfang des Speichers
        add.l   #sizeof_taskstruct,d0 ; Tasksctruct Größe weg
        move.l  d0,tc_SPLower(a1) ; Untere Grenze des Stacks

        add.l   #STACKSIZE,d0
        move.l  d0,tc_SPReg(a1) ; Zeiger auf Stack

        move.l  d0,tc_SPUpper(a1) ; Obere Grenze des Stacks+2

        rts

************** Task in freien Speicher kopieren

CopyTaskToMem
        lea     TaskAnfang,a0 ; Zeiger auf Prg, Quelle
        move.l  a1,a2 ; Ziel in a2
        lea     (sizeof_taskstruct+STACKSIZE)(a2),a2
          ; Tasktstruct und Stack weg,
          ; dann Zeiger auf Programmstart
        move.l  a2,a3 ; diesen auch in A3

        move    #TaskEnde-TaskAnfang-1,d0
CopyTask move.b  (a0)+,(a2)+ ; Struct und Prg kopieren
        dbra    d0,CopyTask
        rts

************** Task ausführen ***************
```

```
        ; a1 ist Zeiger auf Task Struktur im Speicher
StartTask ; in a1 = Task-Struct
        move.l  a3,a2 ; a2 = Prgcode
        sub.l   a3,a3 ; a3 = 0, kein Ende-Stack
        jmp     AddTask(a6)
        rts

************** Was wir schon wissen nocheinmal

InfoOfTask
        lea     TaskName,a1
        jsr     FindTask(a6)
        move.l  d0,TaskStruct
        rts

************** Name des Tasks ***************

TaskStruct
        dc.l    0                   ; hier finden wir ihn
TaskName
        dc.b    "Farbenspiel",0
        cnop    0,2

     ************************************
     * Task, der in den Mem kopiert wird *
     ************************************

TaskAnfang
        lea     Tstmem,a4

        bsr     Farben

        move.l  4.w,a6
        sub.l   a1,a1 ; wir selbst
        jmp     FindTask(a6)

        ; in d0 das gleiche wie TaskStruct

        rts

************** erstes große Unterprogramm im Task *******

Farben  moveq   #1,d0 ; jetzt das Prg
        moveq   #-1,d1
Schleife bsr     Farben1
        dbra    d1,Schleife
        dbra    d0,Schleife
Farben1 bsr     Farben2
        rts
Farben2 bsr     Farben3
        rts
Farben3 bsr     Farben4
        rts
Farben4 bsr     Farben5
        rts
Farben5 move    d1,0(a4)
        move    0(a4),$dff180
        rts

************** Speicher für Task *************

Tstmem  ds      1           ; 1 Word reservieren

TaskEnde
```

Die in der oben genannten Aufzählung aufgerufenen Unterprogramme sind `HoleMemFürTask`, `InitTaskStruct`, `CopyTaskToMem`, `StartTask` und `InfoOfTask`. Nach diesen Aufrufen ist der Task im System eingebunden, und läuft, oder es wurde kein Speicher gefunden, und der Vorgang musste abgebrochen werden.

Mit `HoleMemFürTask` wird für den Task, die Taskstruct und den Stack Speicher geholt. Die Größe dieses Blockes beträgt `TaskEnde-TaskAnfang+92+STACKSIZE` Bytes. `TaskEnde` und `TaskAnfang` sind zwei Label, 92 ist die Größe der Struktur, und `STACKSIZE` ist die Größe des Stacks, mit 100 Bytes für unseren Task ziemlich groß bemessen.

`InitTaskStruct` richtet die Task-Struktur im Speicher ein. `A1` ist der Zeiger auf dem freien Speicherplatz. Dort wird, wie die kleine Tabelle zeigt, der Speicher verteilt:

Taskstruktur	92 Byte
Taskstack	100 Byte
Taskcode	der Rest

Zuerst wird dir Struct generiert. Auffallen werden lediglich die Zeilen

```
        move.l   a1,d0                  ; Anfang des Speichers
        add.l    #sizeof_taskstruct,d0  ; Tasksctruct Größe weg
        move.l   d0,tc_SPLower(a1)      ; Untere Grenze des Stacks
```

und folgende.

Da der Stack hinter der Struktur ist, muss zum Anfang des Speichers (immer in `A1`) die Länge der Struct addiert werden. Das ist dann die untere Stackgrenze. Ebenso wie durch eine Addition mit `STACKSIZE` der Eintrag `tc_SPReg` mit der Adresse geladen.

Jetzt ist der Task dran, er wird mit `CopyTaskToMem` in den Speicher kopiert. Den Anfang im allokierten Speicherbedarf errechnen wir durch Startadresse plus `sizeof_taskstruct+STACKSIZE`.

`StartTask` startet den Task, der mit `AddTask()` eingebunden wird.

Etwas abseits steht noch ein kleines Unterprogramm namens `InfoOfTask`. Benutzt wird hier eine neue Funktion aus der `Exec`-Lib, `FindTask()`. Mit dem Namen in `A1` sucht sie in der Systemliste nach dem Task nach dem Merkmal Name. Blöd ist es natürlich, wenn mehrere Tasks die selben Namen haben, da hilft nur Eigenbau der `FindTask()` Routine. Näheres verrät aber hier das ROM-Listing im Anhang.

Die komplette Task-Struktur

Die wichtigsten Einträge wurden durch die Kapitel erklärt ein eigenständig ablaufender Task ist kein Problem. Dennoch sollte ein kleiner Blick in die Task-Struktur gewährt werden. Die Länge ist bekannterweise 92 ($5c) Bytes.

```
$00   0  Node         ; Eine Node, die die Tasks verkettet
$0e  14  Flags        ; bestimmte Task-Flags, über den Zustand
$0f  15  State        ; Task hinzugefügt, aktiv, entfernt, wartet?
$10  16  IDNestCnt    ; Zähler für Disable()
$11  17  TDNestCnt    ; Zähler für Forbid()
$12  18  SigAlloc     ; besetzte Signal-Bits
$16  22  SigWait      ; Signale, auf die gewartet wird
$1a  26  SigRecvd     ; empfangene Signale
$1e  30  SigExcept    ; Signale, die eine Exception auslösen
$22  34  TrapAlloc    ; besetzte Trap-Befehle
$24  36  TrapAble     ; erlaubte Trap-Befehle
$26  38  ExceptData   ; Daten der Exceptions
$2a  42  ExceptCode   ; Code der Exceptions
$2e  46  TrapData     ; Daten des Traps
$32  50  TrapCode     ; Code des Traps
$36  54  SPReg        ; SP Speicher
$3a  58  SPLower      ; untere Stackgrenze
$3e  62  SPUpper      ; obere Stackgrenze (+2)
$42  66  Switch       ; Task gibt CPU weiter an nächsten Task
$46  70  Launch       ; Task gekommt CPU vom vorigen Task
$4a  74  MemEntry     ; vom Task benutzter Speicher
$58  88  UserData     ; Wer was anhängen will, kann hier einhaken
```

Die Bibliothek - Die Library

Eine Library, oder in der Übersetzung Bibliothek, ist eine Sammlung von Funktionen. In der Bücherei kann man zu fast jedem Gebiet ein Buch finden, so sollte es mit den Libraries auch sein, sie sollten zu jedem Problemgebiet eine Betriebssystemfunktion anbieten können. Was ist, wenn in der Bücherei einmal kein Buch zu einem Thema vorhanden ist? Übertragen wir dies auf die Programmierung: Zu einem ganz speziellen Gebiet existiert keine Funktion. Dann sollte mit anderen Funktionen das gesuchte Problem gelöst werden, nicht aber etwas erfunden werden, was noch nicht aufgeschrieben wurde. Oder, wir leihen uns ein Buch aus, was übertragen auf den Computer einer externen Implementierung gleichkommen würde.

Mit der `Exec`-Library haben wir eine leistungsstarke Library kennengelernt, die sich um die Verwaltung des Speicher, die Interrupts, die Tasks und noch einiges mehr kümmert. Die `Exec`-Library ist aber dennoch ein Sonderfall, denn sie ist durch die Absolute Adresse $4 sofort ansprechbar. Dies ist bei anderen Libraries nicht so. Aber..., was gibt es denn da noch so für Libraries? Die folgende Tabelle soll eine kleine Übersicht über Librarynamen und Funktion bieten.

Library	Aufgabengebiet
`intuition.library`	Verwaltung der intuitiven Objekte z. B. Fenster, Screen
`graphics.library`	Grafische Grundoperationen, Linie, Kreis zeichnen
`dos.library`	Alle Funktionen zum Zugriff auf die Speichermedien
`layers.library`	Nimmt sich der Fensterschichten an
`diskfont.library`	Läd und verwaltet die externen Zeichensätze
`icon.library`	Benötigt von der Workbench für die kleinen Pictogramme
`mathffp.library`	Mathematische Grundfunktionen
`mathtrans.library`	Transzendente Funktionen
`mathieeedoubbas.library`	doppelte genaue Fleißkommazahlen
`mathieeedoubtrans.library`	die dazugehörigen doppelt genauen trans. Funk.

translator.library	Eine Funktion zum Umwandeln der Worte
expansions.library	Verwendet bei der Einbindung von externen Geräten

Hinzu kommt noch eine ganze Menge mehr. Es hat sich z. B. unter OS 2.0 die Anzahl verdoppelt. Zudem kommt eine gewaltige Menge externer Libraries auf dem PD-Sektor hinzu.

Die in der Tabelle genannten Libraries sind alle intern, d. h. sie müssen nicht erst von Diskette geladen werden. Interne Libraries sind in das ROM verlagert worden, da die Funktionen oft benötigt werden. Externe Libraries enthalten Funktionen, sie seltener benötigt werden, und wofür ROM-Plaz zu Schade war.

Libraries öffnen

Um eine Library zu nutzten, muss sie geöffnet werden. Die `Exec`-Library ist ein Sonderfall, sie ist immer geöffnet, und der Zeiger auf die Basisadresse ist in 4 gesichert. Bei allen andere Libraries, sei es extern, oder intern, muss die Funktions-Bibliothek mit der Funktion `OpenLibrary()` geöffnet werden. Nach dem Benutzen muss sie wieder mit `CloseLibrary()` geschlossen werden. Das Schließen ist besonderes wichtig, uns sollte nicht vergessen werden. Dies liegt am Vorgang des Ladens, der etwa in groben Schritten wie folgt verläuft:

- Suche Library mit einem Namen in der internen Library-Liste
- Nicht in interne Liste? Dann im Verzeichnis LIBS: nachschauen
- nicht gefunden, dann Fehlercode NULL
- war extern, dann in den Speicher laden
- einen Zähler hochseten, der die Anzahl der Öffnungen zählt

Bei dem Vorgang des Öffnens wird ein Zähler erhöht, der Auskunft über die Anzahl der Lib-Benutzer gibt. Wird die Library geschlossen, so wird der Zähler vermindert. Eine externe Library kann aus dem Speicher gelöscht werden, wenn kein Benutzer mehr auf die Funktionen zurückgreifen will. Damit nicht unnötig Platz verschwendet wird, muss die Library geschlossen werden.

Die Funktion zum Benutzen der Libraries lautet:

```
LibPtr = OpenLibrary(LibName,Version)(A1,D0)
```

LibName ist der Name der Library. Sind Funktionen aus der `intuition`.library zu nutzen, so ist ein Zeiger auf dem String `"intuitions.library"` in `A1` zu übergeben.

Version ist die Versionsnummer der Library. Das Übergeben ist mit verschiedenen Vorteile verbunden. Sollten mehrer Libraries im Speicher vorhanden sein, so wird anhand der Versionsnummer unterschieden. Gibt es eine Library mit einer hohen Versionsnummer nicht, so ist die Funktion ebenso beendet, als ob der Name der Library nicht existiert.

Die Gegenüberstellung zeigt noch einmal kurz, welche Versionsnummer welcher Kickstartversion zugeordnet sind.

Nr	Kickstartversion
0	jede Version
30	1.0
31	1.1 (NTSC-Version)
32	1.1 (PAL-Version)
33	1.2
34	1.3
35	1.3 (neu dabei ist ein Treiber für A2024 Monitore)
36	2.0
37	2.1

Ist die Versionsnummer egal, so kann `OldOpenLibrary()` verwendet werden. Die Versionsnummer kann entfallen, und 2 Bytes werden gespart.

Nach dem Öffnen wird die Library mit `CloseLibrary()` wieder geschlossen.

```
* OpenLibs Version 1 13.4.1992 56 Bytes

************** VARIABLEN *********************

OldOpenLibrary  = -408
CloseLibrary    = -414

************** HAUPTPROGRAMM *****************

        bsr.s    OpenDos
        bsr.s    OpenAsl

        moveq    #0,d0          ; in Startup-sequence
        rts                  ; kein Fehler Melden! ,sonst AslBase in d0

************** Dos-Library öffnen ************
```

```
OpenDos move.l  4.w,a6
        lea     DosName,a1
        jmp     OldOpenLibrary(a6)

************** Asl-Library öffnen ************

OpenAsl lea     AslName,a1
        jmp     OldOpenLibrary(a6)

************** Library Namen *****************

DosName  dc.b    "dos.library",0
AslName  dc.b    "asl.library",0
```

Libraryeinträge ändern (patchen)

Als der Amiga 1000 auf den Computermark einzog, konnte man alle fünf Minuten mit einem Absturz rechnen. Nach vor einem Jahr war es für viele potentielle Käufer ein Kriterium gewesen, sich keinen Amiga zuzulegen. Heute sieht die Sache natürlich ganz anderes aus. Commodore und eine Unmenge Entwickler und Programmierer sorgen ständig dafür, das die Fehler ausgemerzt werden.

Sollte eine Funktion einmal Fehlerhaft arbeiten, oder der Benutzer möchte sie nach seinem Willen verändern, so kann der dazu die `Exec`-Funktion `SetPatch()` verwenden. Im *C/*-Verzeichnis findet man ebenfalls eine Datei, das die ROM-Fehler patcht. Auch eine Menge Erweiterungskarten Hersteller bringen Patches mit den Karten auf den Markt, da oft OS-Routinen Schwierigkeiten haben.

Mit der Zeit aber wurde das Betriebssystem runder, und erlaubt nun absturzfreies Arbeiten — wenn es nicht gerade vom Benutzer provoziert wird.

Das folgende Demo ersetzt die Funktion `DisplayBeep()` aus der `intuition.library` durch eine eigene.

```
* Patch1 Version 1 15.2.92 130 Bytes

************** VARIABLEN **********************

OldOpenLibrary = -408
CloseLibrary = -414

AllocMem = -198
SetFunktion = -420
DisplayBeep = -96

MEMB_PUBLIC = 0 ; Speichersorte egal

************** HAUPTPROGRAMM *****************

************** Intuition-Lib öffnen **********

OpenIntui move.l 4.w,a6
        lea IntuiName,a1
        jsr OldOpenLibrary(a6)
        move.l d0,IntuiBase

************** Display Beep Patchen **********

LetsPatch moveq #DisplayBeepEnd-DisplayBeepAnf,d0
        move.l #MEMB_PUBLIC,d1
        jsr AllocMem(a6)
        move.l d0,NewDisBeepMem

        lea DisplayBeepAnf,a0
        move.l d0,a1 ; In den neuen Speicher
        moveq #(DisplayBeepEnd-DisplayBeepAnf)/2-1,d0
CopyNewDisBeep
        move (a0)+,(a1)+ ; kopieren
        dbra d0,CopyNewDisBeep

        move.l NewDisBeepMem,d0 ; Wo ist sie zu finden?
        move.l #DisplayBeep,a0  ; Offset
        move.l IntuiBase,a1   ; Basis
        jsr SetFunktion(a6)

************** Intuitions-Lib schließen ******

CloseIntui
        move.l IntuiBase,a1
        jmp CloseLibrary(a6)

************** Hier ist die neue Routine *****

DisplayBeepAnf
        move.l d0,-(SP)
        moveq #-1,d0
Flash   move d0,$dff180
        dbra d0,Flash
```

```
        move.l (SP)+,d0
        rts

DisplayBeepEnd

************** Daten ************************

IntuiName dc.b "intuition.library",0
IntuiBase dc.l 0

NewDisBeepMem dc.l 0
```

Wie bei den Tasks muss das Programm in einen freien Speicherbereich kopiert werden.

Um das Programm zu testen soll der Cursor so weit bewegt werden, bis es im Editor von Devpac oben oder unten nicht mehr weitergeht. Der Devpac-Editor ruft die Routine `DisplayBeep()` auf, und, da sie von uns gepatcht ist, wird unser kleines Farbprogramm aufgerufen.

Devices

Ein Devise ist eine Ebene zwischen Hardware und Library, die Daten austauscht. Devices haben eine große Ähnlichkeit mit Libraries, auch sie enthalten viele grundlegende Funktionen, die der Benutzer nutzen sollte. Bei genauerer Betrachtung des ROM-Listings zeigt sich, das sie die gleichen Funktionen zur Tabellengenerierung nutzen.

Einen wesentlichen Unterschied zur Library gibt es allerdings doch. Nach dem Öffnen der Library steht eine initialisierte „Ebene“ zur Verfügung, dies ist bei Devices nicht so. Hier geschieht der Datenaustausch über Strukturen. Die wichtigste Struktur ist hierbei `IORequest`. `Exec` verfügt daher über viele Funktionen, die die Kommunikation mit Devices und deren Strukturen vereinfachen und sogar erst ermöglichen.

Die nachfolgende Tabelle enthält die wichtigsten Amiga-Devices:

Device	Aufgabe
`audio.device`	Soundausgabe
`clipboard.device`	Übertragung der Clipboard-Inhalte
`console.devicre`	Ein- und Ausgabe über die Konsole
`gameport.device`	Joystick, Paddle und Maus
`input.device`	Alle Eingaben und Ausgaben
`keyboard.device`	Tastaturverwaltung
`keymap.device`	Tastaturbelegung
`narroator.device`	Sprachausgabe
`paralell.device`	Paraller Port
`serial.device`	Serieller Port
`timer.device`	Systemzeit
`trackdisk.device`	Diskettenzugriff

Das Trackdisk.device

Das `Trackdisk.device` ist zuständig für alle Operationen rund um das Laufwerk. Die `dos.library` greift auf viele der Funktionen zurück.

Das `trackdisk.device` ist durch die Hardware bedingt flexibel zu programmieren. Neue Diskettenformate sind leicht zu schaffen, bestehende leicht zu emulieren. Der Amiga ist einer der wenigen Rechner, der MS-DOS (damit auch Atari) und Macintosh Disketten lesen kann.

Disk-Klacken entfernen

Ein Device wird wie ein Task gehandelt, und kann auch so entfernt werden. Ein Aufruf von `RemTask()` würde ihn also aus den Speicher verbannen.

Dieses machen wir uns nun bei einer kleinen, nicht ganz ungefährlichen Aktion zu nutze. Das `trackdisk.device` überprüft in regelmäßigen Abständen, ob eine Diskette eingelegt ist. Dabei wird ein mit störendes Klicken gewahr. Dieses Klicken kann ganz einfach abgestellt werden, indem das `trackdisk.device` ausgeschlossen, d. h. abgemeldet wird.

```
* TrackDiskWeg Version 1 10.2.92 41 Bytes

************** VARIABLEN ********************

FindTask = -294
RemTask = -288

************** HAUPTPROGRAMM ****************

        move.l 4.w,a6
```

```
        lea TaskName,a1
        jsr FindTask(a6)
        move.l d0,a1
        beq.s Was?DasGibsNicht

        jmp RemTask(a6) ; 17 KB sparen

Was?DasGibsNicht
        rts

************ Daten ***********************

TaskName dc.b "trackdisk.device",0
```

Zunächst einmal muss man einen Task mit dem Namen `trackdisk.device` suchen. Dieser wird dann mit `RemTask()` entfernt.

Ein Device vorbereiten

Zunächst muss ein Device geöffnet werden. Dies geschieht mit Funktionen, der Ähnlichkeiten mit der Funktion zum Öffnen und Schließen einer Library hat. Sie heißen `OpenDevice()` und `CloseDevice()`.

Um eine Device-Kommunikation zu betreiben, muss ein Datenblock, die sogenannte `IOReqStd`-Struktur, gefüllt und verschicket werden. In diese Struktur werden mit Daten geschrieben, die später einmal dem Betriebssystem mitteilen, was es zu tun hat.

Neben dem Block, der später die Kommunikation als Message-Packet übernimmt, müssen weitere Parameter an die `OpenDevice()`-Funktion übergeben werden: Die Unit, DF0=0, DF1=1; der Unit-Name, wir wollen das `trackdisk.device`; und bestimmte Flags, die im Regelfall null sind.

Nachdem diese Parameter in den Registern `D0`, `D1`, `A0` und `A1` übergeben worden sind, initialisiert `Exec` den I/O-Request-Block. Eine null wird bei fehlerfreiem Initialisieren zurückgegeben.

```
* InitDevice Version 2 17.3.92

************** VARIABLEN ******************

OpenDevice      = -444
CloseDevice     = -450
SendIO          = -462

************** HAUPTPROGRAMM **************

        move.l 4.w,a6

        moveq #0,d0 ; Laufwerk df0:
        moveq #0,d1 ; keine Flags
        lea TrackDiskDev,a0 ; Device-Name
        lea IOStdReq,a1     ; IOReq-Struktur
        jsr OpenDevice(a6)
        tst.l d0
        bne.s Schluss ; Wenn # Null, dann Fehler

        ; hier kommt bald unser Prg rein

        lea IOStdReq,a1
        jsr CloseDevice(a6)
Schluss rts

************** Daten **********************

TrackDiskDev
        dc.b "trackdisk.device",0
        cnop 0,2

IOStdReq ds.l 20
```

Eine Spur der Diskette einlesen

Um eine Spur von der Diskette zu lesen, müssen wir uns etwas näher mit der IOReq-Struktur befassen, die Schlüssel aller Device-Zugriffe ist.

Die Struktur hat folgende Einträge:

```
APTR    IO_DEVICE  ; Device-Node-Pointer
APTR    IO_UNIT    ; Laufwerknummer (Unit)
UWORD   IO_COMMAND ; Kommando
UBYTE   IO_FLAGS   ; special Flags
BYTE    IO_ERROR   ; Error- or Warning-Code
```

Dieses Einträge werden unter dem Strukturnamen `IORequest` zusammengefasst.

Besonders für die Diskettenprogrammierung mit dem `trackdisk.device`, muss diese Struktur noch erweitert werden. Folgende Einträge kommen hinzu:

```
ULONG   IO_ACTUAL ; actual # of bytes transfered
ULONG   IO_LENGTH ; requested # of bytes transfered
APTR    IO_DATA   ; pointer to data area
ULONG   IO_OFFSET ; offset for seeking devices
```

Alle Einträge zusammen ergeben die `IOStdReq`-Struktur.

Die `IOStdReq`-Struct ist also eine Erweiterung der `IORequest`-Strukur.

Für uns sollen vier Einträge wichtig sein:

COMMAND

Was soll unser angesprochenes Device machen?

```
CMD_INVALID     =  0 ; "invalid command"
CMD_RESET       =  1 ; resetieren
CMD_READ        =  2 ; Standard-Rread
CMD_WRITE       =  3 ; Standard-Write
CMD_UPDATE      =  4 ; alle Puffer werden geschrieben
CMD_CLEAR       =  5 ; alle Puffer werden gelöscht
CMD_STOP        =  6 ; hold current and queued
CMD_START       =  7 ; restart after stop
CMD_FLUSH       =  8 ; abort entire queue

TD_MOTOR        =  9 ; Motor ein- und ausschalten
TD_SEEK         = 10 ; suchen eines Tracks
TD_FORMAT       = 11 ; formatieren
TD_REMOVE       = 12 ; Diskettenwechsel melden
TD_CHANGENUM    = 13 ; Anzahl Diskettenwechsel
TD_CHANGESTATE  = 14 ; Ist eine Disk im Drive?
TD_PROTSTATUS   = 15 ; Ist die Disk write-protected?
TD_RAWREAD      = 16 ; Raw-Bits von Disk
TD_RAWWRITE     = 17 ; Schreibe Raw-Bits auf Disk
TD_GETDRIVETYPE = 18 ; Type des Laufwerks
TD_GETNUMTRACKS = 19 ; Anzahl Tracks ermitteln
TD_ADDCHANGEINT = 20 ; TD_REMOVE done right
TD_REMCHANGEINT = 21 ; removes softint set by ADDCHANGEINT
TD_GETGEOMETRY  = 22 ; Disk-Geometry-Table holen
TD_EJECT        = 23 ; Medium (wenn möglich) rausschmeißen
TD_LASTCOMM     = 24 ; Platzhalter für das Ende der Liste
```

LENGTH

Wie viele Bytes sollen eingelesen werden?

Ein Track besteht aus 11 Sektoren (Blöcken), die wiederum eine Größe von 512 Bytes haben. Folgende Konstanten existieren hierzu:

```
NUMSECS    =  11
TD_SECTOR  = 512
```

DATA

Wohin sollen die Daten geschrieben werden?

OFFSET

Wo soll mit dem Lesevorgang begonnen werden.

Da die `IORequest`-Struktur soweit nötig beschrieben wurden, hierzu ein ein Block aus dem gleich folgenden Programm:

```
        lea     IOStdReq,a1
        move    #CMD_READ,COMMAND(a1)  ; Read
        move.l  #2*512,LENGTH(a1)      ; Länge: 2 Sektoren
        move.l  #Puffer,DATA(a1)       ; Puffer
        move.l  #0*512,OFFSET(a1)      ; Ab Sektor 0
        jsr     DoIO(a6)
```

In der Struktur werden also Informationen über das Kommando (Lesen von Tracks), der Einleselänge, der Startadresse und dem Anfangssektor abgelegt.

Ein neue `Exec`-Funktion, namens `DoIO()` sendet diesen Block ab, und bewegt die Hardware zum Einlesen der Diskettendaten. Nach dem Absenden sind hoffentlich die Daten im Pufferbereich.

Ist der Returnwert von `DoIO()` ungleich null, so ist einer der Fehler aufgetreten:

```
NotSpecified   = 20 ; general catchall
NoSecHdr       = 21 ; Kann keinen Sektor finden
BadSecPreamble = 22 ; Sektor fehlerhaft
BadSecID       = 23 ; Sektor fehlerhaft
BadHdrSum      = 24 ; Header hat falsche Checksumme
BadSecSum      = 25 ; Daten haben falsche Checksumme
TooFewSecs     = 26 ; Nicht genug Sektoren gefunden
BadSecHdr      = 27 ; Noch ein Sektor fehlerhaft
WriteProt      = 28 ; Disk ist schreibgeschützt
DiskChanged    = 29 ; kein Disk im Laufwerk
```

```
SeekError      = 30 ; Track 0 nicht gefunden
NoMem          = 31 ; kein Speicher mehr
BadUnitNum     = 32 ; Unitnummer zu groß
BadDriveType   = 33 ; Drive nicht von trackdisk unterstützt
DriveInUse     = 34 ; Drive schon benutzt
PostReset      = 35 ; Benutzer hat Reset ausgelöst
```

Zu den Strukturen und Konstanten jetzt ein ablauffähiges Programm. Hier wird noch eine kleine Neuheit auf uns warten. Wir müssen nämlich einen Port reservieren, um die Daten vorschriftsgemäß übermitteln zu können, also nicht erschrecken.

```
* Track Lesen Version 7 17.3.92 1258 Bytes

************** VARIABLEN *********************

FindTask = -294
AddPort = -354
RemPort = -360

OpenDevice = -444
DoIO = -456
CloseDevice = -450

SigTask = 16 ; MsgPort

Device = 14 ; IOStdReq
COMMAND = 28
LENGTH = 36
DATA = 40
OFFSET = 44

CMD_READ = 2
TD_MOTOR = 9

************** HAUPTPROGRAMM *****************

        bsr.s Init
        tst.l d0
        bne.s Schluss

        bsr.s ReadAndStop

Schluss bsr.s End
        rts

************** Unterprogramme ***************************

Init    move.l 4.w,a6
        sub.l a1,a1 ; eigener Task
        jsr FindTask(a6)
        lea Port,a1
        move.l d0,SigTask(a1)
        jsr AddPort(a6)

        moveq #0,d0 ; Laufwerk df0:
        moveq #0,d1 ; keine Flags
        lea DevName(PC),a0
        lea IOStdReq(PC),a1
        jsr OpenDevice(a6)
        move.l #Port,Device(a1)
        rts

ReadAndStop
        lea IOStdReq,a1
        move #CMD_READ,COMMAND(a1)    ; Read
        move.l #2*512,LENGTH(a1) ; Länge: 2 Sektoren
        move.l #Puffer,DATA(a1)  ; Puffer
        move.l #0*512,OFFSET(a1) ; Ab Sektor 0
        jsr DoIO(a6)

        move #TD_MOTOR,COMMAND(a1)  ; Motorsteuerung
        clr.l LENGTH(a1) ; Motor aus
        jmp DoIO(a6)

End     lea IOStdReq,a1
        jsr CloseDevice(a6)
        lea Port,a1
        jmp RemPort(a6)

************** Daten ************************

DevName dc.b "trackdisk.device",0
        cnop 0,2

Port    ds.b 34 ; MsgPort hat 34 Bytes
```

```
        dc 0

IOStdReq ds.b 42 ; Struktur hat 42 Bytes
        dc 0

Puffer  ds.b 512*2 ; Platz für 2 Sektoren
```

Das Hauptprogramm besteht nur aus drei Sprüngen. `Init` initialisiert den Port und das Device, `ReadAndStop` holt einen Block, und `End` beendet das Programm.

Beginnen wir daher bei `Init`. Wir müssen uns zuerst suchen, d. h. unsere Task Struktur, denn die wird bei dem Anlegen eines Ports benötigt. Den Port legen wir in `A1`, und den Task in `D0`. Danach kann die `Exec`-Funktion `AddPort()` aufgerufen werden. Für den Port muss am Programmende Platz von 34 Bytes geschaffen werden, denn der `MsgPort` hat eine sizeof von 34 Bytes.

Nach dem Ermitteln des Ports müssen wir unser Device vorbereiten und daher öffnen. Wie dies funktioniert lernten wir ein Kapitel voraus. Das Laufwerk soll df0: sein, da dies jeder eingebaut haben sollte.

Mit `ReadAndStop` wird die erste große Aktion ausgeführt, das lesen einen Sektors. In der `IOStdReq` tragen wir unseren Wunsch nach 2 Sektoren ein, und besiegeln es mit dem `Exec`-Aufruf `DoIO()`. Jetzt wird der Block eingelesen. Danach soll der Motor ausgeschaltet werden. Die Strukturinitialisierung ist jetzt kleiner, lediglich der `COMMAND` muss angegeben werden.

End ist das Anschlussunterprogramm. Das Device wird geschlossen, und der Port mit `RemPort` entfernt.

Am Programmende ist Platz für die Portstruktur, die `IOStdReq`-Struktur, und für den Puffer, der mit einer Länge von 512*2 Bytes Platz für zwei Sektoren hat.

Komfortabler Boot-Block-Gucker

Im Bootblock einer Diskette kann sich leicht ein Virus einnisten. Das muss nicht so sein, wenn man einfach mal nachschauen könnte, was für einen Inhalt der Bootblock hat.

Zum Nachschauen soll das folgende Programm dienen. Es liest dazu wie das obige Programm die ersten beiden Sektoren von der Diskette. Danach wird der Speicherbereich auf der aktuellen Ausgabeeinheit ausgegeben. Die Routine zum Ausgeben der Zeichen kennen wir nicht und soll daher erst einmal Mittel zum Zweck sein. Im Kapitel der DOS Programmierung gehen wir auf die Ausgabe aber näher ein.

Kurz die Vorgehensweise des Programms:

1. Initialisieren
2. Platz für 2 Sektoren schaffen
3. Sektoren lesen
4. Bootblock mit den möglichen Zeichen ausgeben
5. Ende

```
* BootBlockRead Version 2 31.5.92 298 Bytes

************** VARIABLEN *********************

OpenLibrary = -408 ; Exec
CloseLibrary = -414

AllocMem = -198
FreeMem = -210

FindTask = -294
AddPort = -354
RemPort = -360

SigTask = 16 ; MsgPort

OpenDevice = -444
DoIO = -456
CloseDevice = -450

Device = 14 ; IOStdReq
COMMAND = 28
LENGTH = 36
DATA = 40
OFFSET = 44

CMD_READ = 2
TD_MOTOR = 9

Output = -60 ; Dos
Write = -48

************** Lib öffnen, Dev/Port init *****

        move.l 4.w,a6
        lea DosName(PC),a1
        jsr OpenLibrary(a6)
        move.l d0,d5 ; DosBase in d5

        move.l #1032,d0 ; etwas mehr schadet nie
        moveq #2,d1 ; Chip Mem
        jsr AllocMem(a6)
```

```
        move.l d0,d4 ; Zeiger auf Speicher in d4

        sub.l a1,a1 ; eigener Task
        jsr FindTask(a6)
        lea Port(PC),a1
        move.l d0,SigTask(a1) ; Zeiger auf Task in MsgPort
        jsr AddPort(a6) ; neuen MsgPort einrichten

        moveq #0,d0 ; Laufwerk df0:
        moveq #0,d1 ; keine Flags
        lea DevName(PC),a0
        lea IOStdReq(PC),a1
        jsr OpenDevice(a6)
        move.l #Port,Device(a1)

************** ersten beiden Blöcke einladen *

        move #CMD_READ,COMMAND(a1)    ; Read
        move.l #2*512,LENGTH(a1)      ; Länge: 2 Sektoren
        move.l d4,DATA(a1) ; Puffer
        clr.l OFFSET(a1) ; Ab Sektor 0
        jsr DoIO(a6)

        move #TD_MOTOR,COMMAND(a1)    ; Motorsteuerung
        clr.l LENGTH(a1) ; Motor aus
        jsr DoIO(a6)

************** BootBlock auf dem Screen ausgeben *

        moveq #"!",d0
        moveq #".",d1
        move.l #1024,d3 ; Länge ist 1024 Bytes
        move d3,d2 ; für die Schleife
        move.l d4,a0

ErstellAscBoot
        cmp.b (a0)+,d0
        bls.s UndWeiter
        move.b d1,-1(a0)
UndWeiter
        dbra d2,ErstellAscBoot

        move.l d5,a6
        jsr Output(a6)
        move.l d0,d1
        move.l d4,d2
        jsr Write(a6)

************** alles wieder schließen ********

        move.l 4.w,a6
        lea IOStdReq(PC),a1
        jsr CloseDevice(a6)
        lea Port(PC),a1
        jsr RemPort(a6)

        move.l d4,a1
        move.l #1032,d0 ; Länge in d0 zum Löschen
        jsr FreeMem(a6)

        move.l d5,a1
        jmp CloseLibrary(a6) ; d0 = Null (normalerweise)

************** Daten ************************

DosName dc.b "dos.library",0
DevName dc.b "trackdisk.device",0
        cnop 0,2

Port    ds.b 34 ; MsgPort hat 34 Bytes
        dc 0

IOStdReq ds.b 42 ; Struktur hat 42 Bytes
        dc 0
```

Erstmalig sollte das Programm mal nicht in Unterprogramm-Technik geschrieben sein. Der Programmierer wird Unterschiede in der Lesbarkeit und Struktur bemerken.

Neu ist das Öffnen der `Dos.Library`, die sich um die spätere Zeichenausgabe kümmern soll.

Danach beschaffen von 1032 Bytes Chip Mem. Den Zeiger auf den Speicher halten wir in `D4`, denn dieser wird zum Laden, zur Ausgabe und zum Speicherdeallokieren weiter notwendig sein.

Dann folgt die Port- und Device-Initialisierung.

Auch das Lesen der zwei Sektoren und das Ausschalten des Motors bereitet keine Schwierigkeiten.

Den BootBlock auf dem Screen ausgeben bedeutet etwas Neues in der Programmierung.

Der Bootblock ist ja ein Programm und mit dem u. U. ein bisschen Text. Da Programme aus allen erdenklichen ASCII-Zeichen bestehen, müssen nur die Text-Zeichen dargestellt werden. Dazu holen wir aus dem Speicherbereich ein Byte heraus, und vergleichen es mit dem ASCII-Ausrufezeichen. Ist das Zeichen nicht darstellbar, so wird ein Punkt ausgegeben, andernfalls das ASCII-Zeichen.

Das Schließen ist wieder einfach.

Anmerkung: Die Strukturen, die am Ende 32 + 42 = 74 Bytes belegen, können selbstverständlich auch in den Alloc einfließen.

Das Disk Operating System (DOS)

Noch einmal müssen wir in der Historien-Kiste herumkramen. Amiga plante einen Spielecomputer, so wie es heute viele Telespiele bzw. Spielkonsolen mit meist faszinierenden Möglichkeiten (mit z. B. 3D-Chip) sind. Sie bekommen ihre Daten über externe Karten und Module. Somit war ein Zugriff auf ein Medium wie Diskette, Festplatte oder sogar CD nicht eingeplant. Als das Projekt von Commodore übernommen wurde, änderte sich jedoch die Lage völlig. Nun galt es, ein DOS (Disk-Operation-System) zu entwerfen oder zu finden, die die Kommunikation mit externen Speichern möglich machte, denn der Amiga sollte nun nicht mehr eine reine Spielemaschine sein (wie es Jay Miner schon plante), sondern sich auch im professionellen Bereich etablieren. So fing Commodore leicht gedrängt mit CAOS (Commodore Amiga Operating System) (nicht verwechseln mit KAOS, einer Kopie des Atari-Betriebssystem TOS, in feinstem Assembler programmiert, mit einer enormen Geschwindigkeit und Ausmerzung von angeblich 80 Fehler - wurde aber von Atari abgelehnt) an, doch das wurde soweit nichts. Fast mit Panik kaufte Amiga das schon existierende Tripos ein, ein Betriebssystem, das auf der Basis von 68000 Prozessoren lief. Doch das OS war für die damalige Zeit kein gewöhnliches, denn Tripos war multitaskingfähig. Somit war man Atari, die unter Druck den ST herstellten, und dabei Digital Research bemühten, schon eine Nasenlänge voraus. Die englische Firma Metacomco jedoch programmierte das DOS jedoch nicht in der sonst üblichen Sprache C, sondern in BCPL (Basic Combined Programming Language), einem anderen Zweig, der aus der Ursprache beider, B, hervorging. Dies wäre auch alles nicht so schlimm, wenn da nicht wieder ein kleiner Haken bei der Sache wäre. BCPL unterscheidet sich insofern von C, dass die Strings nicht wie gewohnt mit einem Null-Byte abgeschlossen werden, sondern dass die Länge des Strings am Anfang einer Zeichenkette steht und die Daten somit um ein Zeichen verschoben werden. Eine weitere Sache von BCPL ist das Zeigerproblem. Pointer, in BCPL auch BPTR genannt, haben den Nachteil, dass sie nur auf Long-Word Adressen zeigen, es gibt somit nur durch vier teilbare Speicherbereiche. Die Pointer müssen erst mit vier multipliziert werden, um auf die richtige Adresse zu zeigen. Doch diese Nachteile vielen vorerst gar nicht so auf, und die Portierung, von Tim King vorgenommen, dauerte nur vier Wochen, und fertig war im Februar das DOS. Jedoch dauerte es noch bis zum 23.06.1985 bis er vorgestellt wurde. Zurück zum BCPL-Mischmasch. Dieser Programmier-Fremdkörper ist nun im Amiga-Betriebssystem zu finden und kann unter Umständen noch zu Problemen führen. Daher begann Commodore diesen Fremdkörper langsam zu ersetzten, und begannen Teile neu zu schreiben. In OS 2.0 ist nun BCPL (fast) verschwunden, es wurden nur noch einige kleine Pointer-Routinen beibehalten, um Kompatibilität zu wahren. Doch auch vor OS 2.0 waren deutlich Bemühungen vorhanden, und so widmete sich der Programmautor Steve Beats einem neuen Dateisystem, das unter 1.3 als Fast-File-System eingeführt wurde. Diese neue File System ist speziell für Festplatten entwickelt wurden, und kann durch Assemblerprogrammierung gegenüber dem lahmen BCPL-DOS überzeugen.

Die DOS-Versionsnummer hat sich im Laufe der Zeit auch gewandelt, und aus 1.2 wurde 1.3, jedoch unterscheiden sich die Versionen nur durch die Autoboot-Fähigkeit der Festplatte und des Speichers (RAD). Es gibt außer dieser Änderung keine weitere Modifikation, es wurde hauptsächlich an der Workbench ein paar Schönheitskorrekturen durchgeführt (das hatte sie auch nötig!). Wegen der geringen ROM-Unterschiede ist die alte Workbench 1.3 auch unter 1.2 System lauffähig, da es sich bei der Workbench ja nur um ein Programm handelt. Wer also über OS 1.2 verfügt, kann zwar die resetfeste RAM-Disk nutzen, sie ist aber nicht AutoBoot-fähig, da dies über das ROM läuft. Eigentlich konnte das Ur-OS 1.3 noch viel mehr als nur durch Autoboot laden zu können. Viele nützliche Funktionen wurden eingearbeitet, und Fehler korrigiert. Doch leider kam ein sehr mächtiges Organ dazwischen, und vermasselte die Tour von Commodore. Dieses Organ hieß Presse. Sie machte das Ur-OS 1.3 so runter und inkompatibel, so dass sich, angesichts der Schwarzmalerei, Commodore nur Traute die Autobootfähigkeit neu aufzunehmen, um den Kunden nicht zu verunsichern.

Wir haben schon erfahren, dass der Amiga über Libraries seine Funktionen verwaltet. Die `dos.library` ist eine davon, und wird mit der OpenLibrary-Funktion des `Exec` aufgerufen. Wir erhalten in `D0` unseren Pointer (inoffiziell auch in `A0`) auf die Struktur zurück, und können nun direkt über unsere negativen Offsets auf die Funktionen zugreifen.

Die Ein und Ausgabe

Eines der wesentlichen Aufgaben des Computers ist es, Daten zu verarbeiten. Die Daten, die jedoch zum Bearbeiten herangezogen werden, müssen aber irgendwie verfügbar sein. Einige Programme haben dazu Schnittstellen in Form einer Textzeile, um die Informationen einlesen zu können. Jedoch dürfe das auf die Dauer etwas anstrengend werden, wenn die Daten, die benötigt werden, immer neu eingetippt werden müssen. Der Computer muss die Daten also speichern können. Dies geschieht mit Hilfe der Dateien, die die Informationen aufnehmen. Dabei sind die Daten sequentiell hintereinander angeordnet. Die Dateien, englisch files genannt, werden auf Festspeicher gesichert. Die DOS-Library hat nun die Aufgabe eine Verbindung zwischen den Dateien und den Medien zu schaffen. Der Benutzer, der nun über ein CD-Laufwerk verfügt, muss sich genauso wenig über das Speicherverfahren kümmern, wie der Benutzer eines Laufwerkes. Die Art und Weise übernimmt also alleine das DOS. Die Routinen, die das DOS zur Kommunikation mit den Dateien bietet sind also ziemlich allgemein gehalten, reichen jedoch zum Arbeiten völlig aus.

Öffnen und Schließen einer Datei

Um mit Dateien arbeiten zu können müssen sie natürlich erst einmal geöffnet werden. Aus Hochsprachen kennen wir z. B. Funktionen wie `Open` aus BASIC und Pascal, die uns eine Datei verfügbar machen. Auch DOS bietet eine Funktion `Open()` und das entsprechende Gegenstück, die `Close()`-Funktion. Damit wir die DOS-Funktionen nutzen können, steht natürlich zuerst die Aufgabe an, die DOS-Library mittels `OpenLibrary` und dem String `"dos.library"` zu öffnen. Den neu enthaltenden Zeiger sichern wir in eine Speicherstelle `DosBase` und befördern sie im Folgenden ins `A6`-Register. Diese Registerwahl ist besonders bei 2.0 sehr wichtig, das intern das `A6`-Register gebraucht wird. Bei alten Betriebssystemversionen soll die `DosBase` auch in anderen Registern zu Ergebnissen verhelfen. Das liegt daran, dass unter 1.2/1.3 die DOS-Library eigentlich keine richtige Library war, und unrichtige Libraries benötigen nun mal die aktuelle Base (hier `DosBase`) nicht im `A6`-Register. Somit hätte man ein paar Bytes sparen können, denn mit dem verschieben des Registerinhaltes von `D0` nach `A1`, welches wir von `OpenLibrary` bekommen haben, hätten wir nun mittels `A1` auf die Dos-Routinen zurückgreifen können, und sofort nach der Anwendung `CloseLibrary()` aufrufen können, da die `Exec`-Base noch in `A6` stand. (Logisch, nicht?!). Auch das AmigaDOS Manual von Commodore wies auf diesen Trick hin, aber es kam ja (leider) wieder eine neue OS-Version heraus, die diesen Trick zunichte macht. Wir sollten uns daher merken, alle Bases in das `A6`-Register zu schreiben um damit Komplikationen aus dem Wege zu gehen.

Unser erstes Demo-Programm nutzt die Befehle `Open()` und `Close()`, um eine Datei zu öffnen. Schauen wir uns es an:

```
*************   VARIABLEN *********************

OldOpenLibrary  =       -408
CloseLibrary    =       -414

Open            =       -30
Close           =       -36

MODE_OLDFILE    =       1005

*************   HAUPTPROGRAMM *****************

        bsr     OpenDos
        move.l  d0,DosBase

        bsr     OpenFile
        move.l  d0,DateiHandle
        beq     Fehler

        bsr     CloseFile

Fehler  bsr     CloseDos
        rts

*************   Dos-Library öffnen ************

OpenDos move.l  4.w,a6  ; ExecBase
        lea     DosName,a1
        jmp     OldOpenLibrary(a6)

*************   Datei öffnen ******************

OpenFile
        move.l  DosBase,a6
        move.l  #DateiName,d1
        move.l  #MODE_OLDFILE,d2
        jmp     Open(a6)

*************   Datei schließenn **************

CloseFile
        move.l  DateiHandle,d1
        jmp     Close(a6)

*************   Dos-Library schließen *********

CloseDos
        move.l  4.w,a6
        move.l  DosBase,a1
        jmp     CloseLibrary(a6)

*************   Daten *************************

DosName         dc.b    "dos.library",0
DosBase         ds.l    1

DateiName       dc.b    "Tst",0
DateiHandle     ds.l    1
```

Wie immer ist unser Programm modular aufgebaut. Es besteht aus den Unterprogrammen `OpenDos`, `OpenFile`, `CloseFile` und `CloseDos`. Das Unterprogramm zum Öffnen der Library muss bekannt sein. Neuer wird es schon bei `OpenFile`. Zunächt einmal wird die `DosBase` in `A6` verlangt. Nun muss der Zeiger auf den Namen der Datei im `D1`-Reg stehen. Der Name kann dabei alle Pfad- und Laufwerkssetzungen einschließen, ein Mega-Dateiname wie "dh0:Up1/Prgs/Procs/Neu/Dos/Laden.TXT" wäre somit denkbar. Der Dateiname darf aber nicht länger als 30 Zeichen sein, denn mehr wird vom DOS nicht erkannt. (Das reicht ja wohl aus, wenn wir bedenken, dass der PC und Atari (PC DOS übernommen) nur billige 8 Zeichen annehmen, etwas zu wenig in unserer Zeit ist. (Und Windows erst einmal, die haben auch nur 8 Zeichen, einfach lächerlich). Ganz wichtig ist, das der String mit einem Nullbyte terminiert ist.

Vergleichen wir die Funktion einmal mit einer Pascal Prozedur. Auch dort wird eine Datei mit `Open(Name)` verfügbar gemacht. Jedoch kann man nach dem Öffnen noch nicht darauf zugreifen. Es muss dem Computer verständlich gemacht werden, um was für eine Datei es sich handelt. Haben wir ein schon bestehende Datei, z. B. eine Textdatei oder eine Grafik, und wollen sie auslesen, so wird in Pascal die Prozedur `Reset` verwendet. Eine neue Datei, die z. B. einen Speicherinhalt sichern soll, wird mit `Rewrite` eingeleitet. Vorhandene Dateien, mit dem gleichen Namen werden dabei gelöscht. Dieses System, dass der Dateimodus mit angegeben werden muss ist auch beim `Open()`-Befehl in Assembler realisiert worden. Jedoch werden nicht zwei Aufrufe benötigt, vielmehr steht im `D2`-Register der Modus der Dateien. Drei Modi stehen zur Verfügung:

```
MODE_READWRITE  =  1004
MODE_OLDFILE    =  1005
MODE_NEWFILE    =  1006
```

In unserem Programm zum reinen Öffnen einer Datei verwenden wir nur `MODE_OLDFILE`, da auf ein Programm zugegriffen werden soll. Die drei Modi erlauben dem OS-Programmierer nun zu bestimmen, wie die Datei geöffnet werden soll. `MOD_OLDFILE` öffnet eine Datei und setzt den Dateizeiger auf das erste Byte der Datei. Man hat nun Gelegenheit die Datei zu Beschreiben und zu

Lesen, sinnvoll z. B. bei Personen-Dateien, die ausgelesen werden müssen, aber auch ständig Neuerungen, in Form von Löschen und Hinzukommen unterworfen sind. Über MODE_READWRITE kann die Datei auch gelesen oder beschrieben werden, jedoch können andere Tasks, die evtl. diese Datei auch benutzen, diese nicht verändern. Mit MODE_OLDFILE wäre ein Zugriff von mehreren Tasks auf ein Programm durchaus machbar. Die Anwendung liebt etwa im Mehrbenutzerbetrieb. Eine Werkzeug-Datei soll von vielen Leuten einer Firma erweitert werden. Dazu hat jeder Mitarbeiter ein eigenes Terminal, und hängt seine Daten nun an die Datei an, oder korrigiert evtl. Fehler, z. B. in der Stückzahl oder dem Preis. DOS kontrolliert nun den Zugriff der Mitarbeiter auf diese Datei, und hängt bei Bedarf die einzelnen Teilstücke immer an die schon vorhandene Datei an. Wenn nun zwei Benutzer gleichzeitig die Datei mit Infos versorgen wollen, gelangen die Daten eines Benutzers in einen Zwischenspeicher und die Daten des anderen werden sofort gesichert. Nach dem sichern, werden die Puffer-Daten ebenso an das Programm gehängt, wie die anderen auch. Das nichts bei dem Sichern oder Laden durcheinanderkommt ist eine schwierige Aufgabe und hängt fest mit einem Multitasking Programm des Amigas zusammen. Ein Beispiel für MODE_READWRITE ist eine Textdatei. In einem Büro werden Texte für die Zeitung erstellt. Auch dabei ist eine Vernetzung der Computer Standard. Ein Mitarbeiter will nun einen Text ändern, wir allerdings feststellen, das die Datei gerade bearbeitet wird, der Zugriff ist also nicht möglich.

Die dritte Konstante MODE_NEWFILE legt nun, wie bereits gesagt, ein neues Programm an, und löscht evtl. alte gleichnamige.

Nach dem Aufruf der Funktion Open() mit der Registerbelegung D1 und D2 erhalten wir einen Wahrheitswert zurück. Dieser steht im D0-Register. Einigen wir vielleicht aufgefallen sein, das D0 als Eingangswert nicht benutzt wurden. So wäre es ja auch möglich gewesen in D0 den Namen, und in D1 den Modus als Aufrufparameter der Funktion zu definieren. Doch die Lösung dieses Rätsels ist in der Programmiersprache BCPL zu finden. BCPL nimmt als Übergabeparameter immer die Datenregister, angefangen bei D1. D0 ist nur als Rückgaberegister von Funktionen gedacht. Doch wenn dies einmal stimmen würde. Eigentlich ist unter BCPL auch das D1-Register, dass Register, in dem die Rückgabewerte erscheinen, jedoch wird intern daraus ein D0-Register gemacht. Daher finden wir bei den Funktionen unser Ergebnis auch im D1-Register Das sollte man aber nicht verwenden! Wenn sich jeder daran halten würde wäre es schön, aber es gibt ja immer welche die querschießen, Übeltäter ist hier wieder Commodore, die fleißig das D1-Reg, verwenden, da D0 für bessere Aufgaben gedacht ist. Unter 2.0 wurde dies geändert, und das Ergebnis taucht nur in D0 auf, ein Vorteil, um nicht immer den File-Handle neu ins D1-Register schreiben zu müssen. Jedoch mit einer Ausnahme (Ich will hier mal vorgreifen (sozusagen ein FORWARD)): Die LoadSeg-Funktion der DOS-Library. (Ein Produkt von Metacomco, die in der Overlay-Datei „ovs.asm“ auftaucht.) Diese Funktion erlaubt es, Benutzern Programme einzuladen, und diese auch zu starten. Da diese Funktion fleißig vom C-Compiler Lattice mit dem Ergebnis in D1 benutzt wird, gönnte man einzig dieser Funktion unter 2.0 auch das Ergebnis in D1, um ein reibungsfreies Arbeiten zu ermöglichen.

Dem aufmerksamen Programmierer wird etwas aufgefallen sein. Der Name ist ein String und keine Zahl, wie der Modus. Wie wir aus Exec kennen, sind Stringpointer aber immer in Adressregister untergebracht, und nicht in Datenregister. DOS macht hier eine Ausnahme, und verlangt auch Stringpointer in Datenregistern. Auch dies liegt an BCPL, er macht eben alles über die kostbaren Datenregister. Der Nachteil ist für Assemblerprogrammierer klar, einfache und kurze Adressierungsarten, wie xxx(PC) können nicht genutzt werden.

Ist das Returnregister mit 0 geladen, so war der Aufruf fehlerhaft. Ist der Wert ungleich null, so zeigt D0 auf eine Datei-Struktur. Schon wieder wird ein Datenregister als Zeiger missbraucht. Den Zeiger nennen wir Handle, und da es sich um Dateien handelt nennen wir ihn etwas präziser File-Handle. Das File-Handle enthält die Informationen über unsere Datei, und es ist bei weiterem Arbeiten mit dem DOS, z. B. beim Auslesen oder Schreiben, immer zu benutzen, denn es dient ihm zur Unterscheidung und Identifizierung der geöffneten Datei.

Zurück zum Programm. Wir haben uns jetzt schon so eingelesen, dass wir wieder von unserem OpenClose abgewichen sind. Wieder im Hauptprogramm sichere ich den Rückgabewert in der Variablen DateiHandle. Nun kann ich mittels einer bedingten Verzweigung nach Fehler springen, wenn die Datei nicht geöffnet werden konnte. Es ist also möglich über diese Funktion festzustellen, ob ein Datei existiert oder nicht. (Da dies aber unschön ist, werden wir noch eine weitere Funktion kennenlernen, die auch noch mehr Infos über die Datei bietet.)

Nach dem Öffnen ist der Spaß nun zu Ende und die Datei wird mittels Close() geschlossen. Verbleibende Zeichen, die Zwischengepuffert wurden, werden noch schnell auf s Medium geschrieben. Der Datei-Handle wird hier benötigt, er muss im D1-Register sein. Es ist äußerst wichtig, die genutzten Dateien, die einmal im Programm geöffnet wurden, auch wieder zu schließen, denn sollte dies nicht geschehen, so kann unter Umständen (siehe Konstante beim Open()-Aufruf) kein anderes Programm auf die Datei zugreifen, die Datei ist quasi gegen Fremdeindringen geschützt. “Freigemacht“ wird es erst wieder nach einem Reset. Die Datei hat dann im Regelfall null Bytes, die Diskstruktur kann dabei durcheinanderkommen.

Beschreiben eines Files

Nun möchte ich zeigen, wie eine Datei beschrieben werden kann. Dazu benutzen wir die Dos-Funktion Write(). Sie benötigt, genau wie andere Routinen auch, unser Datei-Handle. Mit ihm kann nun eine Zeichenkette auf das Speichermedium geschrieben werden. Bei der Zeichenkette handelt es sich um die DOS-Offsets, den DOS-Funktionen, und den entsprechenden Übergabewerten bzw. Parametern. Man sollte hierbei, falls man das Programm abtippt, auf die Leerzeichen achten. Jede Zeile ist gleich lang. Damit legen wir einen Grundstein für ein weiteres Programm.

```
* WriteFile     Version 2       1.2.1992        579 Bytes

*************   VARIABLEN **********************

OldOpenLibrary  =       -408    ; Exec
CloseLibrary    =       -414

Open            =       -30     ; Dos
Write           =       -48
Close           =       -36

CR              =       10
MODE_NEWFILE    =       1006

*************   HAUPTPROGRAMM *****************

        bsr     OpenDos
        move.l  d0,DosBase

        jsr     OpenFile
        move.l  d0,DateiHandle
        beq     OpenFehler
```

```
        bsr     Schreiben

        bsr     CloseFile

OpenFehler
        bra     CloseDos

*************   Dos-Librarys öffnen ***********

OpenDos move.l  4.w,a6
        lea     DosName,a1
        jmp     OldOpenLibrary(a6)

*************   Datei öffnen *****************

OpenFile        move.l  DosBase,a6
        move.l  #DateiName,d1
        move.l  #MODE_NEWFILE,d2
        jmp     Open(a6)

*************   Schreiben *********************

Schreiben
        move.l  DateiHandle,d1
        move.l  #DateiAnf,d2
        move.l  #DateiEnd-DateiAnf,d3
        jmp     Write(a6)

*************   Datei schließen ***************

CloseFile
        move.l  DateiHandle,d1
        jmp     Close(a6)

*************   Dos-Library schließen *********

CloseDos
        move.l  4.w,a6
        move.l  DosBase,a1
        jmp     CloseLibrary(a6)

*************   Daten *************************

DosName         dc.b    "dos.library",0
DosBase         ds.l    1

DateiName       dc.b    "ram:Tst",0
DateiHandle     ds.l    1

DateiAnf
        dc.b    "- 30   -$01e   Open(name,accessMode)(D1/D2)        ",CR
        dc.b    "- 36   -$024   Close(file)(D1)                     ",CR
        dc.b    "- 42   -$02a   Read(file,buffer,length)(D1/D2/D3)  ",CR
        dc.b    "- 48   -$030   Write(file,buffer,length)(D1/D2/D3) ",CR
        dc.b    "- 54   -$036   Input()                             ",CR
        dc.b    "- 60   -$03c   Output()                            ",CR
        dc.b    "- 66   -$042   Seek(file,position,offset)(D1/D2/D3)",CR
        dc.b    "- 72   -$048   DeleteFile(name)(D1)                ",CR
        dc.b    "- 78   -$04e   Rename(oldName,newName)(D1/D2)      ",CR
DateiEnd
```

Nach dem erfolgreichen Anlegen einer Datei „Tst“ auf der RAM-Disk, kann nun mit Hilfe der `Write()`-Funktion die Datei geschrieben werden. Unser Datei-Handle ist wiederum im `D1`-Register erwünscht, dass `D2`- und `D3`-Register wird ebenfalls benutzt. Da oft eine ganze Zeichenkette gesichert werden muss, oder ein Speicherbereich verewigt werden soll, erscheint im `D2`-Register ein Zeiger auf den Speicherbereich. Im unserem Beispiel ist das `DateiAnf`. Da der Computer ja auch nicht weiß, wie lang unser gewünschter Block ist, wir von uns im D3-Register die Länge gefordert. Anstatt nun wieder Byte für Byte auszuzählen, setzen wir ein Labe an das Ende des Textes, und erhalten die Länge durch `DateiEnd-DateiAnf`. Nach dem Initialisieren der Werte im den Registern `D1`/`D2`/`D3` kann nun mittels `Write()` beschrieben werden. Ist der Vorgang vollendet, ist im `D0`-Register die tatsächliche Länge der geschriebenen Daten zu finden. Normalerweise ist diese gleich dem vorher im `D3`-Register geschriebenen Wert. Haben wir jedoch auf der Diskette kein Platz mehr, und es können nur die Hälfte der Daten geschrieben werden, so kann dies über `D0` kontrolliert werden. Im Allgemeinen können wir aber sagen, dass bei negativen Werten, und dem Wert null im `D0`-Register ein Fehler auftrat. Dieser könnte dann noch genauer festgestellt werden, und an den Benutzer weitergeleitet werden.

Nach dem Starten des Programms können wir einmal mit AmigaRechts+i die Datei von der RAM-Disk einladen. Wir sehen die Offset-Informationen, die durch die `dc.b`-Zeilen gegeben sind.

Mit diesem Wissen können wir auch ein kleines Rescue-Programm schreiben. Der Speicher kann nach einer Bytefolge durchsucht, (z. B. ein Labelname), die Adresse gespeichert, das Ende gesucht und gesichert werden. Somit ist es nach Abstürzen möglich das Programm zu retten.

Eine Datei kodieren

Nun sind wir durchaus in der Lage ein Paar nützliche Tools für unseren Computer zu programmieren. Da wir schon die Datei Ein- und Ausgabe gut drauf haben, soll nun ein Programm entwickelt werden, welches ein Byte aus einer Datei liest, dies verschlüsselt, und so dekodiert wieder auf den Festspeicher schreibt. Eine Funktion, die Daten schreibt, ist uns bekannt, eine äquivalente

Eingabefunktion namens `Read()` ist auch durch DOS gegeben. Ist kein Zeichen mehr vorhanden, endet die Funktion. Der Name der kodieren Datei wir dabei um den Suffix `.COD` verlängert, damit immer zwischen der normalen und kodierten Datei unterschieden werden kann. Der Dekodieralgorithmus ist sehr einfach, und beschränkt sich auf die Umwandlung einzelner Bits. Dies hat jedoch den Vorteil, dass die Kodierroutine und Dekodierrroutine dieselbe ist.

Doch vorerst zu Lesen-Routine. Der Aufruf dieser ist derselbe wie beim Schreiben. `D1`-Register ist der Datei-Handle, `D2` ist ein Zeiger auf den Puffer und `D3` die Länge der zu lesenden Bytes. Auch die Rückgabewerte kommen uns bekannt vor. Wenn die in `D0` stehende Anzahl null oder negativ ist, so ist ein Fehler aufgetreten.

```
*************   VARIABLEN *********************

OldOpenLibrary  =       -408
CloseLibrary    =       -414

Open            =       -30
Read            =       -42
Write           =       -48
Close           =       -36

MODE_OLDFILE    =       1005
MODE_NEWFILE    =       1006

Muster  =       %10101010

*************   HAUPTPROGRAMM *****************

        move.l  a0,DateiNamePoint

        clr.b   -1(a0,d0)       ; CR durch Null ersetzen

        bsr     OpenDos
        move.l  d0,DosBase

        bsr     OpenEinFile
        move.l  d0,DateiEinHandle
        beq     FehlerEinFile

        bsr     OpenNewFile
        move.l  d0,DateiNewHandle
        beq     FehlerNewFile

        bsr     DateiKodieren

        bsr     CloseNewFile

FehlerNewFile
        bsr     CloseEinFile

FehlerEinFile
        bra     CloseDos

*************   Dos-Library öffnen *************

OpenDos move.l  4.w,a6
        lea     DosName,a1
        jmp     OldOpenLibrary(a6)

*************   Quelldatei öffnen **************

OpenEinFile
        move.l  DosBase,a6
        move.l  DateiNamePoint,d1
        move.l  #MODE_OLDFILE,d2
        jmp     Open(a6)

*************   Neue Datei öffnen **************

OpenNewFile
        move.l  DateiNamePoint,a0
        lea     Puffer,a1
CopyName
        move.b  (a0)+,(a1)+
        bne     CopyName
        subq.l  #1,a1   ; vom Kopieren korrigieren
        move.b  #".",(a1)+      ; Suffix anhängen
        move.b  #"C",(a1)+
        move.b  #"O",(a1)+
        move.b  #"D",(a1)+
        clr.b   (a1)    ; zum Schluß Nullbyte setzen
        move.l  #Puffer,d1
        move.l  #MODE_NEWFILE,d2
        jmp     Open(a6)

*************   Daten kodieren ****************
```

```
DateiKodieren
        move.l  DateiEinHandle,d1
        move.l  #Puffer,d2
        moveq   #1,d3
        jsr     Read(a6)        ; 1. Lesen
        tst.l   d0
        beq     DateiKodEnd

        eor.b   #Muster,Puffer  ; 2. Kodieren

        move.l  DateiNewHandle,d1
        move.l  #Puffer,d2
        moveq   #1,d3
        jsr     Write(a6)       ; 3. Schreiben

        cmp     #1,d0   ; Kein Fehler?
        beq.s   DateiKodieren

DateiKodEnd
        rts

*************   Eingabedatei schließen ********

CloseEinFile
        move.l  DateiEinHandle,d1
        jmp     Close(a6)

*************   Neue Datei schließen **********

CloseNewFile
        move.l  DateiNewHandle,d1
        jmp     Close(a6)

*************   Dos-Library schließen *********

CloseDos
        move.l  a6,a1
        move.l  4.w,a6
        jmp     CloseLibrary(a6)

*************   Daten ************************

DosName dc.b    "dos.library",0
DosBase dc.l    0

DateiEinHandle
        ds.l    1
DateiNewHandle
        ds.l    1

DateiNamePoint
        ds.l    1

Puffer  ds.b    32+4
```

Wiederum bauen wir unser Programm modular auf. Jedoch kommen vorerst als Konstanten die Definitionen für Read, mit dem Offset -42, und die beiden Modi für das Öffnen der Quelldatei und Zieldatei dazu. Mittels `OpenEinFile` öffnen wir die kodierte oder unkodierte Datei. Mit dem Unterprogramm `OpenNewFile` wird die Ausgabedatei erstellt. In früheren Kapiteln habe ich die Möglichkeit Dateiparameter zu verarbeiten schon einmal angesprochen. Doch noch mal zur Wiederholung: Ein Zeiger auf den Eingabestring ist in `A0`, abgeschlossen mit einem CR und in `D0` ist die Länge der Eingabe. Da wir zum Öffnen einer Datei den Namen benötigen, sichern wir am Anfang des Programms diesen in `DateiNamePoint`, um in an späterer Stelle wieder zu benutzen. Da allerdings die Dos-Konvention verlangt, das die Strings mit einem Nullbyte abschließen, und nicht mit dem CR, können wir über den CLR-Befehl un ARI mit negativer Adressdistanz schnell ein Nullbyte ans Ende bringen. Wir haben gesagt, dass sich der neue Dateinamen aus dem Alten und der Kennung `.COD` zusammengesetzt. Also kopieren wir den Dateinamen in einen freien Speicherbereich, den wir mit Puffer vordefiniert haben, und hängen einzeln die fünf Bytes `.`, `C`,`O`,`D` und Nullbyte an. Der neue Dateiname stet nun im Puffer, und die Ausgabedatei kann geöffnet werden.

Die Hauptroutine des Programme heiß `DateiKodieren`. Der Vorgang lässt sich im Wesentlichen in drei Zwischenstufen darstellen. Zuerst muss einmal ein Byte, das es dann später zu kodieren gilt, gelesen werden. Dabei haben wir die Möglichkeit, unseren Puffer, den wir für unseren Dateinamen angelegt haben, wiederzuverwenden. Die geht aus dem einfachen Grund, da er im Weiteren nicht mehr verwendet wird, denn der Dateiname ist für die Dateiidentifikation unwichtig, es zählt alleine der Datei-Handle. Nachdem wir nun ein Zeichen gelesen haben, müssen wir auch testen, ob die Datei zu Ende ist. Wir erinnern uns, bei null ist kein Byte gelesen, daher verzweigen wir bei dem Wert null im `D0`-Register zu `DateiKodEnd`. Der nächste Schritt ist der eigentliche Dekodierschritt. Wir verwenden zur Verschlüsselung die XOR-Funktion Als Konstante haben wir Muster am Anfang definiert. So ist eine leichte Änderung an neue Dekodierkonstanten gewährleistet. Die XOR-Funktion negiert nun einige Bits unseres Puffers. Im Folgenden muss dieses Byte natürlich wieder an die neue Datei angehängt werden. `DateiNewHandle` enthält die Informationen über diese, und nach dem Schreiben müssen wir natürlich noch abfragen, ob der Schreibvorgang erfolgreich war. Da wir ein Byte schreiben wollten, vergleichen wir nun, ob auch tatsächlich im `D0`-Register eine eins für alleiniges Byte steht. Wenn das so ist, kann es weitergehen mit einem Sprung nach `DateiKodieren`. Letztendlich müssen die Dateien noch geschlossen werden. Ist dies geschehen, können wir die neue Datei bewundern.

Das Programm wäre in folgendem noch zu erweitern: Der Zugriff auf die Diskette ist sehr langsam, da die Datei evtl. auf sehr verschieden Sektoren gesichert werden muss. (Da wo gerade Platz ist). Um dem entgegenzuwirken ist zwar ein Puffer eingerichtet, doch hat das meist keinen Zweck. Wer also seine Dateien umwandeln will, der sollte sie vorher auf die RAM-Disk kopieren, und nach Erstellen der neuen Datei diese auf seinen Festspeicher übertragen.

Ein vielleicht weiterer Nachteil ist die Indexvergrößerung. Die kodierte Datei enthält die Endung `.COD`, jedoch enthält eine zurückkopierte Datei den die Endung `.COD.COD`. Wer will, kann nun das Programm insofern erweitern, dass die Endung erkannt wird, und der Rattenschwanz `.COD.COD` wegfällt.

Fenster als Spezialfall einer Datei

Nun wird es auf die Dauer wenig sinnvoll erscheinen, unsere erstellten Dateien immer durch einen Editor oder ein TVP (Text-Verarbeitungs-Programm) auf dem Schirm zu bekommen. Daher wollen wir nun einen Sonderfall einer Datei kennenlernen. Das CLI-Fenster. Natürlich haben wir mit dem Umgang des Amigas schon mit dem CLI (Command-Line-Interpreter) Bekanntschaft gemacht. Als Alternative zur Workbench ist man mit ihm in der Lage mittels Befehlen, z. B. `dir`, `list`, `mount`, usw. auf Speichermedien zurückzugreifen, und so dem Rechner nicht über Icons und Maus, sondern vielmehr über die Tastatur Kommandos zu erteilen. Schon beim Booten des Rechners erscheint ein Fenster, mit der typischen Copyright Meldung und der Versionsnummer. Nun kann gegebenenfalls über der Startup-Sequence (im `s`-Verzeichnis) Stapelprogramme abgearbeitet werden (wie die `autoexec.bat` bei PCs oder der Auto-Ordner bei Atari). Ist diese leer, erscheint unser Cursor. Nun können wir die Befehle eintippen. Beenden können wir das CLI-Fenster es mit dem Befehl `EndCli`. Wollen wir uns einmal einen CLI-Befehl genauer anschauen. Dazu soll uns der `ECHO`-Befehl helfen, da wir ihn auch einmal nachschreiben wollen. Mittels dieses `ECHO`-Befehls lassen sich Informationsmeldungen auf dem aktuellen Window (normalerweise das CLI-Fenster) ausgeben, zum Beispiel

```
ECHO "Hallo, du da vor dem Bildschirm"
```

Nach dem Drücken der Returntaste erscheint dieser Text, allerdings ohne Anführungsstriche. Diese sind nur dazu da, den Text als ganzes zu interpretieren, denn die Trennung durch Leerzeichen würde sonst das Ende anzeigen, und eine Fehlermeldung würde uns bescheren. Für einzelne Wörter braucht man keine Anführungsstriche, und für mehrere Meldungen sollte man besser mittels `TYPE` eine Datei ausgeben.

Wer sich schon etwas näher mit dem DOS beschäftigt hat, der weiß, das man die Ausgabe, die normalerweise auf dem Window erfolgt, in eine Datei umleiten lassen kann. Dies ist mit dem Zeichen `>` möglich. Nach dem erwünschten Befehl, z. B. `ECHO` geben wir das `>` und die Ausgabedatei an. Die Standart-Ausgabe erfolgt nun über die Datei, die z. B. durch `PAR:`, `SER:`, oder etwa durch Programmverzeichnisse (`DFx:`, `RAM:`, `RAD:`) gegeben sein kann. Mit Hilfe dieser Umleitung ist ein auch ein teilweises Retten einer kaputten Datei möglich. Würden wir mit dem `COPY`-Befehl eine kaputte Datei kopieren wollen, so wehrt sich das OS und gibt eine Fehlermeldung aus. Nehmen wir unseren neuen Weg, und leiten nun den Inhalt einer Datei mittels `>` auf eine neue um, so wird das, was ganz ist, kopiert, und das weitere nicht. Die kaputte Datei soll z. B. `PRG1` heißen, die neue `PRG1.RETTUNG`. Mittels der Anzeigefunktion `TYPE` lässt sich nun folgendes schreiben:

```
TYPE df0:PRG1 > df0:PRG1.RETTUNG
```

Um eine Datei auszudrucken kann man mittels Ansprechen der Parallelen Schnittstelle folgendes schreiben:

```
TYPE PRG1 > PRT:
```

Doch kommen wir nach dem Abschweifen langsam zu unserem Ausgangsthema zurück. Die Ausgabe auf ein Window. Bei Benutzung des CLI ist die Ausgabe immer auf das Fenster gelenkt. Es besteht aber die Möglichkeit die Ausgabe umzulenken. Wer nun Kombiniert, wird sich sagen: „Es handelt sich bei der Ausgabe um ein Sonderfall“. Dies ist richtig. Das Fenster unter OS ist auch als `CON:` für „Console“ oder `RAW:` für ein etwas neueres Fenster unter 1.3 bekannt. Soll also die Datei `Text.TXT` auf einen neues Window ausgegeben werden, lenken wir die Datei um, und schreiben:

```
TYPE Text.TXT > CON:
```

Nun wir ein neues Fenster geöffnet, und unsere Datei ist sichtbar. Wir haben also erreicht, was wir wollten, die Anzeige eines Textes in einem Window. Dieses Window kann aber noch vielfältig modifiziert werden, da eigentlich zum Öffnen noch Parameter notwendig sind. So können wir die Größe des Windows frei bestimmen. Dazu lassen sich nach `CON:` die Fensterabmessungen durch Trennung eines Schrägstriches eingeben. Unser Fenster soll die Koordinaten 10,10,600,200 haben. Nach einem weiteren Schrägstrich kann ein Fenstername definiert werden. Diese Möglichkeiten bietet ein Fenster unter 1.3, jedoch wissen wir aus vielen anderen Programmen, das da noch weitere Dinger (Gadgets) am Window sind, die es uns ermöglichen das Fenster nach hinten zu klappen oder z. B. zu verkleinern. Diese Möglichkeiten bietet das DOS aber nur ab OS 2.0. Danach sind folgende Fensterdefinitionen hinzugekommen. (`NO`)`SIZE`, (`NO`)`DDARG`, (`NO`)`DEPTH`, (`NO`)`CLOSE`, `NOBORDER`, `BACKDROP`. Die Wörter sprechen für sich, und so kann man der Fensterdefinition das Verkleinerungs-, Verlegungs-, und Schließsymbol entnehmen. Zudem besteht die Möglichkeit, das Fenster ohne Rahmen darzustellen, und es im Hintergrund aufzubauen. Unser Fenster könnte also so aussehen:

```
"CON:10/10/600/200/DOS-Fenster/NOSIZE/NOBORDER"
```

Ein Programm, das nun anstatt einer Datei ein Fenster öffnet wird nun vorgestellt. Es gibt mittels `Write`-Befehl aus der `Dos.Lib` einen Text aus. Nach einer kleinen Wartezeit wird das Fenster wieder geschlossen.

```
* Fensterausgabe        Version 4       1.2.91  220 Bytes

*************   VARIABLEN **********************

OldOpenLibrary  =       -408    ; Exec
CloseLibrary    =       -414

Open            =       -30     ; Dos
Write           =       -48
Delay           =       -198
Close           =       -36

MODE_NEWFILE    =       1006
```

```
*************   HAUPTPROGRAMM *****************

        bsr     OpenDos
        move.l  d0,DosBase

        bsr     OpenFenster
        move.l  d0,FensterHandle
        beq.s   WinFehler

        bsr     Schreiben
        bsr     Warten

        bsr     CloseFenster
WinFehler
        bra     CloseDos

*************   Dos-Library öffnen ************

OpenDos move.l  4.w,a6
        lea     DosName,a1
        jmp     OldOpenLibrary(a6)

*************   Fenster öffnen ****************

OpenFenster
        move.l  DosBase,a6
        move.l  #Fenster,d1
        move.l  #MODE_NEWFILE,d2
        jmp     Open(a6)

*************   Schreiben ins Fenster *********

Schreiben
        move.l  FensterHandle,d1
        move.l  #TextAnf,d2
        moveq   #TextEnd-TextAnf,d3
        jmp     Write(a6)

*************   Warten ************************

Warten  moveq   #100,d1
        jmp     Delay(a6)

*************   Fenster schließen *************

CloseFenster
        move.l  FensterHandle,d1
        jmp     Close(a6)

*************   Dos-Library schließen *********

CloseDos
        move.l  4.w,a6
        move.l  DosBase,a1
        jmp     CloseLibrary(a6)

*************   Daten *************************

DosName dc.b    "dos.library",0
DosBase dc.l    1

Fenster dc.b    "CON:10/10/600/200/Dies ist ein einfaches DOS-Fenster!"
;       dc.b    "/NOSIZE/NODRAG/NODEPTH/NOCLOSE"
        ; Für 2.0 leer
        dc.b    0
FensterHandle
        ds.l    1

TextAnf dc.b    " Hallo, ich bin der Text"
TextEnd
```

Das Programm ist annähern gleich aufgebaut wie die anderen auch. Daher erscheint es wieder sinnvoll beim Abtippen ein schon vorher getipptes Programm um die neuen Blöcke zu kürzen, und das dazukommende einzufügen. Was wieder wichtig zu wissen ist, das auch das Fenster ein Handle benutzt. Es ist wesentlich für DOS, das über Handles der Zugriff auf alle Medien erfolgt. Mittels unseres Handles können wir den Text ausgeben. Das ist allerdings schon bekannt. Auch das Öffnen des Fensters geschieht über den Modus `MODE_NEWFILE`, da ja unser Fensterchen neu geöffnet wir. Daher ist eine große Unabhängigkeit erreicht.

Ausgabe einer Datei im aktuellen CLI-Fenster

Nun soll unser Ausgabeprogramm um einige Features erweitert werden. Häufig ist es störend, dass ein neues Fenster geöffnet wird. Nehmen wir das obere Programm und compilieren es, und führen es auf dem CLI aus. Obwohl schon ein Fenster geöffnet ist, erscheint ein neues. Das ist in den meisten Fällen ärgerlich, und für die Ausgabe kleinen Meldung, die z. B. die Parameterübergabe

als Info auf dem Schirm bringen, überflüssig, und es wäre völlig ausreichend und wünschenswert, dass sie in dem Fenster erscheinen, aus dem gerade das Programm aufgerufen wurde. Das OS bietet daher eine kleine Besonderheit, und zwar, auf das gerade geöffnete Window zugreifen zu könne. Mit einem kleinen Sternchen ist dies machbar: anstatt der `CON:….`-Zeile erscheint nur einfach `*` als Fensterdefinition. DOS nimmt also automatisch das aktuelle Fenster, und gibt dort die Meldungen und Text aus.

```
* ReadFile      Version 4       2.1.1991        551 Bytes

*************   VARIABLEN *********************

OldOpenLibrary  =       -408
CloseLibrary    =       -414

Open            =       -30
Read            =       -42
Write           =       -48
Delay           =       -198
Close           =       -36

MODE_NEWFILE    =       1006
MODE_OLDFILE    =       1005

*************   HAUPTPROGRAMM *****************

        bsr     OpenDos
        move.l  d0,DosBase

        bsr     OpenFenster
        move.l  d0,FensterHandle
        beq     WinFehler

        bsr     OpenFile
        move.l  d0,DateiHandle
        beq     OpenFehler

        bsr     DateiAusgeben

        bsr     Warten

OpenFehler
        bsr     CloseFenster

WinFehler
        bsr     CloseDos

        rts

*************   Dos-Library öffnen ************

OpenDos move.l  4.w,a6
        lea     DosName,a1
        jmp     OldOpenLibrary(a6)

*************   Fenster öffnen ****************

OpenFenster
        move.l  DosBase,a6
        move.l  #Fenster,d1
        move.l  #MODE_NEWFILE,d2
        jmp     Open(a6)

**********      Datei öffnen ******************

OpenFile
        move.l  #DateiName,d1
        move.l  #MODE_OLDFILE,d2
        jmp     Open(a6)

**********      Datei lesen + ausgeben ********

DateiAusgeben
        move.l  DateiHandle,d1
        move.l  #Puffer,d2
        move.l  #300,d3
        jsr     Read(a6)

        move.l  d0,AnzZeichen
        beq     DateiAusgabEnd  ; keine Zeichen in
Datei

        move.l  FensterHandle,d1
        move.l  #Puffer,d2
        move.l  AnzZeichen,d3
        jsr     Write(a6)
DateiAusgabEnd  rts
```

```
*************   Warten ************************

Warten  moveq   #100,d1
        jmp     Delay(a6)

*************   Fenster schließen *************

CloseFenster
        move.l  FensterHandle,d1
        jmp     Close(a6)

*************   Dos-Library schließen *********

CloseDos
        move.l  4.w,a6
        move.l  DosBase,a1
        jmp     CloseLibrary(a6)

*************   Daten *************************

DosName dc.b    "dos.library",0
DosBase ds.l    1

Fenster dc.b    "*",0   ; aktuelles Fenster
FensterHandle   ds.l    1

DateiName       dc.b    "df0:Daten/Texte/a68k.doc",0
DateiHandle     dc.l    0

AnzZeichen      dc.l    0

Puffer          ds.b    301
```

Das Programm soll langsam zu einem kleinen `TYPE`-Befehl heranwachsen. Jetzt allerdings soll erst einmal ein Textfile geöffnet werden, und aus diesem 300 Byte gelesen werden. Diese werden in Puffer gesichert, und anschließend über den `Write`-Befehl im Fenster ausgegeben. Das Programm hat wiederum große Ähnlichkeit mit den schon beschriebenen, viel zu sagen gibt es hier nicht.

CLI Ausgabe

Wenn wir unseren `TYPE`-, `LIST`- oder `DIR`-Befehl nehmen, bei allen fällt auf, dass die Ausgabe auf ein Fenster erfolgt. Wie wir nun wissen, können wir dies allerdings auf eine Datei umlenken. Doch in unseren obigen Programmen erzwingen wir immer eine Ausgabe auf einem Window, oder als Sonderfall im aktuellen CLI-Window. Doch was ist nun, wenn der Benutzer nicht an die Meldungen interessiert ist, und sie z. B. mittels `NIL:` ins nichts, mit `SER:` an andere Computer, mit `PRT:` an den Drucker, oder sogar mit `SPEAK:` an den internen Laberkopp schickt? Dann ist unser Programm aufgeschmissen, und auch der uninteressierteste Programmuser bekommt unsere Meldungen.

Doch damit dies nicht geschieht, und aufwendige Abfragen nötig werden, greift uns unser Betriebssystem ein wenig unter die Arme. Es Existiert in der `Dos.Lib` einen Befehl namens Output. Er wir ohne Übergabeparameter mit dem Offset -60 aufgerufen, und holt uns unsere Datei-Indentifikation herein, die im Augenblick als Ausgabedatei definiert ist. Dies ist natürlich wunderbar, und da die Standartausgabe meist im CLI-Fenster abgeht, ersparen wir uns so ein Öffnen des Window, denn Output lieft uns sofort einen Handle im `D0`-Register zurück. Eigentlich eine sehr nützliche Funktion. Wenn man ganz sicher gehen will, das die Ausgabe in ein Window gehen soll, so muss man zwangsläufig ein DOS-Fenster selber öffnen. Doch die Output Funktion liefert die idealen Parameter, da auch gewünschte Ausgaben an die betreffende Stelle, wie z. B. Drucker, sofort und einfach über ein Handle möglich ist.

```
* CLI Ausgabe Version 3 1.2.91  140 Bytes

*************   VARIABLEN **********************

OldOpenLibrary  =       -408
CloseLibrary    =       -414

Output          =       -60
Write           =       -48

*************   HAUPTPROGRAMM *****************

        bsr     OpenDos
        move.l  d0,DosBase

        bsr     FindeDatInfo
        move.l  d0,FensterHandle
        beq.s   WinFehler

        bsr     Schreiben

WinFehler
        bra     CloseDos

*************   Dos-Librarys öffnen ***********

OpenDos move.l  4.w,a6
```

```
        lea     DosName,a1
        jmp     OldOpenLibrary(a6)

*************   Datei Indentifikation holen ***

FindeDatInfo
        move.l  DosBase,a6
        jmp     Output(a6)

*************   Schreiben ins Fenster *********

Schreiben
        move.l  DosBase,a6
        move.l  FensterHandle,d1
        move.l  #TextAnf,d2
        moveq   #TextEnd-TextAnf,d3
        jmp     Write(a6)

*************   Dos-Library schließen *********

CloseDos
        move.l  4.w,a6
        move.l  DosBase,a1
        jmp     CloseLibrary(a6)

*************   Daten ************************

DosName dc.b    "dos.library",0
DosBase ds.l    1

FensterHandle
        ds.l    1

TextAnf dc.b    10," Hallo, ich bin der Text!",10,10
TextEnd
```

Das Unterprogramm `FindeDatInfo` ist hierbei der Informationsgeber, denn in diesem wird die Output Funktion aufgerufen. Wieder zurück im Hauptprogramm wird mittels eines TST-Befehls wieder bei einem Fehler verzweigt. Ging alles klar kann das Unterprogramm `Schreiben` in Aktion treten. Es gibt den Text, stehend an `TextAnf`, mit der Länge `TextEnd-TextAnf` nun mittels dem `FensterHandle` aus. (Wir hätten den Handle auch `DruckerHandle` nennen können, wenn die Ausgabe meistens an den Drucker läuft. Ich habe hier nur `FensterHandle` genommen, da es sich fast immer um ein Fenster als Ausgabemedium handelt.) Ist die Ausgabe beendet, so wir die Dos-Lib. geschlossen, aber seltsamerweise nicht unsere Datei. Das dürfte allerdings einleuchten, warum nicht. Es handelt sich ja um eine schon bestehende Datei, wenn wir es einmal so nennen wollen, und auf diese greifen wir nun zu. Und da wir sie auch nicht öffnen, sondern uns nur einen Zugriff erlauben, wir diese Datei auch nicht geschlossen.

CLI Ausgabe mit Textstyle

Bestimmt grinste einem schon einmal eine bunte, schräge, unterstrichene, dicke, inverse Schrift an. Das Geheimnis dieser Schrift Styles ist, kann einfach gelüftet werden. Das RAW-Fenster. In diesem Fenster, das im Gegensatz zu dem Console-Fenster steht, kann neben ganz normalen Textausgaben auch Steuerzeichen übersandt werden. Alle diese Sequenzen, wie sie genannt werden, sind mit einem Einleitungszeichen versehen. Dieses Zeichen, mit dem Hex-Wert $9B, wird auch "Control Sequence Introducer", oder kürzer CSI genannt. Die nun folgenden Zeichen bewirken die Funktion, wobei noch Dezimalzahlen, gekennzeichnet mit *n*, auftauchen können. Eine Liste der wichtigsten Sequenzen ist in der nun folgenden Tabelle gegeben.

```
\begin{tabular}{lll}
{\bf Sequenz} & {\bf Funktion} \\
\$9b,n,\$40 & n Leerzeilen einschieben & \\
\$9b,n,\$41 & Cursor um n Zeilen nach oben & \\
\$9b,n,\$42 & Cursor um n Zeilen nach unten & \\
\$9b,n,\$43 & Cursor um n Spalten nach rechts & \\
\$9b,n,\$44 & Cursor um n Spalten nach links & \\
\$9b,n,\$45 & Cursor um n Zeilen nach oben und in Spalte 1 & \\
\$9b,n,\$46 & Cursor um n Zeilen nach unten und in Spalte 1 & \\
\$9b,n1,\$3b,n2 & Cursor in Zeile n1 und Spalte n2 setzen & \\
\$9b,\$4a & Window von Cursor ab löschen & \\
\$9b,\$4b & Zeile von Cursor ab löschen & \\
\$9b,\$4c & ganze Zeile einfügen & \\
\$9b,\$4d & ganze Zeile löschen & \\
\$9b,n,\$50 & n Zeilen von Cursor an löschen & \\
\$9b,n,\$53 & n Zeilen von Cursor an hochschieben & \\
\$9b,n,\$54 & n Zeilen von Cursor an runterschieben & \\
\$9b,st$|$vf$|$hf,\$6d & st = Stiel (0,1,3,4,7) & \\
 & vf = Vordergrundfarbe 30-37 & \\
 & hf = Hintergrundfarbe 40-47 jeweils Farbreg. 0-7 & \\
 & \$9b,``0m`` = normal & \\
 & \$9b,``1m`` = fett & \\
 & \$9b,``3m`` = kursiv & \\
 & \$9b,``4m`` = unterstrichen & \\
 & \$9b,``7m`` = invers & \\
 & \$9b,``30m`` = Vordergrund blau & ; alte Farben!!\\
 & \$9b,``31m`` = Vordergrund weis & \\
 & \$9b,``32m`` = Vordergrund schwarz & \\
```

```
 & \$9b,``33m`` = Vordergrund orange & \\
 & \$9b,``40m`` = Hintergrund blau & \\
 & \$9b,``41m`` = Hintergrund weis & \\
 & \$9b,``42m`` = Hintergrund schwarz & \\
 & \$9b,``43m`` = Hintergrund orange & \\
 &  & \\
\$9b,n,\$74 & setze mit n die maximale Anzahl darstellbarer Zeichen & \\
\$9b,n,\$75 & setze mit n die maximale Zeilenlänge in Zeichen & \\
\$9b,n,\$78 & Ausgabe gegenüber linken Fensterrand um n Pixel nach links verschoben & \\
\$9b,n,\$79 & Ausgabe gegenüber oberem Fensterrand um n Pixel nach unten verschoben & \\
\$9b,\$20,\$70 & Cursor wird unsichtbar & \\
\$9b,\$30,\$20,\$20 & Cursor wird wieder sichtbar & \\
\$9b,\$71 & es kommen Fensterausmaße, die folgendes Format haben: & \\
 & \$9b,\$31,\$3b,\$31,\$3b,n1,\$3b,n2,73 & \\
 & n1=Anzahl Zeilen, n2 = Anzahl Spalten & \\
\end{tabular}
```

Andere Sequenzen wirken wie:

$o8	Backspace
$oa	Linefeed und Cursor nach unten
$ob	Cursor hoch
$oc	Fensterinhalt löschen
$od	Wagenrücklauf; Carriage Return (CR)
$of	Sonderzeichen einstellen
$oe	von Sonderzeichen zurück zu Normalzeichen

In der Tabelle finden wir viele Sequenzen, die für den Umgang mit dem CLI sinnvoll sind. Der Ursprung der meisten Codes ist historisch zu sehen. Als es noch Terminals gab, vergleichbar mit Telex, musste durch eine einfache Ansteuerung eine Wirkung auf dem Schirm erfolgen, quasi eine Art Programmiersprache für Zeichensetzung.

Mit dem folgenden Programm will ich demonstrieren, wie die Ausgabe im Praktischen aussieht. Um sich die Arbeit einfach zu machen, und nicht immer $9b.... zu schreiben benutze ich Makros, die dann an den entsprechenden Positionen eingesetzt werden.

```
* CLI Ausgabe_mit_Stil  Version 2  1.2.91   262 Bytes

*************   VARIABLEN *********************

OldOpenLibrary  =       -408
CloseLibrary    =       -414
Output          =       -60
Write           =       -48

CLR_SCREEN      MACRO
        dc.b    $c
        ENDM

STILTYP MACRO
        dc.b    $9b,\1+"0",";31;40m"
        ENDM

*************   HAUPTPROGRAMM *****************

        bsr     OpenDos
        move.l  d0,DosBase

        bsr     FindeDatInfo
        move.l  d0,FensterHandle
        beq.s   WinFehler

        bsr     Schreiben

WinFehler
        bra     CloseDos

*************   Dos-Librarys öffnen ***********

OpenDos move.l  4.w,a6
        lea     DosName,a1
        jmp     OldOpenLibrary(a6)

*************   Datei Indentifikation holen ***

FindeDatInfo
        move.l  DosBase,a6
        jmp     Output(a6)
```

```
*************   Schreiben ins Fenster *********

Schreiben
        move.l  DosBase,a6
        move.l  FensterHandle,d1
        move.l  #TextAnf,d2
        move.l  #TextEnd-TextAnf,d3
        jmp     Write(a6)

*************   Dos-Library schließen *********

CloseDos
        move.l  4.w,a6
        move.l  DosBase,a1
        jmp     CloseLibrary(a6)

*************   Daten ************************

DosName dc.b    "dos.library",0
DosBase ds.l    1

FensterHandle
        ds.l    1

TextAnf CLR_SCREEN
        dc.b    10," Hallo, ich bin der CLI Text, und kann noch besonders hervorgehoben
werden",10,10
        STILTYP 1
        dc.b    10,"Fett",10
        STILTYP 4
        dc.b    10,"und Unterstrichen",10
        STILTYP 0
        dc.b    10,"Wieder normal",10,10
TextEnd
```

Vieles ähnelt dem vorigen Programm, was hinzukam sind die Makros und der veränderte Text.

Eigenes CLI-Programm: STIL

Nun wollen wir aber richtig in die CLI-Tool Box einsteigen. Wir wissen ja, dass das OS uns einen Zeiger auf die CLI-Übergabe-Zeichenkette in A0 übergibt, und damit kann man schon etwas anfangen. Unser Programm STIL soll dazu dienen den Zeichensatz-Modi zu ändern. Dies ist zwar völlig hohl und Panne, da es über CLI natürlich auch geht (wir machen es ja nicht anders), jedoch ist ein Programm immer übersichtlicher und sowieso schöner, wenn so etwas in einer startup-sequence auftaucht.

```
* STIL  Version 3       1.2.91  75 Bytes

*************   VARIABLEN ********************

OldOpenLibrary  =       -408
CloseLibrary    =       -414

Output          =       -60
Write           =       -48

*************   HAUPTPROGRAMM ****************

        subq    #1,d0
        beq.s   NormalZeichen   ; Keine Angabe

        move.b  (a0),Zeichen

NormalZeichen
        move.l  4.w,a6
        lea     DosName(PC),a1
        jsr     OldOpenLibrary(a6)
        move.l  d0,a6

        jsr     Output(a6)
        move.l  d0,d1
        beq.s   Fehler

        move.l  #TextAnf,d2
        moveq   #TextEnd-TextAnf,d3
        jsr     Write(a6)

Fehler  move.l  a6,a1
        move.l  4.w,a6
        jmp     CloseLibrary(a6)

*************   Daten ************************

DosName dc.b    "dos.library",0
```

```
TextAnf dc.b    $9b
Zeichen dc.b    "0"
        dc.b    ";31;40m"
TextEnd
```

Ein ECHO-Klon

Der ECHO-Befehl dürfe allgemein bekannt sein. Unsere jetzige Aufgabe soll es sein, einen einfachen `ECHO`-Befehl nachzuschreiben. Dabei setzen wir unser ganzes Optimierer-Wissen ein, um eine möglichst kurze Version zu haben.

```
* ECHO  Version 1       4.2.91  52 Bytes

*************   VARIABLEN *********************

OldOpenLibrary  =       -408
CloseLibrary    =       -414

Output          =       -60
Write           =       -48

*************   HAUPTPROGRAMM *****************

        move.l  a0,d2   ; d2 = Zeiger auf String
        move.l  d0,d3   ; d3 = Länge des Strings

*************   Dos-Library öffnen ************

        move.l  4.w,a6
        lea     DosName(PC),a1
        jsr     OldOpenLibrary(a6)

*************   Datei Indentifikation holen ***

        move.l  d0,a6
        jsr     Output(a6)
        move.l  d0,d1   ; d1 = FensterHandle
        beq.s   Fehler

*************   String schreiben **************

        jsr     Write(a6)

*************   Dos-Library schließen *********

Fehler  move.l  a6,a1
        move.l  4.w,a6
        jmp     CloseLibrary(a6)

*************   Daten *************************

DosName dc.b    "dos.library",0
```

Bei dem Programm machen wir uns wieder zu nutze, dass der Übergabestring in `A0`, und die Länge diesen in `D0` steht. Wir kopieren einfach die Register nach `D2` und `D3`, denn sie werden bei dem OS-Aufruf `OpenLibrary()` und `Output()` nicht zerstört. Und da haben wir sie ja gleich in den richtigen Register, wo wir sie für `Write()` haben wollen. Besser kann es ja nicht kommen.

Anzeige des verfügbaren Speichers

Passt es, oder passt es nicht? Gerade bei 512 KB machen sich einige Leute noch Gedanken, ob das Programm nun bei drei Editoren und fünf Uhren noch in den Speicher passen. Für diese Sorte Mensch ist unser Programm `AvailMem` bestimmt das richtige, denn es wird der verfügbare Chip/Fast und TotalRAM greifbar sichtbar gemacht. Natürlich werden machte erst zögern und denken, „Oh Schreck, oh Graus, eine ASCII-Zahl-Wandel-Routine wird benötigt!“, doch halt, man denke an die geniale `RawDoFmt()` Funktion, die uns dabei die Arbeit abnimmt. Und unter so wenig Aufwand programmieren wir ein 244 Byte langes Mem-Anzeige-Tool.

```
* AvailMem Version 4    10.3.92 244 Bytes

*************   VARIABLEN *********************

OldOpenLibrary  =       -408
CloseLibrary    =       -414
AllocMem        =       -198
FreeMem         =       -210

AvailMem        =       -216
RawDoFmt        =       -522

Output  =       -60
Write   =       -48
```

```
*************   Hole Speicher ***************

        move.l  4.w,a6
        moveq   #2,d1   ; (oder 3) Chip
        jsr     AvailMem(a6)
        move.l  d0,ChipFree

        moveq   #4,d1   ; (oder 5) Fast
        jsr     AvailMem(a6)
        move.l  d0,FastFree

        moveq   #0,d1   ; (oder 1) Total
        jsr     AvailMem(a6)
        move.l  d0,TotalFree

*************   Ausgabestring erstellen*******

        moveq   #120,d0
        moveq   #0,d1
        jsr     AllocMem(a6)    ; Dafür Speicher
                        ; anfordern
        move.l  d0,a3   ; Zeiger a3 sofort für
                        ; RawDoFmt

        lea     Text(PC),a0
        lea     EinsetzData(PC),a1
        lea     AusgabeRoutine(PC),a2
        jsr     RawDoFmt(a6)    ; a3 wird nicht
                                ; zerstört!

*************   DosLib öffnen ***************

        lea     DosName(PC),a1
        jsr     OldOpenLibrary(a6)
        move.l  d0,a6   ; DosBase gleich in a6

*************   Fenster öffnen ***************

        jsr     Output(a6)
        move.l  d0,d1   ; Fensterhandle in d1
        beq.s   WinFehler       ; Schade!

*************   Schreiben ins Fenster ********

        move.l  a3,d2   ; Strg Anfang von RawDoFmt
        moveq   #-1,d3
WelcheLen
        addq.l  #1,d3
        tst.b   (a3)+   ; StrgLen in d3 für Write
        bne.s   WelcheLen
        jsr     Write(a6)

*************   Dos-Library schließen ********

WinFehler
        move.l  a6,a1
        move.l  4.w,a6
        jsr     CloseLibrary(a6)
        moveq   #120,d0
        move.l  d2,a1   ; Noch von Write in d2
        jmp     FreeMem(a6)

*************   Ausgaberoutine ***************

AusgabeRoutine
        move.b  d0,(a3)+
        rts

*************   Daten ************************

DosName dc.b    "dos.library",0

Text    dc.b    10," Der verfügbare Speicher beträgt: ",10,10
        dc.b    "%7ld Bytes Chip Mem",10
        dc.b    "%7ld Bytes Fast Mem",10
        dc.b    "%7ld Bytes Total Mem",10,10
        dc.b    0

        cnop    0,2

EinsetzData
ChipFree        ds.l    1
FastFree        ds.l    1
TotalFree       ds.l    1
```

Zuerst einmal holen wir uns über die Exec-Funktion AvailMem() den verfügbaren Speicher. Drei Speicher-Kategorien werden von uns abgefragt. Die Spezialisierung auf eine bestimmte Speicherart war ja durch Festlegung von D2 möglich. Den freien Chip-RAM (durch 2 oder 3) schreiben wir in ChipFree, FastRAM (4 oder 5) in FastFree und den gesamten Speicher (0 oder 1) in TotalFree.

Im nächsten Schritt muss der Ausgabestring erstellt werden. Auch bei der maximalen Speicherbelegung wird die Länge des effektive Textes 120 Zeichen nicht überschreiten. Wir besorgen also 120 Bytes, und mit dem Zeiger auf den freien Speicherblock in D0 werden wir durch bewegen nach A3 diesen gleich weiterverwenden. Die Funktion RawDoFmt() generiert mit dem Formatstring Text und den Daten einen String. Diesen geben wir wie gewohnt über den Output-Handle aus. Zum Schluss dürfen wir nicht vergessen, den belegten Speicher wieder freizugeben. (Sonst fehlt er uns nachher nach 4000 Aufrufen!)

Die Grafische Seite des Amigas

Als der Amiga im Jahre 1987 der Öffentlichkeit vorgestellt wurde, war die Grafik im wahrsten Sinne revolutionär. Hohe Grafikauflösung, große Anzahl Farben, einfach Wahnsinn. Für frühere Verhältnisse einfach umwerfend, kannte man nicht die Grafik von Amiga Vorgänger, den C-64. Das sich das Feld heute gewandelt hat, liegt in der normalen Entwicklung der Computerperipherie. Die Tüten (ähm, Verzeihung: DOSen) mit den Super-VGA Karten liegen voll im Trend nach mehr Farben und höherer Auflösung. Der Amiga konnte sich nicht lange auf den Vorsprung ausruhen, schnell wurde von vielen mehr Leistung verlangt. Leider erfüllt Commodore die Anwenderwünsche nur unzureichend, und zwangsläufig wanderten viele von der einst umworbenen Maschine ab. Mit der Einführung der neuen Grafikmodi unter ECS und besonders unter AA und AAA will der Amiga wieder zurück in die Hobby-Grafik Studios.

Grafik als Darstellungsart

Der Amiga ist schon immer ein reiner Grafikcomputer gewesen. Doch warum Grafikcomputer? Ist Grafik denn nicht eine Eigenart, die jeder Rechner beherrscht? Nein, dies noch lange nicht. In früheren Zeiten war Grafik eine Seltenheit, Textschirme waren da die Regel. Dies lag am unzureichenden Speicherplatz der damaligen Systeme. Eine Grafikseite kostet viel Platz, und dieser Platz war wertvoll, zu wertvoll, um ihn zu "verschwenden". Eine Textseite bestand nur als Textzeichen, z. B. in der Größe von 80 * 25. Wenn für jeden Buchstaben ein Byte einkalkuliert würde, so wäre der notwendige Speicherplatz lediglich 80 * 25 = 2000 Bytes groß, lächerlich für heutige Verhältnisse. Eine entsprechende Grafikseite mit einer Buchstabenbreite von 8*16 Pixel nimmt dann schon 80 * 26 * 16 = 16 * 2000 = 32 kb ein. Nicht umsonst weichte man wenn möglich auf Textseiten aus. Doch es gibt noch einen anderen Grund, der nicht zu verachten ist. Arbeitet der Rechner nur mit Grafikseiten, dann muss das System schnell genug sein, um die Zeichen ausreichend zügig darzustellen. Bei Text-Seiten übernimmt die Hardware die Zeichendarstellung, ein Byte, ein Zeichen, bei Grafikseiten, ein Byte gerade einmal ein Sechzehntel. Wenn man Vergleiche zum C-64 zieht, so muss man sagen, dass dort eine gelungene Kombination zu finden ist. Auf der einen Seite der Textbildschirm für schnelle Ausgaben, und auf der anderen der Grafikschirm für Spiele und hochauflösende Anwendungen. Benutzer des GEOS werden es merken, die Geschwindigkeit ist sehr gering.

Mit dem Amiga wurde ein Computer auf den Markt gebracht, der schnell genug war, die anfallenden Grafikdaten zu verarbeiten. So hilft der Blitter mit, Text auf dem Schirm darzustellen, überhaupt ist der Blitter wesentlich mitverantwortlich für das gelingen des gesamten Systems.

Die Grafikbefehle der graphics.library

Wie könnte es auch anders sein, die Grafikfunktionen sind wieder in einer Library organisiert, die graphics.library. Sie ist natürlich intern, und muss nicht nachgeladen werden. Dabei ist sie sehr groß, eine Menge Funktionen werden unterstützt. Es ist möglich, dass einige Programmierer Funktionen vermissen werden, doch zur Diskussion werde ich an anderer Stelle noch kommen.

Wie die Intuition- oder Exec-Lib kann sie in Bereiche unterteilt werden, die folgend aufgelistet sind. Im Verlauf dieses Kapitels werden wir fast ausschließlich Funktionen von Punkt eins kennenlernen.

- Zeichenoperationen im Rastport
- Area-Funktionen im RastPort
- Zeichensatz-Funktionen
- GEL s und Sprites
- Initialisierung und Bearbeiten von Rastern
- Die Blitterbefehle
- Die Copperbefehle
- Die Color-Map-Befehle
- Layers

Konzentrieren wir uns auf die Zeichenoperationen im Rastport. Doch, was ist eigentlich ein RastPort? Dieser Frage gehen wir in auch in diesem Kapitel, jetzt soll es erst einmal reichen, dass der RastPort eine Datenstruktur ist, die Einträge wie Farbstift oder Zeiger auf die Bitplanes enthält. Wer sich noch erinnert, wir haben den RastPort auch schon mal verwendet, als wir nämlich mit Intuition Grafikfunktionen ausführten. Wir können jetzt die Grafikfunktionen aus Intuition erweitert denken, nur, das die Routinen jetzt in der graphics.library zu finden sind. Der Begriff RastPort ist also gar nicht mehr so neu.

Jeder der Grafikausgabefunktionen muss ja wissen, wohin die Grafik kommen soll, es ist nicht so, dass wir einfach wild auch den Schirm malen können, es gibt ja noch so was wie Clipping. Also übergeben wir den Funktionen alle im A1-Reg. ein Zeiger auf den RastPort, der das Fenster oder das Window kennzeichnet. Er ist bei einem Fenster der Window- Struktur zu entnehmen. Das sieht dann etwa so aus.

```
RastPort = 84
...
jsr     OpenScreen(a6)
move.l  d0,a0
lea     RastPort(a0),a0
move.l  a0,RPort
...
```

Äquivalent sieht die Sache bei einem Window aus. Ist es geöffnet, so liegt der RastPort (leider) an einer anderen Stelle. Die Konstante RastPort ist dann mit 50 anzugeben. Da die Variable RastPort nicht zwei Werte gleichzeitig annehmen kann, macht es Sinn, die Kürzel mit anzugeben, also:

```
sc_RastPort = 84 ; für den Screen
wd_RastPort = 50 ; für das Fenster
```

Kommt nun eine der Grafikfunktionen von graphics.library, so schreiben wir einfach

```
move.l RPort,a1
```

und die Parameter können in den Registern übergeben werden, und die Operationen ausgeführt. Ach, das hätten wir ja fast vergessen, welche Funktionen gibt es denn da? Also, die normalen, durchgängigen sind:

SetAPen(RastPort,ColorPen)

Alle Grafikroutinen nutzen eine bestimmte Farbe, wenn sie die Zeichenoperationen durchführen. Mit der Funktion `SetAPen()` kann der Vordergrundstift verändert werden, und dementsprechend auch die Ausgabe farbiger gestaltet werden. Die Vordergrundfarbe heißt deswegen Vordergrundfarbe, da mit diesem Stift z. B. die Fensterumrahmungen gezeichnet werden. Der Farbstift ist eine Farbnummer, die ähnlich eines Farbkastens zu sehen ist. Der Farbkasten hat z. B. 10 Farben, rot, blau, usw. Der Maler kann jetzt den Pinsel mit einer der Farben färben, was er aber nicht kann, ist die Farbe von Rot nach Grün ändern. Er kann lediglich die Auswahl ändern. Die Anzahl der Farben hängt vom View ab, mehr sagt uns aber der Begriff Srceen. Der Workbench Screen hat oft 4 Farben, die manuell durch das Preferences Programm geändert werden können. Wird auf dem WB-Screen ein Window geöffnet, so stehen auch nur vier Stifte zur Verfügung. Farbe Null ist der Hintergrund.

Obwohl die maximale Anzahl der möglichen Farbregister 32 ist, erlaubt die Funktion größere Werte, denn bei den Grafikmodi `EXTRA_HALFBRITE` oder `HAM` sind es ja auch 64 bzw. 4096 Farben. Die Farben werden also nicht immer direkt ins entsprechende Farbregister der Hardware geschrieben, sondern u.U. anders bewertet.

SetBPen(RastPort,ColorPen)

Um beispielsweise zweifarbige Muster zu erzeugen, gibt es neben dem Farbstift A auch noch Farbstift B. Dieser wird entsprechend mit SetBPen() gesetzt.

SetDrMode(RastPort,Mode)

Um den Betriebssystem mitzuteilen, wie die Punkte oder Objekte gezeichnet werden, existieren verschiedene Konstanten, die den Zeichenmodus festlegen:

```
RP_JAM1       = 0
RP_JAM2       = 1
RP_COMPLEMENT = 2
RP_INVERSVID  = 4
```

JAM1 ist normalerweise aktiviert. Ein Linie oder ein Grafikelement wird mit der Stiftnummer APen gezeichnet. Mit JAM2 wird zweifarbig gezeichnet. Bei einem Muster werden die gesetzten Bits mit dem APen und die ungesetzten mit dem BPen umgesetzt. Beispiel: Das Bitfeld 0101010101010101 definiert die Grafikausgabe so, das abwechselnd PenB und PenA genutzt werden. Ist der Mode INVERESID, so werden alle Ausgaben invertiert, daraus folgt: zweimaliges zeichnen löscht die Operation.

Move(RastPort,x,y)(A1,D0,D1)

Die meisten Grafikfunktionen arbeiten mit einem Grafikcursor. Dieser wird mit dem Befehl `Move()` in einem RastPort an die angegebene Position gesetzt. Um die Routine schneller abarbeiten zu können, prüft das Betriebssystem die Koordinaten nicht auf den richtigen Wertebereich. Daher sollte der Programmierer darauf besonders achten.

Draw(RastPort,x,y)(A1,D0,D1)

Die Funktion `Draw()` zeichnet von der mit `Move()` gesetzten Cursor-Position bis zum Punkt x,y eine Linie. Die Koordinaten von x,y werden automatisch zur neuen Grafikcursorposition gemacht. Daher ist es ziemlich einfach, ein Polygon zu zeichnen. Man beginnt mit einer Koordinate, und zeichnet dann immer mit Draw() die nächste Linie.

PolyDraw(RastPort,Anzahl,PolyTable)(A1,D0,A0)

Sollte ein Feld mit Koordinaten zur Verfügung stehen, ist es mühselig, jede der Werte auszulesen, und mit `Draw()` von Linie zu Linie zu kommen. Daher wurde die Funktion `PolyDraw()` mit in das Betriebssystem eingebunden, denn mit ihr ist es einfach möglich, ein Feld mit Paaren zeichnen zu lassen. Man stelle sich nur vor, einen Bildschirmrahmen zu zeichnen, immerwieder mit `Draw()`-Aufrufen. Doch wie wird die Zeichnung erstellt? Wesentlich in die Arbeit eingebunden ist der Wert im `D0`-Reg., die Anzahl der Punkte, die durch die Funktion verbunden werden sollen. `PolyTable` ist ein Zeiger auf ein Array der Form X-Koordinate, Y-Koordinate. Die Wertepaare, also die Polynompunkte, liegen als Word vor.

DrawEllipse(RastPort,cx,cy,a,b)(A1,A0-D3)

Mit dieser Funktion erstellt das Betriebssystem auf dem aktuellen RastPort eine Ellipse. Die Mittelpunktkoordinaten sind cx und cy. Die Radien sind durch a und b gegeben. Wer nun meint, man bekäme einen Kreis, wenn a=b ist, der täuscht sich gewaltig. Der Kreis, der einer sein sollte, wird als Ellipse gezeichnet, denn die Form des Pixels ist nicht quadratisch, sondern vielmehr im Verhältnis 4:3 oder sogar noch anders. Bei unterschiedlichen Auflösungen, 320 * 256, 640 * 256, 1280 * 256 müssen also individuelle Werte her, damit ein Kreis ein Kreis wird. Die in C vordefinierte Routine

```
#define DrawCircle(rp,cx,cy,r) DrawEllipse(rp,cx,cy,r,r);
```

bringt daher nicht immer was.

Leider ist die Funktion `DrawEllipse()` nicht sehr schnell. Deshalb sollte man zu eigenen Funktion greifen, wenn es auch Geschwindigkeit ankommen sollte. Zudem ist diese Routine in der Handhabung sehr unflexibel. Wer beispielsweise einen Ellipsenausschnitt für ein Tortendiagramm zeichnen möchte, der kann gleich anfangen selbst zu programmieren.

RectFill(RastPort,x1,y1,x2,y2)

Das Amiga OS bietet zwar keine Funktion zum Zeichnen von Rechtecken (sehen wir von den Bordern einmal ab), wohl aber eine

zum Zeichnen eines gefüllten Rechtecks. Die Koordinaten x1,y2 bestimmen die linke obere Ecke, x2,y2 die rechte untere. Es ist wichtig die Tatsache zu Betonen, das x1 < x2 und y1<y2 sein muss, denn andernfalls kommt ein Guru, das geht Ruck Zuck. Die Koordinaten müssen also verglichen, und u.U. vertauscht werden, wenn eine allgemeine Funktion programmiert wird.

Soll ein Rechteck mit zweifarbigem Muster erzeugt werden, so ist der APen und der BPen zu nutzen. Als Modus muss der eben erwähnte JAM2-Mode aktiviert werden. Das Muster muss per Makro übergeben werden, da hierzu keine OS-Funktion zur Verfügung steht. Wie dies gemacht wird, erfährt der Leser in 13.3.1

WritePixel(RastPort,x,y)(A1,Do,D1)

Ein Punkt mit den Koordinaten x,y wird gezeichnet. Es wird ein Ruckgaberegister gesetzt. Ist `D0=-1`, so waren die Wertepaare nicht im RastPort-Bereich. Interessant ist, das hier nicht der Cursor eingesetzt wird, sondern dass direkt der Punkt auf den Schirm kommt, die Funktion hätte ja auch `WritePixel(RastPort)` heißen können.

Möchte man einen Punkt löschen, so muss vorher die aktuelle Zeichenfarbe gesichert werden, `SetAPen(0)` aufgerufen werden, um den Stift auf die Hintergrundfarbe zu legen, dann der Punkt gesetzt (der ja dann gelöscht wird) und letztendlich der Stift wieder restauriert werden.

FarbStift = ReadPixel(RastPort,x,y)(A1,Do,D1)

Um die Farbe eines Punktes heraus zu bekommen muss die `WritePixel()` Funktion quasi umgekehrt werden. Dazu dient `ReadPixel()`, der die Farbe des unter x,y, liegenden Punktes herausfindet. Ist der Übergabeparameter negativ, also -1, so konnte keine Farbe ermittelt werden, die die Koordinaten lagen außerhalb des RastPorts, eine Farbe kann nicht übergeben werden. Die Farbe ist nicht in ihre Farb-Komponenten Rot/Grün/Blau aufgeteilt, sondern es wird die Stiftnummer übergeben.

Text(RastPort,Strg,Count)(A1,Ao,Do)

Auch in der `graphics.libarary` findet sich eine Funktion zum Ausgeben von Zeichenketten. Mit `Text()` kann der String `Strg` in den RastPort an der mit `Move()` festgelegten Position geschrieben werden.

Lenght = TextLenght(RastPort,Strg,Count)

Diese Funktion ist eine Zugabe, denn sie hat direkt nichts mit dem Zeichen auf dem RastPort zu tun. Mit ihrer Hilfe wird lediglich die Ausdehnung einer Zeichenkette ermittelt. Damit ist nicht etwa die Anzahl Zeichen gemeint, sondern die Anzahl der Pixel, die der Text in der Horizontalen einnimmt. Die Angabe des RastPort ist notwendig, um den aktuellen Font in die Berechnung einfließen zu lassen. Andernfalls wüsste das OS ja nicht, welcher Zeichensatz verwendet wird, da immer mehrere Zeichensätze im System sein können. Im RastPort allerdings kann immer nur ein Zeichensatz aktuell eingestellt sein.

ClearEOL(RastPort)

Mit dieser Funktion wird ab der aktuellen Grafikcursor-Position eine Zeile gelöscht. Die Zeile kann natürlich, zeichensatzbedingt, verschiedene Höhen haben. Der Zeichensatz, der im RastPort eingetragen ist, gibt dabei die Höhe vor.

Anwendung findet die Funktion z. B. bei Editoren wo eine Zeile gelöscht werden soll.

ClearScreen(RastPort)

Durch die `ClearScreen()` Funktion wird ab der augenblicklichen Grafikcursor-Position der Bildschirm nach unten hin gelöscht. Dies ist insbesondere bei Editor-Scroll-Funktionen sinnvoll, wenn alle Zeilen eine Position nach oben geschoben werden, und die unterste gelöscht werden muss. Wenn natürlich der gesamte Bildschirm frei gemacht werden soll, ist es möglich, mit `Move(0,0)` den Stift in der extremsten Position anzusetzen, und dann mit `ClearScreen()` den gesamten Bildschirm löschen zu lassen. Ein Macro zum Löschen des RastPorts-Schirm hätte folgendes Aussehen:

```
CLS MACRO
  movem  d0-a5,-(SP)
  move.l a1,a3
  move   cp_x(A1),d4     ; 1. cp_x/y Cursorposition aus
  move   cp_y(A1),d5     ; RastPort lesen
  moveq  #0,d0
  moveq  #0,d1
  jsr    Move(a6)        ; 2. Koordinaten auf Null setzen
  move.l a3,a1
  jsr    ClearScreen(a6) ; 3. löschen
  move   d4,d0
  move   d5,d1
  move.l a3,a1
  jsr    Move(a6)        ; 4. alte Werte setzen
  movem  (SP)+,d0-a5
```

SetRast(RastPort,FarbStift)

Warum ein aufwendiges Makro benutzen, wenn es auch einfacher geht? Mit der Funktion `SetRast()` wird ein RastPort mit einer Farbe eingefärbt. Während bei `ClearScreen()` die aktuelle Farbe benutzt wird, muss sie hier manuell angegeben werden, da die Hintergrundfarbe aber immer Null ist, dürfe das Löschen des Screens kein Problem sein.

Wer sich nun freut, die Routine zum Löschen des Window nutzen zu können, der muss enttäuscht werden, Windows haben zwar einen RastPort, jedoch nicht einen speziellen eigenen, auf einen rechteckigen Bereich beziehenden. Sollte also einmal das Problem kommen, ein Fenster zu löschen, dann kann nicht einer der Befehle (`ClearScreen` oder `SetRast`) benutzt werden. Man muss etwas umständlich auf `RectFill()` zurückgreifen, und ein Rechteck in der Größe des Fensterinhaltes zeichnen.

Jetzt mal eigenes Fenster zeichnen

Nach den gewaltigen Funktionen, ein kleines Beispiel, wie die Routinen genutzt werden können. Die Betonung liegt bei können, ich habe vielfach gefüllte Rechtecke eingesetzt, um nicht vier Linien zeichnen zu müssen, schön ist das auch nicht, aber praktisch.

```
* Gfx1  Version 1       12.5.93 620 Bytes

************** VARIABLEN *********************

OldOpenLibrary  =       -408    ; Exec
CloseLibrary    =       -414
```

```
WaitPort        =       -384

OpenScreen      =       -198    ; Intui
CloseScreen     =       -66

Move            =       -240    ; Gfx
Draw            =       -246
RectFill        =       -306
SetAPen         =       -342
SetRast         =       -234
Text            =       -60
DrawEllipse     =       -180

RastPort        =       84

HEADERHOEHE     =       8
WINTITLE        =       8       ; in eigener Struct

************** HAUPTPROGRAMM *****************

        bsr.s   OpenLibs

        bsr     ÖffneScreen
        move.l  d0,ScrHandle
        beq     OpenScrnFehler

        bsr     Malen
        bsr     Warten
        bsr     SchließeScreen
OpenScrnFehler
        bra     CloseLibs

************** Libs öffnen ******************

OpenLibs        move.l  4.w,a6
        lea     IntuiName,a1
        jsr     OldOpenLibrary(a6)
        move.l  d0,IntuiBase

        lea     GfxName,a1
        jsr     OldOpenLibrary(a6)
        move.l  d0,GfxBase
        rts

************** Screen öffnen *****************

ÖffneScreen
        move.l  IntuiBase,a6
        lea     NewScreen,a0
        jsr     OpenScreen(a6)
        move.l  d0,a0
        lea     RastPort(a0),a0
        move.l  a0,RPort
        rts

************** Malen im Fenster **************

Malen   move.l  GfxBase,a6
        move.l  RPort,a1
        moveq   #0,d0   ; in der Hintergrundfarbe
        jsr     SetRast(a6)

        lea     FensterStruct,a0
        bsr     OpenNewWindow

Loop    move.l  RPort,a1
        move    StiftNr,d0      ; in der Hintergrundfarbe
        jsr     SetAPen(a6)
        addq    #1,StiftNr

        lea     FensterStruct,a0
        movem   (a0),d0-d3
        lsr     d2              ; Mittelpunkt finden
        lsr     d3
        add     d2,d0
        add     d3,d1

        move    StiftNr,d2
        lsl     d2
        add     #30,d3
        move    StiftNr,d3
        move.l  RPort,a1
        jsr     DrawEllipse(a6)

        cmp     #70,StiftNr
```

```
        bne.s   Loop

        rts

StiftNr dc      0       ; Farbstiftnummer

************** neues Fenster erstellen *******

OpenNewWindow
        movem.l d0-a5,-(SP)
        move.l  a0,a4   ; Zeiger auf Daten sichern
        move.l  GfxBase,a6

        moveq   #1,d0   ; schwarz
        move.l  RPort,a1
        jsr     SetAPen(a6)

        movem   (a4),d0-d3      ; Datenreg. mit Koord. füllen
        move.l  RPort,a1
        jsr     RectFill(a6)    ; schwarzer Rahmen

        moveq   #2,d0   ; weiß
        move.l  RPort,a1
        jsr     SetAPen(a6)

        movem   (a4),d0-d3
        addq    #1,d0
        addq    #1,d1
        subq    #1,d2
        subq    #1,d3
        move.l  RPort,a1
        jsr     RectFill(a6)    ; weißer Inhalt

        moveq   #3,d0   ; dunkelgrau
        move.l  RPort,a1
        jsr     SetAPen(a6)

        movem   (a4),d0-d3
        addq    #2,d0
        addq    #2,d1
        subq    #1,d2
        move    d1,d3
        addq    #HEADERHOEHE,d3
        move.l  RPort,a1
        jsr     RectFill(a6)

        moveq   #0,d0   ; hintergrundsgrau
        move.l  RPort,a1
        jsr     SetAPen(a6)

        movem   (a4),d0-d3
        addq    #2,d0
        addq    #2,d1
        subq    #2,d2
        move    d1,d3
        addq    #HEADERHOEHE-1,d3
        move.l  RPort,a1
        jsr     RectFill(a6)

        moveq   #1,d0   ; black
        move.l  RPort,a1
        jsr     SetAPen(a6)

        movem   (a4),d0-d1
        add     #HEADERHOEHE+3,d1
        move    d1,d3
        move.l  RPort,a1
        jsr     Move(a6)
        movem   4(a4),d0        ; x2 lesen
        move    d3,d1
        move.l  RPort,a1
        jsr     Draw(a6)

        movem   (a4),d0-d1
        add     #30,d0  ; in x verschieben
        addq    #8,d1
        move.l  RPort,a1
        jsr     Move(a6)

        move.l  WINTITLE(a4),a0
        moveq   #-1,d0
TxtLen  addq    #1,d0
        tst.b   (a0,d0)
        bne.s   TxtLen
        move.l  RPort,a1
```

```
        jsr     Text(a6)

        movem.l (SP)+,d0-a5
        rts

************** Warten ************************

MausKnopf       =       $bfe001 ; Hardware

Warten  btst    #6,MausKnopf
        bne.s   Warten
        rts

************** Screen schließen **************

SchließeScreen
        move.l  ScrHandle,a0
        move.l  IntuiBase,a6
        jmp     CloseScreen(a6)

************** Libs schließen ****************

CloseLibs
        move.l  4.w,a6
        move.l  GfxBase,a1
        jsr     CloseLibrary(a6)

        move.l  IntuiBase,a1
        jmp     CloseLibrary(a6)

************** Daten *************************

IntuiName       dc.b    "intuition.library",0
IntuiBase       ds.l    1

GfxName dc.b    "graphics.library",0
GfxBase ds.l    1

************** Screen Definitionen ***********

V_HIRES         =       $8000
CUSTOMSCREEN    =       $f

NewScreen
        dc      0,0     ; Linker Rand, Oberer Rand
        dc      640,256 ; Breite, Höhe
        dc      4       ; Anzahl BitPlanes
        dc.b    1,1     ; Pens
        dc      V_HIRES ; Hires Screen
        dc      CUSTOMSCREEN    ; eigen
        dc.l    0       ; Kein Spezieller ZS
        dc.l    0       ; kein Screentitel
        dc.l    0       ; keine Gadgets
        dc.l    0       ; keine eigenen Bitplanes

ScrHandle
        dc.l    0

RPort   dc.l    0       ; RastPort des Screens

************** Eigene Fensterdefinition ******

FensterStruct
        dc      10,10
        dc      400,200
        dc.l    WinTitle

WinTitle
        dc.b    "Der Fensterinhalt",0
        cnop    0,2
```

Das Programm ist eine kleine Grafikanwendung, ein Fenster wird selbstständig aufgebaut. Von Prinzip her macht das AmigaOS es nicht anders. Allerdings kommen hier noch ein paar Sachen hinzu!

Unser Programm öffnet einen Screen, um die Grafikoperationen darüber laufen zu lassen. Es wird über ein Unterprogramm `OpenNewWindow` ein eigenes Fenster gezeichnet. Auf dem lassen wir einen Kreis zeichnen. Natürlich wird dieser nicht geclippt.

Die RastPort-Struktur untersucht

Da ich immer die Strukturen auseinander genommen habe, darf ich den RastPort natürlich nicht auslassen. Die Struktur umfasst $64=100 Bytes.

Der RastPort ist in für grafischen Oberfläche eine wichtige Struktur, die zur Verwaltung der Zeichenebene notwendig ist. Nicht umsonst bekommen unsere beiden Zeichenflächen Window und Screen jeweils ein eigenen RastPort, auf dem die

Grafikoperationen durchgeführt werden.

RastPort:

```
$000     0  Layer (LONG)
$004     4  BitMap (LONG)
$008     8  AreaPtrn (LONG)
$00c    12  TmpRas (LONG)
$010    16  AreaInfo (LONG)
$014    20  GelsInfo (LONG)
$018    24  Mask (BYTE)
$019    25  FgPen (BYTE)
$01a    26  BgPen (BYTE)
$01b    27  AOlPen (BYTE)
$01c    28  DrawMode (BYTE)
$01d    29  AreaPtSz (BYTE)
$01e    30  linpatcnt (BYTE)
$01f    31  dummy (BYTE)
$020    32  Flags (WORD)
$022    34  LinePtrn (WORD)
$024    36  cp_x (WORD)
$026    38  cp_y (WORD)
$028    40  minterms[8] (STRUCT)
$030    48  PenWidth (WORD)
$032    50  PenHeight (WORD)
$034    52  Font (LONG)
$038    56  AlgoStyle (BYTE)
$039    57  TxFlags (BYTE)
$03a    58  TxHeight (WORD)
$03c    60  TxWidth (WORD)
$03e    62  TxBaseline (WORD)
$040    64  TxSpacing (WORD)
$042    66  RP_User (APTR)
$046    70  longreserved[8] (STUCT)
```

Neu seit 1.2

```
$04e    78  wordreserved[14]
$05c    92  reserved[8]
```

Offset 0: Layer

Die Layers (eng. für Schicht) sind noch niedrigere Grafikebenen des Systems. Der Amiga benutzt diese Schichten, um etwa Window-Überlappungen und Clippings in Fenstern zu realisieren. Die Layers sind sehr leistungsfähig, und erweisen sich als die Macher der Fenster. Da ihr Zusammenhang jedoch etwas gehobener ist, möchte ich in diesem Buch lediglich das Wort Layer erwähnen, mehr aber auch nicht. (Geplant war ursprünglich ein kleines Scrolldemo für ein Adventure, doch das habe ich aus Platzgründen fallengelassen.)

Offset 4: BitMap

Um an die rohen Grafikbitplanes zu kommen, um evtl. eine schnelle Zeichenfunktion dadurch zu realisieren, ist es notwendig, die BitMap-Struktur zu benutzen, die auch schon einmal bei den Screens angedeutet wurde.

Die BitMapstruktur, mit einer Länge von 40 ($28) Bytes hat folgendes Gerüsst:

```
$000     0  BytesPerRow
$002     2  Rows
$004     4  Flags
$005     5  Depth
$006     6  pad
$008     8  Planes[8]
```

Die ersten Felder enthalten die Abmessungen des Displays, und bei den weiteren sind lediglich die Zeiger auf die Planes wichtig. Um beispielsweise ein Screendump zu realisieren müsste man nur die Zeiger auslesen, und durch die Breiten- und Höhenberechnung die Größe errechnen. Ein kleines Pseudoprogramm:

```
Größe des Bildes = BytesPerRow * Row
Schleife mit Zähler von 1 bis Depth
  Schreibe auf Speichermedium (Planes[Zähler], Größe des Bildes)
```

Offset 8: AreaPtrn

Neben dem Füllen von Flächen mit verschiedenen Farben kann die Fläche auch mit einem Muster gefüllt werden. Der AreaPtrn-Eintrag ist ein Zeiger auf das Area-Fill-Muster, das beim füllen verwendet wird.

Offset 12: TmpRas

Ein temporäres RastPort, in dem Objekte aufgenommen werden, die z. B. gefüllt werden.

Die Struct wird durch folgende Zeilen definiert:

```
$0000     0  tr_RasPtr (APTR) ; * WORD
$0004     4  tr_Size (LONG)
```

Offset 16: AreaInfo

Eine bisher unbesprochene Gruppe der Grafikbefehle sind die Area-Befehle. Im Eintrag `AreaInfo` sind für die Polynomfunktionen wichtige Informationen abgelegt.

Die `AreaInfo`-Struktur sieht so aus:

```
$0000     0  VctrTbl (LONG)
$0004     4  VctrPtr (LONG)
$0008     8  FlagTbl (LONG)
$000c    12  FlagPtr (LONG)
$0010    16  Count (WORD)
$0012    18  MaxCount (WORD)
$0014    20  FirstX (WORD)
$0016    22  FirstY (WORD)
$0018    24  sizeof(AreaInfo)
```

Offset 20: GelsInfo

GELs sind "Graphics ELements", und erweitern die Grafik des Amigas. Die grafischen Elemente sind die Hardwaresprites, die VSprites (virtuelle Sprites; mehr Sprites, durch Interrupt-Umschaltung) und die BOBs. BOBs wiederum sind größere Sprites, die zwar langsamer und aufwendiger, aber dafür eben größer in den Ausmaßen sind.

Um die GELs zu verwalten, muss eine Struktur angelegt werden, die so aussieht (ich übernehme einfach mal vom INCLUDE-File):

```
STRUCTURE   GelsInfo,0
 BYTE   gi_sprRsrvd             ; flag which sprites to reserve from
 BYTE   gi_Flags
 APTR   gi_gelHead
 APTR   gi_gelTail
 APTR   gi_nextLine
 APTR   gi_lastColor
 APTR   gi_collHandler
 WORD   gi_leftmost
 WORD   gi_rightmost
 WORD   gi_topmost
 WORD   gi_bottommost
 APTR   gi_firstBlissObj
 APTR   gi_lastBlissObj
```

Offset 24: Mask

Mit dieser Variablen lassen sich die Bitplanes, die beschrieben werden können, setzen. Die Schreibmaske ist natürlich als Byte abgelegt, denn ein gesetztes Bit an einer bestimmten Bitposition heißt immer: Zeichnen auf dieser Ebene. Daher ist im Normalfall immer der Wert 256 voreingestellt. Dieser Wert darf einen nicht zu hoch vorkommen, acht Bitplanes sind unter OS 3.0 durchaus möglich, obwohl langsam. Um den VGA-Mode mir 2^8 = 256 Farben zu bekommen, muss man schon mal zu den Bitplanes greifen, und da ist der Wert 256 angebracht.

Offset 25-27: FgPen, BgPen,AOlPen

In diesen drei Bits sind die Farbregister für die Hintergrund-, Vordergrund- und Umrandungsfarbe gesetzt. Am interessantesten ist die Umrandungsfarben. Wenn sie gesetzt ist, werden die Objekte, die z. B. `RectFill()` gezeichnet werden, mit Umrandet. Das heißt für ein Rechteck, es werden um ihn vier Linien gezogen.

Offset 28: DrawMode

Die Zeichenmodi hatte ich schon bei `SetDrMode()` vorgestellt. Sie sind hier noch einmal zusammengestellt.

```
RP_JAM1       = 0
RP_JAM2       = 1
RP_COMPLEMENT = 2
RP_INVERSVID  = 4
```

Voreingestellt ist der Mode JAM2, d.h. es wird mit der Vordergrundfarbe gezeichnet.

Offset 29: AreaPtSz

Wenn das System mit Mustern arbeitet, d. h. `AreaPtrn` ist mit einem Zeiger auf einem Muster versehen, muss die Höhe dieses Patterns angegeben werden. Die Höhe kann fast unbegrenzt sein, muss sich aber in Zweierpotenz-Schritten bewegen. Möglich sind also Werte von 1, 2, 4, 8, 16, ... Die Breite darf, wie vermutlich bekannt, nur 16 Pixel breit sein, bevor sie sich automatisch wiederholt. Aus diesem Grunde passen auch gefüllte Muster, obwohl immer an einem anderen Punkt angesetzt, so schon zusammen.

Offset 30: linpatcnt, dummy

Für uns nicht wichtig, da sie intern von Intui benutuzt werden. Dabei soll es ja auch bleiben, oder?

Offset 32: Flags

In den RastPort sind eigene Konstanten eingefügt worden, die da lauten:

```
   BITDEF   RP,FRST_DOT,0     ; draw the first dot of this line
   BITDEF   RP,ONE_DOT,1      ; use one dot mode for
                              ; drawing lines

   BITDEF   RP,DBUFFER,2      ; flag set when RastPorts are
                              ;double-buffered (only used for bobs)
   BITDEF   RP,AREAOUTLINE,3  ; used by areafiller
   BITDEF   RP,NOCROSSFILL,5  ; used by areafiller
```

```
ONE_DOTn  = 1
ONE_DOT   = 2 ; 1<<ONE_DOTn
FRST_DOTn = 0
FRST_DOT  = 1 ; 1<<FRST_DOTn
```

Offset 34: LinePtrn

Neben den Flächen lassen sich auf Linien mit einem vordefinierten Muster zeichnen. Wie bei den Flächen ist die Breite auf 16 Punkte begrenzt, was bedeutet, die Anwendung ist leider etwas eingeschränkt, aber was soll s. Meistens, will man eh nur Punkt gesetzt Punkt frei haben, mit der Funktion das Drehen von Objekten zu ermöglichen, wird wohl kaum einer im Kopf haben.

Offset 36, 38: cp_x, cp_y

In den beiden Feldern sind die wichtigen Grafikcursor-Positionen gespeichert, die mit dem `Move()`-Befehl gesetzt werden, und z. B. von der `DrawLine()`-Funktion genutzt wird.

Offset 40: minterms[8]

Auch diese acht Byte-Einträge sind für uns unwichtig, denn sie sind Intuition intern.

Offset 48, 50: PenWidth, PenHeight

In den beiden Words ist die Cursorbreite und Cursorhöhe gespeichert.

Offset 52: Font

Dieser Zeiger ist eine Referenz auf den gerade aktuellen Zeichensatz, der in dem RastPort aktiv ist. Wie dieser umgestellt werden kann werden wir im nächsten Kapitel erfahren.

Offset 56: AlgoStyle

Neben den grundsätzlichen schrifttypischen Erscheinungsbild, kann der Amiga die Zeichen in einigen verschiedenen Varianten ausgeben, die durch die Konstanten:

```
Normal = 0
Unterstrichen = 1
Fett = 2
Kursiv = 4
```

angegeben werden. Der Strukturoffset heißt nicht umsonst AlgoStyle, der Zeichensatz ist nicht neu, nur durch einen Algorithmus verändert worden. Im nächsten Kapitel werden wir mehr über Zeichensätze erfahren. Wir werden auch die Anzahl der künstlichen Schrifttypen erhöhen, indem wir einen Text outlinen.

Offset 57: TxFlags

In diesem Byte finden wir eine textspezifische Konstante,

```
TXSCALE = 1
```

Sie wird ebenfalls intern von Intuition gebraucht.

Offset 58: TxHeight

Um beim Vorschub in die nächste Zeile die Position errechnen zu können wird die Texthöhe benötigt. Diese wird in der Struktur an Offset 58 abgelegt. Der Zeichensatz, dessen Höhe eingetragen wird, ist immer der aktuelle, der auch in Font gesicherte.

Offset 60: TxWidth

In TxWidth findet der Anwender die durchschnittliche Breite jedes Zeichens. Verwendet werden meist Zeichensätze mit konstanten Textbreite, wie 8 Pixel. Da aber der Trend hin zu proportionalen Zeichensätzen geht, die Zeichen haben verschiedene Breiten, i ist schmaler als ein m, kann in diesem Eintrag nur ein Durchschnittswert eingetragen werden.

Offset 62: TxBaseline

Jeder Text wird mit verschiedenen Größen auftauchen. Unter anderem ist für die Ausgabe die Höhe des Textzeichens und die Höhe des Textzeichens ohne die Unterlänge wichtig. Buchstaben mit Unterlängen sind z. b. "g", "j", "y". Die Texthöhe ohne Unterzeichen des "g" ist also die Höhe eines "o", obwohl die reine Texthöhe natürlich unterschiedlich ist. Wofür braucht man das? Nun, denken wir uns eine Linie, auf der ein Text stehen soll. Man zieht nun von der gewünschten Position (z. B. 30) `TxBaseline` (z. B. 6) ab, und hat dann die Koordinate (30 - 6 = 24), an der die Zeichen ausgegeben werden müssen, damit sie auf der Linie sitzen.

Offset 64: TxSpacing

Wir kennen aus dem Schreibmaschinenunterricht bestimmt das Sperren von Texten zur Hervorhebung. OK, damals konnte eine Schreibmaschine noch nicht fett drucken, da musste man Besonderheiten eben so hervorheben. Mit der Variblen `TxSpacing` lässt sich auf bei der Amiga Textausgabe der Abstand zwischen den Buchstaben ändern. Der Wert ist normalerweise auf Null gesetzt, was bedeutet, es werden Null Leerpixel zwischen den Lettern gesetzt. `TxSpacing = 8` würde acht freie Pixel von Buchstabe zu Buchstabe lassen, was unser normal gesperrtes wäre.

Offset 66: RP_User

Um den Benutzer wieder in die Struktur eingreifen zu lassen, ist ein freies Feld mit einem Long beschreibbar.

Offset 70-92: longreserved[8], wordreserved[14], reserved[8]

Soll uns nicht interessieren.

RastPort-Änderungen und ihre Konsequenzen

In der RastPort Struktur tauchen so einige Variablen auf, die durch Befehle in der graphics.library direkt geändert werden. So z. B. werden die Farben des Vordergrundstiftes oder des Hintergrundstiftes direkt in `RastPort.FgPen` (`ForGroundPEN`) und `RastPort.BgPen` (`BachGroundPEN`) gesetzt. Es existieren aber noch viel mehr Struktureinträge, die nicht durch Funktionen geändert werden, sondern erst durch manuelles Setzen ihre Funktion entfalten. So z. B. der Byte-Eintrag AOLPen, mit dem eine Umrandungsfarbe gesetzt wird. Man kann sich jetzt einige Makros zusammenstellen, die wichtige Funktionsaufrufe emulieren. Ich möchte die Makros, die im Regelfall bei C-Compilern beiliegen, auch für Assemblerprogrammierer eröffnen. Sie sind in der Datei

graphics/gfxmacro.h zu finden.
Umrandung einschalten und Stiftnummer setzen:

```
#define SetOPen(w,c) {(w)->AOlPen = c;(w)->Flags |= AREAOUTLINE;}
```

Festlegen des Linienmusters:

```
#define SetDrPt(w,p) f(w)->LinePtrn=p;w)->Flagsj=FRST_DOT;(w)->linpatcnt=15;
```

Festsetzen, welche Bitplanes beschrieben werden können:

```
#define SetWrMsk(w,m) f(w)->Mask = m;
```

Festlegen des Füllmusters:

```
#define SetAfPt(w,p,n) {(w)->AreaPtrn = p;(w)->AreaPtSz = n;}
```

Umrandung ausschalten:

```
#define BNDRYOFF(w) {(w)->Flags &= ~AREAOUTLINE;}
```

Kleines Malprogramm groß ausbaufähig

Nachdem nun ein kleines Grafikdemo die wichtigsten Funktionen klärte, und auch die RastPort Struktur ausreichend beleuchtet wurde, wir ein weiteres Programm die Grafikzeichenbefehle abrunden. Ich stelle folgend ein kleines Zeichenprogramm vor, das die wichtigsten Funktionen wie Linie, Kreis zeichnen und Punkt setzen beherrscht. Der besondere Clou ist, dass der Programm eine Protokolldatei erstellt, über alles, was gezeichnet wurde. Das Zeichnen der Punkt-Linien ist aber noch fehlerhaft, und könnte verbessert werden.

```
* Malen Version 1       30.3.92 2156 Bytes

************** VARIABLEN *********************

OldOpenLibrary  =       -408    ; Exec
CloseLibrary    =       -414
WaitPort        =       -384
GetMsg          =       -372
ReplyMsg        =       -378
RawDoFmt        =       -522

OpenWindow      =       -204    ; Intui
CloseWindow     =       -72
SetPointer      =       -270
ClearPointer    =       -60
AddGaget        =       -42
RefreshWindowFrame =    -456
ModifyIDCMP     =       -150

Move            =       -240    ; Gfx
DrawEllipse     =       -180
Draw            =       -246
WritePixel      =       -324
SetDrMd         =       -354
RectFill        =       -306

Open            =       -30             ; Dos
Write           =       -48
Close           =       -36

************** KONSTANTEN ********************

MODE_NEWFILE    =       1006    ; Dos
LF              =       10

MausKnopf       =       $bfe001 ; Hardware

Class           =       $14     ; Intui-Message
IAddress        =       $1c

MouseX          =       14      ; Window Struktur
MouseY          =       12
RastPort        =       50
UserPort        =       86

GadgetID        =       $26     ; Gadgetstruct

GADGETDOWN      =       $20     ; NewWindow IDCMP-Flags
```

```
CLOSEWINDOW     =       $200
MOUSEBUTTONS    =       8

WINDOWSIZING    =       1       ; nw.Flags
WINDOWDRAG      =       2
WINDOWDEPTH     =       4
WINDOWCLOSE     =       8
;GIMMEZEROZERO  =       $400
ACTIVATE        =       $1000

RELVERIFY       =       1       ; Gadget.Flags
GADGIMAGE       =       4
GRELRIGHT       =       $10

GADGIMMEDIATE   =       2       ; Activation

BOOLGADGET      =       1       ; Gadgettyp
;GZZGADGET      =       $2000

JAM1    =       0
COMPLEMENT      =       3       ; Drawmode

ADDGADGET       MACRO
         move.l WinHandle,a0
         lea    \1,a1
         moveq  #0,d0
         jsr    AddGaget(a6)
        ENDM

************** HAUPTPROGRAMM *****************

        bsr.s   Inti

Loop    bsr     WaitForReakt

        cmp.l   #CLOSEWINDOW,d0
        beq     EndOfDemo

        cmp.l   #GADGETDOWN,d0
        beq     GadgetNrAuswerten

        cmp.l   #MOUSEBUTTONS,d0
        beq     Malen

        bra.s   Loop

************** Init so alles mögliche *******

Inti    move.l  4.w,a6
        lea     DosName,a1
        jsr     OldOpenLibrary(a6)
        move.l  d0,DosBase

        move.l  d0,a6
        move.l  #DateiName,d1
        move.l  #MODE_NEWFILE,d2
        jsr     Open(a6)        ; die Protokolldatei
        move.l  d0,FileHandle

        move.l  4.w,a6
        lea     IntuiName,a1
        jsr     OldOpenLibrary(a6)
        move.l  d0,IntuiBase

        lea     GfxName,a1
        jsr     OldOpenLibrary(a6)
        move.l  d0,GfxBase

        move.l  IntuiBase,a6

        lea     NewWindow,a0
        jsr     OpenWindow(a6)
        move.l  d0,WinHandle
        move.l  d0,a0
        move.l  RastPort(a0),RPort

        bsr.s   MausEinfügen

        ADDGADGET       gg1
        ADDGADGET       gg2
        ADDGADGET       gg3
        ADDGADGET       gg4
        ADDGADGET       gg5

        move.l  WinHandle,a0    ; Gadets ins Window
```

```
        jmp     RefreshWindowFrame(a6) ; einbinden

************** Mauszeiger ändern *************

MausEinfügen
        lea     SpriteDatenAnf,a1
        moveq   #11,d0  ; Höhe
        moveq   #8,d1
        moveq   #-8,d2
        moveq   #-5,d3
        jmp     SetPointer(a6)  ; Handle war in d0

NormaleMaus
        move.l  WinHandle,a0
        jmp     ClearPointer(a6)

************** Warten auf eine Aktion ********

WaitForReakt
        move.l  IntuiBase,a6
        move    #CLOSEWINDOW|GADGETDOWN|MOUSEBUTTONS,d0
        move.l  WinHandle,a0
        jsr     ModifyIDCMP(a6)

        move.l  4.w,a6
        move.l  WinHandle,a0
        move.l  UserPort(a0),a3
        move.l  a3,a0
        jsr     WaitPort(a6)
        move.l  d0,a4   ; Zeiger von IntuiMessage
        move.l  a3,a0
        jsr     GetMsg(a6)
        move.l  Class(a4),d0    ; IDCMP Code aus
Messagestruct
        rts

************** Welches Gadget wurde angeklickt?

GadgetNrAuswerten
        move.l  IAddress(a4),a0 ; auch aus Messagestruct
        move    GadgetID(a0),GadgetNr
        bra     Loop

GadgetNr        dc      0

************** MACROS zum Zeichen ************

Draw_Normal
        MACRO
         moveq  #JAM1,d0
         move.l RPort,a1
         jsr    SetDrMd(a6)
        ENDM

Draw_Complement
        MACRO
         moveq  #COMPLEMENT,d0
         move.l RPort,a1
         jsr    SetDrMd(a6)
        ENDM

Hol_Koordinaten
        MACRO
         move.l WinHandle,a0
         move   MouseX(a0),d0
         move   MouseY(a0),d1
        ENDM

************** Mausklick und Aktion **********

Malen   move.l  IntuiBase,a6
        move    #CLOSEWINDOW|GADGETDOWN,d0 ; nicht mehr
        move.l  WinHandle,a0    ; auf Mousebuttons hören
        jsr     ModifyIDCMP(a6)

        move.l  GfxBase,a6

        move    GadgetNr,d0
        lsl     #2,d0   ; mal vier
        move.l  JMPTab(PC,d0),a0
        jmp     (a0)    ; Funktion ausführen

JMPTab  dc.l    Linie
        dc.l    Kreis
        dc.l    Punkt
```

```
        dc.l    Rechteck
        dc.l    Box

************** Linie ************************

Move_Start
        MACRO
         move   x1,d0
         move   y1,d1
         move.l RPort,a1
         jsr    Move(a6)        ; Anfangspunkt
        ENDM

Linie   Hol_Koordinaten
        move    d0,x1
        move    d1,y1

        Draw_Complement

LineLoop
        Move_Start

        Hol_Koordinaten
        move    d0,X2
        move    d1,Y2

        move.l  RPort,a1
        jsr     Draw(a6)

        Move_Start

        move    X2,d0
        move    Y2,d1

        move.l  RPort,a1
        jsr     Draw(a6)

        btst    #6,MausKnopf
        beq.s   LineLoop

        Draw_Normal

        Move_Start

        move    X2,d0
        move    Y2,d1

        move.l  RPort,a1
        jsr     Draw(a6)

        move    x1,d0
        move    y1,d1
        move    X2,d2
        move    Y2,d3
        lea     LineTxt,a0
        bsr     Printf  ; Text ausgeben

        bra     Loop

LineTxt dc.b    "Line %d,%d,%d,%d",LF,0

************** Kreis ************************

Kreis   Hol_Koordinaten
        move    d0,x1
        move    d1,y1

        Draw_Complement

KreisLoop
        move.l  WinHandle,a0
        move    x1,d0
        move    y1,d1
        move    d0,d2
        sub     MouseX(a0),d2
        bpl.s   OkX
        neg     d2
OkX     move    d1,d3
        sub     MouseY(a0),d3
        bpl.s   OkY
        neg     d3
OkY     move    d2,X2
        move    d3,Y2
        move.l  RPort,a1
        jsr     DrawEllipse(a6)
```

```
        move    x1,d0
        move    y1,d1
        move    X2,d2
        move    Y2,d3
        move.l  RPort,a1
        jsr     DrawEllipse(a6)

        btst    #6,MausKnopf
        beq.s   KreisLoop

        Draw_Normal

        move    x1,d0   ; entgültiger Kreis
        move    y1,d1
        move    X2,d2
        move    Y2,d3
        lea     ElliTxt,a0
        bsr     Printf  ; Text ausgeben

        move.l  RPort,a1
        jsr     DrawEllipse(a6)

        bra     Loop

ElliTxt dc.b    "Ellipse %d,%d,%d,%d",LF,0

x1      dc      0
y1      dc      0
X2      dc      0
Y2      dc      0

************** Punkt ************************

Punkt   Hol_Koordinaten
        lea     MoveTxt,a0
        bsr     Printf  ; Text ausgeben
        move.l  RPort,a1
        jsr     Move(a6)

PunktLoop
        Hol_Koordinaten
        lea     LineToTxt,a0
        bsr     Printf  ; Text ausgeben
        move.l  RPort,a1
        jsr     Draw(a6)

        btst    #6,MausKnopf
        beq.s   PunktLoop

        bra     Loop

MoveTxt         dc.b    "Move %d,%d",LF,0
LineToTxt       dc.b    "LineTo %d,%d,%d,%d",LF,0

************** Rechteck ********************

Rechteck

        ; dies darf der Programmierer selber machen

        bra     Loop

************** Box ************************

Box     Hol_Koordinaten
        move    d0,x1   ; x1
        move    d1,y1   ; y1 ; normal oben links

        Draw_Complement

BoxLoop Hol_Koordinaten
        move    d0,X2
        move    d1,Y2

        move    d0,d2   ; X2 nach D2; Ecken rechts unten
        move    d1,d3   ; Y2 nach D3

        move    x1,d0   ; oben rechts
        move    y1,d1

        cmp     d0,d2   ; sind die Ecken richtig, d.h.
                        ; oben links (x1) liegt auch
                        ; wirklich links (von d2, x2)?
        bhi.s   NoXVertausch    ; wenn D0>d2, dann ist OK
```

```
        exg     d0,d2   ; und andersherum
                        ; Punkt jetzt links von D2

NoXVertausch    cmp     d1,d3   ; sind die Ecken
                        ; richtig, d.h.
                        ; unten (y1) liegt auch
                        ; wirklich unten (von d3, y2)?
        bhi.s   NoYVertausch  ; wenn D1>d3, dann ist OK

        exg     d1,d3   ; und andersherum
                        ; Punkt d1 jetzt über d3

NoYVertausch
        move.l  RPort,a1
        jsr     RectFill(a6)

        Move_Start

        move    x1,d0
        move    y1,d1
        move    X2,d2
        move    Y2,d3

        cmp     d0,d2   ; siehe oben
        bhi.s   NoXVertausch2
        exg     d0,d2

NoXVertausch2
        cmp     d1,d3
        bhi.s   NoYVertausch2
        exg     d1,d3

NoYVertausch2
        move.l  RPort,a1
        jsr     RectFill(a6)

        btst    #6,MausKnopf
        beq     BoxLoop

        Draw_Normal             ; jetzt der richtige Kasten

        move    x1,d0
        move    y1,d1
        move    X2,d2
        move    Y2,d3

        cmp     d0,d2   ; siehe oben
        bhi.s   NoXVertausch3
        exg     d0,d2

NoXVertausch3
        cmp     d1,d3
        bhi.s   NoYVertausch3
        exg     d1,d3

NoYVertausch3
        move.l  RPort,a1
        lea     BoxTxt,a0
        bsr     Printf  ; Text ausgeben
        jsr     RectFill(a6)

        bra     Loop

BoxTxt  dc.b    "RectFill %d,%d,%d,%d",LF,0

************** Ausgabe der Aktivitäten *******

Printf  ; Text mit Formatierungen in A0

        movem.l d0-a6,-(SP)

        move.l  4.w,a6
        movem   d0-d3,EinsetzData
        lea     EinsetzData,a1
        lea     AusgabeRoutine(PC),a2
        lea     Puffer,a3       ; wohin damit
        jsr     RawDoFmt(a6)    ; a3 wird nicht zerstört!

        move.l  a3,d2   ; Strg Anfang von RawDoFmt
        moveq   #-1,d3
WelcheLen
        addq.l  #1,d3
        tst.b   (a3)+   ; StrgLen in d3 für Write
        bne.s   WelcheLen
```

```
        move.l  DosBase,a6
        move.l  FileHandle,d1
        jsr     Write(a6)

        movem.l (SP)+,d0-a6
        rts

AusgabeRoutine
        move.b  d0,(a3)+
        rts

EinsetzData     ds.l    4
Puffer          ds.b    100     ; 100 Bytes Puffer

************** Schließen des Demos ***********

EndOfDemo
        move.l  IntuiBase,a6
        move.l  WinHandle,a0
        jsr     CloseWindow(a6)
        bsr     NormaleMaus

        move.l  4.w,a6
        move.l  GfxBase,a1
        jsr     CloseLibrary(a6)

        move.l  IntuiBase,a1
        jsr     CloseLibrary(a6)

        move.l  DosBase,a6
        move.l  FileHandle,d1   ; Protokolldatei
        jsr     Close(a6)       ; schließen

        move.l  a6,a1
        move.l  4.w,a6
        jmp     CloseLibrary(a6)

************** Daten ************************

IntuiName       dc.b    "intuition.library",0
IntuiBase       dc.l    0

GfxName         dc.b    "graphics.library",0
GfxBase         dc.l    0

DosName         dc.b    "dos.library",0
DosBase         dc.l    0

FileHandle      dc.l    0
DateiName       dc.b    "RAM:Protokolldatei.SimPai",0
                cnop    0,2

************** Window Struktur ***************

NewWindow       dc      0,0,640,256
                dc.b    0,1
                dc.l    0
                dc.l    WINDOWSIZING|WINDOWDRAG|WINDOWDEPTH|
                        WINDOWCLOSE|ACTIVATE ; |GIMMEZEROZERO
                dc.l    0
                dc.l    0,WinTitle,0,0
                dc      260,100,0,0
                dc      1
WinTitle        dc.b    "Simple Paint",0
                cnop    0,2

WinHandle       dc.l    0
RPort           dc.l    0

************** Gadgetdaten ******************

GADGET  MACRO
         dc.l   0
         dc     \1,2,16,7       ; "Hit box" Koordinaten
         dc     GADGIMAGE|GRELRIGHT
         dc     GADGIMMEDIATE|RELVERIFY
         dc     BOOLGADGET      ;|GZZGADGET
         dc.l   \2
         dc.l   0       ; keine Selekt Zeichnung
         dc.l   0,0,0
         dc     \3
         dc.l   0
        ENDM

IMAGE   MACRO
```

```
         dc.l   0       ; keine Verschiebung
         dc     16,7
         dc     1
         dc.l   \1
         dc.b   1,0
         dc.l   0
        ENDM

gg1     GADGET -66,Image1,0
gg2     GADGET -66-16-1,Image2,1
gg3     GADGET -66-32-2,Image3,2
gg4     GADGET -66-48-3,Image4,3
gg5     GADGET -66-64-4,Image5,4

Image1  IMAGE ImageData1
Image2  IMAGE ImageData2
Image3  IMAGE ImageData3
Image4  IMAGE ImageData4
Image5  IMAGE ImageData5

ImageData1
        dc      %1110000000000000
        dc      %0011100000000000
        dc      %0000111000000000
        dc      %0000001110000000
        dc      %0000000011100000
        dc      %0000000000111000
        dc      %0000000000001110

ImageData2
        dc      %0000001111000000
        dc      %0001110000111000
        dc      %0011000000001100
        dc      %0110000000000110
        dc      %0011000000001100
        dc      %0001110000111000
        dc      %0000001111000000

ImageData3
        dc      %0000000000000000
        dc      %0110000000000000
        dc      %0011000000111000
        dc      %0001110001101100
        dc      %0000011110000110
        dc      %0000000000000011
        dc      %0000000000000000

ImageData4
        dc      %0000000000000000
        dc      %0111111111111110
        dc      %0100000000000010
        dc      %0100000000000010
        dc      %0100000000000010
        dc      %0111111111111110
        dc      %0000000000000000

ImageData5
        dc      %0000000000000000
        dc      %0111111111111000
        dc      %0111111111111000
        dc      %0111111111111000
        dc      %0111111111111000
        dc      %0111111111111000
        dc      %0000000000000000


SpriteDatenAnf
        dc.l    0
        dc      %0000000100000000,%0000000100000000
        dc      %0000000100000000,%0000000100000000
        dc      %0000000100000000,%0000000100000000
        dc      %0000000100000000,%0000000100000000
        dc      %0000000000000000,%0000000000000000
        dc      %0011110001111000,%0011110001111000
        dc      %0000000000000000,%0000000000000000
        dc      %0000000100000000,%0000000100000000
        dc      %0000000100000000,%0000000100000000
        dc      %0000000100000000,%0000000100000000
        dc      %0000000100000000,%0000000100000000
```

Der View-Port

Mit dem View-Port wird fast die unterste Ebene des Systems erreicht. Von hier an geht es fast immer an die Hardware. Das erklärt

aber noch nicht, was ein ViewPort ist, bisher haben wir in lediglich charakterisiert. Versuchen wir ihn im folgenden etwas genauer zu präzisieren. Der ganze Bildschirm wird als sein Gesamtes Display genannt. Ein Display besteht aus mehreren Viewports. Jeder Viewport kann seine eigene Grafikauflösung, Farben und Bitplanes nutzen. Wir schöpfen verdacht, das kennen wir doch schon! Na klar, von den Screens. Ein Screen kann auch seine Auflösungen und Farben haben. Wo ist also der Unterschied? Das ist einfach, ein Screen ist nichts anderes als ein ViewPort mit etwas drum herum. Jeder Screen ist ein ViewPort.

Obwohl ViewPort ziemlich flexibel sind, können wir einiges nicht mit ihnen machen, etwas, was von den Screens schon bekannt sein sollte.

ViewPorts

1. können nicht nebeneinander liegen
2. dürfen sich nicht überschneiden
3. müssen eine Pixelzeile Abstand haben

Steigen wir gleich in die Aufgaben des ViewPort ein. Wir wollen Farben verändern. Da der View-Port die Struktur ist, die für Farbänderungen zuständig ist, müssen wir sie verwenden. Der Screen, dessen Farbe wir mal ändern wollen, hat daher für uns Programmierer den RastPort und den ViewPort für uns als benutzbare Zeiger hinterlegt.

ViewPort, Größe 40 ($28) Bytes:

```
$0000     0  Next (LONG)
$0004     4  ColorMap (LONG)
$0008     8  DspIns (LONG)
$000c    12  SprIns (LONG)
$0010    16  ClrIns (LONG)
$0014    20  UCopIns (LONG)
$0018    24  DWidth (WORD)
$001a    26  DHeight (WORD)
$001c    28  DxOffset (WORD)
$001e    30  DyOffset (WORD)
$0020    32  Modes (WORD)
$0022    34  SpritePriorities (BYTE)
$0023    35  reserved (BYTE)
$0024    36  RasInfo (APTR)
```

Offset 0: Next

Das der gesamte Display aus oft aus mehreren Viewports besteht, hier die Verbindung zu den anderen.

Offset 4: ColorMap

Um jeden ViewPort seine eigenen Farben zu erlauben, muss eine Datenstruktur definiert werden. Diese `ColorMap` gibt die Farben für den Bildschirmausschnitt an.

Ich zitiere auch hier aus den INCLUDE-Dateien:

```
STRUCTURE   ColorMap,0
 BYTE    cm_Flags
 BYTE    cm_Type
 WORD    cm_Count
 APTR    cm_ColorTable
 APTR    cm_vpe
 APTR    cm_TransparencyBits
 BYTE    cm_TransparenyPlane
 BYTE    cm_reserved1
 WORD    cm_reserved2
 APTR    cm_vp
 APTR    cm_NormalDisplayInfo
 APTR    cm_CoerceDisplayInfo
 APTR    cm_batch_items
 LONG    cm_VPModeID

COLORMAP_TYPE_V1_2      = 0
COLORMAP_TYPE_V1_4      = 1
COLORMAP_TYPE_V36       = COLORMAP_TYPE_V1_4

COLORMAP_TRANSPARENCY   = 1
COLORPLANE_TRANSPARENCY = 2
BORDER_BLANKING = 4
BORDER_NOTRANSPARENCY   = 8
VIDEOCONTROL_BATCH      = $10
USER_COPPER_CLIP        = $20
```

Offset 24 und 26: DWidth, DHeight

Hier findet der Suchende Breite und Höhe des ViewPorts.

Offset 32: Modes

In `Modes` wird der aktuelle Ausgabestand vermerkt. Folgende Konstanten sind vertreten. Wir haben sie aber schon kennengelernt, als wir nämlich einen Screen öffneten.

```
GENLOCK_VIDEO     = 2
V_LACE            = 4
V_SUPERHIRES      = $20
V_PFBA            = $40
```

```
V_EXTRA_HALFBRITE = $80
GENLOCK_AUDIO     = $100
V_DUALPF          = $400
V_HAM             = $800
V_EXTENDED_MODE   = $1000
V_VP_HIDE         = $2000
V_SPRITES         = $4000
V_HIRES           = $8000
```

Offset 36: RasInfo

Eine Struktur folgenden Formates:

```
STRUCTURE  RasInfo,0
 APTR    ri_Next
 LONG    ri_BitMap
 WORD    ri_RxOffset
 WORD    ri_RyOffset
```

Gfx-Operationen im View-Port

Um Farben zu verändern kann der RastPort nicht mehr helfen, der View-Port muss her. Denn speziell bei Farbregisterbeschreibungen ist die Copper-Liste im Spiel, und die wird durch den View verwaltet.

SetRGB4(ViewPort,Reg,r,g,b)(A0,D0-D3)

Mit dieser Funktion kann innerhalb eines Views ein Farbregister verändert werden. Der Farbwert ist in die Komponenten Rot, Grün und Blau aufgespalten. Jeder der Komponenten kann Werte von 0 bis 15 annehmen, die Gesamtzahl der Farben ergibt sich damit zu 16 * 16 * 16 = 4096.

Vorteil der Funktion ist, dass die Farbwerte direkt in die ColorMap übertragen werden, und dies bedeutet wiederum, dass die mitberechnete Copper-Liste sofort die Farbe auf den Bildschirm bringt.

Farbe = GetRGB4(ColorMap,FarbReg)(A0,D0)

In einigen Fällen ist es notwendig die Farben der Color-List auszulesen. Mit `GetRGB4()` steht dem Benutzer so eine Funktion zur Verfügung. Die Farbwerte werden z. B. in einem Farbrequster verlangt, man muss ja wissen, wie die Grundfarben sind, oder in einer Bilder-Sicherungs-Routine, die die Farbinformationen auch auf das Medium schreibt. Wo die `GetRGB4()`-Funktion noch drei Register für den Farbwert benötigte, ist hier die Farbe in einem Register zusammengefasst, so wie sich auch von der Hardware aufgenommen wird. Wir haben im Kapitel über Preferences schon einmal darüber gesprochen.

Bit	Farbkomponente
12-15	keine, da ungenutzt
8-11	Rot
4-7	Grün
3-0	Blau

Wir von der Funktion eine -1 an die aufrufende Einheit zurückgegeben, so wurde versucht, von einem Farbregister auszulesen, was keine gültige Farbe enthielt. Es ist also nicht möglich auf einem 3 Plane-Screen mit 8 Farben den Farbwert von Register 18 auszulesen.

Eine Information, die verwirren könnte: -1 assoziiert man mit allen gesetzten Bit, der Farbwert Weiß ist aber auch durch alle gesetzten Bits definiert. Man darf natürlich nicht annehmen, das weiß eine ungültige Farbe ist, denn -1 sind 32 gesetzte Bits, die Farbe Weiß aber nur die untersten 12!

LoadRGB4(ViewPort,Palette,Anzahl)

Mit den beiden oberen Funktionen konnten lediglich einzelne Farbregister verändert werden. Dies verkompliziert jedoch bei großen Palettenänderungen das Programm, sinnvoll wäre nun eine Routine, die mehrere Einträge gleichzeitig ändert. Mit der Funktion `LoadRGB4()` wird aus einer Palette die Farbeinträge in die `ColorMap` des ViewPort geschrieben. Die Farbpalette ist eine Tabelle, die aus aneinandergehängten Words besteht, die das oben genannte 4-Bit-Pro-Farben Format haben. Die Farbeinträge werden bei Null beginnend geändert. Gezielt von einem Startregister die Anzahl abzulaufen ist nicht möglich, und in vielen Fällen auch nicht sinnvoll.

Um sie auch noch sichtbar zu machen, muss ein weiterer Schritt ausgeführt werden.

Mit dem Befehl `LoadView(ViewPort)` wird die erstellte Copper-Liste ausgeführt, und dadurch der View mit einer neuen Farbe dargestellt.

Die Gfx-Base

Wie jede Library gibt auch die Gfx-Lib einiges an Informationen in der Gfx-Base her. Ich habe nur ein paar herausgegriffen, und zwar die, die in den INCLUDES beschrieben sind.

```
GfxBase
  $000     0  LibNode (APTR)
  $022    34  ActiView (APTR)
  $026    38  copinit (APTR)
  $02a    42  cia (APTR)
  $02e    46  blitter (APTR)
  $032    50  LOFlist (APTR)
  $036    54  SHFlist (APTR)
```

```
$03a    58  blthd (APTR)
$03e    62  blttl (APTR)
$042    66  bsblthd (APTR)
$046    70  bsblttl (APTR)
$04a    74  vbsrv (STRUCT)
$060    96  timsrv (STRUCT)
$076   118  bltsrv (STRUCT)
$08c   140  TextFonts (STRUCT)
$09a   154  DefaultFont (APTR)
$09e   158  Modes (UWORD)
$0a0   160  VBlank (BYTE)
$0a1   161  Debug (BYTE)
$0a2   162  BeamSync (UWORD)
$0a4   164  system_bplcon0 (WORD)
$0a6   166  SpriteReserved (BYTE)
$0a7   167  bytereserved (BYTE)
$0a8   168  Flags (WORD)
$0aa   170  BlitLock (WORD)
$0ac   172  BlitNest (WORD)
$0ae   174  BlitWaitQ (STUCT)
$0bc   188  BlitOwner (APRT)
$0c0   192  TOF_WaitQ (STRUCT)
$0ce   206  DisplayFlags (WORD9
$0d0   208  SimpleSprites (APTR)
$0d4   212  MaxDisplayRow (WORD)
$0d6   214  MaxDisplayColumn (WORD)
$0d8   216  NormalDisplayRows (WORD)
$0da   218  NormalDisplayColumns (WORD)
$0dc   220  NormalDPMX (WORD)
$0de   222  NormalDPMY (WORD)
$0e0   224  LastChanceMemory (APTR)
$0e4   228  LCMptr (ATR)
$0e8   232  MicrosPerLine (WORD)
$0ea   234  MinDisplayColumn (WORD)
$0ec   236  reserved[5] (STRUCT)
```

Neu hinzugekommen ab 2.0:

```
        UBYTE   gb_ChipRevBits0 ; agnus/denise new features

        STRUCT  gb_crb_reserved,5

        STRUCT  gb_monitor_id,2 ; normally null
        STRUCT  gb_hedley,4*8
        STRUCT  gb_hedley_sprites,4*8
        STRUCT  gb_hedley_sprites1,4*8
        WORD    gb_hedley_count
        WORD    gb_hedley_flags
        WORD    gb_hedley_tmp
        APTR    gb_hash_table
        UWORD   gb_current_tot_rows
        UWORD   gb_current_tot_cclks
        UBYTE   gb_hedley_hint
        UBYTE   gb_hedley_hint2
        STRUCT  gb_nreserved,4*4
        APTR    gb_a2024_sync_raster
        WORD    gb_control_delta_pal
        WORD    gb_control_delta_ntsc
        APTR    gb_current_monitor
        STRUCT  gb_MonitorList,LH_SIZE
        APTR    gb_default_monitor
        APTR    gb_MonitorListSemaphore
        APTR    gb_DisplayInfoDataBase
        APTR    gb_ActiViewCprSemaphore
        APTR    gb_UtilityBase
        APTR    gb_ExecBase
```

Offset 34: ActiView
Ein Zeiger auf den aktuellen View.
Offset 38: copinit
Ein Zeiger auf die startende Copper-Liste.
Offset 42: cia
Für den 6526 Resource-Gebrauch.
Offset 46: blitter
Für den Blitter resource-Gebrauch.
Offset 50 und 54: LOFlist, SHFlist
Zeiger auf Copper-Liste, die gerade läuft.
Offset 58: blthd

Ein Zeiger auf die Blitternode.

Offset 206: DisplayFlags

Folgenden `DisplayFlags` werden angeboten:

```
NTSCn       = 0
NTSC        = 1<<NTSCn

GENLOCn     = 1
GENLOC      = 1<<GENLOCn
PALn        = 2
PAL         = 1<<PALn

TODA_SAFEn = 3
TODA_SAFE  = 1<<TODA_SAFEn
```

Die diskfont.library

Die Zeichensätze im Amiga OS

Immer denselben Zeichensatz vor Augen zu haben ist auf die Dauer ziemlich anödend. Nicht umsonst wurde der alte Topaz-Zeichensatz, der bis 1.3 den Rechnern im ROM mitgegeben wurde, durch einen neuen ersetzt, der viel ansprechender ist. Wer kein 2.0 hat, muss aber nicht frustriert sein, denn es gibt ja immer noch den Befehl `SetFont` im c/-Verzeichnis der Originaldiskette. Damit kann ein Font als CLI Font aktiviert werden. Als ich noch kein 2.0 hatte, griff ich auf den Zeichensatz `pearl.font` zurück. Im Nachhinein muss ich sagen, dass er dem neuen Topaz recht nahe steht.

Neuere Programme überzeugen durch gekonnte Zeichensätze, die nicht nur zur Lesbarkeit neu konstruiert werden. Auch viele Sonderzeichen können in das System eingebunden, und benutzt werden. Oft wird in den Fenstern und Menüleisten neue Zeichensätze neue Schriftzeichen angeboten, die sehr gut aussehen. Unter 2.0 können die Zeichensätze auch gesetzt werden.

Das OS unterscheidet zwischen 2 Zeichensätzen. Die aus dem ROM und die nicht aus dem ROM kommen, also externen Ursprungs sind. Im ROM sind zwei Zeichensätze gespeichert, Topaz 8 und Topaz 9 (9 ist größer als acht), die externen müssen sich im FONTS Verzeichnis einer Diskette befinden.

Grundlegende Strukturen

Um nun die Zeichensätze ansprechen zu können bedarf es eigentlich keinerlei Aufwand. Lediglich eine kleine Struktur ist zu füllen. Sie heißt `TextAttr` und hat eine Länge von 8 Bytes.

```
0  Name
4  YSize
6  Style
7  Flags
```

Hier sind die Wünsche an den Zeichensatz gestellt.

Ein Beispiel, wie so eine Struktur aussehen kann:

```
TextAttr dc.l Name
         dc 8
         dc 0

Name     dc.b "topaz",0
```

Die Verwaltung der internen und externen Fonts geschieht über dieselbe Struktur.

Nachdem `TxtAttr` geladen wurde, kann ein Zeichensatz angesprochen werden. Dazu werden zwei Funktionen verwendet. Ist der Zeichensatz intern, so arbeitet man mit der `OpenFont()`-Funktion aus der `graphics.library`, andernfalls wird eine neue bisher unbekannte Library eingeschaltet, die `diskfont.library`. Sie ist extern, und wird daher beim Öffnen von der Diskette geladen. Sie enthält nur vier Funktionen. Die wichtigste davon ist `OpenDiskFont()`, um externen Zeichensätze zu laden. `OpenDiskFont()` wird lediglich ein Zeiger auf die `TextAttr`-Struktur übergeben, und der Zeichensatz wird geladen.

Zurückgegeben wird eine andere Struktur, die `TextFont` heißt. Für weitere Arbeiten (`OpenFont()`, `AddFond()`) ist sie notwendig.

Eine genaue Beleuchtung spare ich mir.

```
TextFont
  $000     0  Message
  $014    20  YSize
  $016    22  Style
  $017    23  Flags
  $018    24  XSize
  $01a    26  Baseline
  $01c    28  BoldSmear
  $01e    30  Accessors
  $020    32  LoChar
  $021    33  HiChar
  $022    34  CharData
  $026    38  Modulo
  $028    40  CharLoc
```

```
  $02c    44  CharSpace
  $030    48  CharKern
```

Die Größe der Struktur beträgt 52 ($34) Bytes.

Grundprogramm zum Einlesen und Darstellen der Fonts

Um einen Font zu laden, und ihn in das System einzubinden, muss dieser noch aktiviert werden. Dies geschieht mit `SetFont()`.

Auch ein internen ROM Zeichensatz muss mit `SetFont()` aktiviert werden. Ein geladener externer Zeichensatz, und ein interner sind also dann von der Bearbeitung her gleich.

Ob ein Font aus dem RAM oder dem ROM kommt muss mit `AvailFonts()` erfahren werden. Die Funktion wird am Ende des Kapitels erläutert.

Das nachfolgende Programm lädt einen Zeichensatz, und gibt einen kleinen Test aus. Die Vorgehensweise ist folgende:

1. Font öffnen
2. mit `SetFont()` aktivieren
3. Text ausgeben
4. mit `CloseFont()` löschen

```
* SetFont       Version 1       6.3.92  555 Bytes

**************  VARIABLEN ********************

; Exec

OpenOldLibrary  =       -408
CloseLibrary    =       -414

; Intuition

OpenScreen      =       -198
CloseScreen     =       -66

; Disk Font

OpenDiskFont    =       -30

; Grafik

OpenFont        =       -72
SetFont         =       -66
CloseFont       =       -78

Move            =       -240
Text            =       -60
SetRast         =       -234
SetAPen         =       -342
SetDrMd         =       -354

**************  HAUPTPROGRAMM ****************

        bsr.s   OpenLibs
        move.l  d0,DiskFontBase
        beq.s   NoDiskLib
        bsr.s   ÖffneScreen
        tst.l   ScrHandle
        beq.s   NoScreen

        lea     FontName,a0     ; Name
        moveq   #20,d0  ; Größe
        bsr     HolZeichensatz
        tst     Font
        beq.s   FontNichDa

        moveq   #100,d0 ; Schreib Demo
        moveq   #110,d1
        lea     TestText,a0
        jsr     PrintText

WaitTilMouse
        btst    #6,$bfe001      ; Maustaste
        bne.s   WaitTilMouse

        bsr     SchließeZS
FontNichDa
        bsr     ScreenZu

NoScreen
        bsr     CloseDiskFontLib
NoDiskLib
        bra     CloseGfxIntui
```

```
**************  Libs öffnen ******************

OpenLibs
        move.l  4.w,a6
        lea     IntuiName,a1
        jsr     OpenOldLibrary(a6)
        move.l  d0,IntuiBase

        lea     GfxName,a1
        jsr     OpenOldLibrary(a6)
        move.l  d0,GfxBase

        lea     DiskFontName,a1
        jmp     OpenOldLibrary(a6)

**************  Öffne Test-Screen ************

ÖffneScreen
        move.l  IntuiBase,a6
        lea     NewScreen,a0
        jsr     OpenScreen(a6)
        move.l  d0,a0
        move.l  d0,ScrHandle
        add.l   #84,a0  ;Rastport
        move.l  a0,RPort
        move.l  a0,a1
        moveq   #0,d0
        move.l  GfxBase,a6
        jmp     SetRast(a6)
        moveq   #4,d0
        move.l  RPort,a1
        jmp     SetAPen(a6)

**************  Öffne den Zeichensatz ********

HolZeichensatz
        move.l  a0,TextAttr
        move    d0,TxtGröße

        move.l  DiskFontBase,a6
        lea     TextAttr,a0
        jsr     OpenDiskFont(a6) ; in den Speicher laden
        move.l  d0,Font
        beq.s   NoFontFind

        move.l  GfxBase,a6      ; läuft alles über Gfx
        move.l  Font,a0
        jsr     OpenFont(a6)    ; Fertig zum Gebrauch machen

        ; u.U.  AddFont(), damit später nicht geladen werden braucht

        move.l  Font,a0
        move.l  RPort,a1
        jmp     SetFont(a6)

NoFontFind
        rts

**************  Zeichensatz schließen ********

SchließeZS
        move.l  Font,a1
        jmp     CloseFont(a6)
        ; evtl. noch RemFont, um ZS aus dem Speicher zu entfernen

**************  Ausgaberoutine ***************

PrintText
        movem.l d0-a5,-(SP)     ; Mal so alles sichern
        move.l  a0,a5   ; a5 ist Zeiger
        move    d0,d6   ; d6 ist x Achse
        move    d1,d7   ; d7 ist y Achse

        moveq   #-1,d5  ; Länge berechnen
TxtLen  addq    #1,d5   ; Länge in d5
        tst.b   (a0)+
        bne.s   TxtLen

        move    d6,d0   ; x
        move    d7,d1   ; y
        bsr     StringAus
        movem.l (SP)+,d0-a5
        rts
```

```
**************  String ausgeben mit x/y *******

StringAus
        move.l  RPort,a1        ; Übergabeparameter d0/d1
        jsr     Move(a6)        ; Locate
        move.l  a5,a0   ; String
        move.l  RPort,a1
        move.l  d5,d0   ; Count
        jmp     Text(a6)

**************  Screen schließen **************

ScreenZu
        move.l  ScrHandle,a0
        move.l  IntuiBase,a6
        jmp     CloseScreen(a6)

**************  Libs schließen ***************

CloseDiskFontLib
        move.l  4.w,a6
        move.l  DiskFontBase,a1
        jmp     CloseLibrary(a6)

CloseGfxIntui
        move.l  GfxBase,a1
        jsr     CloseLibrary(a6)

        move.l  IntuiBase,a1
        jmp     CloseLibrary(a6)

*************   Daten ************************

IntuiName       dc.b    "intuition.library",0
IntuiBase       dc.l    1

GfxName         dc.b    "graphics.library",0
GfxBase         dc.l    1

DiskFontName    dc.b    "diskfont.library",0
DiskFontBase    dc.l    1

TextAttr        dc.l    0
TxtGröße        dc      0
                dc      0

Font            dc.l    0

NewScreen       dc      0,0,640,256,3
                dc.b    0,1
                dc      $8000,$f
                dc.l    0,Titel,0,0
Titel           dc.b    "Screen für neuen Font",0       ; Name des Screens

ScrHandle       dc.l    0
RPort           dc.l    0

FontName        dc.b    "helvetica.font",0

TestText        dc.b    "Ich bin ein Zeichensatz",0
```

Der Zeichensatz ist Helvetica. Die Größe von 20 Punkt wird der ladenden Prozedur beigegeben.

Unterprogramm zur Text-Schattierung

Um einen Text zu schattieren muss einfach mit der Farbe schwarz — sie muss aber dann vorhanden sein — oder einer dunkleren Farbe etwas unter dem Originaltext der dunklere erscheinen. Wir setzen daher erst den Text ersten um einen Pixel nach oben und unten versetzt, und dann den Originaltext.

Das alte Unterprogramm `PrintText` muss gegen das neue ausgetauscht werden.

```
**************  Ausgaberoutine ***************

PrintText
        movem.l d0-a5,-(SP)     ; Mal so alles sichern
        move.l  a0,a5   ; a5 ist Zeiger
        move    d0,d6   ; d6 ist x Achse
        move    d1,d7   ; d7 ist y Achse

        moveq   #-1,d5  ; Länge berechnen
TxtLen  addq    #1,d5   ; Länge in d5
        tst.b   (a0)+
        bne.s   TxtLen
```

```
        moveq   #2,d0
        move.l  RPort,a1
        jsr     SetAPen(a6)

        move    d6,d0   ; x
        move    d7,d1   ; y
        bsr     StringAus

        moveq   #4,d0
        move.l  RPort,a1
        jsr     SetAPen(a6)

        moveq   #0,d0
        move.l  RPort,a1
        jsr     SetDrMd(a6)

        move    d6,d0   ; x
        move    d7,d1   ; y
        addq    #1,d0   ; x+1
        addq    #1,d1   ; y+1
        bsr     StringAus
        movem.l (SP)+,d0-a5
        rts
```

Unterprogramm zum Text-Outline

Um einen Text einen Outline zu geben ist in folgenden Schritten vorzugehen:

1. Zeichenmode JAM1 setzen
2. Outline Farbe setzen item Text in alle Himmelsrichtungen ausgeben
3. weitere Farbe setzen
4. Text an Originalposition setze

Auch hier muss das alte Unterprogramm `PrintText` muss gegen das neue ausgetauscht werden.

```
**************  Ausgaberoutine ****************

PrintText
        movem.l d0-a5,-(SP)
        move.l  a0,a5   ; a5 ist Zeiger
        move    d0,d6   ; d6 ist x Achse
        move    d1,d7   ; d7 ist y Achse

        moveq   #-1,d5
TxtLen  addq    #1,d5   ; Länge in d5
        tst.b   (a0)+
        bne.s   TxtLen

        moveq   #0,d0
        move.l  RPort,a1
        jsr     SetDrMd(a6)

        move    d6,d0   ; x
        move    d7,d1   ; y
        bsr.s   StringAus
        move    d6,d0
        addq    #1,d0   ; x+1
        move    d7,d1   ; y
        bsr.s   StringAus
        move    d6,d0   ; x
        move    d7,d1
        addq    #1,d1   ; y+1
        bsr.s   StringAus
        move    d6,d0
        addq    #1,d0   ; x+1
        move    d7,d1
        addq    #1,d1   ; y+1
        bsr.s   StringAus

        move    d6,d0
        subq    #1,d0   ; x-1
        move    d7,d1   ; y
        bsr.s   StringAus
        move    d6,d0   ; x
        move    d7,d1
        subq    #1,d1   ; y-1
        bsr.s   StringAus
        move    d6,d0
        subq    #1,d0   ; x-1
        move    d7,d1
        subq    #1,d1   ; y-1
        bsr.s   StringAus
```

```
        moveq   #5,d0
        move.l  RPort,a1
        jsr     SetAPen(a6)

        move    d6,d0   ; x
        move    d7,d1   ; y
        bsr.s   StringAus
        movem.l (SP)+,d0-a5
        rts
```

Was noch alles mit den Fonts zu veranstalten ist

Die Zeichensätze allgemein verfügbar machen

Mit `OpenDiskFont()` sind die externen Zeichensätze zwar geladen, und haben den gleichen Status wie Interne, sie sind jedoch nach dem Programmende irgendwie verloren. Um unser Programm nicht immer zu einem neuen Laden zu bewegen, wäre es sinnvoll, dass ein geladener Zeichensatz einmal im Speicher steht, und dann von jedem Programm genutzt werden kann. Um das zu erreichen, können wir mit `AddFont()` und `RemFont()` sie in das System einbinden bzw. entfernen. Beide Funktionen kommen aus der `graphics.library`.

Fontliste erstellen

Mit einer neuen Funktion aus der `diskfont.library` können alle Fonts aus bestimmten Bereichen geholt werden. Die Prozedur heißt

```
AvailFont(Buffer,LenOfBuffer,Modus)
```

Dabei ist:

Buffer
: Adresse eines freien Speicherbereiches. Dieser wird dann mit `AvialFontsHeader` Struktur gefüllt.

BufferLen
: Größe des Speichers

Modus
: 1=RAM/ROM, 2=DISK, 3=Egal

Status = 0, dann alles OK, sonst Anzahl der Bytes, um die der Buffer zu klein war

Ist der Status nicht null, so muss ein größerer Speicherbereich zur Verfügung gestellt werden, und die Funktion neu aufgerufen werden. Dies kann leider lange dauern, so dass es sinnvoll ist, genügend Speicher mit auf den Weg zu geben.

Verändern der Zeichensätze

Alle Zeichensätze können verändert werden. Dazu dient die Funktion `SetSoftStyle()`, die wir schon im `graphics.library`-Kapitel kennenlernten. Für die externen Fonts sind diese Funktionen natürlich auch zu benutzen. `AskSoftFont()` erfragt dabei eine Maske. Alle Änderungsmöglichkeiten werden von der Funktion zurückgeben. Dann kann mit `SetSoftStyle()` auch gesetzt werden.

File-Selector

Die req.library

Die `req.library` ist eine re-entrant library, die von Colin Fox und Bruce Dawson programmiert wurde. Sie soll dem Programmierer alle möglichen Dialoge (Requester) bereitstellen: Color-Requester, File-Requester, Message-Display-Requester und andere Funktionen zur Gadget-Erstellung.

Wir wollen nur eine kleine Funktion der riesigen Bibliothek nutzen. Es sei dem Leser eine weitere Nutzung überlassen, die nicht versäumt werden sollte.

Wir wollen einen File-Requester erstellen, der dem Benutzer eine Auswahl einer Datei erlaubt. Dazu benutzen wir die Funktion `FileRequester()` aus der `req.library`, die selbstredend vorher geöffnet werden muss. Zu übergeben ist der Funktion ein Zeiger auf die `FileRequesterStructure` im `A0`-Register. Die Einträge können fast überall null sein. Der Rückgabeparameter kann `TRUE` oder `FALSE` sein. Der Zeiger auf den gewählten Dateinamen wird nicht übergeben, er muss aus der Struktur ausgelesen werden.

Wichtig ist das die Versionsnummer stimmt. Bei Darstellen des Dateiauswahldialogs ist die Versionsnummer der Library anzugeben.

```
REQVERSION = 2
```

Noch zwei kleine weitere Konstanten sind in der Headerdatei zu finden.

```
DSIZE      = 130
FCHARS     = 30
WILDLENGTH = 30
```

Nun zur FileRequester-Struktur, der der Funktion zum Öffnen übergeben werden muss:

```
UWORD   frq_VersionNumber       ; da haben wir´s
APTR    frq_Title               ; Titeltext
APTR    frq_Dir                 ; Directoryvoreinstellung
APTR    frq_File                ; Wo der Dateiname hinkommt
APTR    frq_PathName            ; Wo der Path-Name hinkommt
APTR    frq_Window              ; Erschein-Window, o. NULL
UWORD   frq_MaxExtendedSelect   ; Anz. makierbarer Fileeinträge, NULL =alle
UWORD   frq_numlines            ; Zeilenanzahl im Window
UWORD   frq_numcolumns          ; Spaltenbreite
UWORD   frq_devcolumns          ; Spaltenbreite im Device-Window.
ULONG   frq_Flags               ; Flags, siehe später
UWORD   frq_dirnamescolor       ; Dir-Farbe
UWORD   frq_filenamescolor      ; Filename-Farbe
UWORD   frq_devicenamescolor    ; Device-Name-Farbe
UWORD   frq_fontnamescolor
UWORD   frq_fontsizescolor
UWORD   frq_detailcolor
UWORD   frq_blockcolor          ; null, wenn Block-Pen
UWORD   frq_gadgettextcolor     ; Farbe der Bool-Gadgets
UWORD   frq_textmessagecolor    ; Voreinstellung: 1
UWORD   frq_stringnamecolor     ; Textfarbe für Drawer, File, Hide, Show. Norm.:3
UWORD   frq_stringgadgetcolor   ; Farbe für Borders und String-Ggadgets: Norm.:3
UWORD   frq_boxbordercolor      ; Farbe für Boxen des File/Directories : Norm.:3
UWORD   frq_gadgetboxcolor      ; Farbe für Boxen um Bool-Gadgets. Norm.:3
STRUCT  frq_RFU_Stuff,36        ; für spätere Erweiterungen
STRUCT  frq_DirDateStamp,ds_SIZEOF ; Kopie der cached Directories-Date-Stamp.
UWORD   frq_WindowLeftEdge      ; wenn FRQABSOLUTEXY gesetzt
UWORD   frq_WindowTopEdge       ; s.o. Ecken des Fensters
UWORD   frq_FontYSize           ; für die Fonts
UWORD   frq_FontStyle           ; Font-Bit muss gesetzt sein
APTR    frq_ExtendedSelect      ;ESStructures für mehrere Dateien
```

Sollten nicht unbedingt geändert werden

```
STRUCT  frq_Hide,WILDLENGTH+2   ; Wildcards für versteckt Dateien
STRUCT  frq_Show,WILDLENGTH+2   ; Wildcards für gezeigte Datei
WORD    frq_FileBufferPos       ; Cursor Position
WORD    frq_FileDispPos         ; der drei
WORD    frq_DirBufferPos        ; Gadgets
WORD    frq_DirDispPos
WORD    frq_HideBufferPos
WORD    frq_HideDispPos
WORD    frq_ShowBufferPos
WORD    frq_ShowDispPos
```

Folgendes darf nicht benutzt werden und ist privat:

```
APTR    frq_Memory              ; Speicher für Dir-Entries.
APTR    frq_Memory2             ; Für versteckte Dateien
APTR    frq_Lock                ; mögl. Lock versch. Dirs beim Lesen
STRUCT  frq_PrivateDirBuffer,DSIZE+2 ;Für Namen des Dirs
APTR    frq_FileInfoBlock       ; struct FileInfoBlock
WORD    frq_NumEntries
WORD    frq_NumHiddenEntries
WORD    frq_filestartnumber
WORD    frq_devicestartnumber
LABEL   frq_SIZEOF
```

Nachfolgend sind die Bits der Flags aufgeführt. Es sind nur die Bits, noch nicht die Werte!

```
FRQSHOWINFOB    =  0 ; .info-Dateien werden angezeigt
FRQEXTSELECTB   =  1 ; Extended Select
FRQCACHINGB     =  2 ; Directory-Caching. Einstellung: Nein
FRQGETFONTSB    =  3 ; Fontrequester, kein Filerequester
FRQINFOGADGETB  =  4 ; Info-Files-Gadget nicht dargestellt
FRQHIDEWILDSB   =  5 ; 'show' und 'hide' nicht angezeigt
FRQABSOLUTEXYB  =  6 ; Absolute x,y Positions
FRQCACHEPURGEB  =  7 ; Cache löschen, wenn Date-Stamp Änderung
FRQNOHALFCACHEB =  8 ; Wenn Directory nicht eingelesen, auch nicht
FRQNOSORTB      =  9 ; Keine sortierten Directories
FRQNODRAGB      = 10 ; Kein Drag- und Deep-Gadget
FRQDIRONLYB     = 13 ; Auch ein Directory kann angewählt werden.
```

File Requester aus der asl.library

Benutzer des 2.0 Betriebssystems können auf die `requester.library` verzichten (müssen es aber natürlich nicht). Sie können einen File-Requester der externen `asl.library` nutzen.

Wie so ein Programm aussieht, können wir leicht am nachfolgenden Demo demonstrieren.

```
* FileReq       Version 1       13.4.92 226 Bytes

**************  VARIABLEN ********************

OldOpenLibrary  =       -408    ; Exec
CloseLibrary    =       -414

AllocFileRequest=       -30     ; Asl
RequestFile     =       -42
FreeFileRequest =        -36

**************  HAUPTPROGRAMM *****************

        move.l  4.w,a6
        lea     AslName,a1
        jsr     OldOpenLibrary(a6)
        move.l  d0,AslBase
        beq     ErrOpenAsl

        bsr     FileReq

ErrOpenAsl
        move.l  4.w,a6
        move.l  AslBase,a1
        jmp     CloseLibrary(a6)

**************  Aufruf der Requester Routine **

FileReq move.l  AslBase,a6
        jsr     AllocFileRequest(a6)
        move.l  d0,FileRequester

        move.l  d0,a0
        move.l  #ReqTitel,(a0)
        move.l  #FilePuffer,4(a0)
        move.l  #DirPuffer,8(a0)
        move    #50,22(a0)
        move    #10,24(a0)
        move    #350,26(a0)
        move    #230,28(a0)
        jsr     RequestFile(a6)

        move.l  FileRequester,a0
        jmp     FreeFileRequest(a6)

**************  DATEN *******+******************

AslName         dc.b    "äsl.library",0
AslBase         dc.l    0

FileRequester   dc.l    0    ; Zeiger auf Speicher für FileReq

ReqTitel        dc.b    "Titel",0
FilePuffer      ds.l    10
DirPuffer       ds.l    10
```

Nach dem Öffnen werden einige Felder gefüllt, die aber nicht vorgestellt werden.

Der File-Requester ist sowieso nicht so toll, da kann man lieber auf PD-Requester zurückgreifen, die sind zurzeit noch wesentlich besser und schneller. Kein Wunder, dass der Asl-Requester oft gepatcht wird. Ich will daher auf den Asl-Requeser nicht weiter eingehen.

Copper und die Hardware

In diesem Kapitel wollen wir uns von dem Amiga Betriebssystem lösen, und uns auf den Weg in die Weiten der Hardware gegeben.

Die Hardware ist ja neben den RAM-Bausteinen und Floppylaufwerken noch vielseitiger, und man versteht eigentlich unter Hardwareprogrammieren das Benutzen der Systembausteine.

Allgemeine Hardwareinformationen

Die Hardware des Amigas ist in mehrere Komponenten strukturiert. Da wären zunächst einmal die CPU, die RAM-Bausteine, der Floppy-Controller, die Grafikbausteine, der Taktgeber, der Musi-Mann, und noch so einige kleine Sachen. CPU Programmierung dürfte klar sein, RAM wird immer angesprochen. Bleiben noch die Bauteile, die die Datenkommunikation und die audio-visuelle Darstellung übernehmen.

Kleine Hardware-Programme

In bestimmten Speicherzellen, die wir gleich etwas näher kennenlernen wollen, finden wir Werte, die ausgelesen werden können. Dies hört sich zwar selbstverständlich an, ist es aber noch lange nicht. Denn, nicht jeder Speicherbereich kann geschrieben und gelesen werden.

Eine Speicherzelle, mit der Adresse $BFE001 haben wir schon kennengelernt. Der Inhalt ist der Zustand einer gedrückten Maustaste, ob gedrückt, oder nicht, 0 oder 1 ist der Inhalt. Die Hardware regelt das selber, es ist also nicht so, das irgend ein Interrupt da arbeitet, und die Werte beschreibt.

```
* Taste oder Maus abwarten.MOD Version 1

GetKey   =  $bfec01
GetMouse =  $bfe001

       move.b   GetKey,d1
Warte  move.b   GetKey,d0
       cmp.b    d1,d0
       bne.s    End
       btst     #6,GetMouse
       bne.s    Warte
End    rts
```

Zufallswerte selbstgemacht

```
Random movem.l  d1,-(sp)
       move     #100,d1
NeuRnd clr.l    d0
       move.b   $dff007,d1

       mulu     $dff006,d1
       move.b   $dff007,d0
       rol.l    d0,d1
       nop
       move.b   $dff007,d2
       mulu     $dff006,d2
       move.b   $dff007,d3
       rol.l    d3,d2
       or.l     d2,d1
       move     d1,d0
       beq.s    NeuRnd
       and.l    #$ff,d0
       move     LoRnd,d1
       cmp.b    d1,d0
       blo.s    NeuRnd
       move     HiRnd,d1
       cmp.b    d1,d0
       bhi.s    NeuRnd
       movem.l  (sp)+,d1
       rts

LoRnd  dc       60
HiRnd  dc       100
```

Zufallswerte mathematisch

```
* Zufallswerte berechnen Version 1 62 Bytes
       moveq    #100,d0 ; Maximalbereich
       bsr      Rnd
       move.l   d0,Tst
       rts

Rnd    lea      RndSeed(PC),a0
       move     d0,d1
       ble.s    NewSeed
       move.l   (a0),d0
       add.l    d0,d0
       bhi.s    Hi
       eor.l    #$1d872b41,d0
Hi     move.l   d0,(a0)
       and.l    #$ffff,d0
       divu     d1,d0
       swap     d0
       rts

NewSeed neg     d1
       move.l   d1,(a0)
       rts

RndSeed dc.l    0
Tst     dc.l    0
```

Die erste Routine nutzt zur Ermittlung einer Zufallszahl eine Adresse, an der sich die Rasterzeile befindet. Der Wert wird noch ein bisschen verändert, und wir erhalten einen Zufallswert.

Leider ist dieser Weg nicht besonders schön, es gibt noch eine weitere Variante, Zufallszahlen quasi zu errechnen. Man geht von einem Startwert aus, verändert diesen, und nimmt ihn wieder als neuen Startwert. Eine Routine, die auch häufig in C oder anderen Programmiersprachen zu finden ist.

Die Basis-Adressen der Customchips

Alle Koprozessoren sind über Speicherzellen zu „erreichen“. Diese Speicherzellen sollte man schon kennen, daher eine belehrende Tabelle.

Adresse	Offset	Variable
$dff000	0	bltddat
$dff002	2	dmaconr
$dff004	4	vposr
$dff006	6	vhposr
$dff008	8	dskdatr
$dff00a	10	joy0dat
$dff00c	12	joy1dat
$dff00e	14	clxdat
$dff010	16	adkconr
$dff012	18	pot0dat
$dff014	20	pot1dat
$dff016	22	potinp
$dff018	24	serdatr
$dff01a	26	dskbytr
$dff01c	28	intenar
$dff01e	30	intreqr
$dff020	32	dskpt
$dff024	36	dsklen
$dff026	38	dskdat
$dff028	40	refptr
$dff02a	42	vposw
$dff02c	44	vhposw
$dff02e	46	copcon
$dff030	48	serdat
$dff032	50	serper
$dff034	52	potgo
$dff036	54	joytest
$dff038	56	strequ
$dff03a	58	strvbl
$dff03c	60	strhor
$dff03e	62	strlong
$dff040	64	bltcon0
$dff042	66	bltcon1
$dff044	68	bltafwm
$dff046	70	bltalwm
$dff048	72	bltcpt
$dff04c	76	bltbpt
$dff050	80	bltapt
$dff054	84	bltdpt
$dff058	88	bltsize

$dff05a	90	pad2d[0]
$dff060	96	bltcmod
$dff062	98	bltbmod
$dff064	100	bltamod
$dff066	102	bltdmod
$dff068	104	pad34[]
$dff070	112	bltcdat
$dff072	114	bltbdat
$dff074	116	bltadat
$dff076	118	pad3b[]
$dff07e	126	dsksync
$dff080	128	cop1lc
$dff084	132	cop2lc
$dff088	136	copjmp1
$dff08a	138	copjmp2
$dff08c	140	copins
$dff08e	142	diwstrt
$dff090	144	diwstop
$dff092	146	ddfstrt
$dff094	148	ddfstop
$dff096	150	dmacon
$dff098	152	clxcon
$dff09a	154	intena
$dff09c	156	intreq
$dff09e	158	adkcon
$dff0a0	160	aud[0]
$dff0a0	160	aud[0].ac_ptr
$dff0a4	164	aud[0].ac_len
$dff0a6	166	aud[0].ac_per
$dff0a8	168	aud[0].ac_vol
$dff0aa	170	aud[0].ac_dat
$dff0ac	172	aud[0].ac_pad[]
$dff0e0	224	bplpt[]
$dff0f8	248	pad7c[]
$dff100	256	bplcon0
$dff102	258	bplcon1
$dff104	260	bplcon2
$dff106	262	pad83
$dff108	264	bpl1mod
$dff10a	266	bpl2mod
$dff10c	268	pad86[]
$dff110	272	bpldat[]
$dff11c	284	pad8e[]
$dff120	288	sprpt[]

$dff140	320	spr[]
$dff140	320	spr[].pos
$dff142	322	spr[].ctl
$dff144	324	spr[].dataa
$dff146	326	spr[].datab
$dff180	384	color[]

VPOSR nutzen um Chipsatz zu prüfen

Die Speicherstelle $dff004 kann geschickt dazu genutzt werden, um an die Hardwareinformation ECS, BigAgnus zu kommen. Die Möglichkeit zeigt das folgende Programm.

```
* Checkagnus    Version 2       14.6.92 366 Bytes

OldOpenLibrary  =       -408    ; Exec
CloseLibrary    =       -414

Output          =       -60     ; Dos
Write           =       -48

VPOSR           =       4

        move.l  4.w,a6
        lea     DosName,a1
        jsr     OldOpenLibrary(a6)
        move.l  d0,a6
        jsr     Output(a6)
        move.l  d0,WinHandle

        lea     $dff000,a5
        move    VPOSR(a5),d0
        and     #$7F00,D0
        tst     D0
        beq.s   PALAgnusChip
        cmp     #$1000,D0
        beq.s   NTSCChip
        cmp     #$2000,D0
        beq.s   ECSPalChip
        cmp     #$3000,D0
        beq.s   ECSNTSCChip

        ;       kann keine Angabe über ihn gemacht werden!

        bra.s   CheckDenisChip

**************  Agnus Chip testen *************

PALAgnusChip
        move.l  #PALAgnus,d2
        bsr.s   WriteTxt
        bra.s   CheckDenisChip

NTSCChip
        move.l  #NTSCAgnus,d2
        bsr.s   WriteTxt
        bra.s   CheckDenisChip

ECSPalChip
        move.l  #ECSPALAgnus,d2
        bsr.s   WriteTxt
        bra.s   CheckDenisChip

ECSNTSCChip
        move.l  #ECSNTSCAgnus,d2
        bsr.s   WriteTxt

**************  Denise Chip testen ************

CheckDenisChip
        move    $f8(a5),d0
        and     #$ff,d0
        cmp     #$fc,d0
        beq.s   ECSDeniseChip

        move.l  #NormalDenise,d2
        bsr.s   WriteTxt
        bra.s   Ende
```

```
ECSDeniseChip
        move.l  #ECSDenise,d2
        bsr.s   WriteTxt
        bra.s   Ende

**************  Textausgabe und Ende **********

WriteTxt
        move.l  WinHandle,d1
        move.l  d2,a0
        moveq   #-1,d3
StrgLen addq.l  #1,d3
        tst.b   (a0)+
        bne.s   StrgLen
        jmp     Write(a6)

Ende    move.l  a6,a1
        move.l  4.w,a6
        jmp     CloseLibrary(a6)

**************  Standartvariablen *************

DosName         dc.b    "dos.library",0

WinHandle       dc.l    0

**************  Aussagetexte ******************

PALAgnus        dc.b    "Normaler PAL Agnus Chip",0
NTSCAgnus       dc.b    "Normaler NTSC Agnus Chip",0
ECSPALAgnus     dc.b    "Erweiterter ECS PAL Agnus Chip",0
ECSNTSCAgnus    dc.b    "Erweiterter ECS NTSC Agnus Chip",0

NormalDenise    dc.b    " mit normalem DENISE Chip im Amiga.",10,10,0
ECSDenise       dc.b    " mit ECS DENISE Chip im System.",10,10,0
```

Copperprogrammierung

Der Copper ist einer der Koprozessoren, die die Bildschirmdarstellung übernehmen.

Copperprogramm Nr. 1

Der Copper, ist als richtiger Koprozessor zu sehen. Er kann eigene Programme ausführen. Dazu benötigt er einen eigenen Speicherbereich, der selbstverständlich im Chip-RAM liegen muss. Das Programm, welches ausgeführt wird, wird in der sogenannten Copper-Liste abgelegt. Die Befehle, drei an der Zahl sind vielleicht wenig, aber für die Programmierung absolut ausreichend.

Wie sieht eine Copper-Liste aus? Ein Beispiel:

```
dc   $6001,$fffe
dc   $0180,$111
dc   $6201,-2
dc.l $01800122
```

Das Beispiel wird die Programmierung, die recht einfach ist, illustrieren.

```
* Copper1Demo Version 3 19.3.91 262 Bytes

OldOpenLibrary =  -408 ; Exec
CloseLibrary   =  -414

LOFlist        =      $32
INTENA         =      $dff09a
COP2LCL        =      $dff084

************** Hauptprogramm *****************

           bsr.s  Init

MausTaste  btst   #6,$bfe001
           bne.s  MausTaste

           bra.s  End

************** Fertigmachen ******************

Init       move.l 4.w,a6
           lea    GfxName,a1
           jsr    OldOpenLibrary(a6)
           move.l d0,GfxBase

           move.l d0,a0
```

```
            move.l LOFlist(a0),OldCopperList
            move   #%0100000000000000,INTENA ; Ints sperren
            move.l #CopperList,COP2LCL ; Copper-Liste darstellen
            rts

************** Alles wie's war machen ********

End         move.l GfxBase,a0
            move.l OldCopperList,LOFlist(a0)
            move   #%1100000000000000,INTENA ; Ints freigeben

            move.l 4.w,a6
            move.l GfxBase,a1
            jmp    CloseLibrary(a6)

************** Daten ************************

GfxName          dc.b   "graphics.library",0
GfxBase          dc.l   0

OldCopperList    ds.l 1

************** Hier kommt die Copper-Liste ****

CopperList dc      $0180,$0100
           dc      $6001,$fffe
           dc      $0180,$111
           dc      $6201,-2
           dc.l    $01800122
           dc.l    $6401fffe
           dc.l    $01800133
           dc.l    $6601fffe
           dc.l    $01800144
           dc.l    $6801fffe
           dc.l    $01800155
           dc.l    $6a01fffe
           dc.l    $01800166
           dc.l    $6c01fffe
           dc.l    $01800177
           dc.l    $6e01fffe
           dc.l    $01800188
           dc.l    $7001fffe
           dc.l    $01800199
           dc.l    $7201fffe
           dc.l    $018001aa
           dc.l    $7401fffe
           dc.l    $018001bb
           dc.l    $7601fffe
           dc.l    $018001cc

           dc.l    $7801fffe
           dc.l    $018001dd
           dc.l    $7a01fffe
           dc.l    $018001ee
           dc.l    $7c01fffe
           dc.l    $018001ff
           dc.l    $7e01fffe

           dc.l    $01800000

           dc.l    -2           ; Enderkennung für Copper-Liste
```

Zunächst die obligatorischen Unterprogrammaufrufe. `Init` initialisiert den Copper und startet die Copper-Liste. Nach dem Drücken der Maustaste wird die unsere Copper-Liste gelöscht, und die alte wieder eingetragen.

Kommen wir zu `Init`. Nach dem Öffnen der Gfx-Lib wird ein Zeiger auf die vom Betriebssystem verwendete Copper-Liste in `OldCopperList` gespeichert. Die Copper-Liste, die vom OS verwendet wird, ist durch den Eintrag LOFlist für uns transparent. Wegen der Schönheit wollen wir auch gleich die Interrupts sperren. Dies geschieht durch das Schreiben von %0100000000000000 in die Adresse `INTENA`. Dann soll unsere Copper-Liste dargestellt werden. Es ist darauf zu achten, das sie CHIP-Mem steht. Als ich erst einen Amiga ohne Speichererweiterung hatte, und dann mein Programm auf einem anderen laufen lassen wollte, musste ich feststellen, dass es nicht lief. Das war ziemlich frustrierend!

Na ja. Das Unterprogramm `End` trägt die `OldCopperList` wieder in `LOFlist` zurück. Die Interrupts werden wieder erlaubt, das Programm ist zu Ende.

Copperprogramm Nr. 2

Farben untereinander geben ein gutes Bild, doch auch nebeneinander sind sie als Abgrenzung effektvoll. Um dieses Resultat zu erzeugen, werden die Bildschirmanfahrbefehle fallengelassen, und lediglich die Farben gesetzt. Da dies eine Zeitlang dauert entsteht ein guter, aber simpler Effekt.

```
* Copper2Demo Version 3 19.3.91 442 Bytes

OldOpenLibrary =  -408 ; Exec
CloseLibrary   =  -414
```

```
LOFlist     =       $32
INTENA      =       $dff09a ; Hardware
COP2LCL     =       $dff084

************** Hauptprogramm *****************

            bsr.s   Init

            bsr.s   BaueCopAuf

MausTaste   btst    #6,$bfe001
            bne.s   MausTaste

            bra.s   End

************** Fertigmachen ******************

Init        move.l  4.w,a6
            lea     GfxName,a1
            jsr     OldOpenLibrary(a6)
            move.l  d0,GfxBase

            move.l  d0,a0
            move.l  LOFlist(a0),OldCopperList
            move    #%0100000000000000,INTENA ; Ints sperren
            move.l  #Cop,COP2LCL ; Copper-Liste darstellen
            rts

************** Alles wie's war machen ********

End         move.l  GfxBase,a0
            move.l  OldCopperList,LOFlist(a0)
            move    #%1100000000000000,INTENA ; Ints frei

            move.l  4.w,a6
            move.l  GfxBase,a1
            jmp     CloseLibrary(a6)

************** Copper-Liste aufbauen **********

AnzFarb     =   45 ; verschiedene Farben

BaueCopAuf  lea     CopFarben,a0
            lea     FarbTabelle,a1
            moveq   #AnzFarb-1,d0 ; Anzahl Farben
            move    #$180,d1 ; Anfang ist $dff180
AlleFarben  move    d1,(a0)+ ; Farbregister
            move    (a1)+,(a0)+ ; Farbe setzen
            dbra    d0,AlleFarben
            rts

************** Daten ************************

GfxName     dc.b    "graphics.library",0
GfxBase     dc.l    0

OldCopperList dc.l 0

FarbTabelle dc      $ce3,$ae3,$8e3,$7e3,$5e3,$4e3,$3e3,$3e5,$3e7
            dc      $3e8,$3ea,$3eb,$3ec,$3ee,$3de,$3ce,$3ae,$39e
            dc      $37e,$34e,$33e,$43e,$63e,$73e,$83e,$a3e,$b3e
            dc      $c3e,$e3e,$e3d,$e3b,$e3a,$e39,$e37,$e36,$e34
            dc      $e33,$e53,$e63,$e83,$e93,$ea3,$eb3,$ec3,$ee3

************** Hier kommt die Copper-Liste ****

Cop         dc      $0180,0 ; Zuerst Schwarz
            dc      $5037,$fffe ; Dann warten bis Zeile $50

CopFarben   ds.l    AnzFarb ; Freihalten

            dc      $0180,0 ; Zum Schluss schwarz

            dc.l    -2 ; Enderkennung für Copper-Liste
```

In der Copper-Liste wird `ANZFAR` Words platztgelassen, und eine Schleife kopiert die Farbwerte in die Copper-Liste.

Der Prozessor MC68000

In den folgenden Kapiteln wollen wir etwas mehr über den MC68000 und seine Nachfolger kennenlernen.

Entstehung und Philosophie

Der Mikroprozessor, der im Amiga 500 (500+) aktiv (mit)arbeitet, ist der MC68000 von der Herstellerfirma Motorola . Der Name, oder besser die Zahl, kommt daher, dass er ca. 68000 Transistoren enthält. Diese sind auf einer Fläche von 6,2 mal 7,1 mm untergebracht. Der Prozessor gehört schon der vierten Generation an. Zur damaligem Zeit machte Intel, heute in allen PCs als Hauptprozessor angestellt, das Rennen (Diese Prozessoren sind eigentlich noch aus der 3. Generation). Motorola, und z. B. Zilog mit dem Z80, versuchten in das lukrative Geschäft einzusteigen. Dies versuchten sie mit Prozessoren höherer Leistungsfähigkeit, die z. B. mehr Befehle hatten, schnellere Ausführbarkeit aufwiesen und mehr Speicher adressieren (ansprechen) konnten. Der Trend ging in den späteren Jahren einfach nach Prozessoren, die idealer für höhere Programmiersprachen, insbesondere Compiler, waren. Eine strukturierte Programmierung und eine Überschaubarkeit sollte auch auf der untersten Ebene noch gewährleistet sein.

Motorola wollte einen Super-Prozessor dieser Art herausbringen. Im Jahre 1977 wurde das neue Projekt gestartet. Mit dem Grundgerüst vom MC6800 gelang es den Spezialisten Tom Gunter und Co. XC 68000 (gelegentlich auch M68000 genannt) im Jahre 1979 auf den Markt zu bringen. Doch für den großen Markt war es zu spät, denn Intel hatte die Macht im Chip-Geschäft, und dass dies immer noch so ist, zeigt das heutige Bild. (Der damalige Intel Prozessor konnte jedoch nur einen kleinen Speicherbereich ansprechen. Er entsprach 64 KB. (Hätte gut in den C-64 reingepasst). Doch heute verfügen die Rechner über größere Mengen Speicher (CAD Anlagen und Grafik Studios zur Bildnachbearbeitungen benötigen Giga-Speicher), so dass Intel, die ja Prozessoren herstellen wollten die zu den alten kompatibel sein sollten heute rummurkst ohne Ende, da ja der erste Intel-Prozessor nur 64 KB ansprechen konnte. Man trickste das bei den 640 KB PC-Rechnern so hin, dass man Bänke einführte. Diese Bänke waren je 64 KB groß und daraus folgte, dass immer zwischen den Bänken rumgeschaltet werden musste, um in den Genuss des ganzen Speichers zu kommen. Wer einen C-128 hat, dem wird das vielleicht bekannt vorkommen, denn auch dort ist der Speicher in Bänke eingeteilt. Dieses Umschalten nennt man Bank switching, und der Hardwareteil, der dies übernimmt nennt man Bank-Select-Logic. (Für Insider: Auch der C-64 hat dies im kleinen Format, denn man bedenke nur das nutzbare RAM, welches unter dem ROM noch zu verwenden ist! (Ist nur ein Vergleich!!) Bei den PCs prägt ein 64K-Bänkesystem ein Assemblerprogramm, und über eine Berechnung mit Hilfe von Segmenten und Offsets kommt man zu der ersehnten Adresse. (Doch zum Trost sei gesagt, dass die neuen Intel-Prozessoren (ab den 80286) auch diese Bänke weglassen können und einen durchgehenden Speicher haben!)

Zweithersteller

Neben Motorola stellen jedoch auch noch andere Chiphersteller den 68000 her. Genauso wie NEC der Intel 8088 neu herausbrachte, der sogar schneller war und einige Fehler verbesserte, wird auch der 68000 von Zweitherstellern (Second Sources) gebaut. Unter ihnen sind Firmen wie Hitachi (HD68000), Mostek (MK68000), Rockwell (R68000), Thomson CSF (EF68000), Signetics/Phillips/Valvo (SC68000). Die Anzahl der Zweithersteller zeigt, wie begehrt der Prozessor ist. In den zehn Jahren seit der Vorstellung sind schon mehr als 30 Millionen Exemplare aus der Familie verkauft werden. Jeden Monat kommt eine Million dazu!

Erster Einsatz

Allerdings wurde der 68000 erst 1984 so richtig eingesetzt. Es war der Platinencomputer Gepard, der erstmals zeigte, welche faszinierenden Möglichkeiten in dem Prozessor stecken. Die in Florida sitzende Firma Amiga, die als Joystick- und Konsolenfirma bekannt war (sie baute auch den Joyboard, eine Art Joystick, den man mit den Füßen bedienen konnte), fing 1982 mit dem Bau eines Super-Computers an. Dieser, unter dem Namen Lorraine angekündigt, sollte ein Heimcomputer werden, dessen Ziel Flugsimulationen waren. Der Amiga hatte 3 Väter, und wurde von Jay Miner (Chip-Entwickler bei Atari), David Morse (Manager einer Spielwarenfirma) und später R. J. Mical (Programmierer)eingeleitet. Doch da der Firma Amiga zwischendurch das Geld ausging, obwohl die Entwickler schon Hypotheken auf ihren Häusern hatten, übernahm Commodore dieses Projekt. Zuvor jedoch versuchte der bei Commodore gekündigte und Aufkäufer von Atari, Jack Tramiel, sein Glück, pokerte jedoch etwas zu hoch, und bekam das Angebot nicht. Vielleicht hat er jedoch mit dem Falcon (oder dem Jaguar, der mal so nebenbei 800 Mill. Punkte in der Sekunde plottet) etwas mehr Glück als mit den STs! (Das Commodore die Firma letztendlich übernahm hatte auch Vorteile, denn sie sorgte für eine Erweiterung eines Coprozessors (Blitter), der heute Linien zeichnen kann, dies hatte die Firma Amiga im Ur-Blitter nicht eingeplant.)

MC68000 in anderen Geräten

Neben Amiga erkannten auch andere Computerhersteller die Vorzüge des 68000 und setzten ihn ein. Zu den nennenswertesten zählen Apple (Macintosh, engl. Regenmantel)), NeXt und Atari. Da diese Rechner den gleichen Prozessor haben wie der Amiga, ist es auch einfach, Atari oder Macs zu emulieren, da hier keine aufwendige, softwaremäßige Prozessoremulation veranstaltet werden muss, wie beim PC-Emulator, der ja durch seine Langsamkeit im Emulationsmodus gerade einen C-64 überrundet. Das einzige, was bei einer Betriebssystem-Emulation noch übernommen werden muss, ist eben das Betriebssystem. Doch dies ist kein Problem, da es genauso im Speicher steht wie alle anderen Programme auch. Die Geschwindigkeit von 7 MHz ist also voll zum Arbeiten da, und man merkt bei ordentlich programmierter Atari Software (z. B. Calamus, nicht aber Omikron BASIC) nicht, dass dies eigentlich ein Amiga ist. (Jedoch nur mit dem Einsatz einer Flicker-Flacker-Karte, denn der Atari hat eine Vertikalauflösung von 400 Bildpunkten, und da flackert der Amiga nun mal, da diese Auflösung nur im Interlace-Modus erreicht wird (die Augen dürfen sich freuen). Doch ändert sich dies bei den Rechnern 500+ und 3000er, die sie im Gegensatz zu den alten Amigas über neue Grafikauflösungen, z. B. den Produktivitätsmodus (640 * 400 Punkte, vier Farben) verfügen, wo ohne Flackern locker diese Auflösung erreicht wird.) Ganz zu schweigen vom Amiga 1200 und Amiga 4000.

Was kommt noch von Motorola?

In den Anfangszeiten, in der die Firma Motorola Mikroprozessoren herstellte, sind integrierte Schaltungen wie der 6800, 6802, 6803, 6809 entstanden. Zu dem schon auf den Markt befindlichen MC68000 brachte Morotola noch den MC68008 heraus. Er hat den gleichen Befehlssatz wie der MC68000, doch unterscheidet er sich vom ihm in einem wesentlichen Punkt. Er spricht nur einen kleineren Speicherbereich an, und da er mit 48 Pins deutlich kleiner ist als der 68000 mit 64 Pins, eignet er sich besonders gut für kleine Einplatinenrechner. Der 68008/10 (die Zahl hinter dem Querstrich gibt die maximale Taktgeschwindigkeit an (die die Firma für angemessen hält)) kostet etwa 25 DM.

Weiterentwicklungen

Eine direkte Weiterentwicklung des 68000, obwohl doch schnell fertig, ist der MC68010. Er ist in der Lage die Trap-Vektoren-Tabelle (Exception-Vektortabelle) zu verschieben. Zudem hat er die prima Einrichtung, den Loop Modus, der eine Weiterentwicklung des Befehl-Prefetches (Befehl aus dem Speicher in den Prozessor holen) ist. Durch diese Technik kann meist der letzte Befehl und die Registerwahl gespeichert werden. Der Prozessor verfügt intern über ein Zwei-Wort-Prefetch und ein Ein-Wort-Befehls-Register. Ist der Prozessor einmal in den Loop Modus gelangt (durch die spezielle Schleifenkonstruktion DBcc), brauchen die Befehle (33 sind Möglich) nicht erneut aus dem Speicher geholt zu werden, sondern stehen in den Prefetch-Registern zur Verfügung. Der Prozessor benötigt also nur noch die Zeit für die Ausführung der Befehle, führt jedoch keinen Befehls-Lade-Zyklus (Fetch) durch. Dies kann bei Verschiebungen und Suchaktionen sehr sinnvoll sein. Der 68010/8 kostet 28 DM und ist eigentlich gut einzusetzen. (Siehe bei „Was ist nutzbar“)

Ein weiterer Prozessor der 680x0 Familie ist der 68012, der aber eine unwichtige Rolle spielt. (Wenn er überhaupt noch eingesetzt wird). Er erweitert den 68010 insofern, dass er mehr Speicher adressieren kann.

MC68020

Eine nennenswerte Erweiterung der Familie ist der 1984 vorgestellte MC68020. Er unterscheidet sich gänzlich von den Vortypen, was auch sofort im Preis deutlich wird. Er hat im Gegensatz zu seinen Vorgängern keine gemultiplexte Adressierung (d. h. die Daten-Longword werden in 2 Bytes aufgespalten), sondern verfügt über volle 32 Bit, wodurch der Buszyklus von 4 Taktzyklen auf 3 vermindert wird, weil die 32 Bit Werte nicht mehr hintereinander über den 16 Bit Datenbus geholt werden müssen. Das bedeutet natürlich auch eine höhere Pinzahl (114 Pins), so dass der Prozessor nicht mehr in einem Dual-In-Line (Doppelreihen oder Doppellinien Gehäuse, wie die RAM-Bausteine) untergebracht wird sondern erstmals in einem Pin-Grid-Gehäuse (Die Pins sind in mehreren Reihen aufgelistet, so wie der Agnus Grafik Chip) gebaut werden musste. Zudem hat der Prozessor erstmals eine Coprozessor Schnittstelle und ein Cache (Speicher im Prozessor). Der Cache von 256 Bytes erlaubt es Maschinensprachebefehle in dem kleinen Speicher schon vor der Ausführung parat zu haben. Auch werden bei Multiplikationen 64 Bit Zahlen (Quadworte unterstützt), im Gegensatz zu den Alten, die nur 32 Bit Zahlen bei der Multiplikation erhalten. Die erste Version des 68020 lief nur auf einer Taktfrequenz von 12,5 MHz. Heutige Typen sind bis 33 MHz taktbar. Er kostet in der 12 MHz Version 290 DM, in der 16 MHz Version 378 DM und in der 20 MHz Version 500 DM.

Eine embedded Version des MC68020 schuftet auch im Amiga 1200.

MC68030

Der darauffolgende Prozessor, der MC68030, ist eine komplette Neukonstruktion. Er kann auf 33 MHz getaktet werden, eine Steigerung von 9 MHz gegenüber den alten, deren Maximalfrequenz durch 25 MHz gegeben waren. Auch das Cache-Prinzip wurde verbessert. Anstatt einem Cache-Speicher spendierte man dem 68030 gleich zwei.

Auch mit ihrem neusten Flagschiff, den MC68040, Sommer 1990 vorgestellt, hat Intel mit dem 80468 gegen Motorola keine Schnitte mehr. Da Intels Prozessoren im Allgemeinen schneller sind, musste man etwas machen. Und man schrieb ein Programm, das feststellen sollte, welche Befehle am häufigsten in Programmen benutzt werden. Es zeigt sich, dass zu 90% die Befehle zur Speichermanipulation benötigt werden. Man hat nun das Prinzip, nach dem der Intel-Prozessor arbeitet (Intel ist kein reiner RISC Prozessor, sondern eine Mischung aus RISC und CISC) teilweise übernommen und diese Befehle werden ohne Microcodes verarbeitet, benötigen daher im Optimalfall auch nur noch einen Taktzyklus. Befehle, die weniger häufig benutzt werden, werden immer noch von Microzyklen aufgerufen (CISC-Prinzip). Außerdem ist der eingebaute Matheprozessor im Motorola Chip um einiges schneller als der Intel-Mathe. Eine weitere Verschnellerung bieten die beiden Caches, mit etwas schwachen 4 KB s, die jedoch für die meisten Anwendung ausreichen.

RISC von Motorola

So nebenbei: Um sich besser an den Markt anpassen zu können, entwickelte auch Motorola einen RISC Prozessor. Er ist unter dem Namen 88100 bekannt, und wird sogar häufig eingesetzt. Dieser 32-Bit Prozessor verfügt über 52 Befehle und natürlich 32-Bit-Register und erledigt bis zu 17 MIPS (Millionen Instruktionen pro Sekunde; Man lässt den Prozessor den schnellsten Befehl abarbeiten, und ermittelt dadurch die Zahl, die angibt, wieviel Befehle der Prozessor in einem Zeitraum verarbeitet. Der Pentium soll einmal 100 MIPS haben, unser 68000 hat vielleicht 0,7 MIPS). Doch alleine hat dieser Prozessor nicht viel zu sagen, denn es kommen noch 2 Cache-Verwaltungs-Chips, die MC88200er hinzu. Erst sie machen den MC88100 zu einem fähigen Gesamtpaket, das unter dem Namen 88000-System zusammengefasst wird.

Übrigens können wir bald eine Koproduktion der Superfirmen Intel und Motorola erwarten, denn sie planen einen gemeinsamen Chip. Intel hat in der Herstellung von den Chips bezüglich der internen Verdrahtung einen Vorteil und Motorola bringt ihr Wissen bezüglich der komplexen Befehle in das Projekt ein.

Zusatzkomponenten (PMMU, FPP)

Neben den Prozessoren stellt Motorola auch noch andere Computerkomponenten her. Es sei hier nur der Peripheriekontrollbaustein 68120 und 68121, das Bus-Interrupt-Modul 68153, der Parallelinterface und Zeitgeberbaustein 86230, die seriellen Ein- /Ausgabebausteine 68564, (oder Multi-Protocol Communications Controller (MPCC) 68652) oder der Multi-Funktions-Peripheribaustein (MFP) 68901 genannt.

Wichtig in diesem Zusammenhang ist auch die Page-Memory-Management-Unit PMMU 68451 zu nennen, die die virtuelle Speicherverwaltung, Einsatzbereich Mehrbenutzersysteme, z. B. Unix, zu verwirklichen vermag. Als Ergänzung zum 68020 gedacht, und im 88030 schon integriert, ist sie in der Lage den Speicher in Seiten (engl. pages) von 256 Byte bis 32 KB aufzuteilen. Diese Seiten können dann die gleichen Adressen (logische Adresse) haben (es erscheint für den Benutzer so), aber intern wird der Speicher z. B. nacheinander verwaltet (physikalische Adresse).

Bekannt ist des Weiteren ein Fließkomma-Arithmetik-Koprozessor (FPP) oder einfach FPU (Float-Pointing-Unit) genannt. Diese Art von Coprozessoren lassen Fließkommaoperationen 100-mal schneller über die Bühne gehen. Der Koprozessor FFP 68881 ist eine ideale Ergänzung zum MC68020 und stellt dem Benutzer Fließkommaarithmetik zur Verfügung. Neben den 4 Grundoperationen sind auch Restwertbestimmung, Vergleiche, Wurzel, Sin-, Cos-, Arc/Tan-, Log-, e^x-, ln-Funktionen implementiert. In einem Kapitel wird ja auf den Matheprozessor etwas genauer eingegangen, damit die Geschwindigkeit des Computersystems durch eine FPU weiter erhöht werden kann.

Eine Weiterentwicklung ist der FFP 68882, der immer noch auf Erweiterungskarten Platz findet. Er ist etwas doppelt so schnell wie der MC 68881. Bin ich jedoch in Besitz eines MC68040 kann mir dies egal sein, da die implementierte FPU des 68040 ca. 30-mal schneller arbeitet (bei der gleichen Frequenz versteht sich). Der Nachteil ist nur: Die implementierte FPU kennt nur die Grundrechenarten. Das ist zwar schade, da man für die speziellen Rechenoperationen immer noch eine zweite FPU benötigt, aber somit spart man wertvollen Chip-Platz.

Was ist einbaubar im Amiga?

Wenn wir die Amigazeitungen aufschlagen, werden uns immer wieder Werbungen ins Auge fallen. Slogans wie „Noch billiger“ und „Noch besser“ sowie „Aktueller Update“ prägen dieses Bild. Fast unscheinbar sind die Beschleunigerkarten (klar, bei dem Preis). Sie reichen vom 68000 auf 16 MHz getaktet bis 68020-68040er Karten die bis zu 50 MHz getaktet werden können. Einen Computer mit 68000, der auf eine höhere Taktgeschwindigkeit läuft, kann sich jeder selbst anfertigen, wenn er den eingebauten 8 MHz Chip (MC68000P8) (der für eine Zugriffszeit etwa 500 ns benötigt) gegen einen 16 MHz Chip (MC68000P16) austauscht und ihn mit einem eigenen Oszillator (Taktgeber) (der ca. 5 DM kostet) am Taktbein 20 des Prozessors versieht. Ich bitte aber um Vorsicht, denn die Beine (Pins) könnten leicht abbrechen, denn eigentlich braucht man zum herausheben eines so großen Chips eine Spezialzange. Die Utensilien, wie Prozessor, Oszillator und Zange sind im Elektronik Fachhandel zu bekommen. Der Normale MC68000 ist standardmäßig auf 8 MHz getaktet und kostet 12 DM. Typen, die bei höheren Frequenzen arbeiten sind selbstverständlich teurer. So kostet der 68000/10 14,50 DM, der 68000/12 14,50 DM, der 68000/14 17 DM und der 68000/16 47 DM.

Die Erweiterungskarten bieten dem Amiga User höhere Taktgeschwindigkeit an, und die Prozessoren haben ab dem 30er einen Mathecoprozessor, der schnellere Fließkommazahlenoperationen erlaubt. Zudem ist durch Cachespeicher (Speicher im Prozessor) (beim 40er auch zwei mit je 4 KB) der Speicherzugriff um 80-90% auf diesen verlagert, da nicht immer alles über den Datenbus laufen muss. Überhaupt ist der Speicher viel zu langsam und die heutigen Supermodern-Prozessoren sind schon schneller als der Zugriff auf den RAM-Speicher. Wenn man durch Erweiterungskarten den Cachespeicher noch erhöht, so was gibt es schon für den 3000er lassen sich umgerechnet 75 MHz erreichen. Ein Computer müsste also auf 75 MHz getaktet werden, um diese Leistung ohne Cache zu erbringen. Cache bringt Vorteile. Doch auch durch einen Prozessor wie der 68010, den jeder in seinem Computer einsetzen kann, kann man die Geschwindigkeit erhöhen. Denn mit diesem Prozessor lassen sich Schleifendurchläufe schneller erledigen. Somit wird die Prozessorleistung im Durchschnitt um 20% erhöht. Leider hat dieser Prozessor einen kleinen Nachteil in den TRAP-Routinen. Doch gibt es da ein PD-Programm auf der Amiga-Lib-Disk (Fisch-Disk) Nr 18. namens DeciGEL (Zitat des Autors: „Relief from MC68010 pains on the Amiga“), die diesem Fehler entgegenwirken. Doch ein PC- oder Atari-Emulator habe ich bis jetzt auf diesem Prozessor noch nicht zum Laufen gebracht. Und wer wie der A500+ oder 3000 ein 2.0 Betriebssystem hat, bei dem funktioniert es auch mit Normalprozessor 68000 nicht.

Was die Erweiterungskarten nicht können, ist jedoch die Geschwindigkeit der Coprozessoren (Commodore gab ihnen Namen wie Copper (Co-Prozessor), Blitter (Block-Transfer), Paula oder Denise) (auch dazu später mehr) heraufsetzen. Selbst bei dem A3000 bleibt sie bei 7 MHz, die Taktgeschwindigkeit eines normalen Amigas. Etwas schwach. Doch lässt sich im Laufe des Jahres 1992 auf einen Amiga++ mit 14 Megaherz Takt- und Coprozessor-Frequenz warten. Die Geschwindigkeit ist u. a. ein Grund, warum die Demos auf dem 3000 fast alle nicht mehr laufen. Es liegt einfach daran, das immer angenommen wird, die Coprozessoren seien schneller als der Prozessor, und wenn eine super 40er Turbokarte ihren Dienst in einem voll aufgerüsteten Amiga tut, ist das nicht mehr der Fall, denn durch die Unabhängigkeit der vielen Prozessoren (der arme PC) ist der Prozessor schon fertig, wenn der Blitter gerade mal anfängt seine Blöcke zu verschieben. Das Programm kommt aus dem Takt, und verabschiedet sich mit einem roten Ticket zum Guru nach Indien. Das ist in den meisten Fällen natürlich sehr ärgerlich, denn jeder sollte sich den Reiz der Demos nicht entziehen, denn nur sie zeigen wirklich, was ein Computer so alles draufhat. Am PC wäre dies schon gar nicht möglich, denn die Hardware ist zu verschieden. Es gibt zwar nun VGA als Grafik-Standart und Soundblaster als Microsoft-Window-Musikstandart, doch sind immer noch andere Karten im Einsatzbereich, auf denen nun wiederum einige Tricks nicht laufen. Zudem sind die Taktgeschwindigkeiten zu verschieden, eine Abstimmung mit der Hardware ist also nicht zu erzielen. Der C-64 und Amiga sind Idealcomputer, da sie alle die gleichen Hardware-Voraussetzungen erfüllen.

Die Prozessorbefehle und ihre Opcodes

Ein Prozessor hat viel zu tun. Den ganzen Tag rechnen und vergleichen, rumspringen und verknüpfen. Sollten wir da nicht erst einmal eine Gedenkstunde für den Gestressten einführen?

Die ganze Informationsflut verarbeitet unser 680x0 ja grandios, und der große Befehlssatz macht ihm zum Mnemonik-Dolmetscher sowieso. Wenn man beachtet, wie er mit der großen Anzahl Befehlskombinationen umgeht, denkt man sich manches Mal: „Wie macht er das bloß?“ Uns schwupp sind wir schon im Kapitelstoff.

Die Frage, die in diesem Kapitels diskutiert wird, lautet: Wie erkennt und bearbeitet die CPU unsere Programme, sprich Befehle, die das Programm formen?

Vorab lassen sich einige Gruppen finden, in der sich Befehle einordnen lassen. Einleuchtend wird sein, dass SUB und ADD, oder LSL und LSR näher zusammengehören als beispielsweise EXG und MOVE oder NOT und RESET. Verbindungen sind in den Aufgabengebieten des Prozessors somit leicht zu entdecken.

Des Weiteren kommen wir vom Aufbau der Befehle sehr schnell zum Assemblerbau. Wie und welche Techniken man benutzen kann, will ich ebenfalls zur Sprache bringen.

Befehle ohne Parameter

Schon in den ersten Kapiteln war zu erfahren, dass ein Befehl aus mindestens 16 Bit d. h. einem Word besteht. Mit 16 Bit lässt sich schon eine Menge anfangen, so sind theoretisch 2^{16} = 65636 Befehleskombinationen denkbar. Es haben jedoch einige Bit an bestimmten Positionen vordefinierte Bedeutungen, sodass die Anzahl der Bit, die Befehle identifizieren, gewaltig schrumpft. Bei Einzelbefehlen ist dies anders. Jeder Mnemonik hat eine eigene Kennung. Die folgende Tabelle zeigt Befehlswörter mit ihren Bitkombinationen:

Mnemonik	Bitkombination (15-0)
ILLEGAL	%0100101011111100
RESET	%0100111001110000
NOP	%0100111001110001
STOP	%0100111001110010
RTE	%0100111001110011
RTD	%0100111001110100
RTS	%0100111001110101
TRAPV	%0100111001110110
RTR	%0100111001110111

Sehen wir von Mnemonik ILLEGAL ab, tritt deutlich hervor, dass die übrigen mit der Bitkombination %010011100111.... beginnen. Im Folgenden wollen wir die Bitkombination auch Opcode nennen.

Suchroutinen eines Assemblers

Besprochen sind die einfachsten Befehle, und ein Assembler wird keine Schwierigkeit haben, diese in entsprechende Opcodes (so nennen sich die Bitbelegungen) umzusetzen.

Aber wie kommt es dazu, dass der Assembler die Befehle erkennt, und in Bit umgesetzt? Ich möchte verschiedene Möglichkeiten

zur Befehlserkennung diskutieren.

Vor dem Suchen muss der Compiler erst einmal zu einem Schlüsselwort kommen. Wichtig ist, dass die Schlüsselwort-Erkennungsroutine Returns, Leerzeichen und Tabulatoren überspringt, denn sie sind Trennzeichen und gehören nicht zum Befehl. Oft kommen im Programmquelltext Bemerkungen hinzu, die ebenfalls übersprungen werden müssen. Erschwerend wirkt, dass eine Labeladresse, eine Variablenzuweisung oder Variablendeklaration (Konstante) auch als solche erkannt werden muss, und nicht fälschlicherweise als Schlüsselwort.

Ein Beispiel aus einem willkürlichen Programmtext:

```
; label ; Bemerkung 1,2,3
Ulli
Willi = 8-12

Klaus nop; Langeweile
```

Hier müssen die ersten vier Zeilen und der Labelname Klaus sowie das Leerzeichen übersprungen werden, um zum NOP zu kommen. Nachdem alles unnütze (für die Befehlserkennung) beiseite gelassen wurde, kann der Assembler den Befehl in einen Puffer kopieren. Kopiert wird ab dem ersten Zeichen und die Routine abgebrochen, wenn ein Trennzeichens (Delimiter) gefunden wurde. Nach dem Abschluss mit einem Nullbyte ist der Befehl zur Weiterbearbeitung fertig. Der Puffer hat etwa folgendes Format.

```
Puffer  dc.b "nop",0
```

Ist nach erfolgreicher Separation vom Programmtext ein Befehl „frei“ (isoliert), kann die Erkennung gestartet werden, ob der Befehl existiert, und welche Maßnahmen entsprechend ergriffen werden müssen.

Die Erkennung besteht dabei aus dem Suchen der Befehlswörter aus einer Tabelle und springen zum passenden Unterprogramm, wo weiters erfolgt. Zum Suchen gibt es viele Möglichkeiten. Kurz ansprechen werde ich folgende Techniken:

1. Sequentielles Suchen
2. Binäres Suchen
3. Suchen über Längen
4. Hash-Suche

Die Erste Suchmöglichkeit wäre:

Der Assembler durchsucht sequentiell eine Tabelle, in der alle Befehlswörter tabellarisch mit einer Sprungadresse verbunden sind, und kann dann das Unterprogramm anspringen, welches die Bitkennungen in den Speicherbereich schreibt, in dem das Programm hineinassembliert werden soll.

Eine Beispieltabelle:

```
Tab     dc.b    "MOVE",0
        dc.b    "SUB",0
        dc.b    "ADD",0
        .
        .
        dc.b    0       => Ende

Adr     dc.l    _Move
        dc.l    _Sub
        dc.l    _Add
        .
        .
        .
```

Der Nachteil dieses Suchens liegt auf der Hand. Um zum letzten Befehl zu kommen, müssen wir alle 55 restlichen Befehle um mindestens einen Buchstaben verglichen haben. Aufgrund dessen kann man nun oft gebrauchte Befehle vorne einordnen, sodass ein Vergleich möglichst wenig ausgeführt wird. Statistiken besagen deutlich, dass der MOVE-Befehl am häufigsten auftaucht. Logischerweise ordnen wir ihn in der Tabelle so an, dass der Puffer direkt mit ihm verglichen wird. Was nach dem MOVE kommt könnte etwa in der Häufigkeit ein ADD sein. Also ordnen wir ein ADD so an, der er direkt nach dem Vergleich mit MOVE kommt. Die Suche kann also immens beschleunigt werden, indem die Befehle geschickt angeordnet werden.

Leider ist dieses Suchen, obwohl der Algorithmus einfach ist, langsam, und so suchen wir uns eine andere Technik aus, mit der wir schneller zum Ziel kommen. Denn einleuchtend ist, dass die Anzahl der Befehle, die oft gebraucht werden, schnell erschöpft ist, und Befehle auftauchen, die gebraucht werden, aber alle sporadisch, und nicht unbedingt häufig. So kann es wegen der Befehlsvielfalt leicht vorkommen, das ein Vergleich mit LSL erst an 20. Stelle auftaucht, denn nach einem subjektiven Befinden sind andere Befehle häufiger, sodass z. B. MULU an 19. Stelle gestellt wird.

Auch Compiler haben Einfluss auf die Befehlswahl, denn sie sind nie optimaler als Assembler-Programmierer, und der wird versuchen, Befehlsmöglichkeiten und Adressierungsarten auszunutzen. Im Gegensatz dazu stehen Compiler, die oft MOVEen, PEAen und EXTen. Jeder kann zugeben, das der PEA-Befehl von Assemblanern nicht (oder nur ganz wenig) (vielleicht von Atarianern) gebraucht wird. Ebenso eine Vorzeichenerweiterung mit EXT (sehen wir von den mathematischen Operationen ab), denn als Programmierer haben wir immer im Kopf, ob sich nun ein Word, Long oder Byte im Register befindet. Compiler dagegen nutzen PEAs immer, wenn Adressen auf den Stack sollen, und das passiert häufig, man denke nur, wie oft `printf`, `scanf` oder ähnliches aufgerufen wird. Auch EXTed werden häufig genötigt, denn unser Compiler hat so seine Schwierigkeiten mit dem Merken von Wertbereichen.

Sollte das sequentielle Suchen bevorzugt werden, so müssen im Fall, dass schnelle Ergebnisse gewünscht werden, auch verschiedene Häufigkeitstabellen angelegt werden, ganz danach, was für einen „Befehlsvorrat“ der Programmier oder Compiler besitzt. Wir merken schon, viel zu langsam, und viel zu aufwändig.

Über schnellere Suchroutinen kann man sich allerdings auch wieder streiten. Doch besprechen wir erst einmal eine neue Möglichkeit bevor wie tadeln.

Binäres Suchen:

Jeder hat bestimmt schon etwas vom binären Suchen behört. Diese Technik soll folgend diskutiert werden.

Wir beginnen in der Mitte mit dem Suchen des Schlüsselwortes, und beim Nicht finden halbieren und suchen wir solange weiter, bis das Wort gefunden wurde. Bei 56 Befehlen benötigten wir im Ernstfall sechs Wort-Vergleiche. Dies ist wenig und dementsprechend schnell. Die Zwischenstopps, die erreicht werden, um den ersten Befehl in der ASC-Tabelle zu finden, sind 28, 14, 7, 4, 2 und 1.

Sehen wir uns einmal praktisch eine solche Teiltabelle an, die nach Buchstaben sortiert ist.

```
        dc.b    "ADD",0
        dc.b    "BRA",0
        dc.b    "BSR",0
        dc.b    "EXT",0
        dc.b    "MOVE",0
        dc.b    "SUB",0
        dc.b    0       ; Ende der Tabelle => nicht gefunden

Tab     dc.l    AddUP
        dc.l    BraUP
        dc.l    BsrUP
        dc.l    ExtUP
        dc.l    MoveUP
        dc.l    SubUP
```

Wir fangen in der Mitte mit dem Suchen an, und nehmen als Startwert drei, denn die Anzahl der Befehle ist sechs. Also die Mitte mal vier (Ein Adresse besteht aus vier Byte) und eine Addition von Tab, und wir haben den Pointer auf den ersten zu vergleichenden Befehl. Das wäre BRA. Nun werden mit einer Stringvergleichroutine die Zeichenketten überprüft. Liefert der Vergleich das Ergebnis, String kleiner, dann halbieren wir und suchen weiter. Ist der String größer so halbieren wir ebenfalls (einmaliges shiften nach links) und addieren es zur Mitte in. Wenn der Befehl gefunden ist, hören wir selbstverständlich auf.

Dieses Suchverfahren ist schnell, hat allerdings immer noch den Nachteil, dass der String mit vielen anderen verglichen werden muss. Es muss immer noch das Suchen nach dem String mit den passenden Anfangsbuchstaben durchgeführt werden, und mit passenden ersten Zeichen die folgenden verglichen werden, usw. Das bedeutet z. B. beim ILLEGAL Befehl immer noch ein Vergleichen um sieben Zeichen (obwohl diesen Befehl sowieso keiner braucht). Auch wenn ein Makroname mit `ILLEGAL_Adresse` gewählt wurden, muss noch 8-mal verglichen werden, um zum Schluss zu kommen: Oh, kein Schlüsselwort!

Kleiner Trick an dieser Stelle: Strings von hinten an zu untersuchen kann schneller sein.

Die dritte Möglichkeit:

Eine weitere Suchmöglichkeit wäre ein Hash-Algorithmus. Der Trick dabei ist, über ein Suchschlüssel die Wörter zu identifizieren. Einfaches Beispiel: Eine Vorauswahl kann über die Stringlänge erfolgen. Die Tabelle enthält dabei als ersten Byte die Stringlänge (Befehle länger als 255 Zeichen sind mir nicht bekannt (oder dem Leser vielleicht?)) und dann folgen die eigentlichen Zeichen. Ein Zeichenkettenabschluss mit dem Nullbyte ist nicht mehr nötig, denn um das Ende zu erreichen, addieren wir die Stringlänge auf den Anfangszeiger des Strings, und schon sind wir beim nächsten Tabelleneintrag, der erneut mit der Länge beginnt. Beim Suchen kann anhand der Länge, die entweder 2 (OR, ST, Assemblerschlüsselwort, z. B. `dc`, `ds`), 3 (NOP, EXT), 4 (MOVE), 5 (MOVEM, RESET, TRAPV), 7 (ILLEGAL) eine Auswahl der Befehle erfolgen.

Doch überdenken wir einmal ernsthaft diese Technik. Bringt sie beim Suchen überhaupt etwas? Die Antwort ist einfach, und kann verneint werden. Da sowieso fast alle Befehle drei oder vier Zeichen haben, kann man sich die Suchtechnik ersparen, Befehle wie ILLEGAL oder RESET kommen höchstens ein, zwei mal im ganzen Quelltext vor, beim sequentiellen Suchen lassen sich diese Befehle auch ganz hinten finden. Na ja, mit der Vorauswahl war's wohl nix. Ich möchte dennoch nicht unerwähnt lassen, dass bei ASCII Texten dieses Suchen dennoch effizienter als sequentielles Suchen ist, denn dort finden wir Wörter mit verschiedenen Längen, und die Vorauswahl bringt Geschwindigkeit ein, wenn man nur diese Satz anschaut wird es einem klar, er enthält nicht nur Wörter, die durchschnittlich 3 oder 4 Zeichen lang sind, sondern auch viele Wörter mit mehreren Buchstaben.

Hash-Suche

Das Beispiel mit der Längeangabe gehört nicht ganz zum Hash-Sort, macht aber die Idee deutlich, über bestimmte Merkmale zum Befehl zu kommen. Allerdings ist die Länge dabei kein eindeutiges Merkmal jedes Befehles, sondern es existieren oft mehrere Befehle mit gleicher Länge. Und nach der Vorauswahl muss immer noch sequentiell die Tabelle mit Mnemoniks gleiche Länge durchsucht werden.

Dem Programmierer muss es nun gelingen, eine eindeutige Zuordnung zu finden, sodass das ASC-Suchen ersetzt werden kann.

Nehmen wir als Beispiel an, unser Prozessor hätte vier Befehle:

```
O
AD
SUB
MOVE
```

Bei der Erkennung von SUB kann nun direkt daraus geschlossen werden, drei Zeichen d. h. in einer Tabelle mit Sprungadressen die dritte Adresse nehmen, und zu ihr springen. Jetzt ist aber das Problem der Längengleichheit, was tun? Wir könnten die Buchstaben nehmen, und miteinander verknüpfen (durch odern der ASCII-Zeichen)! Bei zwei Befehlswörtern, z. B. MOVE und MULU sind jetzt die Werte, die durch Verknüpfung entstanden sind, unterschiedlich. Da bei der Separation direkt ein Verknüpfungswert ermittelt werden kann, benötigen wir kein ASCII-Suchen und Vergleichen mehr, sondern nur ein Suchen in einer Tabelle mit Verknüpfungswerten. Diese Werte können nun geordnet vorliegen, und über binärem Suchen ist schnell eine Zahl gefunden, die als Index zu einer Tabelle verwendet werden kann.

Ein Beispiel zu dieser vierten Technik, mit fiktiven Verknüpfungswerten:

- Der Befehl DIVU ergibt den Verknüpfungswert 123
- ADD ergibt den Verknüpfungswert 92
- ROR ergibt den Verknüpfungswert 102
- ST ergibt den Verknüpfungswert 82

Geordnet in einer Tabelle, wenn man binär sucht (sonst natürlich auch beliebige Anordnung, wenn sequentiell gesucht werden

soll):

```
WertTab dc      82
        dc      92
        dc      102
        dc      123

JMPTab  dc.l    ST_Up
        dc.l    ADD_Up
        dc.l    ROR_Up
        dc.l    ST_Up
```

Taucht im Quellcode ROR auf, so ermittelt der Assembler den Verknüpfungswert 102. Das geht auch ohne ihn vorher in einen Puffer zwischenzusichern. In der Werte-Tabelle wird nun nach dem Teilungsverfahren gesucht, und die Adresse `ROR_Up` ermittelt. Dorthin springt der Assembler, und fertig. Noch ein weinig schneller geht es, wenn zu jedem Schlüsselcode (0-65536) eine Sprungadresse existieren würde, aber das wäre in der Praxis etwas zu lang. Ein in ein Word codiertes Wort (schwierig, schwierig) würde dann ein einziger Eintrag aus einer Tabelle mit der Größe von 64 KB sein, und wer spendiert das schon?

So gut wie die Routine auch sein mag, Nachteil hat sie natürlich auch. Die liegt aber nicht am Prinzip, sondern an der Einfachheit. Einige Beispiele, woran unsere Routine scheitert:

MOVE, MVOE, MEOV, MEVO, EVOM, und so viele mehr

Alle enumerierten Wörter werden gleich angesehen. „Verständlich“, wird man sagen, denn beide ergeben denselben Verknüpfungswert, die gleichen Buchstaben werden ja benutzt. Also, was tun? Ganz einfach, man muss die Position, an der das Zeichen steht, mit einem Definitionswert verbinden, also die Position mit verschlüsseln. Etwas so:

```
  1. ASC Zeichen * 128
+ 2. ASC Zeichen *  64
+ 3. ASC Zeichen *  32
+ 4. ASC Zeichen *  16
+ 5. ASC Zeichen *   8
```

Wenn man jetzt die Reihenfolgen der Buchstaben ändert, ändert sich automatisch der Schlüssel mit.

Nachdem nun mittels eines Codes die zum Befehl gehörige Sprungadresse ermittelt wurde springen wir unser Befehlsunterprogramm an. Dort wird der entsprechende Opcode in den Speicher geschrieben, und der Befehlszeiger zwei Byte weiter gesetzt, um die nächste(n) Bitkombination(en) anzufügen.

Befehle mit einem konstanten Übergabeparameter

Etwas komplizierter wird die Programmierung der Unterprogramme bei Übergabeparametern. Wir beginnen mit Befehlen, die einen konstanten Ausdruck fordern. Konstante Ausdrücke sind z. B. in der Form 3 + 4 - %101 im Quelltext anzutreffen. Es können jedoch auch Variablen oder Konstanten in den Term eingebaut werden, sodass diese Werte mit in die Berechnung einfließen müssen. Der Assembler muss aufgrund dieses Problems der zusammengesetzten mathematischen Ausdrücke über einen Parser verfügen, der diese Einzelausdrücke auswertet. Dies erschwert die Programmierung ungemein, denn Parser mit Auswertroutinen sind im Regelfall rekursiv, und somit aufwendiger in Assembler umzusetzen.

Ist der Ausdruck einmal ermittelt, kommt das zweite Problem. Der Absolutwert muss in irgendeiner Form im generierten Code untergebracht werden. Generell bieten sich dazu zwei Möglichkeiten an.

1. Es können die Werte selbst im Opcode-Word untergebracht werden, wenn nicht alle Bit zur Erkennung herangezogen werden, oder
2. ebenfalls denkbar ist eine Erweiterung des Words um ein anderes, indem die Konstante abgelegt wird.

Bei der zweiten Realisierung besteht die Programminformation aus zwei Words, wovon das Erste (vielmehr die ersten vier Bit) die Kennung, und das Zweite die Information darstellt. Nach der Realisierung von Typ a) werden die Informationsträger im Hauptword untergebracht, sonst angehängt.

Unser Motorola Chip benutzt bei Befehlen beide Darstellungsarten.

Mnemonik	Bitkombination (15-0)
TRAP	%010011100100—-
BRA.S	%01100000——–
BSR.S	%01100001——–
BRA	%0110000000000000, 16 Bit Distanzwert
BSR	%0110000100000000, 16 Bit Distanzwert

Es handelt sich dabei als erstes um den TRAP Befehl. Der Aufruf ist bekannt, als Übergabeparameter eignen sich die Werte von $00 bis $0f, denn die CPU erlaubt nicht mehr als 16 Vektoren. Um eine Palette von 16 „Fallen“ zu erlauben, sind 4 Bit $2^4 = 16$) erforderlich. Aufnahmeplatz für diesen Wert findet man in den letzten vier Bit des `TRAP`-Opcodes.

Ebenso befinden sich zwei Relative Sprungbefehle in der Tabelle, BRA und BSR. Sie sind insofern etwas besonderes, als dass sie gleichzeitig zur Kategorie a und b zählen. Bekannt ist, dass bei relativen Sprüngen Kurzformen existieren, wenn der Sprungbereich klein ist. Sie kennzeichenen sich durch die Endung .s aus, und haben eine Ersparnis von 2 Bytes und bringen etwas Geschwindigkeit ein. Bei Nichtbenutzung der Kutzform erweitert sich die Programmlänge um zwei Bytes, der Sprungbereich kann aber auf +-32 KB ausgeweitet werden. Ein Nicht-Shorty kostet also 4 Bytes im Gegensatz zum Shorty mit 2 Bytes.

Die CPU hat jedoch einige Probleme die Relativ-Shorties zu erkennen und sie von weiten Sprüngen zu unterscheiden, denn man sieht, dass der Anfang mit der Bitkombination %01100001 bzw. %01100000 derselbe ist. Um daher einen Shorty zu erzeugen belegt

man die unteren acht Bit des Opcode-Word mit der relativen Adresse und erreicht Sprünge von +-128 Bytes. Ist dieses Byte Null, so handelt es sich um einen weiteren Sprung im Adressbereich von +-32 KB. Die nachfolgenden 16 Bit im Programmcode sichern dann den Distanzwert, wobei der Wert vorzeichenbehaftet sein kann, um eben Sprünge nach vorne und hinten zu erzeugen. (Es wäre ja traurig, wenn man nur nach vorne springen könnte!).

Meines Erachtens ist das Null-Byte aber eine Vergeudung, und es macht mich traurig, dass zur Identifizierung eines Shorties ein ganzes Nullbyte benötigt wird. Ich möchte gern einmal einen Entwickler sprechen, der mir dies erklären kann. Es wäre um einiges sinnvoller gewesen, wenn für weite Sprünge einen Distanzwert von 24 Bit erlaubt würde. Um die Sprungbefehle dann zu unterscheiden zu können müsste man allerdings einen neuen Opcode festsetzen, doch was macht das schon aus, die Nobel-Prozessoren 68040/68050 sind sowieso viel Befehlsbepackter? Der Gewinn dieser Konstruktion wäre fantastisch, und die JMP/JSR Befehle wären somit fast passe. Und noch ein weiterer unverkennbarer Vorteil: Das Multitasking und seine verschiebbaren Programme. Welcher Computer kennt nicht die Last, erst einmal die JMP Adressen umzurechnen, damit das gleiche Programm auch 20 Bytes weiter läuft?

Neben diesen Vorteilen, ein Nachteil für den Prozessor: Er müsste diesen Distanzwert erst ausmaskieren, und die Datenbearbeitung würde zusätzliche Verarbeitungszeit kosten, aber was soll's, ich glaub schon, dass das drin wäre.

Als Assemblerprogrammierer von Assemblern hätten wir auch keine Schwierigkeit, die Befehle mit konstanten Übergabeparametern zu einem Programm anzuhängen. Ein Ablaufschema ist etwas so denkbar:

1. Überspringe Trennzeichen oder Labelnamen
2. Hole Schlüsselwort
3. Handelt es sich um einen „einfachen" Mnemonik?
4. Ja, dann hole den entsprechenden Opcode
5. Hole Ausdruck, evtl. aus interner Tabelle, und berechne ihn
6. Verknüpfe Opcode mit Wert
7. Sichere Wert in adäquater Speicheradresse
8. Erhöhe Zeiger auf freien Speicherplatz um 2 oder 4 Byte

Eine Codegenerierung ist einfach, denn der Absolutwert erscheint in den letzten Bit, sodass auch eine Verschiebung der Konstanten Werte im Opcode entfallen kann. Wenn sich in `A5` ein Zeiger auf freien Speicherplatz befindet, in `D0` der Absolutwert und in `D7` der Opcode, so würden die Zeilen für einen kurzen Sprung folgendermaßen lauten:

```
        bsr     HolAusdruck     ; holt Ausdruck in d0 Register
        beq.s   Fehler
        bsr     BerechneRelaiveAdresse
        or      d0,d7
        move    d7,(a5)+
```

Für einen nicht optimierten Sprung, der einen größeren Bereich einnimmt, könnte das Listing so lauten:

```
        move    d7,(a5)+
        bsr     HolAusdruck
        beq     Fehler
        bsr     BerechneRelaiveAdresse
        move    d0,(a5)+
```

Befehle mit einem Register

Die Technik, wie absolute Zahlen gesichert werden, ist beispielhaft und grundlegend für weitere Befehle, denn anstatt eines Absolutausdruckes kann auch ein Datenregister stehen. Überlegen wir, welche Änderungen notwendig wären. Auszuschließen ist zunächst das Ablegen der Information im zweiten Word, denn um eine Registernummer anzusprechen werden lediglich 3 Bit (2^3 = 8) benötigt, eine Verschwendung von 16 - 3 = 13 Bit wäre Wahnsinn.

Es bleibt die Möglichkeit die Registernummer in drei freie Bit unterzubringen. Ein Freiraum ist in der Tabelle mit drei Strichen gekennzeichnet.

Mnemonik	Bitkombination (15-0)
SWAP	%0100100001000—
UNLK	%0100111001011—

Der Befehl „weiß" dabei selbst, ob ein Adress- oder Datenregister benötigt wird. Eine Zusatzinformation kann demnach also entfallen.

Ein Beispiel für Befehle mit Registern ist der SWAP Befehl. Er verlangt ein Übergaberegister, dessen Inhalt eine Teilword Austauschaktion mitmachen soll. Von den Entwicklern ist jedoch nur ein Vertausch in Datenregistern vorgesehen. Wenn allerdings eine weitere Information (gesetztes, ungesetztes Bit) zur Verfügung stände, so wäre ebenso ein Vertausch im Adressregister denkbar. Die Entwickler planten dies nicht ein, und auch nach reiflicher Überlegung wird man kaum Anwendungen finden, in denen ein Verlangen nach Adressword-Vertauschen aufkommt.

Ebenso ist der UNLK Befehl in der Tabelle aufgeführt, der den mit LINK angelegten Lokalstack aufhebt. Als Übergaberegister wird eins der acht Adressregister erwartet.

Entsprechendes Unterprogramm zur Verwaltung der beiden Befehle wäre schnell formuliert:

```
        bsr     HolRegNr    ; holt Regnummer. ins D0-Register
        beq.s   Fehler      ; z. B. A9/D9, Konstante
        or      d0,d7
        move    d7,(a5)+
```

```
        rts
```

Befehle mit einem Register und folgendem Absolutwert

Nun kommen verzwicktere Befehle, die zwei Übergabewerte verlangen. Dies wird im Folgenden ein Register und ein Absolutwert sein.

Zwei diese Fälle sind in der Auflistung zu finden.

Mnemonik	Bitkombination (15-0)
LINK	%0100111001010—, Word
MOVEQ	%0111—0——–

Eine Mischung von Registerwert und Absolutwert ist LINK, das Gegenstück von UNLK. Einleuchtend, dass auch dieser Befehl ein Adressregister verlangt. Daneben muss für die Einrichtung eines Lokalstacks die Größe des Adress-Offsets angegeben werden. Der Größenbereich bewegt sich innerhalb eines Words, positive und negative Zahlen sind erlaubt, allerdings sind nur negative sinnvoll.

Der angegebene Distanzwert wird nach dem mit Registereintrag versehende Opcode gesichert.

Ein Unterprogrammauszug:

```
        move    #%0100111001010000,d7
        bsr     HolAdrReg
        beq     Fehler
        or      d0,d7
        move    d7,(a5)+
        bsr     HolKomma
        beq     Fehler
        bsr     HolAbsChar      ; folgt ein "#"?
        beq     Fehler
        bsr     HolAbsWord
        move    d0,(a5)+
        rts
```

Anders als beim LINK Befehl werden die Absolutwerte bei MOVEQ gesichert. Durch die Restriktion, dass die Werte nur eine Breite von 8 Bit einnehmen können, ist die CPU in der Lage, diese in den Opcode unterzubringen. Die Vorteile sind: Geschwindigkeit bei der Ausführung und 16 Bit bei der Speicherplatzbelegung. Bei dem MOVEQ finden wir auch eine Eigenschaft wieder, die bei den kommenden Befehlen häufiger zu finden sein wird. Die Registernummer wird ab Position 9 im Word abgelegt. Bei einer Länge von drei Bit bedeutet das einen Belegung von Bit 11, 10 und 9. Für uns Programmierer heißt dies: Das Datenregister Nr. x um 9 Positionen nach rechts schieben. Als ehrgeizige und sparsame Assemblaner wissen wir aber auch, dass siebenmaliges linksrotieren vom Prozessor schneller durchgeführt werden kann als neunmaliges Rechtsschieben. Hier kommt uns die CPU Fähigkeit insofern entgegen, als das die Bit, die rechts herausfallen, links wieder reingeschoben werden.

Explizit ist bei der Parameterübergabe zu beachten, dass die Absolutzahl ein vorzeichenbehaftetes Byte sein kann. Jedoch ist mancher Programmierer fix mit den Fingern, und denkt nicht an eine Beschränkung der positiven Zahlen bis 127, denn Werte, die höher sind, werden von der CPU als negative Zahlen angesehen, und dementsprechend auf negative Longs erweitert (so wie alles auf Longs ausgedehnt wird). Unter diesem Aspekt sollte der Assembler eine Warnmeldung auszugeben, wenn der Bereich von 0-127 überschritten wird. Fehler, die sich bei Zeilen wie

```
MOVEQ #180,D0+
```

einschleichen, erübrigen sich somit. Ist der Assembler „klug“, so erscheint bei Übergabeparametern mit führendem Minus keine Fehlermeldung, obwohl die Werte hoch sind, denn ein negativer Wert war beabsichtigt. Es wäre vergeudete Bildschirmausgabe, wenn der Programmierer bei Ablegung der Booleschen Variablen `TRUE = -1` in ein Register eine Warnung erhalten würde.

Ein Unterprogramm, welches MOVE-Quickies verwaltet, hätte etwa folgenden Aufbau:

```
        bsr     HolAbsChar      ; Abszahl führend mit #
        beq     Fehler
        bsr     HolByte
        beq     Fehler
        or      #%0111000000000000,d0
        move    d0,d7
        bsr     HolKomma
        beq     Fehler
        beq     HolDatenReg
        beq     Fehler
        ror     #7,d0
        or      d0,d7
        move    d7,(a5)+
        rts
```

Befehle mit zwei Registern

Es gibt welche, die haben einen, andere haben zwei davon. Die Rede ist von Registern, die Befehlen übergeben werden. Das heißt allerdings für den folgenden Befehl nicht, dass der zu verarbeitende Wert aus einem Register kommen muss. Um das zu verstehen erst einmal einen Blick auf den Befehl selbst.

Mnemonik	Bitkombination (15-0)
CMPM	%1011—1–001—

Von CoMPareMemory existiert dabei nur die Version:

```
CMPM.x (Ax)+,(Ay)+
```

Jetzt wird verständlich, warum ich Register meinte, aber keinen Registerwert. Da von diesem Befehl sowieso nur eine Indirekt mit Postdekrement Version existiert, können wir uns Angaben über die Adressierungsart sparen. Es ist lediglich von Wichtigkeit, welches Adressregister als Basis genommen wird.

Im Befehl wird die Information folgendermaßen gesichert: Ab Bit 9 wird das Ziel-Adressregister, und ab Bit 0 das Quell-Adressregister abgelegt. Die beiden Bit in der Mitte sind zur Längenangabe verurteilt, die entweder 00, 01 oder 10 ist.

Zum Einlesen der Operandengröße ist ein zweckmäßig eine Funktion zu definieren, die diese Aufgabe übernimmt. Sie könnte etwas wie folgt aussehen, unter der Annahme, dass die Befehle schon erkannt wurden, und der Zeiger auf dem nächsten Zeichen im Quelltext steht. Im Regelfall ist das folgende Zeichen ein Punkt (Operationslänge), ein Semikolon (angefügte Bemerkung), Leerzeichen oder Tabulator (Werte folgen) oder ein Return (bei einem Befehl ohne Parameter).

Das folgende Unterprogramm sucht nach einer Endung, und schreibt die OpLänge als Kürzel ins `D0` Register. Als Pointer auf den ASCII-Text habe ich das `A4`-Register gewählt.

```
HolOpLänge      cmp.b   #".",(a5)   ; ist ein Operand geg.?
        bne.s   GegOp

WordLen addq.l  #1,a5   ; Punkt oder Char "w" weg
        moveq   #%01,d0
        rts

LongLen addq.l  #1,a5   ; "l" überspringen
        moveq   #%10,d0
        rts

ByteLen addq.l  #1,a5   ; "b" überspringen
        moveq   #%00,d0
        rts

GegOP   addq.l  #1,a5   ; Punkt weg
        cmp.b   (a5)+,d0        ; OpLenChar in d0
        cmp.b   #"w",d0
        beq.s   WordLen
        cmp.b   #"W",d0
        beq.s   WordLen
        cmp.b   #"l",d0
        beq.s   LongLen
        cmp.b   #"L",d0
        beq.s   LongLen
        cmp.b   #"b",d0
        beq.s   ByteLen
        cmp.b   #"B",d0
        beq.s   ByteLen
        moveq   #-1,d0  ; Fehler, Ausnahme mit -1
        rts
```

Zum Unterprogramm muss noch folgendes gesagt werden: Da im Übergabeparameter `D0` die Kennzeichnung (00 = Byte, 01 = Word, 10 = Long) stehen soll, kann eine Null nicht für Unbekannt stehen (Im Regelfall liefert eine Funktion als Rückgabeparameter Null für Falsch, -1 (oder 1) für Wahr), da die Kennung für ein Byte notwendig ist. Somit ist es notwendig eine andere Bitbelegung einzuführen. Anbieten würde sich -1. Der Sprung muss dann abgeändert werden, und aus einem „beq Fehler“, wird ein „bmi Fehler“. Zunutze machen wir uns hierbei eine Auswahl von 3 Möglichkeiten: Null, ungleich Null, und negativ (7./15./31. Bit gesetzt). Da der Benutzer durch solcherlei Umdefinierungen - mal ist Null ein Fehler, dann wieder -1 - ins Schleudern kommen kann, sollte dies auch nicht zu häufig angewandt werden.

Auch dieser Befehl kann leicht verarbeitet werden. Ist er erkannt, holt der Assembler die Länge und verknüpft sie. Jetzt muss sichergestellt werden, dass ein "(a" folgt. Bei wahrer Aussage holen wir den Registerwert und verknüpfen ihn mit dem Opcode. Folgt nach der Zahl 0-7 eine Zeichenkette "`),(a" ist die nächste Zahl zu holen, und um sieben Positionen nach links gerollt mit dem Opcode zu verknüpfen. Beim richtigen Abschluss mit ")`" kann die Bitfolge gesichert werden.

Befehle mit zwei Registern und Richtungswechsel

Mit den drei noch kommenden Befehlen verabschieden wir uns von den einfachen Befehlen, deren Umsetzung keine Probleme bereitet. Kleine Routine, sequentieller Ablauf, stimmt etwas nicht, Fehler.

Die Mnemoniks, die noch nachhoppeln sind einfach wie ihre anderen Befehle auch, aber in einer bestimmten Weise auch Wegbereiter für die kommenden. Dies liegt daran, dass die flexibler in ihrer Anwendung sind.

Mnemonik	Bitkombination (15-0)
SBCD	%1000—10000—-
ABCD	%1100—10000—-
EXG	%1100—1——–

ADDX	%1101—1–00—-
SUBX	%1001—1–00—-

Beim SBCD- und ABCD-Befehl fällt zunächst ihre Abstammung auf. Beide sind Funktionen aus dem Bereich der BCD-Verarbeitung. Da dieses Anwendungsgebiet auch nicht allzu oft angeschnitten wird, spendierten die Entwickler auch nicht viele Adressierungsarten. Lediglich zwei Adressierungsarten sind drin.

```
ABCD/SBCD  Dx,Dy
```

oder

```
ABCD/SBCD  -(Ax),-(Ay)
```

Bei Betrachten der beiden Befehle finden wir sieben freie Stellen. Sechs können wir zuordnen. Drei Bit für das Zielregister und drei Bit für das Quellregister ergeben diese Summe. Doch wofür das Übrigbleibende? Da die Information über die Länge entfällt, kann die eine Informationseinheit zur Angabe der Adressierungsart genutzt werden. Es wird definiert, dass ein gesetztes Bit an Position drei als Adressregister, andernfalls bei nicht gesetztem Bit als Datenregister interpretiert wird.

Bei beiden Befehlen erscheint an Position 9 das Zielregister und an Position 0 das Quellregister.

Ebenso durch einen Modus erweitert, präsentiert sich uns der EXG-Befehl. Da beim Vertauschen der Registerinhalte zunächst die zweimal drei Bit zur Registerspezifikation benutzt werden, und fünf andere vordefiniert sind, bleiben fünf freie Bit. Um das zu verstehen, eine Liste der Möglichkeiten, welche Kombinationen möglich sind. Da nur eine Langword-Operation möglich ist, schrumpft die Zahl deutlich.

```
 EXG  Dx,Dy
 EXG  Dx,Ay
 EXG  Ay,Ay
```

Wir sehen: Die Information, durch welche Registerklasssen die Werte vertauscht werden, ist zwingend notwendig. Jedoch bei drei Möglichkeiten die sich ergeben, und dazu zwei notwendigen Bit, werden zur Dekodierung fünf Bit benutzt, gerade die Zahl der Bit, die noch frei sind.

Bit	Modi
010000	D, D
010001	A, A
100001	D, A

Sollte also der Befehl EXG A6,D2 dekodiert werden, ergibt sich:

```
%1100---1--------    Grundbefehl
%----111---------    6
%------------001    2
%--------10001---    Datenregister und Adressregister
-----------------    Summe
%1100111110001001    Ergebnis ist ein exg a6,d2
```

Ebenso bedient sich der ADDX bzw. SUBX Befehl der was-bin-ich-Bit-Technik. Denn wie beim ABCD sind nur zwei Adressierungsarten erlaubt, nämlich

```
ADDX/SUBX  Dx,Dy
```

oder

```
ADDX/SUBX  -(Ax),-(Ay)
```

Ist das achte Bit gesetzt, so wird als Adressierungsart Register Direkt benutzt, andernfalls Adressregister Indirekt mit Dekrement.

An den Bitpositionen 11, 10 und 9 steht das Zielregister, an der Stelle 2, 1 und 0 das Quellregister. Zwei Stellen bleiben jetzt noch unbesetzt, hier wird die Operandenlänge eingesetzt.

Befehle mit einem Register und Spezifikation

Der MOVEQ ist ein oft und gern genommener Befehl. Es zeichnet sich hier schon ab, dass Übergabeparameter auf den Opcode verteilt werden, um Platz zu sparen, und den Prozessor nicht zusätzlich zu einem Lesevorgang zu bewegen, der nur Zeit kostet.

Das Feld mit Registern und Absolutwerten ist mit den beschriebenen Befehlen abgedeckt, und es folgen Opcodes, die neben der Registerzahl und Absolutaufnahmen noch einiges mehr aufnehmen.

Mnemonik	Bitkombination (15-0)
EXT/EXTB	%0100100—000—

Bcc	%0110————
DBcc	%0101—-11001—

So beim ersten Befehl EXT bzw. EXTB, zweitens nur von Prozessoren 68020 und höher unterstützt. Es lässt sich mit diesem Befehl das Ziel vorzeichenerweitern.

Die Registernummer findet wieder Obhut in Bit 2,1 und 0. Wachsame entdecken drei weitere freie Bit an Position 6, 7 und 8. Es ist eine Befehlserweiterung, die sich durch die Anzahl der Striche verrät. Es handelt sich um die Operantenlänge, eine Informationszunahme, die auch bei anderen Befehlen beobachtet werden kann. `EXT` erlaubt die Endung `.w` und `.l`, `EXTB` auch `.l`. Das Ziel ist immer ein Datenregister, Adressregister würden überhaupt keinen Sinn machen. In diesem Freiraum trägt man den Modus ein, der nach folgender Tabelle codiert ist:

010 unterstes Byte wird zum Word	vorzeichenerweitert	Bit 7 wird in die Bit 8-15 kopiert
011 unterstes Word wird zum Long	vorzeichenerweitert	Bit 15 wird in die Bit 16-31 kopiert
111 unterstes Byte wird zum Long	vorzeichenerweitert	Bit 7 wird in die Bit 8-31 kopiert (>= 68020)

Leider benutzt der `EXT(G)`-Befehl eine andere Längenkennung als unser Unterprogramm liefert. Die meisten Befehle, die eine Ergänzung um die Oplänge haben, weisen eine feste Position im Erkennungsword und eine vordefinierte 2-Bit Kennung zur Identifikation der Endung `.b`, `.w` und `.l` auf. Für den `EXT`-Befehl gelten sie jedoch nicht, und müssen umdefiniert werden. Die Vorgehensweise ist einfach:

Das Unterprogramm zum `EXT`-Befehl, welches die Längen benötigt, ruft zunächst einmal `HolOpLänge` auf, in dem die Längen nach Standart ins `D0`-Register geholt werden. Bei erfolgreicher Erkennung dieser kann der `D0`-Inhalt in den Modus umgewandelt werden. Die Vorgehensweise zeigt die kleine Tabelle.

Endung	OpLänge	Modus
.b	00	010
.w	01	011
.l	10	111

Wie immer gibt es mehrere Möglichkeiten die Längen in den Modus umzuwandeln. Als erstes die CMP-Methode, jede Oplänge wird verglichen, und einem Modus zugeordnet, oder der Modus wird aus einer Byte-Tabelle ermittelt, indem die OpLänge der Feldindex ist. (Er eignet sich gut, denn 00, 01 und 10 sind nichts anderes als 0,1,2 in dezimaler Schreibweise.)

Aufgabe: Es soll ein `EXT.L D4` dekodiert werden

Lösung:

```
%0100100---000---    Grundbefehl
%-------------100    4
%-------011------    Word zum Langwort
 ----------------
%0100100011000100    Summe
```

Etwas spannender in der Kodierung wird es mit Befehlen, die eine cc-Endung haben. Als Beispiele für diese Mnemoniks sind in der Tabelle `Bcc` und `DBcc` genannt. Fast schon mit Langeweile können wir die Information entgegennehmen, dass auch die cc's über zugehörige Bitwerte verfügen. Da die Kennung nur 4 Bit lang ist, kann sie im Opcode-Word untergebracht werden.

Bevor es ins Befehlsdetail geht, die Tabelle:

cc	Bitwert
T	0000%
F	0001%
HI	0010%
LS	0011%
CC/HS	0100%
CS/LO	0101%
NE	0110%
EQ	0111%
VC	1000%
VS	1001%
PL	1010%
MI	1011%
GE	1100%

LT	1101%
GT	1110%
LE	1111%

Es fällt auch nach näherem Betrachten kein Zusammenhang der Condition Codes ins Auge. Einzig wären Zusammenhänge in der Bitabfrage zu finden, denn komplexere cc's wie GE, LT, GT und LE, am Ende der Tabelle stehend, reagieren auf mehrere Kondition-Bit. Außer HI und LS machen diese keine anderen Endungen. Ob dies nun als Zusammenhang festgehalten werden kann, muss jeder für sich entscheiden.

Weitsichtig lässt sich ein Unterprogramm formulieren, welches, wie bei den Operationslängen, die Endungen entschlüsselt. Im Folgenden eine Möglichkeit, wie's mit dem Zeiger in A4 auf einen String gemacht werden könnte. Nach dem Verlassen der Routine steht entweder der Bitwert in D0 oder eine negative Zahl, einfacher halber -1, die anzeigt, dass die CC-Kennung nicht gefunden wurde.

```
Holcc   moveq   #0,d0
        move.b  (a4)+,d0        ; 1. Buchstabe
        lsl     #8,d0
        or.b    (a4)+,d0        ; 2. Buchstabe in ein Register
        ; es liegt nun eine 2 Byte Buchstabenkombination vor
        cmp     #"EQ",d0
        beq.s   EQEnd
        cmp     #"NE",d0
        beq.s   NEEnd
        .
        .
        ; es werden alle 2 Byte Endungen überprüft.
        .
        .
        ; wenn nicht dabei, dann vielleicht ein Ein-Byte cc
        subq.l  #1,a4
        cmp.b   #"T",d0
        beq.s   TEnd
        cmp.b   #"F",d0
        beq.s   FEnd

        moveq   #-1,d0
        rts

NEEnd   moveq   #%0110,d0
        rts

EQEnd   moveq   #%0111,d0
        rts
        .
        .       ; u.s.w.
```

Nachdem die Endungen erkannt und die Bitkombinationen in D0 stehen, kann weiter gehen.

Aus den Opcodes lassen sich über die Belegungen wenig Aussagen machen. Es geht nun um die Auflösung. Die cc-Bedingung ist in Bit 11, 10, 9 und 8. Nach erfolgreichem Finden der cc-Endung rollen wir das D0-Register 7-mal nach rechts und or-en es mit %0110000000000000. Fast fertig. Was dann noch folgt ist alter Kram, denn es ist äquivalent zum BRA/BSR Ausgang. Label holen, relativieren, ist er im Bereich +-128 Bytes dann Shorty, wenn nicht dann Lo-Word vom Erkennungsword mit Null belegen, und die Sprungadresse im Folgeword. Ganz simpel.

Extremisten, die die CC-Abfrage-Technik als zu langsam empfinden (ich bin auch so einer!), obwohl die cc's weiter für Scc oder DBcc benutzt werden können, haben die Möglichkeit, die Befehle einzeln zu definieren. Mit einer eigenen Unterprogrammadresse lassen sich dann ein übergeordnetes Programm anspringen, das dann mit den Opcodes die weiteren Verrenkungen macht.

Die Anordnung der Sprungadressen dürfe klar sein, und dass folgende Demolisting zeigt, wie die Verwaltung der einzelnen Bcc-Befehle möglich wäre.

```
BEQ_Up move  %0110011100000000,d7
       bra.s BccUp

BNE_Up move  %0110011000000000,,d7
       bra.s BccUp
BHI_Up move  %0110001000000000,,d7
       bra.s BccUp
BCC_Up move  %0110010000000000,,d7
       bra.s BccUp
       .
       .
       .
BccUp ; wie von BSR/BRA bekannt
```

Machen wir mal einen Aufmerksamkeitsversuch. Man nehme die Bcc-Konstante %0110000000000000. Nun verknüpfen wir dies mit der cc-Kennung „T" (True) %0000. Wir erhalten %0110000000000000. Jetzt schauen und mit schon bekannten Opcodes vergleichen! Und? Ei, da sieh mal einer an, die Kennung stimmt mit der von BRA überein. Dies mag verständlich erscheinen, denn ein BT ist ein BRA, denn da die Bedingung immer Wahr ist, kann auch ein Sprung immer ausgeführt werden. Überhaupt keinen Sinn ergibt sich allerdings, wenn wir %0110000000000000 mit „F" (wie BF) verknüpfen. Das Ergebnis %0110000100000000 stimmt mit dem Opcode von BSR überein. Aber wo liegt da die Logik?

Ein Assembler kennt 16 verschiedene Kondition-Kodes. Mit dem Zweideutigen Condition Code CC/HS bzw. CS/LO kommen wir auf 18 verschiedene Endungen. Sollte man der Einfachheit halber eine Alround-Routine zum Auswerten benutzen, dann muss entsprechend auf den Sonderfall BRA/BT und BSR/BF geachtet werden, denn ein BT ist möglich und bedeutet nichts anderes als ein BRA, aber bei einem BF muss eine Fehlermeldung ertönen, denn ein BSR wird nicht beabsichtigt sein. In einem Unterprogramm zur Auswertung und Weiterleitung kann aufgrund dieses Problems nicht jeder cc unkontrolliert verarbeitet werden. Viel mehr ist darauf zu achten, das ein Fehler, möglicherweise wegen eines Tippfehlers, auffällt.

Nach dem Bcc und auf zum nächsten Befehl, DBcc. Er unterscheidet sich im Aufbau erheblich, einzige Gemeinsamkeit die führende Kennung %01 und die Kondition Kodes an Position 11, 10, 9 und 8. Hier sind nun endgültig alle cc's erlaubt, und einige Programmierer verwenden immer ein DBF, wenn ich stattdessen DBRA schreibe. Aber es ist eine reine Gewöhnungssache, man kann's machen wie man will, es ist sowieso der gleiche Befehl, und wir verstehen jetzt auch warum. Ehrlich gesagt dürfte es einen DBRA auch gar nicht geben, denn RA ist kein cc. Es hat sich allerdings so eingebürgert, so dass unser Assembler hier wieder schlau sein muss.

Da im Gegensatz zum Bcc-Befehl ein Zählregister hinzukommt, muss dementsprechend Platz für 3 Bit geschaffen werden. Sie finden ihren Platz gewohnter weise an Position 2, 1 und 0. Diese Tatsache schließt allerdings etwas aus. Der DBcc-Befehl erlaubt keine Shorties. Die untersten Bit sind belegt, und Platz für 8 Bit findet sich nicht mehr. Die jetzt unbenutzten fünf übrigen Bit werden aber nicht mit Nullen aufgefüllt, sondern mit dem Bitfeld %11001. Dies hat einen Grund. Um den zu verstehen vergleichen wir einmal

```
DBF    D0,      %0101000111001000
```

und

```
SUBQ   #0,A0    %01010001--001000
```

Heu, das wird knapp. Um die Spannung in die Höhe zu treiben: Wie werden wohl die zwei Bit vom Subi noch zu belegen sein? Nun, die Information über den Suffix fehlt noch. Da die Länge nur 00, 01 oder 10 sein kann, bleibt nur noch 11, um die beiden Befehle zu unterscheiden. Es liegt nun nur am gesetzten sechsten Bit, ob ein `DBF` oder `SUBQ` (noch mit ihrem Gedöns dran) compiliert wird. Dies zeigt die wichtige Aufgabe eines Programmierers, auf die Bit zu achten, und keinen Fehler im Abschreiben (!) zu machen. Das testen des Assemblers dauert dementsprechend lange, denn Verknüpfungen müssen einfach stimmten, und es darf nicht vorkommen, dass hin und wieder ein Bit nicht ausmaskiert wurde.

Befehle mit Effektiver Adresse

Bisher waren die Befehle von keiner komplexen Struktur. Wir hatten keinen Übergabeparameter, wenn, dann einen, selten zwei, und dann einfache, durchschaubare. Durch eine Modusangabe wurden die Befehle dann um Befehlsmöglichkeiten erweitert.

Wenn wir uns im Gegensatz zu den einfachen Befehlen einmal die Konstruktionen mit komplexeren Adressierungsarten, wie

```
IVU    D0,10(A0,D4)
MOVE.L (A0)+,18(a6)
```

anschauen, so kommt etwas zum durchscheinen, was bei allen Mnemoniks bisher noch nicht auftrat. Bisher waren die Adressierungsarten nur Register direkt, oder Adressregister indirekt. Mit den vielfältigen Befehlsmöglichkeiten ändert sich dies, und die Information über diese Adressierungsart muss ebenso wie die Registernummer im Opcode gesichert werden.

Neben den phantastischen Eigenschaften der CPU alle Register gleichberechtigt zu halten, sinkt man auch nicht so auf das Intel Niveau herab, wo noch immer gilt: Lieber ein Register zum Zählen, eins zum Rechnen, und eins, wo wir unsere Adressen ablegen. Die Stärke, die unser Motorola Chip aufweist, ist besonders die Orthogonalität. Bei den meisten Befehlen können mehrere Adressierungsarten angewendet werden, die Register sind allgemein, spezielle Registerbindungen an Befehle gibt es nicht.

Einmal bei den Adressierungsarten angelangt kommt man zu dem Schluss, dass die Mnemoniks kombiniert mit den Adressierungsarten sehr viele Möglichkeiten bilden. Und dann kommt man zu dem Schluss: Da muss eine Technik hinter stecken! Denn so wie früher mit Modi-Bit ein oder zwei Adressierungsarten zu unterscheiden, ist absurd.

In den ersten Kapiteln des Buches wurden die verschiedenen Adressierungsarten ausreichend erläutet, sodass nun die Thematik behandelt werden kann, wie sie dekodiert werden.

Wenn die Antwort Modi-Bit lautet (gesetztes Bit, Möglichkeit b, sonst a) liegen wir schon ganz gut. Das Verfahren zur Unterscheidung der Adressierungsarten ist aber etwas komplexer und ausgedehnter, denn nun gilt es nicht mehr unter Ja/Nein Zuständen zu unterscheiden, sondern unter mehreren Adressierungsarten zu unterscheiden.

Die eigentliche Information über die Adressierungsart liegt in drei bzw. sechs Bit verborgen. Die ersten drei Bit werden Modus genannt, und durch die Angabe einer Zahl von %000 (0) bis %111 (7) angegeben. Die zweiten drei Bit nehmen, wenn möglich, Register auf. Wenn keine Register-Informationen vorliegen erweitern die drei Bit den Modus. Zusammengesetzt aus Modus plus Register ergeben die sechs Bit eine Kombination, den man Effektive Adresse (EA) nennt.

In der Tabelle wird dieser Zusammenhang allerdings besser sichtbar, denn sie beschreibt mögliche Adr. Arten und ihre Bitbelegungen in Modus und Register.

Adressierungsart	Mode	Register	Aufbau
Datenreg. direkt	000	Nr.	Dn
Adressreg. direkt	001	Nr.	An
Adressreg. indirekt	010	Nr.	(An)
Adressreg. indirekt mit Postinkrement	100	Nr.	(An)+
Adressreg. indirekt mit Predekrement	101	Nr.	-(An)
Adressreg. indirekt mit	110	Nr.	d16(An)

Distanzwert			
Adressreg. indirekt mit 8-Bit Wert und Reg.	110	Nr.	d8(An,Rn)
Absolut kurz	111	000	xxxx.w
Absolut lang	111	001	xxxx.l
PC indirekt mit Distanzwert	111	101	d16(PC)
PC indirekt mit 8-Bit Wert und Reg.	111	101	d8(PC,Rn)
Direkt	111	100	#xxxx

Zur Übung basteln wir ein paar EAs zusammen:

Beispiel	Bitwerte
`D1`	000001
`(A2)`	010002
`#12`	111100

Bei den Befehlen, die in der Tabelle folgen werden, handelt es sich nur um Mnemoniks die eine EA Angabe verlangen. Diese EA-Angabe spezifiziert selbstverständlich ein Ziel, und keine Quelle, denn Befehle, die nur eine Quelle haben, aber kein Ziel, gibt es nicht.

Mnemonik	Bitkombination (15-0)
NEGX	%01000000–––
CLR	%01000010–––
NEG	%01000100–––
NOT	%01000110–––
TST	%01001010–––
TAS	%0100101011––
JSR	%0100111010––
JMP	%0100111011––
NBCD	%0100100000––
PEA	%0100100001––
ASR	%1110000011––
ASL	%1110000111––
ASR	%1110001011––
ASL	%1110001111––
ROXR	%1110010011––
ROXL	%1110010111––
ROR	%1110011011––
ROL	%1110011111––

Die effektive Adresse benötigt sechs freie Bitplätze. In der Praxis sieht es so aus, dass die Opcodes sechs freie Bit am Ende aufweisen (Bit 5-0).

Wenn in der Tabelle neben den sechs freien Positionen noch zwei weitere auftauchen, dann sind Längenangaben erforderlich. Diese werden wie üblich in Bit 6 und 7 gesichert.

Wenn ein Befehl in seine Bitwerte umgesetzt werden soll, müssen alle Einzelinformationen, d. h. Informationen über das benutzte Register, die Operandenlänge, zusammengefügt werden.

Als Beispiel wähle ich folgenden Befehl, den wir von Hand kodieren wollen:

```
CLR.L (A5)+
```

Aufgespaltet in die einzelnen Komponenten ergibt sich:

```
%01000010--------   Grundbefehl
%--------10------   Länge ist Long
----------010---   Modus
%-------------101   Register
 ----------------
%0100001010010101   Ergebnis
```

Ein Assembler versucht nun den Vorgang soweit zu automatisieren, dass ein Unterprogramm Hauptarbeiten abnimmt, so z. B. EAs auswerten. Es werden zur Generierung und Eintragung des Opcodes weitere Unterprogramm genutzt, die aber im einzelnen nicht weiter beschrieben werden.

Für ein Unterprogramm, das ein `CLR` auswertet, könnte der Code etwa so aussehen.

```
UP_Clr  move    #%0100001000000000,d7
        bsr     HolOpLänge
        bmi     Fehler
        ror     #7,d0
        or      d0,d7
        bsr     MergeZielEA
        beq     Fehler
        rts
```

Das Prinzip ist nun für alle anderen Befehle dasselbe. Man überträgt den Opcode in ein Register (ich habe `D7` gewählt), holt die Länge über das Unterprogramm HolOpLänge und verknüpft diese mit dem Opcode. Im Unterprogramm `MergeZielEA` wird dann die EA geholt und mit dem Opcode verknüpft.

Selbstverständlich ist diese Art das Unterprogramm zu gestalten noch nicht das gelbe vom Ei. Daher ist man geneigt, noch einfacher vorzugehen, soweit, dass man keine Unterprogramme mehr für die einzelnen Befehle hat, sondern nur noch Tabellen, in denen der Compiler eine Adressierungsart erlaubt, oder ausschließt. Für Befehle, die komplexer sind, z. B. `MOVEM` kann immer noch auf die Möglichkeit zurückgegriffen werden.

Etwas abgesondert in der Tabelle stehen die Rotations- und Schiebebefehle. Für sie gilt: Das Ziel ist eine EA, und als Quelle wird kein Wert angegeben, denn die Bewegung ist immer konstant eine Position.

Der Befehl wirkt kurz und knapp, etwa:

```
LSL D0
```

Bei machen Schreibweisen muss der Assembler aber „schlau“ sein. Das untere Beispiel macht dies deutlich.

```
LSL #1,(A0)
```

Aus dieser eigentlich unerlaubten Zeile (wenn ein Absolutwert abgebenden ist, darf das Ziel nur ein Datenregister sein) muss folgende übersetzt werden.

```
LSL (A0)
```

Um noch einmal klarzustellen: Die besprochenen Verschiebe- und Rotationsbefehle, die mit einer EA arbeiten, nehmen keine Absolutwerte auf, und können auch um keine bewegen. Ihre Änderung ist konstant auf einmaliges Bewegen festgelegt. Jeder Befehl dieser Form ist 2 Bytes lang.

Apropos Bytes und Opcode. Nach einem Blick in die Tabelle finden sich sofort Auffälligkeiten in den Bewegungsbefehlen. Denn in Bezug auf ihre Adressierungsarten sind sie gleich, und warum sollte man da nicht Zusammenhänge finden? Zunächst folgendes: Bit 3/4 enträtselt den Bewegungsbefehl wie in der Tabelle angegeben.

Bit	Befehl
00	ASx
01	LSx
10	ROXx
11	ROx

Da es nur in zwei Richtungen abgeht, kann man sich vorstellen, dass auch hier ein Bit als Richtungsweiser verwendet wird. Und so ist es. Es gesetztes achtes Bit lässt die Wirkung nach links erfolgen, und ein ungesetztes Bit lässt nach rechts bewegen. Mit den drei Bitinformationen lassen sich die acht Scroller abdecken. Jede der drei Bit wird genutzt, und 3^2 verrät ebenso die Anzahl.

Nicht alle Adressierungsarten sind erlaubt

Vorsicht ist die Mutter der Programmierer. Die in der Tabelle aufgeführten EAs sind alle möglichen, aber nicht unbedingt erlaubten. So ist es z. B. nicht gestattet ein Absolutwert als Ziel zu nehmen. Ein Beispiel macht dies deutlich:

```
CLR.B #14
```

Diese Kombination ist natürlich unmöglich.

Ebenso ist es nicht vorgesehen, das `Ax`-Register als Ziel zu benutzen.

```
TST.B A0
```

Um dies EA als Ziel zu vermeiden legen wir uns ein Unterprogramm mit dem Namen `MergeZielEA` an, das dann nur Effektive Adressen holt, die auch als Ziel erlaubt sind.

Wir müssen dieses Unterprogramm allerdings für viele Fälle erweitern. So ist es z. B. ebenso unzulässig

```
JMP D7
```

zu schreiben. Zu diesem Zweck ist es unumgänglich ein Blick auf die Tabelle mit erlaubten Adressierungsarten zu werfen.

Wie kann nun der Assembler wissen, ob eine EA erlaubt ist, oder nicht? Zum Vergleich der PD Assembler A68k. Auch wenn er in C geschrieben ist, verdeutlicht und beschreibt er uns eine tolle Möglichkeit, Effektive Adressen von Befehlen zuzulassen oder nicht. Er geht dabei von vordefinierten Befehlskombinationen aus, und gibt ihnen Namen. Durch eine Tabelle, die zu jeden Befehl einen Eintrag mit einer Information über erlaubt-nicht-erlaubt-EA hat, ist der Assembler in der Lage zu entscheiden, ob eine richtige Anwendung der Adressierungsart vorliegt. Stößt A68k auf einen Befehl, so wir mit dem Verknüpfungswert die EA verglichen, und bei fehlerhaftem Auftreten eine Meldung über die falsche Anwendung ausgegeben.

Schwierigkeiten für „Assembler“-Programmierer

Den Assemblerprogrammierern von Assemblern stellen sich in der Umsetzung von ASCII Zeichenkette, mit seinem Anhängsel (Operandenlänge, Register, Absolutwerte) in die EA, unzählige Probleme in den Weg.

Ein Beispiel aus der Praxis:

```
move d0,D4
```

Bei der Variablenzuweisung von dem Register `d0` nach `D4` hat unser Assembler ein kleines Problem, denn er kann das Ziel nicht eindeutig festlegen. Was ist `D4`? Ein Register oder eine Variable? Kompliziert, kompliziert. Auch unter der Annahme, dass alles, was mit einem `d`/`D` oder `a`/`A` beginnt als Register zu nehmen ist, scheitert bei der folgenden Zeile:

```
move d9,d7
```

Auch wenn `d9` als Variable deklariert wäre, würde nach unser oben genannten Möglichkeit ein Fehler ausgegeben, denn `9` ist kein mögliches Register. Überhaupt träte ein Fehler auf, bei Variablen, die mit `d` oder `a` begännen. Dies ist bitter, und ein Assembler, der nur „schöne“ Variablen zulässt wäre von keiner großen Bedeutung.

Wie ist nun eine Überprüfung möglich, ob es sich um eine Variable, oder ein Register handelt? Nun, das auf "d" oder "a" folgende Zeichen muss eine Ziffer im Bereich von "0" bis "7" sein. Erst wenn dies gewährleistet ist, und die Registerspezifikation mit einem Trennzeichen endet, kann von einem Register ausgegangen werden.

Überprüfen wir dies praktisch:

- "t8": Es handelt sich um kein "d" oder "a", also Variable.
- "a9": Mit einem "A" beginnend stehen die Chancen gut. Die folgende Variabelenprüfung hält der Ausdruck allerdings nicht stand. Da das folgende Zeichen zwar eine Ziffer ist, diese aber nicht im Bereich $0 < x < 7$ ist, handelt es sich um eine Variable.
- "a08": Die Ersterkennung war erfolgreich, und auch die nächste Ziffer ist im korrekten Bereich. Da allerdings kein Trennzeichen folgt, handelt es sich auch hier um eine Variable.
- "d6,+": Der Anfang ist wieder gut, und die folgende Ziffer ist im richtigen Bereich. Da auch ein Trennzeichen folgt, ist die Überprüfung abgeschlossen, es handelt sich um ein Register.

Da auch Return, Leerzeichen, Tabulator oder Klammer Trennzeichen sind, dürfen sie auf keinen Fall vergessen werden.

Das Problem mit der vorher nicht ermittelbaren Quelle, es kann auch das Ziel sein, muss weiter verfolgt werden, denn es bedeutet für einen Programmierer eines Assemblers einen großen Programmieraufwand. Warum?

Ein Blick auf die folgende Zeile macht die Problematik noch einmal deutlich.

Beispiel Nr. 1: `clr.b D2`

Was ist `D2`? Ein Register wird man sagen. Man kann auch einfach von einem Register ausgehen. Wenn man dies so interpretiert, wird der Befehl 2 Bytes lang.

Beispiel Nr. 2:

```
D4      dc.b 12
        ...
Up      clr.b D4
```

Jetzt sieht die Sache etwas anders aus. Nun muss `D4` als Speicherstelle interpretiert werden, nicht mehr als Register. Das bedeutet, dass der Befehl nicht mehr 2 Bytes lang ist, sondern durch die Speicherzellenangabe auf 6 Bytes anwächst.

Beim ermitteln den effektiven Adresse ist also die Speicherstelle höherwertiger als das Register. Es muss daher zuerst nach einer möglichen Speicherstelle Ausschau gehalten werden, die so wie ein Register aussieht. Wie ein mögliches Register aussieht wurde ja ein paar Zeilen früher ermittelt.

Der Assembler ist gut dran, denn der Label ist zuerst definiert, und dann erst in einer EA benutzt worden. So kann die ermittelte Adresse dann gleich eingesetzt werden, und ein Befehl mit 6 Bytes Länge entsteht. Was ist aber, wenn der Label erst später eingesetzt wird, wie in Beispiel 3:

```
Up      clr.b D4
        rts

D4      dc.b 12
```

Beim Assembliervorgang gelangt der Assembler an diese Stelle, und weis nicht, was er machen soll, denn `D4` ist noch nicht definiert worden. Was folgt daraus: Assemblerprogramme sind nur in zwei Pässen voll übersehbar. Im ersten Pass werden die Befehle auf ihre Syntax hin überprüft (oder auch nicht, kommt drauf an), und Labeladressen ermittelt. Das dauert natürlich. Im zweiten Pass können die Labels nun eingesetzt werden.

Wir kann man nun einen schnellen Assembler programmieren?

Frage: Der Vorgang dauert lange, wie ist es möglich, einen schnellen Assembler zu programmieren?

Wir gehen dazu dreimal durch den Programmtext, haben also einen 3-Pass Assembler. Die Pässe übernehmen folgende Aufgaben:

Pass 1:

Im ersten Durchlauf werden nur Labelnamen in einer Tabelle eingetragen. Dies geht ziemlich schnell, denn es muss nur überprüft werden, ob am Anfang ein Trennzeichen ist (Befehl), oder nicht (dann ein Label). Es werden nur die Labelnamen eingetragen, keine Adressen oder ähnliches.

Pass 2:

Der zweite Pass übernimmt die Aufgabe des Compilieren, also des Codeerstellens. Nun ist dem Assembler bekannt, ob Label vorliegen. Demnach kann bei Konstruktionen wie `clr d9` ein Fehler ausgegeben werden. Im zweiten Durchlauf wird die Tabelle mit den Labeladressen gefüllt. Eine Labeladresse, die ein Mnemonik benötigt, kann nun bekannt sein, oder nicht, es kommt ganz darauf an, ob er schon eingetragen wurde, oder nicht. Wenn die Labeladresse O.K. ist, wird sie aus der Tabelle kopiert, die Benutzung kam also nach dem Eintragen. Im Normalfall dürfe es aber so sein, dass für Variablen Platz unter dem Programm angelegt wird. Da davon die Adresse nicht bekannt ist, legen wir einen Zeiger auf den unbearbeitbaren Befehl und den Zeiger auf den Aktuellen Programmtext in einer zweiten Tabelle ab. Da nur die Adresse fehlt, kann Null oder etwas anderes eingetragen werden. Jetzt kann fortgefahren werden.

Pass 3:

Im letzten Assemblerschritt wird die Tabelle abgerast, in der die Zeiger auf die unbearbeitbaren Zeilen liegen, die wegen unermittelbarer Labeladressen zurückgestellt wurden. Da nach dem zweiten Pass alle Adressen bekannt sind, gehen wir nicht mehr den ganzen Programmtext ab, sondern nur die Stellen, die durch die Tabelle gegeben sind. Der Geschwindigkeitszuwachs ist groß!

Nach diesem Prinzip kann ein Assembler arbeiten. Ich selber habe einen Assembler in Assembler programmiert (aber nur mit zwei Pässen, wenn ein `D2` auftrat, und `D2` war ein Label, habe ich dies nicht erlaubt, sondern `D2` als Register angenommen). Nach den ersten Testläufen hüpfte ich wegen der Geschwindigkeit zwar nicht gerade an die Decke, konnte jedoch Compilerzeiten feststellen, die 6-mal schneller waren als Devpac, und 1,5-mal schneller als AsmOne. Das ist dann schon erfreulich. Da ich jedoch jeden einzeln Befehl übersetzte, war die Arbeit zu groß, und ich habe das Projekt eingestellt, vielleicht gäbe es ihn sonst als PD-Assembler, und er könnte den etwas langsamen A68k ablösen.

Leider gibt es den leistungsfähigen TurboAss (Atari-PD) nicht auf dem Amiga.

Befehle mit EA und Register

Da wir die EA nur als Ziel kennengelernt haben, ist es nun an der Zeit sie auch als Quelle einzusetzen. Die nächsten Befehle sind ein Mischmasch aus welchen, die EA als Ziel oder auch als Quelle einsetzen.

Mnemonik	Bitkombination (15-0)
BTST	%0000—100——
BCHG	%0000—101——
BCLR	%0000—110——
BSET	%0000—111——
CHK	%0100————
LEA	%0100—111——
DIVS	%1000—111——
DIVU	%1000—011——
MULS	%1100—111——
MULU	%1100—011——
CMP	%1011———— (?)
CMPA	%1011———— (?)
ASR	%1110—0–100—
ASL	%1110—1–100—
LSR	%1110—0–101—
LSL	%1110—1–101—
ROXR	%1110—0–110—

ROXL	%1110—1–110—
ROR	%1110—0–111—
ROL	%1110—1–111—

Bei Betrachtung fällt durchgehend ein Loch an der Position 9 auf. An dieser Stelle wird die Registernummer eingetragen, die mit dem Befehl gekoppelt ist. Es ist jedoch nicht sichergestellt, ob das Register als Ziel oder Quelle eingesetzt wird. Die Vergewisserung kommt dann beim Blick in die Tabelle.

Gönnen wir uns einen Blick „in“ die Befehle. Die Bit-Manipulations-Befehle (mit Ausnahme der relativen Sprungbefehle beginnen alle mit einem „B“) haben durchgehend an der Position 9 ihr benutztes Datenregister, das hier als Quelle benutzt ist. Bitveränderer, die Absolutwerte bearbeiten, folgen im kommenden Kapitel; sie belegen die drei Bit an der Position 9 anders.

Ausschlaggebend für den Bitmanipulationsbefehl ist somit Bit 6, 7 und 8. Zur Verdeutlichung ein Blick in die Tabelle.

Bitbelegung	Befehl
100	BTST
101	BCHG
110	BCLR
111	BSET

Auffallend sind ebenso die dritte und vierte Grundrechenart. Bei der Multiplikation und Division werden lediglich Datenregister als Zieloperand zugelassen, die EA findet Verwendung als Quellspezifikation.

Ob die Operation signed oder unsigned durchgeführt wird, liegt am gesetzten bzw. ungesetzten 8 Bit.

Beim LEA Befehl ist, ebenso wie bei den oben genannten Befehlen, als Ziel nur ein Register erlaubt. Das Register ist natürlich kein Datenregister, sondern ein Adressregister. Wenn dies nicht so wäre, würde die Abkürzung auch nicht „Load Effective address to Address Register“ heißen!

Nun, der LEA-Befehl nimmt also die EA auf, allerdings nicht alle, denn der Befehl ist wählerisch. So sind folgende Kombinationen unerlaubt (Möglichkeiten durch Schrägstrich getrennt):

```
LEA Dx/Ax/(Ax)+/-(Ax)/#xxxx,Ay
```

Es sind also nur EAs erlaubt, die indirekt sind, und keine Adressmodifikationen anleiten, wie die Dekrements.

Der CHK-Befehle, dessen Benutzung sowieso im User-Betrieb eingeschränkt ist, erlaubt es den Mnemonik-Generatoren, verschiedene Längen anzugeben. Doch für Prozessoren, die unter dem 68020 liegen, ist nur die Wordbreite benutzbar, die die Längenkennung %110 erhält. Die Privis (Leute, die privilegiert sind) können auch auf Langword Checken. Sie haben dann die Kennung %100 einzusetzen.

Bei den beiden Compare-Befehlen, die in der Liste auftauchen, sind nur die ersten vier Bit belegt. Die restlichen 12 haben somit andere Bedeutungen. Wie gewohnt finden wir unsere EA an Pos 0-5. Es sind alle Adressierungsarten erlaubt. Bei beiden Befehlen, CMP und CMPA gilt, dass das Ziel nur ein Datenregister oder ein Adressregister sein kann. Der Registerwert findet sich an Position 9 und belegt wie gewohnt 3 Bit. Übrig bleibt nun ein Loch von 3 Bit an der Stelle 6.

Die freien Bit bestimmen den Modus der Befehle. Da die Operandenlänge noch nicht erwähnt wurden, ist nachvollziehbar, dass sie durch den Modus repräsentiert wird. Leider sind die Modiwerte für die Befehle `CMP` und `CMPA` ungleich. So gilt für den CMP Befehl folgende Modus-Bitbelegung:

Länge	Modus-Bit
.b	000
.w	001
.l	010

Ein Beispiel zur Dekodierung:

```
CMP.L -(A2),D8
```

```
%1011------------   Grundbefehl
%-------010------   Länge ist Long
%----------100---   Modus
%-------------010   Register
%----xxx---------   Fehler, da unerlaubtes Register!!
```

→ Abbrechen

Für den CMPA-Befehl gilt:

Länge	Modus-Bit
.w	011
.l	111

Eine Byte Angabe ist nicht erlaubt. Es würde auch keinen Sinn machen. Bei Word-Angaben wird vorzeichenrichtig auf ein Long erweitert, und dann verglichen.

Befehle mit EA und Absoluten

Ebenso wie ein Datenregister als Ziel oder Quelle eingesetzt werden kann, kann auch ein Absolutwert benutzt werden. Dies jedoch, ganz klar, nur als Quelle.

Mnemonik	Bitkombination (15-0)
ORI	%00000000———–
ANDI	%00000010———–
SUBI	%00000100———–
ADDI	%00000110———–
EORI	%00001010———–
CMPI	%00001100———–
SUBQ	%0101—0———–
ADDQ	%1101—0———–
ASR	%1110—0–100—
ASL	%1110—1–100—
LSR	%1110—0–101—
LSL	%1110—1–101—
ROXR	%1110—0–110—
ROXL	%1110—1–110—
ROR	%1110—0–111—
ROL	%1110—1–111—

Es sind eine ganze Menge an Befehlen. Sie stammen aus dem Bereich mathematischer Operationen bzw. Rotationen, und auch vom Vergleich.

Beim ersten flüchtigen Blick, ist durchgehend eine 2-Bit-Lücke an Position 6 auffallend. Hier wird die OpLänge gesichert. Die Rotations- und Schiebebefehle werden mit keiner Längeninformation versehen, da die Rotation von Speicherinhalten, die durch die EAs ja ermöglicht werden, nur auf Word Basis erfolgt.

Bei den Befehlen `ORI` bis `CMPI` lautet die Endung immer „I“, ein Anzeichen, dass Immediate, also unmittelbare Werte benötigt werden. Der Assembler benötigt jedoch im Regelfall keine Endung auf I, um zu Entscheiden, dass ein Immediate Wert eingesetzt wird. Er kann vielmehr selbst auf eine Unterroutine verzweigen, die die Absolutzahlen auswertet, und sie mit anderem Opcode schreibt.

Für alle `xx(x)I`-Befehle gilt, dass außer die EA „Adressregister direkt“ alle weiteren benutzbar sind. Zeilen wie

```
subi #12,a2
```

sind nicht möglich, denn das Ziel ist ein Adressregister, uns somit kommt ein anderer Befehl zum Tragen, der SUBA lautet. Er wird im folgenden Kapitel besprochen, da er neben dem Registerwert und der EA noch die OpLänge kodiert aufnimmt.

Bei den Rollern muss der Absolutwert angegeben werden. Da sie maximal 8 Positionsverschiebungen erlauben, ist die Anzahl der Bitverschiebungen durch 3 Bit ($2^3 = 8$) darstellbar. Wir kennen aus dem vorigen Kapitel die Verschiebebefehle mit zwei Registern. Platz für die Quelle ist an Position 11, 10 und 9, für das Ziel sind die Bit an Position 3, 2, 1 bedeutungsvoll. Da zum Bewegen der Werte um eine konstante Zahl kein zweites Datenregister benötigt wird, können die Quell-Bit, Bit 11, 10, 9 den Absolutwert aufnehmen.

Wenn man nun die 2-Register-Bewegebefehle und die 1-Reg-1-AbsWert-Bewegungsbefehle vergleicht, so kann man bei Betrachtung von Bit 5 folgendes feststellen: Ist Bit 5 gesetzt, so ist das Ziel ein Absolutwert, andernfalls ein Register.

Von den Schiebe- und Rollbefehlen gehen wir weiter zu den anderen. In der EA-Befehls-Auflistung finden sich zwei sehr häufig benutzte Mnemoniks, die besonders durch die Lücke bei 11, 10 und 9 auffallen. Es sind die gern genommenen Mathe-Quickies. Da sie auf alle „normalen“ EAs anwendbar sind, und eine OpLänge aufweisen, ist das Lo-Byte schon abgeharkt. Doch Achtung, eine Sache darf nicht unerwähnt bleiben: Wenn die Operantenlänge Byte ist, darf die EA nicht Adressregister indirekt sein!

Bleibt nur noch das High-Byte, das unklar ist. `SUBQ` und `ADDQ` unterscheiden sich durch ein 15. gesetzte Bit im OpCode. Die vier Bytes für die Kennung sind somit auch geklärt. Letztendlich ist noch ein Platz für 3 Bit an der Position 9, 10 und 11. Klar, was da rein kommt. Der konstante Wert. Leider kann der Wert nur eine Zahl sein, die durch drei Bit darstellbar ist, also eine Zahl aus dem Bereich 0...7.

Eine Addition (oder Subtraktion) mit (oder von) Null ist jedoch wenig sinnvoll. Besser wäre es, wenn eine Addition mit Null gleich einer Addition mit Acht wäre. Es entspricht z. B. bei den Verschiebe- und Rollbefehlen der Wert Acht gleich dem 3-Bit-Wert 000, aber acht Verschiebungen.

Der mathematische Koprozessor

Freunde von Raytracing-Bildern verzweifeln häufig, wenn die Feinberechnung von Bildern Zeitspannen von Stunden einnehmen. Schuld daran, ist nicht nur die Taktgeschwindigkeit der Prozessoren, die mit 8 MHz reichlich unterbemessen ist, hauptsächlich verantwortlich für die langen Wartezeiten, sind die komplizierten trigonometrischen Berechnungen. Um Abhilfe zu schaffen kann nur bedingt mit schnelleren Prozessoren entgegenwirkt werden, denn diese müssen die Berechnungen immer noch selber ausführen. Die Algorithmen sind zwar ausgeklügelt, jedoch immer noch zu langsam, um vernünftige Ergebnisse in angemessener Zeit zu erwarten. Aufgrund dessen, kam die Idee, die Algorithmen auf einem Chip zu integrieren, und dann eine hardwaremäßige Berechnung durchzuführen. Die Zusatzchips, auch Koprozessoren genannt, die sich auf die Aufgabe spezialisiert haben, mathematische Operationen zu erledigen, heißen MC68881 und im Nachfüllpack MC688821. Die Chip-Kollegen, die Speicher oder Ein- Ausgaben verwalten, sind somit grundlegend anders.

Zur Namensgebung muss gesagt werden, dass sich MC68881 und der MC68882 nicht so grundlegend unterscheiden wie z. B. MC68000 und Aufsteiger MC68020. Der mathematische Koprozessor (FPU, Float-Pointig-Unit oder FPP, Float-Pointing-Processor) MC68882 unterscheidet sich nur insofern vom Vorgänger, dass er mit einer höheren Taktgeschwindigkeit von 33 MHz gegenüber 25MHz beim MC68881 gefahren werden kann. Eine weitere Änderung (die ernorm Geschwindigkeit bringt) ist eine Art Pipline Verfahren (CU, Conversion Unit), die Befehle schon liest, während die APU (Arithmetic Processing Unit) (vergleichbar mit der ALU im Prozessor) noch rechnet.

Die FPU Formate

Nicht nur der Befehlsatz eines Prozessors ist wichtig, auch die Möglichkeiten, verschiedene Datenformate zu benutzen, ist hörenswert. Manche werden ein Lied davon singen können, wie umständlich es sein kann, Zahlen, die vom C-Compiler benutzt werden in das Motorola Format umzuwandeln. Die Formate IEEE und FFP bekriegen sich ein wenig.

Im Folgenden sollen die Formate kurz vorgestellt werden.

Integer Data Format

Normales 680x0 Format, dass das höchste Bit (das kann das 32-ste, 16-te oder 8-te Bit sein) zum Vorzeichenbit ernennt.

Binary Real Data Format

Bei diesem Format, welches etwa die dtsch. Entsprechung „Binäres Realzahlen-Format“ hätte, werden entweder 32 Bit (Single Real), 64 Bit (Double Real) oder 80 Bit (Extended Real) zur Darstellung benutzt. Die interne Verwaltung der Formate wird durch die normalisierte Darstellung erreicht, was bedeutet, dass die Zahlen immer zwischen 1 und 2 liegen.

Packed Decimal Data Format

In diesem dezimal gepackten Realformat werden die Zahlen als 24 Zeichen gepackter String (BCD Ketten) verwaltet. Gepackt deshalb, weil sich zwei Zeichen ein Byte teilen müssen.

Wer jetzt denkt alle diese Formate sind hardwarewemäßig realisiert, der irrt, denn angesichts der Komplexität und verschiedenen Verhaltensweisen ist es nahezu unmöglich auf einem begrenzen Chipplatz alle Formate und dessen Operationen zu implementieren. Die FPU wandelt alle Werte die ankommen in Standarisierte Werte um. Der Standart, der für die Fließkommazahlen von Bedeutung ist, ist der IEEE Standart. Nach ihm ist z. B. die Aufteilung der Mantisse und viele Operationen genormt. Intern ist die Ablegung der Zahlen, mit der Verteilung der Bit, wie folgt geregelt.

Typ	Vorzeichen	Exponent	Mantisse	Summe
Einfache Genauigkeit	1	8	32	32
Doppelte Genauigkeit	1	11	52	64
Erweiterte Genauigkeit	1	15	64	80

Befehlssatz

Kommen wir in diesem Kapitel zum Eingemachten, der Programmierung der FPU. Wie andere Prozessoren auch, verfügt auch sie über einen Befehlssatz. Dieser ist nicht, wie man vielleicht denken könnte, recht klein (wie nur +, -, *, /, sin, cos) sondern relativ groß. So hat der Anwender eine Auswahl von 64 Mnemoniks, um sein Programm zu verfassen. Da sehr komplexe Befehle enthalten sind, sollte man schon den gesamten Befehlssatz kennen, um optimal arbeiten zu können.

Die Befehle können von ganz unterschiedlicher Bedeutung sein. So wie es vergleichsweise Datentransportbefehle, arithmetische und logische Befehle usw. beim 680x0 gibt, lässt sich auch eine Klassifizierung der FPU Befehle finden. Diese lassen sich allgemein in fünf Gruppen einteilen.

- Datentransportbefehle
- Monadische Operationen (ein Operand)
- Dyadische Operationen (zwei Operanden)
- Programmsteuerbefehle
- Sonderbefehle zur Systemsteuerung

Wir wollen diese Kategorien folgend einzeln besprechen.

Datentransportbefehle bewegen

1. Daten aus dem Speicher in die 80-Bit breiten Register
2. Daten aus dem Speicher in ein CPU-Register
3. Daten aus dem CPU-Register in ein FPU-Register
4. Daten aus FPU-Register in den Speicher

Der Befehl, der die Übertragung einleitet heißt `FMOVE`, und hat jeweils eine Quelle und ein Ziel. Die Quelle kann eins der acht FPU Register sein, die über `FP0` bis `FP7` angesprochen werden können, oder auch eine ganz normale Adressierungsart, wie wir sie aus dem ersten Kapitel schon kennen. Etwa:

```
FMOVE   d0,FP0 ; D0 wird als Word in FP0 geladen
FMOVE.S FP2,2(a3)
```

Doch die Angabe der Richtung reicht noch nicht aus, vielmehr muss der Mathematische Koprozessor noch wissen, in welchem Datenformat die Werte übertragen werden sollen (intern werden sie nur in einem Format bearbeitet. Zu den bekannten Operandenlängen `.b`, `.w` und `.l` kommen noch `.s` für das Single Real Format , `.d` für das Double Real Format, und `.x` für das Extended Real Format dazu. Des Weiteren existiert noch die Endung `.p` für das gepackte Feld, jedoch muss bei Benutzung dieser Darstellungsart hinter der Zielangabe in geschweiften Klammern und `#` noch ein sogenannter Formatparameter angegebnen werden. Dieser ist im Zweierkomplement anzugeben, und kann Werte zwischen -64 und 17 annehmen. Die angegebenen Werte werden formatiert, und zwar bestimmen die negativen Zahlen des Formatparameters die Anzahl der Nachkommastellen, und die positiven Werte die Anzahl Vorkommastellen, z. B. `FMOVE.P FP1,Speicherstelle{#3}`.

Bei diesem Beispiel wird der Wert im FPU-Register `FP7` als packed decimal in (die) Speicherzelle bewegt. Es werden drei Stellen angenommen. Sollte die Zahl beispielsweise 12,345678 lauten, dann würde das Ergebnis 1.23E2 sein.

Konstante Werte sind immer verfügbar

Aus alten C-64 Zeiten kennen wir als Assembler-Programmierer sicherlich noch die internen ROM Tabellen, wo so interessante Werte wie Pi, Log, e, u. ä. drin war. Diese Konstanten dürfen natürlich auch bei professionellen Prozessoren nicht fehlen. Daher existiert eine Tabelle im ROM der FPU, die wichtige Konstanten enthält. Um sie anzusprechen gehen wir über den Befehl +FMOVECR.X + und einem Offset an die Werte ran. Im Folgenden zeigt die Tabelle, welche wichtigen Werte mit welchen Offsets verfügbar sind.

Offset	Konstante
$00	Pi, 3.14159265..
$0e	lg(2), 0.30102..
$0c	e, 2.718281828..
$0d	lb(e), 1.44269..
$0e	ln(e), 0.43429..
$0f	0.0
$30	ln(2), 0,69314..
$31	ln(10), 2.3026..
$32	10^0
$33	10^1
$34	10^2
$35	10^4
$36	10^{16}
$37	10^{32}
$38	10^{32}
$39	10^{64}
$3a	10^{128}
$3b	10^{256}
$3c	10^{512}
$3d	10^{1024}
$3e	10^{2048}
$3f	10^{4096}

Da sich nicht alle Werte aus dem Speicher oder aus der Tabelle nehmen lassen, kommen wir zu den Berechnungen, eigentlich die Hauptaufgabe des Mathematischen Koprozessors. (Die Tabellen sind ein gute Ergänzung.)

Monadische Operatoren

Zuerst wären da einmal die monadischen Operationen. Man findet bei ihnen nur einen Funktionsparameter. Dieser wird verarbeitet, und es folgt, wie beispielsweise bei der Addition, keine Verknüpfung der Werte. Die 25 monadischen Mnemoniks sind in der folgenden Tabelle aufgelistet.

Befehl	Bedeutung
FABS	Positiver Absolutwert
FACOS	Arcus Cosinus, arccos(x)
FASIN	Arcus Sinus, arcsin(x)
FATAN	Arcus Tangens, arctan(x)
FATANH	Arcus Tangenshyperbolicus, arctanh(x)

FCOS	Cosinus, cos(x)
FCOSH	Hypercosinus, cosh(x)
FETOX	e^x
FETOXM1	e^x -1
FGETEXP	Exponent der Zahl
FGETMAN	Mantisse der Zahl
FINT	ab- oder aufrunden auf die nächste ganze Zahl
FINTRZ	abrunden auf dir nächste ganze Zahl
FLOGN	Logarithmus zur Basis e, ln(x)
FLOGNP1	ln(x-1)
FLOG10	Log. zur Basis 10, lg(x)
FLOG2	Log. zur Basis 2, lb(x)
FNEG	Negieren
FSIN	Sinus, sin(x)
FSINH	Hyperbelsinus, sinh(x)
FSQRT	Quadratwurzel, sqrt(x)
FTAN	Tangens
FTANH	Hyperbeltangens, tanh(x)
FTENTOX	10^x
FTWOTOX	2^x

Dyadische Operatoren

Aber was haben wir davon, wenn wir Werte berechnen können, diese aber nicht mittels der Grundrechenarten verknüpfen können? Hinzu kommen noch Befehle, die zwei Operanden erlauben und somit Verknüpfungen ermöglichen. Sie heißen dyadische Operatoren. Bei diesen Operationen kann die Quelle entweder ein 68020 Register, eine Speicheradresse oder ein FPU Datenregister sein. Allerdings haben sie auch eine Restriktion in der Handhabung, denn bei den zwei Übergabewerten kann das Ziel nur ein Floatpointing-Register sein. Dies ist deshalb notwendig, da auch das Ergebnis immer in dem FPU Register erscheint. Der Koprozessor bietet uns 10 dyadische Befehle.

Befehl	Bedeutung
FADD	Addition, Quelle + FPn (*)
FCMP	Vergleich, FPn - Quelle (*)
FDIV	Division, FPn / Quelle (*)
FMOD	Ganzzahl, FPn MOD Quelle (*)
FMUL	Multiplikation, FPn * Quelle (*)
FREM	Differenz von FPn zum nächst höheren Vielfachen (*)
FSCALE	FPn * INT(2^{Quelle})
FSGLDIV	Division (Ergebnis im Single Real Format)
FSGLMUL	Multiplikation (Ergebnis im Single Real Format)
FSUB	Subtraktion, FPn - Quelle (*)

(*) von IEEE gefordert

Natürlich reichen auch diese Befehle nicht aus, um einen abgerundeten Programmfluss zu erreichen. Daher kommen noch Befehle hinzu, die Verzweigungen erlauben und Exceptions, aufgrund der diversen gesetzten Statusbit, auslösen können.

Das Statusregister

Vergleichbar mit dem Statusregistern der 680x0, wo z. B. das Negativ, Null Flag, o. ä., von Befehlen verändert werden, muss auch bei Matheprozessoren diverse vergleichbare Bit existieren. Das Statusregister hat den Namen Float Point Status Register (FPSR), und ist auch im Programm über dies verfügbar. Ein Auslesen wäre z. B. mit dem Befehl `FMOVE FPSR,D0` möglich.

Aufgrund einer Menge von Möglichkeiten, die zur Verfügung stehen, beschränkt sich die Größe nicht auf ein paar Bitchen, sondern umfasst schon ein Longword, also 32 Bit, die komplett genutzt werden. Doch was viel wichtiger ist: Wie heißen denn die einzelnen Bit, und welche Funktion haben sie?

- Die Bit 24...31, also das höchstwertige Byte, wird als Floating Point Condition Code Byte bezeichnet. Die Abkürzung für dieses Monstrum ist FPCC. Sie entsprechen den Bedingungscodes der mathematischen Operationen. Allerdings sind nur vier von Belang, und somit sind die restlichen vier Bit mit Nullen aufgefüllt.
- Die Bit 16..23 erhielten den Namen „Quotient Byte“. Gesetzt wird das Byte am Ende der FMOD und FREM Operation und fasst den Quotienten auf, den die Operation lieferte. Wie bei Zahlen mit Vorzeichen werden die niedrigsten sieben Bit als Zahl (der Quotient) genommen, und das übergebliebene als Vorzeichenbit genutzt.
- Die Bit 8..15 sind die Exception Status Bit (EXC). Da bei verschiedenen Operationen Exceptions ausgelöst werden können, können hier die gesetzten Bit ausgelesen werden. Wie die Belegung dieser Bit im Einzelnen ist, erfahren wir gleich, denn es gibt ein spezielles Register namens FPCR, mit den diese Bit gesetzt werden können. Da sie die gleiche Reihenfolge und Bitbelegung haben, wollen wir sie gleich besprechen.
- Die letzten Bit 7..0 haben die Aufgabe, dem Benutzer ein Auftreten von Exceptions zu melden. Der Vorteil dieses Bytes, mit dem kunstvollen Namen AEXE (Accrued Exception Byte), ist, dass der Programmierer von langen Rechenoperationen sich ein dauerndes Überprüfen der EXC-Bytes sparen kann, und erst nach Ablauf der Termberechnung nachprüfen muss, ob ein Fehler, sprich Ausnahmebedingung auftrat. Damit dies allerdings auch funktioniert, muss dem Prozessor verboten werden, Exceptions auszulösen.

Bitnummer	Bedeutung
31-28	Null da nicht belegt
27	Negativ-Bit, Negatives Resultat
26	Null-Bit, Ergebnis = 0
25	Infinity-Bit, Resultat ist Unendlich
24	Ergebnis keine gültige Zahl
23	Vorzeichen des Quotienten
22-16	Quotient
13	Branch/Set fehlerhaft
14	Keine Nummer
13	Operand fehlerhaft
12	Übertrag
11	Untertrag
10	Division durch Null
9	Unerlaubte Operation
8	Unerlaubte Dezimaleingabe
7	Unzulässige Operation
6	Übertrag
5	Untertrag
4	Division durch Null
3	Unerlaubte Dezimaleingabe
0-2	Nicht besetzt, Null

Im letzten Abschnitt sind wir bei den Exception stehengeblieben. Man kann der FPU nur durch Setzten der Bit mitteilen, bei welchem Ereignis eine Exception ausgelöst wird. Dazu schauen wir uns mal die Tabelle an, die angefügt ist. Sollte ein Bit dieses Registers gesetzt sein, wird beim Auftreten dieses Falles eine Exception ausgelöst. In diesem Ausnahmezustand kann dann weiters zur Sicherung vorgenommen werden.

Letztendlich geben die Bit 0-7 das Mode Control Byte an, mit dem verschiedene Rundungsprozesse angeleitet werden können. Der Programmierer kann dabei zwischen vier grundlegenden Prozessen wählen:

Runden

- zur nächstliegenden Zahl
- zur nächst kleineren Zahl
- zur nächst größeren Zahl
- gegen Null

Zusammengefasst werden diese Bit als *Floating Point Contol Register* (FPCR) bezeichnet. Wie schon erwähnt finden wir die Bit 15..8 in dem Flot Pointing Status Register (FPSR), gleiche Stelle, gleicher Ort, gleiche Zeit, wieder.

Bitnr.	Abkürzung	Bedeutung
15	(BSUN)	branch/set on unorderd
14	(SNAN)	signaling not a numbler

13	(OPERR)	operand error
12	(OVFL)	overflow
11	(UNFL)	underflow
10	(DZ)	division zero
9	(INEX2)	inacceptable operation
8	(INEX1)	inacceptable decimal input
7-6		precision
5-4		mode
3-0		zero

Condition Codes

Der Zusammenhang zwischen FPCR und FPSR ist geklärt, gehen wir nun einen Schritt weiter. Die CPU enthält Mnemoniks, womit der Programmablauf verändert werden kann. Dazu dienen Branch-Befehle, die in Kombination mit den Condition Codes zu Labeln verzweigen können. Es gibt entsprechend, wie bei den schon besprochenen Befehlsendungen der CPU (unter cc bekannt) auch Bedingungen, mit denen die FPU unter Kontrolle gebracht werden kann. Es lassen sich dabei Vergleiche zwischen den cc's der CPU und der FPU anstellen, und stellt dabei fest, dass zwei ähnliche Mnemoniks existieren. Diese zwei FPU cc's unterscheiden sich allerdings grundlegend, denn beim ersten Typ wird aufgrund eines ungültigen Operation eine Exception ausgelöst, und beim zweiten keine, auch wenn ein NAN, d. h. eine ungültige Operation, im Spiel war. Diese Trennung kann im Zusammenhang mit Programmiersprachen sinnvoll sein, denn es kann schnell aufgrund einer Fehlerquelle auf eine Auswertroutine verzweigt werden, die z. B. dem Benutzer mitteilt, dass ein Fehler auftrat. In der folgenden Tabelle sind die Endungen mit cc1 gekennzeichnet, die eine Exception auslösen, und die mit cc2, die eben keine zur Folge haben.

Die Mnemonik-Endungen lauten wie folgt:

cc1	cc2	Bedeutung
GE	OGE	größer gleich
GL	OGL	ungleich (größer oder kleiner)
GLE	OR	größer, kleiner oder gleich
GT	OGT	größer
LE	OLE	kleiner gleich
LT	OLT	kleiner
NGE	UGE	größer gleich, NAN Bit muss gesetzt sein
NGL	UGL	größer kleiner, mit NAN
NGLE	UR	größer, kleiner oder gleich, mit NAN
NGT	UGT	größer, mit NAN
NLE	ULE	kleiner gleich, mit NAN
NLT	ULT	kleiner, mit NAN
EQ	SEQ	Werte gleich
NE	SNE	Werte ungleich
F	SF	Immer falsch
T	ST	Immer wahr

Natürlich existieren auch CPU ähnliche Befehle, die mit den Condition Codes Programmverzeigungen anleiten können. Sie sind so aufgebaut wie die bekannten, und in der Schreibweise unterscheiden sie sich nur durch ein führendes „F".

Befehl	Operand	Bedeutung
FBcc	<label>	wie Bcc (spring nach Label, wenn cc wahr)
FDcc	Dn,<label>	wie DBcc
FScc	<ea>	wie Scc (cc wahr, dann <ea>=-1, sonst 0)

Zusätzlich zu den Sprungbefehlen können noch Zahlen getestet, und Zeitspannen gewartet werden. Der Schreibunterschied ist auch hier nur im „F".

Befehl	Operand	Bedeutung
FTST	<ea>	teste Zahl und aktualisiere FPSR

FNOP		faulenzen

Letztendlich bietet und der komplexe Baustein FPU auch noch Befehle zur Systemsteuerung.

In der vorigen Tabelle war gerade die Rede von Exceptions. Diese können auch mit einem FTRAPcc-Befehl eingeleitet werden. Die cc's sind der Tabelle zu entnehmen.

Die FPU im Multitasking

Eine sehr interessante Erweiterung der FPU ist die Fähigkeit den Multitasking Betrieb zu unterstützen. Wenn ein, zwei oder drei (oder ...) Tasks sich der FPU bedienen wollen, so muss sich einer entscheiden, der tatsächlich zugreifen kann, die anderen müssen warten. Das Problem bei der Sache ist nur, wenn lange Operationen anstehen, muss die eine Berechnung unterbrochen werden, und die anderen kommen zum Zuge. Die Unterbrechung kann im Programmablauf Probleme bereiten, denn Condition Zustände werden unter Umständen nicht in die neuen Operationen hineingezogen, und das kann Schwierigkeiten mit sich bringen, insofern, dass Überläufe oder unerlaubte Divisionen nicht korrekt verarbeitet werden. Um diesem Problem zu entweichen existieren zwei Befehle FSAVE und FRESTORE, die FPU Zustände (Register) sichern und wiederherstellen können. Sollte nach Bitten ein Task dem Wunsch eines anderen Tasks nachkommen, die FPU freizugeben, können die internen Zustände der FPU mit einem Befehl im Speicher abgelegt werden (man bezeichnet diese abgelegten Bereiche als State Frames), der andere Task holt die Register aus dem Speicher in die FPU und das Spiel geht von vorne los.

Zu diesem Thema ist noch das Floating Point Instuction Adress Register zu nennen. In FPIAR (dies ist die Abkürzung) wird die Adresse gesichert, bei dem die FPU stehenblieb. Dies hat praktische Gründe für den Programmierer, denn wie soll er (ohne Megacode) bei einer Exception die Adresse des letzten Befehls ermitteln?

Doch was für viele interessant sein dürfte, ist die Kommunikation mit diesem Chip. Leider können 68000-Prozessoren nur sehr umständlich auf den Befehlssatz zugreifen. 68020er aufwärts haben es entschieden einfacher, da sie Befehle zur Kommunikation mit Koprozessoren bieten. Letztendlich verschmelzen beim 68030 die Befehle so, dass man gar nicht mehr weis, wer letztendlich etwas macht. Bei diesem Prozessortyp, ist die FPU direkt über 32 Bit Leitungen mit der CPU verbunden, sodass dadurch allerschnellste Übertragungen von Daten möglich sind.

Interner Aufbau der Befehle

Die Formate und Befehle sind kurz angeschnitten worden, und in diesem Kapitel wollen wir uns dem Befehlssatz des Mathematischen Koprozessors etwas intensiver widmen. Wenn wir allerdings von einem vordefinierten Befehlssatz zum Ansprechen der FPU ausgehen, sind wir auf der falschen Spur. Vielmehr hat Motorola ca. 20 % Mikrocodeplatz auf dem 68000 freigehalten, um später noch einige Erweiterungen einzubauen. An Ideen mangelte es nicht, und unsere Intel Konkurrenz zeigt, wie man's auch machen kann. Sie hängen zwar mit allen anderen Befehlen kräftig nach, jedoch bieten sie Mnemoniks zur Stringverarbeitung. So ist das Suchen nach einem String und vergleichen von Strings eine Leichtigkeit. Auch Speicherkopierbefehle (wie sie selbst der Z80 bietet!) sind im Intel Chip verfügbar. Diesem Befehlsverlust sollte schon nachgetrauert werden, doch was soll's, man kann da nichts machen. Der Kopf der Entwickler war aber noch voller, und Gedanken an einer Aufpepplung der Matheroutinen (32-Bit Multiplikation = 64-Bit Ergebnis), Bitfelder, schnellere Divisionsalgorithmen gab es. Doch leider fuhr auch bei diesen Ideen der Zug schneller ab. Vor lauter Frust an die schönen Dinge, die noch hätten implementiert werden können, schufen sie eine Schnittstelle, die es erlaubt, zu späteren Zeiten noch Befehle einzubauen. Da jeder Befehl eine führende 4 Bit Kennung (Operationscodes) hat, muss einfach eine Bitkombination gefunden werden, die von keinem Mnemonik genutzt wird. Bei den Möglichkeiten $2^4 = 16$ (das kommt einen sehr wenig vor) werden zwei ausgeschlossen. Dies ist der Operationscode 1010 und 1111. Bekannt sind beiden Werte allerdings ehr als Line-A und Line-F-Emulation ($axxx und $fxxx), die binär eben 1010 und 1111 ergeben. Keiner der bekannten Befehle vergreift sich an diese armen Kennungen, sie unterscheiden sich alle in den ersten vier Bit. Was passiert nun, wenn der Prozessor merkt das man ihn linken will, und der Befehl überhaupt nicht existiert? (Ein Tätä gibt er bestimmt nicht aus!) Um sich bemerkbar zu machen hüpft er in den Ausnahmezustand, also einer Exception. Diese Exception hat wie jede, ihre eigene Adresse. Sie ist für $axxx bei $28 und für $fxxxx bei $2c zu finden. Jeder findige Programmier wird jetzt die Idee schüren: „Tolle Sache, damit kann ich endlich meine xy ungelöst Funktion aufrufen". Grundsätzlich ist die Idee nicht schlecht, aber was machen wir denn, wenn die Funktionscodes in späteren Chip Versionen vergeben werden? Leicht sind so Wege verbaut, denn läuft einmal eine Routine, und dann teilen die Motarolaner den Softwareherstellern mit, dass die beiden Operationscodes für Befehlserweiterungen benutzt werden, ist es schwierig alte Pogramme am laufen zu lassen, Abstürze sind vorprogrammiert. So wird für 68020'er die Kennung $fxxx als Zugriffscode für Koprozessoren aller Art angesehen. Beim entdecken der %1111 Bitkombination freut sich die CPU immer auf die Koprozessor Befehle, die ankommen, denn so kann er sich wieder auf die faule Haut legen. Haben wir allerdings einen Koprozessor und keinen 68020 aufwärts, dann müssen wir etwas umständlicher zugreifen, denn in der Exception, die aufgrund des nicht Vorhandenseins ausgelöst wurde, muss ein Koprozessor Interface emuliert werden. Verdutzt schauten auch unsere Atari-Entwickler drein, denn in alten TOS-Versionen (vor unendlich geraumen Zeit) haben sie die Möglichkeit der Line-F-Emulation benutzt, um Betriebssystemaufrufe zu verarbeiten. Das war's erstmal mit der einfachen Ansprechung. Das Gewurschtel wurde dementsprechend groß und die Übersicht unübersichtlich. Solange nicht auch noch die Line-A-Befehle vom Prozessor neu benutzt werden, kann auch der Blitter im Atari ruhig weiterschnurren. (Wäre es so aufwendig, kompliziert, unschön, ..., gewesen, wenn die Entwickler eine einfache Unterroutine benutzt hätte? (So wie im Amiga OS!))

Die Befehle, die die Koprozessoren ansprechen, sind Line-F-Befehle. In einer nach dem Auftreten ausgelösten Exception können sie ausgewertet werden, und somit ihrer speziellen Arbeit nachgehen. Bewusst habe ich bisher Koprozessor geschrieben, und nicht FPU, um die es hauptsächlich gehen soll. Bei einem tollen 68020-Chip, lässt sich nicht nur ein Koprozessor ansprechen, sondern gleich acht! Zu den vordefinierten Bit %1111, die eine Koprozessorbenutzung einleiten (WICHTIG: Nur wenn wir einen haben kann auch einer angesprochen werden (logisch, nicht?), sonst Exception!!), folgen noch drei Bit an den Positionen 11-9, welche die Koprozessor-Identifikation angegeben. Reserviert wurden von Motorola die CP-Kennungen %000-%101, wovon %000 für die MMU 68551 und %001 für unsere FPU draufgehen. Und da haben wir was wir brauchen. Die ersten 4 + 3 = 7 Bit sind dementsprechend mit %1111001 vorbelegt.

Doch bleibt die bohrende Frage: „Und? Wie geht's denn jetzt weiter, mit einem $fxxx kann man doch keine vier Grundrechenarten angeben?" Eine weitere Differenzierung ist in weiteren Bit-Angaben zu finden. Bei vier Grundoperationen brauchen wir zur Darstellung auch nur 2 Bit, denn $2^2 = 4$. Es ist definiert, dass die Bit

00	eine Addition,
01	eine Subtraktion,
10	eine Multiplikation und
11	eine Division

einleiten. Neben dieser grundlegenden mathematischen Festsetzung, was eigentlich für eine Operation gewünscht ist, ist ebenso eine Quell- und Zielangabe von Nöten. Einigen wir uns der einfacher halber auf Datenregister, die zur Aufnahme des Quell- und Zieloperanden benutzt werden. Um die Fülle von acht Registern anzusprechen benötigen wir 3 Bit, nach der alten Rechenregel, $3^2 = 8$. Wie im vorigen Kapitel entspricht dementsprechend eine Drei-Bit-Kombination von 000 dem Datenregister `D0` und eine Full-House-Kombination von 111 dem Register `D7`.

Jetzt ist auch alles Nötige für die Operation festgesetzt, und wir errechnen die Summe der Einzelbit, die letztendlich benutzt werden. Dies gibt nach Aufaddierung ein Ergebnis von 4 + 2 + 3 + 3 = 12 Bit. Was bleibt sind vier „übrige“ Bit (16 - 12 = 4), die für unsere Zwecke nicht benutzt werden. Die nachstehende Tabelle gibt die Verteilung der Bit im Word an.

Bit Nr.	Bedeutung
15-12	Kennung, muss für Line-F %1111 sein
11-9	Koprozessorkennung, vordefiniert %001
8-5	??????
4-3	Angabe der Rechenart
2-0	Quellregister

Gehen wir noch einmal zur Wiederholung alle Fälle durch:

1. Haben wir einen 68020 aufwärts mit Matheprozessor, wird $fxxx als Anspruch auf den Koprozessor verwaltet. Eine Exception wird nicht ausgelöst, die Befehlskennung wir „richtig“ verarbeitet, und sofort an die FPU weitergeleitet, die dann die Rechenoperationen anleiert. Eine Unterbrechung der CPU ist nicht die Folge, und der Programmfluss kann flüssig erfolgen.
2. Sollte allerdings kein Mathekoprozessor mit dem 68020 verbunden sein, so wird auch hier eine Exception ausgelöst, in der dann von Hand eine Emulation programmiert werden muss. Dies ist in der Exception möglich
3. Wenn ein „billiger“ 68000, der über keine Koprozessor-Befehle verfügt, mit einem tollen 68882 kommunizieren soll, dann ist dies nur über ein genormtes Protokoll möglich. Da die CPUs bis 68020 über kein Koprozessor-Interface verfügen, muss die Übertragung von Hand gemacht werden.
4. Privilegierte 68040-Besitzer haben mit diesem Chip-Model einen Prozessor erworben, der sagenhaft in der Ausführung ist. Sie spart sich die Arbeit über den Bus die FPU zu bemühen und führt selbst grundlegende Operationen, d. h. +, -, *, / aus. Berechnungen werden somit noch schneller im inneren des 68040 ausgeführt. Weitere Berechnungen im FPU-Bereich führt die CPU allerdings nicht durch. (Sonst wäre der Mathematiker ja auch unterbelastet!)

Umstieg Atari auf Amiga, Systemvergleich

Dieses Kapitel widme ich drei Gruppen. Zum einen Entwickler, die bestehende Atari Software auf dem Amiga umsetzen wollen (gibt es die?), dann dem Atari-Programmierer, der sich nach reiflicher Überlegungen einen Amiga zulegte und jetzt gerne entsprechende Routine kennenlernen würde, und schließlich einer großen Masse Amiganer, die zwar über Ataris lästern, aber keine Ahnung vom Betriebssystem haben. Für die, die noch nie mit dem Atari gearbeitet haben, werden ich das TOS des Ataris vorstellen, um Tendenzen zu entwickeln, wo das AmigaOS besser ist, und (oder) wo das Atari TOS Vorteile hat. Die Atarianer meinen immer noch, dass wir keine Ahnung von ihren Rechnern haben, zeigen wir ihnen, dass es doch so ist!

Der Streit über den besseren Rechner wird es noch lange geben, versuchen wir ihn sachlich zu führen. Als ich einst in der Mailbox meinen Senf zum Amiga zugab, und ihn, zugegebenermaßen, etwas hoch lobte, musste ich einiges an Schlägen (auch unter die Gürtellinie) einstecken. Die Reaktion war von Spinner bis Ahnungsloser, arroganter Besserwisser. Na ja, so was tut auch weh.

Das Betriebssystem des Atari, TOS

Das TOS (gern Tramiel [ja, ja, der wollte den Amiga für’n Appel und e’n Ei kaufen] Operation System genannt) besteht aus drei Teilen, dem

1. GEMDOS (TRAP #14),
2. BIOS (TRAP #13) und
3. XBIOS (TRAP #14).

Das GEMDOS ist das eigentliche Betriebssystem, es kennt alle Funktionen, die wir von einem DOS her kennen, eben Funktionen zum Behandeln von Dateien. Dazu hilft ihm das BIOS, das „Basic Input Output System“. Da diese Funktionen allein nicht reichen, wird es vom XBIOS, dem „eXtend BIOS unterstützt“.

Die Graphische Oberfläche unter GEM

Der Atari Rechner hatte wohl als erstes mit eine graphische Oberfläche. Größtenteils von Mac’s abgekupfert, findet sich eine Implementierung von GEM, die auch später auf dem PC Kariere machte. Nachdem allerdings Windows in den PC Markt Einzug, finden wir von GEM nichts mehr. Auf dem Atari allerdings existiert es weiter, und es wurden immer mehr Erweiterungen beigefügt.

GEM steht für Graphic Enviroment Manager und ist eine grafische Umgebung. Im Einzelnen spaltet es sich in zwei Hälften, dem

1. VDI (Virtual Device Inferface) und dem
2. AES (Applikation Enviroment System).

Das VDI enthält grundlegende Grafikroutinen, wie Linie zeichnen und Kreise füllen, und das AES stellt die Übergeordneten Elemente wie Fenster und Box dar. Das AES nutzt dafür natürlich die Funktionen des VDI.

Um die Funktionen der VDI- und AES-Library aufrufen zu können, ist wie immer ein Trap-Aufruf gefragt. Besetzt für GEM-Aufrufe alle Art, ist der TRAP #2. Im Datenregister `D0` wird nun dem Betriebssystem mitgeteilt, welche der beide Hälften benutzt wird. Ist `D0` = $c8, dann ist das AES gefordert, andernfalls bei der Konstanten $73 das VDI. `D1` ist ein Zeiger auf ein Datenfeld, das u. a. auch den Opcode der spez. Funktion enthält. Wir nennen das Datenfeld AES- oder VDI Parameterblock. Der Opcode identifiziert,

wie beim AmigaOS der Offset, die Funktion. So ist der Opcode für eine Funktion, die einen Text ausgibt z. B. 8 und der, der ein Fenster öffnet 101. Der Opcode ist im Control-Array des Parameterblocks enthalten.

Sofort können Parallelen zum AmigaOS entdeckt werden. Was beim AmigaOS die `intuition.library` übernimmt, regelt bei GEM die AES-Library. Grafikausgaben laufen beim Amiga über die `graphics-`, `diskfont-` und `layers.library`, der Atari spricht das VDI an. Größere Unterschiede sind beim Aufruf der jeweiligen Funktion zu finden. Rufen wir beim Amiga eine Library-Funktion auf, so werden alle Übergabeparameter, wie Koordinaten, Zeiger usw. über Register übergeben, der Aufruf geschieht letztendlich über die Adressierungsart -xxxx(a6). Atari bzw. GEM regelt den Aufruf anders, alle Übergabeparameter werden in ein Datenfeld abgelegt, und der GEM-Funktion muss lediglich ein Zeiger auf den Datenblock übergeben werden. Der Aufruf über den TRAP-Befehl finde ich persönlich etwas veraltet, eine Exception aufzurufen und dadurch einen Sonderzustand zu verursachen, eine schlechte Lösung. Na ja, der Apple-Mac macht's auch so, und der ist immer noch Vorbild.

VDI (Virtual Device Inferface)

Das VDI als Ausgebende Grafikeinheit hat so einige Features parat, wovon Amiga Benutzer nur träumen können. Denn das VDI ist unabhängig vom Grafikausgabegerät. Dies ist was feines, die Ausgaben können auf dem Monitor erfolgen, auf dem Plotter oder Drucker, und sogar in eine Datei umgeleitet werden. Diese Datei wird Metafile genannt. Es ist eine Datei mit objektorientiertem Format, die Daten liegen also nicht bitmaporientiert vor, sondern á la BASIC-Programm. Metafiles können anschließend auch auf Peripheriegeräte "kopiert" werden, das Ausgabegerät erstellt dann aufgrund der Daten die Grafik. Anwendungsbeispiel DTP: Ein Rechner ohne Drucker erstellt eine Datei, dieses wird weitergereicht, und evtl. auch einem anderen Computer unter GEM laufend (wenn man so etwas noch findet!) ausgegeben. Bei PC's läuft dies ähnlich unter Post-Script, die Befehle können auch als Programm in eine Datei umgeleitet werden, und dieses kann später auf einen Post-Script-Drucker ausgegeben werden, der dann die Grafik erstellt.

Die Unabhängigkeit der Grafikausgaben wird durch verschiedene Gerätetreiber unterstützt. Doch Gerätetreiber reichen ja allein nicht aus, vielmehr müssen diese auch angesprochen werden. Wenn das VDI folgende Aufgabe erhält: "Zeichne Linie", so muss zuerst abgefragt werden: "Wohin eigentlich?" Es muss also eine Einheit unter AES geben, die sich mit der Verwaltung der einzelnen Grafikausgabegeräte beschäftigt. Diese Einheit heißt

GDOS (Graphic Device Operating System)

GDOS verwaltet also die Gerätetreiber. Wenn auch alle Programmierer GDOS verwenden würden, wäre ja eine optimale Benutzerfreundlichkeit gegeben. Doch viele finden die Steuerung über das GDOS zu kompliziert, und auch die Langsamkeit trägt nicht gerade zur Verbreitung bei. Doch durch das neue GDOS Speedo unter NVDI wird dies hoffentlich der Vergangenheit angehören.

GDOS hat allerdings immer noch nicht den Kontakt zu den Peripheriegeräten, er hantiert allenfalls mit Geräteadressen. Die unterste Ebene ist das

GIOS (Graphic Input/Output System)

GIOS enthält die gerätespezifisch I/O-Funktionen. Der Programmierer kommt allerdings nicht mit dem GIOS in Berührung, alles läuft über das GDOS. Die Gerätenummer des gewünschten Gerätes muss GDOS übergeben werden, und dann wird mit dem dazu existierenden Gerätetreiber das GIOS angesprochen. GIOS ist also Schnittstelle zwischen I/O-Geräte und GDOS.

Interessant ist, wie die geräteunabhängige Programmierung erreicht wird. Dazu existieren zwei Koordinatensysteme:

1. NDC (Normalized Device Coordinates) (0,0)(32767,32767)
2. RC (Raster Coordinates) (0,0)(640,400)

Mit dem NDC wird Geräteunabhängigkeit erreicht, die übergebenden Koordinaten werden geg. auf das Ausgabemedium umgerechnet. RC ist allenfalls für das Ansprechen des SW-Bildschirm wichtig.

VDI-Funktionen

Das grundlegende Grafiksystem besteht, wie bereits gesagt, aus fundamentalen Grafikroutinen, die geräteunabhängig sind. Ungeachtet dessen lassen sich Oberbegriffe nennen, in denen die Funktionen eingeteilt werden können.

Kontroll-Funktionen	dienen der Initialisierung der Applikation
Ausgabe-Funktionen	kümmern sich um die grafischen Grundoperationen wie Linien und Kreise
Attribut-Funktionen	setzen die Attribute für Farbe, Schrifttyp und Stiftnummer
Raster-Funktionen	ermöglichen Operationen auf Speicherbereiche wie Kopieren und Verknüpfen
Eingabe-Funktionen	fragen Eingabemedien wie Tastatur und Maus ab
Nachfrage-Funktionen	sind das Gegenteil der Attribut-Funktionen, mit ihnen kann die jeweilige Einstellung ermittelt werden

Im Folgenden werden alle VDI-Funktionen der entsprechenden Amiga-Funktionen gegenübergestellt. Da oftmals keine entsprechende Funktion existiert, muss diese durch andere Befehle nachgestellt werden, oder ganz einfach generell neu programmiert werden. Ich werde bei den Amiga-Funktionen die Library angeben, oder eine Struktur, wo ein Eintrag zu ändern ist.

Die Funktionen beider Systeme sind in einem C-ähnlichem Code verfasst. Dies vereinfacht die Schreibweise, und ist trotzdem verständlich. Links finden wir die Atari-Funktionen, rechts eine mögliche Umsetzung.

Kontrollfunktionen zum Ansteuern von Geräten

`vopnwk(workin, &handle, workout)`, Open Workstation, Gerätetreiber werden geladen	keine verg. Routine
`vclswk(handle)`, Close Workstation	keine verg. Routine
`vopnvwk(workin, &handle, workout)`, Open Virtual Screen Workstation, einen Bildschirm ansprechen, nicht aber den Bildschirm des ST	keine verg. Routine

vclsvwk(handle), Close Virtual Screen Workstation, virtuelles Ausgabegerät schließen	keine verg. Routine
vclrwk(handle), Clear Workstation, Löscht Bildschirm oder Seitenvorschub, Drucker bzw. Plotter	keine verg. Routine
vupdwk(handle), Update Workstation, Befehle, die im Puffer zwischengespeichert sind, werden ausgeführt	keine verg. Routine
vstloadfonts(handle, select), Load Fonts, für Gerätetreiber werden Zeichensätzte zur Verfügung gestellt	keine verg. Routine
vstunloadfonts(handle, select), Unload Fonts, Speicherplatz der Zeichensätze wieder freigeben	keine verg. Routine
vsclip(handle, clipflag, pxyarray), Set Clipping Rectangle, Grafikoperationen werden in einem Bildschirmausschnitt begrenzt	layers.library, InstallClipRegion(Layer,Region)

Grafische Ausgaberoutinen

v_pline(handle, count, pxyarray), Polyline/Polygon zeichnen	graphics.library, PolyDraw(RastPort,Anzahl,PointArray)
v_pmaker(handle, count, pxyarray), Polymaker, Im ptsin Array gesetzen Makierungen werden gezeichnet	siehe v_pline()
v_gtext(handle, x, y, string), Text ausgeben	graphics.library, Move(RastPort,x,y), Text(RastPort,String,strglen(String))
v_fillarea(handle,count,pxarray), Filled Area, Ein Polygon ausfüllen	graphics.library, Anfangspunkt mit AreaMove(RastPort,x,y), weitere Punkte mit AreaDraw(RastPort,x,y) füllen mit AreaEnd(RastPort)
v_cellarray(...), Cell Array, Bildschirm in Farb-Rechtecke aufteilen	keine verg. Routine
v_contourfill(handle,x,y,index), Contour Fill, Bereich ausfüllen	graphics.library, TmpRas-Struktur erstellen, dann Flood(RastPort,Modus,x,y)
vr_recfl(handle,pxyarray), Fill Rectangle, Rechteck ausfüllen	graphics.library, RectFill(RastPort,x1,y1,x2,y2)
+v_bar(handle,pxyarray), Bar, ausgefüllten Balken zeichnen	siehe vr_recfl()
v_arc(handle,x,y,radius,begang,endang), Arc, Kreisausschnitt zeichnen	muss selber programmiert werden
v_ellarc(handle,x,y,radius,begang,endang), Elliptical Arc, Ellipsenausschnitt zeichnen	muss selber programmiert werden
v_pieslice(handle,x,y,radius,begang,endang), Pie, zeichnet ausgefüllten Kreisbogen	muss selber programmiert werden
v_ellpie(handle,x,y,xradius,yradius,begang,endang), Elliptical Pie, zeichnet ausgefüllten Ellipsenausschnitt	muss selber programmiert werden
v_circle(handle,x,y,radius), Circle, ausgefüllten Kreis darstellen	graphics.library, AreaEllipse(RastPort,x,y,Radius,Radius), füllen mit AreaEnd(RastPort)
v_ellipse(handle,x,y,yradius,xradius), Circle, ausgefüllte Ellipse darstellen	graphics.library, AreaEllipse(RastPort,x,y,XRadius,YRadius), füllen mit AreaEnd(RastPort)
v_rbox(handle,pxyarray), Rounded Rectangle, Rechteck mit abgerundeten Ecken	muss selber programmiert werden
v_rfbox(handle,pxyarray), Filles Rounded Rectangle, ausgefülltes Rechteck mit abgerundeten Ecken	muss selber programmiert werden
v_justified(handle,x,y,string,length,word_space,char_space), Justified Graphics Text, Formatiert Text ausgeben, Zwischenräume werden automatisch eingefügt	muss selber programmiert werden

Attribut-Funktionen

vswr_mode(handle,mode), Set Writing Mode, Ob Grafikoperationen replaced, transparent, XOR, oder reverse-transparent	graphics.library, SetDrMd(RastPort,Modus)

vs_color(handle,index,rgb_in), Set Color Repräsentation, Farbnummer setzen, Intensität hier zwischen 0 und 1000	graphics.library, SetRGB4(ViewPort,FarbReg,r,g,b)
vs_color(handle,color_index), Set Polyline Color Index, Farbstift setzen	graphics.library, SetAPen(RastPort,Nr)
vsl_type(handle,style), Set Polyline Line Type, Linentyp angeben	Struct RastPort, RastPort.LinePtrn ändern
vsl_width(handle,width), Set Polyline Line Width, Linenbreite angeben	Blitterbedingt ist die Breite immer 16 Pixel
vsl_ends(handle,beg_style,end_style), Set Polyline End Styles, Ende und Anfang der Linen mit Abschluss, z. B. Pfeil oder abgerundet	muss selber programmiert werden
vsm_type(handle,symbol), Set Polymaker Type, Markierungen nicht Punkt, sondern bzw. Plus, Stern, Quadrat	muss selber programmiert werden
vsm_height(handle,height), Set Polymaker Height, Höhe der Markierungen ändern	muss selber programmiert werden
vsm_color(handle,color_index), Set Polymaker Color Index, Farbe der Markierungen ändern	muss selber programmiert werden, evtl. SetAPen()
vst_height(handel,heigh,&char_width,&char_heigh,&width,&cell_height), Set Character Heigh, Absolute Mode Ausgabetext in Höhe varieren	evtl. in diskfont.library, TextAttr.YSize ändern, dann neuen Zeichensatz laden
vst_point, Set Character Cell Heigh, Points Mode	siehe vst_heigh()
vst_rotation(handel,angel), Set Character Baseline Vector, Grundlinie drehen	muss selber programmiert werden
vst_font(handle,font), Set Text Face, Zeichensatz auswählen	graphics.library, OpenFont(TextAttr), Font muss sich in der Systemliste befinden, andernfalls diskfont.lib
vst_color(handle,color_index), Set Graphic Text Color Index, Farbe des Textes einstellen	evtl. SetAPen()
vst_effect(handle,effect), Set Graphic Text Special Effects, Fettschrift, helle, dunkle, unterstrichene,ausaueindergezogene Schrift	graphics.library, SetSoftStyle(RastPort,font,Effekt
vst_alignment(handle,hor_in,vert_in, &hor_out,&vert_out), Set Graphic Text Alignment, Hori- oder Vertikale Textausrichtung	Struct TextFont, z. B. TextFont.Baselin
vsf_interior(handle,style), Set Fill Interior, Fülltyp für Füllfunktion setzen	Struct RastPort, RastPort.AreaPtrn ändern
vsf_style(handle,style_index), Set Fill Style Index, Füllmuster auswählen	siehe voriges
vsf_color(handle,color_index), Set Fill Color Index, Farbe des Füllmusters einstellen	evtl. SetAPen()
vsf_perimeter(handle,per_vis), Set Fill Perimeter Visibility, Umrandung ein-/ausschalten	Struct RastPort, OPen ändern durch RastPort.AOlPen
vsf_updat(handle,pfill_pat,planes), Set User Defined Fill Pattern, eigenes Füllmuster definieren	siehe vsf_interior()

Raster-Operationen

vro_cpyform(handle, wr_mode, pxyarray, psrcMFDB, pdesMFDB), Copy Rater, Opaque, Quellraster mit Zielraster verknüpfen	graphics.library, von Bitmap nach Bitmap: BltBitMap(Quelle, x1, y1, Ziel, x2, y2, Breite, Höhe, Miniterm, Mask, Buffer) von Bitmap nach RastPort: BltBitMapRastPort(Quelle, x1, y1, Ziel, x2, y2, Breite, Höhe, Miniterm)
vro_cpyfm(handle, wr_mode, pxyarray, psrcMFDB, pdesMFDB, color_index), Copy Raster Transparent, einfarbiges Raster in farbiges wandeln	graphics.library, mit Blitter durch Schablone emulieren BltMaskBitMapRastPort(Qulle, x2, y2, Breite, Höhe, Miniterm, BltMask)
vr_trnfm(handle, psrcMFDB, pdesMFDB), Transform Form	keine Entsprechung
v_get_pixel(handle, x, y, pel, index), Get Pixel, Farbe des Punktes ermitteln	graphics.library, FarbNr = ReadPixel(RastPort, x, y)

Eingabefunktionen

set_mode(handle, dev_type, mode), Set Input Mode, Modus der log. Eingabeeinheit festsetzen, Sample oder Request	keine Entsprechung

vrq_locator(handle, x, y, &xout, &yout, &term), Input Locator, Request Mode, Position des Grafikcursor feststellen	nur mit Maus: Struct RastPort, RastPort.cp_x und RastPort.cp_y
vrm_locator(handle,x,y,&xout,&yout, &term), Input Locator, Sample Mode, Position des Grafikcursor feststellen	schwierige Entsprechung nur mit MessagePort lösbar
vrq_valuator(handle, valuator_in, &valuator_out, &terminator), Input Valuator, Request Mode, Wartet auf Eingabeänderung mit Cursor-Tasten (nicht auf dem ST, sonst aber normale GEM-Funktion)	siehe vrm_locator()
vsm_valuator(handle, valuator_in, &valuator_out, &terminator), Input Valuator, Sample Mode, Eingabeänderung mit Cursor-Tasten (nicht auf dem ST, sonst aber normale GEM-Funktion)	siehe vrq_valuator()
vrq_choise(handle,ch_in,ch_out), Input Choise, Request Mode auf Funktionstaste warten	1. Window-Handle ermitteln, 2. IDCMP auf Tastendruck, 3. Messageport einrichten, 4. Waiten und replyen, 5. aus IntuiMessage Tastencode, oder über Dos.Lib read() aus RAW:, 6. man erhält CSI,0X-CSI,9X
vsm_choise(handle, ch_in, ch_out), Input Choise, Sample Mode, zuletzt gedrückte Funktionstaste auswerten	ähnlich wie vrq_choise(), 1. Window-Handle ermitteln, 2. IDCMP auf Tastendruck, 3. Messageport einrichten, 4. aus IntuiMessage letzten Tastencode
vrq_string(handle, max_lenght, echo_mode, echo_xy, &string), Input String, Request Mode, String einlesen	dos.library, Read(Output(),Mode,max_len), intuition.library, StringGadget einrichten
vsm_string(handle, max_lenght, echo_mode, echo_xy, &string)	siehe vrq_string()
vsc_form(handle, pcur_form), Set Mouse Form, Neue Form des Grafikcursors	intuition.library, SetPointer(Window, Heigh, Width, XOffset, YOffset)
vex_time(handle, tim_addr, otim_addr, &tim_conv), Exchange Timer Interrupt, Systeminterrupt verbiegen	exec.library, neuen Systeminterrupt: SetIntVektor(IntNr,Interrupt), Interrupt an Server anfügen, AddIntServer(IntNr,Interrupt)
v_show/hide_c(handle [,reset]), Show Cursor/Hide Cursor, Grafikcursor ein/ausschalten	intuition.library, Mauszeiger über SetPointer(), löschen oder darstellen oder CSI nutzen
vq_mouse(handle, &pstatus, &x, &y), Sample Mouse Button State, Ermitteln der gedrückten Maustaste und der Positionen der Maus	intuition.library, IDCMP Flags setzten (MOUSEBUTTONS und MOUSEMOVE) daraufhin bei Nachricht IntuiMessage auslesen oder dos.library bei Read() CSI, Klasse(2);Unterklasse(0);Taste;Status(1000=linke Maustaste,2000=rechte Maustaste);MausX,MausY;Sekunden;Mikrosekunden
vex_butv(handle, pusrcode, psavcode), Exchange Button Change Vektor, Bei Bestätigen der Maustaste Routine	input.device, auf Mausdruck warten und reagieren anspringen
vex_motv(handle, pusrcode, psavcode), Exchange Mouse Movement Vector, bei Bewegen der Maus Routine anspringen	input.device, auf Mausbewegung warten und reagieren
vex_curv(handle, pusrcode, psavcode), Exchange Cursor Change Vector, bei Cusoränderung Routine anspringen	input.device
vq_key_s(handle,&pstatus), Sample Keybord State Information, Welche Sondertasten sind gedrückt worden?	Auslesen der IntuiMessage oder dos.library,Read() siehe vq_mouse(), Taste hier: 1=lk. Shift-Taste, 2=re. Taste, 4=CapsLock, 8=Control, usw.

Nachfrage-Funktionen

vq_extnd(handle, owflag, work_out), Extended Inquire Function, erweiterte Nachfragefunktion, Bildschirmart, Anz. Hintergrundfarben, unterstützt Texteffekte j/n, Vergrößerung j/n, Anz. Farbebenen für Raster, look-up-Unterstützung, Text-Rotation j/n, Anzahl Raster Operationen in der Sekunde, Anzahl Zeichen-Modi, Textausrichtung j/n, Farbwechsel Plotter j/n, Farbwechsel Farbbandverschiebung bei Matrixdruckern j/n, max. Anz. Punkte in polyline, max Größe des int-Array, Anz. Maustasten, Linientyp für breite Linen j/n	(TODO)
vq_color(handle, color_index, set_flag, rgb), Inquire Color Representation, Einstellung der Farbmischung ermitteln	graphics.library, Farbe = GetRGB4(ColorMap,FarbReg)
vql_attributes(handle, attrib), Inquire Current Polyline Attributes, Linien-Attribute ermitteln	Struct RastPort, Linientyp: RastPort.LinePtren,

	Linenfarbe: `Rast.Port.FbPen`, Zeichenmodus: `RastPort.DrawMode`
`vqm_attributes(handle, attrib)`, Inquire Current Polymarker Attributes, Markierungstypen ermitteln	keine Entsprechung
`vqf_attributes(handle, attrib)`, Inquire Current Fill Attributes, Markierungstypen ermitteln	Struct `RastPort`, Fülltyp: `RastPort.LinePtren`, Linenfarbe: `Rast.Port.FbPen`, Zeichenmodus: `RastPort.DrawMode`
`vqr_attributes(handle, atrtrib)`, Inquire Current Graphic Text Attributes, Text-Attribute ermitteln	Struct `RastPort`, z. B. `RastPort.TxHeigh`
`vqr_extend(handle, string, extent)`, Inquire Text Extend, Ausmaße des Strings berechnen	`graphics.library`, `TextLenght(RastPort, Strg, AnzZeichen)`
`vqr_width(handle, char, &cell_width, &left_delta, &right_delta)`, Inquire Character Cell Width, Ausmaße eines Zeichens ermitteln	Struct `TextAttrib`

Escape Funktionen

Die folgenden Funktionen sind durch die Streuerzeichen des CLI's zu emulieren. Jeder Sequenz geht ein $9b für CSI voraus.

`vq_chells(handle,&rows,&colums)`, Größe des Bildschirm ermitteln	CSI,$71, es kommt: $CSI,31,3b,Zeilen,$3b,Spalten,$73
`v_exit_cur(handle)`, von Text-Modus nach Zeichen-Modus	wofür?
`v_enter_cur(handle)`, von Zeichen-Mode nach Text-Mode	wofür?
`v_curup(handle)`, Text-Cursor herauf	CSI,"1",41
`v_curdown(handle)`, Text-Cursor herunter	CSI,"1",42
`v_curright(handle)`, Text-Cursor nach rechts	CSI,"1",43
`v_curleft(handle)`, Text-Cursor nach links	CSI,"1",44
`v_curhome(handle)`, Text-Cursor nach links oben	CSI,"1",$3b,"1",$48
`v_eeos(handle)`, Text bis Bildende löschen	CSI,$4a
`v_eeol(handle)`, Text bis Zeilenende löschen	CSI,$4b
`vs_curadress(handle)`, Cursor seten	CSI,Zeile,$3b,Spalte,$48
`v_curtext(handle)`, Text ausgeben	`Write(Output(),Strg,Len)`
`v_rvon(handle)`, Textausgabe invertiert	CSI,"7",$6d
`v_rvoff(handle)`, Invertierung rückgängig	CSI,"0",$6d
`vq_curadress(handle)`, Cursorposition erfragen	CSI,$6e
`vq_tabstatus(handle)`, Fragen ob Maus verfügbar	warum?
`v_hardcopy(handle)`, Hardcopy	
`v_rmcur(handle)`, Maus abschalten	Cursor unsichtbar: CSI,$30,$20,$70
`v_dspcur(handle, x, y)`, Mauscursor an Position einschalten	Cursor sichtbar: CSI,$20,$70

AES (Applikation Enviroment System)

Das AES verwaltet die "höheren" Objekte. Es lassen sich zu AES selber wieder 11 Bibliotheken aufzählen:

- **APPL** (APPLication Manager). Anmeldung und Abmeldung eines GEM-Programms.
- **EVNT** (EVeNT Manager). Routinen zur Überwachung von Ereignissen. Bei uns macht das `Exec` mit den Message-Funktionen.
- **MENU** (MENU Manager). Es befinden sich hier Routinen zur Menüverwaltung. Zuständig beim Amiga ist die `intuition.library`.
- **OBJC** (OBJeCt Manager). Sie umfasst Routinen zum Erstellen und Verändern von Objekten. Beim Amiga werden diese Objekte Gadgets genannt. Sie finden sich in der `intuition.library`. Sonderobjekte in 2.0 sind in die `gadtools.library` ausgelagert worden.
- **FORM** (FROM Manager). Der Form-Manager ist eine Untereinheit, die zur Verwaltung von Dialogen eingesetzt wird. Auch ein Alert ist ein Dialog, um entsprechende beim Amiga zu suchen, ist die `exec.library` und die `intuition.library` heranzuziehen.
- **GRAF** (GRAphics Manager). Diese Einheit verwaltet bewegende Rechtecke.

- **SCRP** (SCRP Manager), GEM erlaubt es, mehrere Programme im Speicher zu halten. Damit diese auch Daten austauschen können, existiert ein Scrap-Manager. Scrap ist wie immer engl. und steht für Stückchen. Nur zwei Funktionen finden sich in diesem Teil von AES. Sie setzten bzw. lesen das Scrap-Directory. Dem Amiga wird die Kommunikation über AREXX leicht gemacht.
- **FSEL** (File SELect Manager). Der einzigartige File-Requester. (Uaaaah). Beim Amiga vielfältig erreichbar: `asl.library`, `arp.library`, `requester.library`, `reqtools.library`, `req.library`, und vermutlich noch viele mehr.
- **WIND** (Window Manager). Dieser Teil umfasst die meisten Funktionen, nicht umsonst werden wir verstärkt nach Parallelen zum AmigaOS suchen.
- **RSRC** (ReSouRCe Manager). Mit den Funktionen der RSRC-Library können die mit einem Resource-Constuction-Set erstellen Resource-Files geladen und verwaltet werden. Ein Resource-Construction-Programm erstellt Menüs, Alerts, Dialogboxen, Icons und mehr. Der Programmierer fährt also einfach mit der Maus über ein Arbeitsfeld, und bestimmt so das das Aussehen seiner späteren Oberfläche. Auf dem Amiga existieren zwar auch arbeitsvereinfachende Construction-Set, z. B. PowerWindow, aber es existieren keine Programme, die diese Dateien, die ja ein spezielles Format haben, laden. Na ja, so tragisch ist das auch nicht, immerhin erstellt PowerWindow ja eine Datei in C- oder Assembler-Format, das reicht für die Anwendungen aus.
- **SEHLL** (SHeLL Manager). Er enthält einige Routinen, mit der sich andere z. B. Applikationen starten lassen, oder die Kommandozeile ausgelesen werden kann.

Wie bei den VDI-Funktionen werden hier die AES-Funktionen mit Amiga Funktionen verglichen. Eine Gegenüberstellung entfällt, da die Routinen eine etwas genauere Erklärung benötigen.

Application Manager

```
ap_id = appl_Init()
```

Jede GEM-Anwendung muss diese Funktion aufrufen, um sich als Applikation anzumelden. Dadurch wird erreicht, dass eine Liste der laufenden Programme erstellt wird, und somit eine Verwaltung ermöglicht wird. Zurückgegeben wird eine Identifikationsnummer.

Ähnlich kann die Exec-Funktion `FindTask(NULL)` gesehen werden, hier wird auch die Systemliste nach dem aktuellen Task durchsucht, zurückgegeben wird der Zeiger auf die Task-Struktur, was man auch als Identifikationsnummer ansehen kann.

```
ap_rreturn = appl_read(ap_rid, ap_rlenght, ap_rpbuff)
```

Mit der Lese-Funktion kann aus einem Ereignispuffer eine bestimme Anzahl von Bytes gelesen werden. Die Abkürzung `ap_rid` wird im Weiteren Verlauf immer für die Identifikationsnummer stehen.

Umzusetzen auf den Amiga bedeutet einen Message-Port einzurichten, und ihn dann vom anderen Programm auszulesen. Eine andere Möglichkeit bietet das Device PIPE:

```
ap_wreturn = appl_write(ap_wid, ap_wlenght, ap_wpbuff)
```

Gegensatzfunktion zu `app_read`, in den Ereignispuffer werden eine bestimme Anzahl von Zeichen geschrieben. Bei beiden Funktionen ist ap_r/wlenght die Anzahl der Zeichen, und `ap_r/wpbuff` der Puffer.

Um diese beiden Funktionen auf dem Amiga zu portieren, können wir auch einfach den unbenutzten Zeiger von `tc_UserData` der Task-Struktur nutzen. Da dieser ja für den Benutzer frei ist, und für jede Applikation ein anderer Puffer existieren muss, ist er gut zu verwenden. Man muss lediglich eine kleine Funktion schreiben, die Speicher allokiert, und den Zeiger auf diesen in `Task.UserData` einträgt.

```
ap_fid = appl_find(ap_fpname)
```

Mit dem Dateinamen der Funktion `(ap_fpname)` die Identifikationsnummer holen.

Der Amiga bietet die Funktion `Task=FindTask(name)`, die haargenau das gleiche macht, den Tasknamen suchen.

```
ap_trreturn = appl_trecord(ap_trmem, ap_trcount)
```

Diese Funktion dient zum Speichern der auftretenden Events. Sie trät dazu jedes auftretende Ereignis ein einer Liste ein, die durch den Zeiger `ap_trmem` gegeben ist. Ist die Anzahl der durch `ap_trcount` geg. Events überschritten, so werden keine Meldungen mehr gesichert. Anwendungsbeispiel: Dokumentationen von Software mit Automatischer Abarbeitung; Makrorecording.

Da diese Routine sehr komplex ist, muss zur Übertragung einiges an Aufwand getrieben werden. Wesendlich zur Arbeit herangezogen wird das `input.device`, welches uns die Ereignisse liefert. Wir generieren zunächst einmal einen Interrupt, der eine hohe Priorität hat, so dass die Priorität des Systemhandlers von 50 übertroffen wird. Zum Beispiel durch 51. Jetzt kommen die Ereignisse zunächst zu uns, und können in einem Speicherbereich abgelegt werden. Ein Demo-Programm findet sich im Amiga-Intern.

Der Events-Block von Atari besteht aus der Event-Art (Byte 0-1) und aus dem Event-Parameter (Byte 2-5). Die Werte sind folgende:

EventArt	Event-Parameter
Timer-Event (0)	Milli-Sekunden
Button-Event (1)	1 = gedrückt
Mouse-Event (2)	Lo-Word X/Hi-Word Y Koordinate
Keybord-Event (3)	Bit 0 = Shift rechts

	Bit 1 = Shift links
	Bit 2 = Ctrl
	Bit 3 = Alt

Die auf dem Amiga entsprechenden Event-Typen sind:

Timer-Event (0)	IECLASS_TIMER (6)
Button-Event (1)	IECLASS_RAWKEY (1)
	IEQUALIFIER_LBUTTON($1000)
	IECLASS_RAWKEY (1)
	IEQUALIFIER_RBUTTON($2000)
Mouse-Event (2)	IECLASS_RAWMOSE (2)
	Koordinaten in InputEvent.ie_xy
Keybord-Event (3)	IECLASS_RAWKEY (1)

```
ap_tpreturn = 1 = appl_tplay(ap_tpmem, ap_tpnum, ap_tpscale)
```

Mit dieser Routine werden `ap_tpnum`-Events mit einer abgebbaren Geschwindigkeit (tpscale) (von 1-1000 regelbar) abgespielt. Der Block, wo die Events zu finden sind, wird der Funktion über den Zeiger `ap_tpmem` mitgeteilt.

Ähnlich wie `appl_trecord()` ist auch hier vorzugehen. Wieder wird das `input.device` herangezogen. Es müssen nun die Events aus dem Speicherblock ausgelesen, und an alle Input-Handlern weitergeleitet werden. Dies ist relativ einfach, für das Input-Device existiert das Kommando `IND_WRITEEVENT`, so kann der InputEvent-Block reisen.

Hier ein kleines Pseudoprogramm:

```
Input = OpenDevice("input.device")
 Input->io_Date = Speicherblock
 Input->io_Lenght = Länge
 Input->io_Command = IND_WRITEEVENT
 DoIO(Input)
CloseDevice(Input)
```

Event Manager

```
ev_kreturn = evnt_keybd()
```

Das AES wartet auf einen Tastendruck. Der Returncode ist der der gedrückten Taste.

Alle Event-Funktionen lassen sich am besten über das Message-System steuern. Da die AES-Funktionen Window-bezogen sein sollen, werden wir beim Amiga zu dem Window, der den Tastendruck empfangen soll, einen Port anlegen. Dieser wird dann mit der `Exec`-Funktion `Wait()` abgefragt, und nach dem Auslesen der `IntuiMessage` u.U. repliet. `IDCMPFlag` muss mit `VANILLAKEY` oder `RAWKEY` gesetzt werden, der Tastendruck ist in der `IntuiMessage.Code` und `IntuiMessage.Qualifier` abgelegt.

```
ev_breturn = evnt_button(ev_bclicks, ev_bmaskm, ev_bstate, &evbmx,
                         &evbmy, &evbutton, &evbkstate)
```

Das AES wartet, bis ein oder mehrere Maustasten gedrückt wurden. Selbst die Anzahl der Klicke bis zum Ereignis kann eingestellt werden (`ev_bclicks`), jedoch machen Werte >2 den Anwender zum Feinmotoriker. Sollen nicht alle Maustasten berücksichtigt werden, so kann `ev_bmaskm` den Werte von 1 (linker Knopf), 2 (RK) und 3 (beide Knöpfe) annehmen. Zurückgegeben wird

Atari	Amiga IDCMP = MOUSEBUTTONS
ev_breturn=Anzahl Mausklicks	kann nicht festgestellt werden
ev_bmx = X-Koordinate	IntuiMessage.MouseX
ev_bmy = Y-Koordinate	IntuiMessage.MouseY
ev_bbutton = welche Maustaste	eigentlich immer die rechte,
	wenn aber bspw. IDCMP=MENUVERIFY
	dann wird ja eine Message auch
	bei gedrückter Linken gesendet
ev_bkstate = Sondertasten	IntuiMessage.Qualifier

```
ev_morevd = evnt_mouse(ev_moflags, ev_mox, ev_moy, ev_mowidth,
                       ev_moheight, &ev_mobuttonm &ev_mokstate)
```

Das AES sendet eine Nachricht, wenn der Mauszeiger ein rechteckiges Feld mit den Koordinaten `ev_mox`, `evmoy`, `ev_mox+ev_mowidth`, `ev_moy+ev_moheight` betritt oder verlässt (`ev_moflags == 0 ? Eintritt : Austritt`).

Schade, dass es so eine Routine nicht standardmäßig gibt. Der Amiga erkennt zwar angeklickte Bereiche, nicht jedoch berührte und angefahrene. Um so eine Funktion nachzuschreiben ist jedoch nicht viel Aufwand erforderlich. Wenn man die Aktionen aufzählt die GEM durchführt, dann kann man etwa diese Liste führen: Warte auf Mausbewegung, ist die Maus im Window, ist der Zeiger im Rechteck, dann sende Message. Diese Sequenzen können auch auf dem Amiga übernommen werde, nur auf den letzten Schritt muss verzichtet werden, es sei denn man möchte noch etwas mehr Aufwand treiben, dann ist die Routine aber „gefühlsechter“, d. h. näher am Original dran. So emulierten wir lediglich die Abfrage, die wie folgt gestaltet werden kann: Man setzt das `MOUSEMOVE`-IDCMP-Flag, und ließt dann die Message aus. Jetzt vergleicht man die Koordinaten, die man durch `IntuiMessage.MouseX(Y)` bekommt, mit den Ausmaßen des Rechteckes. Ist der Wert zwischen `ev_mox` und `ev_mox+ev_mowidth` bzw. zwischen `ev_moy` und `ev_moy+ev_moheight`, dann Treffer.

```
ev_mgresvd = evnt_mesag(ev_mgbuff)
```

AES wartet solange, bis eine Nachricht im Ereignis-Puffer vorliegt.

Zu dieser Funktion haben wir das Abbild `Message = WaitPort(Port)`.

```
ev_tresvd = evnt_timer(ev_tlocount,ev_thicount)
```

Bestimme Zeit lang warten, der 32-Bit Wert wird in Low- und Hi-Word aufgespalten.

Zur Umsetzung können zwei Pfade begangen werden:

1. nur einfach so warten, dann kann man die `dos.library` nutzen. (`Delay()`-Funktion)
2. Message empfangen, wenn Zeit um ist. Dazu z. B. Interrupt einrichten, der die Systemzeit angibt, und mit der Wartezeit addiert. Ist die Summe erreicht, so kann der Interrupt eine eigene Message senden, auf die unser Programm dann reagieren kann.

```
ev_mwich = evnt_multi(ev_mflags, und noch viele Parameter)
```

(Da würden Register nicht ausreichen)

Diese AES-Funktion ist eine Zusammenfassung der bisher aufgezählten. Während die anderen Funktionen nur auf jeweils ein Ereignis warteten, kann mit dieser Funktion auf mehrere Signale gewartet werden.

Die Ereignisse, auf die gewartet werden kann, sind durch Bits in `ev_mflags` kodiert.

```
ev_dspeed= evnt(ev_dnew,ev_dgetset)
```

Mit dieser Funktion lässt sich die Zeit für einen Doppelklick von 0 (lahm) bis 4 (fix) setzen.

Diese Event-Funktion würde auf dem Amiga in der `intuition.library` eine Äquivaltenz finden. Dazu müßten wir `GetPrefs()` aufrufen, die `Preferences` holen, und den Eintrag `DoubleClick` ändern.

Menu Manager

```
me_breturn = menu_bar(me_btree,me_bshow)
```

Mit dieser Funktion kann die Menüleiste eines Fensters gesetzt (`me_bshow=1`) oder gelöscht werden.

Das AmigaOS hat zwei verschiedene Funktionen um ein Menü darzustellen bzw. zu löschen. Mit `SetMenuStrip(Window,Menu)` wird es dargstellt, mit `ClearMenuStrip(Window)` wieder entfernt.

```
me_creturn = menu_icheck(me_ctree,me_citem,me_ccheck);
```

Das Prüfsymbol (Check Mark) wird einem Menüpunkt (`me_citem`) hinzugefügt. `me_ccheck` bestimmt wieder darüber, ob der Check-Mark gesetzt wird (1) oder nicht (Konstante 0).

Notwendig ist so ein Befehl beim Amiga nicht, denn er kann einfach simuliert werden, auch durch ein Makro. Zur Identifikation ist natürlich der Menüpunkt wichtig, wenn wir diesen heranziehen, können wir in der Struct des MenuItem die Flags ändern. `CHECKIT` setzt dazu das Checksymbol, das wir im Gegensatz zum Atari auch selbst definieren können, und durch `MENUETOGGLE` wird dies automatisch gesetzt und gelöscht. Prüfen wir `CHECKED`, so können wir erfahren, ob der Punkt abgeharkt ist, oder nicht.

```
me_nreturn = menu_normal(me_ntree,me_ntitle,mennormal)
```

Die Menüleiste des Fensters wird revers oder normal dargestellt.

Dies ist beim Amiga nicht zu implementieren, oder aufwändig durch einen `SetPatch()`. Der Aufwand ist aber in jedem Fall nicht lohnenswert, sonst würde das AmigaOS dies auch beherrschen.

```
me_treturn = menu_text(me_ttree,me_titem,me_nnormal)
```

Der Texte eines Menüeintrages kann verändert werden.

Auf dem Amiga ist der Vorgang ähnlich wie bei `menu_icheck()`. Der Benutzer muss den Zeiger auf die `MenuItem`-Struktur vorweisen, damit weitergearbeitet werden kann. Jetzt einfach den Zeiger von `MenuItem.MenuName` ändern, und schon ein neuer Name. Sollten wir den Speicherplatz des alten Eintrages nutzen wollen, so darf natürlich der Name nicht größer sein als der vorherige. Aber dies ist ja kein Problem, einfach einen neuen Zeiger einsetzten, und die Länge kann beliebig sein. Hier auch der Nachteil des Atari-OS. Der Menüname wird überschrieben, d. h. der Eintrag hat die maximale Länge des Vorgängers, und man kann nicht Text-toggeln. Durch den Pointereintrag müssen wir lediglich die Textzeiger ändern, und schon haben wir einen neuen Menütext.

```
me_rmenuid = menu_register(me_rapid,me_rpstring)
```

Hier kann eine Accessory im ersten Menükasten aktiviert werden.

Kein Vergleich am Amiga-Markt. Möglichweise Task-Liste durchlaufen, und als Menüpunkt darstellen lassen.

Object Manager

```
ob_areturn = objc_add(ob_atree,ob_aparent,ob_achild)
```

Es wird in den Obkektbaum (`ob_atree`) ein Objekt hinzugefügt.

Etwas komfortabler als beim Atari arbeitet die AmigaOS-Funktion `AddGadget(Window,Gadget,Position)`.

```
ob_dereturn = objc_delete(ob_detree,ob_dlobject)
```

Es wird im Objektbaum (`ob_detree`) ein Objekt mit dem Index `ob_dlobject` entfernt.

Parallel existiert die Funktion `RemoveGadget(Window,Gadget)`. Da bei bei den GEM-Funktionen lediglich ein int-Index zur Verfügung steht, und kein Zeiger auf das Objekt, muss zur Übertragung der beiden Funktionen eine Schleife programmiert werden, die die Gadget-Struct Index-Mal mit dem weiterführenden Zeiger `NextGadget` durchläuft.

```
ob_drreturn=objc_draw(ob_drtree,ob_drstartob,ob_drdepth,
                      ob_drxclip,ob_dryclip,ob_dtwclip,ob_drhclip)
```

Eine Funktion mit vielen Übergabeparametern. Sie dient zur Darstellung des Objekt-Baumes. Gehen wir einmal die Parameter durch, denn diese Funktion gibt es auf dem Amiga nicht in der Form. Also, `ob_drstartob` gibt den Startindex an `ob_drdepth` die Anzahl der Hierarchieebenen, `ob_drxclip` und `ob_dryclip` die x/y-Koordinaten des Ausschnittes, ab dem dargestellt werden darf, und ergänzend dazu `ob_dtwclip` und `ob_drhclip`, die Breite und Höhe.

Um sie der Commodore Maschine beizubringen muss zu der existierenden Gadget-Liste eine zweite angelegt werden. Dies ist aus dem Grunde notwendig, da die einzige Funktion zum Neuzeichnen von Gadgets, RefreshGadgets (und natürlich auf RefreshWindowFrame), keine Clip-Bereiche erlauben. Das heßt, die bestehende Gadget-Liste muss durchlaufen werden, und es dürfen nur die Objekte in der neuen Übernommen werden, die im richtigen Bereich liegen. Wenn die neue Liste dann gezeichnet wurde, kann sie verworfen werden.

```
ob_ofreturn = objc_offset(ob_oftree,ob_ofobject,&ob_ofxoff, &ob_ofyoff)
```

Eine Routine zum Berechnen der absoluten Objekt-Koordinaten relativ zum Bildschirmnullpunkt.

Die Koordinaten sind immer relativ zum rechten, linken, oberen oder unteren Rand, wenn man beispielsweise ein Objekt verändern will, und dazu die Koordinaten benötigt, muss die Gadget-Struct ausgelesen werden, und zunächst einmal festgestellt werden, „Wozu ist mein Objekt relativ zu?“ (Abfragen von Gadget-Relativ (GREL) Konstanten `GRELBOTTOM`, `GRELRIGHT`, `GRELWIDTH`, `GRELHEIGH`). Dann über die aktuelle Fensterbreite `Window.Width` und `Window.Height` die Koordinaten des Objektes berechnen.

```
ob_orreturn = objc_order(ob_ortree,ob_orobject,ob_ornewpos)
```

Ein Objekt wird als Unter-Objekt verschoben.

Sehe ich keine Umsetzung, da wir keine Unter-Objekte haben. Sonst Gadget-Koordinaten holen, und einfach verändern.

```
ob_edreturn = objc_edit(ob_edtree,ob_edooject,ob_edchar,ob_edidx,
                        ob_edkind,&ob_ednewidx)
```

Der Benutzer kann im Objekt einen Text eingeben.

Funktionell wie ein String-Gadget.

```
ob_drrerturn = objc_drreturn(ob_ctree,ob_cobject,ob_cresvd,
                             ob_cxclip,ob_cyclip,ob_cwclip,ob_chclip,
                             ob_cnewstate,ob_credraw)
```

Es werden Objekte verändert, die sich im Bereich `ob_cxclip`, `ob_cyclip` und `ob_cxclip+ob_cwclip`, `ob_cyclip+ob_chclip` befinden.

Form Manager

```
fo_doreturn = form_do(fo_dotree,fo_dostartob)
```

Ein Benutzerdialog erscheint auf den Bildschirm, der mit EXIT zu beenden ist.

Alle Dialoge können als Requester gesehen werden. Bei dieser Funktion ist es denkbar für einfache Rechts-Links-Requester ein `AutoRequest()` zu benutzen, oder ein komplexeren Requester mit `InitRequester()`, `Request()`.

```
fo_direturn = form_dial(fo_diflag,fo_dix,fo_diy,fo_diw,fo_dih)
```

Eine Hilfefunktion für die Formularverwaltung. Sie enthält eigentlich view Funktionen, die in `fo_diflag` angegeben werden.

0 = Bildschrim-Speicher reservieren
1 = ausdehnender Kasten
2 = schrumpfender Kasten
3 = reservierten Grafikbereich wieder freigeben.

Die Funktion unterstützt ein Spezial-Feature, den Flying-Dial, den fliegenden Dialog. Bei einem Fenster, das geschlossen oder geöffnet wird, wird gerne dieser Flieger benutzt.

Auf dem Amiga muss diese Funktion wie viele andere selber gecoded werden. Schwierig ist dies ja nicht, man zeichnet einfach über `DrawBorder()` ein Rechteck, oder nutzt die `Line()`-Funktion. Was etwas Timing und Probieren kostet, ist die Geschwindigkeit des Erscheinens und die der Schrittweite.

```
fo_aexbttn = form_alert(fo_adefbttn,fo_astring)
```

Die bekannte Warnmeldung wird auf dem Bildschirm gebracht. Die Informationen über den Text und über die Buttons werden über den Zeiger `fo_astring` verwaltet. Das Format des Strings ist "[Piktogramm][Text][Knopf]". Piktogramm ist eine Nummer, die folgenden Icons erscheinen lässt:

0 kein Pictogramm
1 NODE-Pic (Ausrufezeichen)
2 WAIT-Pic
3 STOP-Pic

Der Text, mit max. 5 Zeilen und 30 Zeichen, wird mit |-Zeichen zeilenweise getrennt.

Die Knöpfe, von denen es mehrere geben kann, haben alle ihren eigenen Text. Dieser wird in der Form [Text1|Text2|..] angegeben.

Eine schöne Funktion, die durch Libraries wie `requester.lib` schon nachgestellt wurde. Zur Selbstprogrammierung ist eine eigene Requester-Struct wieder gefragt. Dabei ist das Icon ein nicht wählbares Gadget, der Text ein Eintrag in `Requester.ReqText` (bei mehreren Zeilen natürlich verknüpft) und die Buttons ganz normale, umrandete Bool-Gadgets.

```
fo_eexbttn = form_alert(fo_enum)
```

Eine TOS-Warnmeldung wird dargestellt. Grundlage für diesen Dialiog bildet die GEM-Funktion `form_alert()`, die den Text "TOS error #fo_enum" ausgibt. Als Icon wird das Stoppschild benutzt.

```
fo_cresvd = form_center(to_ctree,%fo_cx,&fo_cy,&fo_cw,&fo_ch)
```

Jeder Objet Baum hat bestimmte Startkoordinaten. Diese Funktion errechnet nun die Größe dieses Komplexes, und liefert als Resultat die Koordinaten des Baumes, wie es zentriert erscheinen würde.

Implementiert hieße dies: Gadgetliste durchlaufen, und die Koordinaten auf einem Rechteck ausdehnen. Das heißt, immer die extremsten Ausdehnungen nach unten, oben rechts und links speichern. Dann dieses Rechteck halbieren, und wir hätten den Mittelpunkt. Die Atari-Routine trägt jetzt noch diese Kooridinaten in der Wurzel des Baumes ein.

```
fo_kreturn = form_keydb(fo_ktree,fo_kob,fo_kobnxt,fo_kchar,
                        &fo_koboutnxt,&fo_kouthchar);
```

Eine Unterroutine von `form_do()`.

So wie beim String-Gadget das `input.device` angesprochen wird, so ist es auch bei der Umsetztung von `form_keydb` zu nutzen.

```
fo_breturn = form_button(fo_btree,fo_bob,fo_bclks,&fo_bobnxt)
```

Sie arbeitet ähnlich wie `form_keybd`, nur dass sie die Mausdrücke überwacht. Bei einem Anklicken des Exit-Symbols wird bzw. `fo_breturn` mit 0 zurückgegeben, andernfalls mit einem Wert ungeleich Null.

Graphics Manager

Alle Funktionen des Graphic Managers müssen selber programmiert werden. Somit werde ich nur die Atari-Funktionen beschreiben, aber keine Umsetztungen. Diese sind in allen Fällen sehr einfach.

```
gr_dreturn = graf_dragbox(gr_dwidth,gr_dheight,gr_dstartx,gr_dstarty,
                          gr_dboundx,gr_dboundy,gr_dboundw,gr_dboundh,
                          &dfinishx,&dfinishy)
```

Mit der Maus kann der Benutzer ein Rechteck mit den Ausmaßen `gr_dstartx`, `gr_dstarty`, `gr_dwidth`, `gr_dheight` in einem anderen Rechteck mit den Ausmaßen `gr_dboundx`, `gr_dboundy`, `gr_dboundw`, `gr_dboundh` verschieben. In den beiden Variablen `dfinishx` und `dfinishy` wird die X- und Y-Koordinate beim Loslassen der Maus gesichert.

Diese Funktion wird zum Verschieben der Fenster eingesetzt. Da nur bei sehr wenigen Rechnern (die genug Rechenleistung haben) der komplette Fensterinhalt mit verschobnen wird, (z. B. bei Archimedes (lässt sich aber auch abschalten) wird durch das Rechteck das Fenster angedeutet.

```
gr_rreturn = graf_rubberbox(gr_rx,gr_ry,gr_rminwidth,
                            gr_rminheight,&gr_rlastwidth,&gr_rlastheight)
```

Die Funktion zeichnet einen Kasten, der in der linken oberen Ecke beginnt. Die rechte untere Ecke folgt den Mausbewegungen. Wird die linke Maustaste gedrücket, so ist die Prozedur beendet.

Einsatzbereich: Sizen des Fensters. Auch hier wird das Fenster nicht sofort vergrößert, sonden erst, nachdem der Benutzer die Recheckvorgabe beendet hat. Der Archimeds zeigt auch hier, was möglich ist, bei ihm ist (natürlich wieder optional) eine Vergrößerung mit gleichzeitiger Fensterrefreshung eine Kleinigkeit.

```
gr_mreturn = graf_movebox(gr_mwidth,gr_mheight,gr_msourcex,
                          gr_msourcey,gr_mdestx,gr_mdesty)
```

Diese Funktion zeichnet auf dem Schirm ein Rechteck, dass sich von einer Position zur anderen bewegt.

```
gr_greturn = graf_growbox(gr_gstx,gr_gsty,gr_gstwidth,
                          gr_gstheight,gr_gfinx,gr_gfiny,gr_gfinwidth,gr_grinheight)
```

Ein sich ausdehnendes Rechteck mit den Ausmaßen `gr_gstx`, `gr_gsty`, `gr_gstwidth` und `gr_gstheight` zeichnet diese Funktion. Es wird am Ende so groß sein wir die Box mit der Größe von `gr_gfinwidth`, `gr_grinheight` ab der Position `gr_gfinx`/`gr_gfiny`.

```
gr_sreturn = graf_shrinkbox(gr_sfinx,gr_sfiny,gr_sfinwidt,
                            gr_sfinheight,gr_sstx,gr_ssty,gr_sstwidth,gr_sstheight)
```

Ähnlich der Funktion `graf_growbox()`, nur, dass das Rechteck schrumpft.

```
gr_wreturn = graf_watchbox(gr_wptree,gr_wobject,gr_winstate,gr_woutstate)
```

Ist der Mauszeiger über einem Objekt, und wird der Mauszeiger gedrückt, so wird dieses Objekt verändert. Mögliche Aktionen sind in `gr_winstate` vermerkt:

0	Normal
1	Selected
2	Crossed
4	Checked
8	Disabeled
16	Outtlined
32	Shadowed

Der Übergabewert `gr_wreturn` ist null, wenn die Maus beim Loslassen auserhalb des Rechecks war, andernfalls eins.

```
gr_slreturn = graf_silderbox(gr_slptree,gr_slparent,sr_slobject,gr_slvh)
```

Ähnlich der Funktion `graf_rubberbox()`, jedoch erlaubt `graf_silderbox()` nur die horizontale (`gr_slvh = 0`) oder vertikale Verschiebung (`gr_slvh=1`).

Einsatzbereich sind Fenster, die nur in einer Ausrichtung verändert werden dürfen.

```
gr_handle = graf_handle(&gr_hwchar,&gr_hhchar,&gr_hwbox,&gr_hhox)
```

Mit dieser Funktion werden Grafikdaten wie Handle der geöffneten VDI-Station, Breite und Höhe eines Buchstabens, Breite und Höhe eines buchstabenumfassenden Kastens ermittelt.

```
gr_moreturn = graf_mouse(gr_monumber,gr_mofaddr)
```

Die Form des Mauszeigers kann mit dieser Funktion verändert werden. Wenn die Symbole nicht ausreichen, kann ein eigener definiert werden, die Referenz auf den 35 Byte großen Speicherblock sollte in `gr_mofaddr` stehen. Die Mausform stehend in `gr_monumber` kann folgendes Aussehen haben:

Table 5. Mausformen

0	Pfeil
1	senkrechter Balken (Cursor)
2	Biene

3	Hand mit Zeigefinge
4	flache Hand
5	dünnes Fadenkreuz
6	dickes Fadenkreuz
7	Fadenfreuz als Umriss
255	Mausform ist durch `gr_mofaddr` neu defiert worden
256	Maus abschalten
257	Maus anschalten

```
gr_mkresvd = graf_mkstate(&gr_mkmx,&gr_mkmy,&gr_mkmstate,&gr_mkkstate)
```

Der Status der Eingabegeräte Maus und Tastatur werden ermittelt. In den Speicherbereich `gr_mkmx` und `gr_mkmx` schreibt die Funktion die Koordinaten, `gr_mkmstate` ist der Maus-Taste, in dem Bit 0 gesetzt ist, wenn die linke Maustaste gedrückt ist, `gr_mmkstate` leztendlich der Status der Tasten. (Bit:0=Shift rechts; 1=Sh.lks, 2=Cntr, 3=Alt)

Scrap Manager

```
sc_rreturn = scrp_read(sc_rpscrap)
```

Ein String aus dem AES-Puffer in die eigende Anwendung schreiben. Der Zeiger, der den eigenden Puffer beschreibt ist `sc_rpscrap`.

```
sc_wreturn = scrp_write(sc_wpscrap)
```

Ein String aus dem eigenen Puffer (`sc_wpscrap`) wird in globalen Puffer des AES geschrieben.

Beide Funktionen werden von Programmierern nicht sehr häufig benutzt. Als AmigaOS-Programmierer ist man sehr mit AREXX verwöhnt, dass so ein Bim-Bim nicht nötig wird. Die Kommunikation kann schon etwas komfortabler sein.

File Select Manager

```
fs_ireturn = fsel_input_input(fs_iinpath,fs_iinsel,&fs_iexbutton)
```

Diese Funktion stellt einen File-Selector dar. Da dieser aber völlig daneben ist, gehört diese Funktion vermutlich zu den meist gepatcheden überhaupt im Atari-TOS. Übergeben werden der Prozedur lediglich zwei Parameter, ein Zeiger auf den Pfad-Namen (`fs_iinpath`) und ein Zeiger auf den Dateinamen. Da der Returnwert schon angibt, ob die Funktion an sich fehlerhaft war (z. B. keine Disk), wird eine weitere Variable beschrieben, `fs_iexbutton`. Ist der Wert 0, so ist der ABBRUCH-Knopf gedrückt worden, andernfalls der OK-Knopf.

Window Manager

```
wi_ccreturn = wind_create(wi_crkind,wi_crwx,cr_wy,wi_crww,wi_crwh)
```

Speicher für ein neues Fenster wird von GEM bereitgestellt und es wird eine Window-Struktur erstellt. Den Übergabeparameter kenne wir schon, es ist ein Handle, der in `wi_ccreturn` zurückgegeben wird, der Window-Handle. Auch das Aussehen des Fensters kann durch eine Konstante (`wi_crkind`) festgelegt werden. Folgende Möglichkeiten bietet das System:

Table 6. Options zum Steuern des Fenstersaussehens

Bit	Name	Amiga	Bedeutung
0	NAME	Title=0	Titelzeile wird mit Fenstersnamen belegt
1	CLOSE	WINDOWCLOSE	Eine Close-Gadget
2	FULL	ab 2.0	Ein Feld zum Vergrößern des Bildes
3	MOVE	WINDOWFRAG	Das Window kann am Bildschirm verändert werden
4	INFO	-	Eine Informationsazeile wird mit dargestellt
5	SIZE	-	Die Größe kann modifiziert werden
6	UPARRAW	-	Pfeil nach oben wird beigefügt
7	DNARROW	-	Pfeil nach unten

8	VSLIDE	-	vertikaler Schieber
9	LFARROW	-	Pfeil nach links
10	RTARROW	-	Pfeil nach rechts
11	HSLIDE	-	horizontaler Schieber

Neben dem Aussehen ist die Größe des zukünftigen Windows anzugeben.

Im Gegensatz zum Amiga, der mit `OpenWindow()` ein Fenster erstellt und gleich auch darstellt, sind beim Atari zwei Vorgänge nötig. Denn mit `wind_create()` wird nur Speicher uns Aussehen festgelegt, erscheinen wird das Fenster erst mit dem Befehl:

```
wi_oreturn = wind_open(wi_ohandle,wi_owx,wi_oxy,wi_oww,wi_owh)
```

Das mit win_create Fenster wird dargestellt. Dieses kann jedoch andere Ausmaße und Koordinaten haben. Neue Koordinaten könnte in `wi_owx` und `wi_oxy` übergeben werden, `wi_oww`/`wi_owh` lassen andere Höhen und Breiten zu.

Kann man sich sparen, wenn man allerdings `BACKDROP` setzt (siehe `wind_close()`, sollte man das `BACKDROP`-IDCMP-Flag löschen).

```
wi_clreturn = wind_close(wi_clhandle)
```

Das Fenster, gegeben durch den Handle `wi_clhandle`, wird vom Bilschirm geholt. Die Strukturen usw. existieren jedoch noch, ein erneuter Aufruf von `wind_open()` würde er wieder hervorkommen lassen.

Auf dem OS von unserem Rechner müsste das Fenster versteckt werden. Dazu könnten es wir in den Hintergrund legen, indem wir das `BACKDROP`-Flag durch `ModifyIDCMP()` setzen.

```
wi_dreturn = wind_delete(wi_dhandle)
```

Um das Fenster letztendlich zu verstören, d. h. der Speicherplatz wird freigegeben muss `wind_delete()` eingesetzt werden. Bei uns ein `CloseWindow()`.

```
wi_sreturn = wind_set(wi_shandle,wi_sfield,wi_sw1,wi_sw2,wi_sw3,wi_sw4)
```

Um das Aussehen der Fenster zu verändern, ist diese Funktion integriert worden. Sie erlaubt es dem Benutzer, aufgrund von gesetzen Bits, die in der Maske `wi_sfield` zu setzen sind, Komponenten oder Optionen zu gestatten. Die Komponten sind Randbereich und Titelzeile. Druch die Maske können Fensterschieber gesetzt werden, aktive Fenster festgelegt werden, u.a. Ebenso lassen sich der Fenstertitel und der Titel der darunter liegenden Informationszeile verändern.

Eine Übertragung ist lediglich durch direkte Veränderung der Window-Struktur möglich. Eine Inforationszeile gibt es sowieso nicht, sie müsste evtl. als ein unwählbares Gadget dargestellt werden.

```
wi_greturn = wind_get(gi_ghandle,wi_gfield,&wi_gwl,&wi_gw2,
                      &wi_gw3,&wi_gw4)
```

Mit Hilfe dieses Unterprogramms können Fensterattribute gelesen werden. Die Variable `wi_gfield` wird auf eine Zahl zwischen 4 und 16 gesetzt, und dann werden die Datenfelder mit bestimmten Werten gefüllt. Folgende Auflistung zeigt, welche Werte ausgelesen werden können:

Wert	Funktion
4	Koordinaten des Arbeitsbereiches wird übergeben
5	Koordinaten der Gesamtgröße
6	Koordinaten des vorhergehenden Fensters
7	Koordinaten des Fensters in seiner größtmöglichen Form
8	relative Position des horizontalen Schiebers
9	relative Position des vertikalen Schiebers
10	Window Handle des aktiven Fensters wird übergeben
11	Koordinaten des ersten Rechtecks in der Rechteckliste
12	Koordinaten des nächsten Rechtecks in der Rechteckliste
13,14	reserviert
15	Größe des horizonalen Schiebers relativ zu Größe des Kastens
16	Größe des vertikalen Schiebers relativ zu Größe des Kastens

Auslesen der Window-Strukt. Gegebenenfalls Sizer in Prop-Gad-Strukt auslesen.

```
wi_freturn = wind_find(wi_fmx,wi_fmy)
```

In der Variablen `wi_freturn` wird das Handle des Fensters zurückgegeben über dem sich gerade der Mauszeiger befindet. Bei dem Wert null, so ist, das kann man sich schon denken, kein Fenster unter dem Zeiger, sondern lediglich leerer Desktop.

```
wi_ureturn = wind_update(wi_ubegend)
```

Diese Mehfunktions-Funktion hat die Aufgabe bei den Werten

- GEM AES mittzuteilen, dass die Applikation einen Bildschirmbereich refresht, und daher nicht dazwiscken gefummelt werden darf
- wieder GEM AES über das Ende eines Bildaufbaus zu informieren
- der Applikation die Kontrolle über die Mausfunktionen zu entreißen. Somit liefert AES keine Events mehr bei Größenänderungen von Fenstern oder auch keine Infos über die Drop Menüs.
- die Kontrolle wieder dem GEM zu übergeben

```
wi_creturn = wind_calc(wi_ctype,wi_ckind,wi_cinx,wi_ciny,wi_cinw,
                       wi_cinh,&wi_coutx,wi_couty,&wi_coutw,wi_couth)
```

Bei bekannter Größe des Fensters errechnet diese Routine die Umrissausmaße des Fensters.

In den einzelnen Variablen werden folgende Werte übergeben, die abhängig sind vom Inhalt `wi_ctype`, denn ist dieser Wert 0, so werden die Ausgaben auf die Gesamtgröße bezogen, andernfalls, bei 1, auf die Außmaße des Arbeitsbereiches. Wenn also die folgenden Werte abhängig von `ctype` sind, so sind die anderen Übergabewerte ebenfalls angegeben.

`wi_cinx`	X Koordinate des Arbeitsbereiches/Gesamtfensters
`wi_ciny`	Y Koordinate des Arbeitsbereiches/Gesamtfensters
`wi_cinw`	Breite des Arbeitsbereiches/Gesamtfensters
`wi_cinh`	Höhe des Arbeitsbereiches/Gesamtfensters
`wi_coutx`	X Koordinate des Gesamtfensters/Arbeitsbereiches
`wi_couty`	Y Koordinate des Gesamtfensters/Arbeitsbereiches
`wi_coutw`	Breite des Gesamtfensters/Arbeitsbereiches
`wi_couth`	Höhe des Gesamtfensters/Arbeitsbereiches

Um die Größe korrekt berechnen zu können, müssen selbtsverstämdlich die Komponenten des Randbereiches mit berücksichtigt werden. Sie sind daher in `wi_ckind` zu übegeben, die Bedeutung der Bits die aus `wind_create` bekannt.

Resource Manager

Alle Resource-Funktionen müssen selber programmiert werden, da der Amiga Resource-Dateien nicht bietet.

```
re_lreturtn = rsrc_load(re_lpfname)
```

Eine Datei mit dem Namen `re_lpfname` wird geladen und vom Zeiger-Format ins Pixel-Format umgewandelt.

```
re_freturn = rsrc_free()
```

Das Resouce-File wird aus dem Speicher verbannt.

```
re_gretrn = rscr_gaddr(re_gtype,re-gindex,&re_gaddr)
```

Die Adresse einer Datenstruktur wird ermittelt.

```
re_sreturn = rsrc_saddr(re_stype,re_sindex,re_saddr)
```

Die Adresse einer Datenstruktur wird in den Baum eingearbeitet.

```
re_oresvd = rsrc_obfix(re_otree,re_oobject)
```

Wie Koordiaten eines Objektes werden umgewandelt. Es ist die Umwandlung, die bei `rsrc_load()` automatisch geschieht.

Shell Manager

```
sh_rreturn = shel_read(&sh_rpcmd,&sh_rptail)
```

Der Kommandostring wird ein einen Speicherplazt kopiert. Der Pfadname wird ebenso übermittelt.

Da ein Zeiger auf den Kommandostring der gestarteten Appliaktion immer in `A0` übergeben wird, brauchen wir dazu keine Funktion.

```
sh_wreturn = shel_write(sh_wdoex,sh_wisgr,sh_wiscr,sh_wpcmp,sh_wptail)
```

Sollte eigentlich aus einer Anwendung ein Programm ausführen, und dann zu dieser zurückkehren. Klappt nur leider nicht, das einzige was kommt, ist der Desktop.

Glücklicherweise haben wir ja die `Execute()`-Funktion unter DOS.

```
sh_greturn = shel_get(sh_glen,sh_gpdata)
```

Aus dem globalen Enviroment-Speicher wird eine bestimmte Anzahl Zeichen `sh_glen` in einen Puffer `sh_gpdata` kopiert.

Da die Funktion meistens dazu verwendet wird, das Desktop-Info-File zu lesen, ist auf dem Amiga `GetPrefs()` vergleichbar.

```
sh_preturn = shel_put(sh_plen,shppdata)
```

Wie `shel_get()`, nur, die Daten werden in den GEM-Puffer geschrieben.

```
sh_freturn = shel_find(sh_fpbuff)
```

Im Hauptverzeichis auf dem aktuellen Laufwerk, oder in Laufwerk *A:*, wird nach der angegebenen Datei gesucht, die durch `sh_fpbuff` bestimmt ist.

Diskussion

Wir haben das Atari-OS jetzt von den Seiten der Betriebssytemaufrufe kennengelernt. Nun sollte es an der Zeit, einige Punkte zu diskutieren. Was dem Programmierer bestimmt schon auffiel was die Inkonsistenz der Befehle. Um nun auf eine Taste zu warten, kann man über das TOS gehen, das VDI und das AES benutzen. Man weiss allerdings nicht, wo letztendlich der Tastencode herkommt. Beim AmigaOS haben wird das Input.Device, das liefert die Tasten. Das AmigaOS kann über `Read()` eine Taste holen und über eine Message. `Read()` ist aber verbunden mit einer Message.

Dieses gibt gleich Anhaltspunkt für den großen Komplex, an dem es meineserachtens scheitert. Der Programmierer kann zwar auf viele Funktionen des Betriebbsystemes zurückgreifen, und für Anwenderzwecke reicht die Organisation aus, jedoch fehlt für tiefergehende Programmierung die Struktur. Ein Beispiel: Das AmigaOS ist hierarchisch strukturiert, die unterste Ebene bildet (das ist natürlich Überall so) die Hardware. Was folgt sind Resourcen. Die Devices sind schon eine Stufe höher, und sie wiederum werden von den Libraries verwendet. Der Programmierer nutzt die Libraries, aber möchte er einmal eine Stufe tiefer steigen, dann nutzt er die Devices. Wem also die dos.library nicht reicht, der kann über das trackdisk.device mehr machen, z. B. Tracks formatieren. Zu jedem Libraryaufruf kennt man die Strukturen, so kann man immer nachvollziehen was gesetzt wird, und was aufgerufen wird. Im Vergleich dazu das Atari-OS. Die Funktionen sind wirklich sehr leistungsfähig, die Funktionsvielfalt überstiegt in einigen Bereichen die Amiga-Funktionen, aber, sie müssen es sein. Wenn es keine Funktion zum formatieren eines Tracks gäbe, so müsste gleich die Hardware angeknabbert werden, und jetzt sollte der Leser entscheiden, was ist da wohl besser, ein Device, oder die Hardware. Wir merken schnell, die niedrigere Ebene zum Sound, Speicher-Medium ist schon die Hardware, da zwischen ist nichts mehr. Klar, die Grafikausgaben sind wesentlich flexibler als die des Amiga, das GDOS und GIOS sind starke Einheiten, sie können quasi als Device gesehen werden, nur, davon sollte es mehr geben. Wenn es nicht so viele Funktionen gäbe, dann würden sehr viele Probleme auftauchen.

Als normaler Programmierer wird der Unterschied kaum auftauchen, die Funktionen, die sein müssen sind da. Dort haben beide Betriebssysteme reichlich Auswahl, der Atari ist sogar noch im Vorteil wegen der unabhängigen Grafiksteuerung. Vom Verstehen und von der Ästhetik hat das AmigaOS ruckzuck die Nase vor, hier ist eben alles durchscheinend, und das macht es flexibler. Es gibt viele Strukturen mit vielen Verweisen und Eintragungen. Ich habe dem Eindruck, das die Atari-Entwickler bewusst von diesen Strukturen weglenken wollen. Die Anzahl der Funktionen ist eben nicht alles, obwohl die Programmierer von Atari-Software oft das AmigaOS als zu kompliziert empfinden. Da haben sie nicht unrecht, es fehlen manchmal einige Funktionen, die einfach benutzbar sind, wer mal so eben einen Track einlesen muss, der muss das Message-System kennen, ein Device öffnen, ich kann verstehen, dass das für viele abschreckend wirkt.

Ich möchte einige Beispiele nennen, wo es keine Betriebssystemfunktionen und keine frei hängende Strukturen gibt, und der Atari dann ganz schnell ins schleudern kommt. Dazu noch einige Punkte, wo der Atari leichte Nachteile hat:

Exec

- Resources
- Devices
- Libraries nur begrenzt, nicht shared
- das mit den Tasks
- Semaphoren zur Aufteilung der Resourcen
- Interrupts übers System, und nicht über die Hardware
- was ist mit individuellen Vektoränderungen für jedes Programm?

DOS

- Mehr als 8+3 Dateizeichen (wurde selbst bei PCs erst unter Windows 4.0 besser!)
- Einbindung von Untermenüpunkten

Menu

- Grafiken im Menü
- jeweils andere Zeichensätzte im Menü verwendbar

- Menü ziemlich frei in der Größe bestimmbar

Window
- RastPort gibt wichtige Informationen, z. B. über die Bitmap
- GZZ/kein GZZ
- sofort kann die Gadgetliste ausgelesen werden
- Windowliste durchgehen um so Einträge zu finden
- neue Systemgadgets, Prioritäten

Gadgets
- Proportionalgadgets fehlen
- Automatisches Setzen von Gadget-Zuständen

Grafik
- Alle Funktionen zur Animation wie `AddAnimObj()`, `AddBob()`, `AddVSprite()`, `Animate()`, uvm.

Sonstiges
- Screen gibt es beim Atari gar nicht
- Layers

Hardware
- 4-Kanal-Sound, auch wenn's nur veraltete 8-Bit sind
- HAM-Mode, auch der STE hat 4096 Farben, aber nicht gleichzeitig
- einen Blitter, der nicht nur Blöcke verschiebt und verknüpft, sondern auch noch Flächen füllt und Linien zeichnet
- Hardware-Sprites (auch beim beim STE und größer)
- Einen schönen Copper, mit dem man ganz easy an VSprites kommt, und einen Real-4096-Mode in 700 * 500 Punkten programmiert

Preis/Leistung
- Neu ab min. OS 2.0 oder 3.0, dabei AREXX, TrueType-Fonts, Monitortreiber, umfassende Preferences

Tastatur
- Mantel des Schweigens breitet sich aus

Andersherum; was haben die, was wir nicht haben:

- Es gibt beim AmigaOS kein GIOS oder GDOS. Diese Komponenten sind aber wegen der Systemoffenheit einfacher einzubinden als es beim Atari möglich wäre.
- viele Funktionen zur Grafikunterstürzung, die wir, wenn wir sie wirklich brauchen, selber programmieren müssten. Doch mal ehrlich, wann braucht man denn schon einmal eine gefüllte Ellipse (nachbilden mir `DrawElipse()`, `Flood()`), oder ein abgerundetes Rechteck. Meines Erachtens Spielerei.
- Viele Dos-Funktionen. Ganz vorteilhaft, wir können den Stress über das `trackdisk.device` sparen

Dazu Schlusswort: Wenn der Falcon guten Umsatz macht, kann der Amiga in Punkto Hardware einstecken, denn der hat bessere Leistungsdaten, aber warten wir mal ab, das OS muss erst einmal so gut werden.

Büchertipp

Der gesamte Computermarkt macht mit Büchern sehr guten Umsatz. Da er somit für viele Verlage ein Leckerbissen darstellt, sollte man mit der Wahl der Bücher (Bookware) schon vorsichtig sein, um keine „Billigprodukte“ ohne Inhalt zu kaufen. Die folgende Kurzvorstellung kann da unter Umständen hilfreich sein.

Ich möchte besonders darauf hinweisen, dass die Ansicht subjektiv ist. Die Buchwahl ist nur aus einer Auswahl von Werken, die ich gelesen bzw. durchgeblättert habe. In der Aufzählung versuche ich sie nach ihrem sachlichen Inhalt darzustellen.

Amigabücher zum Thema Assembler

AMIGA, Amiga Maschinensprache für Einsteiger, Data Becker

Ein wirkliches Einsteigerbuch, steht wenig drin, man lernt wenig, man weiß nicht, wie man weiter arbeiten muss. Das Buch arbeitet ein wenig in DOS hinein (Datei öffnen, auslesen und ausgeben), Höhepunkt ist ein eigenes Fenster mit Punkt und Text. Über Strukturen erfährt der Programmierer so gut wie nix. Fazit: Dieses Buch ist nicht zu gebrachen. Wer er kauft, kann es höchstens auf dem Flohmarkt wieder verkaufen, aber der Verkaufspreis ist dann höchstens 5 Mark. Tut mit leid für den Autor.

AMIGA, ASSEMBLER-BUCH, Markt und Technik (M&T)

Das Buch mag besser sein als das Data Becker Buch, es hat jedoch noch viele Schwächen. Der Anfang ist schnell, dem Lernenden werden die Mnemoniks um den Kopf geschmissen, und dann los. Die Einführung in die Betriebssystemkomponenten Intuition, Grafik und Diskfont ist gut, nur vom anderen könnte etwas mehr sein, von `Exec` erfährt der Benutzer auch nicht gerade viel, außer vielleicht, wie man Speicher beschafft. Das letzte Kapitel ist durchaus kopierenswert, der Anwender findet massig Strukturen und wichtige Erklärungen zu den Struktureinträgen.

Die Demos, die auch auf der beigelegten Disk zu finden sind, sind in einem schlechten Assemblerstil geschrieben, wer so programmiert sollte auf Assembler verzichten, C Compiler könnten besser sein. Die Unterprogramme ohne Zusammenhang, einfach drangepappt. Und wenn ich von einer Ellipsenroutine „schnell“ höre, dann will ich auch eine schnelle sehen. Die Programmierung ist aber ein Witz. Diese 60 DM müssen nicht sein.

Peter sollte besser bei C und beim PC bleiben, das liegt im wesentlich besser, sagen wir mal, das Assemblerbuch war ein Ausrutscher.

ASSEMBLER, M&T

Ein Assemblerbuch ganz anderer Art. Endlich wird einmal auf die Hardware des Amigas eingegangen, OS Programmierung ist

hier nur Mittel zum Zweck. 40 DM kann man anlegen, Tipps sind allemal zu finden, nur vom Betriebssystemteil darf nicht zu viel verlangt werden. Auf jeden Fall ist es sinnvoll Assemblerkenntnisse mitzubringen.

Prozessorbücher MC680x0

Die 68'oooer - Grundlagen und Programmierung, AT Verlag

Das Assemblerbuch aus dem Jahre 1983, hat zum Ziel Assemblersprache und Assemblerprogramm zusammen zu vermitteln. Die Kapitel unterteilen sich in Einführung, Cross-Makro-Assembler, Befehlssatz zum Einstieg, und zur Weiterführung in 6 weite Kapitel. Der Leser erfährt von mathematischen Routinen, und Listen, zwei Abschnitte, die ich zum Vorbild für mein Buch genommen habe. Die letzen vier Kapitel beschreiben die Hardware von Außen, die verschiedensten Zustände (dieses Kapitel hätte chronologisch wo anders besser hingepasst), den Anschluss von Peripheriebausteinen und sonstigen Erweiterung, auch auf Prozessorsysteme ausgeweitet.

Die Erklärungen für die Befehle sind sanft und feinfühlig, viele Tabellen und Skizzen helfen zum Verständnis. Ein schönes Buch, das leider nicht weiter auf die nachfolgenden CPUs der Prozessorreihe eingeht. Zum MC68010 weiß der Autor 17 Seiten zu schreiben, zum 20ger lediglich 12 Seiten. Das ist doch etwas dürftig, aber angesichts des Alters OK.

68000 Mikroprozessorhandbuch, Osborne/McGraw-Hill

Dieses eher etwas untersetzt wirkende Buch, gerade einmal 130 Seiten, widmet sich größtenteils der Hardware des MC68000 Prozessors. Hier erfährt der Leser alles Wissenswerte über die Zeitabläufe auf den Bussen, die CPU-Schnittstellen und weiterem. In Anhang listet Tafel A jeden Befehls mit seinen Operanden, Ausführungszeiten, Status-Beeinflussung sowie der Länge und einer kleinen Befehlsbeschreibung auf. Tabelle B ist für Assemblerbauer ein wichtiges Nachschlagewerk, jeder Befehl wird in seine Adressierungsarten aufgespaltet, und dazu werden die Befehlscodes geliefert. Leider beziehen sich die Tabellen nur auf den 68000 Prozessor, ansonsten sehr gute Informationen. Auf weitere Prozessoren wird im dritten und letzten Anfang eingegangen. Ausreichende Informationen zum MC68020 werden gegeben, der Autor beschreibt Anschlüsse, Bus, und den neuen Befehlssatz. Da das Buch auch schon von 1985 ist, dürfe man es gebraucht oder getragen billig bekommen (ich kaufte es für fünf Mark), und zuschlagen sollte da jeder.

Die M68000 Familie - Band 2, tewi Verlag

Ein weiteres Buch, das hilf, Wissen über 68000er zu vermehren. In erster Linie richtet es sich an (Zitat) „Hardware- und Systementwickler, Fachleute aus Fertigung und Reparatur, 68000- Assembler-Programmierer und Studenten aus den technischen Fachrichtungen". Also für mich spricht nur das 68000-Ass. Prog. an, alles weitere ist mir zu hoch. Aber trotzdem, ein Buch von Leuten mit viel Ahnung und Spitzenwissen, das hat man ja nicht so häufig im Computergeschäft. Na ja. In den Kapiteln werden Anwendungsbeispiele von Motorola Prozessoren aufgezählt und dargestellt. Des Weitern die Prozessorfamilie bis zum 68020. Auch der Matheprozessor kommt nicht zu kurz.

Wer tiefer einsteigen will, kann die 79 DM investieren, notwendig ist es aber nicht, denn vieles ist und bleibt zu speziell und Hardwaregebunden, für uns als Programmierer ehr nebensächlich.

Das PROZESSOR BUCH zum 68000 - Technik und Programmierung, Data Becker

Dieses Buch hat mir besondere Hilfe im Kapitel über Assemblerbau geliefert. Es half mir auch größtenteils Befehlsadressierungsarten des MC68020 zu erklären. Man kann sagen dass es ein richtig gutes Data Becker Werk, dass es auf jeden fall verdient, im Regal zu stehen, und benutzt zu werden. Das Buch ist schon etwas älter, und ich weiß nicht, ob ein Udpate herausgekommen ist, aber die Informationen zum Motorola-Prozessor sind präzise. Nicht umsonst möchte ich dieses Werk etwas näher beschreiben, welches noch so richtig buchmäßig aussieht und gelesen werden kann. Die Seiten sind prall gefüllt mit Informationen, nicht mit so'm Geblubbere. Es unterteilt sich in 8 Kapitel, die folgenden Themengebiete abhandeln:

I	Vorwort (ja, dies ist auch ein Kapitel für sich)
II	Entwicklung zum 68000 (11 Seiten)
III	Der Aufbau des 68000 (38 Seiten)
IV	Signal- und Busbeschreibung (46 Seiten)
V	Weitere Prozessoren der Familie (44 Seiten)
VI	Peripheriebausteine (22 Seiten)
VII	Der Befehlssatz (satte 300 Seiten)
VIII	Der 68000 im Betriebssystem (28 Seiten)
IX	Programmierbeispiele (12 Seiten)

Ein den ersten Kapiteln erfährt der wissensbegierige Leser einiges über die Motorola-Prozessoren, und warum sie in den Markt einzogen. Der Aufbau wird detailliert beschrieben, erstmalig entdeckte ich Informationen über den Mikrocode, wie der abgearbeitet wird, und wie der „Saft" fließt. Strom und Signale sind wesentliche Bestandteile des vierten Kapitels, der Bus, wird auseinanderklamüsert. Schlagwörter sind hier Spannungsversorgung und Systemtakt, asynchrone und synchrone Bussteuerung, Interrupt-Steuerung, Buszuteilungssteuerung, Pinbelegung, asynchroner Buszugriff (Lesen, Schreiben, Read-Modify). Viele Skizzen ermöglichen den Vergleich der Pinzustände, Tabellen ermöglichen ein optimales Verständnis z. B. für die Belegung der Interrupt-Leitungen bei bestimmen Interrupt-Masken im Statusregister. Hardwarebastler, die schon immer mal davon träumten, einen Kleincomputer zu bauen, erfahren im sechsten Kapitel über Peripheriebausteine, die parallel zum Motorola Prozessor entwickelt wurden, oder aus der 6800 Reihe stammen, so dass sie an unseren Chipli passen. Begriffe wie IPC, DMA ("Das Macht An"), BIM, BAM (BIM und BAM heißt nicht Bimbo und Bambi!), MMU, MFP, FPP werden ausreichend erklärt, so dass man sich was unten den Wörtern was vorstellen kann. Kapitel fünf widmet sich den Prozessoren 68008 (kurz), 68010 (68012 (noch kürzer)) und dem 68020. Die Erklärungen zum 68020 sind sehr gut, man wird gleich den Wunsch verspüren, einen 20er zu besitzen. Damit es nicht nur bei der Hardware der Prozessoren bleibt, verschafft Kapitel sieben Klarheit über Adressierungsarten, Flags und die Ausführungszeiten. Ein Extraabschnitt von 3 Seiten widmet sich der schnelleren Ausführung von Befehlen des 68010 im Loop Modus durch den Befehls Prefetch. Wofür man sich dieses Buch eigentlich holen sollte ist wegen dem Kapitel acht. Auf fast 300 Seiten sind alle Befehle vom 68000 bis 68020 alphabetisch sortiert. Der Clou bei dieser Sache ist folgende:

- Es wird der Mnemonik erklärt (warum z. B. ABCD ABCD heißt und nicht WILAW, ganz klar, es heißt „Add Binary Coded Decimals (with extended)", und nicht „Weil Ich Lustig Addieren Will")

- die Operation wird allgemein in einer Symbolsprache dargestellt
- die Operandenlänge wird angegeben
- die Flags, die durch den Befehl verändert werden, werden aufgelistet
- eine Erklärung zu den Flags folgt, warum wird bei einem Befehl der Flag so oder so gesetzt
- das Befehlsformat wird angegeben. In keinem anderen Buch fand ich so eine schöne Auflistung, die mich doch glatt motivierte, einen Assembler zu programmieren.
- die im Befehlsformat vorkommenden Kürzel bekommen eine Bedeutung
- abschließend eine Befehlsbeschreibung

Leider haben sich im Befehlsformat ein paar Fehler eingeschlichen. Sollte ein Leser sich dieses Buch zulegen (vielleicht in einer alten, unverbesserten Version), kann er die fehlerhaften Stellen verbessern. Korrigiert werden muss:

Seite	Befehl	Fehler	Korrektur
286	BTST	0000100011——	0000100000——
302	CMP	1011————	
338	EOR	1011————	

Am meisten Kopfverbrechen hat mir folgender Fehler gemacht:

Seite 360, MOVE-Befehl. Nicht, dass die Bitbelegung des Befehls falsch wäre, nein, auch noch die Operandenlängen sind anders. Und dann sitzt man vor seinem Rechner, und grübelt, grübelt, flucht (alles Teufelswerk). Man zweifelt ja immer zuerst am eignen Programm, als an der Quelle. Aber damit anderen Lesern dies nicht auch passiert, hier die Verbesserung: Anstatt "0,0,Länge,ModusY,Reg.Y,Reg.X,ModusX" muss es heißen "0,0,Länge,RegY,ModusY,ModusX,RegX" Auch die Längenangaben sind fehlerhaft, wenn man die nimmt, die im Buch stehen, dann kommen trotzdem andere Befehle raus. (PS. Seite 224, Befehl ADDA, Schreibfehler bei Assemblersyntax, es heißt nicht `ADDA.X "ea",Dn` sondern `ADDA.X "ea",An`)

Programmierhandbücher zu den Amiga Rechnern

AMIGA TECHNICAL REFERENCE SERIE, Addison-Wesley

Die Originalausgabe der Commodore Dokumentation. Immer wieder aktuell und immer wieder verdienen die Bücher der Trilogie das Prädikat gut, nicht zuletzt dadurch, dass Commodore versucht, die Dokus auf dem neusten Stand zu halten. Alle Bände existieren zurzeit in der dritten Auflage, und sie sind nicht so kleine Büchlein, sondern große, die fast DIN A4-Format haben.

Bücher der Reihe sind:

AMIGA ROM KERNEL REFERENCE MANUAL, Include and Autodocs

Dieses Werk gibt es jetzt in der zweiten Auflage. Es enthält alle Funktionsaufrufe und Datenstrukturen. Diesen Umfang macht es 1000 Seiten stark.

Hilfreich ist das Buch beim ständigen Suchen der unterschiedlichsten Structs, unbedingt nötig aber nicht. Alle Strukturen finden sich auch auf den INCLUDE Disk, ausreichende Erklärungen dazu.

AMIGA ROM KERNEL REFERENCE MANUAL, Devices

Auch in diesem Band ist die Release 2 vom OS berücksichtigt. Es enthält alles über die Programmierung von Devices und in einem Zusatzkapitel alles über die Techniken des Testens und Programm-Debbugings. Hier füllen die Devices 512 Seiten.

AMIGA ROM KERNEL REFERENCE MANUAL, Libraries

Obwohl die Libraries und Devices gerne zusammengewürfelt werden, ist die Libraryprogrammierung in einem extra Band von 960 Seiten erscheinen. Rein Technisch würden 512+960=1472 Seiten auch etwas hoch aussehen. Dieses bullige Band enthält Informationen über alle internen und externen Commodore Libraries. Von 1.3 bis zu 2.0 hat sich die Anzahl ja verdoppelt!

AMIGA HARDWARE REFERNCE MANUAL

Dieser Band gehört mit zu den einzigen Informationsquellen, wo der Programmierer etwas über die Hardware erfährt. Auf 512 Seiten ist die Amigahardware bis zum 3000er beschrieben. Die letzten Spezial-Kapitel widmen sich des Zoro-Buses. Es sei allerdings gleich gesagt, das sich die Informationen, obwohl oft sehr detailliert, im Rahmen halten (nein, das widerspricht sich nicht unbedingt!!), und den Hardware-Hackern nicht alle Wege eröffnet werden. Das dürfte nicht Commodores Ziel dieses Werkes sein. Grafik uns Sound werden groß geschrieben, das Disklaufwerk geht aber fast unter, die paar Seiten sind wirklich etwas wenig. Noch nicht einmal eine Dekodierroutine, die man sich wünschen würde. Und der Matheproz., überhaupt nichts, traurig.

AMIGA intern, Data Becker

Die Intern-Reihe ist wohl beim VC20 angefangen über den C64 bis zum Amiga fortgesetzt worden. Der Preis von 100 DM ist für 1100 Seiten zwar happig, aber das Einschreiben als Entwickler bei Commodore kostet immerhin fast 1000 DM, wenn man's so sieht, hat man schon gespart. Ein Standartwerk ist es allemal, obwohl für meinen Geschmack zeitweise etwas dürftig, so z. B. an den Strukturen, die nicht minder wichtig sind wie die Betriebssystemaufrufe. Was habe ich davon, wenn diese genau geschildert werden, und Inhalte der Strukturen, die beim Aufruf benötigt werden, unerklärt bleiben? Na ja, OK, dieses Makel kann man übersehen. Spärlich ist ebenfalls die 2.0 Dokumentation, denn da man findet man nicht nur wenig, nein, nix. Als Aufbaubuch kann man da folgendes empfehlen:

AMIGA PROFI KNOW-HOW, Vatta Becker

Komisch, 1020 Seiten kosten hier nur 80 DM, na ja, ist noch kein Standartwerk. Dies kann es aber noch werden, denn im Gegensatz zum Intern findet der Programmierer hier alle Libraries und Strukturen von 2.0. Auch die Hardware ist genügend dargestellt worden so wird er neue A3000 und seine Grafikfähigkeiten geschildert. Man kann das Buch als eine Ergänzung zum Intern ansehen.

Über das Amiga OS3.0 sowie den AA Chipsatz und die Rechner A1200 und A400 existiert noch wenig Straßen-Literatur.

Amiga - Das Programmierhandbuch, Sybex Verlag

In diesem 460 Seiten starken Buch erfährt der Programmierer viel über das Betriebssystem. Alle Beispiele sind in C, eine Umsetzung (falls Gewünscht) dürfte nicht so schwierig ausfallen. Da es in erster Linie ein C Buch ist, ist hier nichts über interne Vorgänge zu finden. Leider nicht mehr so ganz aktuell, es fehlt 2.0 und höher, aber immer noch gut.

Das endgültige AMIGA 500 Handbuch, Dadda Becker

Na ja, ein gutes Einsteigerbuch, für ernste Anwendungen überhaupt nicht geeignet. Es beschreibt gut und deutlich die Workbench, nur, die brachen wir nicht! Wer trotzdem 40 DM opfern will, hat dann wenigstens ein dickes Buch im Regal, das nach was aussieht, zwar kein HC (Hardcover) aber immerhin. Seitenpreis 10 Pf.

Sonstiges zum Amiga

Profi-Tips und Power-Tricks für den Amiga, M&T

Oje, wenn der Peter da nicht schon wieder danebengegriffen hat. So powerig uns profimäßig sind die Tipps nicht, nicht umsonst kostete das Buch auf der Messe nur einen Fünfer. Wer sich einen Amiga frisch kauft, der kann vielleicht was mit anfangen, aber spätestens nach einem Jahr Computerpraxis ist jeder findige Benutzer in der Lage selber so ein Buch zu schreiben.

Das neue Supergrafikbuch, Data Becker

Dieses Buch hat sich zum Ziel gesetzt, das grafische Drumherum zu beschreiben. Wer jetzt Assembler oder C Listings verlangt, der ist enttäuscht, hier wird gezeigt, wies s im guten alten Amiga-BASIC und streckenweise GFA-BASIC geht. Das heißt aber nicht, das dies Buh irgendwie schlecht ist, nein, auf keinen Fall. Die Informationen, die gegeben sind, sind sehr gut, die Schreibweise der Strukturerklärungen habe ich so übernommen, auch viele Informationen konnten verwertet werden.

Worauf das Buch nicht eingeht: 3D-Programmierung, BOBS, Sprites, und eben Assembler oder C Programme und Optimierung in Punkte Geschwindigkeit.

Programmierhandbücher anderer Systeme

Atari Profibuch ST-STE-TT, Sybex Verlag

1500 informative Seiten des (meines Erachtens) besten Ataribuches. Ein Muss für jeden Atarianer, und das Wissen kann einem Amiganer auch nicht schaden. Im Gegensatz zum Amiga Intern ist der Preis von 79 DM auch viel besser (Seitenpreis 5 Pf gegenüber 9 Pf!) Diesem Buch entnahm ich die Informationen über den Mathecoprozessor.

Das endgültige Atari ST Handbuch, Data Becker

Auf jeden Fall ist dieses Buch besser als sein Namensvetter. Für 1000 Seiten bezahlt man lediglich 40 DM, ein Preis, über den man bei weitem nicht meckern kann. Wer also einen Überblick über die Ataris haben will, und wem die Referenzen und Betriebssystemaufrufe beim Profibuch stören, dem sei dieses Werk ans Herz gelegt. Leicht und locker wird das Desktop, TOS und GEM Programmierung und die Hardware dargestellt.

PC intern 3.0, Data Becker

Mit 1400 Seiten Muskelstärke und einer Reserve von 1,5 MB auf Disk geht dieses Buch ins Rennen. Der Organismus des PCs wird von Herz über innere Organe auf der Anatomischen Seite, und seine Software auf der anderen, deutlich beschrieben. Hier findet sich alles von DOS, BIOS über Windows uns sonstiges Interna. Für 1400 Seiten ist 100 DM nicht zu viel verlangt.

Sonstiges

Human Interface Guidelines, engl., Addison-Wesley, 166 Seiten

Ein Buch über die Programmierrichtlinien der Apple-Oberfläche. Apple Computer legt großen Wert auf die einheitliche Bedienung und auf das gleiche Aussehen unterschiedlicher Applikationen, um dadurch ein neues Anlernen möglichst zu vermeiden. Dieses Richtlinienbuch beschreibt deutlich, wie Icons uns Fenster auf dem Schirm dargestellt werden sollen, wie die Farbpalette zu wählen ist, und wie groß Objekte zu sein haben. Das Buch ist gut illustriert.

AMIGA user inerface style guide, Addison-Wesly, 200 Seiten

Der Amiga versucht auch im Bereich grafische Oberflächen Zeichen zu setzen. Eine einheitliche Benutzerführung ist da ohne Frage Ziel jedes Projektes. Wie Apple-Computer brachte auch Commodore-Amiga ein Richtlinienbuch heraus, wie die Oberfläche zu designen ist. Das sich vieles um 2.0 dreht dürfte klar sein.

DUDEN Informatik, B.I.-Wissenschafsverlag

Ein Duden, wie es der Mensch kennt, Stichwörter und Erklärungen. Auf jedenfalls sollte es ein Standardwerk im Bücherregal sein. Man findet wichtige Algorithmen, Übersicht über Assembler-Mnemoniks verschiedener Prozessoren, Betriebssystemvergleiche und eine Menge mehr. Im Anschluss des Buches findet der Leser ein Register, in dem auch die nicht aufgezählten Stichwörter einem Überbegriff zugeordnet werden. Ca. 500 Abbildungen auf 770 Seiten machen das Buch fast zum Comic.

Die Geheimnis der der Macintosh-Programmierung, Addison-Wesley

Mit Freude möchte ich meinen Favoriten vorstellen: Scott Knaster und sein Informationsbuch über die Interna der Macs. Ein super Buch mit mehr als 400 Seiten, in einem lockeren, witzigen Stiel geschrieben. Immer wieder Anekdoten und Geheimnisse des Rechners, die es spannend machen, dieses Buch zu lesen. So kommt viel zu Tage, z. B. das der Programmcode vom Typenraddruckertreiber beim Umzug verloren ging. Tja, alles kann dann gegen Mac verwendet werden ...

Liste der Sprungbefehle !nur! für das OS 2.0

```
jmp $F829F8
jmp $F829F8
jmp $F829F8
jmp $F829F8
jmp $F80BEE
jmp $F80C42
jmp $F8269A
jmp $F8268C
jmp $F826BE
jmp $F8269E
jmp $F81808
jmp $F80CAE
jmp $F82C94
jmp $F821B8
```

```
jmp $F821AE
jmp $F821AC
jmp $F821A8
jmp $F81BB4
jmp $F81C64
jmp $F82CB8
jmp $F8093A
jmp $F808EC
jmp $F808D0
jmp $F808A0
jmp $F80C4C ; CacheControll
jmp $F80BE4 ; CacheClearE
jmp $F80BE4 ; CacheClearU
jmp $F829FC ; CopyMemQuick
jmp $F82A00 ; CopyMem
jmp $F81F26 ; AddMemList
jmp $F80E48 ; SumKickstart
jmp $F82C88 ; RemSemaphore
jmp $F82C76 ; AddSemaphore
jmp $F82C8C ; FindSemaphore
jmp $F82C60 ; ReleaseSemaphoreList
jmp $F82BEA ; ObtainSemaphoreList
jmp $F82BBE ; AttemptSemaphore
jmp $F82B20 ; ReleaseSemaphore
jmp $F82ACC ; ObtainSemaphore
jmp $F82AA8 ; InitSemaphore
jmp $FC80B2 ; OpenLibrary
jmp $F82A86 ; Vacate
jmp $F82A70 ; Procedure
jmp $F81D16 ; TypOfMem
move CCR,D0 ; GetCC
rts
dc $1578
jmp $F827CC ; RawDoFmt
jmp $F82942 ; RawPutChar
jmp $F8291A ; RawMayGetChar
jmp $F82910 ; RawIOInit
jmp $F821C8 ; OpenResource
jmp $F821C4 ; RemResource
jmp $F821BC ; AddResource
jmp $F80892 ; AbortIO
jmp $F8081E ; WaitIO
jmp $F8087A ; CheckIO
jmp $F807F6 ; SendIO
jmp $F80808 ; DoIO
jmp $FC803C ; CloseDevice
jmp $FC807C ; OpenDevice
jmp $FC8034 ; RemDevice
jmp $F80778 ; AddDevice
jmp $F819F2 ; SumLibrary
jmp $F819B8 ; SetFunction
jmp $FC8040 ; CloseLibrary
jmp $F8196C ; OldOpenLibrary
jmp $FC8038 ; RemLibrary
jmp $F819E6 ; AddLibrary
jmp $F8219E ; FindPort
jmp $F82176 ; WaitPort
jmp $F820AA ; ReplyMsg
jmp $F82148 ; GetMsg
jmp $F820BC ; PutMsg
jmp $F820A6 ; RemPort
jmp $F82090 ; AddPort
jmp $F8260E ; FreeTrap
jmp $F825EE ; AllocTrap
jmp $F82650 ; FreeSignal
jmp $F82618 ; AllocSignal
jmp $F82448 ; Signal
jmp $F824CC ; Wait
jmp $F82418 ; SetExcept
jmp $F82422 ; SetSignal
jmp $F823C8 ; SetTaskPri
jmp $F8236C ; FindTask
jmp $F822CC ; RemTask
jmp $F821D8 ; AddTask
jmp $F8192A ; FindName
jmp $F81904 ; Enqueue
jmp $F818E4 ; RemTail
jmp $F818D4 ; RemHead
jmp $F818C6 ; Remove
jmp $F8189C ; AddTail
jmp $F8188C ; AddHead
jmp $F81864 ; Insert
jmp $F81EE8 ; FreeEntry
jmp $F81E54 ; AllocEntry
jmp $F81DD2 ; AvailMem
```

```
 jmp $F81BBE ; FreeMem
 jmp $F81D3A ; AllocAbs
 jmp $FC7F74 ; AllocMem
 jmp $F81B10 ; Deallocate
 jmp $F81C02 ; Allocate
 jmp $F81728 ; Cause
 jmp $F81634 ; RemIntServer
 jmp $F815F4 ; AddIntVector
 jmp $F815AE ; SetIntVector
 jmp $F815A2 ; UserState
 jmp $F81578 ; SuperState
 jmp $F8155C ; SetSR
 jmp $F82578 ; Permit
 jmp $F82572 ; Forbit
 jmp $F817F8 ; Enable
 jmp $F817EA ; Disable
 jmp $F82D44 ; Debug
 jmp $F8399E ; Alert
 jmp $F80F38 ; InitResisent
 jmp $F80ECE ; FindResistent
 jmp $F81AD0 ; MakeFunctions
 jmp $F81A46 ; MakeLibrary
 jmp $F80FFE ; InitStruct
 jmp $F80EFE ; InitCode
 jmp $F8141E ; Execption
 jmp $F8137A ; Dispatch
 jmp $F8132C ; Switch
 jmp $F82550 ; Reschedule
 jmp $F812D2 ; Schedule
 jmp $F812B0 ; ExitIntr
 jmp $F80AF4 ; Supervisor
 jmp $F829F8 ; LIB_EXTFUNC
 jmp $F829F8 ; LIB_EXPLUNGE
 jmp $F829F4 ; LIB_CLOSE
 jmp $F829EC ; LIB_OPEN
```

Aussschnitt aus dem ROM des Amiga 500+

Im folgenden habe ich das ROM meines Amiga 500+ etwas auseinandergenommen. Vielleicht noch einige Zusatzbemerkungen zur Dokumentation: Die Funktionen des Exec sind größtenteils aufgelistet. Zuerst findet man (gefolgt nach drei führenden Sternen) die Betriebssystemfunktion und ihre Sprungadresse. Nach einer Leerzeile sind mitunter Kommentare angefügt. Einige Unterprogramme beginnen nicht mit **. Sie stehen für Funktionen, die es nicht als Betriebssystemfunktion gibt, die allerdings trotzdem intern gebraucht werden. Sie beginnen mit dem Kürzel ++ (Plus wie zugefügt.)

Ist im Text ein SNIP* zu finden, so war das umsetzten zu undurchsichtig und aufwändig. Aus diesem Grund habe ich die Teile weggelassen.

```
*** SumKickData                $f80e48

* Prüfsumme der KickMem-Struct und des KickTagPtr-Eintrages

SumKickData     movem.l d2-d4,-(SP)
        lea     KickMemPtr(a6),a0 ; Zeiger auf Mem, der beim
                        ; Reset wieder belegt wird
        movem.l (a0),d3/d4      ; MemList -> d3; KickTagPtr
        clr.l   (a0)+   ; MemList löschen
        clr.l   (a0)+   ; KickTagPtr löschen
        moveq   #-1,d0  ; hier kommt die Checksumm rein
        move.l  d3,d2   ; Zeicher auf MemListe= D2
SumKickDatLoop  tst.l   d2      ; gibt es Speicher, der bei einem
                        ; Reset belegt wird?
        beq.s   SKDNoMemList
        move.l  d2,a0   ; MemList als Zeiger nach A0
        move.l  ml_Succ(a0),d2  ; nächsten
        move    ml_NumEntries(a0),d1 ; Anzahl der Einträge
        add     d1,d1   ; mal zwei
        add     #4,d1   ; und vier dazu
        bsr     SKickMemCheck   ; Checksumme berechnen
        bra.s   SumKickDatLoop

SKDNoMemList    move.l  d4,d2   ; KickTagPrg nach D2 holen
        beq.s   SKDNoTagPtr     ; keine Resident-Tabelle
        move.l  d2,a0   ; und zum Zeiger machen
        bra.s   SKDTagInLoop

SKDTagResLoop   add.l   d2,d0   ; Resident Modul zur Chechsumm
                        ; addieren
SKDTagInLoop    move.l  (a0)+,d2
        beq.s   SKDNoTagPtr
        bpl.s   SKDTagResLoop
        bclr    #%11111,d2
        move.l  d2,a0
        bra.s   SKDTagInLoop
```

```
SKDNoTagPtr     movem.l d3/d4,KickMemPtr(a6) ; KickMemPrt und
                        ; KickTagPtr aktualliesieren,
                        ; nicht aber KickCheckSum!
        movem.l (SP)+,d2-d4
        rts

SKickMemCheck   add.l   (a0)+,d0        ; Checksumme in D0
SumKickMemEnd   dbra    d1,SumKickMemLoop

        rts


/* Schnipp */


*** Resident(D0)=FindResident(Name)(A1)          $f80ece

* suche residentes Modul anhand des Namens

FindResident    movem.l a2/a3,-(SP)
        move.l  ResModule(a6),a2 ; Zeiger auf alle ResModule
        move.l  a1,a3   ; der zu suchende Name in A3
FindResLoop     move.l  (a2)+,d0        ; ResModul
        beq.s   FindResEnd      ; keiner mehr da
        bgt.s   FindResFind
        bclr    #%11111,d0      ; untersten 5 Bits löschen
        move.l  d0,a2   ; wegen Verkettung nächsten
        bra.s   FindResLoop

FindResFind     move.l  d0,a1   ; Zeiger von Node
        move.l  a3,a0   ; zu suchender Name in A0
        move.l  rt_Name(a1),a1  ; Resident Name aus Node
FindResStrCmp   cmp.b   (a0)+,(a1)+
        bne.s   FindResLoop
        tst.b   -1(a0)  ; stimmt auch das letzte Zeichen?
        bne.s   FindResStrCmp
FindResEnd      movem.l (SP)+,a2/a3     ; alles zurückgeben
        rts

; erstaunlicherweise findet sich hier noch ein zeichenweiser Vergleich, die
; Routine FindName kann nicht aufgerufen werden.


*** InitCode(StartClass,Version)(D0,D1)          $f80efe

* Initialisiert alle Residen-Module, deren Versiosnummer größer oder gleich
* der angegebenen ist, mit Resident_Flags

InitCode        movem.l d2/d3/a2,-(SP)
        move.l  ResModules(a6),a2 ; Zeiger auf die residente
                        ; Module, die beim Reset
                        ; aufgerufen werden
        move.b  d0,d2   ; Resident-Flags nach d2
        move.b  d1,d3   ; Version nach d3
InitCodeLoop    move.l  (a2)+,d0        ; erste Node
        beq     InitCodeEnd     ; nichts zu holen
        bgt.s   InitCodeFind    ; und was gefunden
        bclr    #%11111,d0
        move.l  d0,a2   ; Nachfolger
        bra.s   InitCodeLoop    ; verzweigen

InitCodeFind    move.l  d0,a1   ; ein gefundenes ResMod
        cmp.b   rt_version(a1),d3 ; übergenene Version mit
                        ; mit der Resident-Struct vergl.
        bgt.s   InitCodeLoop    ; größer dann weiter
        move.b  rt_Flags(a1),d0
        and.b   d2,d0   ; Mit den Flags vergleichen
        beq.s   InitCodeLoop    ; stimmen die Flags mit überein
        ; dann muß das Modul neu initialisiert werden
        moveq   #0,d1   ; keine Segmentliste
        jsr     InitResident(a6) ; und Resident-Modul initialisieren
        bra.s   InitCodeLoop    ; bis zum bitteren Ende

InitCodeLoop    movem.l (SP)+,d2/d3/a2
        rts



*** InitResident(Resident,SegList)(A1,D1)                $f80f38

InitResident    btst    #7,rt_Flags(a1) ; RTF_AUTOINIT gesetzt?
        bne.s   IResNoAutoInit
        move.l  rt_Init(a1),a1  ; Zeiger auf Startcode
        moveq   #0,d0
```

```
        move.l  d1,a0   ; Segmentliste übergeben
        jmp     (a1)    ; und er startet sich selber

IResNoAutoInit  movem.l d1/a1/a2,-(SP)  ; beliebtes Spiel
        move.l  rt_Init(a1),a1  ; Zeiger auf AutoStart Prg.
        movem.l (a1),d0/a0/a1   ; für Library DataSize,
                        ; FuncInit und StructInit
        sub.l   a2,a2   ; keine Lib-Init
        jsr     MakeLibrary(a6)
        movem.l (SP)+,d1/a0/a2
        move.l  d0,-(SP)
        beq.s   InitResEnd
        move.l  d0,a1
        move.b  12(a0),8(a1)
        move.l  14(a0),10(a1)
        move.b  #6,14(a1)
        move.b  11(a0),$15(a1)
        move.l  $12(a0),$18(a1)
        move.l  a0,-(SP)
        move.l  $16(a0),a0
        move.l  12(a0),d0
        beq.s   f80F9A
        exg     d0,a1
        move.l  d1,a0
        jsr     (a1)
        move.l  d0,4(SP)
f80F9A  move.l  (SP)+,a0
        move.l  (SP),d0
        beq.s   InitResEnd
        move.l  d0,a1
        move.b  12(a0),d0
        cmp.b   #3,d0
        bne.s   f80FB2
        jsr     AddDevice(a6)
        bra.s   InitResEnd

f80FB2  cmp.b   #9,d0
        bne.s   f80FBE
        jsr     AddLibrary(a6)
        bra.s   InitResEnd

f80FBE  cmp.b   #8,d0
        bne.s   InitResEnd
        jsr     AddResource(a6)
InitResEnd      move.l  (SP)+,d0
        rts


/* Schnipp */


*** InitStruct(InitTable,Memory,Size)(A1,A2,D0) $f80ffe

* Speicher nach einer Tabelle initen
* InitTable besteht aus Befehlsbytes und Daten
* Befehlsbyte in Hi- und LoNibble.
* HiNibble  = Befehle in den oberen zwei Bits:
* 00 = Daten kopieren, 01 = Daten "Lo" mal kopieren
* 10 = Datenword Offset, 11 = 24-Bit Offset
* unteren Bits, Datengröße
* 00 = Long, 01 = Word, 10 = Byte, 11 = Absturz
* LoNibble = Anzahl Befehlsausführungen

* BeDaRept
* 76543210
        move.l  A2,A0   ; Speicher nach a0
        lsr     D0      ; Größe durch zwei
        bra     IS_ClrMem

IS_ClrMemLoop   clr     (A0)+
IS_ClrMem       dbra    D0,IS_ClrMemLoop

        move.l  A2,A0   ; Speicher noch mal nach A0
IS_EntryLoop    clr     D0      ; schon mal löschen
        move.b  (A1)+,D0        ; Token aus InitTable lesen
        beq     IS_EndOfTable   ; ist die Tabelle zu Ende?
        bpl     IS_NoOffset     ; ist das oberste Bit gesetzt?
                        ; prüfen, ob im Befehlsbyte
                        ; Offset-Kombination
        bclr    #6,D0   ; sechste Bit löschen, damit
                        ; herausbekommen, ob 10 oder 11
        beq     IS_Offset10     ; es war der Befehl 10
        subq.l  #1,A1   ; Tabelle -1
        move.l  (A1)+,D1        ; Datenwort holen
        and.l   #$FFFFFF,D1     ; nur 24 Bit bearbeiten
```

```
        bra     IS_Schreibe

IS_Offset10     moveq   #0,D1    ; hier kommt der Offset rein
        move.b  (A1)+,D1        ; Daten lesen
IS_Schreibe     move.l  A2,A0    ; Startadresse der Tabelle
        add.l   D1,A0   ; zum Offset addieren
IS_NoOffset     move    D0,D1   ; das Token ohne 6. Bit nach d1
        lsr     #3,D1   ; vier Bits verschwinden,
                        ; genauer gesagt: die Repeat-Anz
        and     #%1110,D1
        move    IS_Tab(PC,D1),D1 ; einer der acht Befehle
        and     #%1111,D0       ; Repeat-Wert, Befehl ausmaskieren
        jmp     IS_Tab(PC,D1)

IS_CopyMem      move.b  (A1)+,(A0)+
        dbra    D0,IS_CopyMem

        move.l  A1,D0
        addq.l  #1,D0
        bclr    #0,D0
        move.l  D0,A1
        bra     IS_EntryLoop

        add     D0,D0
        addq    #1,D0
        move.l  A1,D1
        addq.l  #1,D1
        and.b   #$FE,D1
        move.l  D1,A1
IS_Copymem      move    (A1)+,(A0)+
        dbra    D0,IS_Copymem

        bra     IS_EntryLoop

IS_EndOfTable   rts

IS_Tab  dc      $FFE8
        dc      $FFEC
        dc      $FFD6
        dc      $FFFE
        dc      $FF6A
        dc      $FF7E
        dc      $FF60
        dc      $FFFE


        movem.l D0/D1/A0/A1/A5/A6,-(SP)
        lea     $DFF000,A0      ; Chip Basisadresse
        move    INTENAC(A0),D1  ; Interrupt-Enable-Register
        btst    #14,D1  ; Ist Master-Interrupt gesetzt?
        beq     f8110A  ; nein
        and     INTREQR(A0),D1
        move.l  4,A6
        btst    #0,D1
        beq     f810AC
        movem.l IntVector(A6),A1/A5 ;  Pri 1, Int 1
        move.l  A6,-(SP)        ; Basereg. sichern
        pea     ExitIntr(A6)
        jmp     (A5)    ; und iv_Code ausführen


f810AC  btst    #1,D1
        beq     f810C0
        movem.l (IntVector+$c)(A6),A1/A5 ; Pri 1, Int 2
        move.l  A6,-(SP)
        pea     ExitIntr(A6)
        jmp     (A5)    ; iv_Code ausführen

f810C0  btst    #2,D1
        beq     f81104
        movem.l $6C(A6),A1/A5   ; Pri 1, Int 3
        move.l  A6,-(SP)
        pea     ExitIntr(A6)
        jmp     (A5)

        movem.l D0/D1/A0/A1/A5/A6,-(SP)
        lea     $DFF000,A0
        move    $1C(A0),D1
        btst    #14,D1
        beq     f8110A
        and     $1E(A0),D1
        move.l  4,A6
        btst    #3,D1
        beq     f81104
        movem.l $78(A6),A1/A5   ; Pri 2, Int 1
```

```
        move.l  A6,-(SP)
        pea     ExitIntr(A6)
        jmp     (A5)

f81104  movem.l (SP)+,D0/D1/A0/A1/A5/A6
        rte

f8110A  movem.l (SP)+,D0/D1/A0
        add     #12,SP
        rte

        movem.l D0/D1/A0/A1/A5/A6,-(SP)
        lea     $DFF000,A0
        move    $1C(A0),D1
        btst    #14,D1
        beq     f8110A
        and     $1E(A0),D1
        move.l  4,A6
        btst    #6,D1
        beq     f81144
        movem.l $9C(A6),A1/A5   ; Pri 3, Int 3
        move.l  A6,-(SP)
        pea     ExitIntr(A6)
        jmp     (A5)

f81144  btst    #5,D1
        beq     f81158
        movem.l $90(A6),A1/A5   ; Pri 3, Int 2
        move.l  A6,-(SP)
        pea     ExitIntr(A6)
        jmp     (A5)

f81158  btst    #4,D1
        beq     f81104
        movem.l $84(A6),A1/A5   ; Pri 3, Int 1
        move.l  A6,-(SP)
        pea     ExitIntr(A6)
        jmp     (A5)

        movem.l D0/D1/A0/A1/A5/A6,-(SP)
        lea     $DFF000,A0
        move    $1C(A0),D1
        btst    #14,D1
        beq     f8110A
        and     $1E(A0),D1
        move.l  4,A6
f81188  btst    #8,D1
        beq     f8119E
        movem.l $B4(A6),A1/A5   ; Pri 4, Int 2
        move.l  A6,-(SP)
        pea     $F811E6
        jmp     (A5)

f8119E  btst    #10,D1
        beq     f811B4
        movem.l $CC(A6),A1/A5   ; Pri 4, Int 4
        move.l  A6,-(SP)
        pea     $F811E6
        jmp     (A5)

f811B4  btst    #7,D1
        beq     f811CA
        movem.l $A8(A6),A1/A5   ; Pri 4, Int 1
        move.l  A6,-(SP)
        pea     $F811E6
        jmp     (A5)

f811CA  btst    #9,D1
        beq     f811E0
        movem.l $C0(A6),A1/A5   ; Pri 4, Int 3
        move.l  A6,-(SP)
        pea     $F811E6
        jmp     (A5)

f811E0  movem.l (SP)+,D0/D1/A0/A1/A5/A6
        rte

        move.l  (SP)+,A6        ; $f811e6
        lea     $DFF000,A0
        move    #$780,D1
        and     $1C(A0),D1
        and     $1E(A0),D1
        bne     f81188
        move.l  A6,-(SP)
        jmp     ExitIntr(A6)
```

```
f81202  movem.l (SP)+,D0/D1/A0
        add     #12,SP
        rte

        movem.l D0/D1/A0/A1/A5/A6,-(SP)
        lea     $DFF000,A0
        move    $1C(A0),D1
        btst    #14,D1
        beq     f81202
        and     $1E(A0),D1
        move.l  4,A6
        btst    #12,D1
        beq     f8123C
        movem.l $E4(A6),A1/A5   ; Pri 5, Int 2
        move.l  A6,-(SP)
        pea     ExitIntr(A6)
        jmp     (A5)

f8123C  btst    #11,D1
        beq     f81250
        movem.l $D8(A6),A1/A5   ; Pri 5, Int 1
        move.l  A6,-(SP)
        pea     ExitIntr(A6)
        jmp     (A5)

f81250  movem.l (SP)+,D0/D1/A0/A1/A5/A6
        rte

        movem.l D0/D1/A0/A1/A5/A6,-(SP)
        lea     $DFF000,A0
        move    $1C(A0),D1
        btst    #14,D1
        beq     f81202
        and     $1E(A0),D1
        move.l  4,A6
        btst    #14,D1
        beq     f81286
        movem.l $FC(A6),A1/A5   ; Pri 6, Int 2
        move.l  A6,-(SP)
        pea     ExitIntr(A6)
        jmp     (A5)

f81286  btst    #13,D1
        beq     f81250
        movem.l $F0(A6),A1/A5   ; Pri 6, Int 1
        move.l  A6,-(SP)
        pea     ExitIntr(A6)
        jmp     (A5)

        movem.l D0/D1/A0/A1/A5/A6,-(SP)
        move.l  4,A6
        movem.l $108(A6),A1/A5
        jsr     (A5)
        movem.l (SP)+,D0/D1/A0/A1/A5/A6
        rte


*** ExitIntr                            $f812b0

ExitIntr        move.l  (SP)+,A6
        btst    #5,$18(SP)
        bne     f812C6
        tst.b   TdNestCnt(A6)
        bge     f812C6
        tst     SysFlags(A6)
        bmi     f812CC
f812C6  movem.l (SP)+,D0/D1/A0/A1/A5/A6
        rte

f812CC  move    #0,SR
        bra     Schedule_2


*** Schedule                            $f812d2

* Zeitverteilung berechnen

Schedule        movem.l D0/D1/A0/A1/A5/A6,-(SP)
Schedule_2      move.l  ThisTask(A6),A1 ; für unseren Task
        move    #0,SR   ; kein SR
        bclr    #7,SysFlags(A6); ???
        btst    #EXCEPT,tc_Flags(A1)
        bne     f8130A
        lea     TaskReady(A6),A0
```

```
        cmp.l   8(A0),A0
        beq     f812C6
        move.l  (A0),A0
        move.b  9(A0),D1
        cmp.b   9(A1),D1
        bgt     f8130A
        btst    #6,SysFlags(A6)
        beq     f812C6
f8130A  lea     TaskReady(A6),A0
        bsr     Enqueue
        move.b  #TS_READY,tc_State(A1) ; Task
        move    #0,SR
        movem.l (SP)+,D0/D1/A0/A1/A5
        move.l  (SP),-(SP)
        move.l  -$34(A6),4(SP)
        move.l  (SP)+,A6
        rts


*** Switch()                            $f8132c

* Task-Umschaltung

Switch  move    #0,SR
        move.l  A5,-(SP)
        move    USP,A5
        movem.l A0-A6,-(A5)
        movem.l D0-D7,-(A5)
        move.l  4,A6
        move    IDNestCnt(A6),D0
        move    #-1,IDNestCnt(A6)
        move    #$C000,INTENA
        move.l  (SP)+,$34(A5)
        move    (SP)+,-(A5)
        move.l  (SP)+,-(A5)
f8135A  move.l  $230(A6),A4
        move.l  ThisTask(A6),A3
        move    D0,tc_IDNestCnt(A3) ; Anz der unterb. Tasks
        move.l  A5,tc_SPReg(A3)  ;
        btst    #6,tc_Flags(A3)  ; TF_SWITCH
        beq     f8138C
        move.l  tc_ExpectCode(A3),A5
        jsr     (A5)
        bra     f8138C


*** Dispatched                          $f8137a

Dispatched      move.l  $230(A6),A4
        move    #-1,IDNestCnt(A6)
        move    #$C000,INTENA
f8138C  lea     TaskReady(A6),A0
f81390  move    #0,SR
        move.l  (A0),A3
        move.l  (A3),D0
        bne     f813AA
        addq.l  #1,IdleCount(A6)
        bset    #7,SysFlags(A6)
        ILLEGAL ; STOP
        move.l  D0,D0
        bra     f81390

f813AA  move.l  D0,(A0)
        move.l  D0,A1
        move.l  A0,4(A1)
        move.l  A3,ThisTask(A6)
        move    Elapsed(A6),Quantum(A6) ; Prozessorzeit jedes
                        ; Tasks in die bisher gebrauchte
                        ; Rechenzeit
        bclr    #6,SysFlags(A6)
        move.b  #2,15(A3)
        move    $10(A3),IDNestCnt(A6)
        tst.b   IDNestCnt(A6)
        bmi     f813DC
        move    #$4000,INTENA
f813DC  move    #0,SR
        addq.l  #1,DispCount(A6)
        move.b  14(A3),D0
        and.b   #$A0,D0
        beq     f813F0
        bsr     f81406
f813F0  move.l  $36(A3),A5
        jmp     (A4)

        lea     $42(A5),A2
```

```
        move    A2,USP
        move.l  (A5)+,-(SP)
        move    (A5)+,-(SP)
        movem.l (A5),D0-D7/A0-A6
        rte

f81406  btst    #7,D0
        beq     f81416
        move.b  D0,D2
        move.l  $46(A3),A5
        jsr     (A5)
        move.b  D2,D0
f81416  btst    #5,D0
        bne     f8141E
        rts


*** Exception                   ; $141e

Exception
        bclr    #5,14(A3)
        move    #$4000,INTENA
        addq.b  #1,IDNestCnt(A6)
        move.l  $1A(A3),D0
        and.l   $1E(A3),D0
        Eor.l   D0,$1E(A3)
        Eor.l   D0,$1A(A3)
        subq.b  #1,IDNestCnt(A6)
        bge     f8144E
        move    #$C000,INTENA
f8144E  move.l  $36(A3),A1
        move.l  14(A3),-(A1)
        tst.b   IDNestCnt(A6)
        bne     f8146A
        subq.b  #1,IDNestCnt(A6)
        bge     f8146A
        move    #$C000,INTENA
f8146A  move.l  #$F8148A,-(A1)
        move    A1,USP
        btst    #0,(AttnFlags+1)(A6)
        beq     f8147E
        move    #$20,-(SP)
f8147E  move.l  $2A(A3),-(SP)
        clr     -(SP)
        move.l  $26(A3),A1
        rte

        move.l  4,A6
        lea     f81496(PC),A5
        jmp     Supervisor(A6)

f81496  move.l  $230(A6),A4
        btst    #0,(AttnFlags+1)(A6)
        beq     f814A4
        addq.l  #2,SP
f814A4  addq.l  #6,SP
        move.l  ThisTask(A6),A3
        or.l    D0,$1E(A3)
        move    USP,A1
        move.l  (A1)+,14(A3)
        move.l  A1,$36(A3)
        move    $10(A3),IDNestCnt(A6)
        tst.b   IDNestCnt(A6)
        bmi     f814CC
        move    #$4000,INTENA
f814CC  rts


/* Schnipp */


*** OldSR(D0) = SetSR(NewSR,Mask)(D0,D1)                $f8155c

* Setzt das Status-Register, Mask ist die Maske der Bits,
* die gesetzt werden

SetSR   move.l  A5,A0
        lea     _SetSRPrg(PC),A5 ; Programm im
        jmp     Supervisor(A6) ; Super-Mode ausführen

_SetSRPrg       move.l  A0,A5   ; wieder zurückgeben
        move    (SP),A0 ; Statusreg. in A0
        and     D1,D0   ; die Mask-Bits setzen die
                        ; anderen
        not     D1      ; negerieren
```

```
        and     D1,(SP) ; und neu sezten
        or      D0,(SP) ; und vertiefen
        moveq   #0,D0   ; Null zurückgeben
        move    A0,D0   ; alte Statuswerte in D0
                        ; übergeben
        rte             ; mit rte SR setzen


*** OldSysStack(D0) = SuperState()              $f81578

* geht mit dem User-Stack in den Supervisor-Mode

SuperState      move.l  a5,a0   ; a5 wird benötigt, daher sichern
        lea     _SuperStatePrg(PC),a5
        jmp     Supervisor(A6)


*++ Programm, das im Supervisor-Mode ausgeführt wird    $f81582

_SuperStatePrg  move.l  A0,A5
        clr.l   D0      ; hier kommt der SP rein
        bset    #5,(SP) ; Supervisor-Bit (Bit 13) setzen
        bne     SuperStateSet   ; schon gesetzt
        move    (SP)+,SR        ; Statusregister setzen
        move.l  SP,D0   ; alten Stackpointer sichern
        move    USP,SP  ; User-Stack ist jetzt
                        ; Super-Stack
        btst    #0,AttnFlags+1(A6) ; welchen Prozessor
                        ; haben wir?
        beq     SSP_NormProz    ; einen normalen 68000
        addq.l  #2,D0   ; Stack für andere Prozessoren
                        ; ist größer
SSP_NormProz    addq.l  #4,D0   ; das reicht aus
        rts             ; gebe in D0 alten Stack zurück

SuperStateSet   rte


*** UserState(SysStack)(D0)                     $f815a2

* Zurück zum Unser-Status

UserState       move.l  (SP)+,A0        ; Sprungadresse
        move    SP,USP  ; Stackpointer ist
        move.l  D0,SP   ; den Stackpointer neu setzen
        and     #%1101111111111111,SR ; Supervisor-Bit
                        ; ausmaskieren, also löschen
        jmp     (A0)    ; und zum Prg springen


************ Interrupt-Funktionen ******************************

*** SetIntVector(IntNr,Interrupt)(D0,A1)                $f815ae

* ein neuer Interrupt wird eingesetzt, der alte gelöscht


SetIntVector    mulu    #12,D0  ; Int-Vector länge
        lea     IntVector(A6,D0),A0 ; Interrupt holen
        move    #$4000,INTENA   ; Ints abschalten
        addq.b  #1,IDNestCnt(A6) ; und kenntlich machen
        move.l  iv_Node(A0),D0  ; Int-Node holen
        move.l  A1,iv_Node(A0)  ; Neuer Int eintragen
        beq     f815DA
        move.l  is_Data(A1),iv_Data(A0) ; aus Interrupt nach
                        ; Int-Vector
        move.l  $12(A1),4(A0)
        bra     f815E4

f815DA  moveq   #-$1,D1
        move.l  D1,0(A0)
        move.l  D1,4(A0)
f815E4  subq.b  #1,IDNestCnt(A6)
        bge     f815F2
        move    #$C000,INTENA
f815F2  rts



**** AddIntServer(IntNumber,Interrupt)(D0,A1)   $f815f4

* dem Sytsem einen Interrupt-Server anfügen. IntNr entspricht den
* Interrupts mit der Priorität (0-15).
* 5 z. B. Rasterstrahlinterrupt
```

```
AddIntServer    move.l  D2,-(SP)
        move.l  D0,D2   ; Int Nummer nach d2
        move.l  D0,D1   ; und nach d1
        mulu    #12,D0  ; mal 12 = sizeof(IntVector)
        lea     IntVector(A6,D0),A0 ; Zeiger auf Interrupt
        move.l  (A0),A0 ; Interruptzeiger in A0 holen
        move    #$4000,INTENA   ; Interrupts sperren
        addq.b  #1,IDNestCnt(A6) ; Interrupts sperren, Disable
        bsr     Enqueue ; Int-Node in Liste einsortieren
        move    #$8000,D0       ; Grundmaske 15 Bit gesetzt,
                        ; d. h. Daten schreiben
        bset    D2,D0   ; Interrupt-Nummer
        move    D0,INTENA       ; und dieses bestimme Bit, also
                        ; den best. Interrupt erlauben
        subq.b  #1,IDNestCnt(A6) ; Interrupts erlauben, Enable
        bge     AISNoIntEnable  ; keine Interrupts erlaubt
        move    #$C000,INTENA   ; Interrupts erlauben
AISNoIntEnable  move.l  (SP)+,D2
        rts


**** RemIntServer(IntNumber,Interrupt)(D0,A1)   $f81634

* Interrupt-Server aus Server-Liste entfernen

RemIntServer    move.l  D2,-(SP)

        move.l  D0,D2   ; Int Nummer nach d2
        mulu    #12,D0  ; mal 12 = sizeof(IntVector)
        lea     IntVector(A6,D0),A0 ; Zeiger auf Interrupt
        move.l  (A0),A0 ; Interruptzeiger in A0 holen
        move.l  A0,D1   ; und in D1
        move    #$4000,INTENA   ; Interrupts sperren
        addq.b  #1,IDNestCnt(A6) ; Interrupts sperren, Disable
        bsr     Remove  ; und die Node entfernen
        move.l  D1,A0   ; Zeiger auf Interrupt
        cmp.l   is_Type(A0),A0  ; Int-Node-Typ aus is_Node
        bne.s   RIS_NichtNull
        moveq   #0,D1   ; Alles löschen
        bset    D2,D1   ; den betreffenden Int setzen
        move    D1,INTENA       ; und die Hardware macht's
RIS_NichtNull   subq.b  #1,IDNestCnt(A6)
        bge     RISNoIntEnable  ; man darf noch nicht
        move    #$C000,INTENA   ; weiter interrupten
RISNoIntEnable  move.l  (SP)+,D2
        rts


/* Schnipp */


**** Cause(Interrupt)(A1)                       $f81728

* löst einen Softwareinterrupt aus

Cause   move    #$4000,INTENA   ; Ints sperren
        addq.b  #1,IDNestCnt(A6) ; Int-Zähler +1
        moveq   #NT_SOFTINT,D0  ; 11 ist der Nodetyp für
Ints
        cmp.b   is_Type(A1),D0  ; ist es überhaupt ein
Iterrupt?
        beq     CauseNoInt      ; Nein, dann hör mal schnell auf
        move.b  D0,ln_Type(A1)  ; NT_SOFTINT in Node
eintragen
        move.b  is_Pri(A1),D0   ; Priorität des Ints
        and     #%11110000,D0   ; ausmaskieren
        ext     D0      ; auf n Long
        lea     SoftIntList+$20(A6),A0 ; IntVektor holen
                        ; Interrupts, mit der Priorität Null
        add     D0,A0   ; auf SoftIntList plus Pri drauf
        addq.l  #4,A0   ; und noch vier dazu
        move.l  4(A0),D0        ; aus der IntList den Int holen
        move.l  A1,4(A0)
        move.l  A0,(A1)
        move.l  D0,4(A1)
        move.l  D0,A0
        move.l  A1,(A0)
        move    #$8004,INTREQ
        bset    #5,SysFlags(A6)
CauseNoInt      subq.b  #1,IDNestCnt(A6)
        bge     CauseNoIntAkt   ; Ints noch gesperrt
        move    #$C000,INTENA   ; und wieder Ints erlauben
CauseNoIntAkt   nop
        rts
```

```
/* Schnipp */


*** Disable()                          $f817ea

* Interrupts verbieten

Disable move    #$4000,INTENA
        addq.b  #1,IDNestCnt(A6) ;
        rts

; Ist IDNestCnt = -1 sind Interrupts erlaubt.
; Task-Switching erlaubt bei IDNestCnt < 0
; Interrupt Disable Nesting Counter

*** Enable()                           $f817f8

* Interrupts wieder erlauben

Enable  subq.b  #1,IDNestCnt(A6)
        bge     IntNichtFrei    ; !=
        move    #$C000,INTENA
IntNichtFrei    rts

; Erst wenn IDNestCnt wieder -1 ist, werden Interrupts wieder ermöglicht

/* Schnipp */


************** Node-Funktionen *******************************

*** Insert(Liste,Node,Vorgänger)(A0,A1,A2)              $f81864

* neue Node (A1) hinter dem Vorgänger (A2) anfügen

Insert  move.l  A2,D0   ; Vorgänger nach d0
        beq     AddHead ; kein Vorgänger, daher AddHead
        move.l  ln_Succ(A2),D0  ; hat Vorgänger-Node
                        ; einen Nachfolger?
        beq     Insert_NoEntry  ; wenn nicht, dann einfach
                        ; hinten anhängen
        move.l  D0,A0   ; Vorgänger-Node nach A0,
                        ; d. h. die Liste wird nicht
                        ; benötigt!
        movem.l D0/A2,(A1)      ; Vorgänger und Nachfolger des
                        ; Vorgängers in einzufüg. Node
        move.l  A1,ln_Pred(A0)  ; die Node ist Nachfolger der
                        ; Vorgänger-Node, und
        move.l  A1,ln_Succ(A2)  ; Nachfolger des Vorgängers
        rts

Insert_NoEntry  move.l  A2,ln_Succ(A1)  ; Nachfolger der Einzufügenden
                        ; ist die Vorgänger-Node
        move.l  ln_Pred(A2),A0  ;
        move.l  A0,ln_Pred(A1)
        move.l  A1,ln_Pred(A2)
        move.l  A1,ln_Succ(A0)
        rts


*** AddHead(Liste,Node)(A0,A1)          $f8188C

* Node am Kopf anfügen

AddHead move.l  (A0),D0 ; Zeiger auf erste (alte) Node
        move.l  A1,(A0) ; neue Node in Liste eintragen
        movem.l D0/A0,(A1)      ; alte Node ist Vorgänger
                        ; der zweiten neuen durch A1
                        ; ln_Pred der neuen Node ist
                        ; Zeiger auf Liste
        move.l  D0,A0   ; Zeiger auf alte Node nach A0
        move.l  A1,ln_Pred(A0)  ; in alter Node ist der
Vörgänger
                        ; die neue Node
        rts

; vier Änderungen sind durchzuführen:
; 1) Neue Node in Liste einbinden
; 2) Vorgänger der Neuen Node ist die Liste
; 3) In neuer Node ist Nachfolger die alte Node
; 4) In alter Node ist der Vorgänger die neue Node


*** AddTail(Liste,Node)(A0,A1)          $f8189C
```

```
* Node am Ende einfügen

AddTail addq.l  #4,A0   ; Zeiger auf letzte Node Liste
        move.l  lh_Tail(A0),D0  ; letzter Node in D0
        move.l  A1,lh_Tail(A0)  ; neues Ende in Liste
setzen
        move.l  A0,ln_Succ(A1)  ; Vorgänger der neuen Node
                        ; ist die Liste
        move.l  D0,ln_Pred(A1)  ; alte Node Vorgänger der
Neuen
        move.l  D0,A0   ; alte Node nun in A0
        move.l  A1,(A0) ; neue Node ist Nachfolger
                        ; in der alten Node
        rts

; auch hier sind es vier Schritte
; 1) in der Liste neues Ende auf neue Node setzen
; 2) Vorgänger der neuen Node ist die Liste
; 2) die alte Node hat jetzt einen Nachfolger, die neue Node
; 2) in der neuen Node ist nun der Vorgänger die alte Node


*++ _RemPort(Port)(A1)                  $f818b2

* Port entfernen

_RemPort        addq.b  #1,TdNestCnt(A6)        ; Ints sperren
        move.l  mp_Succ(A1),A0  ; Vorgänger
        move.l  mp_Pred(A1),A1  ; Nachfolgr
        move.l  A0,mp_Succ(A1)  ; in nachfolger den
Vorgänger
        move.l  A1,mp_Pred(A0)  ; und umgekehrt
        bra     Permit  ; Ints wieder erlauben


*** Remove(Node)(a1)                    $f818c6
* Node löschen

Remove  move.l  ln_Succ(A1),A0  ; Nachfolger der Node in a0
        move.l  ln_Pred(A1),A1  ; Vorgänger der Node in a1
        move.l  A0,ln_Succ(A1)  ; Nachfolger der Node
                        ; in Vorgänger-Node
        move.l  A1,ln_Pred(A0) ; Vorgänger der Node
                        ; in Nachfolger-Node
        rts

; alte Node entfernen


*** RemHead(Liste)(A0)                  $f818d4

* erste Node aus Liste entfernen

RemHead move.l  lh_Head(A0),A1  ; erste Node in Liste
        move.l  ln_Succ(A1),D0  ; Nachfolger der ersten
Node
        beq     RemHead_NoSucc  ; gibt es einen Nachfolger?
        move.l  D0,ln_Head(A0)  ; Nachfolger in Liste
        exg     D0,A1   ; Nachfolger (D0) und erste
                        ; Node (A1) vertauschen
                        ; A1 ist dann also der Nachfolger
        move.l  A0,ln_Pred(A1)  ; Vorgänger der jetzt
ersten Node
                        ; ist die Liste
RemHead_NoSucc  rts


*** RemTail(Liste)(A0)                  $f818e4

* letzte Node entfernen

RemTail move.l  lh_TailPred(A0),A1 ; letzte Node holen
        move.l  ln_Pred(A1),D0  ; Vorgänger der Letzten
        beq     RemTail_NoPred  ; es gibt keinen Vorgänger
        move.l  D0,lh_TailPred(A0) ; zweitletzte Node ist
                        ; jetzt die Letzte
        exg     D0,A1   ; jetzt letzte Node in A1
        move.l  A0,ln_Succ(A1)  ; Nachfolger in der Node
                        ; ist die Liste
        addq.l  #4,ln_Succ(A1)  ; Node soll nicht auf lh_Tail
                        ; stehen, sonden auf ln_TailPred
RemTail_NoPred  rts
```

```
*++ Library wird als Node eingefügt             $f818FA

_SortInLibrary  addq.b  #1,TdNestCnt(A6)
        bsr     Enqueue
        bra     Permit


*** Enqueue(Liste,Node)(A0,A1)          $f81904

* Node nach Priorität einsortieren

Enqueue move.b  ln_Pri(A1),D1   ; Priorität der Node
        move.l  lh_Head(A0),D0  ; erste Node in der Liste

Enqueue_Such    move.l  D0,A0   ; in ein Adressreg.
        move.l  (A0),D0 ; Nachfolger der weiteren
        beq     Enqueue_Letzte  ; es geht nicht weiter, hinten
                        ; anfügen
        cmp.b   ln_Pri(A0),D1   ; Priorität der folgenden kleiner
        ble     Enqueue_Such    ; als die der Einzusortierenden

Enqueue_Letzte  move.l  ln_Pred(A0),D0  ; Aus Node mit höherer Priorität
                        ; den Nachfolger sichern, denn der
                        ; wird Nachfoger der Einzufügenden
        move.l  A1,ln_Pred(A0)  ; Node mit h.P., hat jetzt einen
                        ; neuen Nachfolger, unsere Node
        move.l  A0,ln_Succ(A1)  ; Vorgänger sichern
        move.l  D0,ln_Pred(A1)  ; Nachfolger der alten
                        ; ist Vorgänger der Neuen
        move.l  D0,A0   ; Vorgänger der jetzt neuen h.P.
        move.l  A1,ln_Succ(A0)  ; hat als Nachfolger unsere Node
        rts

; je höher die Priorität, desto näher am Kopf der Liste
; wenn 3 Nodes existieren mit den Prioritären 132,34,3
; dann wird die mit 44 zwischen 132 und 34 eingefügt, denn (siehe Listing)
; erst der Vergelich 44<=34 ist wahr


*** Node(D0) = FindName(Liste,Name)(A0,A1)              $f8192a

* die Node mit dem Namen (A1) finden

FindName        move.l  A2,-(SP)        ; Mal so sichern
        move.l  A0,A2   ; Liste zum Bearbeiten wie
        move.l  A1,D1   ; auch den Namen sichern
        move.l  ln_Head(A2),D0  ; erste Node in der Liste
        beq     FN_NoNode       ; es gibt keine Node
FN_WeiterSuche  move.l  D0,A2   ; Nodes duchsuchen
        move.l  ln_Succ(A2),D0  ; schon mal den Nachfolger
        beq     FN_NoNode       ; keinen Nachfolger
        tst.l   ln_Name(A2)     ; hat es einen Namen?
        beq     FN_WeiterSuche  ; kein Name
        move.l  ln_Name(A2),A0  ; Zeiger auf Namen
        move.l  D1,A1   ; den ges. Namen nehmen

FN_Strcmp       cmp.b   (A0)+,(A1)+     ; und byteweise vergleichen
        bne.s   FN_WeiterSuche  ; er stimmt nicht überein!
        tst.b   -1(A0)  ; Oh, doch, ist denn auch der
                        ; Name der Liste zu Ende?
        bne     FN_Strcmp       ; Nein, Schade
        move.l  A2,D0   ; Ja, A2 nun in s Returnreg.
FN_NoNode       move.l  D1,A1   ; damit auch A1 wieder so
                        ; wie früher war
        move.l  (SP)+,A2        ; und das Stackreg. holen
        rts


/* Schnipp */


************** Libraryfunktionen *******************************


*** OldOpenLibrary(LibName)(A0)         $f8196c

OldOpenLibrary  moveq   #0,D0   ; Versionsnummer egal
        jmp     OpenLibrary(A6)

        move.l  D2,-(SP)
        move.l  D0,D2
        lea     LibList(A6),A0
        addq.b  #1,TdNestCnt(A6)
f8197E  move.l  A1,-(SP)
        jsr     FindName(A6)
```

```
        move.l  (SP)+,A1
        tst.l   D0
        beq     f8199C
        move.l  D0,A0
        cmp     $14(A0),D2
        bgt     f8197E
        move.l  A6,-(SP)
        move.l  A0,A6
        jsr     LIB_OPEN(A6)    ; Library öffnen
        move.l  (SP)+,A6
f8199C  move.l  (SP)+,D2
        bra     Permit

        move.l  A1,D0
        beq     f819E4
        addq.b  #1,TdNestCnt(A6)
        move.l  A6,-(SP)
        move.l  A1,A6
        jsr     LIB_CLOSE(A6)   ; und die Lib schließen
        move.l  (SP)+,A6
        bra     Permit


*** OldFunct(D0) =SetFunction(Library,Offset,FuncEntry)(A1,A0,D0)

* neuen Funktionsvektor setzten         $f819b8

SetFunction     addq.b  #1,TdNestCnt(A6) ; damit mir keiner an
                        ; die Library geht!
        bset    #2,lib_Flags(A1) ; LIBF_CHANGED setzen, denn
                        ; Lib wird gleich verändert
        lea     (A1,A0),A0      ; die alte Sprungadresse
                        ; ermitteln
        move.l  2(A0),-(SP)     ; und diese auf den Stack sichern
        move    #$4EF9,(A0)     ; jmp-Code schreiben
        move.l  D0,2(A0)        ; neue Adresse einsetzen
        move.l  A1,-(SP)        ; Librarzeiger auf m Stack
        jsr     CacheClearU(A6); Cachespeicher löschen
        bsr     Permit  ; ??
        move.l  (SP)+,A1        ; wieder brav zurück
        jsr     SumLibrary(A6)  ; und neu durchrechnen
        move.l  (SP)+,D0        ; alte Funktionsadresse nach D0
f819E4  rts


*** AddLibrary(Library)(A1)                    $f819e6

* Library dem System hinzufügen

AddLibrary      lea     LibList(A6),A0  ; Alle Libraries
        bsr     _SortInLibrary  ; und unsere hinzunoden
        jmp     SumLibrary(A6)  ; und die Checksumme neu


*** SumLibrary(Library)(A1)                    $f819f2

* neue Checksumme der Library berechnen, bei Fehler Alert

SumLibrary      btst    #2,lib_Flags(A1) ; ist LIBF_CHANGED gestetzt
        beq     SummLibUnnötig    ; nein, dann Arbeit
sparen
        addq.b  #1,TdNestCnt(A6) ; keiner stört uns
        bclr    #1,lib_Flags(A1) ; LIBF_SUMMING, jetzt wird
                        ; gerechnet
        beq     SumLibNoSumKill ; nicht die Summe löschen
        clr     LibSum(A1)      ; Summe löschen, also ganz neu
SumLibNoSumKill move.l  A1,A0
        move    lib_NegSize(A1),D0 ; Anzahl Vektoreeinträge
        lsr     D0      ; durch zwei, da Wordaddition
        moveq   #0,D1   ; unser Zähler
        bra     SubLibEinstieg  ; und los

SumLibSumLoop   add     -(A0),D1        ; Eintrag addieren
SubLibEinstieg  dbra    D0,SumLibSumLoop ; bis zum bittren Ende

        move    lib_Sum(A1),D0  ; alte Summe nach D0
        beq     SumLibLenIn     ; gab s nicht, dann nur neue
        cmp     D0,D1   ; ist die alte Länge gleich der
                        ; neuen?
        beq     SumLibSumOK     ; ja, dann alles OK.
        movem.l D7/A5/A6,-(SP)
        move.l  #$1000003,D7
        move.l  4,A6    ; können wir uns sparen, oder?
        jsr     Alert(A6)       ; Tja, Fehler
        movem.l (SP)+,D7/A5/A6
```

```
SumLibLenIn     move    D1,$1C(A1)      ; neue Länge eintragen
SumLibSumOK     bsr     Permit
SummLibUnnötig  rts


*** Library(D0) = MakeLibrary(FuncInit,StructInit,      ; $f81a46
                  LibInit,DataSize,CodeSize)(A0-A2,D0,D1)

* Library erstellen Übergabeparameter:
* - FuncInit Zeiger auf Tabelle mit Sprungadressen
* - *InitStruct
* - evtl. eine Routine
* - Datenbereich der Lib
* - Segement Liste

MakeLibrary     movem.l D2-D7/A2/A3,-(SP)
        move.l  D0,D2   ; Größe der Library in D2
        move.l  A0,D4   ; Funk-Adresstabelle
        move.l  A1,D5   ; InitStruct
        move.l  A2,D6   ; eigenes Prg zur Init in d6
        move.l  D1,D7   ; Segment Liste
        move.l  A0,D3   ; Funk-Adresstabelle auch in d3
        beq     MakeLibNoEntry
        move.l  A0,A3   ; Tabellenzeiger A3
        moveq   #-1,D3  ; Größe der VekTab
        move.l  D3,D0   ; mit d0 = -1 starten
        cmp     (A3),D0 ; Relativ-Adresse holen
        bne     f81A6C
        addq.l  #2,A3
f81A64  cmp     (A3)+,D0
        dbeq    D3,f81A64

        bra     f81A72

f81A6C  cmp.l   (A3)+,D0
        dbeq    D3,f81A6C

f81A72  not     D3
        mulu    #6,D3
        addq.l  #3,D3
        and     #$FFFC,D3

MakeLibNoEntry  move.l  D2,D0   ; Größe der Library = D0
        add.l   D3,D0
        move.l  #$10001,D1      ; Speicherbits
        jsr     AllocMem(A6)    ; und Speicher holen
        tst.l   D0
        beq.s   MakeLibNoMem
        move.l  D0,A3   ; Speicher in A3 übertragen
        add.l   D3,A3
        move    D3,lib_NegSize(A3) ; Größe der Vektortabelle
        move    D2,lib_PosSize(A3) ; Größe der Daten
        move.l  A3,A0
        sub.l   A2,A2   ; A2 = 0
        move.l  D4,A1   ; Zeicher auf Adresstabelle
        cmp     #-1,(A1)
        bne     f81AAE
        addq.l  #2,A1
        move.l  D4,A2
f81AAE  bsr     MakeFunction
        tst.l   D5
        beq     MakeLiNoStruct
        move.l  A3,A2   ; der Speicher
        move.l  D5,A1   ; InitTable
        moveq   #0,D0   ; keine Größe
        jsr     InitStruct(A6)
MakeLiNoStruct  move.l  A3,D0
        tst.l   D6      ; gibt es eig. Prg
        beq     ML_NoOwnIntiPrg
        move.l  D6,A1   ; Zeiger des Prgs
        move.l  D7,A0
        jsr     (A1)    ; eig. Prg ausführen
ML_NoOwnIntiPrg
MakeLibNoMem    movem.l (SP)+,D2-D7/A2/A3
        rts


***TableSize(D0=MakeFunction(Target,FuncArray,FuncDispBase)(A0,A1,A2)

* Funktionstabelle aufbauen                      $f81ad0

MakeFunction    move.l  A3,-(SP)        ; A3 sichern
        moveq   #0,D0   ; TableSize kommt hier rein
        move.l  A2,D1   ; Basisadresse
        beq     MFNoBasisAdr    ; keine Basisadresse
```

```
MakeFuncLoop1   move    (A1)+,D1        ; Funktionsadresse
        cmp     #-1,D1  ; kein Eintrag mehr?
        beq     FakeFuncEnd     ; nö, dann sind wir fertig
        lea     (A2,D1),A3      ;
        move.l  A3,-(A0)        ; Adresse nach Taget schreiben,
                        ; so die Tabelle von unten
                        ; aufbauen
        move    #$4EF9,-(A0)    ; jmp-Code schreiben
        addq.l  #6,D0   ; und ein JMP xxxxx weiter
        bra     MakeFuncLoop1

MFNoBasisAdr
MakeFuncLoop1   move.l  (A1)+,D1        ; Funktionsadresse
        cmp.l   #-1,D1  ; kein Eintrag mehr?
        beq     FakeFuncEnd     ; dann schon fertig
        move.l  D1,-(A0)        ; sonst Funk-Adresse und
        move    #$4EF9,-(A0)    ; jmp-Code schreiben
        addq.l  #6,D0   ; und ein JMP xxxxx weiter
        bra     MakeFuncLoop2

FakeFuncEnd     move.l  D0,A3   ; Länge noch mal sichern
        jsr     CacheClearU(A6); und Cache weg
        move.l  A3,D0   ; und TableSize wieder in D0
        move.l  (SP)+,A3        ; A3 wieder zurückgeben
        rts

        dc      0

**************  Speicherfunktionen ************************

/* Schnipp:
*** Deallocate                          $f81b10
*/


*** FreeMem(MemBlock,Size)(A1,D0)               $f81bbe

* belegter Speicher dem System zurückgeben

FreeMem move.l  A1,D1
        beq     FreeMemEnd      ; kein Speicherblock angeg.
        lea     MemList(a6),A0  ; die ganze Speicherliste
        addq.b  #1,TdNestCnt(A6) ; keiner darf an den Speicher
                        ; während der Arbeit
FreeMemLoop     move.l  ml_Succ(A0),A0  ; Schon mal Nachfolger
        tst.l   (A0)    ; gibt es den denn?
        beq     FreeMemErr      ; Schluck, dann würde es
                        ; unseren Block ja nicht geben
        cmp.l   $14(A0),A1      ; mit MemEntrie vergleichen
        bcs     FreeMemLoop
        cmp.l   $18(A0),A1      ; das auch
        bcc     FreeMemLoop
                        ; A0 = FreeList,A1 = MemoryBlock
        bsr     Deallocate      ; in D0 wird Größe
        bra     Permit

FreeMemErr      bsr     Permit
        movem.l D7/A5/A6,-(SP)
        move.l  #$100000F,D7    ; Benutzer versucht nicht
        move.l  4,A6    ; belegten Speicher freizugeben
        jsr     Alert(A6)       ; Guru darstellen
        movem.l (SP)+,D7/A5/A6
FreeMemEnd      rts

        dc      0


*** MemBlock(D0) = Allocate(FreeList,ByteSize)(A0,D0)   f81c02

* FreeList nach Speicher der Größe D0 durchsuchen

Allocate        cmp.l   mh_Free(a0),D0  ; maximale Größe
        bhi     Alloc_Error     ; Fehler, zu hoch
        tst.l   D0      ; ist überhaupt eine Länge
        beq     Alloc_End       ; gefragt?
f81C0C  move.l  A2,-(SP)        ; Sichern
        addq.l  #7,D0   ; +7
        and     #$FFF8,D0       ; und ausmaskieren, damit
                        ; Grenze durch Acht teilbar
        lea     mh_First(A0),A2 ; erste MemChuck-Liste
AllocGoOneSuch  move.l  mc_Next(A2),A1  ; und schon nächstes
        move.l  A1,D1   ; gibt es nächstes?
        beq     f81C5E
        cmp.l   mc_Bytes(A1),D0 ; sind freie Bytes da?
        bls     f81C32
```

```
        move.l  mc_Next(A1),A2  ; s.o.
        move.l  A2,D1
        beq     f81C5E
        cmp.l   mc_Bytes(A2),D0
        bhi     AllocGoOneSuch ; unser Wunsch nicht erfüllt

        exg     A1,A2
f81C32  beq     f81C52
        move.l  A3,-(SP)
        lea     (A1,D0.l),A3
        move.l  A3,(A2)
        move.l  (A1),(A3)+
        move.l  4(A1),D1
        sub.l   D0,D1
        move.l  D1,(A3)
        sub.l   D0,mh_Free(A0)  ; jetzt weniger Memory
        move.l  (SP)+,A3
        move.l  A1,D0
        move.l  (SP)+,A2
        rts

f81C52  move.l  (A1),(A2)
        sub.l   D0,mh_Free(A0)  ; jetzt weniger Memory
        move.l  (SP)+,A2
        move.l  A1,D0
        rts

f81C5E  move.l  (SP)+,A2
Alloc_Error     moveq   #0,D0
Alloc_End       rts

/* Schnipp */

/* Schnipp:
 *** TypeOfMem                          $f81d16
 *** AllocAbs                           $f81d3a
 *** AvailMem                           $f81dd2
 *** AllocEntry                $f81e54
 *** FreeEntry                          $f81ee8
 *** AddMemList                $f81f26
*/

************* Portfunktionen ***********************************

*** AddPort                             $f82090

AddPort lea     $14(A1),A0
        move.l  A0,8(A0)
        addq.l  #4,A0
        clr.l   (A0)
        move.l  A0,-(A0)
        lea     PortList(A6),A0
        bra     _SortInLibrary


*** RemPort                             $f820a6

RemPort bra     _RemPort


*** ReplyMessage                       $f820aa

ReplyMessage    moveq   #7,D0   ; ??
        move.l  mp_Flags(A1),D1 ; Message-Flags
        move.l  D1,A0   ; Flags nach A0
        bne     RepMesFlags     ; es gibt Flags, die
                        ; zu bearbeiten sind
        move.b  #NT_FREEMSG,mp_Typ(A1) ; Typ der Node
        rts


*** PutMsg                              $f820bc

PutMsg  moveq   #5,D0
        move.l  A0,D1   ; Port in D1
RepMesFlags     lea     mp_Flags(A0),A0 ; Flags auch nach A0
        move    #$4000,INTENA
        addq.b  #1,IDNestCnt(A6)
        move.b  D0,8(A1)
        addq.l  #4,A0
        move.l  4(A0),D0
        move.l  A1,4(A0)
        move.l  A0,(A1)
        move.l  D0,4(A1)
        move.l  D0,A0
```

```
        move.l  A1,(A0)
        move.l  D1,A1
        move.l  $10(A1),D1
        beq     f82106
        move.b  14(A1),D0
        and     #3,D0
        beq     f82122
        cmp.b   #1,D0
        bne     f82116
        move.l  D1,A1
        jsr     Cause(A6)       ; Interrupt auslösen
f82106  subq.b  #1,IDNestCnt(A6)
        bge     f82114
        move    #$C000,INTENA
f82114  rts

f82116  cmp.b   #2,D0
        beq     f82106
        move.l  D1,A0
        jsr     (A0)
        bra     f82106

f82122  move.b  15(A1),D0
        addq.b  #1,TdNestCnt(A6)
        subq.b  #1,IDNestCnt(A6)
        bge     f82138
        move    #$C000,INTENA
f82138  move.l  D1,A1
        moveq   #0,D1
        bset    D0,D1
        move.l  D1,D0
        jsr     Exit(A6)
        bra     Permit


*** GetMsg                              $f82148

GetMsg  lea     $14(A0),A0
        move    #$4000,INTENA
        addq.b  #1,IDNestCnt(A6)
        move.l  (A0),A1
        move.l  (A1),D0
        beq     f82166
        move.l  D0,(A0)
        exg     D0,A1
        move.l  A0,4(A1)
f82166  subq.b  #1,IDNestCnt(A6)
        bge     GetMsg_NoInt
        move    #$C000,INTENA
GetMsg_NoInt    rts


*** WaitPort                            $f82176

WaitPort        move.l  $14(A0),A1
        tst.l   (A1)
        bne     f8219A
        move.b  15(A0),D1
        lea     $14(A0),A0
        moveq   #0,D0
        bset    D1,D0
        move.l  A2,-(SP)
        move.l  A0,A2
f8218E  jsr     Wait(A6)
        move.l  (A2),A1
        tst.l   (A1)
        beq     f8218E
        move.l  (SP)+,A2
f8219A  move.l  A1,D0
        rts

*** FindPort                            $f8219e

FindPort        lea     PortList(A6),A0
        jmp     FindName(A6)


************* Resource-Funktionen *******************************

        dc      0

        move.l  a7,d0
        rts

        rts
```

```
        move.l  #$10000,D1
        jmp     AllocVec(A6)

        jmp     FreeVec(A6)


*** AddResource                 $f821bc

AddResource     lea     ResourceList(A6),A0
        bra     _SortInLibrary


*** RemResource                 $f821c4

RemResource     bra     _RemPort


** OpenResource                 $f821c8

OpenResource    lea     ResourceList(A6),A0
        addq.b  #1,TdNestCnt(A6)
        jsr     FindName(A6)
        bra     Permit



************** Task-Funktionen **********************************

*** AddTask                             $f811d8

AddTask move.b  #1,15(A1)
        moveq   #8,D1
        and.b   D1,14(A1)
        beq     f82222
        moveq   #$56,D0
        move.l  #$10001,D1
        move.l  A1,-(SP)
        jsr     AllocVec(A6)
        move.l  (SP)+,A1
        move.l  D0,$22(A1)
        beq     f822C2
        move.l  D0,A0
        move.l  ThisTask(A6),$14(A0)
        addq.b  #1,TdNestCnt(A6)
        addq.l  #1,$240(A6)
        BVC     f82218
        move.l  #$400,$240(A6)
f82218  move.l  $240(A6),$18(A0)
        jsr     Permit(A6)
f82222  moveq   #0,D1
        move    #-1,$10(A1)
        move.l  TaskSigAlloc(A6),$12(A1)
        move.l  D1,$16(A1)
        move.l  D1,$1A(A1)
        move.l  D1,$1E(A1)
        bsr     f825A6
        move    TaskTrapAlloc(A6),D1
        bsr     f825BE
        tst.l   $32(A1)
        bne     f82254
        move.l  TaskTrapCode(A6),$32(A1)
f82254  tst.l   $2A(A1)
        bne     f82260
        move.l  TaskExeptCode(A6),$2A(A1)
f82260  move.l  $36(A1),A0
        move.l  A3,-(A0)
        bne     f8226C
        move.l  TaskExitCode(A6),(A0)
f8226C  moveq   #14,D1
f8226E  clr.l   -(A0)
        dbra    D1,f8226E

        clr     -(A0)
        move.l  A2,-(A0)
        btst    #6,(AttnFlags+1)(A6)
        beq     f82284
        moveq   #0,D0
        move.l  D0,-(A0)
f82284  move.l  A0,$36(A1)
        lea     TaskReady(A6),A0
        move    #$4000,INTENA
        addq.b  #1,IDNestCnt(A6)
        move.b  #3,15(A1)
        bsr     Enqueue
```

```
        move.l  TaskReady(A6),D0
        subq.b  #1,IDNestCnt(A6)
        bge     f822B4
        move    #$C000,INTENA
f822B4  move.l  A1,-(SP)
        cmp.l   A1,D0
        bne     f822BE
        jsr     Reschedule(A6)
f822BE  move.l  (SP)+,D0
        rts

f822C2  moveq   #0,D0
        rts


*** RemTask                              $f822cc

RemTask move.l  4,A6
        sub.l   A1,A1
        movem.l D2-D4,-(SP)
        move.l  A1,D3
        bne     f822DA
        move.l  ThisTask(A6),D3
        bra     f822FE

f822DA  cmp.l   ThisTask(A6),A1
        beq     f822FE
        move    #$4000,INTENA
        addq.b  #1,IDNestCnt(A6)
        bsr     Remove
        subq.b  #1,IDNestCnt(A6)
        bge     f822FE
        move    #$C000,INTENA
f822FE  move.l  D3,A1
        move.b  #6,15(A1)
        btst    #3,14(A1)
        beq     f8231A
        move.l  A1,-(SP)
        move.l  $22(A1),A1
        jsr     FreeVec(A6)     ; ??
        move.l  (SP)+,A1
f8231A  cmp.l   ThisTask(A6),A1
        bne     f82324
        addq.b  #1,TdNestCnt(A6)
f82324  lea     $4A(A1),A0
        move.l  (A0),D2
        beq     f82346
        cmp.l   8(A0),A0
        beq     f82346
        clr.l   (A0)
        lea     4(A0),A0
        move.l  A0,D4
f8233A  move.l  D2,A0
        move.l  (A0),D2
        jsr     FreeEntry(A6)
        cmp.l   D2,D4
        bne     f8233A
f82346  cmp.l   ThisTask(A6),D3
        bne     f82364
        lea     lbW001354(PC),A5
        jmp     Supervisor(A6)

f82354  btst    #6,(AttnFlags+1)(a6)
        beq.s   f8235E
        addq.l  #2,SP
f8235E  addq.l  #6,SP
        jmp     Dispatch(a6)

f82364  moveq   #0,d0
        movem.l (SP)+,d2-d4
        rts


*** FindTask(Task)(A1)                   $f8236c

* einen Task suchen

FindTask        move.l  A1,D0   ; Flags setzten
        bne     FT_NoAkfTask    ; wenn nicht aktueller Task
        move.l  ThisTask(A6),D0 ; sonst aus Exec-Base
                        ; den aktuellen Task holen
        rts

FT_NoAkfTask    lea     TaskReady(A6),A0 ; aus der Task-Ready Liste
        move.l  A1,-(SP)        ; den Task sichern
```

```
        move    #$4000,INTENA   ; Ints-sperren, damit z. B.
                        ; der READY-Task nicht nach
                        ; RUNNING geht, oder die Liste
                        ; durcheinander gerät
        addq.b  #1,IDNestCnt(A6) ; gehört zum sperren dazu
        jsr     FindName(A6)    ; jetzt Namen durch Node-Find
        tst.l   D0      ; suchen.
        bne     FindTaskFound   ; gefunden
        lea     TaskWait(A6),A0 ; auch die wartenden Task
                        ; durchsuchen
        move.l  (SP),A1 ; Namen auf Task wiederholen
        jsr     FindName(A6)    ; suchen
        tst.l   D0
        bne     FindTaskFound
        move.l  ThisTask(A6),A0 ; dann nur noch der akt. Task
        move.l  tc_Name(A0),A0  ; aus Task Node den Namen
        move.l  (SP),A1 ; wieder den Namen
FindTskStrgCmp  cmpm.b  (A0)+,(A1)+     ; Task-Name vergleichen
        bne     FindTskNoFound
        tst.b   -1(A0)
        bne     FindTskStrgCmp
        move.l  ThisTask(A6),D0 ; dann Zeiger auf AktTask
                        ; als Return übergeben
FindTaskFound   subq.b  #1,IDNestCnt(A6)
        bge     FTFNoInts
        move    #$C000,INTENA
FTFNoInts       addq.l  #4,SP   ; Namen vom Stack entfernen
        rts

; 1. Ready-Liste durchsuchen
; 2. Waiting-Tasks durchsuchen
; 3. dann evtl. der aktuelle Task, FindTask nicht benutzt


*** OldPri(D0) = SetTaskPri(Task,Priority)(A1,D0)       $f823c8

* Neue Priorität setzen

SetTaskPri      move    #$4000,INTENA
        addq.b  #1,IDNestCnt(A6)
        move.b  tc_Pri(A1),-(SP) ; Priorität des Tasks aus
Node
        move.b  D0,tc_Pri(A1)   ;  neue Priorität eintragen
        cmp.l   ThisTask(A6),A1 ; handelt es sich um den
                        ; aktuellen Task?
        beq     STP_OwnTask     ; ja, dann muß dieser nicht
                        ; neu einsortiert werden
        cmp.b   #TS_READY,tc_State(A1) ; ein wartender?
        bne     STP_NoWaitTask
        move.l  A1,D0   ; Task nach D0
        bsr     Remove  ; Task aus Liste entfernen
        lea     TaskReady(A6),A0 ; Taskliste
        move.l  D0,A1   ; Priorität ja schon eingetragen
        bsr     Enqueue ; neu Einsortieren
        cmp.l   TaskReady(A6),A1
        bne     STP_NoWaitTask
STP_OwnTask     jsr     Reschedule(A6)  ; neue Timer-Tabelle berechnen
STP_NoWaitTask  subq.b  #1,IDNestCnt(A6)
        bge     STP_NoInts
        move    #$C000,INTENA
STP_NoInts      moveq   #0,D0
        move.b  (SP)+,D0        ; alte Prioritärt
        rts


*** OldExpt(D0) = SetExept(NewOldE,ExeptSet)            $f82418

* bestimmte Signale als Execption-Auslöser setzen

SetExept        move.l  ThisTask(A6),A1 ; immer den akt. Task
        lea     tc_SigExept(A1),A0 ; Zeiger auf Datenword
        bra     _SetSignal


*** OldSig(D0) = SetSignal(OldSign,SigntSet)            $fd82422

* Signale setzen

SetSignal       move.l  ThisTask(A6),A1
        lea     tc_SigRecvd(A1),A0 ; Zeiger auf Signale, die
                        ; empfangen werden
_SetSignal      and.l   D1,D0   ; durch die Maske aussortieren
        not.l   D1      ; das gehört dazu
        move    #$4000,INTENA   ; Ints sperren
        addq.b  #1,IDNestCnt(A6) ; das gehört auch dazu
```

```
        move.l  (A0),-(SP)      ; altes Signal auf m Stack
        and.l   (A0),D1 ; altes Signal mit neuem
        or.l    D0,D1   ; kombinieren
        move.l  D1,(A0)
        move.l  tc_SigRecvd(A1),D0 ; jetzt Signale senden
        bra     _Signal ; ThisTask steht in A1

; 1. Signale setzen
; 2. da sich eine Signaländerung vollzog, dies anzeigen


*** Signal(Task,SignalSet)(A1,D0)              $f82448

* einem Task eine Nachricht signalisieren

Signal  lea     tc_SigRecvd(A1),A0
        move    #$4000,INTENA   ; sperren
        addq.b  #1,IDNestCnt(A6)
        move.l  (A0),-(SP)      ; Signale auf m Stack
        or.l    D0,(A0) ; die geg. Signale verknüpfen
_Signal move.l  tc_SigExcept(A1),D1 ; Exception auslösende
                        ; Signale nach D1
        and.l   D0,D1
        bne     SignalExpect
        cmp.b   #TS_WAIT,tc_State(A1) ; Wartes der Task?
        bne     SigNoSigToWait
        and.l   tc_SigWait(A1),D0 ; Signale, auf die gewartet
        beq     SigNoSigToWait  ; wird
SignalTaskWait  move.l  A1,D0   ; Task sichern
        move.l  tc_Succ(A1),A0  ; Task austragen, siehe
        move.l  tc_Pred(A1),A1  ; Remove
        move.l  A0,tc_Succ(A1)
        move.l  A1,tc_Prd(A0)
        move.l  D0,A1   ; Task wieder in A1
        move.b  #TS_READY,tc_State(A1)
        lea     TaskReady(A6),A0 ; Task-Ready-Liste
        bsr     Enqueue ; neu Einsortieren
        cmp.l   TaskReady(A6),A1
        bne     SigNoSigToWait
SignalEnd       subq.b  #1,IDNestCnt(A6)
        bge     SignalNoInt
        move    #$C000,INTENA
SignalNoInt     move.l  (SP)+,D0
        jmp     Reschedule(A6)

SignalExpect    bset    #TC_EXCEPT,tc_Flags(A1)
        cmp.b   #TS_WAIT,tc_State(A1)
        beq     SignalTaskWait
        bra     SignalEnd       ; Ende und aus

SigNoSigToWait  subq.b  #1,IDNestCnt(A6)
        bge     f824C8
        move    #$C000,INTENA
f824C8  move.l  (SP)+,D0
        rts

* Wait                           $f824cc
Wait    move.l  ThisTask(A6),A1
        move.l  D0,$16(A1)
        move    #$4000,INTENA
        and.l   $1A(A1),D0
        bne     f8253C
        addq.b  #1,TdNestCnt(A6)
f824E6  move.b  #4,15(A1)
        lea     TaskWait(A6),A0
        addq.l  #4,A0
        move.l  4(A0),D0
        move.l  A1,4(A0)
        move.l  A0,(A1)
        move.l  D0,4(A1)
        move.l  D0,A0
        move.l  A1,(A0)
        move.b  IDNestCnt(A6),D1
        move.b  #$FF,IDNestCnt(A6)
        move    #$C000,INTENA
        move.l  A5,A0
        lea     Switch(A6),A5
        jsr     Supervisor(A6)  ; dies im Super-Mode
        move.l  A0,A5
        move    #$4000,INTENA
        move.b  D1,IDNestCnt(A6)
        move.l  $16(A1),D0
        and.l   $1A(A1),D0
        beq     f824E6
        subq.b  #1,TdNestCnt(A6)
```

```
f8253C  Eor.l   D0,$1A(A1)
        tst.b   IDNestCnt(A6)
        bge     f8254E
        move    #$C000,INTENA
f8254E  rts


*** Reschedule                          $f82550

Reschedule      bset    #7,SysFlags(A6)
        sne     D0
        tst.b   TDNestCnt(A6)   ; Interrupts erlaubt?
        bge     f82570
        tst.b   IDNestCnt(A6)   ; Taskswitchin erlaubt
        blt     _Reschedule
        tst.b   D0
        bne     ReschNiInt
        move    #$8004,INTREQ
ReschNiInt      rts


*** Forbit                              $f82572

* Task-Switching abschalten

Forbit  addq.b  #1,TdNestCnt(A6) ; +1
        rts


*** Permit                              $f82578

* Switching wieder zulassen

Permit  subq.b  #1,TdNestCnt(A6)
        bge     PermitEnd       ; >0, d. h. Ende
        tst.b   IDNestCnt(A6)
        bge     PermitEnd       ; >0, d. h. Ende
        tst     SysFlags(A6)
        bmi     _Reschedule
PermitEnd       rts

; Task-Switching ist möglich, wenn TDNestCnt < 0


*++ _Reschedule                 $f8258c

_Reschedule     move.l  a5,-(SP)
        lea     _ReschedulePrg(pc),a5 ; Schedule im
        jsr     Supervisor(a6)  ; Super-Mode
        move.l  (SP)+,a5
        rts

_ReschedulePrg  btst    #5,(SP)
        bne.s   _RePrgNoSched
        jmp     Schedule(a6)

_RePrgNoSched   rte


; von AddTask benutzt

f825A6  btst    #3,tc_Flags(A1)
        beq     f825B8
        move.l  tc_TrapAlloc(A1),A0
        move    D1,$2a(A0)
        rts
f825B8  move    D1,tc_TrapAble(A1)
        rts

f825BE  btst    #3,tc_Flags(A1)
        beq     f825D0
        move.l  tc_TrapAlloc(A1),A0
        move    D1,$28(A0)      ; TrapNr setzen?
        rts
f825D0  move    D1,tc_TrapAlloc(A1)
        rts

f825D6  btst    #3,tc_Flags(A1)
        beq     f825E8
        move.l  tc_TrapAlloc(A1),A0
        move    $28(A0),D1      ; TrapNr holen?
        rts
f825E8  move    tc_TrapAlloc(A1),D1
        rts
```

```
*** TrapNr(D0) = AllocTrap(TrapNr)(D0)          $f825ee

* belegt einen System-Trap

AllocTrap       move.l  ThisTask(A6),A1 ; jetziger Task
        bsr     f825D6  ; belegte Traps holen (?)
        cmp.b   #-1,D0  ; nächst freier Trap
        beq     ATNextFree      ; ja, nächst freien Trap belegen
        bset    D0,D1   ; Trapnr. setzen
        beq     f8260C  ; alles OK
        bra     AT_AllesVoll    ; schon besetzt gewesen

ATNextFree      moveq   #16-1,D0        ; kleine Schleife
AT_SuchFreeBit  bset    D0,D1   ; setzen und dabei testen
        beq     EndOfAllOf      ; freies Bit suchen
                        ; wenn eins gefunden, dann
                        ; Ende der Sucherei
        dbra    D0,AT_SuchFreeBit

AT_AllesVoll    moveq   #-1,D0  ; nix mehr frei
EndOfAllOf      bra     f825BE  ; Trapbelegung sichern (?)


*** FreeTrap(TrapNr)(D0)                        $f8260e

* Prozessor-Trap wieder freigeben

FreeTrap        move.l  ThisTask(A6),A1
        bsr     f825D6  ; belegte Traps holen (?)
        bclr    D0,D1   ; ausmaskieren
        bra     f825BE  ; Trapbelegung sichern (?)

** SigNr(D0) = AllocSignal(SigNr)(D0)           $f82618

AllocSignal     move.l  ThisTask(A6),A1
        move.l  tc_SigAlloc(A1),D1 ; Long?
        cmp.b   #-1,D0
        beq     f8262C
        bset    D0,D1
        beq     f8263A
        bra     f82636


/* Schnipp
*** FreeSignal                          $f82650
*/


************ Sonstige Routinen ***********************************

        dc      0

*++ Zahl(D0) = asc2bin(Strg)(a4)                $df2710

* interne Umwandelroutine von String nach Zahl

asc2bin moveq   #0,D0
        moveq   #0,D2
a2b_Loop        move.b  (A4)+,D2
        cmp.b   #"0",D2
        bhs     a2b_End
        cmp.b   #"9",D2
        bhi     a2b_End
        add.l   D0,D0
        move.l  D0,D1
        add.l   D0,D0
        add.l   D0,D0
        add.l   D1,D0
        sub.b   #"0",D2
        add.l   D2,D0
        bra     a2b_Loop

a2b_End subq.l  #1,A4
        rts


*++ bin2asc(Strg,Zahl)(A5,D4)                   $f82738

* interne Umwandelroutine von Zahl nach String

bin2asc tst.l   D4      ; was für eine Zahl
        bpl     b2a_NoNegZahl   ; ist sie negativ?
        move.b  #"-",(A5)+      ; ja, dann "-"
        neg.l   D4      ; und Zahl negerieren
```

```
b2a_NoNegZahl   moveq   #"0",D0

        lea     DecTab(PC),A0   ; Dec-Tab
b2a_Loop        move.l  (A0)+,D1        ; 1xx-Zahl
        beq     b2_End
        moveq   #"0"+1,D2
b2a_SuchLoop    addq.l  #1,D2   ; ..Hunderter, Zehner, Einer +1
        sub.l   D1,D4   ; 100xx Block von unserer Zahl -
        bcc     b2a_SuchLoop
        add.l   D1,D4
        cmp.l   D0,D2
        beq     b2a_Loop
        moveq   #0,D0
        move.b  D2,(A5)+
        bra     b2a_Loop

b2_End  moveq   #"0",D0 ; Grundzahl
        add.b   D0,D4   ; Stelle hinzu
        move.b  D4,(A5)+        ; und schreiben
        rts

DecTab  dc.l    1000000000
        dc.l    100000000
        dc.l    10000000
        dc.l    1000000
        dc.l    100000
        dc.l    10000
        dc.l    1000
        dc.l    100
        dc.l    10
        dc.l    0


*++ bin2hex(Strg,Zahl)(A5,D4)                   $f82790

* interne Umwandelroutine von Zahl nach Hex-String

bin2hex tst.l   D4      ; ist Zahl da
        beq     b2_End  ; nö, dann Ende
        clr     D1
        btst    #2,D3
        bne     f827A2
        moveq   #3,D2
        swap    D4
        bra     f827A4

f827A2  moveq   #7,D2
f827A4  rol.l   #4,D4
        move.b  D4,D0
        and.b   #15,D0
        bne     f827B2
        tst     D1
        beq     f827C6
f827B2  moveq   #-1,D1
        cmp.b   #9,D0
        BHI     f827C0
        add.b   #"0",D0
        bra     f827C4

f827C0  add.b   #"7",D0
f827C4  move.b  D0,(A5)+
f827C6  dbra    D2,f827A4

        rts


***RawDoFmt(Formatstring,DataStream,PurChProc,PutChData)(A0,A1,A2,A3)          $f827cc

RawDoFmt        movem.l D2-D6/A2-A5,-(SP)
        link    A6,#-$10
        move.l  A1,-(SP)        ; Daten auf Stack sichern
        move.l  A0,A4   ; Formatstring nach A4
        bra     DoTheRawDoFmt

EndRawDoFmt     jsr     (A2)
        move.l  (SP),D0
        unlk    A6
        movem.l (SP)+,D2-D6/A2-A5
        rts

DoTheRawDoFmt   ; in a4 Zeiger auf Format-String
        ; d3 gibt an, was für Attribute gesetzt

        jsr     (A2)
RawDoFmt_Loop   move.b  (A4)+,D0        ; Zeichen nach d0
```

```
        beq     EndRawDoFmt
        cmp.b   #"%",D0
        bne     DoTheRawDoFmt   ; dann weiter
        lea     -$10(A6),A5
        clr     D3      ; alle Attribute löschen
        cmp.b   #"-",(A4)       ; kommt ein Minus?
        bne.s   RDF_KeinMinus
        bset    #0,D3   ; erstes Bit = evtl. Bündig
        addq.l  #1,A4   ; minus weg
RDF_KeinMinus   cmp.b   #"0",(A4)       ; führenden Nullen?
        bne     NoNullVor
        bset    #1,D3   ; zweites Bit = Nullen
NoNullVor       bsr     asc2bin ; f8280E
        move    D0,D6   ; Zeichen nach d6
        moveq   #0,D5
        cmp.b   #".",(A4)
        bne     RDF_KeinPunkt
        addq.l  #1,A4   ; "." weg
        bsr     asc2bin
        move    D0,D5
RDF_KeinPunkt   cmp.b   #"l",(A4)
        bne     RDF_KeinLong
        bset    #2,D3   ; zweites Bit = Long
        addq.l  #1,A4   ; "l" weg
RDF_KeinLong    move.b  (A4)+,D0
        cmp.b   #"d",D0
        bne     RDF_NoDezZahl
        bsr     f8284E
        bsr     bin2asc
        bra     f828B4

RDF_NoDezZahl   cmp.b   #"x",D0 f82840
        bne     RDF_NoHexZahl
        bsr     f8284E
        bsr     bin2hex
        bra     f828B4

f8284E  btst    #2,D3   f8284E
        bne     f82862
        move.l  4(SP),A1
        move    (A1)+,D4
        move.l  A1,4(SP)
        ext.l   D4
        rts

f82862  move.l  4(SP),A1
        move.l  (A1)+,D4
        move.l  A1,4(SP)
        tst.l   D4
        rts

RDF_NoHexZahl   cmp.b   #"s",D0 f82870
        bne     f8287E
        bsr     f82862
        beq     f828F2
        move.l  D4,A5
        bra     f828BA

f8287E  cmp.b   #"b",D0
        bne     f8289A
        bsr     f82862
        beq     f828F2
        lsl.l   #2,D4
        move.l  D4,A5
        moveq   #0,D2
        move.b  (A5)+,D2
        tst.b   -1(A5,D2)
        bne     f828C6
        subq    #1,D2
        bra     f828C6

f8289A  cmp.b   #"u",D0
        bne     f828A8
        bsr     f8284E
        bsr     f82742
        bra     f828B4

f828A8  cmp.b   #"c",D0
        bne     f827E6
        bsr     f8284E
        move.b  D4,(A5)+        ; Zeichen übertragen
f828B4  clr.b   (A5)
        lea     -$10(A6),A5
f828BA  move.l  A5,A0
        moveq   #-1,D2
```

```
f828BE  tst.b   (A0)+
        dbeq    D2,f828BE

        not.l   D2
f828C6  tst     D5
        beq     f828CE
        cmp     D5,D2
        BHI     f828D0
f828CE  move    D2,D5
f828D0  sub     D5,D6
        bpl     f828D6
        clr     D6
f828D6  btst    #0,D3
        bne     f828E4
        bsr     f828F6
        bra     f828E4

f828E0  move.b  (A5)+,D0
        jsr     (A2)
f828E4  dbra    D5,f828E0

        btst    #0,D3
        beq     RawDoFmt_Loop
        bsr     f828F6
f828F2  bra     RawDoFmt_Loop

f828F6  move.b  #" ",D2
        btst    #1,D3
        beq     f8290A
        move.b  #"0",D2
        bra     f8290A

f82906  move.b  D2,D0
        jsr     (A2)
f8290A  dbra    D6,f82906

        rts


*** RawIOInit                          $f82910

RawIOInit       move    #$174,SERPER    ; Baud-Rate einstellen
        rts


*** RawMayGetChar                      $f8291A

* Status der Seriellen-Schnittstelle holen

RawMayGetChar   moveq   #-1,D0
        move    SETDATR,D1      ; Datenbyte der
                        ; seriellen Schnittstelle
        btst    #14,D1  ; RBF, ist der Puffer voll?
        beq     RMGC_NoChar
        move    #%100000000000,INTREQ ; Puffer-voll melden
        and.l   #%11111,D1      ; Databits ausmaskieren
        move.l  D1,D0   ; Daten der ser. Schnitt. in d0
RMGC_NoChar     rts


*++ _WaitForRawChr

* auf Zeichen bzw. Daten der Ser. Schnittstelle warten

_WaitForRawChr  bsr     RawMayGetChar   ; Zeichen holen
        tst.l   D0      ; ist noch -1?
        bmi     _WaitForRawChr
        rts


*** RawPutChar(char)(d0)                       $f82942

* Zeichen ausgeben

RawPutChar      tst.b   D0      ; existiert ein Zeichen?
        beq     RPC_NoChar      ; kein Zeichen empfangen
        move    D0,-(SP)        ; Zeichen sichern
        cmp.b   #10,D0  ; ein LF?
        bne     RPC_NoLF        ; kein Zeilenende
        moveq   #13,D0  ; LF nach CR wandeln
        bsr     RPC_JetztCR     ; uns einspringen
RPC_NoLF        move    (SP)+,D0        ; Zeichen wiederholen
RPC_JetztCR
RPC_Loop        nop             ; kurz warten
        move    SERDATR,D1      ; Zeichen lesen
```

```
        btst    #13,D1  ; ist TBE gesetzt, also
                        ; Sende-Puffer leer?
        beq     RPC_Loop        ; warten bis gefüllt

        and     #%11111111,D0   ; Zeichen extrahieren
        or      #%100000000,D0  ; Stop-Bit setzen
        move    D0,SERDAT       ; Daten senden

        bsr     RawMayGetChar   ; Zeichen zurück
RPC_Loop_       cmp.b   #19,D0  ; DC3 vergleichen, kein Ende
        bne     PRC_OKZeichen   ; brauchbares Zeichen = weiter
        bsr     _WaitForRawChr  ; bis ein Zeichen kommt
        bra     RPC_Loop_       ; bis was kommt

PRC_OKZeichen   cmp.b   #127,D0
        bne     RPC_NoChar
        bsr     Debug   ; Zeichen kamen an, also Debug
        bra     RPC_Loop_

RPC_NoChar      rts


*++ RawStrgOut(Strg)(a0)                        $f8298A

* String wegschicken

_RawStrgOut     move.b  (A0)+,D0        ; Zeichen holen
        beq     _RSO_End        ; Kein Zeichen
        cmp.b   #10,D0  ; LF?
        bne     RSO_NoLF
        moveq   #13,D0  ; CR als Zeichen
        bsr     RSO_NoLF        ; ausgeben,
        moveq   #10,D0  ; und dann ein LF
RSO_NoLF        bsr     RawPutChar
        bra     _RawStrgOut

_RSO_End        rts


/* Schnipp */


(LIB_OPEN)???   move.l  A6,D0
        addq    #1,lib_OpenCnt(A6) ; ein Benutzer mehr
        rts

LIB_CLOSE       subq    #1,lib_OpenCnt(A6) ; ein Benutzer weniger
LIB_EXPLUNGE
LIB_EXTFUNC     moveq   #0,D0
        rts


*** CopyMemQuick(source,dest,size)(A0/A1,D0)            $f829fc

CopyMemQuick    moveq   #0,D1
        bra     _CMQuick


*** CopyMem(source,dest,size)(A0/A1,D0)         $f82a00

CopyMem moveq   #12,D1  ; ab 12 Bytes kopieren, sonst
        cmp.l   D1,D0   ; lohn der Auffwand nicht
        bcs     CopyMemBytes
        move.l  A0,D1   ; Quelle nach D1
        btst    #0,D1   ; ist Quelle gerade Zahl?
        beq     CMemQuellGerad  ; Quelle ist gerade
        move.b  (A0)+,(A1)+     ; Quellebyte nach Zielbyte, jetzt
                        ; kann auf gearaden Adressen
                        ; kopiert wreden
        subq.l  #1,D0   ; da ja schon ein Byte kopiert
                        ; ist, Länge -1
CMemQuellGerad  move.l  A1,D1   ; das Ziel nach D1
        btst    #0,D1   ; ist gerade Zahl?
        bne     CopyMemBytes    ; Quelle ist gerade, aber
                        ; Ziel nicht, man muss leider
                        ; byteweise kopieren, Schade!
        move.l  D0,D1   ; Größe des Blockes
        and     #%11,D1 ; letzten zwei Bits löschen
                        ; also durch vier teilbar
_CMQuick        move.l  D1,-(SP)
        moveq   #120,D1 ; Aufwand abschätzen
        cmp.l   D1,D0   ; sind es mehr als 120 Bytes?
        bcs     CopyMemLong     ; nein, dann lohnt s sich nicht
        movem.l D2-D7/A2-A6,-(SP) ; alte Regs sichern
CopyMem120Loop  movem.l (A0)+,D1-D7/A2-A6 ; von der Quelle in die Regs
```

```
        movem.l D1-D7/A2-A6,(A1)  ; von den Regs in das Ziel
        moveq   #48,D1
        add.l   D1,A1   ; Quelle 48 Bytes weiter
        sub.l   D1,D0   ; 48 Bytes von der Länge
        cmp.l   D1,D0   ; weitermachen?
        bcc     CopyMem120Loop  ; ja, dann Loop, sonst
        movem.l (SP)+,D2-D7/A2-A6 ; Register wiederherstellen
CopyMemLong     lsr.l   #2,D0   ; durch Acht
        beq     CpyMemRest120   ; fein, dann alles in Longs
        subq.l  #1,D0
        move.l  D0,D1
        swap    D0
CopyMemNo120    move.l  (A0)+,(A1)+
        dbra    D1,CopyMemNo120

        dbra    D0,CopyMemNo120

CpyMemRest120   move.l  (SP)+,D1
        beq     CopyMemEnd
        moveq   #0,D0
        bra     CopyMemByte

CopyMemBytes    move    D0,D1
        swap    D0
        bra     CopyMemByte

CoyMemByteLoop  move.b  (A0)+,(A1)+
CopyMemByte     dbra    D1,CoyMemByteLoop

        dbra    D0,CoyMemByteLoop

CopyMemEnd      rts


************* Semaphoren ****************************************

*** Procure(semaport,bidMsg)(A0/A1)             $f82a70

* Semaphore mit Message unterbrechen

        addq    #1,sm_Bids(A0)
        bne     ProcGiveMess
        move.l  A1,sm_SigTask(A0) ; Zeiger auf Task, der
                        ; signalisiert werden soll
        moveq   #1,D0   ; Return 1, keine Mess
Proc_rts        rts

ProcGiveMess    jsr     PutMsg(A6)
        moveq   #0,D0   ; Return 0, Mess geputted
        bra     Proc_rts        ; Oh, Oh, Oh, schwach!


*** Vacate(Semaphore)(A0)                       $f82a86

* Unterbrechung rückgängig

Vacate  clr.l   sm_SigTask(A0)  ; kein sig. Task
        subq    #1,sm_Bids(A0)
        bge     Vacate_GetMess
Vacate_rts      rts

Vacate_GetMess  move.l  a0,-(SP)        ; Sem sichern
        jsr     GetMsg(a6)      ; Nachricht holen
        move.l  (SP)+,a0        ; Sem wieder
        move.l  d0,sm_SigTask(a0) ; Message sichern
        beq.s   Vacate_rts      ; keine Message nuß auch nicht
                        ; beantwotet werden
        move.l  d0,a1   ; Antwort nach A1
        jsr     ReplyMsg(a6)    ; beantworten
        bra.s   Vacate_rts      ; Schon wieder!

/* Schnipp:
*** InitSemaphore                       $f82AA8
*** ObtainSemaphore                     $f82acc
*** ReleaseSemaphore                    $f82b20
*** AttemptSemaphore                    $f82bbe
*** ObtainSemaphoreList                 $f82bea
*** ReleaseSemaphoreList                        $f82c60
*** AddSemaphore                        $f82c76
*** RemSemaphore                        $f82c88
*** FindSemaphore                       $f82c8c
*** Debug                               $f82D44
*/
```

Alle Befehle nach Opcode sortiert im Überblick

Table 7. Mnemoniks nach Opcode sortiert

Mnemonik	Bitkombination (15-0)
ORI	%00000000–––
ANDI	%00000010–––
SUBI	%00000100–––
ADDI	%00000110–––
EORI	%00001010–––
CMPI	%00001100–––
BTST	%0000–100––
BCHG	%0000–101––
BCLR	%0000–110––
BSET	%0000–111––
MOVE	%00–––––
NEGX	%01000000–––
CLR	%01000010–––
NEG	%01000100–––
NOT	%01000110–––
SWAP	%0100100001000–
EXT/EXTB	%0100100–000–
TST	%01001010–––
NBCD	%0100100000––
PEA	%0100100001––
TAS	%0100101011––
ILLEGAL	%0100101011111100
CHK	%0100––––
LEA	%0100–111––
TRAP	%010011100100–-
LINK	%0100111001010–
UNLK	%0100111001011–
RESET	%0100111001110000
NOP	%0100111001110001
STOP	%0100111001110010
RTE	%0100111001110011
RTD	%0100111001110100
RTS	%0100111001110101
TRAPV	%0100111001110110
RTR	%0100111001110111
JSR	%0100111010––
JMP	%0100111011––
SUBQ	%0101–0–––
Scc	%0101––-11––
DBcc	%0101––-11001–

BRA	%0110000000000000
BRA.S	%01100000——–
BSR	%0110000100000000
SSR.S	%01100001——–
Bcc	%0110————
MOVEQ	%0111—0——–
DIVS	%1000—111——
DIVU	%1000—011——
OR	%1000————
SUB	%1001————
SUBX	%1001—1–00—-
SUBA	%1001————
CMP	%1011———— (?)
EOR	%1011———— (?)
CMPA	%1011———— (?)
CMPM	%1011—1–001—
MULS	%1100—111——
MULU	%1100—011——
AND	%1100————
ADDQ	%1101—0——–
ADD	%1101————
ADDX	%1101—1–00—-
ADDA	%1101————
ASR	%1110000011——
ASL	%1110000111——
ASR	%1110001011——
ASL	%1110001111——
ROXR	%1110010011——
ROXL	%1110010111——
ROR	%1110011011——
ROL	%1110011111——
ASR	%1110—0—00—
ASL	%1110—1—00—
LSR	%1110—0—01—
LSL	%1110—1—01—
ROXR	%1110—0—10—
ROXL	%1110—1—10—
ROR	%1110—0—11—
ROL	%1110—1—11—

Alle Befehle nach Namen sortiert im Überblick

Table 8. Mnemoniks alphabetisch sortiert

Mnemonik	Bitkombination (15-0)

ADD	1101————
ADDA	1101————
ADDI	00000110——–
ADDQ	1101—0——–
ADDX	1101—1–00—-
AND	1100————
ANDI	00000010——–
ASL	1110—1—00—
ASL	1110000111——
ASL	1110001111——
ASR	1110—0—00—
ASR	1110000011——
ASR	1110001011——
Bcc	0110————
BCHG	0000—101——
BCLR	0000—110——
BRA	0110000000000000
BRA.S	01100000——–
BSET	0000—111——
BSR	0110000100000000
BSR.S	01100001——–
BTST	0000—100——
CHK	0100————
CLR	01000010——–
CMP	1011———— (?)
CMPA	1011———— (?)
CMPI	00001100——–
CMPM	1011—1–001—
DBcc	0101—-11001—
DIVS	1000—111——
DIVU	1000—011——
EOR	1011———— (?)
EORI	00001010——–
EXT/EXTB	0100100—000—
ILLEGAL	0100101011111100
JMP	0100111011——
JSR	0100111010——
LEA	0100—111——
LINK	0100111001010—
LSL	1110—1—01—
LSR	1110—0—01—
MOVEQ	0111—0——–
MULS	1100—111——
MULU	1100—011——

NBCD	0100100000−−
NEG	01000100−−−
NEGX	01000000−−−
NOP	0100111001110001
NOT	01000110−−−
OR	1000−−−−
ORI	00000000−−−
PEA	0100100001−−
RESET	0100111001110000
ROL	1110−1−11−
ROL	1110011111−−
ROR	1110−0−11−
ROR	1110011011−−
ROXL	1110−1−10−
ROXL	1110010111−−
ROXR	1110−0−10−
ROXR	1110010011−−
RTD	0100111001110100
RTE	0100111001110011
RTR	0100111001110111
RTS	0100111001110101
Scc	0101−-11−−
STOP	0100111001110010
SUB	1001−−−−
SUBA	1001−−−−
SUBI	00000100−−−
SUBQ	0101−0−−−
SUBX	1001−1−00−-
SWAP	0100100001000−
TAS	0100101011−−
TRAP	010011100100−-
TRAPV	0100111001110110
TST	01001010−−−
UNLK	0100111001011−

Libraryfunktionen

Im folgenden werden Libraries und einige Devices mit ihren Offsets und Übergabeparametern beschrieben.

arp.library

DOS-Routinen

```
- 30    -$01e   Open(name,accessMode)(D1/D2)
- 36    -$024   Close(file)(D1)
- 42    -$02a   Read(file,buffer,length)(D1/D2/D3)
- 48    -$030   Write(file,buffer,length)(D1/D2/D3)
- 54    -$036   Input()
- 60    -$03c   Output()
- 66    -$042   Seek(file,position,offset)(D1/D2/D3)
```

```
- 72    -$048   DeleteFile(name)(D1)
- 78    -$04e   Rename(oldName,newName)(D1/D2)
- 84    -$054   Lock(name,type)(D1/D2)
- 90    -$05a   UnLock(lock)(D1)
- 96    -$060   DupLock(lock)(D1)
-102    -$066   Examine(lock,fileInfoBlock)(D1/D2)
-108    -$07c   ExNext(lock,fileInfoBlock)(D1/D2)
-114    -$072   Info(lock,parameterBlock)(D1/D2)
-120    -$078   CreateDir(name)(D1)
-126    -$07e   CurrentDir(lock)(D1)
-132    -$084   IoErr()
-138    -$08a   CreateProc(name,pri,segList,stackSize)(D1/D2/D3/D4)
-142    -$090   Exit(returnCode)(D1)
-150    -$096   LoadSeg(fileName)(D1)
-156    -$09c   UnLoadSeg(segment)(D1)
```

Nicht freigegebene Funktionen

```
-162    -$0a2   GetPacket(wait)(D1)
-168    -$0a8   QueuePacket(packet)(D1)
```

Wieder freigegebene Funktionen

```
-174    -$0ae   DeviceProc(name)(D1)
-180    -$0b4   SetComment(name,comment)(D1/D2)
-186    -$0ba   SetProtection(name,mask)(D1/D2)
-192    -$0c0   DateStamp(date)(D1)
-198    -$0c6   Delay(timeout)(D1)
-204    -$0cc   WaitForChar(file,timeout)(D1/D2)
-210    -$0d2   ParentDir(lock)(D1)
-216    -$0d8   IsInteractive(file)(D1)
-222    -$0de   Execute(string,file,file)(D1/D2/D3)
```

Ende der neuen DOS-Funktionen

```
-228    -$0e4   Printf(string,stream)(a0/a1)
-234    -$0ea   FPrintf(file,string,stream)(d0,a0/a1)
-230    -$0f0   Puts(string)(a1)
-246    -$0f6   ReadLine(buffer)(a0)
```

Nicht freigegebene Funktionen

```
-252    -$0fc   GADS(line,len,help,args,tplate)(a0,d0,a1/a2/a3)
-258    -$102   Atol(string)(a0)
```

Wieder freigegebene Funktionen

```
-264    -$108   EscapeString(string)(a0)
-270    -$10e   CheckAbort(func)(a1)
-276    -$114   CheckBreak(masks,func)(d1/a1)
-282    -$11a   Getenv(string,buffer,size)(a0/a1,d0)
-288    -$120   Setenv(varname,value)(a0/a1)
-294    -$126   FileRequest(FileRequester)(a0)
-300    -$12c   CloseWindowSafely(Window1,Window2)(a0/a1)
-306    -$132   CreatePort(name,pri)(a0,d0)
-312    -$138   DeletePort(port)(a1)
-318    -$13e   SendPacket(action,args,handler)(d0/a0/a1)
-324    -$144   InitStdPacket(action,args,packet,replyport)(d0/a0/a1/a2)
-330    -$14a   PathName(lock,buffer,componentcount)(d0/a0,d1)
-336    -$150   Assign(logical,physical)(a0/a1)
-342    -$156   DosAllocMem(size)(d0)
-348    -$15c   DosFreeMem(dosblock)(a1)
-354    -$162   BtoCStr(cstr,bstr,maxlength)(a0,d0/d1)
-360    -$168   CtoBstr(cstr,bstr,maxlength)(a0,d0/d1)
-366    -$16e   GetDevInfo(devnode)(a2)
-372    -$174   FreeTaskResList()
-378    -$17a   ArpExit(rc,result2)(d0/d2)
```

Nicht freigegebene Funktionen

```
-384    -$180   ArpAlloc(size)(d0)
-390    -$186   ArpAllocMem(size,requirements)(d0/d1)
-396    -$18c   ArpOpen(name,mode)(d1/d2)
-402    -$192   ArpDupLock(lock)(d1)
-408    -$198   ArpLock(name,mode)(d1/d2)
-414    -$19e   RListAlloc(reslist,size)(a0,d0)
```

Wieder freigegebene Funktionen

```
-420    -$1a4   FindCLI(clinum)(d0)
-426    -$1aa   QSort(base,rsize,bsize,comp)(a0,d0/d1,a1)
-432    -$1b0   PatternMatch(pattern,string)(a0/a1)
-438    -$1b6   FindFirst(pattern,AnchorPath)(d0/a0)
-444    -$1bc   FindNext(AnchorPath)(a0)
-450    -$1c2   FreeAnchorChain(AnchorPath)(a0)
-456    -$1c8   CompareLock(lock1,lock2)(d0/d1)
-462    -$1ce   FindTaskResList()
-468    -$1d4   CreateTaskResList()
-474    -$1da   FreeResList(freelist)(a1)
-480    -$1e0   FreeTrackedItem(item)(a1)
```

Nicht freigegebene Funktion

```
-486    -$1e6   GetTracker()
```

Wieder freigegebene Funktionen

```
-492    -$1ec   GetAccess(tracker)(a1)
-498    -$1f2   FreeAccess(tracker)(a1)
-504    -$1f8   FreeDAList(node)(a1)
-510    -$1fe   AddDANode(data,dalist,length,id)(a0/a1,d0/d1)
-516    -$204   AddDADevs(dalist,select)(a0,d0)
-522    -$20a   Strcmp(s1,s2)(a0/a1)
-528    -$210   Strncmp(s1,s2,count)(a0/a1,d0)
-534    -$216   Toupper(character)(d0)
-540    -$21c   SyncRun(name,command,input,output)(a0/a1,d0/d1)
```

Neue Arp-Funktionen

```
-546    -$222   ASyncRun(name,command,pcb)(a0/a1/a2)
-552    -$228   LoadPrg(name)(d1)
-558    -$22e   PreParse(source,dest)(a0/a1)
-564    -$234   StamptoStr(datetime)(a0)
-570    -$23a   StrtoStamp(datetime)(a0)
-576    -$240   ObtainResidentPrg(name)(a0)
-582    -$246   AddResidentPrg(Segment,name)(d1/a0)
-588    -$24c   RemResidentPrg(name)(a0)
-594    -$252   UnLoadPrg(segment)(d1)
-600    -$258   LMult(a,b)(d0/d1)
-606    -$25e   LDiv(a,b)(d0/d1)
-612    -$264   LMod(a,b)(d0/d1)
-618    -$26a   CheckSumPrg(ResidentNode)(d0)
-624    -$270   TackOn(pathname,filename)(a0/a1)
-630    -$276   BaseName(name)(a0)
-636    -$27c   ReleaseResidentPrg(segment)(d1)
-642    -$282   Sprintf(buffer/string/arg)(d0/a0/a1)
-648    -$288   GetKeyWordIndex(kwrd/tmplate)(a0/a1)
-654    -$28e   ArpOpenLibrary(library/version)(a1/d0)
-660    -$294   ArpAllocFreq()
```

asl.library

```
- 30    -$01e   AllocFileRequest()
- 36    -$024   FreeFileRequest(FileReq)(a0)
- 42    -$02a   RequestFile((FileReq)(a0)
- 48    -$030   AllocAslRequest(type,TagList)(a0)
- 54    -$036   FreeAslRequest(request)
- 60    -$03c   AslRequest(request,TagList)(a1)
```

diskfont.library

```
- 30    -$01e   OpenDiskFont(textAttr)(A0)
- 36    -$024   AvailFonts(buffer,bufBytes,flags)(A0,D0/D1)
```

Neue Funktionen seit der Version 34 (V1.3)

```
- 42    -$02a   NewFontContents(fontsLock,fontName)(A0/A1)
- 48    -$030   DisposeFontContents(fontContentsHeader)(A1)
```

Neue Funktionen seit der Version 36 (V2.0)

```
- 54    -$036   NewScaledDiskFont(sourceFont,destTextAttr)(a0/a1)
```

commodities.library

Objektutilities

```
- 30    -$01e   CreateCxObj(type,arg1,arg2)(d0/a0/a1)
- 36    -$024   CxBroker(nb,error)(a0,d0)
- 42    -$02a   ActivateCxObj(co,true)(a0,d0)
- 48    -$030   DeleteCxObj(co)(a0)
- 54    -$036   DeleteCxObjAll(co)(a0)
- 60    -$03c   CxObjType(co)(a0)
- 66    -$042   CxObjError(co)(a0)
- 72    -$048   ClearCxObjError(co)(a0)
- 78    -$04e   SetCxObjPri(co,pri)(a0,d0)
```

Objekt Attachment

```
- 84    -$054   AttachCxObj(headobj,co)(a0/a1)
- 90    -$05a   EnqueueCxObj(headobj,co)(a0/a1)
- 96    -$060   InsertCxObj(headobj,co,pred)(a0/a1/a2)
-102    -$066   RemoveCxObj(co)(a0)
```

Nicht freigegebene Funktion

```
-108    -$07c   FindBroker(name)(a0)
```

Wieder freigegebene Funktionen

```
-114    -$072   SetTranslate(translator,ie)(a0/a1)
-120    -$078   SetFilter(filter,text)(a0/a1)
-126    -$07e   SetFilterIX(filter,ix)(a0/a1)
-132    -$084   ParseIX(descr,ix)(a0/a1)
```

Messages

```
-138    -$08a   CxMsgType(cxm)(a0)
-142    -$090   CxMsgData(cxm)(a0)
-150    -$096   CxMsgID(cxm)(a0)
-156    -$09c   DivertCxMsg(cxm,headobj,ret)(a0/a1/a2)
-162    -$0a2   RouteCxMsg(cxm,co)(a0/a1)
-168    -$0a8   DisposeCxMsg(cxm)(a0)
```

Input-Event-Verarbeitung

```
-174    -$0ae   InvertKeyMap(ansicode,ie,km)(d0/a0/a1)
-180    -$0b4   AddIEvents(ie)(a0)
```

Nicht freigegebene Funktion

```
-186    -$0ba   CopyBrokerList(blist)(a0)
-192    -$0c0   FreeBrokerList(list)(a0)
-198    -$0c6   BrokerCommand(name,id)(a0,d0)
```

console.device

```
- 42    -$02a   CDInputHandler(events,device)(A0/A1)
- 48    -$030   RawKeyConvert(events,buffer,length,keyMap)(A0/A1,D1/A2)
```

dos.library

```
- 30    -$01e   Open(name,accessMode)(D1/D2)
- 36    -$024   Close(file)(D1)
- 42    -$02a   Read(file,buffer,length)(D1/D2/D3)
- 48    -$030   Write(file,buffer,length)(D1/D2/D3)
- 54    -$036   Input()
- 60    -$03c   Output()
- 66    -$042   Seek(file,position,offset)(D1/D2/D3)
- 72    -$048   DeleteFile(name)(D1)
- 78    -$04e   Rename(oldName,newName)(D1/D2)
```

```
- 84    -$054   Lock(name,type)(D1/D2)
- 90    -$05a   UnLock(lock)(D1)
- 96    -$060   DupLock(lock)(D1)
-102    -$066   Examine(lock,fileInfoBlock)(D1/D2)
-108    -$06c   ExNext(lock,fileInfoBlock)(D1/D2)
-114    -$072   Info(lock,parameterBlock)(D1/D2)
-120    -$078   CreateDir(name)(D1)
-126    -$07e   CurrentDir(lock)(D1)
-132    -$084   IoErr()
-138    -$08a   CreateProc(name,pri,segList,stackSize)(D1/D2/D3/D4)
-144    -$090   Exit(returnCode)(D1)
-150    -$096   LoadSeg(fileName)(D1)
-156    -$09c   UnLoadSeg(segment)(D1)
```

Nicht freigegebene Funktionen

```
-162    -$0a2   GetPacket(wait)(D1)
-168    -$0a8   QueuePacket(packet)(D1)
```

Wieder freigegebene Funktionen

```
-174    -$0ae   DeviceProc(name)(D1)
-180    -$0b4   SetComment(name,comment)(D1/D2)
-186    -$0ba   SetProtection(name,mask)(D1/D2)
-192    -$0c0   DateStamp(date)(D1)
-198    -$0c6   Delay(timeout)(D1)
-204    -$0cc   WaitForChar(file,timeout)(D1/D2)
-210    -$0d2   ParentDir(lock)(D1)
-216    -$0d8   IsInteractive(file)(D1)
-222    -$0de   Execute(string,file,file)(D1/D2/D3)
```

Neue Funktionen seit der Version 36 (V2.0)
Objektverwaltung

```
-228    -$0e4   AllocDosObject(type,tags)(d1/d2)
-234    -$0ea   FreeDosObject(type,ptr)(d1/d2)
```

Pakete

```
-240    -$0f0   DoPkt(port,action,arg1,arg2,arg3,arg4,arg5)(d1-d7)
-246    -$0f6   SendPkt(dp,port,replyport)(d1/d2/d3)
-252    -$0fc   WaitPkt()()
-258    -$102   ReplyPkt(dp,res1,res2)(d1/d2/d3)
-264    -$108   AbortPkt(port,pkt)(d1/d2)
-270    -$10e   LockRecord(fh,offset,length,mode,timeout)(d1/d2/d3/d4/d5)
-276    -$114   LockRecords(recArray,timeout)(d1/d2)
-282    -$11a   UnLockRecord(fh,offset,length)(d1/d2/d3)
-288    -$120   UnLockRecords(recArray)(d1)
```

IO

```
-294    -$126   SelectInput(fh)(d1)
-300    -$12c   SelectOutput(fh)(d1)
-306    -$132   FGetC(fh)(d1)
-312    -$138   FPutC(fh,ch)(d1/d2)
-318    -$13e   UnGetC(fh,character)(d1/d2)
-324    -$144   FRead(fh,block,blocklen,number)(d1/d2/d3/d4)
-330    -$14a   FWrite(fh,block,blocklen,number)(d1/d2/d3/d4)
-336    -$150   FGets(fh,buf,buflen)(d1/d2/d3)
-342    -$156   FPuts(fh,str)(d1/d2)
-348    -$15c   VFWritef(fh,format,argarray)(d1/d2/d3)
-354    -$162   VFPrintf(fh,format,argarray)(d1/d2/d3)
-360    -$168   Flush(fh)(d1)
-366    -$16e   SetVBuf(fh,buff,type,size)(d1/d2/d3/d4)
```

DOS Objekte

```
-372    -$174   DupLockFromFH(fh)(d1)
-378    -$17a   OpenFromLock(lock)(d1)
-384    -$180   ParentOfFH(fh)(d1)
-390    -$186   ExamineFH(fh,fib)(d1/d2)
-396    -$18c   SetFileDate(name,date)(d1/d2)
-402    -$192   NameFromLock(lock,buffer,len)(d1/d2/d3)
-408    -$198   NameFromFH(fh,buffer,len)(d1/d2/d3)
-414    -$19e   SplitName(name,seperator,buf,oldpos,size)(d1-d5)
-420    -$1a4   SameLock(lock1,lock2)(d1/d2)
-426    -$1aa   SetMode(fh,mode)(d1/d2)
```

```
-432    -$1b0   ExAll(lock,buffer,size,data,control)(d1/d2/d3/d4/d5)
-438    -$1b6   ReadLink(port,lock,path,buffer,size)(d1/d2/d3/d4/d5)
-444    -$1bc   MakeLink(name,dest,soft)(d1/d2/d3)
-450    -$1c2   ChangeMode(type,fh,newmode)(d1/d2/d3)
-456    -$1c8   SetFileSize(fh,pos,mode)(d1/d2/d3)
```

Fehlerbehandlung

```
-462    -$1ce   SetIoErr(result)(d1)
-468    -$1d4   Fault(code,header,buffer,len)(d1/d2/d3/d4)
-474    -$1da   PrintFault(code,header)(d1/d2)
-480    -$1e0   ErrorReport(code,type,arg1,device)(d1/d2/d3/d4)
```

Nicht belegt

```
-486    -$1e6
```

Prozessverwaltung

```
-492    -$1ec   Cli()()
-498    -$1f2   CreateNewProc(tags)(d1)
-504    -$1f8   RunCommand(seg,stack,paramptr,paramlen)(d1/d2/d3/d4)
-510    -$1fe   GetConsoleTask()()
-516    -$204   SetConsoleTask(task)(d1)
-522    -$20a   GetFileSysTask()()
-528    -$210   SetFileSysTask(task)(d1)
-534    -$216   GetArgStr()()
-540    -$21c   SetArgStr(string)(d1)
-546    -$222   FindCliProc(num)(d1)
-552    -$228   MaxCli()()
-558    -$22e   SetCurrentDirName(name)(d1)
-564    -$234   GetCurrentDirName(buf,len)(d1/d2)
-570    -$23a   SetProgramName(name)(d1)
-576    -$240   GetProgramName(buf,len)(d1/d2)
-582    -$246   SetPrompt(name)(d1)
-588    -$24c   GetPrompt(buf,len)(d1/d2)
-594    -$252   SetProgramDir(lock)(d1)
-600    -$258   GetProgramDir()()
```

Deviselisten verarbeiten

```
-606    -$25e   SystemTagList(command,tags)(d1/d2)
-612    -$264   AssignLock(name,lock)(d1/d2)
-618    -$26a   AssignLate(name,path)(d1/d2)
-624    -$270   AssignPath(name,path)(d1/d2)
-630    -$276   AssignAdd(name,lock)(d1/d2)
-636    -$27c   RemAssignList(name,lock)(d1/d2)
-642    -$282   GetDeviceProc(name,dp)(d1/d2)
-648    -$288   FreeDeviceProc(dp)(d1)
-654    -$28e   LockDosList(flags)(d1)
-660    -$294   UnLockDosList(flags)(d1)
-666    -$29a   AttemptLockDosList(flags)(d1)
-672    -$2a0   RemDosEntry(dlist)(d1)
-678    -$2a6   AddDosEntry(dlist)(d1)
-684    -$2ac   FindDosEntry(dlist,name,flags)(d1/d2/d3)
-690    -$2b2   NextDosEntry(dlist,flags)(d1/d2)
-696    -$2b8   MakeDosEntry(name,type)(d1/d2)
-702    -$2be   FreeDosEntry(dlist)(d1)
-708    -$2c4   IsFileSystem(name)(d1)
```

Handler Interface

```
-714    -$2ca   Format(filesystem,volumename,dostype)(d1/d2/d3)
-720    -$2d0   Relabel(drive,newname)(d1/d2)
-726    -$2d6   Inhibit(name,onoff)(d1/d2)
-732    -$2dc   AddBuffers(name,number)(d1/d2)
```

Date, Time Routines

```
-738    -$2e2   CompareDates(date1,date2)(d1/d2)
-744    -$2e8   DateToStr(datetime)(d1)
-750    -$2ee   StrToDate(datetime)(d1)
```

Image Management

```
-756    -$2f4   InternalLoadSeg(fh,table,funcarray,stack)(d0/a0/a1/a2)
```

```
-762    -$2fa   InternalUnLoadSeg(seglist,freefunc)(d1/a1)
-768    -$300   NewLoadSeg(file,tags)(d1/d2)
-774    -$306   AddSegment(name,seg,system)(d1/d2/d3)
-780    -$30c   FindSegment(name,seg,system)(d1/d2/d3)
-786    -$312   RemSegment(seg)(d1)
-792    -$318
-798    -$31e
-802    -$324
```

exec.library

```
- 30    -$01e   Supervisor(userFunction)(a5)
```

Nicht freigegebene Funktionen

```
- 36    -$024   ExitIntr()
- 42    -$02a   Schedule()
- 48    -$030   Reschedule()
- 54    -$036   Switch()
- 60    -$03c   Dispatch()
- 66    -$042   Exception()
```

Wieder freigegebene Funktionen
Modulfunktionen

```
- 72    -$048   InitCode(startClass,version)(D0/D1)
- 78    -$04e   InitStruct(initTable,memory,size)(A1/A2,D0)
- 84    -$054   MakeLibrary(fucIni,strctIni,libIni,datSiz,codSiz)(A0-A2,D0-D1)
- 90    -$05a   MakeFunctions(target,functionArray,funcDispBase)(A0-A2)
- 96    -$060   FindResident(name)(A1)
-102    -$066   InitResident(resident,segList)(A1,D1)
```

Disagnose

```
-108    -$06c   Alert(alertNum,parameters)(D7,A5)
-114    -$072   Debug()
```

Interrupts

```
-120    -$078   Disable()
-126    -$07e   Enable()
-132    -$084   Forbid()
-138    -$08a   Permit()
-144    -$090   SetSR(newSR,mask)(D0/D1)
-150    -$096   SuperState()
-156    -$09c   UserState(sysStack)(D0)
-162    -$0a2   SetIntVector(intNumber,interrupt)(D0/A1)
-168    -$0a8   AddIntServer(intNumber,interrupt)(D0/A1)
-174    -$0ae   RemIntServer(intNumber,interrupt)(D0/A1)
-180    -$0b4   Cause(interrupt)(A1)
```

Speicherverwaltung

```
-186    -$0ba   Allocate(freeList,byteSize)(A0,D0)
-192    -$0c0   Deallocate(freeList,memoryBlock,byteSize)(A0/A1,D0)
-198    -$0c6   AllocMem(byteSize,requirements)(D0/D1)
-204    -$0cc   AllocAbs(byteSize,location)(D0/A1)
-210    -$0d2   FreeMem(memoryBlock,byteSize)(A1,D0)
-216    -$0d8   AvailMem(requirements)(D1)
-222    -$0de   AllocEntry(entry)(A0)
-228    -$0e4   FreeEntry(entry)(A0)
```

Listen

```
-234    -$00ea  Insert(list,node,pred)(A0/A1/A2)
-240    -$0F0   AddHead(list,node)(A0/A1)
-246    -$0F6   AddTail(list,node)(A0/A1)
-252    -$0FC   Remove(node)(A1)
-258    -$102   RemHead(list)(A0)
-264    -$108   RemTail(list)(A0)
-270    -$10E   Enqueue(list,node)(A0/A1)
-276    -$114   FindName(list,name)(A0/A1)
```

Tasks

```
-282    -$11A   AddTask(task,initPC,finalPC)(A1/A2/A3)
-288    -$120   RemTask(task)(A1)
-294    -$126   FindTask(name)(A1)
-300    -$12C   SetTaskPri(task,priority)(A1,D0)
-306    -$132   SetSignal(newSignals,signalSet)(D0/D1)
-312    -$138   SetExcept(newSignals,signalSet)(D0/D1)
-318    -$13E   Wait(signalSet)(D0)
-324    -$144   Signal(task,signalSet)(A1,D0)
-330    -$14A   AllocSignal(signalNum)(D0)
-336    -$150   FreeSignal(signalNum)(D0)
-342    -$156   AllocTrap(trapNum)(D0)
-348    -$15C   FreeTrap(trapNum)(D0)
```

Messages

```
-354    -$162   AddPort(port)(A1)
-360    -$168   RemPort(port)(A1)
-366    -$16e   PutMsg(port,message)(A0/A1)
-372    -$174   GetMsg(port)(A0)
-378    -$17a   ReplyMsg(message)(A1)
-384    -$180   WaitPort(port)(A0)
-390    -$186   FindPort(name)(A1)
```

Libraries

```
-396    -$18c   AddLibrary(library)(A1)
-402    -$192   RemLibrary(library)(A1)
-408    -$198   OldOpenLibrary(libName)(A1)
-414    -$19e   CloseLibrary(library)(A1)
-420    -$1a4   SetFunction(library,funcOffset,funcEntry)(A1,A0,D0)
-426    -$1aa   SumLibrary(library)(A1)
```

Devices

```
-432    -$1b0   AddDevice(device)(A1)
-438    -$1b6   RemDevice(device)(A1)
-444    -$1bc   OpenDevice(devName,unit,ioRequest,flags)(A0,D0/A1,D1)
-450    -$1d2   CloseDevice(ioRequest)(A1)
-456    -$1d8   DoIO(ioRequest)(A1)
-462    -$1de   SendIO(ioRequest)(A1)
-468    -$1d4   CheckIO(ioRequest)(A1)
-474    -$1da   WaitIO(ioRequest)(A1)
-480    -$1e0   AbortIO(ioRequest)(A1)
```

Resources

```
-486    -$1e6   AddResource(resource)(A1)
-492    -$1ec   RemResource(resource)(A1)
-498    -$1f2   OpenResource(resName,version)(A1,D0)
```

Nicht freigegebene Funktionen

```
-504    -$1f8   RawIOInit()
-510    -$1fe   RawMayGetChar()
-516    -$204   RawPutChar(char)(d0)
```

Wieder freigegebene Funktionen

```
-522    -$20a   RawDoFmt()(A0/A1/A2/A3)
-528    -$210   GetCC()
-534    -$216   TypeOfMem(address)(A1);
-540    -$21c   Procure(semaport,bidMsg)(A0/A1)
-546    -$222   Vacate(semaport)(A0)
-552    -$228   OpenLibrary(libName,version)(A1,D0)
```

Semaphoren Unterstützung seid 1.2

```
-558    -$22e   InitSemaphore(sigSem)(A0)
-564    -$234   ObtainSemaphore(sigSem)(A0)
-570    -$23a   ReleaseSemaphore(sigSem)(A0)
-576    -$240   AttemptSemaphore(sigSem)(A0)
-582    -$246   ObtainSemaphoreList(sigSem)(A0)
-588    -$24c   ReleaseSemaphoreList(sigSem)(A0)
-594    -$252   FindSemaphore(sigSem)(A0)
-600    -$258   AddSemaphore(sigSem)(A0)
-606    -$25e   RemSemaphore(sigSem)(A0)
```

Neu ab 1.3

```
-612    -$264   SumKickData()
-618    -$26a   AddMemList(size,attributes,pri,base,name)(D0/D1/D2/A0/A1)
-624    -$270   CopyMem(source,dest,size)(A0/A1,D0)
-630    -$276   CopyMemQuick(source,dest,size)(A0/A1,D0)
```

Funktionen in V36 (OS 2.0)

Cache

```
-636    -$27c   CacheClearU()()
-642    -$282   CacheClearE(address,length,caches)(a0,d0/d1)
-648    -$288   CacheControl(cacheBits,cacheMask)(d0/d1)
```

Misc

```
-654    -$28e   CreateIORequest(port,size)(a0,d0)
-660    -$294   DeleteIORequest(iorequest)(a0)
-666    -$29a   CreateMsgPort()()
-672    -$2a0   DeleteMsgPort(port)(a0)
-678    -$2a6   ObtainSemaphoreShared(sigSem)(a0)
```

Neue Speicherverwaltungroutinen

```
-684    -$2ac   AllocVec(byteSize,requirements)(d0/d1)
-690    -$2b2   FreeVec(memoryBlock)(a1)
-696    -$2b8   CreatePrivatePool(requirements,Size,Thresh)(d0-d2)
-702    -$2be   DeletePrivatePool(poolHeader)(a0)
-708    -$2c4   AllocPooled(memSize,poolHeader)(d0/a0)
-714    -$2ca   FreePooled(memory,poolHeader)(a1,a0)
```

Sonstiges

Nicht freigegeben

```
-720    -$2d0   ExecReserved00(nothing)(d0)
```

Das kann jeder benutzen

```
-726    -$2d6   ColdReboot()()
-732    -$2dc   StackSwap(newSize,newSP,newStack)(d0/d1/a0)
```

Task-Bäume

```
-738    -$2e2   ChildFree(tid)(d0)
-744    -$2e8   ChildOrphan(tid)(d0)
-750    -$2ee   ChildStatus(tid)(d0)
-756    -$2f4   ChildWait(tid)(d0)
```

Für später privat

```
-762    -$2fa   ExecReserved01(nothing)(d0)
-768    -$300   ExecReserved02(nothing)(d0)
-774    -$306   ExecReserved03(nothing)(d0)
-780    -$30c   ExecReserved04(nothing)(d0)
```

expansion.library

```
 -30    -$01e   AddConfigDev(configDev)(A0)
```

Nicht freigegebene Funktionen

```
 -36    -$024   expansionUnused()
```

Wieder freigegebene Funktionen

```
- 42    -$02a   AllocBoardMem(slotSpec)(D0)
- 48    -$030   AllocConfigDev()
- 54    -$036   AllocExpansionMem(numSlots,SlotAlign,SlotOffset)(D0-D2)
- 60    -$03c   ConfigBoard(board,configDev)(A0/A1)
```

```
- 66    -$042   ConfigChain(baseAddr)(A0)
- 72    -$048   FindConfigDev(oldConfDev,manufacturer,product)(A0,D0/D1)
- 78    -$04e   FreeBoardMem(startSlot,slotSpec)(D0/D1)
- 84    -$054   FreeConfigDev(configDev)(A0)
- 90    -$05a   FreeExpansionMem(startSlot,numSlots)(D0/D1)
- 96    -$060   ReadExpansionByte(board,offset)(A0,D0)
-102    -$066   ReadExpansionRom(board,configDev)(A0/A1)
-108    -$06c   RemConfigDev(configDev)(A0)
-114    -$072   WriteExpansionByte(board,offset,byte)(A0,D0/D1)
-120    -$078   ObtainConfigBinding()
-126    -$07e   ReleaseConfigBinding()
-132    -$084   SetCurrentBinding(currentBinding,bindingSize)(A0,D0)
-138    -$08a   GetCurrentBinding(currentBinding,bindingSize)(A0,D0)
-144    -$090   MakeDosNode(parmPacket)(A0)
-150    -$096   AddDosNode(bootPri,flags,dosNode)(D0/D1/A0)
```

graphics.library

Text Routinen

```
- 30    -$01e   BltBitMap(srcBitMap,srcX,srcY,destBitMap,destX,destY,
                sizeX,sizeY,minterm,mask,tempA)(A0,D0/D1,A1,D2-D7/A2)
- 36    -$024   BltTemplate(source,srcX,srcMod,destRastPort,destX,destY,
                sizeX,sizeY)(A0,D0/D1/A1,D2-D5)
- 42    -$02a   ClearEOL(rastPort)(A1)
- 48    -$030   ClearScreen(rastPort)(A1)
- 54    -$036   TextLength(RastPort,string,count)(A1,A0,D0)
- 60    -$03c   Text(RastPort,string,count)(A1,A0,D0)
- 66    -$042   SetFont(RastPortID,textFont)(A1,A0)
- 72    -$048   OpenFont(textAttr)(A0)
- 78    -$04e   CloseFont(textFont)(A1)
- 84    -$054   AskSoftStyle(rastPort)(A1)
- 90    -$05a   SetSoftStyle(rastPort,style,enable)(A1,D0/D1)
```

Gels Routinen

```
- 96    -$060   AddBob(bob,rastPort)(A0,A1)
-102    -$066   AddVSprite(vSprite,rastPort)(A0/A1)
-108    -$06c   DoCollision(rasPort)(A1)
-114    -$072   DrawGList(rastPort,viewPort)(A1,A0)
-120    -$078   InitGels(dummyHead,dummyTail,GelsInfo)(A0/A1/A2)
-126    -$07e   InitMasks(vSprite)(A0)
-132    -$084   RemIBob(bob,rastPort,viewPort)(A0/A1/A2)
-138    -$08a   RemVSprite(vSprite)(A0)
-144    -$090   SetCollision(type,routine,gelsInfo)(D0/A0/A1)
-150    -$096   SortGList(rastPort)(A1)
-156    -$09c   AddAnimOb(obj,animationKey,rastPort)(A0/A1/A2)
-162    -$0a2   Animate(animationKey,rastPort)(A0/A1)
-168    -$0a8   GetGBuffers(animationObj,rastPort,doubleBuffer)(A0/A1,D0)
-174    -$0ae   InitGMasks(animationObj)(A0)
-180    -$0b4   DrawEllipse(rastPort,cx,cy,a,b)(A1,D0/D1/D2/D3)
-186    -$0ba   AreaEllipse(rastPort,cx,cy,a,b)(A1,D0/D1/D2/D3)
```

Restliche Graphics-Routinen

```
-192    -$0c0   LoadRGB4(viewPort,colors,count)(A0/A1,D0)
-198    -$0c6   InitRastPort(rastPort)(A1)
-204    -$0cc   InitVPort(viewPort)(A0)
-210    -$0d2   MrgCop(view)(A1)
-216    -$0d8   MakeVPort(view,viewPort)(A0/A1)
-222    -$0de   LoadView(view)(A1)
-228    -$0e4   WaitBlit()
-234    -$0ea   SetRast(rastPort,color)(A1,D0)
-240    -$0f0   Move(rastPort,x,y)(A1,D0/D1)
-246    -$0f6   Draw(rastPort,x,y)(A1,D0/D1)
-252    -$0fc   AreaMove(rastPort,x,y)(A1,D0/D1)
-258    -$102   AreaDraw(rastPort,x,y)(A1,D0/D1)
-264    -$108   AreaEnd(rastPort)(A1)
-270    -$10e   WaitTOF()
-276    -$114   QBlit(blit)(A1)
-282    -$11a   InitArea(areaInfo,vectorTable,vectorTableSize)(A0/A1,D0)
-288    -$120   SetRGB4(viewPort,index,r,g,b)(A0,D0/D1/D2/D3)
-294    -$126   QBSBlit(blit)(A1)
-300    -$12c   BltClear(memory,size,flags)(A1,D0/D1)
-306    -$132   RectFill(rastPort,xl,yl,xu,yu)(A1,D0/D1/D2/D3)
-312    -$138   BltPattern(rastPort,ras,xl,yl,maxX,maxY,fillBytes)(a1,a0,D0-D4)
-318    -$13e   ReadPixel(rastPort,x,y)(A1,D0/D1)
-324    -$144   WritePixel(rastPort,x,y)(A1,D0/D1)
-330    -$14a   Flood(rastPort,mode,x,y)(A1,D2,D0/D1)
-336    -$150   PolyDraw(rastPort,count,polyTable)(A1,D0,A0)
```

```
-342    -$156   SetAPen(rastPort,pen)(A1,D0)
-348    -$15c   SetBPen(rastPort,pen)(A1,D0)
-354    -$162   SetDrMd(rastPort,drawMode)(A1,D0)
-360    -$168   InitView(view)(A1)
-366    -$16e   CBump(copperList)(A1)
-372    -$174   CMove(copperList,destination,data)(A1,D0/D1)
-378    -$17a   CWait(copperList,x,y)(A1,D0/D1)
-384    -$180   VBeamPos()
-390    -$186   InitBitMap(bitMap,depth,width,height)(A0,D0/D1/D2)
-396    -$18c   ScrollRaster(rp,dX,dY,minx,miny,maxx,maxy)(A1,D0-D5)
-402    -$192   WaitBOVP(viewport)(a0)
-408    -$198   GetSprite(simplesprite,num)(a0,d0)
-414    -$19e   FreeSprite(num)(d0)
-420    -$1a4   ChangeSprite(vp,simplesprite,data)(a0/a1/a2)
-426    -$1aa   MoveSprite(viewport,simplesprite,x,y)(a0/a1,d0/d1)
-432    -$1b0   LockLayerRom(layer)(a5)
-438    -$1b6   UnlockLayerRom(layer)(a5)
-444    -$1bc   SyncSBitMap(l)(a0)
-450    -$1c2   CopySBitMap(l)(a0)
-456    -$1c8   OwnBlitter()()
-462    -$1ce   DisownBlitter()()
-468    -$1d4   InitTmpRas(tmpras,buff,size)(a0/a1,d0)
-474    -$1da   AskFont(rastPort,textAttr)(A1,A0)
-480    -$1e0   AddFont(textFont)(A1)
-486    -$1e6   RemFont(textFont)(A1)
-492    -$1ec   AllocRaster(width,height)(D0/D1)
-498    -$1f2   FreeRaster(planeptr,width,height)(A0,D0/D1)
-504    -$1f8   AndRectRegion(rgn,rect)(A0/A1)
-510    -$1fe   OrRectRegion(rgn,rect)(A0/A1)
-516    -$204   NewRegion()()
-522    -$20a   ClearRectRegion(rgn)(A0/A1)
-528    -$210   ClearRegion(rgn)(A0)
-534    -$216   DisposeRegion(rgn)(A0)
-540    -$21c   FreeVPortCopLists(viewport)(a0)
-546    -$222   FreeCopList(coplist)(a0)
-552    -$228   ClipBlit(srcrp,srcX,srcY,destrp,destX,destY,sizeX,sizeY,
                minterm)(A0,D0/D1,A1,D2/D3/D4/D5/D6)
-558    -$22e   XorRectRegion(rgn,rect)(a0/a1)
-564    -$234   FreeCprList(cprlist)(a0)
-570    -$23a   GetColorMap(entries)(d0)
-576    -$240   FreeColorMap(colormap)(a0)
-582    -$246   GetRGB4(colormap,entry)(a0,d0)
-588    -$24c   ScrollVPort(vp)(a0)
-594    -$252   UCopperListInit(copperlist,num)(a0,d0)
-600    -$258   FreeGBuffers(animationObj,rastPort,doubleBuffer)(A0/A1,D0)
-606    -$25e   BltBitMapRastPort(srcbm,srcx,srcy,destrp,destX,destY,
                sizeX,sizeY,minterm)(A0,D0/D1,A1,D2/D3/D4/D5/D6)
-612    -$264   OrRegionRegion(src,dst)(a0/a1)
-618    -$26a   XorRegionRegion(src,dst)(a0/a1)
-624    -$270   AndRegionRegion(src,dst)(a0/a1)
-630    -$276   SetRGB4CM(cm,i,r,g,b)(a0,d0/d1/d2/d3)
-636    -$27c   BltMaskBitMapRastPort(srcbm,srcx,srcy,destrp,destX,destY,
                sizeX,sizeY,minterm,bltmask)(A0,D0/D1,A1,D2-D6,A2)
```

Nicht freigegebene Funktionen

```
-642    -$282   GraphicsReserved1()()
-648    -$288   GraphicsReserved2()()
```

Wieder freigegebene Funktionen

```
-654    -$28e   AttemptLockLayerRom(layer)(a5)
```

icon.library

```
- 30    -$01e   GetWBObject()()
- 36    -$024   PutWBObject()()
- 42    -$02a   GetIcon()()
- 48    -$030   PutIcon()()
```

Wieder freigegebene Funktionen

```
- 54    -$036   FreeFreeList(freelist)(A0)
```

Nicht freigegebene Funktionen

```
- 60    -$03c   FreeWBObject(WBObject)(A0)
```

```
- 66    -$042   AllocWBObject()()
```

Wieder freigegebene Funktionen

```
- 72    -$048   ddFreeList(freelist,mem,size)(A0/A1/A2)
```

Normale Funktionen

```
- 78    -$04e   GetDiskObject(name)(A0)
- 84    -$054   PutDiskObject(name,diskobj)(A0,A1)
- 90    -$05a   FreeDiskObject(diskobj)(A0)
- 96    -$060   FindToolType(toolTypeArray,typeName)(A0/A1)
-102    -$066   MatchToolValue(typeString,value)(A0/A1)
-108    -$06c   BumpRevision(newname,oldname)(A0/A1)
```

intuition.library

```
- 30    -$01e   OpenIntuition()()
- 36    -$024   Intuition(ievent)(A0)
- 42    -$02a   AddGadget(AddPtr,Gadget,Position)(A0/A1,D0)
- 48    -$030   ClearDMRequest(Window)(A0)
- 54    -$036   ClearMenuStrip(Window)(A0)
- 60    -$03c   ClearPointer(Window)(A0)
- 66    -$042   CloseScreen(Screen)(A0)
- 72    -$048   CloseWindow(Window)(A0)
- 78    -$04e   CloseWorkBench()()
- 84    -$054   CurrentTime(Seconds,Micros)(A0/A1)
- 90    -$05a   DisplayAlert(AlertNumber,String,Height)(D0/A0,D1)
- 96    -$060   DisplayBeep(Screen)(A0)
-102    -$066   DoubleClick(sseconds,smicros,cseconds,cmicros)(D0-D3)
-108    -$06c   DrawBorder(RPort,Border,LeftOffset,TopOffset)(A0/A1,D0/D1)
-114    -$072   DrawImage(RPort,Image,LeftOffset,TopOffset)(A0/A1,D0/D1)
-120    -$078   EndRequest(requester,window)(A0/A1)
-126    -$07e   GetDefPrefs(preferences,size)(A0,D0)
-132    -$084   GetPrefs(preferences,size)(A0,D0)
-138    -$08a   InitRequester(req)(A0)
-144    -$090   ItemAddress(MenuStrip,MenuNumber)(A0,D0)
-150    -$096   ModifyIDCMP(Window,Flags)(A0,D0)
-156    -$09c   ModifyProp(Gadget,Ptr,Req,Flags,HPos,VPos,HBody,VBody)
                (A0-A2,D0-D4)
-162    -$0a2   MoveScreen(Screen,dx,dy)(A0,D0/D1)
-168    -$0a8   MoveWindow(window,dx,dy)(A0,D0/D1)
-174    -$0ae   OffGadget(Gadget,Ptr,Req)(A0/A1/A2)
-180    -$0b4   OffMenu(Window,MenuNumber)(A0,D0)
-186    -$0ba   OnGadget(Gadget,Ptr,Req)(A0/A1/A2)
-192    -$0c0   OnMenu(Window,MenuNumber)(A0,D0)
-198    -$0c6   OpenScreen(OSargs)(A0)
-204    -$0cc   OpenWindow(OWargs)(A0)
-210    -$0d2   OpenWorkBench()()
-216    -$0d8   PrintIText(rp,itext,left,top)(A0/A1,D0/D1)
-222    -$0de   RefreshGadgets(Gadgets,Ptr,Req)(A0/A1/A2)
-228    -$0e4   RemoveGadget(RemPtr,Gadget)(A0/A1)
-234    -$0ea   ReportMouse(Boolean,Window)(D0/A0)
-240    -$0f0   Request(Requester,Window)(A0/A1)
-246    -$0f6   ScreenToBack(Screen)(A0)
-252    -$0fc   ScreenToFront(Screen)(A0)
-258    -$102   SetDMRequest(Window,req)(A0/A1)
-264    -$108   SetMenuStrip(Window,Menu)(A0/A1)
-270    -$10e   SetPointer(Window,Pointer,Height,Width,Xoffset,Yoffset)(A0/A1,D0-D3)
-276    -$114   SetWindowTitles(window,windowtitle,screentitle)(A0/A1/A2)
-282    -$11a   ShowTitle(Screen,ShowIt)(A0,D0)
-288    -$120   SizeWindow(window,dx,dy)(A0,D0/D1)
-294    -$126   ViewAddress()()
-300    -$12c   ViewPortAddress(window)(A0)
-306    -$132   WindowToBack(window)(A0)
-312    -$138   WindowToFront(window)(A0)
-318    -$13e   WindowLimits(window,minwidth,minheight,maxwidth,
                maxheight)(A0,D0-D3)
-324    -$144   SetPrefs(preferences,size,flag)(A0,D0/D1)
-330    -$14a   IntuiTextLength(itext)(A0)
-336    -$150   WBenchToBack()()
-342    -$156   WBenchToFront()()
-348    -$15c   AutoRequest(Window,Body,PText,NText,PFlag,NFlag,W,H)
                (A0,A1,A2,A3,D0,D1,D2,D3)
-354    -$162   BeginRefresh(Window)(A0)
-360    -$168   BuildSysRequest(Window,Body,PosText,NegText,Flags,W,H)
                (A0,A1,A2,A3,D0,D1,D2)
-366    -$16e   EndRefresh(Window,Complete)(A0,D0)
-372    -$174   FreeSysRequest(Window)(A0)
-378    -$17a   MakeScreen(Screen)(A0)
```

```
-384    -$180   RemakeDisplay()()
-390    -$186   RethinkDisplay()()
-396    -$18c   AllocRemember(RememberKey,Size,Flags)(A0,D0,D1)
-402    -$192   AlohaWorkbench(wbport)(A0)
-408    -$198   FreeRemember(RememberKey,ReallyForget)(A0,D0)
-414    -$19e   LockIBase(dontknow)(D0)
-420    -$1a4   UnlockIBase(IBLock)(A0)
-426    -$1aa   GetScreenData(buffer,size,type,screen)(A0,D0,D1,A1)
-432    -$1b0   RefreshGList(Gadgets,Ptr,Req,NumGad)(A0/A1/A2,D0)
-438    -$1b6   AddGList(AddPtr,Gadget,Position,NumGad,Requester)
               (A0/A1,D0/D1/A2)
-444    -$1bc   RemoveGList(RemPtr,Gadget,NumGad)(A0/A1,D0)
-450    -$1c2   ActivateWindow(Window)(A0)
-456    -$1c8   RefreshWindowFrame(Window)(A0)
-462    -$1ce   ActivateGadget(Gadgets,Window,Req)(A0/A1/A2)
-468    -$1d4   NewModifyProp(Gadget,Ptr,Req,Flags,HPos,VPos,HBody,VBody,
               NumGad)(A0/A1/A2,D0/D1/D2/D3/D4/D5)
```

layers.library

```
- 30    -$01e   InitLayers(li)(A0)
- 36    -$024   CreateUpfrontLayer(li,bm,x0,y0,x1,y1,flags,bm2)
               (A0/A1,D0-D4,A2)
- 42    -$02a   CreateBehindLayer(li,bm,x0,y0,x1,y1,flags,bm2)
               (A0/A1,D0/D1/D2/D3/D4,A2)
- 48    -$030   UpfrontLayer(li,layer)(A0/A1)
- 54    -$036   BehindLayer(li,layer)(A0/A1)
- 60    -$03c   MoveLayer(li,layer,dx,dy)(A0/A1,D0/D1)
- 66    -$042   SizeLayer(li,layer,dx,dy)(A0/A1,D0/D1)
- 72    -$048   ScrollLayer(li,layer,dx,dy)(A0/A1,D0/D1)
- 78    -$04e   BeginUpdate(layer)(A0)
- 84    -$054   EndUpdate(layer,flag)(A0,d0)
- 90    -$05a   DeleteLayer(li,layer)(A0/A1)
- 96    -$060   LockLayer(li,layer)(A0/A1)
-102    -$066   UnlockLayer(layer)(A0)
-108    -$06c   LockLayers(li)(A0)
-114    -$072   UnlockLayers(li)(A0)
-120    -$078   LockLayerInfo(li)(A0)
-126    -$07e   SwapBitsRastPortClipRect(rp,cr)(A0/A1)
-132    -$084   WhichLayer(li,x,y)(a0,d0/d1)
-138    -$08a   UnlockLayerInfo(li)(A0)
-144    -$090   NewLayerInfo()()
-150    -$096   DisposeLayerInfo(li)(a0)
-156    -$09c   FattenLayerInfo(li)(a0)
-162    -$0a2   ThinLayerInfo(li)(a0)
-168    -$0a8   MoveLayerInFrontOf(layer_to_move,layer_to_be_infront_of)(a0/a1)
-174    -$0ae   InstallClipRegion(layer,region)(a0/a1)
```

Neue Funktionen seit V2.0

```
-180    -$0b4   MoveSizeLayer(layer,dx,dy,dw,dh)(a0,d0/d1/d2/d3)
-186    -$0ba   CreateUpfrontHookLayer(li,bm,x0,y0,x1,y1,flags,
               hook,bm2)(a0/a1,d0/d1/d2/d3/d4/a3,a2)
-192    -$0c0   CreateBehindHookLayer(li,bm,x0,y0,x1,y1,flags,
               hook,bm2)(a0/a1,d0/d1/d2/d3/d4/a3,a2)
-198    -$0c6   InstallLayerHook(layer,hook)(a0/a1)
```

mathffp.library

```
- 30    -$01e   SPFix(float)(D0)
- 36    -$024   SPFlt(integer)(D0)
- 42    -$02a   SPCmp(leftFloat,rightFloat)(D1,D0)
- 48    -$030   SPTst(float)(D1)
- 54    -$036   SPAbs(float)(D0)
- 60    -$03c   SPNeg(float)(D0)
- 66    -$042   SPAdd(leftFloat,rightFloat)(D1,D0)
- 72    -$048   SPSub(leftFloat,rightFloat)(D1,D0)
- 78    -$04e   SPMul(leftFloat,rightFloat)(D1,D0)
- 84    -$054   SPDiv(leftFloat,rightFloat)(D1,D0)
```

Neue Funktionen, die seit V1.2 hinzukamen

```
- 90    -$05a   SPFloor(float)(D0)
- 96    -$060   SPCeil(float)(D0)
```

mathieeedoubbas.library

```
- 30    -$01e   IEEEDPFix(double)(D0/D1)
- 36    -$024   IEEEDPFlt(integer)(D0)
- 42    -$02a   IEEEDPCmp(double,double)(D0/D1/D2/D3)
- 48    -$030   IEEEDPTst(double)(D0/D1)
- 54    -$036   IEEEDPAbs(double)(D0/D1)
- 60    -$03c   IEEEDPNeg(double)(D0/D1)
- 66    -$042   IEEEDPAdd(double,double)(D0/D1/D2/D3)
- 72    -$048   IEEEDPSub(double,double)(D0/D1/D2/D3)
- 78    -$04e   IEEEDPMul(double,double)(D0/D1/D2/D3)
- 84    -$054   IEEEDPDiv(double,double)(D0/D1/D2/D3)
```

Funktionen, die ab V1.2 kamen

```
- 90    -$05a   IEEEDPFloor(double)(D0/D1)
- 96    -$060   IEEEDPCeil(double)(D0/D1)
```

mathieeedoubtrans.library

```
- 30    -$01e   IEEEDPAtan(double)(D0/D1)
- 36    -$024   IEEEDPSin(double)(D0/D1)
- 42    -$02a   IEEEDPCos(double)(D0/D1)
- 48    -$030   IEEEDPTan(double)(D0/D1)
- 54    -$036   IEEEDPSincos(double,pf2)(A0,D0/D1)
- 60    -$03c   IEEEDPSinh(double)(D0/D1)
- 66    -$042   IEEEDPCosh(double)(D0/D1)
- 72    -$048   IEEEDPTanh(double)(D0/D1)
- 78    -$04e   IEEEDPExp(double)(D0/D1)
- 84    -$054   IEEEDPLog(double)(D0/D1)
- 90    -$05a   IEEEDPPow(exp,arg)(D2/D3,D0/D1)
- 96    -$060   IEEEDPSqrt(double)(D0/D1)
-102    -$066   IEEEDPTieee(double)(D0/D1)
-108    -$06c   IEEEDPFieee(single)(D0)
-114    -$072   IEEEDPAsin(double)(D0/D1)
-120    -$078   IEEEDPAcos(double)(D0/D1)
-126    -$07e   IEEEDPLog10(double)(D0/D1)
```

mathtrans.library

```
- 30    -$01e   SPAtan(float)(D0)
- 36    -$024   SPSin(float)(D0)
- 42    -$02a   SPCos(float)(D0)
- 48    -$030   SPTan(float)(D0)
- 54    -$036   SPSincos(leftFloat,rightFloat)(D1,D0)
- 60    -$03c   SPSinh(float)(D0)
- 66    -$042   SPCosh(float)(D0)
- 72    -$048   SPTanh(float)(D0)
- 78    -$04e   SPExp(float)(D0)
- 84    -$054   SPLog(float)(D0)
- 90    -$05a   SPPow(leftFloat,rightFloat)(D1,D0)
- 96    -$060   SPSqrt(float)(D0)
-102    -$066   SPTieee(float)(D0)
-108    -$06c   SPFieee(integer)(D0)
```

Neue Funktionen, die seit V1.2 hinzukamen

```
-114    -$072   SPAsin(float)(D0)
-120    -$078   SPAcos(float)(D0)
-126    -$07e   SPLog10(float)(D0)
```

Requester.library

```
- 30    -$01e   Center(nw,x,y)(a0/d0/d1)
- 36    -$024   SetSize(MaxValue,ViewSize)(d0/d1)
- 42    -$02a   SetLocation(MaxValue,ViewSize,Value)(d0/d1/d2)
- 48    -$030   ReadLocation(MaxValue,ViewSize,PotValue)(d0/d1/d2)
- 54    -$036   Format(Buffer,string,values)(a2/a0/a1)
```

Nicht freigegebene Funktionen

```
- 60    -$03c   FakeFunction1
- 66    -$042   FakeFunction2
```

```
- 72    -$048   FakeFunction3
- 78    -$04e   FakeFunction4
```

Wieder freigegebene Funktionen

```
- 84    -$054   FileRequester(FileRequesterStructure)(a0)
- 90    -$05a   ColorRequester(DesiredColor)(d0)
- 96    -$060   DrawBox(rp,MinX,MinY,MaxX,MaxY)(a0/d0/d1/d2/d3)
-102    -$066   MakeButton
-108    -$06c   MakeScrollBar(Buffer,Flags,Size,X,Y)(a0/d0/d1/d2/d3)
-114    -$072   PurgeFiles(FileRequesterStructure)(a0)
-120    -$078   GetFontHeightAndWidth
-126    -$07e   MakeGadget(Buffer,String,X,Y)(a0/a1/d0/d1)
-132    -$084   MakeString(Buffer,StringBuff,UndoBuff,MaxWidthBits,
                MaxNumChars,X,Y)(a0/a1/a2/d0/d1/d2/d3)
-138    -$08a   MakeProp
-144    -$090   LinkGadget(Buffer,String,nw,X,Y)(a0/a1/a3/d0/d1)
-150    -$096   LinkStringGadget(Buffer,StringBuf,UndoBuf,nw,WidthBits,
                NumChars,X,Y)(a0/a1/a2/a3/d0/d1/d2/d3)
-156    -$09c   LinkPropGadget(Buffer,nw,Width,Height,Flags,
                LeftEdge,TopEdge)(a0/a3/d0/d1/d2/d3/d4)
-162    -$0a2   GetString(buffer,title,window,visiblechars,
                maxchars)(a0/a1/a2/d0/d1)
-168    -$0a8   RealTimeScroll(ScrollStruct)(a0)
-174    -$0ae   TextRequest
-180    -$0b4   GetLong(GetLongStruct)(a0)
-186    -$0ba   RawKeyToAscii(Code,Qualifier,IAddress)(d0/d1/a0)
-192    -$0c0   ExtendedColorRequester(ExtendedColorRequester)(a0)
-198    -$0c6   NewGetString(getstringstruct)(a0)
```

timer.device

```
- 42    -$02a   AddTime(dest,src)(A0/A1)
- 48    -$030   SubTime(dest,src)(A0/A1)
- 54    -$036   CmpTime(dest,src)(A0/A1)
```

translator.library

```
- 30    -$01e   Translate(inputString,inputLength,outputBuffer,bufferSize)(A0,D0/A1,D1)
```

Strukturoffsets

```
AnimComp
  $026    38  sizeof(AnimComp)
  $000     0  Flags
  $002     2  Timer
  $004     4  TimeSet
  $006     6  NextComp
  $00a    10  PrevComp
  $00e    14  NextSeq
  $012    18  PrevSeq
  $016    22  AnimCRoutine
  $01a    26  YTrans
  $01c    28  XTrans
  $01e    30  HeadOb
  $022    34  AnimBob

AnimOb
  $02a    42  sizeof(AnimOb)
  $000     0  NextOb
  $004     4  PrevOb
  $008     8  Clock
  $00c    12  AnOldY
  $00e    14  AnOldX
  $010    16  AnY
  $012    18  AnX
  $014    20  YVel
  $016    22  XVel
  $018    24  YAccel
  $01a    26  XAccel
  $01c    28  RingYTrans
  $01e    30  RingXTrans
  $020    32  AnimORoutine
  $024    36  HeadComp
  $028    40  AUserExt
```

```
AreaInfo
  $018    24  sizeof(AreaInfo)
  $000     0  VctrTbl
  $004     4  VctrPtr
  $008     8  FlagTbl
  $00c    12  FlagPtr
  $010    16  Count
  $012    18  MaxCount
  $014    20  FirstX
  $016    22  FirstY

AudChannel
  $010    16  sizeof(AudChannel)
  $000     0  ptr
  $004     4  len
  $006     6  per
  $008     8  vol
  $00a    10  dat
  $00c    12  pad[4]

AvailFonts
  $00a    10  sizeof(AvailFonts)
  $000     0  af_Type
  $002     2  af_Attr

AvailFontsHeader
  $002     2  sizeof(AvailFontsHeader)
  $000     0  afh_NumEntries

BitMap
  $028    40  sizeof(BitMap)
  $000     0  BytesPerRow
  $002     2  Rows
  $004     4  Flags
  $005     5  Depth
  $006     6  pad
  $008     8  Planes[8]

Bob
  $020    32  sizeof(Bob)
  $000     0  Flags
  $002     2  SaveBuffer
  $006     6  ImageShadow
  $00a    10  Before
  $00e    14  After
  $012    18  BobVSprite
  $016    22  BobComp
  $01a    26  DBuffer
  $01e    30  BUserExt

BoolInfo
  $00a    10  sizeof(BoolInfo)
  $000     0  Flags
  $002     2  Mask
  $006     6  Reserved

BootBlock
  $00c    12  sizeof(BootBlock)
  $000     0  id[0]
  $004     4  chksum
  $008     8  dosblock

BootNode
  $014    20  sizeof(BootNode)
  $000     0  bn_Node
  $00e    14  bn_Flags
  $010    16  bn_DeviceNode

Border
  $010    16  sizeof(Border)
  $000     0  LeftEdge
  $002     2  TopEdge
  $004     4  FrontPen
  $005     5  BackPen
  $006     6  DrawMode
  $007     7  Count
  $008     8  XY
  $00c    12  NextBorder

CIA
  $f02  3842  sizeof(CIA)
  $000     0  ciapra
  $001     1  pad0[255]
  $100   256  ciaprb
  $101   257  pad1[255]
```

```
  $200   512  ciaddra
  $201   513  pad2[255]
  $300   768  ciaddrb
  $301   769  pad3[255]
  $400  1024  ciatalo
  $401  1025  pad4[255]
  $500  1280  ciatahi
  $501  1281  pad5[255]
  $600  1536  ciatblo
  $601  1537  pad6[255]
  $700  1792  ciatbhi
  $701  1793  pad7[255]
  $800  2048  ciatodlow
  $801  2049  pad8[255]
  $900  2304  ciatodmid
  $901  2305  pad9[255]
  $a00  2560  ciatodhi
  $a01  2561  pad10[255]
  $b00  2816  unusedreg
  $b01  2817  pad11[255]
  $c00  3072  ciasdr
  $c01  3073  pad12[255]
  $d00  3328  ciaicr
  $d01  3329  pad13[255]
  $e00  3584  ciacra
  $e01  3585  pad14[255]
  $f00  3840  ciacrb

ClipRect
  $024    36  sizeof(ClipRect)
  $000     0  Next
  $004     4  prev
  $008     8  lobs
  $00c    12  BitMap
  $010    16  bounds
  $018    24  _p1
  $01c    28  _p2
  $020    32  reserved

ClipboardUnitPartial
  $012    18  sizeof(ClipboardUnitPartial)
  $000     0  cu_Node
  $00e    14  cu_UnitNum

ColorMap
  $008     8  sizeof(ColorMap)
  $000     0  Flags
  $001     1  Type
  $002     2  Count
  $004     4  ColorTable

CommandLineInterface
  $040    64  sizeof(CommandLineInterface)
  $000     0  Result2
  $004     4  SetName
  $008     8  CommandDir
  $00c    12  ReturnCode
  $010    16  CommandName
  $014    20  FailLevel
  $018    24  Prompt
  $01c    28  StandardInput
  $020    32  CurrentInput
  $024    36  CommandFile
  $028    40  Interactive
  $02c    44  Background
  $030    48  CurrentOutput
  $034    52  DefaultStack
  $038    56  StandardOutput
  $03c    60  Module

ConUnit
  $128   296  sizeof(ConUnit)
  $000     0  MP
  $022    34  Window
  $026    38  XCP
  $028    40  YCP
  $02a    42  XMax
  $02c    44  YMax
  $02e    46  XRSize
  $030    48  YRSize
  $032    50  XROrigin
  $034    52  YROrigin
  $036    54  XRExtant
  $038    56  YRExtant
  $03a    58  XMinShrink
```

```
  $03c    60  YMinShrink
  $03e    62  XCCP
  $040    64  YCCP
  $042    66  KeyMapStruct
  $062    98  TabStops[160]
  $102   258  Mask
  $103   259  FgPen
  $104   260  BgPen
  $105   261  AOLPen
  $106   262  DrawMode
  $107   263  AreaPtSz
  $108   264  AreaPtrn
  $10c   268  Minterms[8]
  $114   276  Font
  $118   280  AlgoStyle
  $119   281  TxFlags
  $11a   282  TxHeight
  $11c   284  TxWidth
  $11e   286  TxBaseline
  $120   288  TxSpacing
  $122   290  Modes[3]
  $125   293  RawEvents[3]

ConfigDev
  $044    68  sizeof(ConfigDev)
  $000     0  Node
  $00e    14  Flags
  $00f    15  Pad
  $010    16  Rom
  $020    32  BoardAddr
  $024    36  BoardSize
  $028    40  SlotAddr
  $02a    42  SlotSize
  $02c    44  Driver
  $030    48  NextCD
  $034    52  Unused[14]

CopIns
  $006     6  sizeof(CopIns)
  $000     0  OpCode
  $002     2  u3
  $002     2  u3.nxtlist
  $002     2  u3.u4
  $002     2  u3.u4.u1
  $002     2  u3.u4.u1.VWaitPos
  $002     2  u3.u4.u1.DestAddr
  $004     4  u3.u4.u2
  $004     4  u3.u4.u2.HWaitPos
  $004     4  u3.u4.u2.DestData

CopList
  $022    34  sizeof(CopList)
  $000     0  Next
  $004     4  _CopList
  $008     8  _ViewPort
  $00c    12  CopIns
  $010    16  CopPtr
  $014    20  CopLStart
  $018    24  CopSStart
  $01c    28  Count
  $01e    30  MaxCount
  $020    32  DyOffset

CurrentBinding
  $010    16  sizeof(CurrentBinding)
  $000     0  ConfigDev
  $004     4  FileName
  $008     8  ProductString
  $00c    12  ToolTypes

Custom
  $1c0   448  sizeof(Custom)
  $000     0  bltddat
  $002     2  dmaconr
  $004     4  vposr
  $006     6  vhposr
  $008     8  dskdatr
  $00a    10  joy0dat
  $00c    12  joy1dat
  $00e    14  clxdat
  $010    16  adkconr
  $012    18  pot0dat
  $014    20  pot1dat
  $016    22  potinp
  $018    24  serdatr
```

```
  $01a    26  dskbytr
  $01c    28  intenar
  $01e    30  intreqr
  $020    32  dskpt
  $024    36  dsklen
  $026    38  dskdat
  $028    40  refptr
  $02a    42  vposw
  $02c    44  vhposw
  $02e    46  copcon
  $030    48  serdat
  $032    50  serper
  $034    52  potgo
  $036    54  joytest
  $038    56  strequ
  $03a    58  strvbl
  $03c    60  strhor
  $03e    62  strlong
  $040    64  bltcon0
  $042    66  bltcon1
  $044    68  bltafwm
  $046    70  bltalwm
  $048    72  bltcpt
  $04c    76  bltbpt
  $050    80  bltapt
  $054    84  bltdpt
  $058    88  bltsize
  $05a    90  pad2d[6]
  $060    96  bltcmod
  $062    98  bltbmod
  $064   100  bltamod
  $066   102  bltdmod
  $068   104  pad34[8]
  $070   112  bltcdat
  $072   114  bltbdat
  $074   116  bltadat
  $076   118  pad3b[8]
  $07e   126  dsksync
  $080   128  cop1lc
  $084   132  cop2lc
  $088   136  copjmp1
  $08a   138  copjmp2
  $08c   140  copins
  $08e   142  diwstrt
  $090   144  diwstop
  $092   146  ddfstrt
  $094   148  ddfstop
  $096   150  dmacon
  $098   152  clxcon
  $09a   154  intena
  $09c   156  intreq
  $09e   158  adkcon
  $0a0   160  aud[0]
  $0a0   160  aud[0].ac_ptr
  $0a4   164  aud[0].ac_len
  $0a6   166  aud[0].ac_per
  $0a8   168  aud[0].ac_vol
  $0aa   170  aud[0].ac_dat
  $0ac   172  aud[0].ac_pad[0]
  $0e0   224  bplpt[4]
  $0f8   248  pad7c[0]
  $100   256  bplcon0
  $102   258  bplcon1
  $104   260  bplcon2
  $106   262  pad83
  $108   264  bpl1mod
  $10a   266  bpl2mod
  $10c   268  pad86[0]
  $110   272  bpldat[0]
  $11c   284  pad8e[0]
  $120   288  sprpt[0]
  $140   320  spr[0]
  $140   320  spr[0].pos
  $142   322  spr[0].ctl
  $144   324  spr[0].dataa
  $146   326  spr[0].datab
  $180   384  color[0]

DBufPacket
  $00c    12  sizeof(DBufPacket)
  $000     0  BufY
  $002     2  BufX
  $004     4  BufPath
  $008     8  BufBuffer
```

```
DateStamp
  $00c    12  sizeof(DateStamp)
  $000     0  Days
  $004     4  Minute
  $008     8  Tick

DevInfo
  $02c    44  sizeof(DevInfo)
  $000     0  Next
  $004     4  Type
  $008     8  Task
  $00c    12  Lock
  $010    16  Handler
  $014    20  StackSize
  $018    24  Priority
  $01c    28  Startup
  $020    32  SegList
  $024    36  GlobVec
  $028    40  Name

Device
  $022    34  sizeof(Device)
  $000     0  dd_Library


DeviceData

  $034    52  sizeof(DeviceData)
  $000     0  Device
  $022    34  Segment
  $026    38  ExecBase
  $02a    42  CmdVectors
  $02e    46  CmdBytes
  $032    50  NumCommands


DeviceList

  $02c    44  sizeof(DeviceList)
  $000     0  Next
  $004     4  Type
  $008     8  Task
  $00c    12  Lock
  $010    16  VolumeDate
  $01c    28  LockList
  $020    32  DiskType
  $024    36  unused
  $028    40  Name


DeviceNode

  $02c    44  sizeof(DeviceNode)
  $000     0  Next
  $004     4  Type
  $008     8  Task
  $00c    12  Lock
  $010    16  Handler
  $014    20  StackSize
  $018    24  Priority
  $01c    28  Startup
  $020    32  SegList
  $024    36  GlobalVec
  $028    40  Name

DiagArea
  $00e    14  sizeof(DiagArea)
  $000     0  Config
  $001     1  Flags
  $002     2  Size
  $004     4  DiagPoint
  $006     6  BootPoint
  $008     8  Name
  $00a    10  Reserved01
  $00c    12  Reserved02

DiscResource
  $090   144  sizeof(DiscResource)
  $000     0  Library
  $022    34  Current
  $026    38  Flags
  $027    39  pad
  $028    40  SysLib
  $02c    44  CiaResource
  $030    48  UnitID[0]
```

```
  $040    64  Waiting
  $04e    78  DiscBlock
  $064   100  DiscSync
  $07a   122  Index

DiscResourceUnit
  $056    86  sizeof(DiscResourceUnit)
  $000     0  Message
  $014    20  DiscBlock
  $02a    42  DiscSync
  $040    64  Index

DiskFontHeader
  $06a   106  sizeof(DiskFontHeader)
  $000     0  DF
  $00e    14  FileID
  $010    16  Revision
  $012    18  Segment
  $016    22  Name[0]
  $036    54  TF

DiskObject
  $04e    78  sizeof(DiskObject)
  $000     0  Magic
  $002     2  Version
  $004     4  Gadget
  $030    48  Type
  $032    50  DefaultTool
  $036    54  ToolTypes
  $03a    58  CurrentX
  $03e    62  CurrentY
  $042    66  DrawerData
  $046    70  ToolWindow
  $04a    74  StackSize

DosEnvec
  $044    68  sizeof(DosEnvec)
  $000     0  TableSize
  $004     4  SizeBlock
  $008     8  SecOrg
  $00c    12  Surfaces
  $010    16  SectorPerBlock
  $014    20  BlocksPerTrack
  $018    24  Reserved
  $01c    28  PreAlloc
  $020    32  Interleave
  $024    36  LowCyl
  $028    40  HighCyl
  $02c    44  NumBuffers
  $030    48  BufMemType
  $034    52  MaxTransfer
  $038    56  Mask
  $03c    60  BootPri
  $040    64  DosType

DosInfo
  $014    20  sizeof(DosInfo)
  $000     0  McName
  $004     4  DevInfo
  $008     8  Devices
  $00c    12  Handlers
  $010    16  NetHand

DosLibrary
  $036    54  sizeof(DosLibrary)
  $000     0  lib
  $022    34  Root
  $026    38  GV
  $02a    42  A2
  $02e    46  A5
  $032    50  A6

DosList
  $02c    44  sizeof(DosList)
  $000     0  Next
  $004     4  Type
  $008     8  Task
  $00c    12  Lock
  $010    16  misc
  $010    16  misc.handler
  $010    16  misc.handler.Handler
  $014    20  misc.handler.StackSize
  $018    24  misc.handler.Priority
  $01c    28  misc.handler.Startup
  $020    32  misc.handler.SegList
```

```
  $024    36  misc.handler.GlobVec
  $010    16  misc.volume
  $010    16  misc.volume.VolumeDate
  $01c    28  misc.volume.LockList
  $020    32  misc.volume.DiskType
  $028    40  Name

DosPacket
  $030    48  sizeof(DosPacket)
  $000     0  Link
  $004     4  Port
  $008     8  Type
  $00c    12  Res1
  $010    16  Res2
  $014    20  Arg1
  $018    24  Arg2
  $01c    28  Arg3
  $020    32  Arg4
  $024    36  Arg5
  $028    40  Arg6
  $02c    44  Arg7

DrawerData
  $038    56  sizeof(DrawerData)
  $000     0  NewWindow
  $030    48  CurrentX
  $034    52  CurrentY

ExecBase
  $24c   588  sizeof(ExecBase)
  $000     0  LibNode
  $022    34  SoftVer
  $024    36  LowMemChkSum
  $026    38  ChkBase
  $02a    42  ColdCapture
  $02e    46  CoolCapture
  $032    50  WarmCapture
  $036    54  SysStkUpper
  $03a    58  SysStkLower
  $03e    62  MaxLocMem
  $042    66  DebugEntry
  $046    70  DebugData
  $04a    74  AlertData
  $04e    78  MaxExtMem
  $052    82  ChkSum
  $054    84  IntVects[0]
  $114   276  ThisTask
  $118   280  IdleCount
  $11c   284  DispCount
  $120   288  Quantum
  $122   290  Elapsed
  $124   292  SysFlags
  $126   294  IDNestCnt
  $127   295  TDNestCnt
  $128   296  AttnFlags
  $12a   298  AttnResched
  $12c   300  ResModules
  $130   304  TaskTrapCode
  $134   308  TaskExceptCode
  $138   312  TaskExitCode
  $13c   316  TaskSigAlloc
  $140   320  TaskTrapAlloc
  $142   322  MemList
  $150   336  ResourceList
  $15e   350  DeviceList
  $16c   364  IntrList
  $17a   378  LibList
  $188   392  PortList
  $196   406  TaskReady
  $1a4   420  TaskWait
  $1b2   434  SoftInts[0]
  $202   514  LastAlert[0]
  $212   530  VBlankFrequency
  $213   531  PowerSupplyFrequency
  $214   532  SemaphoreList
  $222   546  KickMemPtr
  $226   550  KickTagPtr
  $22a   554  KickCheckSum
  $22e   558  ExecBaseReserved[0]
  $238   568  ExecBaseNewReserved[0]

ExpansionBase
  $1c8   456  sizeof(ExpansionBase)
  $000     0  LibNode
  $022    34  Flags
```

```
  $023    35  pad
  $024    36  ExecBase
  $028    40  SegList
  $02c    44  CurrentBinding
  $03c    60  BoardList
  $04a    74  MountList
  $058    88  AllocTable[0]
  $158   344  BindSemaphore
  $186   390  Int2List
  $19c   412  Int6List
  $1b2   434  Int7List

ExpansionControl
  $010    16  sizeof(ExpansionControl)
  $000     0  Interrupt
  $001     1  Reserved11
  $002     2  BaseAddress
  $003     3  Shutup
  $004     4  Reserved14
  $005     5  Reserved15
  $006     6  Reserved16
  $007     7  Reserved17
  $008     8  Reserved18
  $009     9  Reserved19
  $00a    10  Reserved1a
  $00b    11  Reserved1b
  $00c    12  Reserved1c
  $00d    13  Reserved1d
  $00e    14  Reserved1e
  $00f    15  Reserved1f

ExpansionInt
  $006     6  sizeof(ExpansionInt)
  $000     0  IntMask
  $002     2  ArrayMax
  $004     4  ArraySize

ExpansionRom
  $010    16  sizeof(ExpansionRom)
  $000     0  Type
  $001     1  Product
  $002     2  Flags
  $003     3  Reserved03
  $004     4  Manufacturer
  $006     6  SerialNumber
  $00a    10  InitDiagVec
  $00c    12  Reserved0c
  $00d    13  Reserved0d
  $00e    14  Reserved0e
  $00f    15  Reserved0f

FileHandle
  $02c    44  sizeof(FileHandle)
  $000     0  Link
  $004     4  Port
  $008     8  Type
  $00c    12  Buf
  $010    16  Pos
  $014    20  End
  $018    24  Funcs
  $01c    28  Func2
  $020    32  Func3
  $024    36  Args
  $028    40  Arg2

FileInfoBlock
  $104   260  sizeof(FileInfoBlock)
  $000     0  DiskKey
  $004     4  DirEntryType
  $008     8  FileName[0]
  $074   116  Protection
  $078   120  EntryType
  $07c   124  Size
  $080   128  NumBlocks
  $084   132  Date
  $090   144  Comment[0]
  $0e0   224  Reserved[0]

 FleLock
  $014    20  sizeof(FileLock)
  $000     0  Link
  $004     4  Key
  $008     8  Access
  $00c    12  Task
  $010    16  Volume
```

```
 FileSysStartupMsg
  $010    16  sizeof(FileSysStartupMsg)
  $000     0  Unit
  $004     4  Device
  $008     8  Environ
  $00c    12  Flags

 FontContents
  $104   260  sizeof(FontContents)
  $000     0  FileName[0]
  $100   256  YSize
  $102   258  Style
  $103   259  Flags

FontContentsHeader
  $004     4  sizeof(FontContentsHeader)
  $000     0  FileID
  $002     2  NumEntries

 FreeList
  $010    16  sizeof(FreeList)
  $000     0  NumFree
  $002     2  MemList

Gadget
  $02c    44  sizeof(Gadget)
  $000     0  NextGadget
  $004     4  LeftEdge
  $006     6  TopEdge
  $008     8  Width
  $00a    10  Height
  $00c    12  Flags
  $00e    14  Activation
  $010    16  GadgetType
  $012    18  GadgetRender
  $016    22  SelectRender
  $01a    26  GadgetText
  $01e    30  MutualExclude
  $022    34  SpecialInfo
  $026    38  GadgetID
  $028    40  UserData

GamePortTrigger
  $008     8  sizeof(GamePortTrigger)
  $000     0  Keys
  $002     2  Timeout
  $004     4  XDelta
  $006     6  YDelta

GelsInfo
  $026    38  sizeof(GelsInfo)
  $000     0  sprRsrvd
  $001     1  Flags
  $002     2  gelHead
  $006     6  gelTail
  $00a    10  nextLine
  $00e    14  lastColor
  $012    18  collHandler
  $016    22  leftmost
  $018    24  rightmost
  $01a    26  topmost
  $01c    28  bottommost
  $01e    30  firstBlissObj
  $022    34  lastBlissObj

GfxBase
  $148   328  sizeof(GfxBase)
  $000     0  LibNode
  $022    34  ActiView
  $026    38  copinit
  $02a    42  cia
  $02e    46  blitter
  $032    50  LOFlist
  $036    54  SHFlist
  $03a    58  blthd
  $03e    62  blttl
  $042    66  bsblthd
  $046    70  bsblttl
  $04a    74  vbsrv
  $060    96  timsrv
  $076   118  bltsrv
  $08c   140  TextFonts
  $09a   154  DefaultFont
  $09e   158  Modes
```

```
  $0a0   160  VBlank
  $0a1   161  Debug
  $0a2   162  BeamSync
  $0a4   164  system_bplcon0
  $0a6   166  SpriteReserved
  $0a7   167  bytereserved
  $0a8   168  Flags
  $0aa   170  BlitLock
  $0ac   172  BlitNest
  $0ae   174  BlitWaitQ
  $0bc   188  BlitOwner
  $0c0   192  TOF_WaitQ
  $0ce   206  DisplayFlags
  $0d0   208  SimpleSprites
  $0d4   212  MaxDisplayRow
  $0d6   214  MaxDisplayColumn
  $0d8   216  NormalDisplayRows
  $0da   218  NormalDisplayColumns
  $0dc   220  NormalDPMX
  $0de   222  NormalDPMY
  $0e0   224  LastChanceMemory
  $0e4   228  LCMptr
  $0e8   232  MicrosPerLine
  $0ea   234  MinDisplayColumn
  $0ec   236  reserved[0]

IOAudio
  $044    68  sizeof(IOAudio)
  $000     0  Request
  $020    32  AllocKey
  $022    34  Data
  $026    38  Length
  $02a    42  Period
  $02c    44  Volume
  $02e    46  Cycles
  $030    48  WriteMsg

IOClipReq
  $034    52  sizeof(IOClipReq)
  $000     0  Message
  $014    20  Device
  $018    24  Unit
  $01c    28  Command
  $01e    30  Flags
  $01f    31  Error
  $020    32  Actual
  $024    36  Length
  $028    40  Data
  $02c    44  Offset
  $030    48  ClipID


IODRPReq

  $03e    62  sizeof(IODRPReq)
  $000     0  Message
  $014    20  Device
  $018    24  Unit
  $01c    28  Command
  $01e    30  Flags
  $01f    31  Error
  $020    32  RastPort
  $024    36  ColorMap
  $028    40  Modes
  $02c    44  SrcX
  $02e    46  SrcY
  $030    48  SrcWidth
  $032    50  SrcHeight
  $034    52  DestCols
  $038    56  DestRows
  $03c    60  Special

IOExtPar
  $03e    62  sizeof(IOExtPar)
  $000     0  IOPar
  $030    48  PExtFlags
  $034    52  Status
  $035    53  ParFlags
  $036    54  PTermArray

IOExtSer
  $052    82  sizeof(IOExtSer)
  $000     0  IOSer
  $030    48  CtlChar
  $034    52  RBufLen
```

```
  $038    56  ExtFlags
  $03c    60  Baud
  $040    64  BrkTime
  $044    68  TermArray
  $04c    76  ReadLen
  $04d    77  WriteLen
  $04e    78  StopBits
  $04f    79  SerFlags
  $050    80  Status

 IOExtTD
  $038    56  sizeof(IOExtTD)
  $000     0  Req
  $030    48  Count
  $034    52  SecLabel

IOPArray
  $008     8  sizeof(IOPArray)
  $000     0  PTermArray0
  $004     4  PTermArray1

IOPrtCmdReq
  $026    38  sizeof(IOPrtCmdReq)
  $000     0  Message
  $014    20  Device
  $018    24  Unit
  $01c    28  Command
  $01e    30  Flags
  $01f    31  Error
  $020    32  PrtCommand
  $022    34  Parm0
  $023    35  Parm1
  $024    36  Parm2
  $025    37  Parm3

IORequest
  $020    32  sizeof(IORequest)
  $000     0  Message
  $014    20  Device
  $018    24  Unit
  $01c    28  Command
  $01e    30  Flags
  $01f    31  Error

IOStdReq
  $030    48  sizeof(IOStdReq)
  $000     0  Message
  $014    20  Device
  $018    24  Unit
  $01c    28  Command
  $01e    30  Flags
  $01f    31  Error
  $020    32  Actual
  $024    36  Length
  $028    40  Data
  $02c    44  Offset

IOTArray
  $008     8  sizeof(IOTArray)
  $000     0  TermArray0
  $004     4  TermArray1

Image
  $014    20  sizeof(Image)
  $000     0  LeftEdge
  $002     2  TopEdge
  $004     4  Width
  $006     6  Height
  $008     8  Depth
  $00a    10  ImageData
  $00e    14  PlanePick
  $00f    15  PlaneOnOff
  $010    16  NextImage

InfoData
  $024    36  sizeof(InfoData)
  $000     0  id_NumSoftErrors
  $004     4  id_UnitNumber
  $008     8  id_DiskState
  $00c    12  id_NumBlocks
  $010    16  id_NumBlocksUsed
  $014    20  id_BytesPerBlock
  $018    24  id_DiskType
  $01c    28  id_VolumeNode
  $020    32  id_InUse
```

```
InputEvent
  $016    22  sizeof(InputEvent)
  $000     0  NextEvent
  $004     4  Class
  $005     5  SubClass
  $006     6  Code
  $008     8  Qualifier
  $00a    10  position
  $00a    10  position.xy
  $00a    10  position.xy.x
  $00c    12  position.xy.y
  $00a    10  position.addr
  $00e    14  TimeStamp

IntVector
  $00c    12  sizeof(IntVector)
  $000     0  Data
  $004     4  Code
  $008     8  Node

Interrupt
  $016    22  sizeof(Interrupt)
  $000     0  Node
  $00e    14  Data
  $012    18  Code

IntuiMessage
  $034    52  sizeof(IntuiMessage)
  $000     0  ExecMessage
  $014    20  Class
  $018    24  Code
  $01a    26  Qualifier
  $01c    28  IAddress
  $020    32  MouseX
  $022    34  MouseY
  $024    36  Seconds
  $028    40  Micros
  $02c    44  IDCMPWindow
  $030    48  SpecialLink

IntuiText
  $014    20  sizeof(IntuiText)
  $000     0  FrontPen
  $001     1  BackPen
  $002     2  DrawMode
  $004     4  LeftEdge
  $006     6  TopEdge
  $008     8  ITextFont
  $00c    12  IText
  $010    16  NextText

IntuitionBase
  $050    80  sizeof(IntuitionBase)
  $000     0  LibNode
  $022    34  ViewLord
  $034    52  ActiveWindow
  $038    56  ActiveScreen
  $03c    60  FirstScreen
  $040    64  Flags
  $044    68  MouseY
  $046    70  MouseX
  $048    72  Seconds
  $04c    76  Micros

Isrvstr
  $01e    30  sizeof(Isrvstr)
  $000     0  Node
  $00e    14  Iptr
  $012    18  code
  $016    22  ccode
  $01a    26  Carg

KeyMap
  $020    32  sizeof(KeyMap)
  $000     0  LoKeyMapTypes
  $004     4  LoKeyMap
  $008     8  LoCapsable
  $00c    12  LoRepeatable
  $010    16  HiKeyMapTypes
  $014    20  HiKeyMap
  $018    24  HiCapsable
  $01c    28  HiRepeatable

KeyMapNode
```

```
  $02e    46  sizeof(KeyMapNode)
  $000     0  Node
  $00e    14  KeyMap

KeyMapResource
  $01c    28  sizeof(KeyMapResource)
  $000     0  Node
  $00e    14  List

Layer
  $0a0   160  sizeof(Layer)
  $000     0  front
  $004     4  back
  $008     8  ClipRect
  $00c    12  rp
  $010    16  bounds
  $018    24  reserved[0]
  $01c    28  priority
  $01e    30  Flags
  $020    32  SuperBitMap
  $024    36  SuperClipRect
  $028    40  Window
  $02c    44  Scroll_X
  $02e    46  Scroll_Y
  $030    48  cr
  $034    52  cr2
  $038    56  crnew
  $03c    60  SuperSaveClipRects
  $040    64  _cliprects
  $044    68  LayerInfo
  $048    72  Lock
  $076   118  reserved3[0]
  $07e   126  ClipRegion
  $082   130  saveClipRects
  $086   134  reserved2[0]
  $09c   156  DamageList

LayInfo
  $066   102  sizeof(LayerInfo)
  $000     0  top_layer
  $004     4  check_lp
  $008     8  obs
  $00c    12  FreeClipRects
  $018    24  Lock
  $046    70  gs_Head
  $054    84  longreserved
  $058    88  Flags
  $05a    90  fatten_count
  $05b    91  LockLayersCount
  $05c    92  LayerInfo_extra_size
  $05e    94  blitbuff
  $062    98  LayerInfo_extra

Library
  $022    34  sizeof(Library)
  $000     0  Node
  $00e    14  Flags
  $00f    15  pad
  $010    16  NegSize
  $012    18  PosSize
  $014    20  Version
  $016    22  Revision
  $018    24  IdString
  $01c    28  Sum
  $020    32  OpenCnt

List
  $00e    14  sizeof(List)
  $000     0  Head
  $004     4  Tail
  $008     8  TailPred
  $00c    12  Type
  $00d    13  pad

MathIEEEBase
  $03c    60  sizeof(MathIEEEBase)
  $000     0  MathIEEEBase_LibNode
  $022    34  MathIEEEBase_Flags
  $023    35  MathIEEEBase_reserved1
  $024    36  MathIEEEBase_68881
  $028    40  MathIEEEBase_SysLib
  $02c    44  MathIEEEBase_SegList
  $030    48  MathIEEEBase_Resource
  $034    52  MathIEEEBase_TaskOpenLib
  $038    56  MathIEEEBase_TaskCloseLib
```

```
MathIEEEResource
  $02c    44  sizeof(MathIEEEResource)
  $000     0  MathIEEEResource_Node
  $00e    14  MathIEEEResource_Flags
  $010    16  MathIEEEResource_BaseAddr
  $014    20  MathIEEEResource_DblBasInit
  $018    24  MathIEEEResource_DblTransInit
  $01c    28  MathIEEEResource_SglBasInit
  $020    32  MathIEEEResource_SglTransInit
  $024    36  MathIEEEResource_ExtBasInit
  $028    40  MathIEEEResource_ExtTransInit

MemChunk
  $008     8  sizeof(MemChunk)
  $000     0  Next
  $004     4  Bytes

MemEntry
  $008     8  sizeof(MemEntry)
  $000     0  Un
  $000     0  Un.meu_Reqs
  $000     0  Un.meu_Addr
  $004     4  Length

MemHeader
  $020    32  sizeof(MemHeader)
  $000     0  Node
  $00e    14  Attributes
  $010    16  First
  $014    20  Lower
  $018    24  Upper
  $01c    28  Free

MemList
  $018    24  sizeof(MemList)
  $000     0  Node
  $00e    14  NumEntries
  $010    16  ME[0]

Menu
  $01e    30  sizeof(Menu)
  $000     0  NextMenu
  $004     4  LeftEdge
  $006     6  TopEdge
  $008     8  Width
  $00a    10  Height
  $00c    12  Flags
  $00e    14  MenuName
  $012    18  FirstItem
  $016    22  JazzX
  $018    24  JazzY
  $01a    26  BeatX
  $01c    28  BeatY

MenuItem
  $022    34  sizeof(MenuItem)
  $000     0  NextItem
  $004     4  LeftEdge
  $006     6  TopEdge
  $008     8  Width
  $00a    10  Height
  $00c    12  Flags
  $00e    14  MutualExclude
  $012    18  ItemFill
  $016    22  SelectFill
  $01a    26  Command
  $01c    28  SubItem
  $020    32  NextSelect

Message
  $014    20  sizeof(Message)
  $000     0  Node
  $00e    14  ReplyPort
  $012    18  Length

MinList
  $00c    12  sizeof(MinList)
  $000     0  Head
  $004     4  Tail
  $008     8  TailPred

MinNode
  $008     8  sizeof(MinNode)
  $000     0  Succ
```

```
  $004     4  Pred

MiscResource
  $032    50  sizeof(MiscResource)
  $000     0  Library
  $022    34  AllocArray[0]

MsgPort
  $022    34  sizeof(MsgPort)
  $000     0  Node
  $00e    14  Flags
  $00f    15  SigBit
  $010    16  SigTask
  $014    20  MsgList

narrator_rb
  $046    70  sizeof(narrator_rb)
  $000     0  message
  $030    48  rate
  $032    50  pitch
  $034    52  mode
  $036    54  sex
  $038    56  ch_masks
  $03c    60  nm_masks
  $03e    62  volume
  $040    64  sampfreq
  $042    66  mouths
  $043    67  chanmask
  $044    68  numchan
  $045    69  pad

NewScreen
  $020    32  sizeof(NewScreen)
  $000     0  LeftEdge
  $002     2  TopEdge
  $004     4  Width
  $006     6  Height
  $008     8  Depth
  $00a    10  DetailPen
  $00b    11  BlockPen
  $00c    12  ViewModes
  $00e    14  Type
  $010    16  Font
  $014    20  DefaultTitle
  $018    24  Gadgets
  $01c    28  CustomBitMap

NewWindow
  $030    48  sizeof(NewWindow)
  $000     0  LeftEdge
  $002     2  TopEdge
  $004     4  Width
  $006     6  Height
  $008     8  DetailPen
  $009     9  BlockPen
  $00a    10  IDCMPFlags
  $00e    14  Flags
  $012    18  FirstGadget
  $016    22  CheckMark
  $01a    26  Title
  $01e    30  Screen
  $022    34  BitMap
  $026    38  MinWidth
  $028    40  MinHeight
  $02a    42  MaxWidth
  $02c    44  MaxHeight
  $02e    46  Type

Node
  $00e    14  sizeof(Node)
  $000     0  Succ
  $004     4  Pred
  $008     8  Type
  $009     9  Pri
  $00a    10  Name

Preferences
  $0e8   232  sizeof(Preferences)
  $000     0  FontHeight
  $001     1  PrinterPort
  $002     2  BaudRate
  $004     4  KeyRptSpeed
  $00c    12  KeyRptDelay
  $014    20  DoubleClick
  $01c    28  PointerMatrix[0]
```

```
  $064   100  XOffset
  $065   101  YOffset
  $066   102  color17
  $068   104  color18
  $06a   106  color19
  $06c   108  PointerTicks
  $06e   110  color0
  $070   112  color1
  $072   114  color2
  $074   116  color3
  $076   118  ViewXOffset
  $077   119  ViewYOffset
  $078   120  ViewInitX
  $07a   122  ViewInitY
  $07c   124  EnableCLI
  $07e   126  PrinterType
  $080   128  PrinterFilename[0]
  $09e   158  PrintPitch
  $0a0   160  PrintQuality
  $0a2   162  PrintSpacing
  $0a4   164  PrintLeftMargin
  $0a6   166  PrintRightMargin
  $0a8   168  PrintImage
  $0aa   170  PrintAspect
  $0ac   172  PrintShade
  $0ae   174  PrintThreshold
  $0b0   176  PaperSize
  $0b2   178  PaperLength
  $0b4   180  PaperType
  $0b6   182  SerRWBits
  $0b7   183  SerStopBuf
  $0b8   184  SerParShk
  $0b9   185  LaceWB
  $0ba   186  WorkName[0]
  $0d8   216  RowSizeChange
  $0d9   217  ColumnSizeChange
  $0da   218  PrintFlags
  $0dc   220  PrintMaxWidth
  $0de   222  PrintMaxHeight
  $0e0   224  PrintDensity
  $0e1   225  PrintXOffset
  $0e2   226  wb_Width
  $0e4   228  wb_Height
  $0e6   230  wb_Depth
  $0e7   231  ext_size

PrinterData
  $aa2  2722  sizeof(PrinterData)
  $000     0  Device
  $034    52  Unit
  $056    86  PrinterSegment
  $05a    90  PrinterType
  $05c    92  SegmentData
  $060    96  PrintBuf
  $064   100  PWrite
  $068   104  PBothReady
  $06c   108  ior0
  $06c   108  ior0.p0
  $06c   108  ior0.s0
  $0be   190  ior1
  $0be   190  ior1.p1
  $0be   190  ior1.s1
  $110   272  TIOR
  $138   312  IORPort
  $15a   346  TC
  $1b6   438  Stk[0]
  $9b6  2486  Flags
  $9b7  2487  pad
  $9b8  2488  Preferences
  $aa0  2720  PWaitEnabled

PrinterExtendedData
  $042    66  sizeof(PrinterExtendedData)
  $000     0  PrinterName
  $004     4  Init
  $008     8  Expunge
  $00c    12  Open
  $010    16  Close
  $014    20  PrinterClass
  $015    21  ColorClass
  $016    22  MaxColumns
  $017    23  NumCharSets
  $018    24  NumRows
  $01a    26  MaxXDots
  $01e    30  MaxYDots
```

```
  $022    34  XDotsInch
  $024    36  YDotsInch
  $026    38  Commands
  $02a    42  DoSpecial
  $02e    46  Render
  $032    50  TimeoutSecs
  $036    54  8BitChars
  $03a    58  PrintMode
  $03e    62  ConvFunc

PrinterSegment
  $04e    78  sizeof(PrinterSegment)
  $000     0  NextSegment
  $004     4  runAlert
  $008     8  Version
  $00a    10  Revision
  $00c    12  PED

Process
  $0bc   188  sizeof(Process)
  $000     0  Task
  $05c    92  MsgPort
  $07e   126  Pad
  $080   128  SegList
  $084   132  StackSize
  $088   136  GlobVec
  $08c   140  TaskNum
  $090   144  StackBase
  $094   148  Result2
  $098   152  CurrentDir
  $09c   156  CIS
  $0a0   160  COS
  $0a4   164  ConsoleTask
  $0a8   168  FileSystemTask
  $0ac   172  CLI
  $0b0   176  ReturnAddr
  $0b4   180  PktWait
  $0b8   184  WindowPtr

PropInfo
  $016    22  sizeof(PropInfo)
  $000     0  Flags
  $002     2  HorizPot
  $004     4  VertPot
  $006     6  HorizBody
  $008     8  VertBody
  $00a    10  CWidth
  $00c    12  CHeight
  $00e    14  HPotRes
  $010    16  VPotRes
  $012    18  LeftBorder
  $014    20  TopBorder

PrtInfo
  $072   114  sizeof(PrtInfo)
  $000     0  render
  $004     4  rp
  $008     8  temprp
  $00c    12  RowBuf
  $010    16  HamBuf
  $014    20  ColorMap
  $018    24  ColorInt
  $01c    28  HamInt
  $020    32  Dest1Int
  $024    36  Dest2Int
  $028    40  ScaleX
  $02c    44  ScaleXAlt
  $030    48  dmatrix
  $034    52  TopBuf
  $038    56  BotBuf
  $03c    60  RowBufSize
  $03e    62  HamBufSize
  $040    64  ColorMapSize
  $042    66  ColorIntSize
  $044    68  HamIntSize
  $046    70  Dest1IntSize
  $048    72  Dest2IntSize
  $04a    74  ScaleXSize
  $04c    76  ScaleXAltSize
  $04e    78  PrefsFlags
  $050    80  special
  $054    84  xstart
  $056    86  ystart
  $058    88  width
  $05a    90  height
```

```
  $05c    92  pc
  $060    96  pr
  $064   100  ymult
  $066   102  ymod
  $068   104  ety
  $06a   106  xpos
  $06c   108  threshold
  $06e   110  tempwidth
  $070   112  flags

RasInfo
  $00c    12  sizeof(RasInfo)
  $000     0  Next
  $004     4  BitMap
  $008     8  RxOffset
  $00a    10  RyOffset

RastPort
  $064   100  sizeof(RastPort)
  $000     0  Layer
  $004     4  BitMap
  $008     8  AreaPtrn
  $00c    12  TmpRas
  $010    16  AreaInfo
  $014    20  GelsInfo
  $018    24  Mask
  $019    25  FgPen
  $01a    26  BgPen
  $01b    27  AOlPen
  $01c    28  DrawMode
  $01d    29  AreaPtSz
  $01e    30  linpatcnt
  $01f    31  dummy
  $020    32  Flags
  $022    34  LinePtrn
  $024    36  cp_x
  $026    38  cp_y
  $028    40  minterms[0]
  $030    48  PenWidth
  $032    50  PenHeight
  $034    52  Font
  $038    56  AlgoStyle
  $039    57  TxFlags
  $03a    58  TxHeight
  $03c    60  TxWidth
  $03e    62  TxBaseline
  $040    64  TxSpacing
  $042    66  RP_User
  $046    70  longreserved[0]
  $04e    78  wordreserved[0]
  $05c    92  reserved[0]

Rectangle
  $008     8  sizeof(Rectangle)
  $000     0  MinX
  $002     2  MinY
  $004     4  MaxX
  $006     6  MaxY

Region
  $00c    12  sizeof(Region)
  $000     0  bounds
  $008     8  RegionRectangle

RegionRectangle
  $010    16  sizeof(RegionRectangle)
  $000     0  Next
  $004     4  Prev
  $008     8  bounds

Remember
  $00c    12  sizeof(Remember)
  $000     0  NextRemember
  $004     4  RememberSize
  $008     8  Memory

Requester
  $070   112  sizeof(Requester)
  $000     0  OlderRequest
  $004     4  LeftEdge
  $006     6  TopEdge
  $008     8  Width
  $00a    10  Height
  $00c    12  RelLeft
  $00e    14  RelTop
```

```
  $010    16  ReqGadget
  $014    20  ReqBorder
  $018    24  ReqText
  $01c    28  Flags
  $01e    30  BackFill
  $020    32  ReqLayer
  $024    36  ReqPad1[0]
  $044    68  ImageBMap
  $048    72  RWindow
  $04c    76  ReqPad2[0]

Resident
  $01a    26  sizeof(Resident)
  $000     0  MatchWord
  $002     2  MatchTag
  $006     6  EndSkip
  $00a    10  Flags
  $00b    11  Version
  $00c    12  Type
  $00d    13  Pri
  $00e    14  Name
  $012    18  IdString
  $016    22  Init

RomBootBase
  $044    68  sizeof(RomBootBase)
  $000     0  LibNode
  $022    34  ExecBase
  $026    38  BootList
  $034    52  Reserved[0]

RootNode
  $020    32  sizeof(RootNode)
  $000     0  TaskArray
  $004     4  ConsoleSegment
  $008     8  Time
  $014    20  RestartSeg
  $018    24  Info
  $01c    28  FileHandlerSegment

Screen
  $15a   346  sizeof(Screen)
  $000     0  NextScreen
  $004     4  FirstWindow
  $008     8  LeftEdge
  $00a    10  TopEdge
  $00c    12  Width
  $00e    14  Height
  $010    16  MouseY
  $012    18  MouseX
  $014    20  Flags
  $016    22  Title
  $01a    26  DefaultTitle
  $01e    30  BarHeight
  $01f    31  BarVBorder
  $020    32  BarHBorder
  $021    33  MenuVBorder
  $022    34  MenuHBorder
  $023    35  WBorTop
  $024    36  WBorLeft
  $025    37  WBorRight
  $026    38  WBorBottom
  $028    40  Font
  $02c    44  ViewPort
  $054    84  RastPort
  $0b8   184  BitMap
  $0e0   224  LayerInfo
  $146   326  FirstGadget
  $14a   330  DetailPen
  $14b   331  BlockPen
  $14c   332  SaveColor0
  $14e   334  BarLayer
  $152   338  ExtData
  $156   342  UserData

Semaphore
  $024    36  sizeof(Semaphore)
  $000     0  MsgPort
  $022    34  Bids

SemaphoreRequest
  $00c    12  sizeof(SemaphoreRequest)
  $000     0  Link
  $008     8  Waiter
```

```
SignalSemaphore
  $02e    46  sizeof(SignalSemaphore)
  $000     0  Link
  $00e    14  NestCount
  $010    16  WaitQueue
  $01c    28  MultipleLink
  $028    40  Owner
  $02c    44  QueueCount

SimpleSprite
  $00c    12  sizeof(SimpleSprite)
  $000     0  posctldata
  $004     4  height
  $006     6  x
  $008     8  y
  $00a    10  num

SoftIntList
  $010    16  sizeof(SoftIntList)
  $000     0  List
  $00e    14  Pad

SpriteDef
  $008     8  sizeof(SpriteDef)
  $000     0  pos
  $002     2  ctl
  $004     4  dataa
  $006     6  datab

StandardPacket
  $044    68  sizeof(StandardPacket)
  $000     0  Msg
  $014    20  Pkt

StringInfo
  $024    36  sizeof(StringInfo)
  $000     0  Buffer
  $004     4  UndoBuffer
  $008     8  BufferPos
  $00a    10  MaxChars
  $00c    12  DispPos
  $00e    14  UndoPos
  $010    16  NumChars
  $012    18  DispCount
  $014    20  CLeft
  $016    22  CTop
  $018    24  LayerPtr
  $01c    28  LongInt
  $020    32  AltKeyMap

TDU_PublicUnit
  $036    54  sizeof(TDU_PublicUnit)
  $000     0  Unit
  $026    38  Comp01Track
  $028    40  Comp10Track
  $02a    42  Comp11Track
  $02c    44  StepDelay
  $030    48  SettleDelay
  $034    52  RetryCnt

Task
  $05c    92  sizeof(Task)
  $000     0  Node
  $00e    14  Flags
  $00f    15  State
  $010    16  IDNestCnt
  $011    17  TDNestCnt
  $012    18  SigAlloc
  $016    22  SigWait
  $01a    26  SigRecvd
  $01e    30  SigExcept
  $022    34  TrapAlloc
  $024    36  TrapAble
  $026    38  ExceptData
  $02a    42  ExceptCode
  $02e    46  TrapData
  $032    50  TrapCode
  $036    54  SPReg
  $03a    58  SPLower
  $03e    62  SPUpper
  $042    66  Switch
  $046    70  Launch
  $04a    74  MemEntry
  $058    88  UserData
```

```
TextAttr
  $008     8  sizeof(TextAttr)
  $000     0  Name
  $004     4  YSize
  $006     6  Style
  $007     7  Flags

TextFont
  $034    52  sizeof(TextFont)
  $000     0  Message
  $014    20  YSize
  $016    22  Style
  $017    23  Flags
  $018    24  XSize
  $01a    26  Baseline
  $01c    28  BoldSmear
  $01e    30  Accessors
  $020    32  LoChar
  $021    33  HiChar
  $022    34  CharData
  $026    38  Modulo
  $028    40  CharLoc
  $02c    44  CharSpace
  $030    48  CharKern

TmpRas
  $008     8  sizeof(TmpRas)
  $000     0  RasPtr
  $004     4  Size

UCopList
  $00c    12  sizeof(UCopList)
  $000     0  Next
  $004     4  FirstCopList
  $008     8  CopList

Unit
  $026    38  sizeof(Unit)
  $000     0  MsgPort
  $022    34  flags
  $023    35  pad
  $024    36  OpenCnt

VSprite
  $03c    60  sizeof(VSprite)
  $000     0  NextVSprite
  $004     4  PrevVSprite
  $008     8  DrawPath
  $00c    12  ClearPath
  $010    16  OldY
  $012    18  OldX
  $014    20  Flags
  $016    22  Y
  $018    24  X
  $01a    26  Height
  $01c    28  Width
  $01e    30  Depth
  $020    32  MeMask
  $022    34  HitMask
  $024    36  ImageData
  $028    40  BorderLine
  $02c    44  CollMask
  $030    48  SprColors
  $034    52  VSBob
  $038    56  PlanePick
  $039    57  PlaneOnOff
  $03a    58  VUserExt

View
  $012    18  sizeof(View)
  $000     0  ViewPort
  $004     4  LOFCprList
  $008     8  SHFCprList
  $00c    12  DyOffset
  $00e    14  DxOffset
  $010    16  Modes

ViewPort
  $028    40  sizeof(ViewPort)
  $000     0  Next
  $004     4  ColorMap
  $008     8  DspIns
  $00c    12  SprIns
  $010    16  ClrIns
  $014    20  UCopIns
```

```
  $018    24  DWidth
  $01a    26  DHeight
  $01c    28  DxOffset
  $01e    30  DyOffset
  $020    32  Modes
  $022    34  SpritePriorities
  $023    35  reserved
  $024    36  RasInfo

WBArg
  $008     8  sizeof(WBArg)
  $000     0  Lock
  $004     4  Name

WBStartup
  $028    40  sizeof(WBStartup)
  $000     0  Message
  $014    20  Process
  $018    24  Segment
  $01c    28  NumArgs
  $020    32  ToolWindow
  $024    36  ArgList

Window
  $084   132  sizeof(Window)
  $000     0  NextWindow
  $004     4  LeftEdge
  $006     6  TopEdge
  $008     8  Width
  $00a    10  Height
  $00c    12  MouseY
  $00e    14  MouseX
  $010    16  MinWidth
  $012    18  MinHeight
  $014    20  MaxWidth
  $016    22  MaxHeight
  $018    24  Flags
  $01c    28  MenuStrip
  $020    32  Title
  $024    36  FirstRequest
  $028    40  DMRequest
  $02c    44  ReqCount
  $02e    46  WScreen
  $032    50  RPort
  $036    54  BorderLeft
  $037    55  BorderTop
  $038    56  BorderRight
  $039    57  BorderBottom
  $03a    58  BorderRPort
  $03e    62  FirstGadget
  $042    66  Parent
  $046    70  Descendant
  $04a    74  Pointer
  $04e    78  PtrHeight
  $04f    79  PtrWidth
  $050    80  XOffset
  $051    81  YOffset
  $052    82  IDCMPFlags
  $056    86  UserPort
  $05a    90  WindowPort
  $05e    94  MessageKey
  $062    98  DetailPen
  $063    99  BlockPen
  $064   100  CheckMark
  $068   104  ScreenTitle
  $06c   108  GZZMouseX
  $06e   110  GZZMouseY
  $070   112  GZZWidth
  $072   114  GZZHeight
  $074   116  ExtData
  $078   120  UserData
  $07c   124  WLayer
  $080   128  IFont
```

Glossar

.info
: Immer wenn eine Datei oder ein Verzeichnis auf der Workbench erscheinen soll, muss ein `.info`-Datei vorhanden sein. In diesen `.info`-Dateien sind die Ausmaße und die Grafikdaten des →Icons gesichert. Beim →OS 2.0 brauchen keine Icons mehr erstellt werden, damit sie auf der Workbench erscheinen.

68000
: Unter der Bezeichnung 680x0 wird eine Prozessorreihe der Motorola-Familie geführt, die in Amiga-Rechnern und anderen

Computern, wie Apple, Next, Atari eingesetzt werden. Der MC68000 ist der Nachfolger des 8-Bit Prozessors 6800. In seiner Form ist er veraltet, und wird daher zwar noch eingesetzt, aber es wird keine Computerhardware mehr damit entwickelt. Nachfolger sind die →Prozessoren 68008, 68012, 68020, 68030, 68040, 68060. Ab dem 68030 wird die Reihe interessant. Der MC68000 ist ein typischer CISC-Prozessor.

A-Trap
Der A-Trap ist einer von zwei →Trap-Möglichkeiten. Der Macintosh verwendet sie zum Aufruf der Betriebssystemaufrufe.

Absturz
Manchmal kann es vorkommen, dass der Amiga so durcheinandergerät, dass ein Weiterarbeiten unmöglich ist. Um dann wieder mit ihm arbeiten zu können, muss ein →Reset ausgelöst werden. Ein Absturz macht sich im Regelfall durch eine →Guru-Meldung bemerkbar. Auch ein Task-Finisch-Held ist ein Zeichen für einen Absturz, ihm folgt dann normalerweise ein Guru.

Adresse
Eine Adresse ist eine eindeutige Kennzeichnung einer Speicherzelle, an der ein Speicher- →Byte zu finden ist. Eine Adresse ist immer im Bereich von Null bis zum Ende des Speicherbereiches.

Adressierungsarten
Um an den Speicherinhalt einer →Variablen zu gelangen, muss der Programmierer die Adresse angeben, an der die →Variable liegt. Diese absolute Adresse (physikalische Adresse genannt) kann nun auf verschiedene Weisen berechnet werden, die wir Adressierungsarten nennen. Der MC68000 kennt 16 verschiedene Adressierungsarten. Die wichtigsten Adr. Arten sind absolute Adressierung und indirekte Adressierung.

Adressbus
Der Adressbus ist direkt mit dem →Speicher verbunden, und spricht die Speicherzellen an. Der Adressbus des MC -68000 ist 24 Leitungen breit (Adressleitungen), die Nachfolger ab dem MC68020 haben 32 Adressleitungen.

Agnus
Einer der drei wichtigsten Zusatzprozessoren, ein →Custom-Chip. Agnus ist nicht nur für eine Aufgabe zuständig, er enthält den →Blitter, den →Copper, und die →DMA-Einheit. Man darf nicht denken, und das wird gerne gemacht, das der Blitter oder der Copper eigenständige IC's sind, sie sind vielmehr in einem einzigen →IC untergebracht.

Aiken-Code
Der Aiken-Code ist ein spezieller Code, in dem Dezimalzahlen von 0-9 einem bestimmten Bitwert (0000 bis 1111) zugeordnet werden. Beim Amiga OS in keiner Weise unterstützt.

Akkumulator
Der Akkumulator ist ein Register, in dem fast alle Operationen ablaufen. Es gibt also keine gleichberechtigten Register wie beim MC68000 die Register `D0-D7`, sondern nur ein einziges. Der Nachteil dieser Akkumulatormaschinen: er muss häufig gesichert werden, die Abarbeitung wird langsam. Aus diesem Grunde haben neue Prozessoren immer mehrere Register, die alle die gleichen Fähigkeiten wie ein Akkumulator haben. →RISC-Prozessoren haben oft mehr als 16 Register.

Algorithmus
Ein Algorithmus ist eine Abarbeitungsvorschrift, die so formuliert ist, dass ein Compiler oder ein Prozessor das dadurch aufgebaute Programm verarbeiten kann.

AmigaDOS
Das Amiga Disk-Operating-System ist die Einheit, die für die Verwaltung der Speichermedien zuständig ist. AmigaDOS greift auf das `trackdisk.` → `device` zurück, das die grundlegendsten Funktionen bietet. AmigaDOS ist vergleichbar mit MS DOS.

ANSI
Was ANSI (American National Standart Institute) für die Amerikaner ist, ist DIN (Deutsche Industrienorm) für uns. Neuerdings unterliegt alles dem ANSI Standart, auch der C Compiler muss heute schon ANSI sein, damit ein Programmaustausch möglich wird, und die Übertragung nicht an irgendwelchen compiler-spezifischen Kleinigkeiten scheitert.

APTR
Bevor die Programmiersprache →C in die Geschichte einging, wurde speziell am AmigaDOS viel mit BCPL gearbeitet, einem Vorgänger von C. Die Vorgängersprache unterscheidet sich in einigen Punkten von C, so in den Zeigern, der APTR genannt wird.

ASCII
Der ASCII (American Standart Code for Information Interchange) Code ist ein weit verbreiteter 7-Bit Kode, der ursprünglich von Fernschreibern verwendet wurde. Dann zog er in den Heimcomputerbereich ein, und wird dort immer noch verwendet. Da die Anzahl der Zeichen, 127 durch 7 Bit darstellbar, nicht ausreichte, wurden 8 Bit verwendet, man spricht dann vom erweiterten ASCII Code. Leider ist dieser nicht genormt, und bei Übertragung von Texten sieht man meist bei Umlauten, dass der Code nicht derselbe ist.

Attribut-Byte
Das Attribut-Byte ist ein →Byte, das Informationen über verschiedene Attribute einer Datei enthält. Verschiedene Attribute werden von →AmigaDOS bereitgestellt, aber nicht immer unterstützt. Beispiele: Verstecken der Datei oder sie vor dem Löschen schützen.

Auflösung
Die Auflösung, eng. Resolution, ist die Anzahl der horizontalen und vertikalen Punkte des Bildschirms. Die Auflösung eines →NTSC →HiRes- Bildes ist 640 * 256 Punkte. Diese Auflösung ist aber nicht unbedingt hoch, CAD-Schirme arbeiten mit 1280 * 1024 Punkten, oder sogar noch höher. Allgemein: je höher die Anzahl der Punkte ist, desto höher ist die Auflösung.

Aufwärtskompatibel
Programme sind aufwärtskompatibel, wenn alte Software auch auf neuen Rechnermodellen läuft. Leider ist das bei Amiga Software nicht so toll mit der Aufwärtskompatibilität. Als ich von 1.3 auf 2.0 umstieg, lief ca. 2/3 meiner Software nicht mehr (Musikprogramme, Grafikprogramme, Demos).

Assembler
Ist die Programmiersprache, die wir nach dem Lesen dieses Buches beherrschen sollten.

Assembler
Ein Programm, das die →Mnemoniks in Maschinensprache Opcode übersetzt.

Attach-Mode
Ist die Bezeichnung für die Möglichkeit durch zwei →DMA-Kanle die Anzahl der Farben von Sprites zu erhöhen.

Austastlücke
Siehe →Blanking-Intervall.

Autoboot
: Wenn Erweiterungskarten sich selbst konfigurieren, oder wenn Festplatten automatisch starten, spricht man von einem Autoboot-Vorgang.

b-Baum
: b-Baum steht für binärer →Baum.

Bandbreite
: Um die Pixel, die für ein Bild notwendig sind, zu übertragen, muss eine bestimmte Frequenz vorhanden sein. Je höher die →Auflösung, desto höher die Bandbreite.

Barrel-Shifer
: In einem →Prozessor ist der Barrel-Shifter eine Aufgabeneinheit, die die →Bits an gewünschte Stellen schiebt. Erst höhere Prozessoren und →RISC-CPU's haben diese Einheit. Die →Blitter-Einheit ist ebenfalls mit einem Barrel-Shifter ausgestattet, um an jeder Bitposition Bilder abzulegen.

Basisadresse
: Beim Öffnen von Libraries erhalten wir als Return-Wert die Basisadresse. Sie sind notwendig für die →Betriebssystemaufrufe. Die Basisadresse sollte immer im Adress-→Register A6 stehen. Exec nimmt eine Sonderstellung ein, siehe dazu SysBase.

Batchprogramm
: Eine Batchdatei (Datei mit einem Batchprogramm) ist eine Textdatei, in der die Schritte abgelegt sind, die normalerweise manuell eingetippt würden. Diese Textdate kann ausgeführt werden, und das →CLI interpretiert die Zeichenfolgen.

Baud
: Ist die Einheit für die Änderungsrate eines Datensignals pro Sekunde. Oft auch nur Einheit für die Bits, die den seriellen Port in einer Sekunde verlassen.

Baum
: Ein Baum ist nicht nur für das ökologische Gleichgewicht auf der Erde wichtig, die Informatik könnte auch nicht ohne Bäume leben. Bäume sind dynamische Datenstrukturen, die aus Knoten und Blättern bestehen. Am Wichtigsten sind die →binären Bäume, die immer einen rechten und linken Knoten haben.

BCD
: Ein Code (Binary Codes Dezimal) zur Darstellung von Binärzahlen, der nicht zu verwechseln ist mit Aiken-Code. Jeder Zahl 0-9 wird ein 4-Bit Wert (→Nibble) zugeordnet.

Betriebssystem
: Der Begriff für alle Benutzerprogramme, die den Benutzer das Arbeiten mit dem Rechner möglich macht. Da mir aber das Wort zu lang ist, schreibe ich fast immer OS (Operating System), was das selbe ist. Selten verwendet man auch das Wort Systemsoftware. Beispiele für Betriebssysteme sind System 7 (Apple); MS-DOS und DR-DOS, Windows (NT) für PC's; UNIX, OS für gehobenere PC's oder Workstations; VM, VS 2000 für Großrechner.

Betriebssystemaufrufe
: Um Funktionen des →Betriebssystems nutzen zu können, muss eine Library oder ein Device angesprochen werden. Die Aufrufe, die die Libraries oder Devices über interne Sprungtabellen regeln, werden Betriebssystemaufrufe genannt. Sie sind immer mit einem negativen Offset verbunden, der letztendlich als Konstante den Offset-Tabellen zu entnehmen ist.

Bidirektional
: Eine Datenübertragung ist bidirektional, wenn die Signale in beiden Richtungen übertragen werden können. Das Gegenteil von unidirektional.

Binär
: Ein Zustand heißt binär, wenn er nur zwei Zustände annehmen kann. Die Zustände sind entweder gesetzt oder nicht gesetzt.

Bit
: Die kleinste Darstellungseinheit für Daten ist Bit. Da ein Bit die Werte Null oder Eins darstellen kann, können 2 Informationen gespeichert werden. Die Basis des binären Zahlensystems ist daher zwei.

Bitmap
: Ein Bild, das aus →Pixeln aufbaut wird, wird Bitmap genannt. Ein →Bit im Speicher entspricht einem Pixel auf dem Bildschirm. Die Struktur `Bitmap` enthält die →Bitplanes und weitere Informationen über die Speicherung.

Bitplane
: Eine Bitplane ist ein rechteckiger Speicherbereich, der zur Darstellung der Grafik verwendet wird. Wird eine Bitplane hinzugefügt, erhöht die Anzahl der darstellbaren Farben um das doppelte.

Blanking-Intervall
: Ist der Zeitraum, indem der →Rasterstahl keinen Grafikbereich darstellt. Es ist dementsprechend die obere und unterer Bildschirmhälfte, und die Ränder. Neben den Begriff Blanking-Intervall verwendet man die Wörter horizontale- und vertikale →Austastlücke. Diese Austastlücke ist der nicht dargestellte Bildschirm.

Blitter
: Ist den Hardware-Programmierern liebstes Spielzeug. Er ist ein →Coprozessor im Amiga, der Daten kopieren und verknüpfen, Linien zeichnen und Flächen füllen kann. Doch ein Blitter ist nichts besonderes, auch andere Rechner haben einen Blitter-Chip (Atari, viele Grafikkarten), aber dieser kann nur das klassische Kopieren und Verknüpfen, die eigentliche Aufgabe, die sich aus dem Wort Blitter — Block Transfer — ergibt. Um seine Aufgabe schnell durchzuführen, muss er Zugriff zum Speicher haben. Daher begann er einen eigenen →DMA-Kanal, um schnell auf den →Chip-Mem zuzugreifen.

Blitting
: Unter blitting versteht man einen Blockkopiervorgang. Blitting wird dann vorgenommen, wenn z. B. ein Menü auf den Schirm erscheint. Der Hintergrund muss gesichert werden, damit später, nach dem verschwinden des Menüs, der alte Hintergrund wieder hergestellt werden kann. Das Kopieren wird durch den →Blitter vorgenommen, der die rechteckige Bereiche schnell vom Bildschirm in einen freien Bereich verschiebt.

Bus
: Ein Bus ist eine Sammelleitung für Funktionseinheiten, die im inneren des Rechners Daten austauschen.

Bus-Zyklus
: Ein Buszyklus ist eine komplette Aktion auf dem →Bus.

Booten
: Beim Starten des Amigas wird ein →ROM-Programm ausgeführt, das den Rechner und seine →Strukturen initialisiert.

Bootsektoren

Die ersten Blöcke (0 und 1) auf der Diskette sind die Bootsektoren, auf dem sich der Diskettentyp und eine Checksumme befindet. Alle Amiga Disketten werden darüber hinaus mit einem kleinen Boot-Programm formatiert, das dann beim →booten abgearbeitet wird. Das Programm kann natürlich selber erstellt werden, eine Gefahr von Boot-Viren droht. Neben dem Begriff Bootsektoren, die speziell die Anfangssektoren betonen, dürfte der Begriff Boot-Block häufiger vorkommen.

Brückenkarte

Der Amiga bietet durch sein offenes System die phantastische Möglichkeit, Steckkarten aller Art anzuschließen. Eine besondere Karte ist die Brückenkarte, die den Amiga um einen PC-Teil ergnzt. Dieses Konzept wird auch Januskonzept genannt. Auf einer Brückenkarte finden wir nicht mehr als einen Hauptprozessor und ein paar Controller, alles andere, was die PC-Karte benötigt, wird vom Amiga System her verwendet.

Byte

Eine Zusammenfassung von 8 →Bits wird Byte genannt. Sie ist die kleinste ansprechbare Speicher-Einheit. Da ein Byte immer noch ziemlich klein ist, setzt man Größenangaben wie KB (1024, nicht 1000, sonst wäre es kB) vor.

C

C ist wohl die wichtigste Programmiersprache unserer Zeit. Sie wurde Ende der sechziger Jahre von Kerighan und Ritchie entworfen. Die Bedeutung von C liegt in der Portierbarkeit von Programmen. UNIX ist ein →Betriebssystem, das größtenteils in C programmiert wurde.

Chip-RAM

Der gesamte Speicher ist in Chip-RAM und dem Fast-RAM aufgeteilt. (Vom Zwitter-Speicher sehen wir mal ab, denn ist halt keins von beidem.) Jeder der beiden Speicherarten hat verschiedene Aufgabengebiete, so ist der Chip-RAM den →Coprozessoren vorbehalten, die dort ihre Grafik- oder Musikdaten bearbeiten können. Wichtig ist, dass sie nur im Chip-RAM werkeln können, und nirgendwo anders. Das Chip-RAM sollte also nur für Daten sein, und nicht für Programme.

CIA

Ein CIA (Complex Interface Adapter) ist ein komplexer Schnittstellenbaustein, der im System für viele Gebiete verwendet wird. Seine Hauptaufgabe ist das Timing. Mit ihm kann man Uhrenfunktionen und Zeitschleifen programmieren. Er kann →Interrupts auslösen, quasi wie ein Wecker, wenn ein bestimmter Zeitpunkt erreicht ist.

CISC

Ein CISC →Prozessor ist eine spezielle Prozessorart, die in den 70iger Jahren oft von Prozessorbauern verwendet wurde. CISC ist eine Technik, nicht jeden Befehl sofort durch eine Art sture Verkabelung auszuführen. Vielmehr werden die Befehle in Mikrocode zerlegt, der dann Ausgeführt wird. Im Gegensatz zu CISC steht RISC→.

CLI

Ist die Abkürzung für „Command Line Interpreter“ (oder auch „Command Line Interface“). Dieses Programm interpretiert die Kommando-Zeilen, wie die Übersetzung des englischen Ausdruckes in Kommando Zeilen Interpreter schon sagt. Die Kommando-Zeilen bestehen aus →AMIGA-DOS-Befehlen. Während die grafische Oberfläche →Workbench alles über →Icons, →Menüs und →Fenstern erledigt, muss im →CLI mit der Tastatur gearbeitet werden. Das CLI gehört mit zu den drei internen built-in unser inferfaces. Eine Erweiterung des CLIs ist die →Shell. Dies kann man aber oft nicht auseinanderhalten, da die Begriffe parallel benutzt werden.

Clipping

„Clip“ heißt abschneiden, und beschreibt einen Vorgang, in den nicht dargestellte Bildschirmteile abgeschnitten werden. Clipping muss immer dann durchgeführt werden, wenn sich überlagernde Windows in die Quere kommen, oder wenn das Fenster zu Ende ist, aber die Zeichenoperation darüber hinaus gehen.

Color Cycling

Viele Präsentationen oder Demos nutzen in Interrupts die Möglichkeit, die Farben der →Farbregister zu vertauschen. Das Farbregister Nr. x wird mit dem Wert Farbregister x+1 geladen, dies geht dann so weiter.

Color-Descriptor-Words

Unter diesem Begriff versteht man das Word-Paar, welches eine Linie des Sprites verschlüsselt.

Color-Register

Siehe →Farbregister.

Composite Signal

Wenn das Video-Signal die Bildinformation und die Synchronisations-Signale enthält spricht man von einem Composite-Signal. Das gesamte Signal wird von Cunmputer zum Darstellungsgerät über ein einziges Coax-Kabel übertragen. Im Gegensatz dazu stehen Chinc-Stecker, bei dem alle Informationen auf getrennten Leitungen übergeben werden. Dadurch sind höhere Bandbreiten möglich.

Controller

Ein Controller ist eine Zwischeneinheit, um Daten von einem Eingabemedium (Maus, Tastatur, Festplatte, Joystick) zum Rechner kommen zu lassen.

Copper

Abkürzung für COP(P)rozEssoR, der die Grafikdarstellung übernimmt. Er ist über das System synchronisiert, und erlaubt dem Programmierer Programme in einer einfachen Sprache zu verfassen, die drei leistungsfähige Befehle zählt. Der Copper ist so gesehen der einzige wahre →Coprozessor, denn nur er kann neben dem →Hauptprozessor eigene Programme unabhängig abarbeiten.

Copper-Liste

Eine Art Programm, welches der Copper verarbeitet, wird in einer Copper-Liste abgelegt. Die Programme können aus drei Befehlen geformt werden, die der Copper dann sequenziell abarbeitet, daher auch das Wort Liste. Die Befehle haben Einfluss auf die Custom-Chips, und können Registerwerte verändern. Ein Copper-Programm endet gewöhnlich mit -2.

Coprozessor

Um den Rechner bei seiner Arbeit zu unterstürzen bringt man gerne Koprozessoren ins System ein. Sie sind spezielle Einheiten, die oft nur für besondere Arbeiten geeignet sind. Als Beispiels werden immer Matheprozessor und Grafikprozessoren genannt, ein programmierbarer Sound-Chip ist aber ebenso ein Zusatzchip. Um den Amiga bei seiner Arbeit unter die Arme zu greifen, stehen drei Coprozessoren zu Seite: →Paula, →Agnus und →Denies. In neueren Systemen sind aber andere Bezeichnungen eingeführt worden wie bzw. Alise.

CPU

siehe →Mikroprozessor.

CSI

Das CSI (Controll Sequence Introducer) Zeichen, eine Konstante mit dem Wert $9B, ist ein Steuerzeichen, das erweiterte Funktionen einleitet. Auf dem →CLI können dann beispielsweise Texte fett dargestellt werden, oder Zeilen gelöscht werden.

Custom-Chip
: Engl. Wort für →Coprozessor.

Denise
: Ist der →Custom-Chip, der die Aufgabe der Farb- und Bilddarstellung der Sprites übernimmt.

DAC (Digital-to-Analog Converter)
: Eine Schalteinheit, die Binärwerte in Spannungsschwankungen, also Analogwerte umsetzt. Wenn Sound abgespielt wird, oder wenn Messwerte ausgegeben werden, müssen die intern digital gespeicherten Informationen in analoge umgewandelt werden. Im Deutschen wird DAC selten verwendet, der Begriff D/A-Wandler ist in unseren Breiten bekannter. Die Abk. D/A steht für Digital-Analog.

Debugging
: Bei der Programmerstellung kommen immer Fehler vor, das ist normal. Statistiken sagen aus, das 80 % der Programmentwicklung nur zum Suchen der Fehler aufgewendet wird. Da das Suchen oft schwierig ist, kann ein Debugger zur Verfolgung und Beseitigung der Fehler dienen, indem er Programme traced, also schrittweise abarbeitet. Debugging hat den Wortkern „bug“, was so viel wie „Wanze“ bedeutet, debugging ist also Wanzensuche.

digital
: Finger kann man zählen. Das ist genau das was digital bedeutet: „Finger“. Dieses lateinische Wort drückt daher aus, was man unter digital verstehen kann, abzählbare Werte. Als Gegensatz wird →analog verwendet.

Directory
: Ein Directory, auch Verzeichnis oder Ordner genannt, ist eine Ebene im Dateisystem, im dem Dateien abgelegt werden. Im Gegensatz dazu steht ein flaches Dateisystem, welches keine Unterverzeichnisse kennt. Flache Dateisysteme sind ausschließlich bei Kleinrechnern zu finden. Der C-64 und sein Floppy bzw. der alte Apple hatte ein flaches Dateisystem.

Disassembler
: Ein Assembler wandelt ein Assembler-Programm ein Maschinensprache-Programm um. Es liegt dann in der direkten Form vor, alle Sprungadressen und →Konstanten sind entfernt worden. Den umgekehrten Weg geht ein Disassembler, er macht aus einem ”hart” codierten Programm eine Textdatei. Die ROM-Dokumentation dieses Buches wurde auch mit einem Disassembler erstellt.

Diskette
: Die Diskette ist ein magnetisches Speichermedium, das Daten mit einer bestimmten Kapazität aufnimmt. Im Regelfall ist es mit dem →MFM Format Formatiert, und bietet Platz für ca. 880 KB.

DMA
: Das Wort DMA (Direct Memory Access) beschreibt eine Technik, um Coprozessoren ohne Hauptprozessorunterbrechung auf den Speicher zugreifen zu lassen. Ein DMA-Kanal ist eine Buslinie für eine Speicheraktion. Insgesamt gibt es sechs DMA Kanle: Bit-Plane-DMA, Sprite-DMA, Disk-DMA, Audio-DMA, Copper-DMA, Blitter-DMA.

Drucker
: Ein Drucker ist ein Peripheriegerät zur Ausgabe von Daten. Diese Daten können Programmlistings, Texte, Tabellen oder Grafiken sein. Man unterscheidet verschiedene Druckerarten, die sich oft wesentlich in den Punkten Geschwindigkeit, Druckqualität und Kosten bzw. Wartung unterscheiden. Unterschiedliche Drucker, die sich alle einer anderen Methode bedinen sind: Typendrucker, Matrixdrucker (auch Tintenstrahldrucker), Laserdrucker, Thermodrucker, Magnetdrucker.

Dual-Playfield-Mode
: Eine Technik, die es erlaubt, zwei unabhängige Grafikbildschirme zu verwalten. Die Playfield sind transparent, also durchscheinend, die meisten Jump-And-Run Spiele machen davon Gebrauch.

Duplex
: Bezeichnung für eine Übertragungsart, bei der beide kommunizierenden Stellen senden und empfangen können.

ECS
: Die ECS (Enhanced Chip Set) Chips sind eine Neuauflage von Agnus und Denise. Durch die neuen Bauteile werden, besonders unter OS 2.0, neue Grafikauflösungen möglich. Doch nicht nur die Auflösungen sind gestiegen, das ganze Grafiksystem ist nun flexibler geworden. Neue Features sind: Blitterblock 32 k * 32 k, Bild-Frequenzen frei programmierbar.

Editor
: Ein Editor ist vergleichbar mit einer kleinen Textverarbeitung, nur ist der Zweck der Texterstellung ein anderer. Während bei einer Textverarbeitung der Wert auf ein äußeres Erscheinungsbild gelegt wird, und die optische Akzeptanz gut sein muss, verfolgt ein Editor den Zweck, Programme oder Tabellen zu bearbeiten. Aus diesem Grunde ist auch die Anforderung und die Programmierung eines Editors eine ganz andere. Ein Editor muss beispielsweise bestimmte Zeilen anspringen, eine Option, die nicht typisch für ein Textverarbeitungsprogramm ist.

Emulation
: Bei der großen Anzahl der verschiedenen Rechnertypen und Betriebssystemen möchte man oft Programme übernehmen. Um sie zu portieren, das heißt auf anderen Rechnern zu übertragen, ist es oft notwendig wegen eines anderen Prozessors oder wegen eines anderen Betriebssystems die Verhaltensweisen des Originalsystems nachzubilden (engl. emulate = nachbilden). Dieses Nachbilden wird Emulieren genannt.

Exec
: Die elementarste →Library im Amiga OS ist die `exec.library`. Sie regelt das Libraryhandling und das →Device-Handling, die Verwaltung von →Tasks, →Speicher, →Semaphoren und noch einiges mehr. `Exec` ist das eigentliche Multitasking-OS, alle anderen Libraries regeln lediglich das drum-herum, wie die Ausgabe auf dem Bildschirm. Ausschnitte aus dem ROM-Listing von Exec werden im Anhang beschrieben.

Farbregister
: Ein Farbregister ist eine Speicherzelle im Bereich $DFF180 bis $DFF1BE. Das Farbregister $DFF180 ist der Hintergrundfarbe vorbehalten. Eine Änderung dieser Register im →Multitasking bedeutet nicht gleichzeitig eine Änderung der Farben. Sie werden vielmehr über die →Copperliste gesetzt.

Fast-File-System (FFS)
: Der Diskettenzugriff ist wegen der recht verworrenen Speicherverwaltung sehr langsam. Neben dem normalen Zugriffssystem gibt daher das Fast-File-System, das speziell für Festplatten entwickelt wurde, aber auch auf Disketten übertragen werden kann. Mit dem FFS wird eine höhere Geschwindigkeit erreicht, und eine etwas bessere Platzausnutzung.

Fast-RAM
: Im Gegensatz zum →Chip-MEM ein Speicherbereich, auf dem die →Custom-Chips nicht zugreifen können. Daher ist der Datentransfer →CPU und Speicher und die Ausführung der Programme schneller.

Fat Agnus

Der Fat-Agnus ist ein erweiteter Agnus-Chip, der in der 500er Reihe eingesetzt wurde. Neue Amigas werden nur noch mit Fat-Agnus ausgeliefert, oder sogar mit ganz einem ganz neuem Agnus in den A1200 oder 4000 Rechnern. Wer wissen will, ob er einen Faten hat, der kann einmal den Rechner aufschrauben (Garantieverlust!) und nachschauen, ob ein Chip mit der Typenbezeichnung 8370 NTSC oder 8371 PAL zu finden ist. Wenn ja, hat er einen, und kann einen Hauptvorteil genießen: er kann einen MB →Chip-RAM ansprechen.

Festplatte
: Ist ein anderes Medium zur Sicherung von Daten. Sie ist wesentlich schneller als eine →Diskette, die Zugriffszeiten guter Platten bewegen sich zwischen 9-12 ms.

Floppy disc
: Ein anderes Wort für →Diskette.

Fragmentierung
: Bei der Aufteilung des Speichers kommt es früher oder später zur einer Fragmentierung. Ist beim Start des Rechners der ganze Speicherblock als ein großer zusammenhängender Bereich verwaltet, ist schon nach ein paar Aufrufen der Speicher von den benutzenden Funktionen zerstückelt. Nach Freigabe der belegten Blöcke ist in den seltensten Fällen wieder ein großer Block zusammengefügt, es verbleiben Fragmente.

Genlock
: Es ist möglich, den Amiga durch eine externe Videoquelle zu synchronisieren, und ihm ein Bild auf den Monitor vorzuspielen. Dieses Bild wirkt als Hintergrund, die normale Computer-Grafik ist aufgesetzt. Die Farbe des eingespielten Bildes ist die der Hintergrundfarbe.

Gemultiplexte Adressierung
: Um Speicher aller Art ansprechen zu können, muss man über Adressleitungen verfügen. Diese werden von einem →Prozessor über den Bus zu den Speicherbauteilen gelegt. Je größer der Speicher wird, desto größer muss daher die Anzahl der Adressleitungen sein. Um die Anzahl der Leitungen zu reduzieren kann nun die Adressierung gemultiplext werden. Nehmen wir an, wir haben 64KB Speicher, also 65536 $=2^{16}$ Speicherbytes. Dann benötigen wir 16 Adressleitungen. Aber man kommt auch mit 8 Leitungen aus, und zwar dann, wenn man zuerst die oberen acht Adressbits, und dann die unteren 8 Adressbits überträgt. Diese Aufspaltung in Lo und Hi wird gemultiplexte Adr. genannt. Das gleiche funktioniert natürlich auch mit einem 2^{32} Bit Speicher und 16 Adressleitungen. Die Kosten werden damit gesenkt, und der Computer billiger. Leider ist der Speicherzugriff dann doppelt so langsam.

Gleitpunktdarstellung
: Da die normalen →CPUs keine Fließkommazahlen verarbeiten, müssen reelle Zahlen nachgebildet werden. Eine reelle Zahl wird daher als Produkt zweier Werte dargstellt. Der eine Faktor ist die Mantisse, uns gibt den Zahlenwert an, und der andere Faktor ist der Exponent.

Globale Variable
: Eine Variable ist global, wenn sie im ganzen Programm benutzbar ist. Das Gegenteil ist →lokal, wenn ihr Geltungsbereich nur auf eine Prozedur oder einem Block beschränkt ist.

Guru
: Bei einem Absturz ist der Rechner noch so freundlich, und gibt einen kleinen roten blinkenden Kasten aus. In diesem sind Informationen über die Absturzart zu finden. Man sollte die Meldung nicht verwerfen und überlesen, sondern sie zur Fehlersuche heranziehen. Die Nummer wird Guru Meditation genannt. Sie gibt wertvolle Hinweise, ob z. B, kein Speicher vorhanden war, oder ob der Prozessor eine Aktion auf eine ungerade Speicheradresse durchführte, oder ob ein nicht-möglicher Befehl auftrat. Das Wort Guru steht natürlich auch für einen Insider in einem bestimmten Fachgebiet.

Halbduplex
: Das Gegenteil von →duplex. Die Übertragung ist nur in eine Richtung vorhanden. Eine Stelle ist entweder nur Empfänger.

HAM
: Aus den 32 Farbregistern generiert der Amiga durch Einbeziehung von Nachbarpixeln eine Farbauflösung von 4096 Punkten. HAM aus Hold and Modify.

Hardware
: Die Geräte, die an einem Computer angeschlossen werden können, werden Hardware genannt. Typische Vertreter dieser Rasse sind Festplatten, Drucker, Monitor, Erweiterungskarten, Maus.

Hash-Suche
: Dieses Suchverfahren ist eines der effizientesten. Durch einen Suchschlüssel wird ein Datensatz errechnet. Man ist bestrebt, die Berechnung möglichst einfach zu gestalten, damit das Suchen schnell ist.

HiRes
: Eine Auflösung von 640 Punkten in der Horizontalen wird HiRes (High Resolution genannt. 1280, also das doppelte wird →Superhires genannt.

Hunk
: Assemblierte Programme, so wie sie im Speicher stehen, können natürlich nicht so auf Diskette übertragen werden. Damit sie später als geladene Dateien ablauffähig sind müssen sie vielmehr in eine spezielle Form gebracht werden. Die Programme müssen überall im Speicher ablauffähig sein, absolute Adressen müssen also gelöscht, und kenntlich gemacht werden. Später dann, wenn das Programm an dem Speicherbereich steht, wo gerade Platz war, müssen die Bezüge wieder hergestellt werden. In den Hunks werden nun die Adressen gesichert, wo absolute Sprünge auftraten, um sie später zu restaurieren. Aber in den Hunks steht noch mehr. Wenn die Assemblerzeile SECTION BSS auftaucht, heißt es: Es wird Speicher einer bestimmten Größe angefordert. Dieser freie Block wird nicht abgespeichert, was Platz spart, sondern in den Hunks vermerkt. Später dann, wird dieser Platz allokiert und beschrieben.

IC
: Ein IC (integated circuit) ist eine integrierte Schaltung. Im Deutschen wird sie daher auch IS genannt. Ein IC ist eine miniaturisierte Schaltung, in der Bauteile wir Transistoren, Widerstände, Kondensatoren und Spulten untergebracht sind. Im Laufe der Zeit nahm die Integrationsstufe, also die Anzahl der Bauteile pro Chip, immer weiter zu, so dass wir heute über 2 Millionen dieser Bauteile unterbringen können.

Icon
: Ein anderes Wort für Icon ist Sinnbild. Dieses Bildchen, eine kleine rechteckige Grafik, steht stellvertretend für eine →Datei oder ein →Verzeichnis. Auf der →Workbench kann mit diesen Icons gearbeitet werden, ab bekanntesten ist der Doppelklick, mit dem eine Applikation gestartet wird.

IFF
: Die Firma Electronic Arts hat im Anfang des Jahres 1985 ein Austausch-Format auf den Markt bebracht, das Grafik-, Musik-, Zeichen-, Textprogrammem den Austausch von Daten erleichtern sollte. Am bekanntesten dürfte das IFF-Format als

Grafikformat sein. Jedes IFF (Interchange File Format)-File hat einen Kopf und Chunks.

Inhaltsverzeichnis
: Das Inhaltsverzeichnis einer Diskette ist die Gesamtheit aller Dateien und Ordern in dem Verzeichnis.

Initialisieren
: Um eine Variable benutzen zu können, muss sie vorher mit einem Wert geladen werden. Meistens ist das Null. Bei Compilern entsteht häufig das Problem, dass in Unterprogrammen, wo häufig der Stack benutzt wird, die Variablen nicht initialisiert werden. Somit sind sie mit zulässigen Werten geladen, und das bringt zusätzlich Ärger.

Interlace
: Um die Zeilenanzahl zu verdoppeln (also von 200 auf 400 und 256 auf 512) kann man die Grafikhardware dazu bringen, die Bildschirmdarstellung in zwei Seiten aufzuspalten, die dann getrennt übergeben werden. Zuerst wird das Long-Frame übergeben (ungerade Zeitennummer), und dann das Short-Frame. Da es natürlich länger dauert zwei Seiten als nur eine zu übermitteln, kommt es zu einem Interlace-Fackern, das recht unangenehm ist. Zur Lösung können Flicker-Fixer eingesetzt werden. Eine Normalauflösung ist →Non-Interlaced.

Interrupts
: Interrupts sind kleine Unterbrecher, die den Prozessor immer anhalten, und ihn dazu bewegen, andere Programme auszuführen. Viele Komponenten können Interrupts erzeugen, um auf sich aufmerksam zu machen. Ist der Blitter fertig, ein Interrupt, ein Block geladen, Interrupt, Wecker schellt, Interrupt. Natürlich lassen sich die Komponenten vorschreiben, ob sie einen Interrupt erzeugen dürfen, oder nicht.

Iteration
: An anderes Wort für Repetition oder Durchlauf ist Iteration. Ein Schleifendurchlauf ist nichts anderes als eine Iteration.

Kickstart
: Ein anderes Wort für Kickstart ist →Betriebssystem. So wie Windows „Windows“ heißt, heißt unser OS eben Kickstart, vielleicht, weil man es erst kicken muss, damit es läuft?

Kommandosprache
: Eine Kommandosprache ist nichts anderes als ein Befehlssatz für Kommunikation zwischen →Betriebssystem und Anwender. Durch die Kommandos werden Aufträge an den Computer gestellt, die ihn z. B das Inhaltsverzeichnis einer Diskette ausgeben lässt.

Konfigurieren
: Der Amiga ist ein Ausbau- und Umbaucomputer. Um die große Anzahl der Erweiterungskarten in den Rechner anzufügen, muss dieser konfiguriert werden. Wenn keine manuelle Einstellung nötig ist, und die Karten es (das Konfigurieren) automatisch machen, spricht man von einer Auto-Konfiguration. Auch Programme müssen u.U. konfiguriert werden, damit sie auf dem Rechner laufen. Einige besondere Programme verlangen Angaben über den Rechentyp, den Speicher oder die Festplatte.

Konstante
: Konstanten sind nichts anderes als Bezeichner mit einem festen Wert. Für uns Assemblerprogrammierer sind Konstanten sehr wichtig, denn sie ermöglichen es uns, nicht die Zahlen zu sehen, sondern einen Platzhalter, einen Text. Konstanten zum Amiga-OS sind in den →INCLUDE-Files zu finden.

Laufzeitfehler
: Ein Programm, das richtig compiliert wurde, enthält zwar keine syntaktischen Fehler mehr, es kann aber immer noch im Ablauf fehlerhaft sein. Eine Division durch Null kann beispielsweise nur beim Ablauf getestet werden, was wir dann haben ist ein Fehler, der beim Lauf des Programms auftrat, ein Laufzeitfehler.

Library
: Libraries sind wichtige Einheiten im →Betriebssystem. Neben den →Devices stellen die Libraries →Betriebssystemfunktionen bereit. Die Libraries sind alle shared, d. h. mehrer Benutzer oder Programme können eine Library benutzen. Das stellt natürlich Ansprüche an die Programmierung, aber machen sie in einer →Multitasking-Umgebung unentbehrlich. Hin und wieder wird auch die dtsch. Übersetzung Bibliothek benutzt.

Local Bus
: An einem lokalen Bus hängen alle Komponenten, wie die Haltestellen an einer Buslinie. Die Haltestellen sind im Rechnersystem →CPU, →Speicher, Video-Chips und auch Erweiterungskarten. Anders als bei PC’s, bei denen nicht (unbedingt) die gesamte Kommunikation über einen zentralen Bus geht, sodern über Schnittestellen. Diese sind vergleichbar sind mit der seriellen Schnittstelle, und die ist bekanntlich langsam. Es ist nun einmal ein Unterschied, ob Daten mit Bustakt, verarbeitet werden, oder sich erste durch die Schnittstelle quengeln müssen.

LoRes
: Die einfache Auflösung von 320 Punkten in der Breite.

Makro
: Durch ein Makro (engl. macro) kann der Assemblerprogrammierer noch einfacher und schneller arbeiten. Ein Makro ist eine Reihe von Befehlen, die an entsprechende Stelle kopiert werden. Ein Makro wird durch seinen Macronamen aufgerufen. Man darf ein Makro nicht mit einem →Unterprogramm verwechseln, denn dieses wird nur einmal angelegt, und dann immer per Assembler-Befehl aufgerufen. Ein Makro wird nicht durch einen Sprungbefehl angesprochen, sondern die Befehle Zeile für Zeile übernommen.

Mikroprozessor
: Ein Mikroprozessor ist ein programmierbarer Mikrochip. Dieser kann im Speicher stehende Programmteile abarbeiten. Dazu besitzt er einen bestimmten Befehlsvorrat. Siehe auch →CPU

MIDI
: Wichtige Hersteller von elektronischen Musikinstrumenten haben einen Standart (MIDI aus Musical Instrument Digital Interface) für die Überragung von Daten geschaffen. Somit können alle MIDI-fähigen Gerte gekoppelt und benutzt werden.

Message
: Programme können Nachrichten über →Ports schicken und empfangen. Damit wird das →Betriebssystem so richtig multitaskingfähig, denn von jeder Seite können Events entgegengenommen werden. Höhere Instanten, wie z. B. Intuition senden regelmäßig Nachrichtenpakete, wenn das Fenster bewegt wird, oder wenn Tasten gedrückt werden.

Mmnemonik
: Mnemonics stehen für "mnemonische Symbole" und sind die Assembler-Befehle, wie MOVE, ADD. Enwickler müssen sich also keine Zahlen für den Maschinencode merken, sondern nur die Befehle.

Multitasking
: Die Eigenschaft mehrere Programme quasi gleichzeitig auszuführen zu können wird Multitasking genannt. Die Programme

(Tasks) werden vom →Scheduler umgeschaltet, dieser Vorgang wird als →Task-Switching bezeichnet.

Modem
: Das Wort Modem ist ein Kunstwort aus den Begriffen Modulator und Demodulator. Mit ihm kann man Daten über die Telefonleitung übertragen. Die Daten werden vom Computer digital an das Modem gegeben, dieses moduliert das Signal auf eine bestimmte Trägerfrequenz. Wenn die Daten ankommen, übersetzt das Empfänger-Modem die analogen Frequenzen wieder in digitale, es demoduliert.

Motherbord
: Ist die Hauptplatine eines Rechners.

Nibbel
: Vier Bits, oder eine Halbbyte werden als Nibble bezeichnet. Ein Nibble entspricht eine →Hex-Zahl. Nicht verwechseln mit Nippel!

Non-Interlaced
: Ein nicht geinterlaces Bild. Also mit einer Höhe von 200 bzw. 256 Punkten. Nicht nur Computer senden das Bild, Monitore müssen es auf die Scheibe werfen. Einige Monitore können hohe Auflösungen, wie z. B. 600 Punkte Höhe nicht mehr darstellen, und müssen es interlacen. Beim Monitorkauf sollte man darauf achten, einen Non-Interlace-Monitor zu erwerben.

NTSC
: NTSC (National Television Standards Committee) ist eine Norm für das →Composite Video Signal. Es ist ein Stardart der USA, und mach sich hier bei uns dadurch bemerkbar, dass nur eine Zeilenzahl von 200 erreicht wird. Im Gegensatz zum NTSC-Mode ist der europäische →PAL-Mode mit 256 Punkten non-intelaced.

Objektcode
: Ein Assembler kann im Normalfall zwei unterschiedliche Wege der Codeerzeugung gehen. Zum einen kann er ein Programm direkt in den Speicher schreiben, oder auf ein Medium. Das schreiben in den Speicher ist verhältnismäßig einfach, Sprungadressen, Labels, Referenzen können direkt eingefügt werden. Wird das Programm aber auf Diskette geschrieben, so muss man weiter unterscheiden. a) Möchte man ein ablauffähiges Programm haben, also mit →Hunks usw., oder soll es ein Objektprogramm werden. Objektprogramme sind nicht ablauffähig sie in einer speziellen Form. Um sie ablauffähig zu machen, müssen sie durch einen →Linker fertig gemacht werden.

OS (Operating System)
: Anderes Wort für →Betriebssystem, nur eben kürzer!

PAL
: Eine Übertragungsart (Phase Alternate Line), die Europaweit eingesetzt wird. Mit ihr ist eine Zeilenhöhe von 256 Punkten normal. In Amerika wird →NTSC genutzt.

Parallel
: Werden Arbeitsschritte gleichzeitig und unabhängig ausgeführt, so spricht man von parallelen Abläufen. Die Programme im Amiga sind nicht richtig parallel, sonst müsste man eine Mehrprozessorstation haben, sondern nur quasi parallel, das Wort →Multitasking kann da täuschen. Parallel ist auch ein Begriff, den man bei Druckern verwendet. Die Daten werden nicht hintereinander ausgegeben, wie bei →seriellen Druckern, sondern es wird immer ein →Byte gleichzeitig übermittelt.

Pass
: Ein Pass, engl. Durchlauf, ist eine Stufe der Codeerzeugung. Im Regelfall benötigen Compiler immer mehrere Durchläufe, bis ein Programm ablauffertig ist. Ein Assembler braucht mindestens 2 Pässe, in einem werden die Labeladressen bestimmt, und im anderen der Opcode ersetzt.

Paula
: Ist ein →Custom-Chip, der drei Aufgabenbereiche hat. Er macht die Musik, die Diskettenoperationen und das →Interrupt-Handling.

PIC
: Eine Erweiterungskarte wird einfach PIC (Pic-In-Card) genannt. Der Amiga 600 ist ein Steckcomputer mit freiem Port.

Pipeline-Verarbeitung
: Immer wieder ist man bestrebt, die Ausführungszeiten von Mikroprozessorbefehlen weiter zu minimieren. Ein Konzept der Beschleunigung ist die Pipeline-Verarbeitung. Ein →Mikroprozessor muss zur Ausführung eines Befehles verschiedene Stufen durchlaufen: Holphase, Decodierphase und Ausführungsphase. Innerhalb des Steuerwerkes werden jetzt diese drei Stufen gleichzeitig für drei Befehle angewendet, während der erste Befehl geholt und dekodiert wurde, ist gleichzeitig der zweite schon in den Prozessorspeicher geladen worden. Nach weiteren Durchläufen wird sich folgendes Bild ergeben: erster Befehl geholt, zweiter Befehl decodiert, dritter Befehl ausgeführt. Und das alles gleichzeitig. Damit dies funktioniert, muss der Chip erheblich aufwendiger konstruiert werden, denn es sind drei verschiedene Funktionseinheiten für drei Befehle notwendig.

Pixel
: Ein Pixel (auch „Dot“ genannt) ist die kleinste Grafikeinheit, ein Punkt auf dem Bildschirm. Der Name Pixel kommt von PIcure(X)ELement. Der Pixel ist ein Bit in der →Bitmap.

Port
: Ein Port ist eine Schnittstelle, die den Computer mit der Außenwelt verbindet. Ein Port ist aber auch in der Betriebssystemprogrammierung ein Begriff. Es ist ein Briefkasten, in dem Messages abgelegt werden, im Allgemeinen spricht man dann von einem MessagePort.

Programmiersprache
: Eine Programmiersprache ist eine Spreche zur Formulierung von →Algorithmen. Auch Assembler ist eine Programmiersprache, nur, keine hohe. Bekannte Sprachen sind: C (Betriebssystemprogrammierung), PASCAL und MODULA (Lehrsprache), BASIC (veraltete Beginner-Programmiersprache), FORTH (Stack-Sprache), PROLOG (KI-Sprache), ADA (vom Verteidigungsministerium der USA geförderte Sprache)

Prozessor
: Siehe →CPU.

RAD
: Eine resetfeste →RAM-Disk wird RAD genannt. Sie ist ein virtuelles Laufwerk, uns muss mit mount RAD: angemeldet werden.

RAM
: RAM (Random Access Memory) ist eine Speicherart, die beliebig beschreibbar ist. Dummerweise ist der Inhalt nach dem Ausschalten weg. Dieses Nach-Strom-Ist-Dann-Weg wird auch flüchtig genannt, der Speicherinhalt flüchtet. Im Gegensatz dazu sind nicht-flüchtige Speicher, oder →ROMs.

Rechenwerk

Ein Prozessor benötigt ein Rechenwerk, um arithmetische und logische Operationen wie Addition, Subtraktion und Verknüpfung ausführen zu können. Abgekürzt wird das Rechenwerk meist mit ALU, das für Arithmetisch-Logische-Einheit (Unit) steht.

Register
: Ein Register ist ein Speicherplatz auf dem Prozessor, in dem Daten und Adressen gesichert werden können. Alle Register des MC860x0 Prozessors haben eine Länge von 32 Bit.

Reset
: Ein Reset ist ein zurücksetzen des Rechners, in den Zustand, in dem er sich beim Einschalten befand. Unterschieden wird häufig noch zwischen Warmstart und Kaltstart, allerdings selten bei Amiga. Dem Kaltstart kommt einem Ausschalten oder eine Druck auf die Reset-Taste gleich, der →Warmstart wird durch Tastaturdrücken, wie Ctrl-LAmiga+RAmiga, durchgeführt. Der Kaltstart ist ein Zurücksetzen des Prozessors, ein Warmstart ein Programm, das alles wieder ins Lot rückt.

Resolution
: Unter diesem engl. Wort verstehen wir →Auflösung.

RISC
: RISC ist eine →Prozessor-Technik, die im Gegensatz zu CISC steht. Während CISC Prozessoren oft viele Befehle und Adressierungsarten kennen, wurde bei einer RISC-CPU zurückgeschraubt, und nur das implementiert, was oft verwendet wird. RISC Prozessoren haben sehr hohe Ausführungszeiten, die über den CISC-Chips liegen.

ROM
: Ein Nur-Lese-Speicher. Wichtige Programme sind im ROM (Read-Only-Memory) jedes Rechners abgelegt. Große Teile unseres OS wird im ROM vor dem Überschreiben gesichert.

Root-Block
: Neben den →Boot-Sektoren sind weitere Sektoren für das Disketten-File-System reserviert. Üblicherweise auf Seite 0, Track 40, Sektor Null (d. h. eigentlich Sektor 880) finden wir den Root-Block. Er enthält die Informationen über die Diskette, und das oberste →Directory.

RS 232
: Siehe →Serielle Schnittstelle.

Sample
: Eine digitalisierte Tonfolge wird Sample genannt. Sample ist wieder englisch und heißt Probe, wir nehmen also, z. B. bei der Musik, beim →digitalisieren von einem minutenlangen Lied eine Probe auf.

Sample Rate
: Die Rate, mit der ein →Sample aufgenommen wird.

Scrolling
: Wenn Grafikdaten auf einer Seite verschwinden, und neue auf neue hineinkommen, spricht man von Scrolling. Wenn die Scrollschritte stufenlos sind und das Ergebnis ruckelfrei ist, dann sind spricht man von Soft-Scrolling.

SCSI
: SCSI (Small Computer System Interface) ist eine Schnittstelle für Peripheriegeräte, hauptsächlich Festplatten.

Sektoren
: Jeder →Track ist in 11 Sektoren unterteilt. Ein Sektor fasst 512 Bytes, also passt auf eine Umrundungsspur 512*11=5632 Bytes. Insgesamt hat dann eine Diskette 512 * 11 * 80 * 2 = 901120 Bytes.

Semaphoren
: Die Kommunikation der Tasks wird durch →Messages und den →Ports geregelt. Wenn jedoch mehrer Tasks auf einmalige Einheiten, wie z. B. Blitter zugreifen wollen, ist die Regelung mit Messages unzureichend. Hier greifen Semaphoren ein, die die Systemressources auf die Task verteilen.

Seriell
: Wenn auf einer Schnittstelle die →Bits nacheinander ausgegeben werden, spricht man von einer seriellen Kommunikation. Modems und Datenfernübertragungsgeräte können bisher nur seriell senden. Die Zeit der seriellen Druckes ist vorbei, →parallel ist's.

Shell
: Oft wird anstatt →CLI der Begriff Shell verwendet. Im Prinzip ist er mit dem CLI gleichzusetzen ist, nur ist er im Umgang etwas freundlicher und hilfsbereiter, so sind durch die Cursortasten vorher getippte Befehle zugängig. Der Begriff Shell und CLI ist oft verwirrend.

Slot
: Ein →Port, in dem Erweiterungskarten eingesteckt werden können, wird auch Slot genannt. Der Amiga 2000 hat fünf 100-polige Slots.

Software
: Weiche Ware ist das Gegenteil von →Hardware. Man versteht unter Software alle →Programme oder →Dateien.

Speicher
: Der Speicher ist ein Ort, an dem Daten aufbewahrt werden. Man unterscheidet Speicher nach einigen Kriterien: Zugriffszeit (70 ns, 20 ns?), Speicherkapazität (256 MB?), Energieabhängigkeit (1 Watt pro MB?).

Speicherschutz (Memory Protection)
: Wenn verschiedene →Tasks im System werkeln, steigt auch die Wahrscheinlichkeit, das bei einem →Absturz oder Endlosschleifen versehendlich Speicher überschrieben wird. Um dieses Problem zu lösen haben moderne Prozessoren eine MMU (Memory Management Unit), die den Speicher in Blöcke einteilt, die sich nicht überschreiben können.

Sprite
: Ein Sprite ist eine kleine Grafik mit einer Breite von 16 Punkten und fast beliebiger Höhe. Sie wird durch die Hardware verwaltet, und ist daher besonderes schnell. Es gibt 8 →DMA Kanäle, um die Sprite-Daten aus dem Speicher lesen zu können.

Spur
: Siehe →Tracks

Stack
: Auf den Stack (engl. Stapel) wird die Rücksprungadresse des aufrufenden Programms gesichert. Der Stack ist ein Zeiger (durch das Register A7 verwaltet) auf den Bereich, in dem diese Rücksprungadressen abgelegt werden können. Neben Stack wird der Begriff Head (engl. Haufen) verwendet. Für Compiler ist der Stack eines der wichtigsten Datenablageeinrichtungen.

Statusregister
: Ein Statusregister ist ein spezielles Prozessorregister, das Informationen über Ablaufart oder Flags enthält. Die Flags werden bei bedingten Sprüngen verwendet.

Stobe-Register
: Ein Register, das man beschreiben muss, damit etwas passiert. Der Inhalt des Registers ist egal, und man kann es auch nicht lesen.

Struktur
: Eine Struktur (engl. structure) ist eine Zusammenfassung von Einträgen. Strukturen sind wichtige Angelpunkte des Betriebssystems, denn in ihnen ist alles, aber auch wirklich alles zu erfahren. Viele →Betriebssystemaufrufe verlangen Zeiger auf Strukturen.

Superhires
: Eine Auflösung von 1280 Punkten mit dem →ESC Chip.

Supervisor
: Der Prozessor kann in zwei Gängen gefahren werden, den User-Mode und den Supervisor-Mode. Im ersten laufen die normalen Programme, und die meisten Betriebssystemfunktionen. Der Supervisor-Mode ist den Interrupts oder den Programmen in Ausnahmebedingungen vorbehalten.

SysBase
: →Exec nimmt unter den →Libraries eine Sonderstellung ein, denn sie muss nicht wie jede andere Library mir OpenLibrary() geöffnet werden, sondern steht sofort bereit. Einen Zeiger auf die →Basisadresse ist im RAM an der Speicherstelle 4.

Task
: Der Task (engl. Aufgabe) ist ein selbstständig laufendes Programm. Tasks können quasi parallel ablaufen, dies wird →Multitasking genannt. Wenn man das Wort Multitasking aufspaltet in Multi und Tasking weiß man auch woher der Name eigentlich kommt. Multi heißt viel, und das bedeutet eben viele (mehrere) Task laufen lassen.

Trap
: Ein Trap ist ein Ausnahmezustand des Prozessors. Er kann mit dem TRAP Befehl künstlich erzeugt werden. Unterschieden werden →A-TRAP und B-TRAP.

Track
: Jede Diskette ist in →Spuren oder Tracks eingeteilt. Die normale Amiga Diskette ist in 80 Spuren unterteilt, wobei sich Track Nr. 0 ganz außen befindet, und Track 70 ganz innen. Tracks werden wiederum in →Sektoren geteilt.

Übersetzter
: Man verwendet das Wort Übersetzer meist als Synonym für →Compiler. Unter Umständen wird der Begriff Übersetzter auch als Programm verstanden, dass Programme aus einer Zielsprache in eine Quellsprache übersetzt. So z. B. Übersetzer, die aus PASCAL-Code C-Code basteln.

UART
: Universalbauteil.

ver/Transmitter
: Diese Abkürzung steht für eine Bauteil, welches die Verwaltung der →seriellen Schnittstelle übernimmt.

Variable
: Eine Variable ist ein Speicherplatz, an dem Werte abgelegt werden. Der Speicherplatz ist durch einen Symbolnamen anzusprechen. Da der Speicherinhalt jederzeit verändert werden kann, ist der Inhalt variabel, daher der Name.

Video-Priorität
: Sie definiert, welches Objekt im Vordergrund liegt. Das Objekt ist entweder ein →Playfield oder ein →Sprite. Durch die Video Priorität wird besonders bei sich überlagernden Objekten die Darstellung geklärt.

Verzeichnis
: Alle Dateien, die auf einem Datenträger sind, sind in Verzeichnissen angeordnet. Ein Unterverzeichnis ist eine neue Ebene im Verzeichnis.

Warmstart
: Eine Restart.

Wildcard
: Ist in einem Dateinamen ein Ausschnitt oder ein Buchstabe nicht gekannt, so kann ein Wildcard dazu verwendet werden, Dateigruppen zu bilden. Die Anwendung liegt in der Shell um z. B. alle Datein mit der Endung `bak` zu löschen, der Befehl laufet dann mit dem Wildcard `delete #?.bak`.

ZORRO
: Ist der Name für den Erweiterungsport der Amiga Rechner. Der Amiga 1000 hatte einen Zorro I Bus, A2000 Rechner einen Zorro II und der Amiga 300 einen Zorro III Bus.

Nachwort

Die letzten Worte eines Buches gehören immer in das Nachwort. Ich möchte nicht versäumen, dem Leser meine Schreibtätigkeit vorstellen, um ihn zu motivieren, auch einmal ein Buchprojekt ins Auge zu fassen.

Historie: Als ich geboren wurde, ... Und dann bekam ich einen C-64. Die Programmierung in Assembler und das ROM-Listing interessierten mich damals so sehr, dass ich immer das C-64-intern dabei hatte, und jede freie Minute (sei es Zuhause oder auf dem Fahrrad) damit verbrachte. Schon damals hatte ich die Idee ein Assembler-Buch zu schreiben. Doch mein Vorhaben schlief ein, denn Weihnachten war nicht mehr fern. Von diesem Zeitpunkt an, würde ich so viel Geld zusammen haben, dass ich mir einen Amiga zulegen konnte. (Weihnachten und Geburtstag waren sowieso große Tage in meinem Leben, ohne sie hätte ich keine zwei C-64, Datasette, Floppy, Monitor, Drucker, Amiga, Monitor, Drucker.)

Den Amiga habe ich damals gebraucht gekauft, und kann mich noch genau an der Gesicht des Mädchens erinnern, als ich sie frage: „Welche Auflösungen hat der denn?“ Oder: „Zeig mal ein paar Demos!“ Na ja, leider daneben, das einzige was dabei war waren Spiele, die ich gleich löschte.

Als ich vom C-64 auf den Amiga umstieg wollte ich gleich mit Assembler beginnen. Das Umsteigen selber war nicht so schwierig, denn wenn einmal die Denkweise von Maschinensprache sitzt, geht’s schneller. Problematisch sind dann lediglich die Feinheiten. Leider fehlte mit anfänglich Literatur, bis zum Kauf des Amiga Intern, das jetzt mein ständiger Begleiter wurde. Weiteres Wissen musste ich mir zusammensuchen. Ich kam in Assembler nur durch auseinandernehmen von C-Listings, durchforsten von

unzähligen Zeitschriften und Besuchen von Büchereien weiter.

Da die Lösung von irgendwelchen Assembler-Problemen, Scrolling, Musik, Demos, mit immer Spaß machte, kam irgendwann der Tick, ein Buch zu schreiben. Der Gedanke, endlich anzufangen, kam mir im Deutschunterricht, als wir (wie so oft) nichts zu tun hatten. Ich glaube, wir haben uns einen Film angeschaut, und der Lehrer war nicht da. Ich schrieb ganze zwei Seiten auf Papier, und Zuhause übertrug ich es in ein Textverarbeitungsprogramm. Das war übriges das erste und letzte Mal, dass ich etwas von Hand vorschrieb. Langsam machte das Schreiben Spaß, und ich überlegte mir eine Reihenfolge, wie ich den Lesern Assembler nahe legen könnte. Nach einem Konzept und vielen Ideen ging die Arbeit erst richtig los. Ich wusste was ich schreiben sollte, aber nicht, welchen Inhalt es haben sollte! Der erste und zweite Abschnitt waren schnell fertig. Andere Assemblerbücher gaben mir Ideen für Routinen im dritte Abschnitt. Der vierte Teil war einer der Schwierigsten. Da kamen mir alte Kopien zur Hilfe, die ich immer von Artikeln machte, die mir interessant erschienen. Ich kopierte mir in meiner Zivildienstzeit viel zusammen, und konnte mir ein Kopier-Reich aufbauen (Das darf aber keiner verraten!). Sechs Ordner sind prall gefüllt. Die Kopien sind kapitelweise in zwei Betriebssystemordnern, einem Hardware- und Programmiersprachen- und Sonstiges-Ordner eingeordnet. Als ich mit dem Schreiben eines jeweiligen Kapitels begann, nahm ich mir einen Ordnerabschnitt heraus, und arbeitete ihn durch. Wichtige Zusammenhänge, und Wörter, die unbedingt fallen müssen, markierte ich mit einem Textmarker. Dann kam das Verfassen in die eigenen Worte. Ich nahm mir die Seiten, und fügte alle hervorgehobenen Wörter und Satzteile in einer Stichwortliste ein. Der letzte Schritt ist, aus der Stichwortliste Sätze zu bilden, und sie in Zusammenhang zu setzten. Parallel programmierte ich kleine Programmteile, die ich erstmalig in meinen Leben gut dokumentierte. Auch die Abtrennung der Unterroutinen kam erst durch das Buch, ich habe vorher keine sonderlichen Kennzeichnungen gemacht. Aber das hat mich zum einem ordentlichen Stil gebracht.

Das Schreiben eines Buches bringt wirklich viel. Das meiste lernt man beim Verfassen und Schreiben selber, weil man immer auf der Suche ist. Wie funktioniert dies und das, wie sind die Zusammenhänge? Nach den ersten Kapiteln möchte man immer mehr und immer mehr schreiben, am liebsten alles erklären, was man weiß. Leider muss ich nach mehr als 700 Seiten aufhören, da die Zeit zum Schreiben fehlt. Wenn man viel recherchiert benötigt man Zeit, das Projekt „Mein erstes Assemblerbuch“ läuft anderthalb Jahre. Und ich wollte doch so gerne 1000 Seiten schaffen. Na ja. Ich beginne bald mein Studium, und dann werde ich nur noch sehr wenig Zeit haben, zu wenig, um dieses Buch weiter zu verfassen. Und immer wieder wird mir bewusst: Es fehlt viel, zu viel. Mein Assemblerbuch ist endlich. Daher auch der Name: „Das endliche Assemblerbuch“. Jetzt ist es zu Ende. Schluss, Fertig, Ende, Aus! Bevor dieser Name von mir gewählt wurde, hatte ich „Die unendliche Assemblergeschichte“ im Gedanken, aber der neue Titel gefällt mir besser. Wunderschöne Kapitel, die geplant waren, und wofür Listings existieren, musste ich auslassen. Exec und das Task-Switching, Grafik näher beleuchtet, externe Libraries wie reqtool-, requester.library, OS 2.0, und, was am meisten schmerzt, die Hardware. Für die Hardware sind massig Programme vorgesehen: Diashow, Linien mit dem Blitter, Abspielroutinen für Sound- Tracker, u.s.v.m.

Ich würde mich ich freuen, wenn animierte Leser weiterschreiben, denn: Schreiben macht Spaß. Ich hatte auch kein Deutsch-Leistung, ich habe es sogar nach der 11 abgewählt! Wenn man motiviert ist, etwas zu machen, wenn man Ideen und Anregungen, Verbesserungen und Mängel ins Auge sieht, dann ist Schreiben genau das Richtige.

Ich würde mich freuen, einen Leser bald als Buch-Schreiber begrüßen zu können, es ist nicht schwer.

Auf ein Wiedersehen (-lesen), bis zum nächsten Buch.
