SYBEX

ATARI ST

Programmieren in Maschinensprache

Christian Nieber

ATARI ST
Programmieren
in Maschinensprache

ATARI ST Programmieren in Maschinensprache

Christian Nieber

DÜSSELDORF · SAN FRANCISCO · PARIS · LONDON

Anmerkungen:

ATARI, 260ST, 520ST, 520ST+, 1040ST, SM124, SF354, SF314, SH324 sind eingetragene Warenzeichen von ATARI Inc., USA
Centronics ist eingetragenes Warenzeichen von Centronics Data Computer Corp.

Satz: SYBEX-Verlag GmbH, Düsseldorf
Titelgestaltung: Patrice Larue/tgr
Gesamtherstellung: Boss-Druck und Verlag, Kleve

ISBN 3-88745-678-5
1. Auflage 1987

Inhaltsverzeichnis

Vorwort

Assembler ist keine Programmiersprache für Computer-Neulinge. Wenn Sie bei "ROM" ausschließlich an eine südliche Stadt denken und "Byte" für ein irrtümlich großgeschriebenes englisches Tätigkeitswort halten – dann sind Sie nicht nur mit diesem Buch, sondern auch mit Assembler und Maschinensprache allgemein falsch beraten. Sie sollten also zumindest schon einmal mit einem Computer gearbeitet und auch einige Grundkenntnisse über die Programmierung von Computern haben. Erfahrungen mit einer beliebigen Programmiersprache sind zwar nicht unbedingt nötig, können aber nützlich sein. Insbesondere gehe ich nicht davon aus, daß Sie schon einmal mit einem Assembler gearbeitet haben; dessen Bedienung und Funktionsweise werden ausführlich erklärt.

Mit den ST-Computern hat es die Firma ATARI fertiggebracht, den Personal-Computer-Markt für einige Zeit in Aufregung zu halten. Die Gründe dafür sind vielfältig. Zum einen schlug das sensationelle Preis-/Leistungs-Verhältnis ein, zum anderen setzte die konsequent verwendete grafische Benutzeroberfläche Maßstäbe. Mit der leistungsfähigen Hardware kam auch der Trend zur höheren Programmiersprache: Selbst professionelle Programme sind nur selten in Assembler geschrieben, denn der ST macht mit seiner Geschwindigkeit die geringe Geschwindigkeit so mancher höheren Programmiersprache wett. Nun stellt sich die Frage: Warum sollte man ein solches System in Maschinensprache bzw. Assembler programmieren?

Tatsache ist, daß es auch auf einem Computer wie dem ATARI ST eine Reihe von Aufgaben gibt, die man nur in Assembler programmieren kann oder die in Assembler zumindest besser als in irgend einer höheren Programmiersprache machbar sind. Es gibt in der Hauptsache zwei Gründe, aus denen die Maschinensprache interessant ist: Zum einen finden sich in fast jedem größeren Programm Funktionen, die die Geduld des Benutzers auf die Probe stellen. Hier kann die Maschinensprache Abhilfe schaffen, denn ihre Geschwindigkeit ist noch immer von keiner höheren Programmiersprache zu erreichen. Zum anderen gibt es bei jedem Computer Aufgaben, die nur in Assembler machbar sind.

Wenn es um das vollständige Ausnutzen der Hardware geht, kommt keine höhere Programmiersprache mehr mit.

Assembler ist eine Programmiersprache für Praktiker. Im Grunde sind die Befehle der Maschinensprache nicht außerordentlich kompliziert; mindestens ebenso wichtig wie das Wissen über die Funktionsweise der einzelnen Befehle ist das Wissen über ihr Zusammenspiel und die Zusammenarbeit des Programms mit der Hardware. Deshalb ziehen sich Beispiele durch das ganze Buch. Der erste Teil befaßt sich damit, Schritt für Schritt die einzelnen Befehle und ihre Verwendung zu erklären. Darauf folgt ein genaues Verzeichnis sämtlicher Maschinensprachebefehle, das auch als Nachschlagewerk geeignet ist. Dann geht es in die Praxis.

Der Umgang mit dem Betriebssystem von Maschinensprache aus wird ausführlich erklärt, und es werden viele speziell auf den ATARI ST zugeschnittene, nützliche Routinen angegeben. Schließlich geht es um die zwei wichtigsten Einsatzgebiete der Assemblerprogrammierung: Hardware-nahe Programmierung und Höchstgeschwindigkeit. An einigen Programmen wird demonstriert, wie man die leistungsfähige Hardware des ATARI ST zu Höchstleistungen animiert. Als besondere Zugabe wird außerdem ein eigenes Kapitel der Optimierung von Assemblerprogrammen gewidmet, denn fast immer ist es wünschenswert, die höchstmögliche Geschwindigkeit zu erreichen.

Um dieses Buch sinnvoll nutzen zu können, brauchen Sie einen Assembler. Davon gibt eine Vielzahl auf dem Markt, und sie unterscheiden sich sowohl in der Leistungsfähigkeit als auch im Preis voneinander. Leider ist die Assemblersprache nicht in allen Punkten genormt. Um den somit anstehenden Anpassungsproblemen vorzubeugen, befinden sich auf der beiliegenden Diskette an alle marktgängigen Assembler angepaßte Versionen der Listings aus diesem Buch. So werden Sie hoffentlich keine Probleme mit Ihrem Assembler haben. Darüber hinaus werden in Anhang B die besonderen Eigenheiten aller zum Zeitpunkt des Erscheinens des Buches für den ATARI ST verfügbaren Assembler besprochen. Auch sonst wird auf die praktische Programmierung eingegangen; so werden etwa Methoden zur Fehlersuche erläutert, oder es wird der eine oder andere Trick gezeigt, der die Arbeit mit einem Assembler erleichtern kann.

Auf einem System wie dem ATARI ST spielt die Verbindung von Assembler mit einer höheren Programmiersprache eine große Rolle – schließlich stellt die Entwicklung von Assemblerprogrammen noch immer einen erheblich höheren Aufwand dar als die Programmierung in einer Hochsprache. Deshalb geht man oft den Mittelweg: Nur die Operationen, bei denen Assembler seine Fähigkeiten wirklich ausspielen kann, werden auch in dieser Sprache programmiert – alles andere wird in einer höheren Programmiersprache geschrieben. Es ist nur manchmal problematisch, die beiden miteinander zu verbinden. Deshalb wird auf die Verbindung von Assembler mit den verbreitetsten höheren Programmiersprachen besonders eingegangen.

Gegenstand dieses Buches ist die Assembler-Programmierung. Wie fast alle Programmiersprachen leitet Assembler sein Vokabular von der englischen Sprache ab. Die Fachausdrücke werden bei ihrem ersten Auftreten erklärt und können auch im Begriffsregister nachgeschlagen werden. Sie brauchen also nicht Englisch zu können, um dieses Buch zu verstehen.

Sie werden sehen: Es macht Spaß, aus dem Computer Höchstleistungen herauszuholen, wie es nur in Maschinensprache möglich ist!

Kapitel 1

Was ist Maschinensprache?

Im folgenden Kapitel werden jene Grundlagen über Hardware und die Programmierung allgemein dargelegt, die zum Verständnis der Maschinensprache unbedingt notwendig sind. Sollten Sie allerdings schon zu jenen Fortgeschrittenen zählen, die Erfahrungen mit Maschinensprache auf anderen Prozessoren haben, so können Sie das erste Kapitel bedenkenlos überspringen.

Der Aufbau eines Computers

Unter Maschinensprache versteht man den Befehlssatz, der vom Mikroprozessor direkt verstanden und ausgeführt wird, ähnlich wie etwa in der Programmiersprache BASIC Befehle wie PRINT, GOTO oder INPUT direkt verstanden werden.

Maschinensprache unterscheidet sich jedoch grundlegend von allen anderen Programmiersprachen, denn sie stellt die unterste Ebene der Programmierung dar. Nur sie wird vom Prozessor wirklich verstanden. Alle anderen Sprachen, die sogenannten Hochsprachen, müssen erst auf die eine oder andere Art in Maschinensprache übersetzt werden. Es wird später in diesem Kapitel noch darauf eingegangen. Im Vergleich zu den zitierten Hochsprachen-Befehlen sind Maschinensprachebefehle viel primitiver; so bewirkt oft ein Hochsprachebefehl die Ausführung vieler hundert Maschinensprachebefehle.

Es liegt einfach in der Struktur der heutigen Hardware, daß Maschinensprachebefehle so wenig mächtig sind. Stark vereinfacht besteht ein Mikrocomputer aus der Zentraleinheit (CPU, Central Processing Unit), dem Speicher und den Chips für bestimmte Aufgaben wie zum Beispiel Ein-/Ausgabe oder die Erzeugung des Monitorbildes. Verbunden wird das Ganze durch ein Bussystem, das den Informationsaustausch zwischen den einzelnen Chips regelt (siehe Abb. 1.1).

Zunächst einmal zum Aufbau des Speichers: Er enthält den Programmcode und die Daten. Informationen können darin abgelegt und später wieder gelesen werden. Der Speicher ist aus einer großen Zahl von Zellen aufgebaut, die ihrerseits aus mehreren Bits bestehen (Abb. 1.2). Jedes Bit kann nur zwei Zustände annehmen, die man beispielsweise mit *An* und *Aus* bezeichnet. Man

kann die Zustände allerdings auch als die Ziffern 0 und 1 interpretieren. Fügt man nun mehrere Bits zusammen, wobei die Stelle jedes Bits festliegen muß, so kann man mit ihnen eine binäre Zahl darstellen. Mit 8 Bits lassen sich so ganze Zahlen von 0 bis 255 darstellen, mit 16 Bits schon Zahlen von 0 bis 65535.

Gewöhnlich besteht eine Speicherzelle aus 8 Bits, da viele Computer 8 Bits nebeneinander verarbeiten. Die Zusammenfassung von 8 Bits wird Byte genannt. Mehr über das binäre Zahlensystem finden Sie in Anhang A.

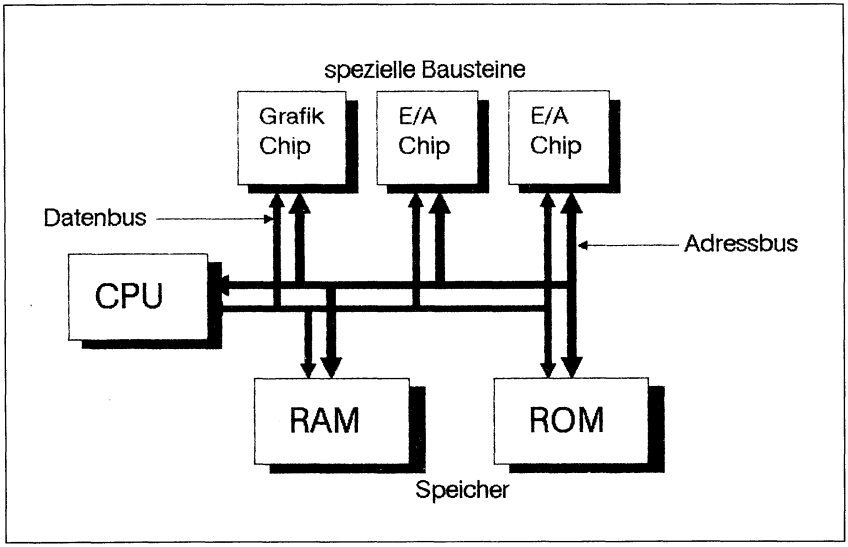


Abb. 1.1: Der allgemeine Aufbau eines Computers

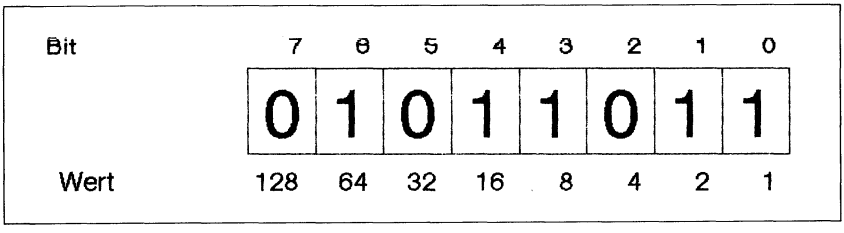


Abb. 1.2: Binäre Darstellung der Zahl 91

Im Grunde ist es aber nur eine von vielen möglichen Interpretationen, die Bits als Ziffern einer binären Zahl zu betrachten. Auf unterster Ebene handelt es sich ja nur um Gruppen von Transistoren, die zwei verschiedene Zustände annehmen können.

Oft genug wird eine Gruppe von Bits tatsächlich als eine binär dargestellte Zahl interpretiert; ein anderes Mal kann sie jedoch auch für einige Buchstaben aus einem Text stehen, und noch ein anderes Mal kann sie als ein paar Punkte auf dem Bildschirm interpretiert werden. Es gibt viele mögliche Bedeutungen für eine Gruppe von Bits. Um den Inhalt von Speicherstellen darzustellen, werden in den meisten Fällen allerdings Zahlen verwendet, weil sie die übersichtlichste und allgemeingültigste Art der Darstellung von Bitmustern bieten.

Eine Art der Interpretation von Bitmustern haben wir noch nicht erwähnt: Maschinenbefehle. Es handelt sich dabei um ein begrenztes Repertoire von einfachen Operationen, die der Prozessor auf seine Art "versteht" und ausführen kann. Damit auf die einzelnen Speicherstellen in vernünftiger Weise zugegriffen werden kann, sind sie mit 0 beginnend durchnumeriert. Man kann sich diese Nummern als "Hausnummern" vorstellen, wobei der "Bewohner" des "Hauses" der Wert der Speicherzelle ist. So wird die Nummer einer Speicherzelle auch als Adresse bezeichnet.

Der Hauptspeicher des Computers kann beschrieben und gelesen werden. Es handelt sich dabei um das sogenannte RAM, was für "Random Access Memory", also "Speicher mit wahlfreiem Zugriff" steht. Das RAM hat jedoch einen Nachteil: Sobald die Stromversorgung abgeschaltet wird, ist der gesamte Inhalt des RAM unwiederbringlich verloren. Um dies zu umgehen und ständig gebrauchte Programme immer im Speicher verfügbar zu haben, ohne daß diese bei jedem Systemstart neu geladen werden müßten, hat man das ROM entwickelt. ROM steht für "Read Only Memory", also Nur-lese-Speicher. Deshalb befindet sich bei den meisten Computern das Betriebssystem (das Programm, welches die Hardware dem Benutzer und den Anwendungsprogrammen zugänglich macht) im ROM.

Die Zentraleinheit (CPU) kann man sich als Gehirn des Computers vorstellen: Mit ihr werden alle Berechnungen durchgeführt und das gesamte System gesteuert. Die CPU hat die Möglichkeit, Speicherzellen zu beschreiben und auszulesen. Was geschieht nun, wenn sie ein Programm ausführt? Die CPU holt sich Befehlscode für Befehlscode aus dem Speicher, analysiert ihn und führt die gewünschte Aktion durch. Falls notwendig, holt sie dafür benötigte Daten auch noch aus dem Speicher. Wo sie diese Daten zu suchen hat, weiß sie durch die Adresse, die Teil des Befehls ist.

Eine CPU verfügt immer über mehrere Register. Ein Register ist eine Art Speicherzelle, die innerhalb der CPU liegt und mit der verschiedene mathema-

tische und logische Operationen durchgeführt werden können. Bei vielen CPU's können Berechnungen nur in Registern ausgeführt werden. Der Ablauf der Manipulation eines Speicherplatzes wird dann oft so aussehen, daß die Daten zunächst vom Speicher in ein Register geladen, dort den gewünschten Operationen unterzogen und danach wieder in den Speicher zurückgeschrieben werden.

Beispiel:

Im Laufe der Programmausführung trifft die CPU auf ein Bitmuster mit der binären Darstellung

```
10010110
```

oder dezimal 150. Sie stellt fest, daß dies der Befehl ist, den Inhalt eines Speicherplatzes zu lesen und in ein Register zu laden. (Dieses Beispiel ist fiktiv und entstammt nicht der Maschinensprache des MC68000.) Die Adresse, aus der gelesen werden soll, steht in den beiden darauffolgenden Bytes.

Nehmen wir an, diese Bytes enthalten binär 00000011 und 11110100, also 3 und 232. Gewöhnlich findet eine Adresse nicht in einem Byte Platz, da man damit nur 256 Speicherplätze unterscheiden könnte. Deshalb werden Adressen je nach der Größe des Arbeitsspeichers in 2, 3 oder sogar 4 Bytes dargestellt. Unser fiktiver Prozessor soll diese beiden 8-Bit-Zahlen deshalb zu einer 16-Bit-Adresse zusammenfügen, wobei das erste Byte die oberen 8, das zweite die unteren 8 Bits liefert.

Es entsteht die Adresse

```
0000001111110100
```

oder dezimal 1000. Auf dieses Ergebnis kommt man auch, wenn man 3 * 256 + 232 * 1 berechnet. Das ergibt sich daraus, daß die Wertigkeit des oberen Bytes 256, die des unteren nur 1 beträgt. Das Ganze läuft also darauf hinaus, daß der Inhalt des Speicherplatzes 1000 (dezimal) in ein Register kopiert wird. Nach der Ausführung dieses Befehls nimmt der Prozessor den im Speicher darauffolgenden Befehl in Angriff. Normalerweise werden die Befehle also linear hintereinander abgearbeitet.

Natürlich besteht ein Computer nicht nur aus CPU, ROM und RAM, denn sonst hätte er ja keinerlei Möglichkeit, mit der Außenwelt in Verbindung zu treten, also zum Beispiel Zeichen auf dem Drucker auszugeben oder ein Bild auf dem Monitor zu erzeugen. Für diese Funktionen verfügt jeder Computer über spezialisierte Chips, die die Schnittstellen bedienen, den Monitor ansteu-

ern oder die Tastatur abfragen. Natürlich muß die CPU mit diesen Chips auch kommunizieren können, um beispielsweise Parameter für eine Schnittstelle einzustellen oder die Bildschirmauflösung zu wählen. Auch dieser Datenaustausch erfolgt über den Datenbus. Die Spezialchips verfügen – ähnlich wie die CPU – über Register, die von der CPU wie Speicherzellen angesteuert werden. Das Bussystem sorgt dafür, daß die Informationen den richtigen Adressaten erreichen.

Wie sage ich's meinem Computer

Nun werden Sie sich sicher fragen, wie man den Maschinencode erzeugt. Die primitivste Form der Eingabe wäre sicherlich, den Code binär Byte für Byte einzugeben. Falls Sie jetzt meinen, dies wäre eine ausgesprochen umständliche Methode, haben Sie nicht unrecht.

Allerdings wurde in der Anfangszeit der Computerei tatsächlich so programmiert. Die Eingabe von Programmen bestand darin, in einer Reihe von 8 Schaltern binäre Zahlen einzugeben und auf einen Knopf zu drücken, sobald die richtige Kombination eingestellt war und in den Speicher gebracht werden sollte. Dann erfolgte die gleiche Prozedur für den nächsten Speicherplatz, und so weiter...

Logisch, daß diese Art der Eingabe nicht nur sehr zeitaufwendig und umständlich (schließlich mußte man ständig mit Befehlstabellen arbeiten), sondern auch äußerst fehleranfällig war, denn es ist sehr schwierig, mit binären Zahlen zu arbeiten, ohne sie durcheinanderzubringen.

Der nächste Schritt, eine Eingabe mit Zehner- oder Hexadezimaltastatur, war auch noch nicht das wahre Vergnügen, mußte man doch immer noch mit unanschaulichen Zahlen statt mit einigermaßen verständlichen Befehlsworten arbeiten, wie sie heute in allen Hochsprachen üblich sind. Den Durchbruch brachten daher erst die sogenannten Mnemonics, Abkürzungen, die die Funktionen der Befehle wiedergeben. So würde etwa der oben beschriebene Befehl, den Inhalt eines Speicherplatzes in ein Register zu laden, so lauten:

```
MOVE 1000,D0
```

MOVE (engl. für "bewege") sagt dabei, was der Befehl tun soll, nämlich Daten von einem Platz zum anderen bewegen. MOVE ist zwar ein Mnemonic, aber ausnahmsweise keine Abkürzung. Nur dort, wo die Arbeitsweise eines Befehls mit komplizierteren Worten beschrieben werden muß, werden Abkürzungen von höchstens fünf Buchstaben Länge verwendet.

Auf das Mnemonic folgen die Operanden, die mitteilen, womit die Operation durchgeführt werden soll: 1000 steht für die Quelle, also die Speicherzelle, aus der die Daten geholt werden sollen. Der zweite Operand ist das Ziel, also der Ort, wohin die Daten geschrieben werden sollen. Quelle und Ziel werden immer durch ein Komma getrennt. In diesem Fall ist das Ziel ein Prozessorregister namens D0.

Für die Programmierung mit Mnemonics braucht man natürlich schon ein komplexeres Eingabeprogramm: Es muß den Befehlscode und die Operanden analysieren, daraus mit Hilfe einer im Computer gespeicherten Befehlstabelle den Befehlscode errechnen und in einem gewünschten Speicherbereich ablegen. Solche Programme, sogenannte Direkt-Assembler, werden heute auch noch oft verwendet, allerdings nicht zur eigentlichen Eingabe von Programmen, sondern zum Austesten und zur Fehlersuche in halbfertigen Programmen, wobei man oft noch kleine Veränderungen vornehmen will.

Mit Hilfe der Mnemonics kann schon recht gut programmiert werden. Nur eine Tatsache erweist sich noch als störend: Jedes Programm muß auf eine Anzahl von Variablen zugreifen, um so mehr, je größer es ist. In Maschinensprache sind Variablen nichts anderes als bestimmte Speicherplätze, die vom Programmierer eben als Variablen benutzt werden. Wenn diese Variablen jedoch nur mit ihren Adressen identifizierbar sind, also durch abstrakte Zahlen dargestellt werden, ist es recht schwierig, ohne Verwechslungen mit ihnen umzugehen. Darüber hinaus ist der Dokumentationswert dieser Adressen gleich null, es ist also sehr schwierig, an einem solchen Programm nachträglich noch Veränderungen vorzunehmen. Deshalb ist es viel angenehmer, wenn man bestimmte Adressen mit anschaulichen, vom Programmierer gewählten Namen bezeichnen kann, die für Außenstehende verständlich sind. Diese Namen müssen nur einmal im Programm festgelegt werden, und die dazugehörigen Adressen können an allen anderen Stellen über ihren Namen angesprochen werden. Es gibt tatsächlich Programme, die dies ermöglichen: Es handelt sich um sogenannte Assembler (engl. "Zusammensetzer"). Die Namen, die für bestimmte Zahlen stehen, werden Symbole genannt.

Wenn sich in Speicherzelle 1000 eine Zählvariable befindet, würde obiges Beispiel unter einem Assembler etwa so aussehen:

```
ZÄHLER EQU 1000
        .
        .          [andere Befehle]
        .
       MOVE ZÄHLER,D0
```

Die Anweisung "ZÄHLER EQU 1000" legt fest, daß künftig das Wort "ZÄHLER" für die Adresse 1000 steht. Diese Anweisung ist eine sogenannte Symboldefinition. Der Befehl "MOVE ZÄHLER,D0" hat also für den Computer die gleiche Bedeutung und erzeugt den gleichen Code wie "MOVE 1000,D0", er ist nur für Menschen leichter zu verstehen. Erst durch die Verwendung eines Assemblers können wirklich lesbare Programme erzeugt werden. Natürlich ist es hierbei nicht mehr möglich, Befehl für Befehl einzugeben und sofort fertig für die Ausführung im Speicher abzulegen, da der Assembler alle Anweisungen auf einmal im Blickfeld haben muß, um die Symbole durch ihre Werte zu ersetzen.

Und so sieht der Umgang mit einem Assembler aus: Zunächst müssen mit einem Editor alle Befehle und Symboldefinitionen in der gewünschten Reihenfolge eingegeben werden. Dann bearbeitet der Assembler den so entstandenen Text. Er versucht nun, alle Symbole durch ihre Werte zu ersetzen und die Mnemonics in Maschinencode umzurechnen. Wenn alles korrekt ist, erzeugt er den ausführbaren Code im Speicher oder als Datei auf der Diskette.

Der Assembler stellt wohl die letzte Stufe in der Linie der Maschinensprache-programmierung dar. Den nächsten Schritt in Richtung höherem Programmierkomfort stellen die Hochsprachen dar, wobei jedoch die einfachere Programmierung mit weniger effizienten Programmen erkauft werden muß.

Und was kommt nach Assembler?

Trotz der symbolorientierten Programmierung mit einem Assembler stellt diese Form des Programmentwurfs noch einen erheblichen Arbeitsaufwand dar. Da die Maschinensprachebefehle nur so elementare Aktionen bewirken, wie sie eben ein Prozessor auf einmal erledigen kann, benötigen selbst kleine Programme schon relativ viele Befehle. Das erschwert die Fehlersuche natürlich gewaltig.

Es wäre viel schöner, wenn sich der Programmierer weniger auf irgendwelche Hardware-Eigenheiten und mehr auf das eigentliche Problem, das sein Programm lösen soll, konzentrieren könnte. Mit anderen Worten, eine Programmiersprache ist gefragt, die weniger auf die Maschine, aber mehr auf den Menschen eingeht.

Das Prinzip der höheren Programmiersprachen ist, übliche Schreibweisen wie zum Beispiel geklammerte mathematische Ausdrücke zu verstehen und außerdem häufig gebrauchte Unterprogramme wie Ein- und Ausgabe von Text direkt als Befehle zur Verfügung zu stellen. Natürlich kann der Mikroprozessor

eine solche Sprache nicht direkt verstehen. Man braucht daher ein Programm, das als Mittler zwischen Mensch und Maschine dient, also die Hochsprache in Maschinensprache übersetzt. Es gibt dafür nun zwei Möglichkeiten, die beide ihre Vor- und Nachteile haben: Interpreter und Compiler.

Falls Sie bei Interpreter (engl. Dolmetscher) an einen Interpreten, also einen "life" auftretenden Künstler denken, liegen Sie gar nicht so falsch. Ein Interpreter ist tatsächlich ein während der Programmausführung, also gewissermaßen "life" arbeitendes Programm. Es analysiert den Programmcode, untersucht ihn auf Befehlswörter und ruft entsprechende Routinen sofort auf.

Gewöhnlich verfügt ein Interpreter über einen integrierten Editor, der auch interaktives Arbeiten gestattet. Man kann Programmroutinen direkt aufrufen und vielfach auch Befehle direkt eingeben und ausführen lassen und daher sehr schnell Änderungen am Programmcode vornehmen.

Das bekannteste Beispiel für Interpretersprachen ist BASIC. Weniger bekannt sind dagegen beispielsweise Logo und LISP. Der bedeutende Nachteil der Interpretersprachen ist die Ausführungsgeschwindigkeit: Da der Interpreter einen großen Teil der Rechenzeit mit Organisation wie etwa der Analyse des Codes oder dem Suchen von bestimmten Codesegmenten zubringt, sind die entstehenden Programme nicht sehr effizient. Sicherlich haben Sie schon so manches BASIC-Programm erlebt, das selbst mit der Initialisierung schon geraume Zeit zubrachte. Dies ist auch der Grund, weshalb es relativ wenige Interpretersprachen gibt: Sie sind nur da gefragt, wo es auf eine sehr schnelle Programmentwicklung, aber nicht auf die Ausführungszeit ankommt.

BASIC war ursprünglich als Lernsprache konzipiert, die durch die interaktive Arbeitsweise einen leichten Einstieg in die Computerei ermöglichen sollte. Bei LISP liegt es dagegen am ungewöhnlichen Konzept, daß diese Sprache nur sehr schwierig anders denn als Interpreter zu verwirklichen ist. Logo, eine relativ junge Programmiersprache, dient heute noch in erster Linie als Lernsprache, bietet jedoch durch die enge Anlehnung an LISP auch Möglichkeiten für die Programmierung von künstlicher Intelligenz.

Die zweite Möglichkeit, eine höhere Programmiersprache zu verwirklichen, besteht darin, das Programm nicht stückchenweise Befehl für Befehl in Maschinensprache umzusetzen, sondern sich den gesamten Code auf einmal vorzunehmen und ihn in ein eigenständiges Maschinenspracheprogramm zu übersetzen. Das ist das Prinzip der Compiler (engl. Zusammensteller). Da die Übersetzung vor dem eigentlichen Programmlauf erfolgt, kann sich der Compiler dabei natürlich Zeit lassen und versuchen, einen möglichst effizienten Code zu erzeugen.

Wenn Sie sich den Interpreter als einen Simultanübersetzer vorstellen, dann wäre der Compiler ein Literat, der sich in sein stilles Kämmerlein zurückzieht, um dort eine möglichst kunstvolle Übersetzung eines fremdsprachigen Romans zu liefern. Somit erzeugt der Compiler Programme, die ein wesentlich besseres Laufzeitverhalten aufweisen als interpretierte.

Um Ihnen einen Anhaltspunkt zu geben: Je nach Programmiersprache, Version und Programm sind compilierte Programme etwa um den Faktor 3 bis 15 schneller als interpretierte. Allerdings muß dieser Vorteil, wie so oft in Technik und Wissenschaft, mit Nachteilen an anderer Stelle erkauft werden: Der Programmiervorgang gestaltet sich wesentlich aufwendiger, und die Vorteile der interaktiven Arbeitsweise sind dahin. Und so sieht die Arbeit mit einem Compiler aus:

1) Der Programmcode wird im allgemeinen mit Hilfe eines separaten Editors erstellt und als Textdatei abgespeichert. (Es handelt sich um den sogenannten Quellcode.)

2) Alsdann wird der Compiler aufgerufen, um sich mit dem Code zu befassen. (Meist muß der entstehende Maschinencode auch noch durch einen Linker geschickt werden. Was das ist, wird im nächsten Abschnitt erklärt.) Falls hierbei Fehler auftreten, gehe zurück zu 1, sonst fahre fort mit 3.

3) Endlich kann das Programm ausgetestet werden. Falls noch Fehler logischer Art im Programm sind, die der Compiler nicht aufspüren konnte, zurück zu 1 ...

Sie sehen also, daß man laufend zwischen verschiedenen Programmen hin- und herspringen muß. Hier erleichtert ein schnelles Speichermedium die Arbeit ganz gewaltig. Schritt 2 kann, wenn alles über die Diskettenstation läuft, durchaus fünf Minuten oder mehr dauern. Wenn das System allerdings groß genug ist, daß alle Programmteile gleichzeitig im Speicher Platz finden, so kann man auf Compilierzeiten von weniger als 30 Sekunden kommen. Dies erklärt auch, warum Compilersprachen im unteren Heimcomputerbereich, also auf 8-Bit-Mikrocomputern mit höchstens 64K Speicher, kaum Verwendung finden. Die Arbeit damit gestaltet sich einfach viel zu langwierig und umständlich.

Die meisten Programmiersprachen sind Compilersprachen: Pascal, Modula II, C, FORTRAN usw. Es gibt sie für den militärischen Bereich sowie als Lernsprachen und auch als hochspezialisierte Sprachen für künstliche Intelligenz oder Robotersteuerung.

Obwohl sie interpretierten Programmen weit überlegen sind, kommen compilierte Programme in der Ausführungsgeschwindigkeit doch niemals an ein gutes Assemblerprogramm heran. Der Grund ist darin zu suchen, daß oft in viel größerer Genauigkeit gerechnet wird, als es für das Programm eigentlich erforderlich wäre, oder aber Programmteile sind deshalb ineffektiv, weil sie schon im Quellcode umständlich formuliert waren. Letzteres kommt daher, daß das Sprachkonzept von Hochsprachen maschinennahe – und somit effektive – Formulierung oft nicht zuläßt. In Assembler hingegen hindert nichts den Programmierer daran, besonders zeitkritische Teile eines Programms mit Blick auf eine Tabelle der Ausführungszeiten der einzelnen Befehle zu optimieren.

Vielleicht werden Sie sich inzwischen fragen, was eine Beschreibung von Interpreter- und Compilersprachen in einem Maschinensprachebuch zu suchen hat. Nun, es geht um folgendes: Auf einem leistungsfähigen System wie dem ATARI ST, das genügend Speicherplatz bietet, werden größere Programme sicherlich nur selten vollständig in Assembler geschrieben. Vielmehr wird man versuchen, besonders zeitkritische Passagen von hochsprachlich formulierten Programmen durch entsprechende Maschinenspracheroutinen zu ersetzen. Daher wird in diesem Buch auch darauf eingegangen, wie Sie Maschinenspracheroutinen zusammen mit den verbreitetsten Hochsprachen verwenden können.

Was tut ein Linker?

Linker heißt auf deutsch "Binder". Er fügt (bindet) Programmteile in Form von Maschinencode zusammen. Und das ist das Prinzip: Der Compiler (oder Assembler) erzeugt keinen direkt ausführbaren Code. Statt dessen wird noch eine weitere Instanz dazwischengeschaltet: der Linker. Dies mag zunächst unpraktisch erscheinen, hat aber durchaus praktische Gründe. Die Wurzeln liegen in der Technik des modularen Programmierens.

Das Wort "Modul" ist Ihnen vielleicht schon von der Unterhaltungselektronik in Form von Fernseh- und HiFi-Geräten bekannt. Moderne Geräte sind in sauber getrennte und oft auf vielfältige Weise kombinierbare, unabhängige funktionale Einheiten gegliedert. Falls nun ein solches Gerät ausfällt, so hat der Servicetechniker nur noch das defekte Modul zu lokalisieren und als Ganzes auszutauschen. Das Arbeiten mit Modulen ist auch bei der Programmierung möglich.

Beispiel:

Nehmen wir an, Sie wollen ein Spiel mit bewegter Grafik und ein Zeichenprogramm erstellen, die beide eine Funktion zu Linienziehen brauchen. Sie könn-

ten nun für jedes der Programme eine eigene Funktion programmieren. Rationeller und eleganter wäre es jedoch, die Funktion als eigenes Modul nur einmal zu schreiben und zu compilieren. Dieses Modul müßte dann nur noch mit den beiden Programmen verbunden werden. Und genau das ist die Aufgabe des Linkers.

Im einzelnen funktioniert das so:

Der Compiler oder Assembler hinterläßt im Objektcode, den er produziert (also dem Maschinensprachemodul), nicht nur den Programmcode, sondern auch Informationen über bestimmte Symbole, die entweder speziell markiert worden sind oder einfach im Quellcode nicht definiert sind. In letzterem Fall nimmt der Compiler an, es waren damit Symbole in irgend einem anderen Modul gemeint. Nun wird der Linker mit der Information, welche Module er zusammenbinden soll, aufgerufen. Für jedes undefinierte Symbol, das in einem Modul auftritt, durchsucht er alle anderen Module in der Hoffnung, daß es in einem von ihnen definiert ist. Zu einer korrekt aufgelösten Referenz gehören also immer zwei: Ein Modul, in dem das Symbol definiert ist, und ein anderes, in dem es aufgerufen (= referenziert) wird. Das Ganze bewirkt, daß Sie Symbole aus anderen Modulen in Ihrem Programm benutzen können, als wären sie dort definiert.

Ein Linker erlaubt in Ihrer Programmierumgebung auch größtmögliche Flexibilität: So ist es beispielsweise ein leichtes, eine einfach zu bedienende Schnittstelle zu den Betriebssystemfunktionen als Modul zu schreiben, die fortan von jedem Programm genutzt werden kann.

Ein Linker erweist sich auch durch einen weiteren Umstand als nützlich: Er ermöglicht es, Programmteile, die in verschiedenen Programmiersprachen geschrieben sind, zu verbinden. Voraussetzung ist allerdings, daß die verschiedenen Compiler und Assembler das gleiche Format für ihre Objektmodule verwenden, also auf denselben Linker zugeschnitten sind. Leider trifft dies oft nur dann zu, wenn sie vom gleichen Softwarehaus stammen. Dann allerdings können Sie ein Programm teilweise in BASIC (compiliert), teils in Pascal, C und Assembler schreiben. Ob das unbedingt so sinnvoll ist, ist eine andere Frage. Eines ist jedoch ganz gewiß sinnvoll: nämlich Assembler mit einer Hochsprache zu verbinden.

Es wäre sehr mühselig, größere Programme vollständig in Assembler zu schreiben, denn trotz des relativen Komforts eines Assemblers ist das Zusammensetzen von großen Programmen aus den vergleichsweise primitiven Befehlen, die der Computer direkt versteht, eine äußerst zeitraubende Beschäftigung. So ist es doch viel einfacher, nur jene Programmteile, die entweder zeitkritisch sind oder eine maschinennahe Programmierung erfordern, in Assem-

bler zu schreiben und andere Dinge, wie beispielsweise den Aufbau von Menüs und die Kommunikation mit dem Benutzer, in einer Hochsprache zu formulieren.

Bei vielen Compilersprachen – C, Pascal und Modula II mögen hier als Beispiele dienen – bietet sich bei der Verwendung eines Linkers sogar die Möglichkeit, die Standardbibliotheken (jene Module, die die Standardfunktionen dieser Sprachen enthalten) um eigene in Assembler geschriebene Funktionen zu erweitern. Auf diese Art können Sie praktisch Ihre eigene Befehlserweiterung schreiben.

Da somit die Einbindung von Assemblermodulen in Hochsprachen gerade auf einem System wie dem ATARI ST eine große Rolle spielt, wird darauf in Anhang D eingegangen.

Kapitel 2

Einführung in Maschinensprache

Im Gegensatz zu den meisten höheren Programmiersprachen ist Assembler keine Sprache, mit der man vom ersten Moment an arbeiten kann, etwa wie in BASIC

```
10 PRINT "HALLO"
```

Vielmehr muß man schon einen gewissen Anteil der Maschinensprachebefehle beherrschen, um überhaupt ein lauffähiges Programm schreiben zu können. Am Anfang werden wir Ihnen daher nur Ausschnitte aus Programmen vorführen, die nicht dazu gedacht sind, für sich allein ausprobiert zu werden. Bedenken Sie folgendes: Beim Erlernen von Maschinensprache liegt die Schwierigkeit nicht so sehr darin, die Befehle zu beherrschen – die sind relativ schnell gelernt –, sondern vielmehr darin, aus so kleinen Bausteinen wie ein paar arithmethischen und logischen Operationen, Vergleichsoperationen und bedingten Verzweigungen das Gebäude des Programms zusammenzusetzen.

Der innere Aufbau des MC68000

Für den Maschinenspracheprogrammierer stellt sich der MC68000, die CPU des ATARI ST, "von innen" wie in Abb. 2.1 dar. Wie jeder Mikroprozessor verfügt er über eine Anzahl von Registern. Beim 68000 fällt sofort ins Auge, daß er damit besonders reichlich ausgestattet ist: 15 Register stehen dem Programmierer frei zur Verfügung. Doch befassen wir uns zunächst mit den Registern, die bestimmten Aufgaben vorbehalten sind.

Da wäre zunächst der Programmzähler (engl. program counter, abgekürzt PC). Er enthält immer die Adresse des nächsten auszuführenden Befehls. Er umfaßt 32 Bits, von denen allerdings die oberen 8 Bits in der gegenwärtigen Version des MC68000 brachliegen. Überhaupt ist das mit der Adressierung so eine Sache: Adressen sind grundsätzlich 32 Bits lang. Man könnte damit also rein theoretisch 2^{32} = ca. 4,29 Milliarden Bytes adressieren. Tatsache ist jedoch, daß eine solche Masse Speicher selbst heute noch ein kleines Vermögen kosten würde. Deshalb hielten es die Entwickler des MC68000 nicht für not-

wendig, tatsächlich alle Bits zu verwenden. Dies schlägt sich darin nieder, daß nur 24 der 32 Adreßleitungen herausgeführt wurden. Die restlichen Bits werden einfach ignoriert. Deshalb schadet es auch nichts, wenn etwa in den oberen 8 Bits einer Adreßvariable irgendwelche zusätzlichen Informationen stehen.

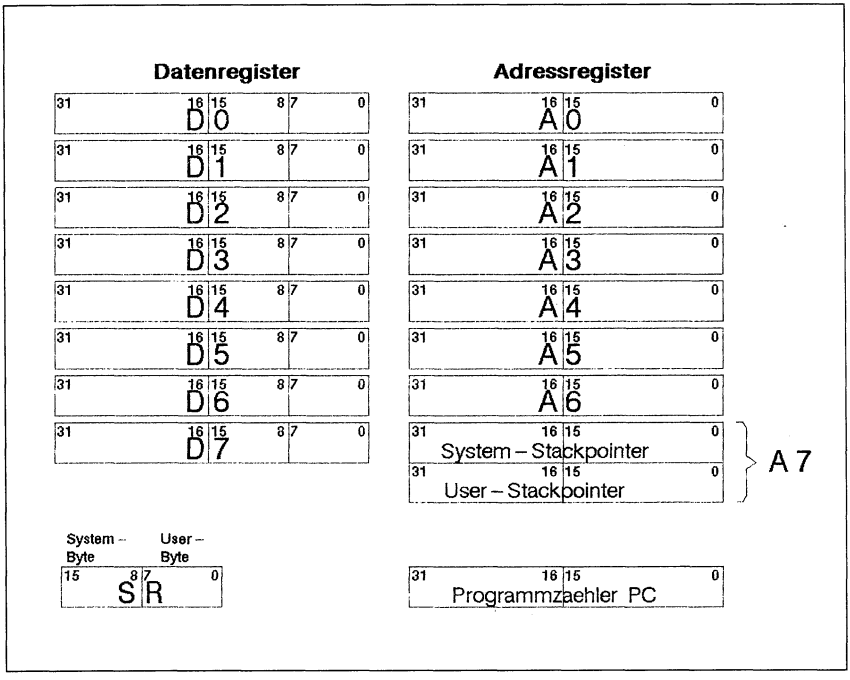


Abb. 2.1: Das Register des MC68000-Prozessors

Lange Rede, kurzer Sinn: Sie haben nur 24 Bit zur Adressierung zur Verfügung, womit genau 16 Megabytes, also 16777216 Bytes adressiert werden können. Übrigens ist der Speicher trotz der 16-Bit-Architektur des MC68000 byteweise aufgebaut, man hat es also mit Einheiten von 8 Bit zu tun. Doch damit werden wir uns später noch eingehender auseinandersetzen.

Jetzt also zurück zum Programmzähler: Er kann über einige Befehle direkt angesprochen werden. Als Programmierer braucht man sich jedoch nur selten um die Berechnung der Adressen von Programmteilen zu kümmern: Das ist

Aufgabe des Assemblers. Daher ist es nicht zu empfehlen, den Programmzähler direkt zu manipulieren.

Interessanter ist hier schon das Statusregister (SR), das 16 Bit umfaßt. Wie der Name schon sagt, enthält es Informationen über den Status des Systems. Beim 68000 wird es in zwei Teile unterteilt: Die oberen 8 Bit werden *Systembyte* genannt, die unteren 8 Bit *Userbyte*. Das Systembyte enthält eher vom laufenden Programm unabhängige Informationen über den gegenwärtigen Zustand der Hardware, während das Userbyte von jedem Programm laufend benutzt wird. Im Userbyte werden Informationen über das Ergebnis der letzten durchgeführten arithmetischen, logischen oder Vergleichsoperationen aufbewahrt. Genauer wird das Userbyte in Kapitel 2.3 beschrieben.

Von großer Bedeutung ist auch der Stackpointer (SP, deutsch: Stapelzeiger). Eigentlich müßte man sagen "die Stackpointer", denn der MC68000 hat zwei davon. Um dieses "Doppelte Lottchen" zu erklären, muß ich etwas weiter ausholen: Der 68000 kann Programme in zwei unterschiedlichen Modi ausführen, dem Supervisormodus und dem Usermodus. Im Supervisormodus ist wirklich alles erlaubt. Im Usermodus (dem Modus, in dem die meisten Programme laufen) ist man hingegen etwas eingeschränkt: Auf bestimmte Speicherbereiche darf nicht zugegriffen werden, und bestimmte Maschinensprachebefehle dürfen nicht ausgeführt werden. Es handelt sich dabei um sogenannte privilegierte Befehle.

Jetzt werden Sie sich vielleicht fragen, warum man sich solche Mühe macht, um Sie als Assemblerprogrammierer jener absoluten Freiheit (und der damit verbundenen Verantwortung) zu berauben, die der Assemblerprogrammierung doch sonst auf Microcomputern zu eigen ist. Der Grund ist darin zu suchen, daß der MC68000 ursprünglich für Mehrplatzsysteme konzipiert war.

In einem Mehrplatzsystem (ein Computer, der mehrere Terminals bedient) muß etwas anders organisiert werden, damit nicht jedes Programm die Möglichkeit hat, die Systemvariablen zu manipulieren. Dies könnte sonst leicht zu unerwünschten Nebenwirkungen auf anderen Terminals führen oder gar das gesamte System zum Absturz bringen. Dem hat man von der Hardwareseite einen Riegel in Form des Usermodus vorgeschoben. Andererseits muß das Betriebssystem und meist auch der Bediener der Systemkonsole die Möglichkeit haben, alle laufenden Programme gleichzeitig zu kontrollieren und auch die Systemvariablen zu ändern. Dafür wurde der Supervisormodus eingerichtet.

In einem Einplatzsystem, das nicht einmal für Multitasking vorgesehen ist, verlieren diese Dinge natürlich ihre Bedeutung. Doch offenbar wollten die

Entwickler des ATARI ST jene Hardwarefähigkeit nicht verschenken und entschlossen sich, nur das Betriebssystem im Supervisormodus laufen zu lassen, während Programmen der Usermodus zugeteilt wird. Dies hat den Sinn, daß ein abstürzendes Programm nicht unbedingt das ganze System mitreißt, da es die vom Betriebssystem benutzten Speicherbereiche nicht beeinflussen kann. Leider lehrt die Erfahrung, daß es trotzdem noch oft genug geschieht, aber manchmal funktioniert es eben...

Für Supervisor- und Usermodus stehen also zwei getrennte Stackpointer zur Verfügung. Der Stack (deutsch: für Stapel) ist ein reservierter Speicherbereich, der hauptsächlich dem Aufruf von Unterprogrammen und der Parameterübergabe von einer Funktion zur anderen dient. Es handelt sich dabei um einen sogenannten LIFO-Stack (engl. Last In – First Out, zuletzt hinein – zuerst heraus). Man kann sich diesen in etwa wie einen Tellerstapel vorstellen, auf den der Tellerwäscher einen Teller nach dem anderen legt und von dem der Ober immer den obersten, also den zuletzt hinaufgelegten, herunternimmt. Auch auf den Stack wird später noch ausführlicher eingegangen.

Kommen wir zu dem zunächst wichtigsten: den frei verwendbaren Prozessor-Registern. Der 68000 bietet 15 davon. Die ersten 8 werden Datenregister genannt und mit D0 bis D7 bezeichnet, die restlichen 7, A0 bis A6, sind die Adreßregister.

Wie schon der Name sagt, dienen die Adreßregister zum Adressieren, während die Datenregister zum Rechnen verwendet werden. Allerdings ist diese Aufteilung nicht ohne Ausnahmen: In begrenztem Maße kann auch mit den Adreßregistern gerechnet werden, und gelegentlich werden sogar die Datenregister zum Adressieren verwendet.

Durch die zahlreichen Register bietet es sich beim 68000 geradezu an, einen großen Teil der Berechnungen in einem Programm nur in den internen Registern ablaufen zu lassen. Dies spart nicht nur Zeit – es sind ja keine aufwendigen Speicherzugriffe mehr nötig, da die Daten gewissermaßen "vor der Tür" liegen –, sondern es spart auch Speicherplatz, da Maschinensprachebefehle, die die Register ansprechen, immer kürzer sind als solche, die Daten im Speicher adressieren.

Die Register sind allesamt 32 Bit lang. Je nach Befehl werden davon allerdings oft nicht alle Bits angesprochen: Es ist möglich, nur die ersten 16 oder die ersten 8 Bit anzusprechen. Weil der MC68000 eine Datenbusbreite von 16 Bits (= ein Wort) hat, ist dies der Standardwert für die Verarbeitungsbreite.

Erste Schritte

Zu den häufigsten Aufgaben der CPU zählt es, Daten zu bewegen: innerhalb des Speichers, vom Speicher in ein Prozessorregister oder umgekehrt, oder von Register zu Register. Dies wird mit dem MOVE-Befehl ermöglicht (engl. to move: bewegen).

Ein Beispiel:

```
MOVE 1000,D0
```

steht für "bewege den Inhalt des Speicherplatzes mit der Adresse 1000 in das Datenregister D0".

Zunächst etwas Allgemeines zu den Befehlen des 68000: Die meisten Befehle haben zwei Operanden. Der erste folgt dem Mnemonic, der zweite wird vom ersten durch ein Komma getrennt. Der erste Operand stellt gewöhnlich die Quelle (engl. source) dar; er wird nicht verändert, sondern gibt nur an, wo die Daten hergeholt werden sollen. Der zweite Operand ist das Ziel (engl.: destination): Dort wird das Ergebnis hingeschrieben. In unserem Beispiel ist also "MOVE" das Mnemonic, "1000" die Quelle und "D0" das Ziel.

Genau genommen bewegt dieser Befehl die Inhalte der Speicherzellen 1000 und 1001 nach D0, denn die Verarbeitungsbreite ist ja 16 Bit, also ein Wort. Dabei werden die 8 Bit des Speicherplatzes 1000 als die oberen, die aus 1001 als die unteren 8 Bit interpretiert. Wenn also in Adresse 1000 (jetzt als Byte betrachtet) eine 1 steht und in 1001 eine 44, dann steht nach dem Befehl im Register D0 1 * 256 + 44 = 300 (Abb. 2.2).

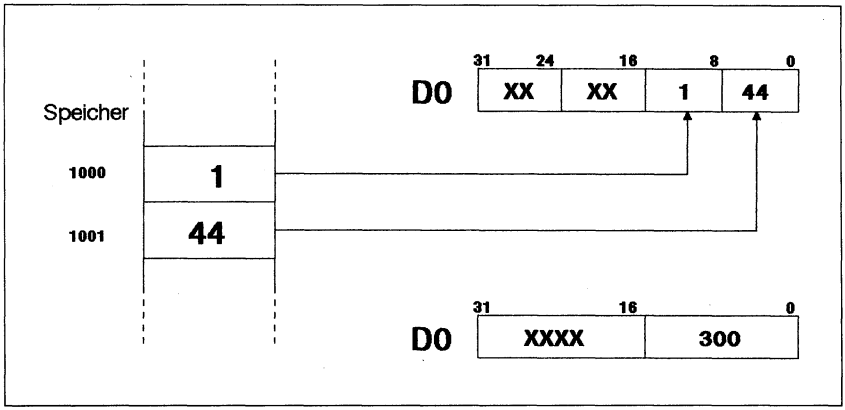


Abb. 2.2: Die Ausführung des Befehls MOVE 1000,D0

Falls Sie vorher auf Prozessoren wie 6502, 6510, Z80 oder 8086 programmiert haben, werden Sie damit vermutlich noch einige Schwierigkeiten haben: Dort werden die Bytes nämlich im Speicher genau andersherum abgelegt. Als Faustregel kann man sich merken, daß es beim MC68000 genau so ist wie bei unserer Zahlendarstellung: zuerst die hochwertigen, dann die niederwertigen Ziffern.

Vielleicht fragen Sie sich inzwischen, warum man denn den Speicher nicht gleich in 16-Bit-Einheiten gliedert, wo doch ohnehin 16 Bits auf einmal bewegt werden. Nun, das kommt daher, daß ein Computer nicht nur mit Zahlen umgeht, sondern auch viel mit Texten zu tun hat. Da ein Zeichen im ASCII-Code 8 Bits füllt, ist es sinnvoll, wenn weiterhin jedes Byte im Speicher einzeln erreicht werden kann.

Außerdem ist es ja manchmal auch durchaus sinnvoll, nur 8 Bit lange Zahlen zu verwenden. Deshalb verfügt der 68000 auch über die Möglichkeit, seine Verarbeitungsbreite auf 8 Bit zu beschränken, um somit jedes Byte im Speicher einzeln ansprechen zu können. Dies tun Sie, indem Sie an das Mnemonic einen Punkt und an den Buchstaben ein B anhängen. Wenn Sie also nur das Byte Nummer 1000 nach D0 bewegen wollten, müßten Sie schreiben

```
MOVE.B 1000,D0
```

Obwohl hier weniger Bits bewegt werden, ergibt sich kein Geschwindigkeitsvorteil, da die oberen 8 Bit der üblichen Verarbeitungsbreite von 16 Bit brachliegen. Das Ergebnis dieses Befehls ist, daß die Bits Nummer 8 bis 31 des Registers D0 (es handelt sich ja um 32-Bit-Register) unverändert bleiben. Nur die unteren 8 Bit, also Nummer 0 bis 7, werden von obigem Befehl überschrieben.

Da der 68000 durch die Adreßbildung viel mit 32-Bit-Zahlen, sogenannten Langworten (engl. longwords), zu tun hat, bietet er auch die Möglichkeit, mit einem Befehl 32 Bit auf einmal zu verarbeiten. Diesmal wird ein Punkt und der Buchstabe L (für longword) angehängt. Der Befehl

```
MOVE.L 1000,D0
```

bewegt somit 4 Bytes ab Speicherstelle 1000 in das Datenregister D0. Wie üblich kommen die höchstwertigsten Bits zuerst. Nach der Operation steht also der Inhalt von Speicherzelle 1000 in den hochwertigsten 8 Bits des Datenregisters D0, der von 1001 in den darauffolgenden 8 Bit, dann kommen die 8 Bits aus 1002 und zuletzt der Inhalt von 1003.

Für Langwort-Operationen braucht der Prozessor deutlich länger als für Wort- oder Byte-Operationen, da er einen solchen Befehl in zwei Schritten zu 16 Bit ausführen muß: Durch die 16-Bit-Architektur des Datenbusses können natürlich nicht 32 Bit auf einmal vom Speicher zur CPU übertragen werden. Allerdings geht es immer noch schneller als mit 2 MOVE-Befehlen, die jeweils ein Wort bewegen.

Übrigens gibt es auch für Wort-Operationen einen sogenannten Extender (deutsch: Anhängsel) wie ".B" oder ".L" : Es ist logischerweise ".W". Ich bin nur deshalb noch nicht darauf eingegangen, weil er bei den meisten Assemblern überflüssig ist. Sie nehmen automatisch einen Wortzugriff an, wenn an einem Mnemonic kein Extender hängt. Es steht Ihnen allerdings frei, trotzdem zu schreiben

```
MOVE.W 1000,D0
```

was völlig äquivalent ist zu

```
MOVE 1000,D0
```

Die Extender gelten nicht nur für den MOVE-Befehl, sondern ebenso für alle arithmetischen und logischen Befehle.

Bekanntlich werden bei der Assemblerprogrammierung auch gerne hexadezimale Zahlen verwendet (siehe Anhang A). Damit die hexadezimalen Zahlen nicht mit Dezimalzahlen verwechselt werden können, müssen sie irgendwie identifiziert werden. Die übliche Konvention verlangt, daß Hexadezimalzahlen mit einem vorangestellten Dollar-Zeichen ($) kenntlich gemacht werden. Da dezimal 1000 der hexadezimalen Darstellung 3E8 entspricht, kann man besagten MOVE-Befehl also auch so formulieren:

```
MOVE $3E8,D0
```

Damit wird genau der gleiche Code erzeugt.

Da oftmals auch binäre Zahlen praktisch sein können, kann jede beliebige Zahl auch als eine Folge von Nullen und Einsen eingegeben werden. Das spezielle Kennzeichen für eine Binärzahl ist das Prozentzeichen:

```
MOVE %1111101000,D0
```

Noch eine gleichwertige Darstellung, da 1111101000 die binäre Darstellung von dezimal 1000 ist.

Die Addition in Maschinensprache und das Userbyte

Natürlich ist es ist auf die Dauer langweilig, den Rechner nur Daten im Speicher herumschieben zu lassen. Daher wollen wir ihn jetzt zwei Zahlen addieren lassen. Nehmen wir an, die erste Zahl steht ab Zelle 1000 im Speicher, die zweite ab 2000. Der Inhalt von 1000 soll zu dem in Speicherzelle 2000 addiert werden. Leider ist man in der Wahl von Quelle und Ziel nicht so frei, daß man einfach schreiben könnte

```
ADD 1000,2000 nicht erlaubt!
```

Mindestens einer der beider Operanden, also Quelle oder Ziel, muß ein Register sein. Deshalb müssen wir das Ganze etwas umständlicher formulieren:

```
MOVE 1000,D0
ADD D0,2000
```

Das Mnemonic ADD (engl. to add: addieren) steht hier für den Additionsbefehl. Sie sehen, daß man nicht nur Register als Ziel verwenden kann, sondern auch Speicheradressen. Bei arithmetischen und logischen Operationen wird immer die Quelle mit dem Ziel verknüpft (also in diesem Fall addiert) und das Ergebnis im Ziel abgelegt, während die Quelle unverändert bleibt.

Obiger Befehl addiert zwei Worte – es war ja kein Extender angegeben. Manchmal will man jedoch auch nur Byte-Werte addieren. Die Befehlsfolge dazu wäre

```
MOVE.B 1000,D0
ADD.B D0,2000
```

Damit wird nur das Byte 1000 zum Byte 2000 addiert. Natürlich bezieht sich die angegebene Verarbeitungsbreite ebenso auf das Ziel wie auf die Quelle; in unserem Beispiel würde also die Speicherzelle 2001 auf jeden Fall unverändert bleiben.

Auch hier kann die Langwort-Adressierung verwendet werden. Dazu dienen folgende Befehle:

```
MOVE.L 1000,D0
ADD.L D0,2000
```

Es werden 4 Bytes ab Nummer 1000 zu 4 Bytes ab Nummer 2000 addiert.

Bevor wir uns mit weiteren Befehlen beschäftigen, ist es wichtig, sich mit der Zahlendarstellung im Computer auseinanderzusetzen.

Bei positiven Zahlen gibt es kein Problem: Die Bitfolge wird direkt als binäre Zahl behandelt. Doch wie werden negative Zahlen dargestellt? Es gibt mehrere Möglichkeiten dafür. Die vielleicht naheliegendste wäre, ein Bit zu reservieren, das nichts weiter als das Vorzeichen darstellt. Tatsächlich wird es auch bei einigen Anwendungen so gemacht. Die Methode hat nur einen Nachteil: Die Addition von so dargestellten vorzeichenbehafteten Zahlen gestaltet sich aufwendig, da das Resultat ganz vom Vorzeichen der beiden Zahlen abhängt und außerdem das Vorzeichenbit auf recht komplizierte Weise neu errechnet werden muß.

Um die Arbeit zu erleichtern, wendet man die Zweierkomplementdarstellung an. Dabei wird der darstellbare Zahlenbereich – jetzt von vorzeichenlosen Zahlen – genau in der Mitte geteilt. Die darunterliegenden Zahlen werden weiterhin als positiv betrachtet, jene darüber als negativ. Die negativen Zahlen ergeben sich, wenn man die eigentlichen (vorzeichenlosen) Zahlen von der höchsten darstellbaren Zahl plus eins abzieht und vor das Ergebnis ein Minuszeichen setzt. In unserem Beispiel müßten Sie also die entsprechende Zahl von 7 + 1 = 8 abziehen. Versuchen wir es mit der Zahl 7:

8 – 7 = 1, ergibt –1

Bei 8 Bit ergibt sich für die vorzeichenlosen Zahlen ein Bereich von 0 bis 255. Die Zahlen ab 128 haben dabei ihr negatives Äquivalent: Sie stehen für den Bereich von –128 bis –1. Dabei entspricht die 128 der –128, 129 steht für –127, und so weiter bis 255, die für –1 steht. Für 16 Bit ergeben sich entsprechend Bereiche von 0 bis 65536 oder von –32768 bis +32767, für 32 Bit von –2.147.483.648 bis +2.147.483.647 oder von 0 bis 4.294.967.295.

Ich habe noch nicht erwähnt, warum die Zweierkomplementdarstellung besser ist als die Methode mit dem Vorzeichenbit. Der Grund ist, daß Zweierkomplementzahlen bei Addition und Subtraktion auf völlig gleiche Art behandelt werden können wie vorzeichenlose Zahlen: Das Ergebnis stimmt in jedem Fall!

Beispiel: Addieren wir zwei 8-Bit-Zahlen:

```
  00101010 =  42  dezimal
+ 11110100 = –12
-------------------
(1)00011110 =  30
```

Der entstehende Übertrag hat nur bei der vorzeichenlosen Betrachtung einen Sinn und wird deshalb hier einfach ignoriert.

Jetzt noch einmal das Gleiche als vorzeichenlose Zahlen betrachtet:

```
  00101010 =  42   dezimal
+ 11110100 = 244
-------------------
 1 00011110 = 256 + 30 = 286
```

Bei diesem Beispiel muß allerdings der Übertrag ausgewertet werden, damit das Ergebnis stimmt. Sie sehen, daß Sie bei Subtraktion und Addition nicht darauf achten müssen, ob die Zahlen mit oder ohne Vorzeichen sind; es hängt ganz von der Interpretation ab, ob den Bitmustern ein Vorzeichen zugeschrieben wird oder nicht. Mehr über die Zahlendarstellung im Computer finden Sie in Anhang A.

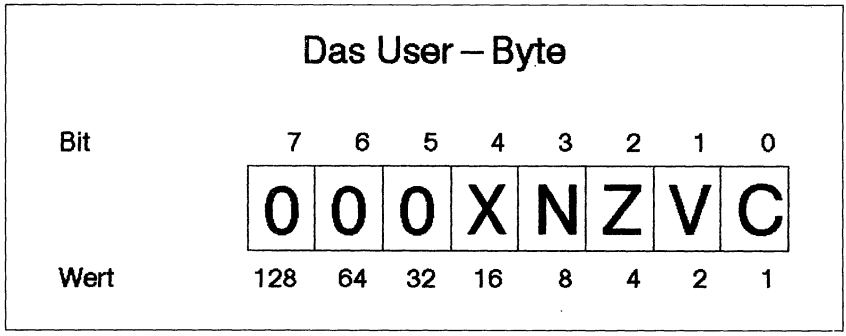


Abb. 2.3: Aufbau des CCR (Condition Code Register)

Nun zurück zur Addition: Bei dieser Operation wird nicht nur das Ziel verändert, sondern es werden auch die Statusflags neu gesetzt. Es handelt sich dabei um bestimmte Bits im weiter oben beschriebenen Statusregister, die je nach bestimmten Eigenschaften des Ergebnisses von logischen und arithmetischen Operationen gesetzt oder gelöscht werden. Die Flags befinden sich im unteren Byte des 16 Bit langen Statusregisters, dem sogenannten Userbyte oder Condition Code Register (CCR). Abbildung 2.3 zeigt die Anordnung der Flags.

Die Bedeutung der Flags:

- Das C-Bit (Carry-Flag) wird auf eins gesetzt, wenn bei einer Addition oder Subtraktion ein Übertrag bei positiven Zahlen auftritt. Bei Operationen mit Zweierkomplementzahlen braucht der Carry allein nicht beachtet zu werden. Das C-Bit wird auch von Schiebeoperationen geändert.

- Das N-Bit (Negative-Flag) wird eingeschaltet, wenn das Ergebnis in Zweierkomplementdarstellung eine negative Zahl ist. Erinnern wir uns, daß eine Zahl genau dann negativ ist, wenn das oberste Bit auf 1 gesetzt ist. Das N-Bit stimmt also mit dem obersten Bit des Ergebnisses überein.

- Das Z-Bit (Zero-Flag) wird genau dann auf 1 gesetzt, wenn das Ergebnis einer Operation Null ist.

- Das V-Bit (Overflow-Flag) wird eingeschaltet, wenn bei einer Operation mit Zweierkomplementzahlen ein Überlauf auftritt, also der Bereich der darstellbaren Zahlen verlassen wird. Solange man mit positiven Zahlen arbeitet, braucht man sich um das V-Flag nicht zu kümmern.

- Das X-Bit (Extend-Flag) hat weitgehend die gleiche Bedeutung wie das C-Bit (Carry-Flag): Beide zeigen an, ob ein Überlauf bei Addition oder Subtraktion auftrat. Der Unterschied besteht einzig und allein darin, daß das C-Flag auch von zahlreichen Operationen beeinflußt wird, die nicht arithmetischer oder logischer Natur sind, wie zum Beispiel dem MOVE-Befehl. Das X-Flag ist also "haltbarer".

Wozu dienen diese Flags nun? Sie können abgefragt werden, damit abhängig von ihrem Zustand verschiedene Aktionen durchgeführt werden. Dazu dienen die sogenannten bedingten Verzweigungen, mit denen wir uns später noch auseinandersetzen werden. Das X-Flag nimmt allerdings eine Sonderstellung ein: Es gibt Rechenbefehle, die das X-Flag direkt in ihre Rechnungen einbeziehen.

Übrigens können die Flags nicht nur von arithmetischen Befehlen beeinflußt werden, sondern sie können auch gezielt mittels MOVE beschrieben werden. Hierfür gibt es den Befehl

```
MOVE wert,CCR
```

CCR steht dabei für "Condition Code Register", also Bedingungs-Code-Register, eben das Register, das die Flags enthält. Ein 8-Bit-Wert wird in dieses Register geschrieben, wobei die oberen 3 Bits keine Bedeutung haben und ignoriert werden. Wenn beispielsweise nur das X-Bit gesetzt und alle anderen Flags gelöscht werden sollen, schreibt man

```
MOVE #%00010000,CCR
```

Betrachten Sie dazu noch einmal Abbildung 2.3. Erinnern wir uns, daß das Prozent-Zeichen eine Binärzahl einleitet.

Hier begegnet uns gleich etwas Neues: die unmittelbare Adressierung. Das bedeutet nichts weiter, als daß ein Operand nicht erst von woanders geholt werden muß, sondern gleich im Befehl enthalten ist. In unserem Beispiel bezeichnet das Doppelkreuz (#), daß die folgende Zahl nicht als Adresse eines Speicherplatzes gemeint ist, sondern einfach für sich selbst steht. Es soll nicht der Inhalt von Speicherplatz 16 (gleich 00010000 binär) ins CCR bewegt werden, sondern die Zahl 16 selbst. Natürlich ist die unmittelbare Adressierung nur bei der Quelle sinnvoll und erlaubt.

Stürzen wir uns gleich wieder in die Praxis: Nehmen wir an, Sie brauchen für eine bestimmte Anwendung besonders große Zahlen, so daß 32 Bit zu deren Darstellung nicht mehr ausreichen. Deshalb wollen Sie 64-Bit-Zahlen verwenden. (Zugegeben, das ist etwas an den Haaren herbeigezogen, da man ja mit 32 Bit auch schon Zahlen bis über 4 Milliarden darstellen kann. Aber dies ist nun einmal die einfachste Methode, den Gebrauch des X-Flags zu zeigen.) Dabei stellt sich nun das Problem, wie man denn mit solchen Zahlen rechnet, da ja für den Umgang mit so großen Zahlen keine speziellen Befehle vorgesehen sind.

Zum Beispiel die Addition: Die erste Idee wäre, die Operation in zwei Schritten zu 32 Bit auszuführen. Problematisch ist dabei nur, daß ja aus der Addition der unteren 32 Bit ein Übertrag auftreten kann, der korrekt behandelt werden muß. Hier hilft uns nun das X-Flag weiter. Nehmen wir an, eine 64-Bit-Zahl ab 1000 soll zu einer weiteren ab 2000 addiert werden:

```
MOVE.L 1004,D0
ADD.L D0,2004
MOVE.L 1000,D0
MOVE.L 2000,D1
ADDX.L D0,D1
MOVE.L D1,2000
```

Es wird dabei angenommen, daß die Zahlen im 68000er-üblichen Format angeordnet werden, also zuerst die oberen, danach die unteren Bits.

Zuerst werden also wie üblich die beiden unteren Hälften der Zahlen addiert. Mit dem Befehl ADD werden gleichzeitig alle Flags entsprechend gesetzt. Dann folgt ADDX, ein bisher unbekannter Befehl: Dieser führt eine Addition durch, bei der der Übertrag einer eventuell vorhergehenden Addition richtig behandelt wird. Genauer gesagt, ADDX wirkt wie ADD, nur daß zum Ergebnis noch Eins addiert wird, wenn das X-Flag gesetzt ist. Andernfalls wird das Ergebnis der Addition unverändert gelassen. Man kann es auch so ausdrücken:

```
ADDX: Ziel := Ziel + Quelle + X-Flag
```

Das Zeichen ":=" steht hier für die Zuweisung, wie es etwa in der Programmiersprache Pascal üblich ist.

Vielleicht wundern Sie sich, daß die ADDX-Operation auf so umständliche Weise ausgeführt werden muß. Der Grund ist, daß ADDX in seinen Adressierungsarten sehr beschränkt ist; hier kommt deshalb nur die Form "Register zu Register" in Frage. Deshalb müssen beide Operanden zuerst in Register geladen und das Ergebnis in den Zieloperanden zurückgeschrieben werden.

Der ADDX-Befehl addiert also die oberen 32 Bit der beiden Zahlen unter Beachtung des Übertrags der unteren 32 Bit. Die Befehle ADD und ADDX sind genau für diese Verwendung bestimmt. Natürlich braucht man sich nicht auf 64-Bit-Zahlen zu beschränken, denn der ADDX-Befehl setzt ja seinerseits wieder das X-Flag. Indem also weitere ADDX-Befehle angehängt werden, können beliebig lange Zahlen addiert werden.

Eine Variante des ADD-Befehles soll nicht unerwähnt bleiben: ADDQ. Dieses Kürzel steht für "Add Quick" und führt die schnelle Addition einer kleinen Konstanten zum Zieloperanden aus. Die Konstante darf nur 3 Bit lang sein, wobei allerdings die Bitkombination 000 als 8 interpretiert wird. Somit kann der Wert von 1 bis 8 reichen. Der Zieloperand kann in Byte-, Wort- und Langwortbreite bearbeitet werden. So können Sie anstatt

```
ADD.L #2,D0
```

also besser folgendes schreiben:

```
ADDQ.L #2,D0
```

Letztere Variante braucht nicht nur weniger Speicherplatz, sondern wird auch schneller ausgeführt.

Die Substraktion in Maschinensprache

Das Mnemonik für die Substraktion ist logischerweise SUB. Um zwei Worte voneinander abzuziehen, schreiben Sie also:

```
MOVE 1000,D0
SUB D0,2000
```

Achtung! Die Operanden liegen hier in einer Reihenfolge, die genau andersherum ist, als man es gewohnt ist: Die Quelle wird hier vom Ziel abgezogen,

also der erste Operand vom zweiten. Das kommt daher, daß der Befehl als eine Kurzschreibweise für "Ziehe den Inhalt von D0 vom Inhalt von 2000 ab" gedacht ist. Das Ergebnis der Subtraktion wird wie üblich im Ziel abgelegt, also in 2000. Natürlich gibt es auch den SUB-Befehl in verschiedenen Verarbeitungsbreiten, also für Bytes:

```
MOVE.B 1000,D0
SUB.B  D0,2000
```

oder für Langworte:

```
MOVE.L 1000,D0
SUB.L  D0,2000
```

Der SUB-Befehl setzt die Statusflags auf die gleiche Weise wie der ADD-Befehl. Deshalb können auch unsere 64-Bit-Zahlen voneinander abgezogen werden:

```
MOVE.L 1004,D0
SUB.L  D0,2004
MOVE.L 1000,D0
MOVE.L 2000,D1
SUBX.L D0,D1
MOVE.L D1,2000
```

Beim SUB-Befehl wird das X-Flag genau dann auf 1 gesetzt, wenn ein Borgen des obersten Bits erforderlich ist. Der SUBX-Befehl berücksichtigt genau diesen Fall: Wenn das X-Flag gesetzt ist, wird das Ergebnis der Subtraktion um eins vermindert. In formaler Schreibweise:

```
SUBX: Ziel := Ziel - Quelle - X-Flag
```

Für den SUBX-Befehl gilt genauso wie für ADDX – wenigstens darin liegt Konsequenz –, daß als Operanden entweder nur Datenregister oder nur Speicheradressen verwendet werden dürfen, aber keine Mischformen.

Auch hier gibt es wieder die Quick-Variante SUBQ, die eine 3-Bit-Konstante von 1 bis 8 vom Zieloperanden abzieht.

Die Multiplikation

Als nächstes wollen wir uns mit der Multiplikation befassen. Der 68000 kann immerhin mit einem einzigen Befehl zwei Zahlen multiplizieren, was für eine

CPU überhaupt nicht selbstverständlich ist, denn auf den meisten Prozessoren muß dafür extra ein Programm geschrieben werden, das die Multiplikation auf die Addition zurückführt.

Wir wollen uns ab jetzt angewöhnen, in den Beispielprogrammen keine direkten Speicheradressen zu verwenden, sondern Symbole. Es wird angenommen, daß am Anfang eines Assemblerprogramms diese Symbole definiert werden, etwa in der Form

```
OP1 EQU 1000
OP2 EQU 2000
```

Hierbei handelt es sich um eine Anweisung an den Assembler, ein Symbol mit einer Konstante oder einer Speicheradresse zu identifizieren. Immer wenn fortan das Symbol benutzt wird, setzt der Assembler dafür den ihm nunmehr zugewiesenen Wert ein. Die Verwendung von Symbolen macht die Programme übersichtlicher, da die Namen zur Dokumentation beitragen.

Jetzt also zurück zur Multiplikation: Den Befehl zur Multiplikation gibt es in wesentlich weniger Ausführungen als die Befehle ADD und SUB. Als Operanden können nur zwei Worte verwendet werden, und das Ziel muß in jedem Fall ein Datenregister sein. Das Ergebnis ist dabei 32 Bit lang, denn das Produkt von zwei 16-Bit-Zahlen kann ja höchstens 32 Bit umfassen. Das sieht etwa so aus:

```
MOVE OP2,D0
MULU OP1,D0
MOVE.L D0,ERG
```

Das Mnemonik MULU steht für "multiply unsigned", also multipliziere vorzeichenlos. Bei der Multiplikation muß im Gegensatz zur Subtraktion und Addition zwischen vorzeichenlosen und vorzeichenbehafteten Zahlen unterschieden werden. Deshalb gibt es auch noch eine Variante für Zweierkomplementzahlen:

```
MOVE OP2,D0
MULS OP1,D0
MOVE.L D0,ERG
```

Übrigens gehören MULU und MULS zu den Befehlen, deren Ausführung am längsten dauert: Eine Multiplikation nimmt ungefähr zehnmal so viel Zeit in Anspruch wie eine Addition, denn im Gegensatz zu letzterer ist sie nicht einfach mit ein paar Logikelementen zu realisieren.

Da die Multiplikation von Bytes nicht implementiert ist, müssen auch hierfür die Befehle MULU und MULS verwendet werden. Vorher müssen die Bytes allerdings auf Worte erweitert werden.

Für vorzeichenlose Zahlen sieht das folgendermaßen aus:

```
CLR D0
CLR D1
MOVE.B OP1,D0
MOVE.B OP2,D1
MULU D0,D1
MOVE D1,ERG
```

Hier begegnet uns ein neuer Befehl: CLR, was für "CLeaR" (deutsch: löschen) steht. Dieser Befehl löscht den angegebenen Operanden, schreibt also eine 0 hinein. Auch diesen Befehl gibt es in Byte-, Wort- und Langwortbreite. Übrigens hätte man statt "CLR D0" ebensogut schreiben können

```
MOVE #0,D0
```

Noch einmal zurück zu unserer Byte-Multiplikation: Zunächst werden die Register D0 und D1 gelöscht (Wortbreite). Dann werden die beiden Byte-Operanden hineinkopiert. Bedenken Sie dabei, daß dadurch nur Bits 0 bis 7 beeinflußt werden, während Bits 8 bis 15 weiterhin auf 0 bleiben. Die Operanden sind also korrekt erweitert worden. Dann erfolgt die Multiplikation und die Abspeicherung des Ergebnisses. Beachten Sie, daß das Ergebnis nur in Wortbreite bewegt wird, denn das Produkt von zwei 8-Bit-Zahlen kann höchstens 16 Bit umfassen.

Natürlich kann die Multiplikation auch mit vorzeichenbehafteten Bytes durchgeführt werden:

```
MOVE.B OP1,D0
MOVE.B OP2,D1
EXT D0
EXT D1
MULS D0,D1
MOVE D1,ERG
```

Zuerst werden die beiden Operanden in die Register D0 und D1 kopiert. Mit dem neuen Befehl EXT, der für EXTend (engl. erweitern) steht, werden die Bytes vorzeichenrichtig auf Worte erweitert. Erinnern wir uns, daß das Vorzeichen einer Zahl gleichwertig mit deren oberstem Bit ist. Ist dieses Bit 1, so ist sie negativ, sonst positiv. Der Befehl EXT überträgt nun den Inhalt von Bit 7 in die Bits 8 bis 15. Wenn die Zahl also positiv ist, so werden dort lauter Nullen hineingeschrieben, bei einer negativen Zahl Einsen. EXT gibt es in Wort- und Langwortbreite. Bei letzterer Variante wird ein Wort vorzeichenrichtig zu einem Langwort erweitert.

Nach den EXT-Befehlen folgt die Multiplikation mit Vorzeichen und die Abspeicherung des Ergebnisses. Auch hier sind nur 16 Bit des Ergebnisses relevant.

Will man zwei 32-Bit-Zahlen multiplizieren, so wird das Ganze schon schwieriger, denn auch diese Operation muß auf 16-Bit-Multiplikationen zurückgeführt werden. Um das Prinzip zu verstehen, überlegen wir uns zunächst einmal, wie eine schriftliche Multiplikation durchgeführt wird:

```
42 * 87
-------
    14   = 7 * 2, Stellenwert   1
   28    = 7 * 4, Stellenwert  10
  16     = 8 * 2, Stellenwert  10
 32      = 8 * 4, Stellenwert 100
-------
 3654
```

Das Prinzip ist also, daß man die Multiplikation von großen Zahlen in die Multiplikation von Zahlen zwischen 0 und 9 aufteilt und so die gesamte Multiplikation vereinfacht. Bei der Multiplikation von 32-Bit-Zahlen geschieht im Prinzip nichts anderes, nur daß das Einmaleins des Rechners eben nicht nur bis 9* 9, sondern bis 65535 * 65535 reicht (Abb. 2.4).

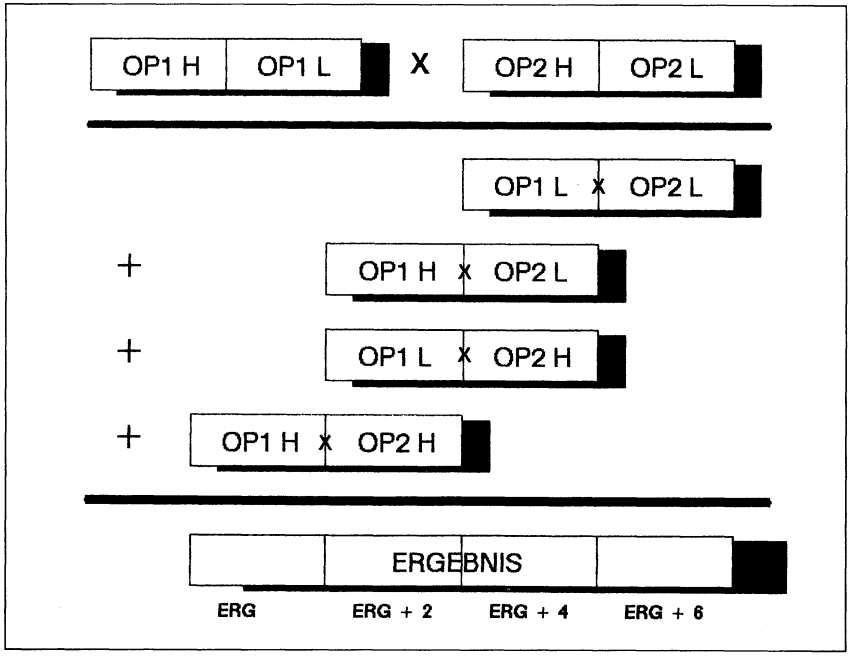


Abb. 2.4: Die Langwortmultiplikation

Nun muß das Ganze nur noch formuliert werden:

```
MOVE OP1+2,D0          * OP1 Low x OP2 Low
MULU OP2+2,D0          *
MOVE.L D0,ERG+4        * Stellenwert 0
MOVE OP1,D0            * OP1 High x OP2 High
MULU OP2,D0            *
MOVE.L D0,ERG          * Stellenwert 2 hoch 32
MOVE OP1+2,D0          * OP1 Low x OP2 High
MULU OP2,D0            *
ADD.L D0,ERG+2         * Stellenwert 2 hoch 16
MOVE ERG,D1            * oberstes Wort holen
CLR D2                 * für ADDX löschen
ADDX D2,D1             * Übertrag addieren
MOVE OP1,D0            * OP1 High x OP2 Low
MULU OP2+2,D0          *
ADD.L D0,ERG+2         * Stellenwert 2 hoch 16
ADDX D2,D1             * Übertrag!
MOVE.L D1,ERG          * oberstes Wort schreiben
```

Hier haben wir es erstmals mit einer etwas komplizierteren Anweisungsfolge zu tun. Sie ist jedoch nicht schwer zu verstehen, da sie völlig geradlinig abläuft.

Die ersten vier Befehle multiplizieren die niederwertigen 16 Bits der beiden Operanden (OP1 + 2, OP2 + 2) miteinander und legen das Ergebnis in den Bytes 4 bis 7 des Gesamtergebnisses ERG ab. Beachten Sie, daß auch hier die beiden Operanden vom Typ Wort sind, das Ergebnis hingegen ein Langwort ist.

Die nächsten drei Befehle bewirken das gleiche mit den oberen 16 Bits der Operanden. Hier wird das Resultat in den Bytes 0 bis 3 des Gesamtergebnisses abgelegt. Es ist wichtig, daß dabei keine Überschneidung entsteht, sonst könnte das Ergebnis nicht einfach mit dem MOVE-Befehl hineingeschrieben werden. Deshalb wurde auch genau diese Reihenfolge für die Berechnung der vier Teilergebnisse verwendet.

Nun zu den restlichen zwei Teilergebnissen: Zunächst wird das Produkt des niederwertigen Worts des ersten Operanden (OP1 + 2) mit dem hochwertigen Wort des zweiten (OP2) berechnet. Dieses muß jetzt "in der Mitte", also von Byte 2 bis 5 des Gesamtergebnisses, addiert werden. Hier entsteht schon eine Falle, die leicht übersehen werden kann: Bei der Addition von zwei Langworten kann ein Übertrag von einem Bit entstehen, der dem Gesamtergebnis der Bytes 0 und 1 hinzugezählt werden muß. Da es in der Maschinensprache des 68000 keinen Befehl gibt, um direkt das Übertragsbit einem Operanden hinzuzuzählen, behilft man sich mit der folgenden Befehlsfolge:

```
MOVE   ERG,D1
CLR    D2
ADDX   D2,ERG
```

Es wird also eine Null mit Übertrag addiert. Wir müssen hier diesen Umweg benutzen statt "ADDX #0,ERG", da ADDX für uns nur in der Adressierungsart "Register zu Register" in Frage kommt. Erinnern wir uns: Das X-Flag wird dann angeschaltet, wenn ein vorhergehender ADD-Befehl einen Übertrag produziert. In diesem Fall fügt ein darauffolgender ADDX-Befehl zur Summe der beiden Operanden Eins hinzu. Obige Befehlsfolge bewirkt also genau das, was wir wollen.

Die nächste Befehlsgruppe multipliziert die oberen zwei Bytes des ersten Operanden (OP1) mit den unteren des zweiten (OP2 + 2). Sie ist also genau komplementär zu der vorherigen Befehlsgruppe. Auch hier muß das Ergebnis zu den Bytes 2 bis 5 des Gesamtergebnisses addiert werden, und es folgt wieder der gleiche Vorgang mit dem Übertrag und schließlich das Zurückschreiben des Wertes aus D1 nach ERG.

Diese Art der Multiplikation kann auch auf Zahlen angewandt werden, die länger als 32 Bit sind. Es müssen nur alle Teilergebnisse mit der richtigen Wertigkeit addiert und die Überträge beachtet werden. Wenn Sie wollen, versuchen Sie einmal, die nötigen Teilergebnisse für eine 64-Bit-Multiplikation zu bestimmen.

Die Division

Nun fehlt uns nur noch die vierte Grundrechenart: Die Division. Auch dafür gibt es einen Maschinensprachebefehl. Da die Division komplementär zur Multiplikation ist, werden durch sie nicht wie bei der Multiplikation zwei Wort-Operanden zu einem Langwort verkettet, sondern ein Langwort wird durch ein Wort geteilt, wobei das Ergebnis seinerseits ein Wort ist. Für die vorzeichenlose Division sieht dies dann etwa folgendermaßen aus:

```
MOVE.L OP1,D0
DIVU   OP2,D0
MOVE   D0,ERG
```

DIVU steht hier für "DIVide Unsigned", also teile vorzeichenlos. Auch beim DIVU-Befehl darf – genau wie bei den Multiplikationsbefehlen – nur ein Datenregister als Ziel angegeben werden. Beachten Sie, daß auch hier die Regel gilt, daß der Quelloperand mit dem Zieloperanden verkettet wird und das Er-

gebnis im Zieloperanden abgelegt wird. Die Operanden werden also auch hier, ebenso wie schon beim SUB-Befehl, genau andersherum geschrieben, als man es gewohnt ist.

Auch DIVU hat sein Pendant für die Division mit Vorzeichen. Es heißt DIVS und steht für "divide signed". Es wird genauso angewandt wie DIVU:

```
MOVE.L OP1,D0
DIVS OP2,D0
MOVE D0,ERG
```

Leider kann die Division einer ganzen Zahl durch eine andere nicht so sauber behandelt werden wie die Multiplikation. Zunächst einmal gilt, daß das Ergebnis der Division ebenfalls nur ganzzahlig sein kann und deshalb immer abgerundet wird. Ein Problem stellt der Fall dar, daß das Ergebnis der Division eines Langwortes durch ein Wort nicht unbedingt in einem Wort darstellbar ist, etwa wenn eine große Zahl durch 1 dividiert wird. In diesem Fall ist das Ergebnis undefiniert, und das Overflow-Flag wird auf 1 gesetzt. Wenn also dieser Fall auftreten kann, sollten Sie immer den Zustand des V-Flags überprüfen.

Was schon Generationen von Mathematikern Kopfzerbrechen bereitete, muß auch hier eine Sonderbehandlung erfahren: die Division durch Null. Es liegt normalerweise in der Verantwortung des Programmierers, es nicht dazu kommen zu lassen. Sollte es doch einmal geschehen, dann löst der Prozessor eine Art Ausnahmezustand aus, mit dessen Varianten wir uns später noch ausführlich befassen werden. Wenn Sie sichergehen wollen, sollten Sie also vor der Division den zweiten Operanden testen.

Will man kleinere Zahlen teilen, zum Beispiel ein Wort durch ein Byte, so muß man die Operanden wieder erweitern. Für eine vorzeichenlose Division sieht das so aus:

```
CLR.L D0
MOVE OP1,D0
CLR D1
MOVE.B OP2,D1
DIVU D1,D0
MOVE D0,ERG
```

Zunächst wird das Register D0 gelöscht und daraufhin der erste Operand in Wortbreite hineingeschrieben, wobei die oberen 16 Bit nicht verändert werden und somit den Wert Null behalten. Aus dem Wort ist also ein Langwort geworden. Mit dem zweiten Operanden wird im Prinzip genauso verfahren, nur daß hier ein Byte in ein Wort verwandelt wird. Schließlich wird die Division durchgeführt und das Ergebnis abgespeichert.

Nun noch einmal der gleiche Vorgang für die Division mit Vorzeichen:

```
MOVE OP1,D0
EXT.L D0
MOVE.B OP2,D1
EXT D1
DIVS D1,D0
MOVE D0,ERG
```

Hier wird zunächst der erste Operand in das Register D0 bewegt und dort mit dem EXT-Befehl vorzeichenrichtig auf Langwortbreite erweitert. Danach geschieht das gleiche mit dem zweiten Operand, der allerdings von Byte- auf Wortbreite erweitert wird. Dann erfolgt die übliche Division.

Bei den Befehlen DIVU und DIVS gibt es noch eine Besonderheit: Es wird nicht nur das Ergebnis der Division berechnet, sondern auch der Rest. Dieser wird in den oberen 16 Bits des Zielregisters abgelegt. Nun fragen Sie sich vielleicht, wie man dort herankommt, da man mit den Verarbeitungsbreiten Wort und Byte ja nur an die unteren Bits kommt. Nun, dafür gibt es den SWAP-Befehl. Er kann nur auf ein Datenregister angewendet werden und vertauscht die oberen 16 Bits mit den unteren 16 Bits. Wenn Sie also bei einer Division auch den Rest ermitteln wollen, müßte das so aussehen:

```
MOVE.L OP1,D0
DIVU OP2,D0
MOVE D0,ERG
SWAP D0
MOVE D0,REST
```

Zuerst wird also wie üblich die Division durchgeführt und das Ergebnis abgespeichert. Darauf folgt der SWAP-Befehl, der die untere Hälfte des Langwortes D0 mit der oberen Hälfte vertauscht. Die neue untere Hälfte wird dann als Rest in den Speicher bewegt.

SHIFT und ROTATE

Manchmal ist es durchaus sinnvoll, die Bits einer Speicherzelle um einige Stellen zu verschieben. Dafür ist eine ganze Gruppe von Befehlen zuständig, die alle möglichen Fälle behandeln.

SHIFT-Befehle

Shift steht für Verschieben. Die Wirkung eines solchen Befehls ist die, daß alle Bits um (zunächst) eine Stelle nach rechts oder links verschoben werden. Dabei rückt eine Null nach, und das herausgeschobene Bit landet im X-Flag und Carry-Flag (Abb. 2.5).

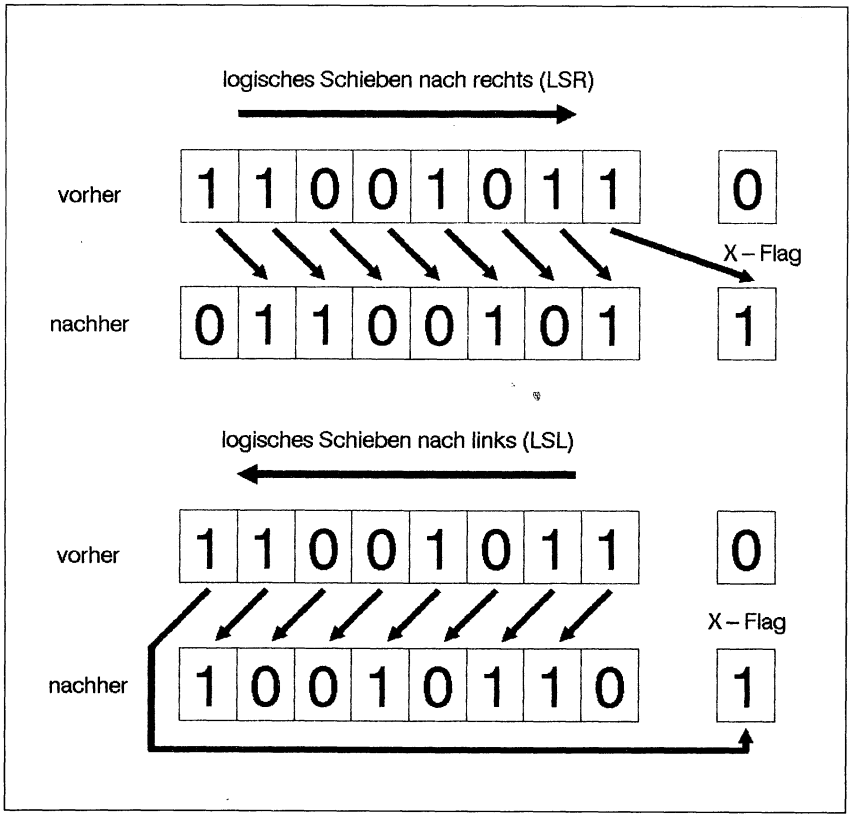


Abb. 2.5: Logisches Schieben nach rechts und links (LSR/LSL)

So gibt es zunächst einmal zwei Befehle, die diese Aufgabe übernehmen: LSL (Logical Shift Left, logisches Verschieben nach links) und LSR (Logical Shift Right, logisches Verschieben nach rechts). "Links" steht hier für ein Verschieben auf die höherwertigen Bits zu, "rechts" auf die niederwertigen, entsprechend der Reihenfolge, wie man die Ziffern einer Zahl auf ein Blatt Papier schreiben würde.

Allerdings sind noch andere Verschiebebefehle notwendig: Wenn mit Zahlen im Zweierkomplement gearbeitet wird, wird ja irgend ein Bit ins Vorzeichen hineingeschoben und somit unter Umständen das Vorzeichen der Zahl verändert. Um dies zu vermeiden, hat man auch SHIFT-Befehle für vorzeichenbehaftete Zahlen implementiert: Der ASR-Befehl (Arithmetic Shift Right, arithmetisches Verschieben nach rechts) läßt einfach das Vorzeichen unverändert

und schiebt nur die restlichen Bits um eine Stelle nach rechts, wobei das Vorzeichen auch in das ihm folgende Bit übertragen wird (Abb. 2.6). Der ASL-Befehl (Arithmetic Shift Left, arithmetisches Schieben nach links) ist jedoch genaugenommen nur eine Attrappe, denn seine Wirkungsweise ist genau identisch mit der von LSL.

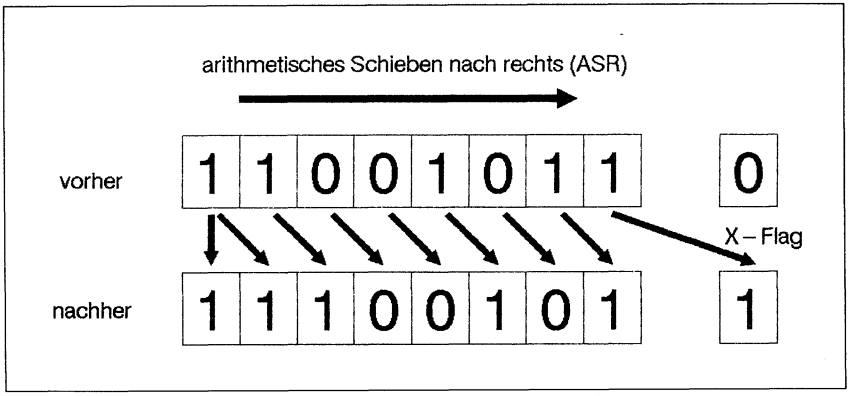


Abb. 2.6: Arithmetsches Schieben nach rechts (ASR)

Wenn Sie eine dezimale Zahl mit 100 multiplizieren wollen, so werden Sie nicht erst zum Taschenrechner oder zu Bleistift und Papier greifen, sondern einfach zwei Nullen anhängen. Genau das gleiche macht der Prozessor bei den Verschiebeoperationen, nur daß er eben im Binärsystem rechnet. So entspricht das Verschieben um eine Stelle nach links (also das Anhängen eines Nullbits) einer Multiplikation mit 2, das Verschieben nach rechts einer Division durch 2. Bei letzterem landet der Rest im Carry-Bit, denn der Prozessor kennt nun einmal keine Nachkommastellen. Eine solche Verschiebeoperation ist für den Prozessor wesentlich einfacher und schneller auszuführen als eine echte Multiplikation mittels MULU oder MULS.

Hieraus ergibt sich nun auch, warum ASL und LSL identisch sind: Damit bei ASL das Vorzeichen verändert wird, ist es notwendig, daß das Bit rechts neben dem Vorzeichenbit ungleich dem Vorzeichenbit selbst ist. In diesem Fall muß es sich ohnehin um eine recht große (oder, wenn sie negativ ist, sehr kleine) Zahl gehandelt haben, deren Multiplikation mit 2 ein Ergebnis liefert, das außerhalb des darstellbaren Bereichs liegt. Da das reelle Ergebnis also sowieso falsch wäre, spielt es auch keine Rolle mehr, ob das Vorzeichenbit verändert wird. Beispielsweise für 8 Bit müßte eine solche Zahl im Bereich von –128 bis

–65 oder +64 bis +127 liegen, deren Multiplikation mit zwei eine Zahl von –256 bis –130 oder +128 bis +254 liefert, was die Grenzen einer 8-Bit-Zweierkomplementzahl sprengt.

Jetzt zur Praxis: Nehmen wir an, Sie wollen die Variable OP1 vorzeichenlos mit zwei multiplizieren. Sofern es sich um ein Wort handelt, schreiben Sie einfach

```
ASL  OP1
```

Wie Sie sehen, kann ASL (und ebenso die anderen Schiebebefehle) direkt auf eine Speicherzelle angewendet werden. In diesem Fall ist nur Wortlänge erlaubt. Es gibt jedoch noch eine zweite Variante, bei der sich der Zieloperand in einem Datenregister befinden muß. In diesem Fall kann man nicht nur in Langwort- und Byte-Breite verschieben, sondern auch um mehrere Stellen auf einmal. Dafür ist natürlich noch ein zweiter Operand notwendig, der die Anzahl der Stellen angibt, um die verschoben werden soll. Wenn ein Operand beispielsweise mit 8 multipliziert werden soll, muß er um 3 Stellen nach links verschoben werden (2 * 2 * 2 = 8):

```
MOVE  OP1,D0
ASL   #3,D0
MOVE  D0,OP1
```

Wichtig für das Verschieben um mehrere Stellen ist folgendes: Die Wirkung ist die gleiche, als ob mehrere Male hintereinander um eine Stelle verschoben wird. Das heißt, für ASL werden von rechts Nullen nachgeschoben, während nur das zuletzt hinausgeschobene Bit im Carry landet. Übrigens darf der direkte Zähler nur von 1 bis 8 reichen. Ein Verschieben um mehr Stellen ist auch möglich, doch muß dann der Quelloperand ein Datenregister sein, das die Anzahl der Stellen enthält. Dabei können Werte bis zu 64 angegeben werden. Das ist eigentlich schon zuviel des Guten, denn selbst von einem Langwort bleiben in jedem Fall nur noch Nullen übrig, wenn es um mehr als 31 Stellen verschoben wird.

Wenn ein Langwortoperand um 16 Stellen verschoben werden soll, sieht das so aus:

```
MOVE.L OP1,D0
MOVE   #16,D1
ASL.L  D1,D0
MOVE.L D0,OP1
```

Auch hier können größere Einheiten als 32 Bit auf einmal verschoben werden. Dabei hilft uns wieder das Extend-Flag, denn es gibt auch Varianten der Ver-

schiebebefehle, die dieses Bit mit einbeziehen. So steht ROXL für "ROtate Left through X-Flag", also "rotiere nach links durch das Extend-Flag", und ROXR für die entsprechende Rotation nach rechts.

In der ersten Verwendungsart, in der ein Wort im Speicher um eine Stelle verschoben wird, gleicht ROXL dem Befehl ASL bis auf den Umstand, daß statt einer Null von rechts der Inhalt des Extend-Flags nachgeschoben wird (Abb. 2.7).

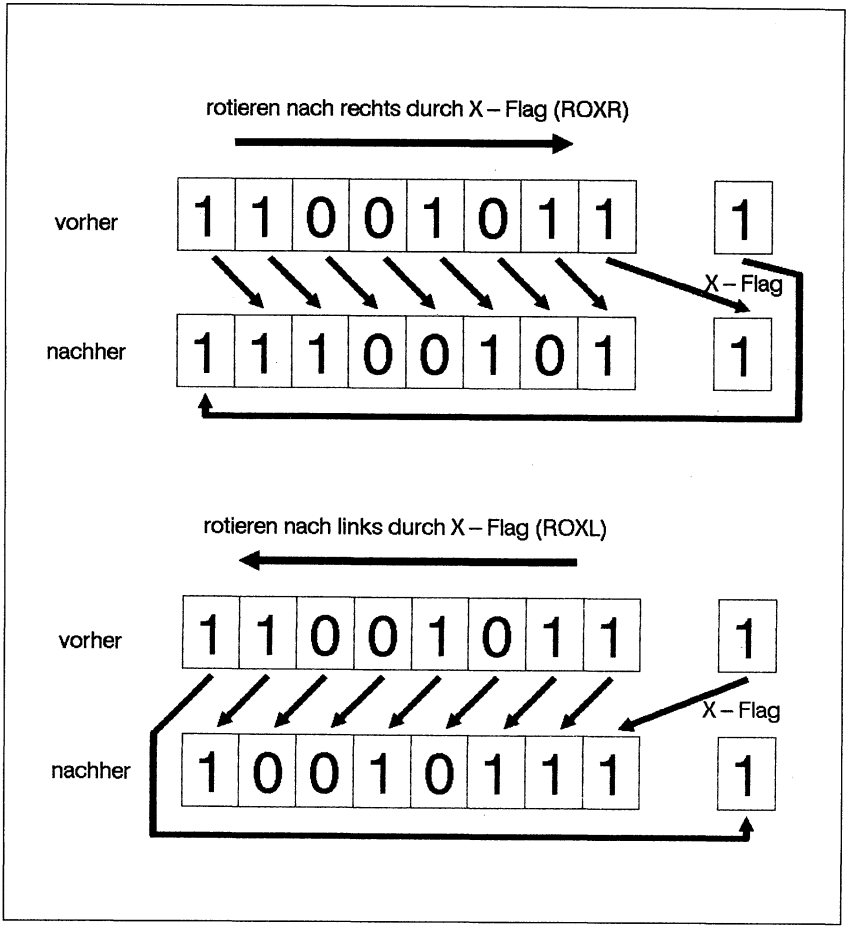


Abb.2.7: Rotieren nach rechts und links durch X-Flag (ROXR/ROXL)

Um eine 64-Bit-Zahl nach links zu schieben, müßten Sie also schreiben

```
ASL OP1+6
ROXL OP1+4
ROXL OP1+2
ROXL OP1
```

Hier machen wir uns den Umstand zunutze, daß ROXL seinerseits das herausgeschobene Bit im X-Flag plaziert und daher beliebig oft aneinandergereiht werden kann. Beachten Sie, daß mit den niederwertigsten Bytes (OP1 + 6) begonnen wurde. Dies ist notwendig, da ja die höherwertigeren Bytes auf das herausgeschobene Bit ihrer Vorgänger angewiesen sind.

Auch in der oben beschriebenen zweiten Variante, bei der ein Datenregister um mehrere Stellen auf einmal verschoben wird, können die ROXL- und ROXR-Befehle verwendet werden. Dies ist jedoch nur selten sinnvoll, da sich die CPU im Extend-Flag ja nur ein Bit "merken" kann. Sie könnten also eine 64-Bit-Zahl nicht korrekt um drei Stellen verschieben, indem Sie einfach ASL und ROXL benutzen, da ja nur eines der drei hinausgeschobenen Bits im höherwertigen Teil der Zahl ankommt. Hier behilft man sich am einfachsten, indem man drei Verschiebungen um eine Stelle vornimmt. Wegen dieser Probleme wollen wir hier nur die Verschiebung um eine Stelle benutzen. Unter Verwendung der Datenregister hätte man obige Befehlssequenz auch so formulieren können:

```
MOVE.L OP1+4,D0
ASL.L #1,D0
MOVE.L D0,OP1+4
MOVE.L OP1,D0
ROXL.L #1,D0
MOVE.L D0,OP1
```

Hierbei wird der Operand nicht in 4 Einheiten zu 2 Byte, sondern in 2 Einheiten zu 4 Byte rotiert – was auf das gleiche hinausläuft.

Übrigens gilt alles, was über den ASL-Befehl gesagt wurde, ebenso für ASR, LSL und LSR – schließlich ist der 68000 Prozessor allgemein so aufgebaut, daß ähnliche Befehle auch in ähnlichen Varianten existieren. Nur eines sollten Sie beachten, wenn Sie nach rechts schieben: Im Gegensatz zum Verschieben nach links müssen sie dabei mit den höchstwertigen Bytes beginnen, denn diesmal müssen die herausgeschobenen Bits in die nächstniedrigen Bytes hineingeschoben werden. Um eine 64-Bit-Zahl ein Bit nach rechts zu schieben (also durch zwei zu teilen), schreibt man:

```
ASR OP1
ROXR OP1+2
ROXR OP1+4
ROXR OP1+6
```

Die ROTATE-Befehle

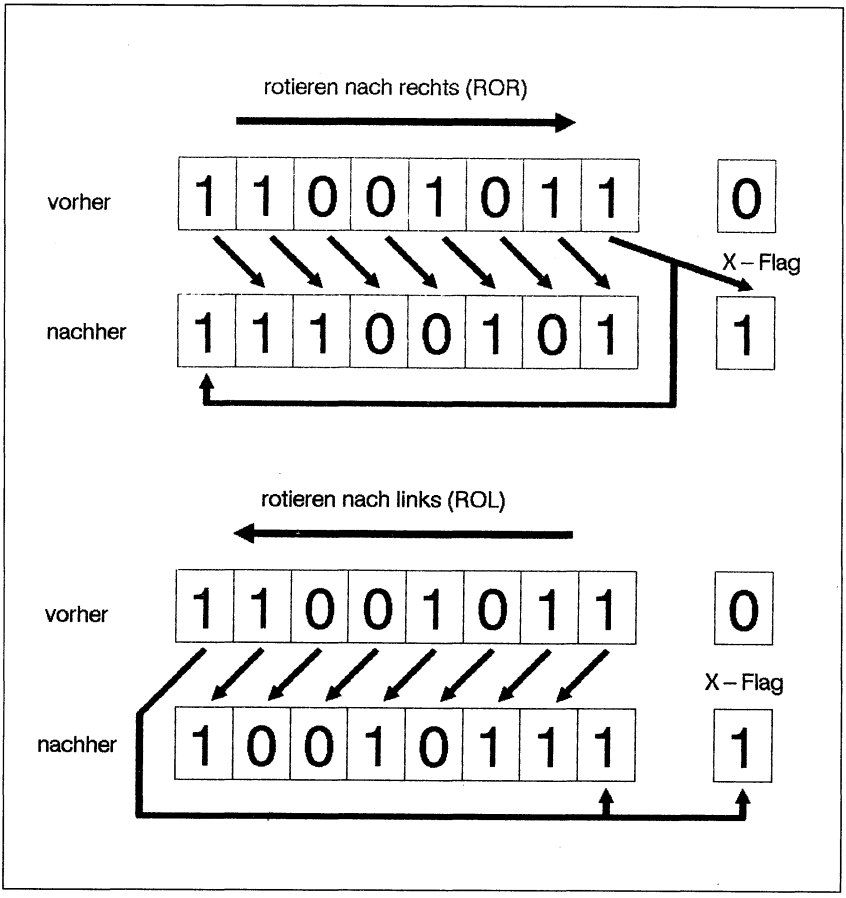


Abb. 2.8: Rotieren nach rechts und links (ROR/ROL)

Der Vollständigkeit halber sollen hier auch die Befehle ROL (ROtate Left, rotiere nach links) und ROR (ROtate Right, rotiere nach rechts) erwähnt werden, auch wenn sie nur sehr selten benutzt werden. Ihre Wirkungsweise ist der von LSL und LSR ähnlich, nur daß das herausgeschobene Bit nicht nur in Carry und X-Flag erscheint, sondern auch an der anderen Seite des Operanden

wieder hineingeschoben wird (Abb. 2.8). Auch hier gibt es zwei Varianten: Bei der ersten muß der Operand im Speicher liegen, ist auf Wortlänge beschränkt und darf nur um eine Stelle verschoben werden, während die zweite nur auf Datenregister angewendet werden kann, aber dafür eine Verschiebung um bis zu 64 Stellen erlaubt.

Wenn das Register D0 den hexadezimalen Wert 4321 enthält und der Befehl

```
ROL.W  #4,D0
```

ausgeführt wird, so befindet sich danach in D0 die Zahl 3214, denn die Verschiebung um vier Binärstellen entspricht einer Verschiebung um eine hexadezimale Stelle. Die vier Bits der 4 sind also nach links hinausgeschoben worden und gleichzeitig rechts wieder angehängt worden.

Die Rotate-Befehle verwendet man allgemein sehr selten. Dies kommt nicht zuletzt daher, daß die Rotation von Bits vom mathematischen oder logischen Standpunkt her nicht besonders sinnvoll ist. Deshalb gibt es sie in dieser Form auch kaum auf anderen Prozessoren. Eine denkbare Anwendung wäre folgende: Man möchte auf das obere Byte eines Langwortes im Byte-Modus zugreifen. Man könnte natürlich schreiben

```
MOVE   #24,D1
LSR.L  D1,D0
MOVE.B D0,Ziel
```

wobei der uns interessierende Wert in D0 steht (Beachten Sie, daß die Anzahl der Verschiebungen in einem Datenregister stehen muß). Eleganter, kürzer und schneller auszuführen wäre aber

```
ROL.L  #8,D0
MOVE.B D0,Ziel
```

Logische Operationen

Neben den bekannten arithmetischen Operationen gibt es auch noch andere Arten, wie man Zahlen miteinander verknüpfen kann: mit den logischen Operationen AND, OR und EOR. Bei Computern dienen diese Operationen nur selten dazu, Wahrheitswerte zu verknüpfen, sondern vielmehr zum Verändern bestimmter Bits. Gemeinsam ist diesen Befehlen, daß sie alle bitweise wirken:

Es wird immer ein Bit aus der Quelle mit dem Bit der entsprechenden Stelle des Zieloperanden verknüpft und das daraus resultierende Bit wird wieder im Ziel abgelegt.

Der AND-Befehl

Die AND-Operation entspricht in etwa dem umgangssprachlichen "und". Zwei Zahlen werden bitweise miteinander verknüpft: Es werden jeweils gleichwertige Bits aus der Quelle und dem Ziel untersucht. Im Ergebnis wird das entsprechende Bit nur dann gesetzt, wenn beide Bits Eins sind, in allen anderen Fällen steht dort eine Null. Um es in einer Wahrheitstafel wiederzugeben:

Quelle	Ziel	Ergebnis
0	0	0
0	1	0
1	0	0
1	1	1

Der AND-Befehl kann in praktisch allen Adressierungsarten und Byte-, Wort- und Langwortbreite benutzt werden. Z- und N-Bit werden entsprechend dem Gesamtergebnis gesetzt. Um ein Beispiel zu geben:

```
MOVE.B #%10101010,D0
AND.B  #%00001111,D0
```

Um die bitweise Wirkung des AND-Befehls deutlicher zu machen, werden die Operanden hier in binärer Schreibweise angegeben. Schreiben wir die beiden Operanden untereinander:

```
     10101010
AND  00001111
     00001010
```

Im Register D0 steht also nach der Operation der Wert 1010, dezimal 10. Wie unser Beispiel zeigt, kann man den AND-Befehl dazu verwenden, bestimmte Bits bedingungslos auf 0 zu setzen, während andere Bits nicht beeinflußt werden. Dies ist beispielsweise bei bestimmten Hardwareregistern sinnvoll, in de-

nen jedem Bit eine Bedeutung zugeschrieben ist. Bei allen Bits, die in dem Register gelöscht werden sollen, wird im Quelloperanden eine Null stehen, bei den zu erhaltenden Bits eine Eins. Bei einer Null kann ja im Ergebnis auf keinen Fall mehr eine Eins erscheinen, während bei einer Eins in einem Operanden direkt der Wert aus dem anderen Operanden übernommen wird.

Der OR-Befehl

Sinnvollerweise zeigt der OR-Befehl Parallelen zum umgangssprachlichen "oder": Der OR-Befehl wirkt ähnlich wie AND, nur wird hier ein Bit genau dann gesetzt, wenn das Bit in der Quelle oder im Ziel gesetzt war. Die Wahrheitsafel der OR-Verknüpfung sieht so aus:

Quelle	Ziel	Ergebnis
0	0	0
0	1	1
1	0	1
1	1	1

Hier steht im Ergebnis also nur dann eine Null, wenn beide Bits zuvor Null waren. Als Gegenstück zu AND wird OR dazu benutzt, bestimmte Bits bedingungslos zu setzen. Überall dort, wo in einem Operanden ein Bit gesetzt ist, wird im Ergebnis auch ein Bit gesetzt sein. Ist das Bit hingegen im ersten Operanden gelöscht, so ergibt sich das Resultat direkt aus dem Bit des zweiten Operanden. Betrachten wir hierzu wieder ein Programmfragment:

```
MOVE.B #%10101010,D0
OR.B   #%00001111,D0
```

Hier bewirkt der OR-Befehl, daß die unteren vier Bits in D0 auf jeden Fall gesetzt werden, während die oberen vier Bits erhalten bleiben. Somit entsteht folgendes Ergebnis:

```
    10101010
OR  00001111
    --------
    10101111
```

Der EOR-Befehl

EOR, in anderen Zusammenhängen auch oft mit XOR bezeichnet, steht für "eXclusive OR", also "ausschließendes oder", was soviel bedeutet wie "das eine oder das andere, aber nicht beides". So ist die Wirkung von EOR denn auch, daß ein Bit im Ergebnis genau dann gesetzt ist, wenn das Bit im ersten oder im zweiten Operanden gesetzt ist, aber nicht, wenn es in beiden gesetzt ist. Eine Wahrheitstafel macht das deutlicher:

Quelle	Ziel	Ergebnis
0	0	0
0	1	1
1	0	1
1	1	0

In der Praxis dient EOR dazu, bestimmte Bits zu invertieren. Wenn in einem Operanden eine Eins steht, so wird das Ergebnis genau der Gegenwert des Bits aus dem zweiten Operanden sein: Eine Null wird zu einer Eins und umgekehrt. Ist jedoch das Bit eines Operanden gelöscht, so wird der Wert des zweiten Operanden unverändert übernommen. Betrachten wir die Wirkung der folgenden Befehle:

```
MOVE.B #%10101010,D0
EOR.B  #%00001111,D0
```

Nach der Ausführung steht in D0

```
    10101010
EOR 00001111
    10100101
```

Die invertierende Eigenschaft des EOR kann beispielsweise benutzt werden, um ein Unterprogramm jedes zweite Mal etwas Bestimmtes ausführen zu lassen:

```
        .
        .
CLR.B       wechsel     Unser Wert muß initialisiert
        .               werden
        .
```

```
unter  EOR.B       #$FF,wechsel
       BNE         markel
         .
       [was hier steht, wird nur]
       [jedes 2. Mal ausgeführt]
         .
markel   .
         .
```

BNE steht für "Branch if Not Equal". Dieser Befehl verzweigt genau dann zum Speicherplatz "markel", wenn das Ergebnis der EOR-Operation Null ist. Genauer wird dieser Befehl in diesem Kapitel unter dem Abschnitt "Bedingte Verzweigungen" erklärt.

Hier wird die Variable "wechsel" also bei jedem Aufruf invertiert, und ein Programmteil wird nur ausgeführt, wenn sie auf 0 steht. Schließlich entspricht der hexadezimale Wert $FF einem Byte von 8 Einsen. Dabei wäre es in diesem Fall völlig egal, welchen Wert man statt $FF nimmt, solange er nur verschieden von 0 ist, denn die einzelnen Bits sind ja völlig unabhängig voneinander. Daraus kann man erkennen, daß es eine der Eigenschaften von EOR ist, daß eine Zahl mit sich selbst "EXKLUSIV-ODER" immer 0 ergibt.

Der NOT-Befehl

Da es oft vorkommt, daß eine Zahl vollständig invertiert werden soll, stellt der Prozessor 68000 dafür noch extra einen Befehl namens NOT zur Verfügung. Seine Wirkung ist genau die gleiche wie die eines EOR, bei dem der erste Operand aus lauter Einsen besteht. Statt

```
EOR.B #$FF,D0
```

kann man also auch schreiben

```
NOT.B D0
```

Manchmal will man auch das Vorzeichen einer Zahl in Zweierkomplementdarstellung ändern. Eine Vorzeichenänderung nimmt der Prozessor vor, indem er alle Bits der Zahl invertiert und 1 hinzuzählt (siehe Anhang A). Um das Vorzeichen eines Wortes in D0 zu ändern, müßte man schreiben:

```
EOR #$FFFF,D0
ADD #1,D0
```

oder, wie wir eben gesehen haben:

```
NOT D0
ADD #1,D0
```

Da die Änderung des Vorzeichens einer Zahl in Zweierkomplementdarstellung ziemlich oft vorkommt, haben die Entwickler des 68000 auch dafür einen Befehl zur Verfügung gestellt: Er nennt sich NEG (von engl. negate, negiere). Der Befehl

```
NEG D0
```

hat genau die gleiche Wirkung wie obige Befehlsfolge, auch das Carry-Bit wird genauso gesetzt.

Bedingte Verzweigungen

Es kommt recht oft vor, daß man einen Speicherbereich mit einem bestimmten Wert füllen will – etwa, um den Bildschirm auf eine bestimmte Farbe zu setzen oder um Variablenfelder zu initialisieren. Entwickeln wir also eine Routine, die einen Speicherbereich byteweise mit einem bestimmten Wert füllt.

```
fuell1
     MOVE.B wert,D0    * Wert ins Register laden
     MOVE.L anfang,A0  * Anfang in A0
loop
     MOVE.B D0,(A0)    * Byte schreiben
     ADDQ.L #1,A0      * Adresse um 1 erhöhen
     CMP.L ende,A0     * Ende erreicht?
     BNE loop          * noch nicht, also weiter
```

An diesem Beispiel lernen Sie sowohl eine neue Adressierungsart als auch den BNE-Befehl kennen. Doch gehen wir systematisch vor: Zuerst wird der Wert, mit dem der Speicherbereich gefüllt werden soll, ins Register D0 geladen, und die Anfangsadresse wird in Register A0 geschrieben. Dann kommt das Programm in eine Schleife (engl. loop), in der immer ein Byte an die Adresse, die in A0 steht, geschrieben wird und dann die Adresse in A0 erhöht wird. Beachten Sie, daß hier die Quick-Variante des ADD-Befehls Verwendung fand. Daraufhin wird untersucht, ob die Endadresse schon erreicht ist. Solange dies nicht der Fall ist, werden die Schritte der Schleife wiederholt. Übrigens füllt diese Schleife den Bereich einschließlich Anfangsadresse, jedoch ausschließlich Endadresse.

Hier haben wir die Methode verwendet, eine berechnete Adresse zu verwenden, die sogenannte indirekte Adressierung.

Mit

```
MOVE.B D0,(A0)
```

ist gemeint, daß der Inhalt von A0 die Adresse bezeichnet, an die der Wert geschrieben wird. Beachten Sie den Unterschied zu

```
MOVE.B D0,A0
```

was einfach den Wert aus D0 nach A0 überträgt. Die Klammern bedeuten somit eine "Indirektion" mehr, das heißt, der Prozessor wird angewiesen, nicht gleich die angegebene Stelle zum Schreiben oder Lesen zu verwenden, sondern nachzuschauen, was für ein Wert dort steht, und den Wert als Adresse zu verwenden. Natürlich kann diese Adressierungsart auch beim Quelloperanden verwendet werden:

```
MOVE.B (A0),D0
```

überträgt die Daten in der umgekehrten Richtung.

Nach dem Erhöhen der Adresse um Eins – Eins deshalb, weil es sich ja hier um Byte-Adressierung handelt – taucht schon wieder ein neuer Befehl auf: der CMP-Befehl. Er steht für "CoMPare" (vergleichen) und vergleicht Quell- und Zieloperand miteinander, woraufhin die System-Flags entsprechend gesetzt werden. Die Operanden werden jedoch nicht verändert. Intern geschieht dies, indem der Zieloperand vom Quelloperand abgezogen wird und die Flags entsprechend dem Ergebnis gesetzt werden. Somit kann man CMP mit dem SUB-Befehl vergleichen, allerdings sind die Funktionen von Quelle und Ziel vertauscht.

Natürlich existiert CMP auch in den Verarbeitungsbreiten Byte, Wort und Langwort. Da kein Ergebnis berechnet wird, ist der CMP-Befehl nur sinnvoll, wenn bald danach die System-Flags ausgewertet werden. Und damit sind wir bei der Gruppe der Branch-Befehle, den bedingten Verzweigungen.

BNE steht für "Branch if Not Equal", also "verzweige, wenn nicht gleich". Das bedeutet, der Programmfluß soll verzweigen, wenn das Zero-Flag nicht gesetzt ist. Allgemein prüfen die Branch-Befehle bestimmte System-Flags ab, ob sie gesetzt oder nicht gesetzt sind. Ist – je nach Befehl – die Bedingung nicht erfüllt, so geschieht nichts weiter, und das Programm wird mit dem nächsten Befehl fortgesetzt. Ist die Bedingung jedoch erfüllt, so wird zu dem angegebenen Label verzweigt. In unserem Beispiel funktioniert das so: Der CMP-Befehl vergleicht die aktuelle Adresse mit der Endadresse. Sind beide gleich, so wird das Z-Bit gesetzt, andernfalls gelöscht. Natürlich werden auch die ande-

ren Flags gesetzt, die uns allerdings hier nicht weiter interessieren. Als nächstes wird BNE ausgeführt, der nur dann wieder zum Label "loop" verzweigt, wenn das Zero-Flag nicht gesetzt ist, solange aktuelle Adresse und Endadresse ungleich sind.

Da unser BNE-Befehl nur einer aus einer ganzen Gruppe von Branch-Befehlen ist, betrachten wir nun die anderen Branch-Befehle. Jeder Branch-Befehl besteht aus zwei Teilen. Der erste – bezeichnet durch die zwei hinteren Buchstaben des Mnemoniks – gibt die Bedingung an, die geprüft werden soll, der zweite das Label, zu dem verzweigt wird, wenn die Bedingung wahr ist.

Flags als Verzweigungsbedingung

Die einfachste Bedingung, die wir überprüfen können, ist der Zustand der vier Flags Carry, Overflow, Zero und Negative. Da wir bei jedem den Zustand "gesetzt" oder "gelöscht" prüfen können, ergeben sich acht Bedingungen.

Carry		
BCS	Branch if Carry Set	Springe, wenn C=1
BCC	Branch if Carry Clear	Springe, wenn C=0
Overflow		
BVS	Branch if oVerflow Set	Springe, wenn V=1
BVC	Branch if oVerflow Clear	Springe, wenn V=0
Zero		
BEQ	Branch if EQual	Springe, wenn Z=1
BNE	Branch if Not Equal	Springe, wenn Z=0
Negative		
BMI	Branch if MInus	Springe, wenn N=1
BPL	Branch if PLus	Springe, wenn N=0

Verzweigungen nach CMP

Oft werden die Branch-Befehle dazu verwendet, je nach dem Ergebnis des CMP-Befehls bestimmte Aktionen zu veranlassen. Mit CMP sollen zwei Zahlen miteinander verglichen werden in der Form

```
CMP B,A
```

wobei A und B für beliebige Register oder Speicherzellen stehen. Beachten Sie, daß hier die Schreibweise genau umgekehrt zur algebraischen Notation ist, wo man etwa schreiben würde "A < B". Der zweite Operand muß also hier vorangestellt werden.

Es sind sechs Bedingungen vorgesehen, nach denen verzweigt werden kann:

1. A = B
2. A <> B
3. A > B
4. A < B
5. A >= B
6. A <= B

Die Fälle 1 und 2 sind mit den erwähnten Bedingungen schon abgedeckt, denn A = B und A <> B werden mit BEQ und BNE behandelt. Bedenken Sie, daß A = B gleichbedeutend ist mit B – A = 0, und daß der CMP-Befehl genau den Ausdruck B – A intern berechnet.

Für die restlichen Fälle müssen wir unterscheiden, ob es sich um vorzeichenlose Zahlen oder Zahlen in der Zweierkomplementdarstellung handelt. Ein Beispiel zeigt, daß beim Vergleichen von Zahlen darauf geachtet werden muß, ob sie vorzeichenbehaftet sind oder nicht:

Bei vorzeichenlosen Bytes ist (binär)

```
11100011 > 00000011
```

während bei vorzeichenbehafteten Zahlen die erste der beiden negativ wäre und somit

```
11100011 < 00000011
```

Betrachten wir zunächst die vorzeichenlosen Zahlen.

Vergleich vorzeichenloser Zahlen

Es gibt vier Möglichkeiten zu unterscheiden:

BHI Branch if HIgher
Verzweige, wenn A > B
erfüllt, wenn (C OR Z) = 0

BCS Branch if Carry Set
Verzweige, wenn A < B
Diese Bedingung ist erfüllt, wenn C=1.
BCS hat somit zwei Bedeutungen.

BCC Branch if Carry Clear
Verzweige, wenn A >= B.
Diese Bedingung ist erfüllt, wenn C=1.
Auch BCC hat zwei Bedeutungen.

BLS Branch if Lower or Same
Verzweige, wenn A <= B erfüllt,
wenn C=1 OR Z=1

Zu jeder der Bedingungen ist auch angegeben, wie sie sich aus den Flags ergeben. In den meisten Fällen kann es dem Assemblerprogrammierer jedoch egal sein, wie die Relation der beiden Zahlen intern festgestellt wird.

Will man mit vorzeichenbehafteten Zahlen arbeiten, so kann die Verwendung der eben besprochenen Bedingungen – wie wir oben gesehen haben – zu einem falschen Ergebnis führen. Daher gibt es von diesen vier Relationen auch Ausgaben für die Zweierkomplementzahlen.

Vergleich von vorzeichenbehafteten Zahlen

Auch hier gibt es wieder vier Bedingungen:

BLT Branch if Less Than
Verzweige, wenn A < B
erfüllt, wenn (N EOR V) = 1

BGT Branch if Greater Than
Verzweige, wenn A > B erfüllt,
wenn (Z AND (N EOR V)) = 1

BLE Branch if Less or Equal
Verzweige, wenn A <= B erfüllt,
wenn (Z OR (N EOR V)) = 1

BGE Branch if Greater or Equal
Verzweige, wenn A >= B erfüllt,
wenn (N EOR V) = 0

Auch hier gilt, daß die Art, in der die Bedingungen festgestellt werden, im allgemeinen nicht weiter interessiert.

Sonstige Verzweigungen

Es gibt noch andere Verzweigungen außer den bisher beschriebenen. Es sind:

BT	Branch if True	verzweige in jedem Fall
BF	Branch if False	verzweige niemals

Statt BT kann auch BRA für "branch" geschrieben werden, da dies eigentlich keine bedingte Verzweigung mehr ist, sondern ein unbedingter Sprung.

BF erscheint gänzlich sinnlos, da dieser Befehl niemals verzweigt und somit überhaupt nichts tut. Er wurde nur der Vollständigkeit halber in den Befehlssatz aufgenommen. Wie wir aber später noch sehen werden, kann die Bedingung F für "False" in anderen Befehlen durchaus sinnvoll verwendet werden.

Die DBcc-Befehle

Betrachten wir noch einmal unsere Füll-Schleife. Eine andere Möglichkeit wäre, eine zusätzliche Variable zu verwenden, die die Differenz zwischen End- und Anfangsadresse enthält, und diese herunterzuzählen. Legen wir die Differenz ins Register D1:

```
Fuell2:
     MOVE.B  Wert,D0    * Wert ins Register laden
     MOVE.L  Anfang,A0  * Anfang in A0
     MOVE.L  Ende,D1    * Differenz=Ende-Anfang
     SUB.L   Anfang,D1  * in D1 berechnen
loop MOVE.B  D0,(A0)    * Byte schreiben
     ADDQ.L  #1,A0      * Adresse um 1 erhöhen
     SUBQ.L  #1,D1      * Differenz herunterzählen
     BNE     loop       * noch nicht Null, weiter
```

Diese Version der Füll-Schleife dürfte sogar etwas schneller sein als die erste.

Es kommt oft vor, daß ein Wert am Ende einer Schleife dekrementiert (um eins verringert) wird und je nach dem Ergebnis – Null oder nicht Null – ver-

zweigt wird. Deshalb haben die Konstrukteure des MC68000 auch dafür einen Befehl – oder vielmehr eine ganze Gruppe von Befehlen – vorgesehen, die diese beiden Aufgaben auf einmal bewältigen. Es handelt sich dabei um die DBcc-Befehle. "DB" steht für "Decrement and Branch", also dekrementiere (verringere um eins) und verzweige. "cc" bedeutet "condition code", also Bedingungs-Abkürzungen, und steht stellvertretend für alle Buchstabenkombinationen, die man erhält, wenn man von den besprochenen Branch-Befehlen das B wegläßt (nicht zu verwechseln mit CC, dem Code für "Carry Clear").

Diese Befehlsgruppe hat zwei Operanden: Der erste ist ein Datenregister (D0 – D7), der zweite ein Label. So haben die DBcc-Befehle die Form

```
DBcc Dn,Label
```

für n = 0, 1 ,2, ..., 7

Die Ausführungsweise ist folgende: Zunächst wird die angegebene Bedingung geprüft. Trifft sie zu, so wird nichts weiter getan und mit dem nächsten Befehl fortgefahren. Wenn sie nicht zutrifft, dann wird zunächst das angegebene Datenregister in Wortbreite dekrementiert. Ist der Inhalt danach verschieden von –1, so wird zum Label verzweigt und somit eine Schleife ein weiteres Mal wiederholt. Ist jedoch –1 erreicht, so wird mit dem nächsten Befehl fortgefahren. Die Flags werden in keinem Fall beeinflußt. Betrachten Sie dazu Abb. 2.9.

Man kann sich das Verhalten von DBcc veranschaulichen, indem man eine Befehlsfolge aus schon bekannten Befehlen aufschreibt:

```
          DBcc     Dn,Schleife
Weiter:    [nächster befehl]
           .
           .
           .
```

ist äquivalent zu

```
          Bcc      Weiter
          SUBQ.W   #1,Dn
          CMP.W    #-1,Dn
          BNE      Schleife
Weiter:    [nächster befehl]
           .
           .
           .
           .
```

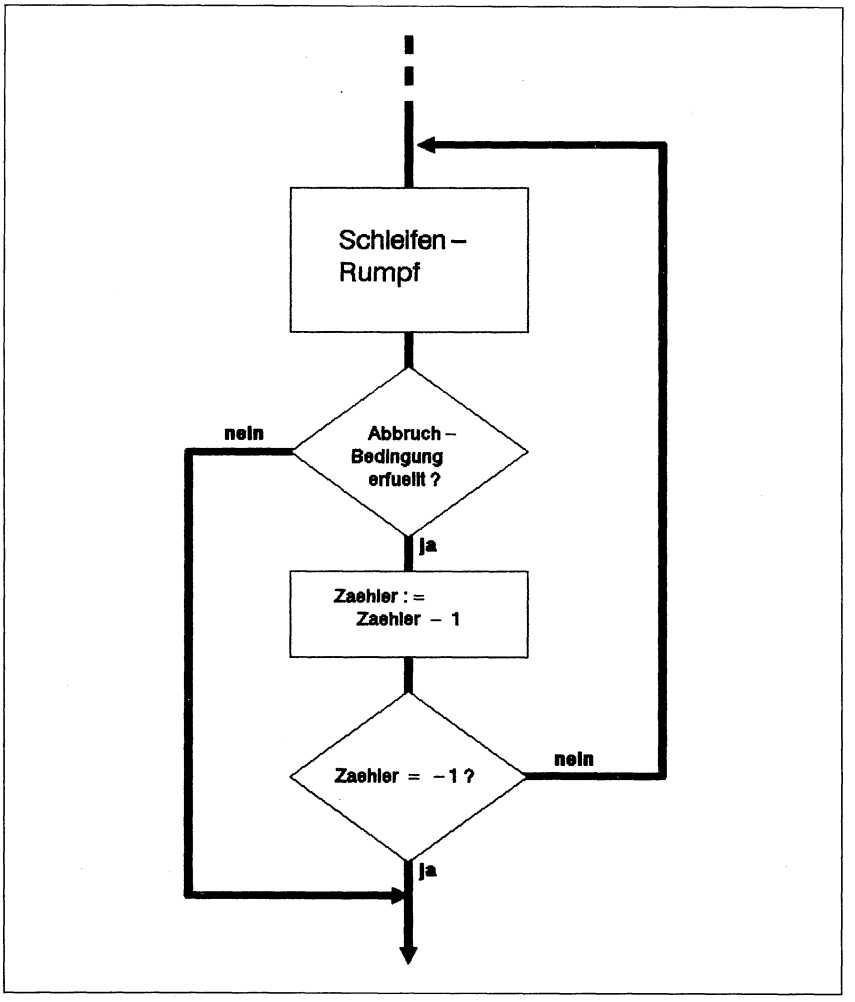


Abb. 2.9: Logischer Aufbau einer DBcc-Schleife

Tatsächlich werden so zwei Bedingungen auf einmal geprüft, die zum Verlassen der Schleife führen. So sind die meisten DBcc-Befehle nur sinnvoll, wenn direkt davor ein CMP-Befehl oder eine andere Operation steht, die die Flags beeinflußt und eine extra-Bedingung zum Verlassen der Schleife liefert. Da dies nicht oft der Fall ist, ist DBF bei weitem der am häufigsten verwendete Befehl dieser Gruppe. Wenn Sie "F" als "condition code" oben einsetzen, wer-

den Sie merken, daß damit die Schleife nur noch verlassen werden kann, wenn der Zähler –1 ist. Beachten Sie, daß in dieser Beziehung die Wirkung des DBcc-Befehls genau entgegengesetzt zu der von Bcc ist: Bei DBcc wird dann verzweigt (solange der Zähler noch nicht –1 ist), wenn die angegebene Bedingung nicht erfüllt ist. Daher hier auch die Verwendung von DBF.

Übrigens – in einem Punkt stimmt die Wirkung der Ersatz-Befehlsfolge nicht mit der von DBcc überein, nämlich darin, daß bei dieser Version die Flags verändert werden, was ja bei DBcc nicht der Fall ist.

Nun wieder zurück zu unserer Füllroutine. Hier können wir DBF gut verwenden:

```
Fuell3:
     MOVE.B Wert,D0    * Wert ins Register laden
     MOVE.L Anfang,A0  * Anfang in A0
     MOVE.L Ende,D1    * Differenz=Ende-Anfang
     SUB.L  anfang,D1  * in D1 berechnen
     SUBQ.W #1,D1      * eins abziehen
Loop MOVE.B D0,(A0)    * Byte schreiben
     ADDQ.L #1,A0      * Adresse um 1 erhöhen
     DBF D1,Loop       * dekrementieren und
                       * verzweigen
```

Diese Version ist noch um einiges schneller. Die Verwendung von DBF bringt allerdings auch Nachteile mit sich: Wir können mit dieser Routine keine Speicherbereiche mehr füllen, die größer als 64K sind, denn DBF behandelt ja nur 16 Bit der Differenz. Außerdem muß man darauf achten, daß DBF die Schleife bei –1 abbricht und nicht wie die bisherigen Versionen der Füllroutine bei 0. Dies ist eigentlich dafür gedacht, daß eine Schleife mit allen Werten vom Anfangswert bis hinunter zu 0 einschließlich durchlaufen wird. In vielen Fällen mag dies ja sinnvoll sein, aber wir können es hier nun gerade nicht gebrauchen. Deshalb wird der Differenzwert vor dem Eintritt in die Schleife um 1 vermindert.

Adreßberechnung bei Verzweigungsbefehlen

Noch ein Wort zur Darstellung der Adressen bei den Branch-Befehlen: Bei den Bcc- und DBcc-Befehlen wird die Adresse normalerweise als 16-Bit-Adreßdistanz angegeben. Das heißt, daß nicht die absolute Adresse abgespeichert wird, sondern die Differenz zwischen der Adresse, zu der gesprungen werden soll, und der Adresse, an der der Branch-Befehl steht. Vorteil dieser Methode ist es, daß Speicher gespart wird, da ja für die Adresse nur ein Wort statt zweien nötig ist. Außerdem kann dieser Befehl auch in relozierbaren Pro-

grammen verwendet werden, da ja die Adreßdifferenz zwischen zwei Befehlen immer gleich ist, egal, wo das Programm nun gerade steht. Bei der Ausführung wird die absolute Adresse errechnet, indem die Adreßdistanz zum aktuellen Stand des PC addiert wird (genaugenommen ist das die Adresse des Bcc plus zwei, da der PC bei der Analyse des Befehlswortes schon ein Wort weitergeschoben worden ist).

Wenn es sich um kurze Sprünge im Bereich –126 bis +129 handelt, kann für die Adreßdifferenz auch nur ein Byte angegeben werden, was wiederum ein Wort und etwas Rechenzeit spart. Kenntlich gemacht wird dies durch den Extender ".S" nach dem Befehl:

```
ewig    BRA.S    ewig
```

Manche Assembler nehmen – zumindest optional – diese Optimierung allerdings selbständig vor, sofern die Adreßdifferenz entsprechend gering ist.

Nachteil dieser Methode der Adreßberechnung ist aber, daß nur Adressen in der Umgebung des Branch-Befehls erreicht werden können, bei 16 Bit von –32766 bis +32769. Will man über diesen Bereich hinaus, so muß der JMP-Befehl verwendet werden, der bedingungslos an eine Adresse verzweigt, die mit einem Langwort angegeben wird:

```
JMP irgendwohin
```

Mit diesem Befehl kann jeder Punkt des Speichers erreicht werden.

Die Adressierungsarten des MC68000

Bisher haben wir immer von Operanden gesprochen, ohne genauer darauf einzugehen, was ein Operand eigentlich alles sein kann. Da der MC68000 hier eine Vielzahl von Varianten bietet, sind diese es wert, daß man ihnen einen eigenen Abschnitt widmet.

Die vielfältigen Adressierungsarten kann man sechs Hauptgruppen zuordnen:

1. Register direkt
2. Konstanten-Adressierung
3. Absolute Adressierung des Speichers
4. Indirekte Adressierung des Speichers
5. Implizite Adressierung eines Registers
6. Programmzähler-relative Adressierung

In praktisch allen Fällen sind sämtliche Adressierungsarten auf die Verarbeitungsbreiten Byte, Wort und Langwort anwendbar. Befassen wir uns zunächst mit der bisher schon oft verwendeten Adressierungsart, die Register direkt anspricht.

Register-direkte Adressierung

Dies heißt nichts weiter, als daß ein Operand sich in einem Register befindet.

Dazu ein Beispiel:

```
CLR.W D0
```

Mit Registern sind Datenregister und Adreßregister gemeint. Es ist nur zu beachten, daß bei Adreßregistern keine Byte-Befehle zulässig sind. Auch Wort-Befehle verhalten sich auf Adreßregistern etwas anders. Während bei einem Wortzugriff auf ein Datenregister die oberen 16 Bits unverändert bleiben, wird ein Wort, das in ein Adreßregister geschrieben wird, automatisch vorzeichenrichtig erweitert. Dies hat den Sinn, daß zur Adressierung in den unteren 32K, in denen sich die Systemvariablen befinden, oder in den oberen 32K, wo die Hardware-Register liegen, nur ein Wort reserviert werden muß.

Konstanten-Adressierung

Konstanten-Adressierung ist das, was wir immer mit einem Doppelkreuz (#) kenntlich gemacht haben: Der Operand folgt direkt dem Befehl. Dazu ein Beispiel:

```
MOVE.L #$314159,D0
```

Dazu könnte man einwenden, daß es sich in Wirklichkeit um Registeradressierung handelt, denn schließlich ist ja D0 betroffen. Ganz unberechtigt ist dieser Einwand nicht: Beim Prozessor 68000 muß man immer zwischen Adressierung von Quelle und Ziel unterscheiden. Um es also ganz exakt auszudrücken: Die Adressierungsart der Quelle ist "unmittelbar", die des Ziels ist "Register-direkt". Diese unmittelbare Adressierungsart ist natürlich nur für die Quelle sinnvoll und erlaubt.

Je nach der Verarbeitungsbreite des Befehls ist der direkt hinter dem Befehlswort abgespeicherte Operand 8, 16 oder 32 Bit lang. Ausnahme: Bei ADDQ und SUBQ ist der konstante Operand in jeder Verarbeitungsbreite nur 3 Bit breit. Bei MOVEQ sind es 8 Bit.

Absolute Adressierung

Absolute Adressierung bedeutet nichts anderes, als daß die Adresse einer Speicherzelle direkt angegeben wird. Es gibt allerdings zwei Arten, die Adresse anzugeben:

Absolut lang

Hier wird die Adresse in einem Langwort (32 Bit) angegeben. Wenn dieses Langwort im Speicher steht, gibt das erste Wort (das mit der niedrigeren Adresse) den höherwertigen Teil, das zweite den niederwertigen Teil der Adresse an. Beim 68000 werden allerdings nur die unteren 24 Bit der Adresse genutzt. Die restlichen 8 Bit werden ignoriert.

Beispiel:

```
CLR.W $20000
```

Absolut kurz

Hier gibt nur ein Wort die Adresse an. Um die vollständige Adresse zu erhalten, wird das Wort intern auf 32 Bit vorzeichenrichtig erweitert. Die Verwendung von kurzen Adressen spart Speicherplatz und Rechenzeit, ist aber nur bei Zugriffen auf die untersten oder obersten 32K des Speicherbereiches anwendbar.

Beispiel:

```
CLR.L $200
```

Indirekte Adressierung des Speichers

Bei allen hier aufgeführten Adressierungsarten ist die Adresse des Speicherplatzes, auf den zugegriffen werden soll, in einem oder in mehreren Registern enthalten. Indirekte Adressierung wird durch Klammern um das oder die Register kenntlich gemacht, worin die Adresse enthalten ist.

Register-Indirekte Adressierung

Hier steht die absolute Adresse eines Operanden in den vollen 32 Bit eines Adreßregisters. Ausgedrückt wird dies dadurch, daß man um die Bezeichnung

des Adreßregisters Klammern setzt. Die folgende Befehlsfolge dient dazu, Speicherstelle 1000 zu löschen:

```
MOVE.L #1000,A0
CLR.L  (A0)
```

Dies läuft so ab, daß der Prozessor, nachdem er einen solchen Befehl erkannt hat, den Wert aus dem angegebenen Adreßregister ausliest und diesen wiederum als Operandenadresse verwertet.

Adreßregister indirekt mit Postinkrement

"Postinkrement" bedeutet soviel wie "Erhöhen nach der Operation". Die Wirkungsweise ähnelt jener der Register-indirekten-Adressierung. Auch hier enthält ein Adreßregister die Adresse des Operanden im Speicher. Der Unterschied ist jedoch, daß nach der Ausführung des Befehls die Adresse im Register erhöht wird, und zwar um die Anzahl der bearbeiteten Bytes. Das heißt: Bei Bytes um 1, bei Wortbreite um 2 und bei Langworten um 4. Diese Adressierungsart ist dazu gedacht, Felder schnell und ohne viel Mitzählen vom ersten bis zum letzten Element durchgehen zu können. Notiert wird das Ganze durch die üblichen Klammern um das Adreßregister und ein nachgestelltes "+". Ein besonders gutes Beispiel dafür liefert die Füllroutine aus dem vorhergehenden Kapitel:

```
Fuell4: MOVE.B  Wert,D0     * Wert in D0 laden
        MOVE.L  Anfang,A0   * Anfang in A0
        MOVE.L  Ende,D1     * Differenz=Ende-Anfang
        SUB.L   Anfang,D1   * in D1 berechnen
        SUBQ.W  #1,D1       * eins abziehen
Loop:   MOVE.B  D0,(A0)+    * Byte schreiben und Adresse erhöhen
        DBF     D1,Loop     * dekrementieren und verzweigen
```

Das ist mittlerweise schon die vierte Version, die wieder etwas schneller und eleganter als ihr Vorgänger ist. Hätten Sie gedacht, daß man eine so simple Routine auf so viele (sinnvolle) Arten formulieren kann? Dabei wird im siebten Kapitel demonstriert, wie man eine solche Schleife noch schneller machen kann.

Der Befehl MOVE.B D0,(A0)+ ersetzt hier die MOVE- und die ADDQ-Anweisung aus der dritten Version. Somit besteht die eigentliche Schleife jetzt nur noch aus zwei Befehlen!

Es ist natürlich auch erlaubt und durchaus sinnvoll, für Quell- und Zieloperand gleichzeitig diese Adressierungsart zu verwenden, etwa in

```
MOVE.L (A0)+,(A1)+
```

eine Anweisung, die, sofern sie in einer Schleife steht, bequem Speicherbereiche kopieren kann.

Adreßregister indirekt mit Predekrement

"Predekrement", was für "verringern vor der Operation" steht, kann man als Gegenstück zum Postinkrement betrachten. Hier wird zuerst die Adresse im Register je nach Verarbeitungsbreite um 1, 2 oder 4 verringert und dann auf diese Adresse zugegriffen. Dargestellt wird das, indem man ein Minus vor die Klammer um das Adreßregister setzt. Die Adressierungsarten mit Postinkrement und Predekrement bieten sich dazu an, einen Stack (Stapel) zu realisieren (die Verwendungs von Stacks wird im nächsten Kapitel noch ausführlich behandelt).

Auf dem ATARI ST wachsen die Stacks nach unten, also von den höheren Adressen zu den niedrigeren. Erinnern Sie sich noch, daß man den System-Stackpointer mit SP bezeichnet und es sich dabei in Wirklichkeit um Register A7 handelt? Eine Methode, den Inhalt von Register D0 nach D1 zu übertragen, könnte man so realisieren.

```
MOVE.L D0,-(SP)
MOVE.L (SP)+,D1
```

Nach Ausführung dieser Befehle enthält D1 den gleichen Wert wie D0. Deshalb muß das Dekrementieren vor und das Inkrementieren nach dem Befehl ausgeführt werden.

Adreßregister indirekt mit Adreßdistanz

Auch hier steht wieder eine Adresse in einem Adreßregister. Doch vor die Klammer um die Bezeichnung des Adreßregisters kann man noch eine 16-Bit-Konstante schreiben, die intern zum Inhalt des Registers addiert wird, um die Adresse des Operanden zu bestimmen: Es handelt sich dabei um die Adreßdistanz. Der Inhalt des Adreßregisters wird dabei nicht verändert.

Beispiel:

```
MOVE.L #1000,A0
CLR.B 2(A0)
```

Nach der Ausführung dieser Befehlsfolge ist Speicherplatz 1002 gelöscht, und in A0 steht noch immer die Adresse 1000.

Gehen wir den Vorgang einmal Schritt für Schritt durch: Nachdem der Befehl von der CPU erkannt worden ist, wird die Adresse aus dem angegebenen Register geholt und der 16-Bit-Index, der hinter dem Befehlswort abgespeichert ist, vorzeichenrichtig auf Langwortbreite erweitert. Danach werden beide Werte intern addiert, und das Ergebnis ist schließlich die Adresse des Operanden. Der Index kann von –32768 bis +32767 reichen.

Adreßregister indirekt mit Index und Adreßdistanz

Dies ist wohl die komplizierteste Adressierungsart. Jedoch im Grunde besteht sie aus nichts weiter als ein paar Additionen. Hier wird in den Klammern zunächst einmal wieder ein Adreßregister angegeben. Ebenfalls innerhalb der Klammern folgt, durch ein Komma getrennt, der sogenannte Index. Es handelt sich dabei um ein weiteres Register, das diesmal nicht nur ein Adreßregister, sondern auch ein Datenregister sein darf. Für den Index kann ein Extender ".W" oder ".L" angegeben werden, der die Verarbeitungsbreite dieses Registers angibt. Wird keiner angegeben, so nimmt der Assembler automatisch ".W" an. Nun kommt die dritte Komponente: Vor der Klammer steht noch eine vorzeichenbehaftete 8-Bit-Konstante, die Adreßdistanz. Die Adresse ergibt sich als Summe aller drei Komponenten, wobei alles, was kürzer als ein Langwort ist, zunächst vorzeichenrichtig auf Langwortbreite erweitert wird.

Beispiel:

```
CLR.L -8(A2,D0.W)
```

Nehmen wir an, daß in A2 die Adresse $1000 steht, während D0 $410 enthält. (Die oberen 16 Bit von D0 interessieren uns dabei nicht.) Die Adresse wird nun folgendermaßen errechnet (hexadezimal):

```
A2:         00001000
D0:         00000410
Konstante:  FFFFFFF8
           ----------
        (1)00001408
```

FFFFFFF8 ist die Zweierkomplementdarstellung von –8. Wie üblich werden bei der Addition von Zweierkomplementzahlen Überträge ignoriert. Der Inhalt von Speicherplatz $1408 wird nun von obigem Befehl gelöscht.

In der Praxis kommt es oft vor, daß man keine Adreßdistanz braucht, wohl aber den Index. In diesem Fall schreibt man für die Adreßdistanz einfach Null. Sie merken es schon – es gibt keine Adressierungsart nur mit Index, aber ohne Adreßdistanz.

Implizite Adressierung eines Registers

"Implizite Adressierung" heißt nichts anderes, als daß Register von einem Befehl beeinflußt werden, obwohl sie dort nicht ausdrücklich erwähnt werden. Betroffen sind davon nur der Programmzähler PC und der Stapelzeiger SP. So wird etwa der PC von jedem Befehl verändert, in dem er einfach weitergeschoben wird, während manche Befehle ihn auf kompliziertere Weise beeinflussen, wie etwa die Branch-Befehle. Es gibt noch eine ganze Reihe anderer Befehle, die sich ähnlich auf PC oder SP auswirken, aber bei ihnen geht schon aus der Funktion hervor, wie die Register beeinflußt werden.

Programmzähler-relative Adressierung

Die meisten Programme sind grundsätzlich auf eine bestimmte Position im Speicher angewiesen, um laufen zu können. Doch manchmal ist es wünschenswert, daß Programme an jedem Platz im Speicher lauffähig sind, etwa wenn man bestimmte Routinen zu jedem Zeitpunkt zur Verfügung haben will, die aber andere Programme nicht an der Ausführung hindern sollen (beispielsweise einen speziell angepaßten Druckertreiber). Mit der Programmzähler-relativen Adressierungsart wurde die Möglichkeit geschaffen, solche Programme zu schreiben. Die Grundidee dabei ist, daß beim Zugriff auf Programm-eigene Daten nicht die absolute Adresse der Daten angegeben wird, sondern die Differenz zwischen der Adresse des Operanden und der Adresse des darauf zugreifenden Befehls. Diese Differenz kann der Assembler während des Assemblierens ausrechnen. Bei der Ausführung des Befehls wird dann die Adresse des Befehls – also der aktuelle Inhalt der Programmzählers – wieder addiert, um die eigentliche Adresse zu erzeugen. Auf diese Art ist das Programm nicht auf eine bestimmte Position der Daten angewiesen: Die Daten wandern mit, wenn der Programmcode im Speicher verschoben wird.

Der für die Adreßberechnung benutzte Programmzählerstand ist die Adresse des ersten Wortes nach dem Befehlscode, da der Programmzähler bei der Adreßberechnung eben auf diesen Wert zeigt.

Leider reicht die PC-relative Adressierungsart allein nicht aus, um Programme gleichzeitig relozierbar und effizient zu machen, da sie einerseits nicht bei allen Befehlen implementiert ist, andererseits praktisch nur beim Quelloperanden verwendet werden darf. Wir werden aber noch sehen, daß das Betriebssystem des ATARI ST trotzdem dafür sorgt, daß Programme an jeder beliebigen Position im Speicher laufen können. Bei der PC-relativen Adressierung gibt es zwei Varianten, die genauso aufgebaut sind wie "Adreßregister indirekt mit Adreßdistanz" und "Adreßregister indirekt mit Index und Adreßdistanz".

Programmzähler-relativ mit Adreßdistanz

Hier wird die Summe aus dem Inhalt des Programmzählers und der vorzeichenerweiterten Adreßdistanz gebildet. Die Distanz kann wieder von –32768 bis +32767 reichen. Beachten Sie, daß bei größeren Distanzen diese Adressierungsart nicht mehr verwendet werden kann.

Beispiel:

Befehl: `MOVE D0,24(PC)`

Dieser Befehl bewegt den Inhalt des Wortes aus der Speicherstelle (Befehl + 26) nach D0. Warum 26? Weil ja der PC im Moment der Adreßberechnung auf (Befehl + 2) zeigt.

Normalerweise brauchen Sie solche Adreßdistanzen nicht selbst zu berechnen, denn dazu ist ja der Assembler da. Bei vielen Assemblern können Sie statt einer Zahl auch einfach ein Label verwenden, der natürlich irgendwo definiert sein muß. Der Assembler übernimmt dann die Berechnung der Adreßdistanz:

```
          MOVE D0,Zähler(PC)
                    .
                    .
                    .
Zähler:   DS.W 1    .
                    .
```

Dieser Befehl wird auf die Speicherstelle "Zähler" zugreifen.

Manche Assembler verwenden sogar, wo immer es möglich ist, PC-relative Adressierung. Das heißt, diese Adressierungsart wird immer da verwendet, wo Label als Operanden benutzt werden. Da jedoch viele Befehle die Möglichkeit der PC-relativen Adressierung vermissen lassen und andere sie nur beim Zieloperanden erlauben, kann diese Methode zu großen Problemen führen. Wenn man wirklich vom Code her relozierbare Programme schreiben will, bleibt in vielen Fällen nur noch die Adreßberechnung mittels LEA.

Programmzähler-relativ mit Index und Adreßdistanz

Hier berechnet sich die Operandenadresse aus drei Komponenten:

1. dem aktuellen Stand des PC
2. dem Inhalt des Indexregisters
3. der 8-Bit-Adreßdistanz

Wie üblich werden alle Werte, die kürzer als ein Langwort sind, zuvor vorzeichenrichtig erweitert.

Beispiel:

```
CLR.L 24(PC,D0.W)
```

Auch hier kann anstelle der Zahl ein Symbol angegeben werden, wobei der Assembler dafür die Differenz zwischen der Befehlsadresse +2 und dem Symbol einsetzt:

```
CLR.L Tab(PC,D0.W)
```

Sinnvoll ist diese Adressierungsart dann, wenn das Indexregister den Index in einer Tabelle enthält und die Adreßdistanz die Entfernung zum Tabellenanfang zeigt. Problematisch ist dabei nur, daß das angegebene Symbol im Bereich –128 bis +127 Bytes vom Befehl liegen muß. Wegen dieser Einschränkung findet diese Adressierungsart recht selten Verwendung.

Stackorganisation und Programmsprünge

Bis jetzt haben wir uns immer nur mit kleinen Routinen beschäftigt. Programme bestehen jedoch im allgemeinen aus vielen Routinen, die sich gegenseitig – auch verschachtelt – aufrufen können. Wie sich später noch zeigen wird, benötigt man dafür einen sogenannten Stack (engl. für Stapel).

Der Stack

Der Stack ist ein reservierter Speicherbereich, in dem Daten abgelegt und später wiedergeholt werden können. Die Daten werden immer nur ans Ende geschrieben und von dort auch wieder gelesen. Daher brauchen wir einen Zeiger auf das Ende des Stacks. Dieser ist auf der Hardware des MC68000 in Form des Stackpointers SP realisiert, was im Grunde ein Deckname für das Adreßregister A7 ist.

Eigentlich gibt es zwei Stackpointer: einen für den Usermodus und einen Anderen für den Supervisormodus. Im jeweiligen Modus ist immer nur der dazugehörende Stackpointer erreichbar. Übrigens können aufgrund der Vielzahl der Adressierungsarten des 68000 alle Adreßregister vom Programmierer als Stackpointer benutzt werden. Nur wird eben A7 von allen Befehlen benutzt,

die implizit auf den Stack zugreifen. Auf dem ST wächst der Stack von oben nach unten. Die Adresse in A7 zeigt immer auf die unterste Adresse des letzten Wertes, der dort abgelegt wurde.

Würde man folgendermaßen einen Wert auf dem Stack ablegen:

```
MOVE.L Wert,-(SP)
```

so wird der Stackpointer vor dem Schreiben des Wertes um die Anzahl der zu schreibenden Bytes verringert. Nach dem Schreiben zeigt er also wieder auf dem Stack die untere Adresse des neuen Wertes. Übrigens dürfen auf dem Stack nur Worte und Langworte abgelegt werden, denn würde ein Befehl ausgeführt werden wie

```
MOVE.B Chaos,-(SP)
```

dann würde zwar bei der Ausführung dieses Befehls alles gut gehen. Wenn aber irgendwann später im Programm wieder ein Wort oder Langwort auf dem Stack abgelegt werden soll, dann zeigt der Stackpointer auf eine ungerade Adresse, und das Resultat wäre ein Absturz des ATARI ST.

Um einen Wert vom Stack wieder herunterzuholen, geht man so vor:

```
MOVE.L (SP)+,D0
```

Damit wird der Wert, auf den der Stackpointer gerade zeigt, heruntergeholt, und danach wird der Stackpointer um die Anzahl der gelesenen Bytes hochgezählt.

Da es recht oft vorkommt, daß mehrere Register auf den Stack gesichert werden müssen – etwa, wenn ein umfangreiches Unterprogramm aufgerufen wird, das wichtige Register verändert –, haben die Konstrukteure des MC68000 auch dafür etwas vorgesehen: den MOVEM-Befehl (MOVE Multiple – bewege mehrfach).

Ein Beispiel:

```
MOVEM D0-D4,-(SP)
```

bewegt die Inhalte der Register D0 bis D4 auf den Stack. Bei diesem Befehl, der in Wort- und Langwortbreite arbeitet, kann eine Liste von Registern angegeben werden, die in nach ihrer Nummer aufsteigender Reihenfolge auf dem

Stack abgelegt werden (Datenregister immer vor Adreßregistern). Auch so etwas ist möglich:

```
MOVEM.L A3-A5/D0-D2/D5/A0,-(SP)
```

Hier werden die Register D0 bis D2 ,D5, A0 und A3 bis A5 als Langworte abgelegt. Wie Sie sehen, können die Register beliebig ausgewählt werden. Intern funktioniert das so, daß dieser Befehl ein Wort mitbekommt, in dem jedes Bit für ein Register steht. Es legt fest, ob das entsprechende Register abgelegt werden soll oder nicht. Die Umsetzung der Registerliste in dieses Wort übernimmt der Assembler.

Um die so gesicherten Register wiederzuholen, schreibt man

```
MOVEM (SP)+,D0-D4
```

beziehungsweise

```
MOVEM.L (SP)+,A3-A5/D0-D2/D5/A0
```

Damit gelangen – sofern der Stackpointer nicht verändert worden ist – wieder die alten Inhalte in die Register.

Eines sollten Sie bei der intensiven Benutzung des Stacks berücksichtigen: Es wird bei Stack-Zugriffen nicht geprüft, ob etwa irgendwelche Grenzen überschritten werden. Normalerweise sollte das nicht zu Problemen führen, da der Stack gewöhnlich mindestens 4K Platz zum Wachsen hat. Kritisch wird es nur dann, wenn irgendwelche Routinen Reste auf dem Stack zurücklassen, die mit der Zeit immer mehr anwachsen. Das Resultat eines Stacküberlaufs ist selten ein Absturz, sondern eher ein Programm, das sich merkwürdig benimmt. Achten Sie daher immer darauf, daß nirgendwo etwas auf dem Stack zurückgelassen wird.

Unterprogramme

Die wichtigste Funktion des Stacks besteht darin, die Aufrufe von Unterprogrammen zu verwalten. Wie Sie es vielleicht von anderen Programmiersprachen kennen, ist es sehr komfortabel, wenn man Codesequenzen mit einem einzigen Befehl aufrufen kann – wie etwa mit dem GOSUB in BASIC. Der Ablauf ist dabei folgender: Aus dem Hauptprogramm findet ein Sprung zum Unterprogramm statt, das mit seinem Namen, seiner Adresse oder – wie in BASIC – mit einer Zeilennummer identifiziert wird. Dort werden nun die Befehle so lange abgearbeitet, bis der Befehl erkannt wird, der den Rücksprung ins auf-

muß irgendwo vermerkt werden, an welcher Stelle im aufrufenden Programm fortgefahren werden soll. Die möglicherweise naheliegendste Möglichkeit, die Adresse, von der das Unterprogramm aufgerufen wurde, in einem festen Speicherplatz abzulegen, können wir gleich wieder verwerfen, denn dies würde keine geschachtelten Unterprogramme erlauben: Beim zweiten geschachtelten Aufruf eines Unterprogramms wäre der erste Wert der Rücksprungadresse verschwunden. Deshalb wird hier eine Datenstruktur gebraucht, die Werte, die darauf abgelegt wurden, so lange behält, bis sie wieder gelesen werden. Und genau diese Voraussetzungen erfüllt der Stack.

Mit dem Stack funktioniert der Unterprogrammaufruf folgendermaßen:

Der Sprung zu einem Unterprogramm erfolgt mit den Befehlen JSR (Jump to SubRoutine) oder BSR (Branch to SubRoutine), deren Operand die Adresse des Unterprogramms ist. Daraufhin wird der aktuelle Wert des Befehlszählers PC auf den Stack gesichert und der PC danach mit der angegebenen Adresse geladen – was auf einen Sprung zum Unterprogramm hinausläuft. Das Unterprogramm wird nun so lange ausgeführt, bis der Prozessor auf die Anweisung RTS (ReTurn from Subroutine) trifft. Dieser Befehl holt den gesicherten Wert des PC wieder vom Stack und fährt mit dem Befehl fort, der dem JSR oder BSR folgt.

Beispiel:

```
linie_ziehen:
            .
            .
            .
            JSR       punkt
            MOVE      #1000,D0
            .
            .
            .
punkt:      .
            [Befehle des Unterprogramms]
            .
            RTS
```

Bei dieser angedeuteten Befehlsfolge wird, sobald der JSR-Befehl ausgeführt wird, zum Label "punkt" gesprungen, und die dortigen Befehle werden ausgeführt. Bei der Ausführung des RTS wird die Kontrolle wieder dem Hauptprogramm übergeben und der dem JSR folgende Befehl ausgeführt, also in diesem Fall MOVE #1000,D0.

Die Befehle JSR und BSR gleichen sich in ihrer Wirkung. Der Unterschied liegt nur in der Art, wie die Adresse des Unterprogramms angegeben wird.

Bei BSR wird – genau wie bei allen Branch-Befehlen – die Adresse als 16-Bit- oder 8-Bit-Adreßdistanz angegeben, das heißt, die Differenz zwischen der Adresse des Unterprogramms und der Adresse, an dem das BSR steht. Diese Art, Adressen anzugeben, ermöglicht relozierbare Programme, da ja die Adreßdistanz zweier Befehle des gleichen Programms immer gleich ist, egal, wo das Programm nun gerade steht. Bei der Ausführung addiert der Prozessor die angegebene Adreßdistanz zum aktuellen Stand des PC, um die Adresse des Unterprogramms zu erhalten.

Nachteil dieser Methode ist jedoch, daß der Bereich, der mittels BSR erreicht werden kann, begrenzt ist. Will man über –32766 bis +32769 Bytes vom Befehl aus gemessen hinaus, so muß JSR verwendet werden. Bei diesem Befehl wird – entsprechend dem JMP-Befehl – meistens die absolute Adresse des Unterprogramms als Langwort angegeben. Allerdings verfügt JSR auch über die PC-relativen Adressierungsarten und noch einige andere.

Eine Alternative zu RTS stellt der Befehl RTR (return from subroutine and restore CCR) dar. Er bewirkt, daß vor dem Rücksprung aus dem Unterprogramm das CCR vom Stack geholt wird. So werden im aufrufenden Programm, durch die Abarbeitung des Unterprogramms, die Flags nicht verändert. Da das CCR beim Aufruf des Unterprogramms nicht automatisch auf den Stack gesichert wird, muß bei der Verwendung dieses Befehls der erste Befehl des Unterprogramms das CCR sichern:

```
Unterprog: MOVE.W SR,-(SP)
```

Beachten Sie, daß das gesamte SR gesichert werden muß (wovon das CCR ja das untere Byte ist), da es keinen Befehl gibt, um allein das CCR-Byte zu lesen. Bei der Ausführung des RTR werden allerdings nur die unteren 8 Bit des gesicherten Wortes ins SR geschrieben, damit das System-Byte unverändert bleibt (Man könnte ja vorher das Wort auf dem Stack manipuliert haben). Also auch hier keine Möglichkeit, vom User-Modus in den Supervisormodus zu gelangen!

Mit Hilfe des Userstacks, der mit Register A7 verwaltet wird, führt der Prozessor die Aufbewahrung der Rücksprungadressen praktisch automatisch durch. Doch um eines muß sich der Programmierer kümmern: Die Übergabe von Parametern zum Unterprogramm.

Parameterübergabe an Unterprogramme

Bei kurzen Unterprogrammen, die keine weiteren Unterprogramme aufrufen, ist wohl die sinnvollste Form der Parameterübergabe, diese einfach in be-

stimmte Register zu laden und dann das Programm aufzurufen. Wir können diese Methode nur dann nicht verwenden, wenn entweder die Anzahl der Parameter die Anzahl der Register übersteigt oder mehrere Unterprogramme geschachtelt werden sollen. Auch hier bietet sich deshalb die Benutzung des Stacks an. Zum Ablegen eines Wortes auf den Stack benutzt man einen Befehl in der Form

```
MOVE D0,-(SP)
```

oder, völlig äquivalent

```
MOVE D0,-(A7)
```

Leider kann das Unterprogramm die Werte nicht einfach in der Form

```
MOVE (SP)+,D2
```

herunterholen, da ja der oberste Eintrag auf dem Stack die Rücksprungadresse des JSR wäre und der Stackpointer im Unterprogramm nicht verändert werden darf. Andernfalls kann der Prozessor die Rücksprungadresse nicht wiederfinden. Deshalb greift man am besten mit der Adressierungsart "Adreßregister indirekt mit Adreßdistanz" auf die Parameter zu:

```
MOVE 4(SP),D2
```

Der Stackpointer zeigt nach dem Aufruf des Unterprogramms direkt auf das erste Byte der gesicherten Rücksprungadresse. Da diese ein Langwort umfaßt, erreicht man vier Bytes höher genau das erste Byte des ersten Parameters. Die Adresse der folgenden Parameter errechnet sich je nach der Länge der vorhergehenden. Nehmen wir an, es sollen ein Wort, ein Langwort und noch ein Wort übergeben werden:

```
          MOVE   parameter3,-(SP)  * dritter Parameter
          MOVE.L parameter2,-(SP)  * zweiter Parameter
          MOVE   parameter1,-(SP)  * erster Parameter
          JSR    routine           * UP aufrufen
          ADDQ.L #8,SP             * Stack korrigieren
          .
          .
          .
routine:  MOVE   4(SP),D0          * Parameter 1 in D0
          MOVE.L 6(SP),D1          * Parameter 2 in D1
          MOVE   10(SP),D2         * Parameter 3 in D2
          .
          .
          .
```

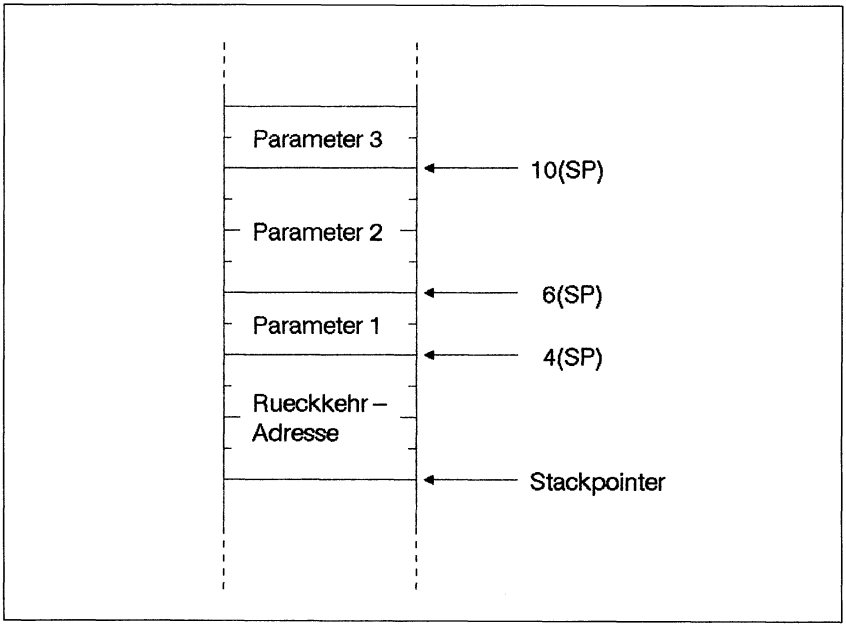


Abb. 2.10: Anordnung der Parameter auf dem Stack

Abbildung 2.10 zeigt die Anordnung der Parameter auf dem Stack nach dem Aufruf des Unterprogramms. Bei dieser Form der Parameterübergabe ist es wichtig, nach der Rückkehr aus dem Unterprogramm den Stack wieder zu korrigieren, das heißt, ihn auf den Stand zu setzen, den er vor dem Ablegen der Parameter hatte. Schließlich wird ja der Stackpointer im Unterprogramm nicht verändert. Deshalb addieren wir zum Stackpointer die Gesamtanzahl von Bytes, die wir als Parameter hinaufgeschoben haben. Die Rücksprungadresse muß dabei nicht mehr beachtet werden, da diese ja schon mit dem RTS-Befehl vom Stack geholt wurde.

Beachten Sie auch, daß der Parameter, der zuletzt auf den Stack geschoben wird, nachher an unterster Stelle steht. Es ist reine Konvention, den letzten Parameter zuerst auf den Stack abzulegen, zumal ja die Reihenfolge und Bedeutung der Parameter ohnehin vom Unterprogramm abhängt. Erwähnenswert sei noch, daß innerhalb eines C-Programms die Parameter ebenfalls genau in dieser Art und Reihenfolge übergeben werden. Dies ist auch deshalb interessant, da die Betriebssystemprozeduren in ähnlicher Weise aufgerufen werden. Doch darauf wird im Kapitel 4 noch ausführlich eingegangen.

Beachten Sie, daß bei der hier vorgestellten Parameterübergabekonvention nur Worte oder Langworte überreicht werden können, aber keine größeren Datenobjekte wie Zeichenketten oder Felder. Will man Prozeduren, die solche Objekte als Eingabe erhalten, trotzdem flexibel gestalten, so empfiehlt sich die Übergabe von Zeigern. Dabei legt der aufrufende Code die Adresse eines solchen Datenobjekts auf dem Stack ab, die vom Unterprogramm wiederum als Zeiger auf den eigentlichen Parameter verwendet wird.

Kontrollstrukturen in Assembler

Der folgende Abschnitt zeigt Ihnen die wichtigsten Kontrollstrukturen der strukturierten Programmierung und wie sie angewendet werden.

IF-THEN-ELSE

In Assembler sind alle Bedingungsabfragen mit Programmsprüngen verbunden. Wir wollen einen Programmteil umsetzen, der in einigen BASIC-Dialekten oder in Pascal etwa so aussieht (Worte in eckigen Klammern stehen natürlich nicht für sich selbst, sondern sind durch entsprechende Befehlsfolgen oder Ausdrücke zu ersetzen):

```
IF [Bedingung] THEN [Befehlsfolge A]
               ELSE [Befehlsfolge B]
ENDIF
```

Wenn die Bedingung wahr ist, wird Befehlsfolge A ausgeführt, andernfalls Befehlsfolge B.

In Assembler wird das so realisiert:

```
IF      [Bedingung testen]
        Bcc ELSE            * Bedingung Falsch
THEN      .
        [Befehlsfolge A]
          .
        BRA ENDIF

ELSE      .
        [Befehlsfolge B]
          .

ENDIF     .
        [Weiter im Programm]
          .
```

Für "Bcc" ist der Branch-Befehl einzutragen, der dann ausgeführt wird, wenn die Bedingung nicht wahr ist. Nehmen wir an, man wollte folgende Befehlsfolge in Assembler umsetzen:

Beispiel:

```
IF a>b THEN b:=a
       ELSE b:=0
ENDIF
```

Nach obigem Schema erhalten wir dann:

```
IF      MOVE  b,D0          * in Register
        CMP   D0,a          * a ? b
        BLE   ELSE          * Wenn a<=b
THEN    MOVE  a,b           * b:=a
        BRA   ENDIF         * weiter im Programm
ELSE    CLR   b             * b:=0
ENDIF     .
          .
```

Natürlich kann der ELSE-Teil auch weggelassen werden. In diesem Fall wird statt zum ELSE-Teil gleich zum ENDIF verzweigt.

Übrigens ist ENDIF kein gültiges Label, da es mit der später besprochenen END-Anweisung kollidiert. Es wurde hier nur der Klarheit halber verwendet.

Realisierung von Schleifen

Fangen wir mit der REPEAT-UNTIL-Schleife an:

```
REPEAT [Befehlsfolge] UNTIL [Bedingung]
```

Die Befehlsfolge wird so lange ausgeführt, bis die Bedingung zutrifft, jedoch mindestens einmal. Nun das gleiche in Assembler:

```
REPEAT    .
        [Befehlsfolge]
          .
UNTIL   [Bedingung testen]
        Bcc   REPEAT        * Wenn Bedingung falsch
REPEND    .
        [Weiter im Programm]
          .
```

Wieder muß die Negation der Bedingung abgefragt werden; die Verzweigung Bcc darf nur dann ausgeführt werden, wenn die Bedingung falsch ist.

Wenn man zwischendurch die Schleife verlassen will, kann man das einfach mit einem

```
BRA     REPEND
```

tun – was natürlich nicht strukturiert ist, aber praktisch sein kann.

Die nächste Schleifenform ist die WHILE-Schleife:

```
WHILE [Bedingung] DO [Befehlsfolge] WEND
```

Solange die Bedingung wahr ist, wird die Befehlsfolge wiederholt. Der Unterschied zur REPEAT-UNTIL-Schleife besteht darin, daß WHILE-Schleifen abweisend sind. Das heißt, daß die Befehlsfolge überhaupt nicht ausgeführt wird, wenn die Bedingung nicht gleich am Anfang zutrifft. In Assembler:

```
WHILE   BRA   entry

DO        .
        [Befehlsfolge]
          .
entry   [Bedingung testen]
        Bcc   DO            * Wenn Bedingung wahr
WEND      .
        [Weiter im Programm]
          .
```

Diesmal muß das Bcc dann ausgeführt werden, wenn die Bedingung wahr ist. Die Abfrage der Bedingung steht hier immer noch physisch am Ende der Schleife. Man könnte sie auch an den Anfang stellen; unser Verfahren hat jedoch den Vorteil, daß dadurch in jedem Schleifendurchlauf ein BRA-Befehl eingespart wird. Auch hier kann natürlich die Schleife jederzeit mit

```
BRA WEND
```

abgebrochen werden.

Nun noch zu einer recht spezialisierten Schleife: der FOR-Schleife. In den meisten BASIC-Dialekten sieht sie so aus:

```
FOR [Zähler]=[Anfang] TO [Ende]
  [Befehlsfolge]
NEXT [Zähler]
```

Sicher ist es manchmal praktisch, wenn diese Schleife abweisend ist, d.h. wenn der Anfangswert gleich beim Eintritt in die Schleife größer ist als der Endwert, wird die Befehlsfolge überhaupt nicht durchlaufen.

Man könnte die FOR-Schleife folgendermaßen in eine WHILE-Schleife umwandeln:

```
[Zähler]=[Anfang]
WHILE [Zähler]<=[Ende]
DO
   [Befehlsfolge]
   [Zähler]=[Zähler]+1
WEND
```

Dementsprechend wird auch die Umsetzung in Assembler vorgenommen, wobei der Zähler sich in D0 befinden soll:

```
FOR     MOVE  [Anfang],D0
WHILE   BRA   TEST
DO        .
        [Befehlsfolge]
          .
        ADDQ  #1,D0
TEST    CMP   [Ende],D0
        BLE   DO
WEND      .
          .
          .
```

Natürlich könnte man hier leicht die Schrittweite von 1 auf einen anderen Wert verändern. Nur bei der Verwendung einer negativen Schrittweite gilt es zu beachten, daß entsprechend auch Ende kleiner als Anfang ist und der Verzweigungsbefehl BLE deshalb durch BGE ersetzt werden muß.

Bei diesem Skelett einer FOR-Schleife wurde davon ausgegangen, daß Anfang und Ende Werte sind, die direkt in bestimmten Variablen stehen. Handelt es sich jedoch um Ausdrücke, so sollten sie vor dem Eintritt in die Schleife berechnet und irgendwo abgelegt werden, denn es wäre unpraktisch, den Ausdruck für jede Abfrage neu zu berechnen.

Organisation von ATARI ST-Programmen

Anders als bei vielen kleineren Computern haben ausführbare Programme auf dem ATARI ST eine klar gegliederte Struktur: Jedes Programm ist in drei sogenannte Segmente (Sections) unterteilt, die unterschiedliche Funktionen haben (Abb. 2.11). Zunächst wäre da die "Text Section". Dies hat nichts mit lesbarem Text zu tun, sondern steht für den eigentlichen Programmcode. Natürlich braucht jedes Programm auch Variablen – also Daten. Hier wird zwischen initialisierten und nicht initialisierten Daten unterschieden. Die initialisierten

Daten, also Konstanten oder Variablen, deren Anfangswert schon vor dem Start des Programms feststehen muß, werden in der "Data Section" abgespeichert. Andere Daten, deren Wert sich erst im Laufe des Programms ergibt, sollten in der "BSS Section" (BSS: Block Storage Section) abgespeichert werden. Die Unterteilung der drei Segmente wird im Quellcode vom Programmierer angegeben.

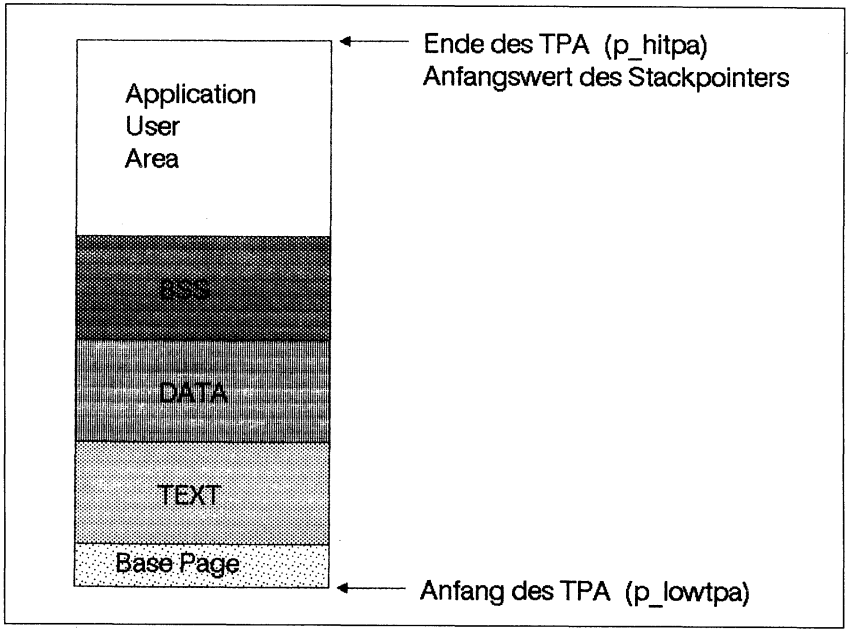


Abb. 2.11: Gliederung eines Programms und GEMDOS

Die Aufteilung in Daten- und Textsegment erscheint wirklich nicht sehr sinnvoll, da die beiden Bereiche völlig gleich behandelt werden. Auch der Assembler bringt keine Fehlermeldungen, wenn man Daten ins Textsegment oder Befehle ins Datensegment schreibt. Man braucht die Aufteilung in Daten- und Textsegment nicht unbedingt vorzunehmen, aber weil dazu wenig Aufwand nötig ist, kann man es genausogut tun.

Die Einführung der BSS-Section ist aber ganz gewiß sinnvoll, da dieses Segment nicht zusammen mit den anderen beiden Segmenten im Programmfile abgespeichert wird. Ein anderes Argument für die Benutzung der BSS-Section ist, daß dadurch der Programmcode etwas übersichtlicher wird.

Einen Bereich aus Abb. 2.11 haben wir noch nicht besprochen: die Basepage. Hier merkt sich das Betriebssystem Informationen über das Programm. So wird die Möglichkeit geschaffen, daß ein Programm ein anderes aufruft, um nach der Beendigung seines Unterprogramms normal weiterarbeiten zu können. Diese Informationen werden in 256 Bytes direkt vor dem Anfang des Textsegments abgelegt. Welche Informationen dort stehen, darauf wird später in diesem Abschnitt noch eingegangen.

Wenn ein Programm gestartet wird, dann wird der gesamte restliche Speicher als "Application User Area" deklariert, also als Speicher, mit dem das Programm irgend etwas anfangen kann. Der Stackpointer wird auf das letzte Byte dieses Speicherbereiches gesetzt, von wo aus er nach unten erweitert werden kann.

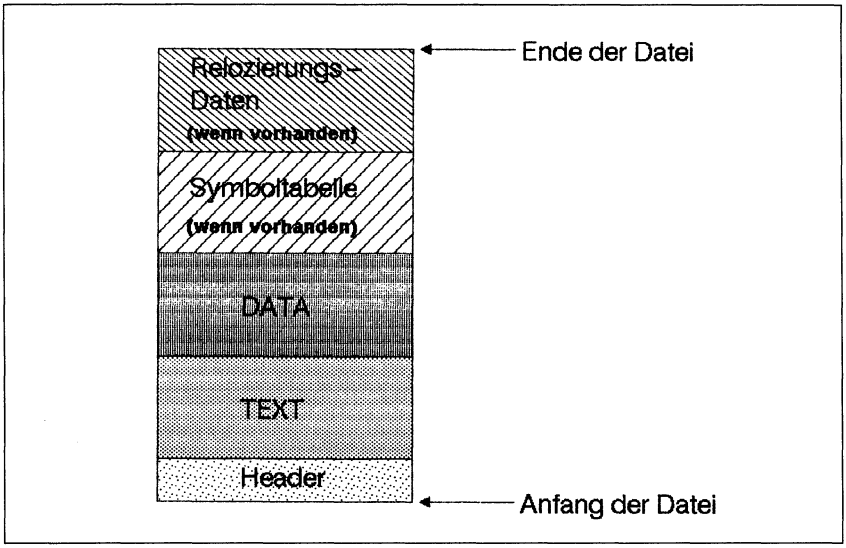


Abb. 2.12: Format einer ausführbaren Datei unter GEMDOS

Als Programmdatei auf der Diskette sieht das Ganze etwas anders aus (Abb. 2.12). Hier finden wir zwar auch Text- und Datensegment. Jedoch zusätzlich am Anfang, im sogenannten Header, befinden sich noch einige Informationen über das Programm, womit hauptsächlich die Längen der 3 Segmente gemeint sind. Nach dem Text- und Datensegment folgt noch ein Abschnitt: die Relozierungs-Daten. Wie schon oben gezeigt, muß es möglich sein, mehrere Programme gleichzeitig im Speicher zu halten. Daraus ergibt sich, daß ein Pro-

gramm nicht immer an die gleiche Adresse geladen wird. Da aber der Code des 68000 nicht von sich aus relozierbar ist und es durch die gravierenden Einschränkungen der PC-relativen Adressierungsart auch nicht sein kann, muß eben das Betriebssystem direkt nach dem Laden des Programms alle Adressen, die nicht von vornherein PC-relativ sind, neu berechnen. Somit stellen die Relozierungs-Daten eine vom Assembler erzeugte Liste von Adressen dar, die all jene Speicherplätze ausweist, für die eine absolute Adresse berechnet werden muß. Um deren Aufbau und Funktion braucht sich der Programmierer jedoch nicht weiter zu kümmern. Wichtig ist nur, daß der vom Assembler erzeugte Code, wie er im Assemblerlisting erscheint, an der logischen Adresse Null beginnt. Die Umrechnung auf physikalische Adressen erfolgt erst direkt vor der Ausführung.

Wahlweise kann den Relozierungsdaten noch eine Symboltabelle folgen, die alle definierten Labels, ihre Art und ihren Wert enthält. Diese wird nur dann angelegt, wenn beim Linken eine bestimmte Option angegeben wird. Bei der Ausführung des Programms hat diese Tabelle keine Bedeutung. Nur ein Debugger kann damit etwas anfangen.

Was geschieht nun, wenn ein Programm unter TOS gestartet wird?

Zunächst schaut das Betriebssystem im Fileheader der Programmdatei nach, wie lang das Programm ist, und teilt dem Programm den gesamten verfügbaren Speicherbereich zu. Dann wird der Programmcode, also Text- und Datensegment, geladen, und die Adressen werden nach den Relozierungsdaten umgerechnet: Das Programm wird reloziert. Nun muß nur noch die Basepage eingerichtet und der Stackpointer initialisiert werden. Beim Start des Programms wird der Userstackpointer auf das obere Ende des freien Speicherbereiches gesetzt. Die Adresse der Basepage wird als Langwort auf dem Stack abgelegt. Darüber wird noch ein weiteres Langwort abgelegt, auf das wir hier aber nicht weiter eingehen. Der Befehl, um an die Adresse der Basepage zu gelangen, würde also so aussehen:

```
progstart: MOVE.L 4(SP),A5
```

Somit steht die Adresse der Basepage in A5.

Organisation der Basepage

Die folgende Tabelle gibt für jeden Eintrag in der Basepage die dezimale Distanz in Bytes zum Anfang, den offiziell dokumentierten Namen und die Bedeutung an (es handelt sich ausnahmslos um Langworte):

Distanz	Name	Beschreibung
00	p_lowtpa	Anfangsadresse des TPA
04	p_hitpa	Endadresse des TPA + 1
08	p_tbase	Anfangsadresse des Textsegments
12	p_tlen	Länge des Textsegments in Bytes
16	p_dbase	Anfangsadresse des Datensegments
20	p_dlen	Länge des Datensegments in Bytes
24	p_bbase	Anfangsadresse des BSS-Segments
28	p_blen	Länge des BSS-Segments in Bytes
32	p_env	Pointer auf den Environment-String

Mit TPA ist der "Transient Program Area" gemeint, also ein Bereich für nicht fest im Betriebssystem verankerte Programme". Somit steht TPA letztlich für nichts anderes als den gesamten Speicherplatz, der nicht vom Betriebssystem oder von vorher schon gelaufenen Programmen belegt wird: also der Speicher, der vor dem Laden eines Programms noch frei ist.

Sicher haben Sie schon einmal ein Programm vom Typ TTP (Tos Takes Parameters) aufgerufen, bei dem die Benutzeroberfläche die Möglichkeit bot, dem Programmaufruf Parameter mitzugeben. Der String, den Sie dort eingeben, muß dem Programm nun irgendwie zugänglich gemacht werden. Er wird einfach in der Basepage ab Distanz 128 (hexadezimal $80) abgelegt, wo er in der C-üblichen Konvention mit einem Nullbyte abgeschlossen wird. Daraus ergibt sich, daß Parameter-Strings bis zu 127 Zeichen lang sein können.

Dem gestarteten Programm sämtlichen verfügbaren Speicherplatz zuzuordnen hat einige Nachteile:

- Da kein Speicher mehr verfügbar ist, kann das Programm keine anderen Programme mehr aufrufen.

- Accessories oder Hintergrundprogramme wie etwa Druckerspooler funktionieren unter Umständen nicht mehr.

- Es können keine Speicherblocks mit der Betriebssystemfunktion malloc() reserviert werden.

- Das Programm könnte bei zukünftigen, eventuell Multitasking-fähigen Versionen des TOS nicht mehr korrekt laufen.

Deshalb ist es für jedes größere Programm empfehlenswert, nur den Teil des Speichers zu reservieren, der wirklich gebraucht wird. Dazu stellt das Be-

triebssystem eine Routine namens "setblock" zur Verfügung, die einen Speicherbereich für das Programm reserviert, der durch seine Anfangsadresse und seine Länge in Bytes definiert wird. Genaugenommen wird vor dem Start ein Block, der eben den gesamten freien Bereich umfaßt, für das Programm reserviert. Wird nun besagte Funktion "setblock" mit der Adresse eines Blocks aufgerufen, der schon existiert, dann wird damit gleichzeitig die vorherige Reservierung rückgängig gemacht und die neue Länge als Länge des Blocks eingesetzt. Deshalb sollten Sie an den Anfang größerer Programme eine Initialisierung dieser Art voranstellen, die die Programmlänge berechnet, den unbenutzten Speicherplatz freigibt und den Stackpointer auf einen korrekten Wert setzt:

```
start:
        MOVE.L    4(SP),A5     * Basepageadresse in A5
        MOVE.L    12(A5),D0    * Länge des Textsegments...
        ADD.L     20(A5),D0    * + Länge des Datensegments...
        ADD.L     28(A5),D0    * + länge des BSS-Segments...
        ADD.L     #$1100,D0    * + 4K (=$1000) für den Stack...
                               * + 256 (=$100) Bytes für die
                               * Basepage
        MOVE.L    A5,D1        * neuer SP = Basepageadresse...
        ADD.L     D0,D1        * + berechnete Länge...
        AND.L     #-2,D1       * auf gerade Adresse abrunden
        MOVE.L    D1,SP        * in den Stackpointer damit
        MOVE.L    D0,-(SP)     * Länge des reservierten Bereichs
        MOVE.L    A5,-(SP)     * Anfangsadresse des Bereichs
        CLR       -(SP)        * überflüssiger Parameter (Dummy)
        MOVE.W    #$4A,-(SP)   * GEMDOS Funktion setblock
        TRAP      #1           * Aufruf des GEMDOS
        ADD.L     #12,SP       * Stack wiederherstellen

init:     .                    * hier kann es mit dem
          .                    * Programm richtig losgehen
```

Zu diesem Programmsegment sind sicher einige Erklärungen erforderlich. Zunächst wird die Adresse der Basepage – wie oben beschrieben – vom Stack geholt. Dann stellt das Programm die Gesamtlänge des Speichers, die benötigt wird, fest. Dieser Wert muß vom Programm folgendermaßen errechnet werden:

256 (=$100) Bytes für die Basepage
+ Länge des Textsegments
+ Länge des Datensegments
+ Länge des BSS-Segments
+ ein beliebig gewählter Wert für den Stackbereich, der hier auf 4 Kilobytes gesetzt wurde

Daraufhin wird der Userstackpointer (das Programm befindet sich beim Start immer im User-Modus) auf das Ende dieses Bereiches gesetzt und die Be-

triebssystemfunktion setblock aufgerufen, um den Programmspeicherbereich zu reservieren. Würde man die Funktion "setblock" von einer höheren Programmiersprache wie etwa C oder Pascal aus aufrufen, so würde das etwa so aussehen:

```
setblock(dummy,Adresse,Länge)
```

wobei "dummy" ein Wort ist, dem keine Bedeutung zukommt, das aber trotzdem vorhanden sein muß, "Adresse" die Anfangsadresse des Blocks ist und "Länge" eben die Länge des Blocks als Langwort. Die Parameterübergabe funktioniert genauso, wie es im vorherigen Abschnitt besprochen wurde, nur der Aufruf des GEMDOS wird hier etwas anders durchgeführt, nämlich mit dem TRAP-Befehl.

Man kann den TRAP-Befehl mit dem JSR-Befehl vergleichen, nur daß hier nicht die Zieladresse im Befehl angegeben wird, sondern nur die Nummer eines Zeigers, in dem die Zieladresse steht. Der Zeiger steht am Anfang des Speichers und wird bei der Initialisierung des Betriebssystems dort hingeschrieben. So wird auch im Fall einer Änderung der Betriebssystemadressen sichergestellt, daß die Programme immer den richtigen Einsprungpunkt finden. Auf die genaue Funktionsweise der Systemaufrufe wird noch in Kapitel 4 eingegangen.

Obige Befehlssequenz wird allerdings in den Programmen aus diesem Buch keine Verwendung finden, da ihre Anwendung nur bei wirklich großen Programmen sinnvoll ist. Zur Verwendung in Ihren eigenen Programmen befindet sich auf der Diskette die Datei MEMINIT.S, die genau obige Kommandosequenz enthält und zum Einfügen in Assemblerprogramme gedacht ist.

Wenn man ein Programm beendet, muß man es dem Betriebssystem mitteilen, damit dieses den vom Programm belegten Speicherplatz wieder freigeben und zum Desktop (oder zum Programm, das es aufgerufen hat) zurückkehren kann. Auch dafür gibt es eine Betriebssystemfunktion:

```
ende:  MOVE #0,-(SP)     * GEMDOS Funktion 0: term()
       TRAP #1           * GEMDOS aufrufen
       ...               * hier kommt man niemals hin
```

Grundlagen der Bedienung eines Assemblers

Nun wird es ernst: Wir wollen uns ansehen, wie Programme in der Praxis eingegeben, assembliert und gestartet werden.

Befehle

Jeder vollständige Assembler versteht sämtliche Befehlskürzel. Da die Bezeichnungen der Prozessorbefehle und der Adressierungsarten vom Hersteller verbindlich festgelegt werden, sollten damit normalerweise keine Probleme auftreten. Im allgemeinen stellen Assembler bestimmte Ansprüche an die Form, in der ein Befehl eingegeben wird. Deshalb soll hier noch einmal in aller Deutlichkeit gesagt werden, wie ein Befehl aussehen muß:

Es gibt zwei Möglichkeiten, wie eine Zeile aussehen darf:

```
[Label] [Befehl] [Operanden]
```

oder einfach

```
[Befehl] [Operanden]
```

Im ersten Fall muß der Assembler irgendwie das Label von einem Befehl unterscheiden können, denn man kann im Prinzip Mnemonics als Label mißbrauchen. Das sollten Sie aber besser nicht tun, da es leicht zu Verwechslungen führen kann. Label werden nun als solche gekennzeichnet, indem sie wahlweise direkt beim ersten Zeichen auf der Zeile beginnen oder aber irgendwo beginnen und von einem Doppelpunkt gefolgt sind. Die folgenden drei Schreibweisen sind also erlaubt:

```
hier:   LSR     #1,D0
```

oder

```
hier    LSR     #1,D0
```

oder

```
   hier: LSR     #1,D0
```

Ein Label darf auch ganz allein auf einer Zeile stehen; es wird dann so interpretiert, als stünde es vor dem folgenden Befehl. Also gibt es noch eine vierte Schreibweise:

```
hier:
        LSR #1,D0
```

In einem Label sind Groß- und Kleinbuchstaben, Ziffern, Punkt (.) und der Unterstrich (_) erlaubt, wobei zwischen Groß- und Kleinschreibung unterschieden wird. Beim Assembler aus dem Entwicklungspaket dürfen Label beliebig lang sein; es werden jedoch nur die ersten acht Zeichen beachtet.

Der Assembler behandelt ein Label so, daß er ihm die Adresse des darauffolgenden Befehls zuweist. Label dürfen nur einmal definiert, aber beliebig oft benutzt werden (referenziert).

Nun zum nächsten Feld, dem Befehlsfeld: Wenn die erste Form verwendet wird, müssen Label und Befehl durch mindestens ein Leerzeichen getrennt werden. Bei der zweiten Form darf der Befehl nicht auf dem ersten Zeichen der Zeile beginnen, damit er nicht für ein Label gehalten wird. Übrigens ist es bei Mnemonics im allgemeinen egal, ob man sie groß oder klein schreibt, was auch für Registernamen gilt. Sie könnten also statt

```
MOVE.B    D0,-(SP)
```

auch schreiben

```
move.b    d0,-(sp)
```

oder, wenn Sie es extravagant lieben

```
MoVe.B    d0,-(sP)
```

In diesem Buch werden bei Befehlsfolgen innerhalb des Textes großgeschriebene Mnemonics benutzt; in größeren Listings werden jedoch wegen der etwas besseren Lesbarkeit kleingeschriebene Mnemonics verwendet. Ob man nun Groß- oder Kleinschreibung wählt, ist Geschmackssache.

Das Operandenfeld wird durch mindestens ein Leerzeichen vom Befehlsfeld getrennt. Als Registernamen erkennt der Assembler:

D0 – D7	Datenregister
A0 – A7	Adreßregister
SP	Stackpointer (=A7)
CCR	Condition Code Register (Userbyte)
SR	Status-Register (Systembyte und Userbyte)
USP	User Stack Pointer
PC	Program Counter, Befehlszähler

Im Operandenfeld selbst dürfen keine Leerzeichen stehen.

Wußten Sie schon, daß der Assembler auch rechnen kann? Er kann es, denn manchmal kann ein Befehl in der Art

```
MOVE.L #$FF,liste+4
```

sinnvoll sein. Hier will man, daß an die Adresse, die sich aus der Summe von 4 und dem Wert des Symbols ergibt, etwas geschrieben wird. Der Assembler kann noch einiges mehr als die simple Addition. Hier finden Sie eine Liste der Operatoren in der Reihenfolge ihrer Priorität:

- \- monadisches Minus (negativer Wert)
- << verschieben nach links (entsprechend der Wirkungsweise von ASL)
- >> verschieben nach rechts (entspricht ASR)
- & logisches UND der einzelnen Bits (entspricht AND)
- ! logisches ODER der einzelnen Bits (entspricht OR)
- * Multiplikation
- / Division
- \+ Addition
- \- Subtraktion

Der Assembler versteht auch beliebig geklammerte Ausdrücke. Einen Umstand gilt es in manchen Fällen zu beachten: Der Assembler arbeitet nur mit ganzzahligen Werten, nicht etwa mit Kommazahlen, wie man es beispielsweise vom BASIC-Interpreter gewohnt ist. Dies schlägt sich darin nieder, daß die Division mit Vorsicht zu genießen ist, denn sie hat immer ein ganzzahliges Ergebnis. So können etwa die Ausdrücke

```
index/4*laenge
```

und

```
index*laenge/4
```

verschiedene Ergebnisse haben, da Multiplikation und Division als gleichwertige Operatoren von links nach rechts abgearbeitet werden. Im ersten Fall wird daher erst die Division durchgeführt, wobei der Rest unter den Tisch fällt, und dann das Ergebnis mit "laenge" multipliziert, während beim zweiten erst das fertige Produkt durch 4 geteilt wird.

Verwechseln Sie die Berechnungen des Assemblers nicht mit den Berechnungen des Programms, denn natürlich kann der Assembler nur mit Werten rechnen, die zur Assemblierzeit schon bekannt sind. Werte von Variablen fallen logischerweise nicht darunter.

Neben den bisher verwendeten Dezimalzahlen können auch binäre, oktale oder hexadezimale Zahlen angegeben werden, die jeweils durch ein bestimmtes Zeichen gekennzeichnet werden:

```
%    binäre Zahl:        MOVE    #%00110011,D0
@    oktale Zahl:        MOVE    #@377,D0
$    hexadezimale Zahl:  MOVE    #$E477,D0
```

Diese Art von Zahlen darf auch in beliebigen Ausdrücken auftauchen. Eine weitere Art, eine Konstante anzugeben, ist die Form des ASCII-Wertes. Dabei geben Sie in Anführungsstrichen einen Buchstaben an, für den sein ASCII-Wert eingesetzt wird.

Beispiel:

```
SUB.B     #"A",D0
```

erzeugt den gleichen Code wie

```
SUB.B     #65,D0
```

denn 65 ist der ASCII-Code des Zeichens "A".

Nach dem Operandenfeld (das bei einigen Befehlen auch wegfallen kann) darf noch eine Bemerkung stehen, die nach der Konvention von Motorola nicht speziell gekennzeichnet werden muß. Der Assembler ignoriert einfach alles, was nach dem Operandenfeld kommt, bis eine neue Zeile begonnen wird. Wenn allerdings eine ganze Zeile nur als Bemerkung dienen soll, so muß an der ersten Stelle ein Stern (*) stehen:

```
* Dies ist eine Bemerkung
```

Eine so gekennzeichnete Zeile wird vom Assembler völlig ignoriert.

Bei manchen Assemblern wird jede Bemerkung mit einem Semikolon eingeleitet. Dort muß das Semikolon auch dann vorhanden sein, wenn nach den Operanden noch eine Bemerkung folgt. In diesem Buch wird allerdings nur die oben beschriebene Art benutzt, da der recht verbreitete Assembler aus dem ATARI-Entwicklungspaket nicht mit Semikolons arbeitet.

Bei den Programmen dieses Buchs werden auch Bemerkungen hinter Befehlen mit einem Stern eingeleitet. Nach obiger Konvention ist das eigentlich unnötig; es ist nur dazu gedacht, die Bemerkungen vom Programmcode abzuheben. Wie Sie sehen, läßt Ihnen der Assembler die Möglichkeit, Ihre Zeilen zu gestalten, wie es Ihnen gefällt, sofern es nicht zu Mißverständnissen führen kann.

Übersichtlich wird ein Programm dadurch, daß man jeden Abschnitt durch eine Tab-Position festlegt, also einen für den Zeilenanfang für Labels, ein Tabulator-Stop für Befehle, einer für die Operanden und ein weiterer für eventuelle Bemerkungen.

Assembler-Direktiven

Manchmal will man den Assembler auch direkt ansprechen, z.B."Schreibe den assemblierten Code an die und die Stelle" oder "Hier beginnt ein neues Segment". Dafür gibt es die Assembler-Direktiven. Sie erinnern an Befehlscodes, nur erzeugen sie keinen Maschinencode, sondern wenden sich direkt an den Assembler. Glücklicherweise hat Motorola auch für diese Direktiven einen Standard gesetzt, an den sich tatsächlich auch die meisten Assembler halten! Deshalb kann hier ein Überblick über die Direktiven gegeben werden, der mit einiger Wahrscheinlichkeit auch für Ihren Assembler gilt (die Abweichungen der verbreitetsten Assembler finden Sie in Anhang B).

Direktiven müssen – genau wie Mnemonics – mindestens durch ein Leerzeichen vom Zeilenanfang getrennt sein. Die meisten können allerdings auch mit einem Label versehen werden, wobei dann der Wert des Labels die Adresse der ersten Speicherstelle ist, die durch die Direktive angesprochen wird.

Oftmals will man Symbolen einen konstanten Wert zuweisen. Dafür verwendet man die Anweisung EQU:

EQU equal

Einem Symbol wird ein Wert zugewiesen. Auf der linken Seite steht ein Symbolname nach den für Labels üblichen Konventionen, rechts eine Zahl, ein anderes Symbol oder ein Ausdruck.

Beispiele:

```
tablen   EQU      99
fehler:  EQU      -1
chaos    EQU      ("A">>2+3)!%110101100-tablen
```

An dieser Stelle ist es sinnvoll, darauf hinzuweisen, daß die meisten Assembler zwischen sogenannten relativen und absoluten Symbolen unterscheiden. Es gibt zunächst einmal zwei Möglichkeiten, wie ein Symbol zu seinem Wert kommen kann:

– durch Angabe einer Zahlenkonstante
  Dies ist sinnvoll, wenn man etwa bestimmte Konstanten beim Namen nennen will. So erspart man es sich, bei einer Änderung des Programms den Wert an allen Stellen zu ändern, an denen er verwendet wird, etwa in der Form:

  ```
  tablen EQU 200
  ```

  Überall nachfolgend wird jetzt "tablen" als Synonym für die Konstante 200 betrachtet.

– durch Setzen des Labels auf eine Adresse
  Hiermit ist einfach die Verwendung eines Labels als Markierung gemeint.

Alle anderen möglichen Belegungen von Symbolen ergeben sich als das Resultat von Operationen auf diese beiden Grundtypen.

Der erste Typ erzeugt einen absoluten Wert, das heißt, der Wert wird als konstant betrachtet, egal wo das Programm im Speicher nun gerade abläuft. Wenn ein solcher absoluter Wert als Adresse betrachtet wird, auf die man zugreift oder die angesprungen wird, dann wird beim Relozieren des Programms nichts daran geändert. Das ist sinnvoll, sofern man auf feste Speicherplätze – wie etwa die Systemvariablen – zugreifen will. Absolute Werte können nicht bei Befehlen verwendet werden, die relative Adressen als Operanden verlangen.

Die andere Art der Symboldefinition führt hingegen zu relativen Symbolen. Das heißt, jede Verwendung eines solchen Symbols, die nicht von vornherein PC-relativ ist, wird vom Assembler vermerkt, um beim Relozieren vor dem Start des Programms berücksichtigt zu werden. Da es sich bei relativen Werten nur um Adressen handeln kann, gestattet der Assembler im allgemeinen ihre Verwendung als Zahlenwert nur dort, wo eine 32-Bit-Zahl angegeben werden kann. So ist zum Beispiel folgendes nicht erlaubt:

```
label1:  .
         .
         .
         MOVE.W    #label1,A0
```

Der Assembler würde mit einer Fehlermeldung oder zumindest mit einer Warnung darauf hinweisen.

Es gelten genau festgelegte Gesetze, was die Verknüpfung dieser beiden Arten von Symbolen ergibt. Um es genau zu erfahren, schlagen Sie am besten in Ihrem Assemblerhandbuch nach. Als Faustregeln möge jedoch folgendes gelten:

– Ein absolutes Symbol verknüpft mit einem absoluten Symbol ergibt immer ein absolutes Symbol.

– Operationen mit einem absoluten und einem relativen Symbol ergeben immer ein relatives Symbol, sofern sie sinnvoll und erlaubt sind.

– Einzige erlaubte Operation mit zwei relativen Symbolen ist die Subtraktion, wobei das Resultat ein absoluter Wert ist.

Im allgemeinen brauchen Sie sich aber um diese Regeln nicht weiter zu kümmern, da es der Assembler meistens so macht, wie man es erwarten würde, und andernfalls wird mit einer Fehlermeldung auf das Problem hingewiesen.

Recht häufig sieht man in Programmen auch die Direktiven DC und DS, die – in Analogie zu den Befehlen – auch mit den Anhängseln ".B", ".W" und ".L" versehen werden dürfen. Ihre Bedeutung:

DC Define Code

Diese Direktive legt einen oder mehrere Werte einfach hintereinander im Speicher ab, entsprechend der Initialisierung von Variablen in einigen Compilersprachen. Es können, durch Kommata getrennt, ein oder mehrere Zahlen bzw. Ausdrücke angegeben werden, die nacheinander im Speicher abgelegt werden:

```
DC.L      123456,-1,$31415927,zahl&%11110000
```

Wie üblich wird Wortbreite angenommen, wenn kein Anhängsel angegeben wird.

Bei der Variante DC.B gibt es noch eine Besonderheit: Hier kann auch eine Zeichenkette als Operand angegeben werden. Die Zeichen werden einfach in "Anführungszeichen" oder 'Hochkommas' eingeschlossen, was dem Assembler befiehlt, sie in aufeinanderfolgenden Bytes abzulegen. Strings können – durch Kommas getrennt – auch mit anderen Operandenarten gemischt werden:

```
meldung: DC.B "Fehler Nummer 99",0
```

Hier werden die ASCII-Werte für die Buchstaben "F", "e", "h" und so weiter hintereinander abgelegt. Das Label "meldung" bezeichnet dabei die Adresse des ersten Zeichens des Strings. Würde man also den Inhalt der Adresse "meldung" auslesen, so erhielte man den ASCII-Wert von "F". Die Null weist den Assembler an, hinter dem String noch ein Nullbyte zu erzeugen. Dies ist die TOS-übliche Konvention, Strings anzugeben: Das Ende des Strings wird durch ein Nullbyte markiert. Dies kommt nicht zuletzt daher, daß ein großer Teil des Betriebssystems TOS in der Hochsprache C geschrieben wurde, denn in dieser Compilersprache werden Strings genauso verarbeitet. Wenn eine Zeichenkette also zu irgendeinem Zeitpunkt dem Betriebssystem übergeben werden soll, sollte sie durch ein Nullbyte abgeschlossen werden.

Achtung Falle! Achten Sie bei Byte-Daten darauf, ob die Anzahl der abgelegten Bytes nicht zufällig ungerade ist, denn manche Assembler versuchen sonst, etwa nachfolgende Daten im Wort- oder Langwortformat an ungeraden Adressen abzulegen. Der Prozessor kann so bekanntlich in arge Bedrängnis kommen. Um sicher zu gehen, geben Sie deshalb nach Strings die Direktive

```
EVEN
```

an, die nichts weiter tut, als den Programmzähler auf die nächste gerade Adresse zu setzen.

DS Define Storage (reserviere Speicher)

Diese Direktive reserviert einen Speicherbereich entsprechend den nicht initialisierten Variablen in Hochsprachen. Die Größe des Bereichs wird als Ausdruck angegeben. Auch hier gibt es wieder die drei Anhängsel ".B", ".W" und ".L", die die Art der Werte angeben, für die Platz reserviert werden soll. So haben etwa die folgenden Anweisungen die gleiche Wirkung:

```
array:  DS.B   200
array:  DS.W   100
array:  DS.L   50
```

In allen drei Fällen werden 200 Bytes reserviert, wobei das Label "array" für die Adresse des ersten Bytes steht. Allerdings sollten Sie nicht vergessen, daß in dem so reservierten Speicher zunächst einmal irgendwelche zufälligen Werte stehen, sofern er sich im Text- oder Datensegment befindet. Sie sollten also zuerst etwas hineinschreiben, bevor etwas gelesen wird.

Übrigens ist DC die einzig erlaubte Möglichkeit, Speicherplätze im BSS-Segment anzusprechen, da das BSS-Segment nicht abgespeichert wird. Andererseits sollten möglichst alle DC-Direktiven im BSS-Segment stehen, um den Speicherverbrauch des Programms auf der Diskette zu reduzieren und die Ladezeiten zu verkürzen. Bei Variablen im BSS kann man sich darauf verlassen, daß sie beim Programmstart mit Nullen gefüllt worden sind. Dafür sorgt das Betriebssystem, bevor dem Programm die Kontrolle übergeben wird.

Eines muß dem Assembler noch mitgeteilt werden: Wie die einzelnen Segmente unterteilt sind. Für jedes Segment gibt es eine Direktive:

TEXT Hier beginnt das Textsegment. Allerdings ist diese Direktive meist nicht erforderlich, da der Assembler am Anfang des Programms

standardmäßig alles als TEXT auffaßt. Nötig ist diese Direktive also nur, wenn man mit einem anderen als dem Textsegment beginnt.

DATA Hier beginnt das Datensegment.

BSS Hier beginnt das Block Storage Segment.

END Hier ist das Programm zu Ende. Alles folgende wird vom Assembler ignoriert. Stellen Sie die END-Direktive immer ans Ende Ihrer Programme, da die meisten Assembler mit einer Fehlermeldung oder zumindest mit einer Warnung reagieren, wenn die END-Direktive fehlt.

Wenn Sie wollen, können Sie jede der drei Segment-Direktiven in einem Programm auch mehrmals verwenden. Der Assembler fügt dann die Daten für je ein Segment beim Assemblieren wieder zusammen, so daß es zuletzt doch nur drei Segmente sind. Zu diesem Zweck verfügt der Assembler gleich über drei "location counter", also Adreßzähler, nämlich für jedes Segment einen. Alle drei stehen am Anfang auf Null. Wenn jedoch eine dieser drei Direktiven zum zweiten Mal gefunden wird, so geht der Assembler vom alten Zählerstand des entsprechenden Segments aus, hängt also die neuen Daten an.

In diesem Buch werden Programme immer in der Reihenfolge Textsegment-Datensegment-BSS gegliedert, was aus der Sicht des Assemblers zwar völlig willkürlich, aber allgemein üblich ist.

Wichtig sind noch die Direktiven XDEF und XREF, die eine Möglichkeit zur Modularisierung bieten.

XDEF eXternal DEFinition. Diese Direktive macht Labels auch außerhalb des Moduls erreichbar. Als Argument werden ein im Modul definiertes Label oder, durch Kommata getrennt, mehrere Labels angegeben.

XREF eXternal REFerence. Dies ist das Gegenstück zu XDEF. XREF macht ein Label zugänglich, das aus einem anderen Modul mit XDEF exportiert worden ist. Fortan kann dieses Label dann angesprochen werden, als wäre es im importierenden Modul definiert. Auch hier wird ein Label oder eine Liste von Labels angegeben.

XREF und XDEF werden vom Assembler nur in der Form behandelt, daß dieser eine Notiz für den Linker hinterläßt, die diesem sagt, daß ein Symbolname für die allgemeine Verwendung freigegeben ist bzw. in den anderen Modulen, die zusammengebunden werden, zu suchen ist. In allen Modulen, die dem Lin-

ker übergeben werden, darf dasselbe Label nur einmal mittels XDEF definiert, jedoch beliebig oft referenziert (XREF) werden. Wird allerdings ein Label mehrfach definiert oder aber referenziert, obwohl es nicht definiert ist, so führt das zu einem Fehler beim Linken, und es wird kein lauffähiges Programm erzeugt. Mit XDEF werden meistens die Adressen von Funktionen exportiert, seltener Variablen.

Beispiel:

Zunächst werden Symbole in einem Modul definiert:

```
        XDEF    funa,funb

        TEXT
funa:   .
        .
        .
funb:   .
        .
        .
        END
```

Ein anderes Modul kann dann folgendermaßen darauf zugreifen:

```
        XREF    funb

        TEXT
modul2:
        .
        .
        .
        JSR     funb
        .
        .
        .
        END
```

Bei größeren Programmen ist die Modularisierung ein praktisches Mittel, nicht nur die Übersichtlichkeit des Programms zu erhöhen, sondern auch die Effizienz bei der Programmentwicklung zu steigern. Wenn ein großes Programm in mehrere thematisch abgegrenzte Module aufgeteilt wird, so braucht immer nur das Modul neu assembliert zu werden, an dem gerade gearbeitet wird. Die anderen Module werden als Objektdateien irgendwo aufbewahrt und können vom Linker für jeden Testlauf zum neu assemblierten Modul gebunden werden, was natürlich wesentlich schneller geht, als jedesmal alles zu assemblieren. Abgesehen davon hat man oft keine andere Wahl, da einige Programmeditoren (etwa ED.TTP aus dem ATARI-Entwicklungssystem) ohnehin keine Programmdateien verwalten, die länger als 32 KByte sind.

Soviel zu den wichtigsten Direktiven. Es gibt noch zur Schreibweise einiges zu sagen:

Die meisten Assembler akzeptieren Direktiven wahlweise in Groß- oder Kleinschreibung. Einige Assembler verlangen vor jeder Direktive einen Punkt. Was also in diesem Buch folgendermaßen geschrieben wird:

```
DATA
DC 123,$ffff
```

müßte man bei anderen Assemblern so schreiben:

```
.data
.dc 123,$ffff
```

Leider wird die Konvention der Direktiven nicht so gründlich eingehalten wie die der Mnemoniks. So kann es sein, daß Sie auch die Bezeichnungen der Direktiven an Ihren Assembler anpassen müssen.

Das erste lauffähige Programm

Am besten erkennt man das Zusammenspiel dieser Direktiven an einem Beispiel. Das erste lauffähige Programm tut nichts anderes, als den Spruch "Hallo, hier bin ich" auf den Bildschirm zu bringen:

```
* Programm "Hallo Welt"
*
* Gibt nur auf dem Bildschirm aus "Hallo, hier bin ich!"
* und wartet dann auf einen Tastendruck
*
* zunächst werden erst einmal Konstanten definiert
CONWS   EQU     $09                 * Code für die
                                    * Betriebssystemfunktion,
*                                   * die eine Zeichenkette
                                    * schreibt
CNECIN  EQU     $08                 * Code für Einlesen eines
                                    * Zeichens
*                                   * von der Tastatur ohne
                                    * Anzeige
TERM    EQU     $00                 * Beenden des Programms
*
* Hier beginnt das Textsegment
        TEXT                        * eigentlich überflüssig
*
* Aufruf der Betriebssystemfunktion conws(text)
start   move.l  #hallo,-(sp)        * 2. Parameter: Adresse
                                    * der Zeichenkette
        move    #CONWS,-(sp)        * 1. Parameter: Funktions-
                                    * nummer 9
```

```
        trap    #1              * Betriebssystemaufruf
        addq.l  #6,sp           * Stack aufräumen
* Fertig mit der Ausgabe!
*
* Jetzt nur noch auf eine Taste warten: cnecin()
        move    #CNECIN,-(sp)   * einziger Parameter: Funkti-
                                * onsnummer 8
        trap    #1              * wieder ab zum Betriebssystem
        addq.l  #2,sp           * immer schön ordentlich sein
*
* ein braves Programm sagt immer Bescheid, wenn es fertig ist
        move    #TERM,-(sp)     * Code für "Programm beenden"
        trap    #1              * und wieder ins Betriebssystem
*                               * hier kommen wir nicht mehr
                                * hin
* Hier beginnt das Datensegment
        DATA
hallo   DC.B    "Hallo, hier bin ich!",0
* BSS-Segment gibt es in diesem Miniprogramm nicht
*
        END                     * nur noch das unumgängliche
                                * Ende
```

Tatsächlich kommt auch das kleinste Programm kaum ohne Betriebssystemaufrufe aus. Sobald es an die Eingabe oder Ausgabe von Ergebnissen geht, ist ein Betriebssystemaufruf fast unumgänglich. Deshalb wollen wir hier einiges darüber vorwegnehmen (ausführlich wird das Ganze in Kapitel 4 behandelt).

Am häufigsten werden die GEMDOS-Aufrufe verwendet. Wie schon besprochen, funktioniert die Parameterübergabe so, daß alle Parameter auf den Stack geschoben werden, die letzten zuerst. Der erste Parameter (also der, der zuletzt auf dem Stack abgelegt wird) ist dabei die Funktionsnummer, die von 0 bis 87 reichen kann. Der Übersichtlichkeit halber geht man am besten so vor wie in diesem Programmbeispiel: Man definiert sie mittels EQU-Konstanten, die für die Nummern bestimmter Betriebssystemfunktionen stehen. Die Namen der Funktionen sind von ATARI festgelegt und in Anhang F beschrieben. Nachdem nun die Funktionsnummer abgelegt ist, wird der Einsprung mit dem TRAP-Befehl vollzogen, wobei für GEMDOS-Aufrufe immer

```
TRAP #1
```

verwendet wird. Danach ist das aufrufende Programm selbst dafür verantwortlich, den Stackpointer wieder auf den ursprünglichen Wert zurückzusetzen.

GEMDOS bietet Funktionen für Ein- und Ausgabe auf verschiedene Geräte und Dateien, die Verwaltung von Disketteninhaltsverzeichnissen, Speicherverwaltung und Aufruf sowie Beendigung von Programmen.

Wir brauchen in unserem Programm zunächst einmal eine Funktion, um eine Zeichenkette auf den Bildschirm zu schreiben. Dafür ist die Funktion Nummer 9, CONWS, gedacht. CONWS steht für "CONsole Write String", also "schreibe eine Zeichenkette zur Console". Als einzigen Parameter außer der Funktionsnummer bekommt sie die Adresse der Zeichenkette, die geschrieben werden soll. Wie üblich muß die Zeichenkette mit einem Nullbyte abgeschlossen sein.

Als nächstes soll ein beliebiges Zeichen von der Tastatur eingelesen werden. Dazu dient CNECIN, "Console No ECho INput", also Lesen von der Console ohne Echo. Mit Echo ist hier die Anzeige des getippten Zeichens auf dem Bildschirm gemeint, aber genau das tut eben diese Funktion nicht. Diesmal gibt es überhaupt keine Parameter. Das gelesene Zeichen wird im unteren Byte von D0 zurückgegeben. Da uns das Zeichen aber nicht interessiert, wird mit dem Inhalt von D0 nichts weiter angestellt.

Schließlich muß das Programm dem Betriebssystem noch irgendwie mitteilen, daß es beendet ist. Dazu dient die Funktion TERM (TERMinate program) mit der Nummer 0. Bei deren Ausführung wird der vom Programm beanspruchte Platz wieder verfügbar gemacht, und die Kontrolle wird dem aufrufenden Programm übergeben, was normalerweise das Desktop sein dürfte.

Soviel zu unserem Miniprogramm. Um den Umgang mit dem Assembler zu lernen, soll dieses Programm lauffähig werden.

Zunächst muß der Quellcode, also obiger Text, in den Computer eingegeben werden. Bei den meisten Assemblern läuft das so, daß der Text zunächst einmal mit einem beliebigen Editor eingetippt wird. Wenn man fertig ist, wird das Ergebnis in Form einer Datei abgespeichert. Nun muß man den Editor verlassen, den Assembler aufrufen und ihm diese Textdatei überreichen. Dieser erzeugt, wenn alles gutgeht, eine Objektcodedatei. Diese muß im allgemeinen noch durch den Linker geschickt werden, damit ein lauffähiges Programm erzeugt werden kann. Auf den letzten drei Instanzen – Assemblieren, Linken, Ausführen – können jeweils Fehler auftreten. In diesem Fall muß noch einmal editiert werden, bis das Programm fehlerlos ist. Abbildung 2.13 stellt diesen Programmierzyklus grafisch dar.

Nachteil dieser Methode ist, daß für die Erzeugung eines lauffähigen Programms mindestens drei Programme ausgeführt werden müssen. Sie können so jeden beliebigen Editor verwenden, also auch die meisten Textverarbeitungsprogramme. Bedingung ist nur, daß der Text im normalen ASCII-Format abgespeichert wird, also einfach Zeichen für Zeichen, so wie er auf dem Bildschirm steht. Achten Sie darauf, denn manche Textverarbeitungsprogramme verwenden ein eigenes Format, um den Text abzuspeichern.

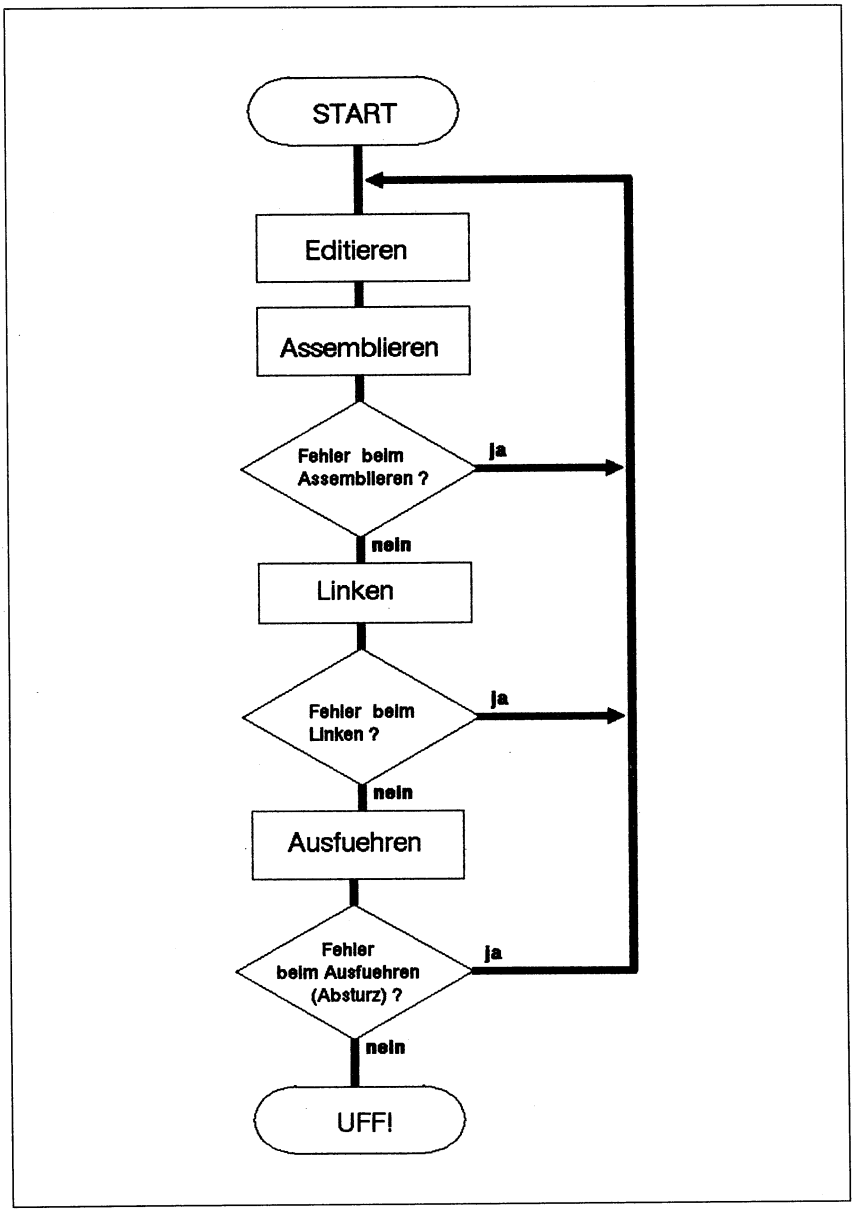


Abb. 2.13: Flußdiagramm des Programmierzyklus

Es geht natürlich auch anders. Es gibt integrierte Assemblerpakete, bei denen der Editor gleich Teil des Assemblers ist. Dort können Sie editieren, assemblieren, ausführen und sogar nach Fehlern suchen, ohne das Programm wechseln zu müssen. Dies ist sicher sehr vorteilhaft, solange Sie reine Assemblerprogramme schreiben müssen. Nur hat diese Art von Assemblern oft den Nachteil, daß sie über keinen Linker verfügen und somit keine Möglichkeit bieten, Assembler mit anderen Programmiersprachen zu kombinieren.

Da die Bedienung eines solchen integrierten Editor-Assembler-Debuggers recht einsichtig ist, wollen wir nur die Verwendung der recht verbreiteten oben beschriebenen Sorte weiter ausführen. Genauer gesagt, wollen wir die Verwendung am Assembler aus dem ATARI-Entwicklungspaket demonstrieren, da sich dieses Paket recht genau an den Motorola-Standard hält, der von den meisten anderen Assemblern kopiert wird. Wenn Sie mit der Bedienung des ATARI-Assemblers zurechtkommen, wird Ihnen auch ein anderer Assembler keine Probleme bereiten. So lassen sich alle Programme in diesem Buch ohne Änderungen mit dem ATARI-Assembler verwenden.

Der ATARI-Assembler ist eigentlich nur von einem Kommandozeileninterpreter aus vernünftig zu bedienen. Klicken Sie deshalb COMMAND.PRG an. Nach dem Laden erscheint eine Copyrightmeldung und der Buchstabe des aktuellen Laufwerks in geschweiften Klammern.

Der ATARI-Assembler hat eine Eigenheit: Wenn Sie ihn zum ersten Mal verwenden, sollten Sie ihn zuerst einmal initialisieren mit dem Kommando

```
as68 -I as68init
```

Daraufhin legt er die Datei "as68init" an, ohne die er sich weigert zu assemblieren.

Nun geht es also an den ersten Schritt: die Eingabe des Programmcodes. Nehmen Sie dazu am besten einen Editor wie etwa ED.TTP. Unser Quellcode soll unter dem Namen HALLO.S abgespeichert werden. Das "S" im Extender steht für "Source code", also Quellcode. Auf anderen Systemen und bei anderen Assemblern ist auch die Endung "ASM" für Assembler recht verbreitet. Der ATARI-Assembler scheint entgegen der Dokumentation etwas gegen jegliche andere Endung als "S" zu haben. Bei "ASM"-Dateien bringt er einige Fehlermeldungen und verabschiedet sich dann, aber nicht ohne vorher Ihre Quellcodedatei ruiniert zu haben. Also bleibt Ihnen nichts anderes übrig, als sich an die Endung .S zu gewöhnen.

Das entsprechende Kommando lautet also:

```
ed.ttp hallo.s
```

Daraufhin befinden Sie sich im Editor und können das Programm abtippen.

Nachdem Sie den Quellcode abgespeichert und den Editor verlassen haben, geht es nun an das Assemblieren. Der Assembler bekommt als Argument den Namen der Datei, die er assemblieren soll. Daneben kann man dem Assembler noch Optionen überreichen, die jeweils aus einem Buchstaben bestehen und mit einem Minus-Zeichen (–) eingeleitet werden. Die Option "–P" (Print) etwa veranlaßt den Assembler, ein Assemblerlisting zu produzieren (sonst tut er es nicht). Dort wird der Quellcode noch einmal aufgelistet, doch zusätzlich werden Informationen zum erzeugten Code und zum Wert von Symbolen ausgegeben. Wenn Sie ein Assemblerlisting auf den Bildschirm ausgeben lassen wollen, rufen Sie den Assembler so auf:

```
as68 -P hallo.s
```

Beachten Sie, daß beim Aufruf von as68 der Extender ".prg" nicht angegeben werden muß. Der Kommandointerpreter erkennt ".prg"-Files und fügt bei ihnen automatisch den richtigen Extender hinzu. Leider kennt er keine ".tos"- und ".ttp"-Programme.

Meistens will man aber das Assemblerlisting nicht über den Bildschirm laufen sehen, sondern zur späteren Weiterverarbeitung in eine Datei schreiben oder auf dem Drucker ausgeben. Dazu bietet der Assembler die Möglichkeit der Ausgabeumleitung. Geben Sie einfach einen weiteren Parameter an, bestehend aus einem Größer-als-Zeichen (>) und – ohne Leerzeichen dazwischen – dem Dateinamen oder dem Gerätenamen, worauf die Ausgabe erfolgen soll. Sie schreiben also

```
as68 -P hallo.s >PRN:
```

um das Listing auf den Drucker auszugeben, und

```
as68 -P hallo.s >hallo.lst
```

um das Listing in die Datei "hallo.lst" zu schreiben.

Die wichtigsten Optionen sind:

U– veranlaßt den Assembler dazu, alle nicht in diesem Modul definierten Symbole als extern zu betrachten, das heißt, sie werden so behandelt, als würden sie mit XREF importiert. So sparen Sie die Verwendung von XREF-Direktiven. Nachteil ist, daß man im Falle eines Tippfehlers in einem Symbolnamen erst beim Linken merkt, daß das Symbol nirgendwo definiert ist.

L– zwingt den Assembler dazu, für alle Adressen Langworte zu verwenden. Wenn diese Option nicht angegeben wird, nimmt der Assembler an manchen Stellen nur Worte – mit dem Resultat, daß das Programm nur noch in den ersten 32K des Speichers laufen kann. Da diese schon vom Betriebssystem belegt werden, wird es leider überhaupt nicht laufen. Deshalb: unbedingt immer Option –L angeben!

Der Grund ist, daß der ATARI-Assembler eigentlich nicht für TOS, sondern für das Betriebssystem CP/M 68k entwickelt wurde. Ursprünglich hatte ATARI ja vor, den ST mit diesem CP/M auszurüsten, man ist jedoch dazu übergegangen, ein eigenes Betriebssystem zu entwickeln, das der ST-Hardware besser gerecht wird: eben TOS. Wie wir noch sehen werden, ist dies nicht die einzige Stelle, an der Reste von CP/M 68k auftauchen.

Der wahre Befehl, um unser Programm zu assemblieren, lautet also

```
as68 -L hallo.s
```

Daraufhin nimmt sich der Assembler unser Programm vor und produziert die Datei "hallo.o", wobei ".o" für "object code" steht.

Der Linker bekommt in unserem Fall nur den Namen der produzierten ".o"-Datei als Argument. Obwohl es hier nur ein einziges Modul gibt, muß es trotzdem durch den Linker geschickt werden. Das Kommando lautet also:

```
link68 hallo.o
```

Daraus erzeugt der Linker eine Datei namens hallo.68k. Aber das ist doch kein üblicher TOS-Extender für ein lauffähiges Programm? Richtig, hier kommt wieder einmal CP/M 68k ins Spiel. Die Datei hallo.68k wäre bestenfalls unter diesem Betriebssystem lauffähig (wenn überhaupt).

Damit das Programm aber trotzdem zum Laufen gebracht werden kann, liefert ATARI das Programm RELMOD.PRG mit. Dieses ist dafür zuständig, ".68k"-Dateien in ausführbare TOS-Programme umzuwandeln. Der Aufruf lautet:

```
relmod hallo.68k hallo.tos
```

Das besagt, daß aus der Datei "hallo.68k" die Datei "hallo.tos" erzeugt werden soll. Natürlich kann man statt letzterem auch "hallo.prg" angeben, wenn es sich um eine GEM-Anwendung handelt.

Nun haben wir es endlich geschafft, ein ausführbares Programm zu erzeugen.

Rufen Sie das Programm nun auf mit

```
hallo.tos
```

und die freundliche Meldung erscheint tatsächlich auf dem Bildschirm.

Wenn Sie nun sagen, das sei eine ganze Menge Aufwand, nur um "Hallo, hier bin ich!" auf den Bildschirm zu schreiben, wird es Sie wahrscheinlich auch nicht trösten, daß das ausführbare Programm kürzer als 100 Bytes ist, was sicher in keiner anderen Programmiersprache möglich wäre. Aber zumindest ein Teil dieses Aufwands kann mit Hilfe einer Batchdatei wegfallen.

Rufen wir uns noch einmal ins Gedächtnis, welche Schritte zum Assemblieren und Linken notwendig waren:

```
as68 -L hallo.s
link68 hallo.o
relmod hallo.68k hallo.tos
```

Außerdem kann es von Vorteil sein, die zwischendurch angelegten Dateien mit dem Programm RM.PRG gleich wieder zu löschen:

```
rm hallo.o
rm hallo.68k
```

Diese Prozedur kann nun mit Hilfe einer Batchdatei automatisiert werden. Dabei geht es um folgendes:

Zum Entwicklungspaket gehört ein Programm namens BATCH.TTP. Ruft man es auf und übergibt ihm dabei als Argument einen Dateinamen, so liest es den Inhalt der Datei Zeile für Zeile aus und führt die darin enthaltenen Kommandos nacheinander aus, als wären sie im Kommandointerpreter eingetippt worden. Praktisch ist dabei, daß beim Aufruf von BATCH.TTP noch weitere Parameter angegeben werden können, die an die Batch-Datei weitergereicht werden. Dort werden diese Parameter mit %1, %2 und so weiter angesprochen. Dabei werden die Parameter an der angegebenen Stelle als Text eingesetzt. Eine Batch-Datei zum Assemblieren und Linken müßte also so aussehen:

```
as68 -L %1.s
link68 %1.o
rm %1.o
relmod %1.68k %1.tos
rm %1.68k
```

Wenn Sie diese Zeilen mit einem Editor eingeben und sie in der Datei "asm.bat" ablegen, können Sie vom Kommandointerpreter aus eingeben:

```
batch.ttp asm hallo
```

und es werden genau die gewünschten Schritte durchgeführt. Sie können es sich noch einfacher machen, indem Sie die Batch-Datei "a.bat" nennen und das Programm "batch.ttp" in "b.prg" umnennen. Dann brauchen Sie nur noch einzugeben:

```
b a hallo
```

Nachteil der Batchverarbeitung ist, daß in dem Fall, daß der Assembler Syntaxfehler findet, alle nachfolgenden Programme trotzdem aufgerufen werden, die Dateien, auf die sie angesetzt sind, nicht finden und mit einer Fehlermeldung aussteigen. Leider gibt es keine einfache und zugleich befriedigende Lösung für dieses Problem; die einzige Möglichkeit ist, daß Sie einfach eine Taste drücken und damit den Ablauf der Batch-Datei unterbrechen.

Sehen wir uns jetzt das Assembler-Listing, das mit der "P"-Option erzeugt werden kann, einmal genauer an. Der Befehl, um ein Listing von unserem Programm in die Datei "hallo.lst" zu schreiben, lautet:

```
as68 -L -P hallo.s >hallo.lst
```

Hier auszugsweise ein paar Zeilen aus dem Assemblerlisting (die Bemerkungen, die auch aufgelistet werden, wurden der Übersichtlichkeit halber weggelassen):

```
.
.
.
17 00000000 2F3C00000000        start move.l #hallo,-(SP)
18 00000006 3F3C0009                 move   #CONWS,-(SP)
19 0000000A 4E41                     trap   #1
20 0000000C 5C8F                     addq.l #6,SP
.
.
.
33 00000000                          DATA
34 00000000 48616C6C6F2C2068    hallo DC.B   "Hallo, hier bin ich!",0
34 00000008 6965722062696E20
34 00000010 6963682100
.
.
.
```

Nun zur Bedeutung der einzelnen Felder:

Die erste Zahl in jeder Zeile ist die Zeilennummer. Die Zeilen werden vom Assembler einfach laufend durchnumeriert, damit er sich bei der Ausgabe von Fehlermeldungen darauf beziehen kann. Die nächste Zahl, eine achtstellige Hexadezimalzahl, zeigt den "location counter", also den Adreßzähler des Assemblers, bevor der Befehl auf dieser Zeile assembliert wird. So steht beim ersten Befehl an dieser Stelle eine Null. Hier zeigt sich auch, daß für jedes Segment ein eigener Adreßzähler existiert: Nach der Direktive DATA beginnt der Zähler wieder bei null. Genau diese Zahl ist es, die einem Label in dieser Zeile als Wert zugewiesen wird.

Die folgenden hexadezimalen Zahlen zeigen den erzeugten Code. Sehen wir uns einmal die Übersetzung des folgenden Befehls genauer an:

```
MOVE.L #hallo,-(SP)
```

Zunächst steht da das Wort $3F3C, was für den Prozessor soviel bedeutet wie "Nimm das folgende Langwort und lege es auf dem Stack ab". Danach folgt der Langwort-Operand, in diesem Fall der Wert des Labels "hallo". Dieser ist einfach null, da dieses Label die erste Speicherstelle des Datensegments bezeichnet. Intern merkt sich der Assembler, daß der Wert des Operanden nicht endgültig ist und vor dem Start reloziert werden muß, indem die physikalische Anfangsadresse des Datensegments addiert wird.

Bei der Definition des freundlichen Textes gibt der Assembler an dieser Stelle den entsprechenden ASCII-Code an, und zwar auf drei Zeilen verteilt, da nicht alles auf eine Zeile paßt.

Danach folgt nur noch der Original-Quellcode. Wenn allerdings in einer Zeile ein Fehler auftritt, so wird danach die Fehlermeldung eingefügt.

Nach der Auflistung des Programms hängt der Assembler noch folgenden Abschnitt an:

```
S y m b o l   T a b l e

CONIN   00000008  ABS  CONWS  00000009  ABS  TERM 00000000 ABS
hallo   00000000  DATA start  00000000  TEXT
```

Hier gibt er die Werte sämtlicher Symbole aus. Hinter dem Wert gibt er die Art des Symbols an, und zwar "ABS" für absolute Symbole, und für relative die Bezeichnung des Segments, in denen sie definiert sind.

Bleibt nur noch die Frage zu klären, was Ihnen das Assemblerlisting nützt. Nun, einerseits dient es dazu, vom Assembler entdeckte Fehler zu lokalisieren. Wenn hingegen alle Syntaxfehler beseitigt sind, kann das Assemblerlisting zur Suche nach logischen Fehlern mit Hilfe eines Debuggers nützlich sein. (Was ein Debugger ist und wie er bedient wird, wird im nächsten Abschnitt noch erklärt.)

Trotz aller Tricks ist die Arbeit mit dem ATARI-Assembler über Kommandoaufrufe recht unkomfortabel. Wenn Sie also über eine flexible Shell wie etwa MENU+ von Metacomco verfügen, können Sie diese an den ATARI-Assembler anpassen. Auf der beiliegenden Diskette befindet sich die Datei MENU.INF, die MENU+ an den Assembler aus dem Entwicklungspaket anpaßt. Sorgen Sie nur dafür, daß sich diese Datei beim Starten von MENU+ im gleichen Directory (Inhaltsverzeichnis) befindet; den Rest macht MENU+.

Im Grunde tut so eine Shell auch nichts anderes, als Kommandos in obiger Art abzusetzen, nur sind diese in der Shell versteckt und werden von der Shell gewöhnlich in der Form verwaltet, daß etwa ein Anklicken der Option "Assemblieren" ein Absetzen des Assembler-Kommandos in der oben beschriebenen Form bewirkt.

Benutzung einer RAM-Disk

Wenn Sie das Testprogramm auf der Diskette assembliert haben, hatten Sie genug Zeit für eine Kaffeepause. Falls Ihr Computer jedoch über genügend Speicherplatz verfügt, ist die Benutzung einer RAM-Disk sehr empfehlenswert (sofern Sie nicht gerade mit einer Festplatte ausgestattet sind). Um völlig in der RAM-Disk arbeiten zu können, müssen Sie zuerst folgende Dateien hineinkopieren:

AS68.PRG	der Assembler
AS68INIT	mit der I-Option erzeugte Datei
AS68SYMB.DAT	gehört zum Assembler
LINK68.PRG	der Linker
RELMOD.PRG	Programm zum Wandeln von .68k in .TOS
RM.PRG	Programm zum Löschen von Dateien
COMMAND.TOS	der Kommandointerpreter
BATCH.TTP (oder B.PRG)	der Batch-Abarbeiter
ASM.BAT (oder A.BAT)	unsere Batchdatei

Dazu kommt noch

- ein Editor
- Routinenbibliotheken und Linkdateien nach Bedarf; für den Umgang mit den Programmen aus diesem Buch werden sie nicht gebraucht.

Wenn Sie als Editor ED.TTP nehmen, kommt das Ganze zusammen auf etwa 165 Kilobytes. So sollte eine 200 Kbyte große RAM-Disk ausreichen, sofern Ihre Programme nicht zu groß werden. Dieser Umstand ist deshalb besonders interessant, da eine 200 Kbyte große RAM-Disk auch auf einem ST mit 512 Kbyte und ROMs zu installieren ist, ohne daß der Assembler beeinträchtigt wird.

Was die Kommandos betrifft, wird eine RAM-Disk gewöhnlich genauso behandelt wie eine Diskette. Nur eines gibt es zu beachten: Da Assemblerprogramme in der Entwicklungsphase gerne einmal abstürzen, sollten Sie vor jedem Testlauf den geänderten Quellcode auf einer Diskette sichern, denn beim Programmabsturz ist gewöhnlich der Inhalt der RAM-Disk verloren. Vorteilhaft ist eine resetfeste RAM-Disk, denn diese kann ihre Dateien meistens über einen Programmabsturz hinweg retten. Leider funktioniert das auch nicht immer, denn auch die beste RAM-Disk kann nicht verhindern, daß ein Programm quer über den ganzen Speicher schreibt und so auch den Inhalt der RAM-Disk zerstört.

Trotz allem ist der Gebrauch einer RAM-Disk sehr empfehlenswert, da er Assemblier- und Linkzeiten von Minuten auf Sekunden reduziert.

Makros

Nicht jeder Assembler hat sie, aber jeder Programmierer wünscht sie sich. Eines gleich vorweg: Der Assembler aus dem ATARI-Entwicklungspaket verarbeitet keine Makros. Außerdem ist es etwas problematisch, daß Makros oft mit etwas unterschiedlicher Syntax behandelt werden. Die Beispiele in diesem Kapitel sind in der Syntax des Metacomco-Assemblers abgefaßt, der sich in diesem Punkt voll an den Motorola-Standard hält.

Was sind Makros?

Wenn man große Assemblerprogramme schreibt, wird man bald feststellen, daß bestimmte Codesequenzen sich ständig wiederholen. Nehmen wir zum Beispiel ein Programm, das viel mit Textausgabe zu tun hat. Dort wird man immer wieder den Systemaufruf zur Ausgabe einer Zeichenkette finden:

```
MOVE.L   string,-(SP)      * Adresse der Zeichenkette
MOVE     #9,-(SP)          * Code 9: CONWS
TRAP     #1                * GEMDOS-Aufruf
ADDQ.L   #6,SP             * Stackpointer korrigieren
```

Um die ständige Wiederholung zu vermeiden, könnte man natürlich ein Unterprogramm schreiben, das dann so aussieht:

```
* Unterprogramm zur Ausgabe eines Strings
* Adresse des Strings wird in Register D0.L überreicht
print     MOVE.L    D0,-(SP)
          MOVE      #9,-(SP)
          TRAP      #1
          ADDQ.L    #6,SP
          RTS
```

Nachteil dieser Methode ist jedoch, daß immer noch zwei Befehle für die Ausgabe eines Strings gebraucht werden, denn zuerst muß die Adresse des Strings ins Register D0 geladen werden, und dann erfolgt der Aufruf des Unterprogramms. Außerdem leidet die Geschwindigkeit des Programms unter den JSR- und RTS-Befehlen. Zugegeben, bei diesem Beispiel macht es kaum etwas aus, da die Betriebssystemroutine zum Schreiben einer Zeichenkette ohnehin einige Zeit braucht. Doch stellen Sie sich vor, wir wollten statt dessen eine Routine zum Plotten eines Punktes aufrufen. Dann könnte das Programm merklich beschleunigt werden, wenn der Code zum Plotten des Punktes direkt da steht, wo er gebraucht wird, und nicht mittels JSR aufgerufen werden muß. Da es ziemlich umständlich ist, zu diesem Zweck Zeilen blockweise mit einem Editor zu kopieren, wurde das Makro erfunden. Ein Makro ist eine an einer Stelle im Programm definierte Codesequenz, die bei einem Aufruf an einer anderen Stelle im Quellcode vom Assembler als Text eingesetzt wird.

Ein Beispiel macht dies klarer. Definieren wir uns zuerst ein Makro, das genau das gleiche tut wie obiges Unterprogramm:

```
PRINT     MACRO
          MOVE.L    D0,-(SP)
          MOVE.L    #9,-(SP)
          TRAP      #1
          ADDQ.L    #6,SP
          ENDM
```

Hier begegnen uns gleich zwei neue Direktiven: MACRO und ENDM.

MACRO sagt dem Assembler, daß hier die Definition eines Makros beginnt. Diese Direktive muß mit einem Label (nach den üblichen Konventionen) versehen sein, das den Namen des Makros angibt.

ENDM END Macro
besagt, daß das Makro hier zu Ende ist. Alle Befehle zwischen MACRO und ENDM werden vom Assembler als "Wert" des Makros abgespeichert.

Den Namen des Makros "PRINT" hätte man natürlich auch klein schreiben können. Es ist jedoch eine Art Konvention, Makronamen in Großbuchstaben zu schreiben, und es hilft Ihnen auch, Makronamen von Labeln zu unterscheiden. Bedenken Sie, daß der Assembler bei Symbolnamen zwischen Groß- und Kleinbuchstaben unterscheidet.

Der Aufruf eines Makros ist denkbar einfach: Man schreibt einfach in das Operandenfeld seinen Namen. Das sieht dann zum Beispiel so aus:

```
MOVE.L  string,D0
PRINT
.
.
.
```

Was den Perfektionisten hier immer noch stört, ist der MOVE-Befehl. Aber auch dieses Problem kann umgangen werden, denn Makros erlauben es auch, daß man ihnen Parameter übergibt. Im Makro werden diese Parameter durch den Backslash (\), gefolgt von einer Zahl, angesprochen (nicht zu verwechseln mit dem Prozentzeichen (%), das der Benutzung von Parametern bei Batchdateien diente). Dabei wird mit \1 der erste Parameter angesprochen, mit \2 der zweite und so weiter. Die Parameter werden als Text eingesetzt, das heißt, wenn in einer Zeile des Makros etwa steht:

```
MOVE.L \1,-(SP)
```

und der erste Parameter "string" war, dann entfernt der Assembler die \1 aus der Zeile und setzt statt dessen den Wert des Parameters, also *string* ein. Erst dann wird der Code für die entstandene Anweisung generiert. Der Assembler tut also so, als ob man geschrieben hätte:

```
MOVE.L string,-(SP)
```

So ist es nicht schwer, unser Makro entsprechend umzuschreiben:

```
PRINT   MACRO
        MOVE.L    /1,-(SP)
        MOVE.L    #9,-(SP)
        TRAP      #1
        ADDQ.L    #6,SP
        ENDM
```

Beim Aufruf braucht man also nur noch folgendes anzugeben:

```
PRINT string
.
.
.
```

Wenn das Makro mehrere Parameter hätte, so würden diese beim Aufruf durch Kommas getrennt.

Der Assembler behandelt also diesen Makroaufruf so, als hätte man an dieser Stelle tatsächlich geschrieben:

```
MOVE.L   string,-(SP)
MOVE.L   #9,-(SP)
TRAP     #1
ADDQ.L   #6,SP
```

Die Ersetzung als Text ist recht flexibel. Bisher haben wir es immer so aufgefaßt, daß an der Stelle, die durch das Label "string" gekennzeichnet ist, ein Zeiger auf die auszugebende Zeichenkette steht. Man kann aber das gleiche Makro auch dann verwenden, wenn der String selbst durch einen Label gekennzeichnet ist, wie es im Programmbeispiel im letzten Abschnitt der Fall war. Wir müssen also dafür die direkte Adressierungsart verwenden:

```
PRINT #hallo
```

Dieser Aufruf generiert aus der ersten Zeile des Makros den Befehl

```
MOVE.L #hallo,-(SP)
```

Natürlich wäre auch der folgende Aufruf denkbar, falls die Adresse des Strings schon im Register A0 steht:

```
PRINT A0
```

Auch hier erzeugt der Assembler genau das, was man erwartet, nämlich

```
MOVE.L A0,-(SP)
```

und den Rest wie gehabt.

Falls Sie allerdings Makros benutzen, die Sie nicht selbst geschrieben haben, sollten Sie mit der Benutzung von Registern vorsichtig sein. Es könnte ja sein, daß im Makro gerade jenes Register, das Sie angeben, verändert wird. Das kann zwar zu einem interessanten Ergebnis führen, aber garantiert nicht zu dem, was Sie erreichen wollten.

Ein Problem stellt es dar, innerhalb eines Makros Label verwenden zu wollen. Denn was geschieht, wenn ein Makro innerhalb eines Progamms mehrmals aufgerufen wird? Das Label wird mehrfach im Programm definiert, worauf der Assembler mit einer Fehlermeldung reagiert. Um dieses Problem zu um-

gehen, stellt der Assembler ein spezielles Symbol zur Verfügung, das mit "\@" (Backslash Klammeraffe) bezeichnet wird und für die Nummer des Aufrufs des Makros steht, in dem es verwendet wird. Genauer: "\@" wird durch die Zeichenkette ".nnn" ersetzt, wobei nnn die Anzahl der Aufrufe des Makros ist. Genau wie die Parameter wird dieses Symbol als Text ersetzt, also beim ersten Aufruf des Makros durch ".001", beim zweiten durch ".002" und so weiter. Um die Verwendung zu zeigen, hier ein Makro, das eine Warteschleife realisiert:

```
WAIT        MACRO
            MOVE      \1,D0
wait1\@     MOVE      #$FFFF,D1
wait2\@     DBRA      D1,wait2\@
            DBRA      D0,wait1\@
            ENDM
```

Dieses Makro besteht aus zwei geschachtelten Schleifen. Nehmen wir an, es handelt sich um den ersten Aufruf dieses Makros (vom Anfang des Quellcodes an), und es wird angegeben:

```
WAIT 50
```

dann wird der Assembler daraus folgendes erzeugen:

```
            MOVE      50,D0
wait1.001   MOVE      #$FFFF,D1
wait2.001   DBRA      D1,wait2.001
            DBRA      D0,wait1.001
```

Manchmal wird man vor der Entscheidung stehen, ob für eine bestimmte Codesequenz ein Makro oder ein Unterprogramm besser geeignet ist. Die Vorteile von Makros sind:

- Das Makro ist effizienter, da kein JSR und RTS ausgeführt wird.
- Makros sind einfacher aufzurufen.
- Makros können Parameter in verschiedenen Adressierungsarten bekommen.

Subroutinen haben dagegen den Vorteil, daß der Code nur einmal Speicherplatz beansprucht, während beim Makro ja bei jedem Aufruf der Platz für den Code beansprucht wird. So werden Makros im allgemeinen nur für relativ kurze Codesequenzen verwendet, bis etwa zu einigen Dutzend Befehlen. Ohnehin fällt bei längeren Codesequenzen auch der Vorteil der größeren Geschwindigkeit des Makros weg, da die Ausführungszeit eines BSR und RTS gegenüber dem restlichen Code vernachlässigbar wird.

Dieser Abschnitt erhebt keinen Anspruch auf Vollständigkeit; die meisten Assembler bieten noch die eine oder andere Direktive, die im Zusammenhang mit Makros nützlich sein kann. Da dies jedoch von Assembler zu Assembler recht unterschiedlich gehandhabt wird, können wir Sie nur auf Ihr Benutzerhandbuch verweisen.

Bei manchen Assemblern werden recht umfangreiche Makrobibliotheken mitgeliefert, die etwa die Betriebssystemfunktionen leichter zugänglich machen. Falls Sie über eine solche Makrobibliothek verfügen, befassen Sie sich ruhig einmal damit, da so etwas recht nützlich sein kann.

Zum Abschluß dieses Abschnitts hier noch einmal das "Hallo Welt"-Programm aus dem letzten Abschnitt, diesmal mit Makros implementiert:

```
* Programm "Hallo Welt"
* 2. Version mit Makros
* Achtung!!! Wird nur von Makro-Assemblern verarbeitet, also
* insbesondere nicht vom ATARI-Assembler
*
* Gibt nur auf dem Bildschirm aus "Hallo, hier bin ich!"
* und wartet dann auf einen Tastendruck
*
TERM    EQU     $00         * Beenden des Programms
*
* Hier werden die Makros definiert
CONWS   MACRO               * schreibt einen String zur Konsole
        move.l  \1,-(sp)    * 2. Parameter: Adresse der Zeichen-
                            * kette
        move    #9,-(sp)    * 1. Parameter: Funktionsnummer 9
        trap    #1          * Betriebssystemaufruf
        addq.l  #6,sp       * Stack aufräumen
        ENDM

CNECIN  MACRO               * liest ein Zeichen von der
                            * Tastatur
        move    #8,-(sp)    * einziger Parameter:
                            * Funktionsnummer 8
        trap    #1          * Ab zum GEMDOS
        addq.l  #2,sp       * wie üblich
        ENDM
*
* Hier beginnt das Textsegment
        TEXT                * eigentlich überflüssig
*
* Aufruf der Betriebssystemfunktion conws(text)
start   CONWS  #hallo
* Fertig mit der Ausgabe!
*
* Jetzt nur noch auf eine Taste warten: cnecin()
        CNECIN
```

```
*
* ein braves Programm sagt immer Bescheid, wenn es fertig ist
        move    #TERM,-(sp)     * Code für "Programm Beenden"
        trap    #1              * und wieder ins Betriebssystem
*                               * hier kommen wir nicht mehr hin
* Hier beginnt das Datensegment
        DATA
hallo   DC.B    "Hallo, hier bin ich!",0
* BSS-Segment gibt es in diesem Miniprogramm nicht
*
        END                     * nur noch das unumgängliche Ende
```

Die Benutzung eines Debuggers

Die hauptsächliche Tätigkeit des Assemblerprogrammierers stellen wohl die oftmals schwer zu findenden logischen Programmfehler, die sogenannten "bugs" (engl. Wanzen, Käfer) dar. Um diese Fehler zu suchen, nimmt man einen Debugger ("Entwanzer").

Was bietet nun so ein Debugger für Möglichkeiten der Fehlersuche? Zunächst kann man ein fertiges Programm laden und sich Informationen über die Lage seiner Segmente im Speicher ausgeben lassen. Dann gibt es immer eine Funktion, um den Inhalt von Speicherplätzen zu betrachten und auch zu verändern. Darüber hinaus hat man die Möglichkeit, sich Teile des Programms disassemblieren zu lassen, also von Maschinensprache in die lesbare Mnemonik-Form verwandeln zu lassen. Sehr praktisch zur Fehlersuche ist die Möglichkeit des "Tracens" von Programmen. Das bedeutet, daß der Debugger die Ausführung des Programms Schritt für Schritt simuliert und dabei jeden ausgeführten Befehl und den Status des Prozessors auflistet.

Einige Debugger bieten die Möglichkeit, auf die Symbole zuzugreifen, die man im Quellcode des Programms definiert hat. Der ATARI-Assembler beispielsweise fügt in die ".o"-Datei eine komplette Liste aller verwendeten Symbole ein. Nur wird diese Liste vom Linker unterschlagen – sofern man beim Linken nicht eine bestimmte Option abgibt. Diese Option lautet bei LINK68 "[s,l]". Dabei steht "s" für "symbol table", womit gemeint ist, daß eine Symboltabelle in der ".68k"-Datei erzeugt werden soll. "l" erweitert diese Option dahingehend, daß nicht nur mit XDEF als global deklarierte, sondern auch modullokale Symbole in der Liste erscheinen sollen. (In unserem Beispielprogramm sind alle Symbole modullokal.)

Das Batchfile zum Produzieren eines Programms mit Symboltabelle sieht also so aus:

```
as68 -l %1.s
link68 [s,l] %1.o
rm %1.o
relmod %1.68k %1.TOS
rm %1.68k
```

Nennen Sie diese Datei zum Beispiel "ad.bat", "a" für Assembler und "d" für Debugger.

Die Symboltabelle wird einfach an die anderen Segmente des Programms angehängt. Wenn das Programm allerdings normal ausgeführt wird, so verbraucht die Symboltabelle keinen Speicherplatz, da sie gar nicht erst geladen wird. Sie ist eben nur für einen Debugger interessant.

Nehmen wir uns beispielhaft den Debugger SID68.TOS aus dem ATARI-Entwicklungssystem vor. Natürlich kann man diesen Debugger nicht nur mit vom ATARI-Assembler erzeugten Programmen verwenden, sondern mit jedem ausführbaren Programm. Die Symboltabelle ist zur Benutzung dieses Debuggers nicht unbedingt erforderlich, aber recht nützlich.

Bei diesem Debugger bestehen die Kommandos nur aus einem Buchstaben, gefolgt von den Parametern des Kommandos. Speicheradressen werden entweder hexadezimal angegeben, oder man verwendet ein Label, vor das man einen Punkt setzt.

Nach dem Aufruf des Debuggers wollen wir zuerst das Programm laden. Das geschieht mit dem Kommando

```
E hallo.tos
```

"E" steht für "load for Execute". Als Antwort gibt der Debugger Informationen über Lage und Länge der drei Segmente aus.

Um das Programm disassemblieren zu lassen, kann das "L"-Kommando (List) verwendet werden. Als Argument kann eine Speicheradresse angegeben werden, oder eben ein Label. Mit dem Befehl

```
L .start
```

werden 12 Befehle unseres Programms disassembliert – ab Label "start", das ja den Anfang unseres Programms markiert.

Wenn man sich die hexadezimale Darstellung eines Speicherbereichs ansehen will, kann dies mit dem "D"-Kommando (Display) geschehen. Gibt man etwa ein:

```
D .start
```

so kann man das Programm in Form von Hexadezimalzahlen sehen.

Interessante Möglichkeiten bietet das "T"-Kommando (Trace). Als Argument erhält es eine Speicheradresse. Bei der Ausführung dieses Kommandos simuliert der Debugger die Maschinenspracheoperationen und gibt nach der Ausführung jedes Befehls dessen Adresse, den disassemblierten Befehl selbst und den Zustand sämtlicher CPU-Register aus. Mit einem Tastendruck kann das Tracing jederzeit unterbrochen werden, damit man sich einen genaueren Überblick über den Zustand des Programms verschaffen kann. Probieren Sie einmal folgenden Befehl aus:

```
T .start
```

Doch Vorsicht, die Befehle werden tatsächlich ausgeführt! Unsere Meldung wird tatsächlich ausgegeben (was leider in den Ausgaben des Trace-Programms etwas untergeht), und der Aufruf der GEMDOS-Funktion TERM führt nicht nur zur Beendigung unseres Programms, sondern auch zu der des Debuggers. Die GEMDOS-Traps werden übrigens der Übersichtlichkeit halber nicht in das Tracing einbezogen; es wird nur der TRAP-Befehl gelistet.

Wenn man ein Listing des Quellcodes neben sich liegen hat, ist es so recht gut möglich, den Programmlauf zu verfolgen.

Wie bei den meisten Bildschirmausgaben, kann auch hier die Ausgabe mit der Tastenkombination <Ctrl>-<S> gestoppt werden. Fortsetzen kann man sie dann mit <Ctrl>-<Q>.

Bei großen Programmen, die schon mit der Initialisierung einige Zeit verbringen, kann es recht müßig sein, den Programmlauf bis zu dem Punkt tracen zu lassen, an dem man den Fehler vermutet. Dafür bietet SID68 sogenannte Breakpoints und Passpoints.

Ein *Breakpoint* ist ein spezieller Maschinenbefehl, der vom Debugger an eine bestimmte Stelle des Programms geschrieben wird. Sobald der Befehl an dieser Stelle ausgeführt wird, geht die Kontrolle wieder an den Debugger über, der sofort den ursprünglichen Befehl an dieser Stelle wiederherstellt. Breakpoints erlauben es, sich gezielt den Zustand des Programms an einer ganz bestimmten Stelle anzusehen und eventuell von dort aus zu tracen.

Breakpoints können mit dem "G"-Kommando (Goto) verwendet werden. Wird als Argument nur eine Adresse bzw. ein Label angegeben, so wird das

Programm an dieser Stelle ausgeführt. Mit folgendem Befehl kann also das Programm gestartet werden:

```
G .start
```

Auch hier wird der Debugger mit dem Aufruf von TERM verlassen.

Natürlich ist es nicht schön, wenn jedesmal bei der Ausführung des Programms der Debugger verlassen wird. Deshalb wird an die Stelle, an der der Aufruf der Funktion TERM stattfindet, ein Breakpoint gesetzt. Breakpoints werden einfach als weitere Parameter beim Aufruf des "G"-Kommandos angegeben, also:

```
G .start,.fertig
```

Dies heißt, daß die Programmausführung am Label "start" beginnen soll, während beim Label "fertig" ein Breakpoint gesetzt wird. Und tatsächlich, nachdem die Meldung auf dem Bildschirm erscheint und eine Taste gedrückt worden ist, meldet sich der Debugger wieder zu Wort, indem er den Zustand der CPU ausgibt und dann auf die nächste Eingabe wartet.

Die so definierten Breakpoints werden beim Abbruch des Programms sofort wieder entfernt. Der Assembler bietet aber noch eine weitere Möglichkeit des gezielten Programmstops: *Passpoints*. Sie werden ähnlich verwaltet wie Breakpoints, nur gehört zu jedem Passpoint ein Zähler, der angibt, wie oft der Passpoint durchlaufen werden soll, bevor das Programm stoppt. Solange der Passpoint nicht entsprechend oft durchlaufen worden ist, bemerkt man nichts von der kurzen Programmunterbrechung. Erst wenn der Zähler null erreicht, wird wie üblich der CPU-Status ausgegeben und auf eine neue Eingabe gewartet. Diese Funktion ist besonders für die Fehlersuche in Schleifen gedacht, bei denen man die Behandlung von bestimmten Werten untersuchen will. Gesetzt wird ein Passpoint mit folgendem Kommando:

```
P <Adresse>
```

oder

```
P <Adresse>,<Zähler>
```

Im ersten Fall wird der Zähler des zu setzenden Passpoints auf 1 gesetzt, das heißt, er verhält sich genauso wie ein Breakpoint. Im zweiten Fall wird auch noch der Zähler zum Passpoint angegeben.

Löschen kann man Passpoints mit dem Kommando:

```
-P <Adresse>
```

Passpoints sind statisch, das heißt, sie überdauern auch Start und Abbruch eines Programms. Wenn Sie also erreichen wollen, daß das Programm immer abgebrochen wird, bevor es TERM aufruft, müßten Sie eingeben:

```
P.fertig
```

GEM-Programme lassen sich mit SID68 manchmal nicht korrekt starten, da SID vom Betriebssystem als TOS-Anwendung betrachtet wird und die GEM-Routinen deshalb nicht unbedingt zur Verfügung stehen. Hier hilft ein kleiner Trick: Benennen Sie SID68.TOS in SID68.PRG um. Wenn Sie nun SID starten, sieht zwar der Bildschirmaufbau zunächst etwas merkwürdig aus, aber dafür läuft SID jetzt offiziell unter GEM, und alle GEM-Programme lassen sich problemlos austesten.

Soviel zum Debugger. Natürlich ist dies keine vollständige Anleitung zur Benutzung von SID68; Absicht dieses Abschnitts ist es nur, an einigen Beispielen die Benutzung eines Debuggers zu zeigen, zumal Sie vermutlich ohnehin einen anderen als SID68 benutzen werden.

Manche Programme lassen sich mit einem Debugger sehr gut bearbeiten. Bei anderen hat man jedoch Probleme, etwa wenn das Programm ständig etwas auf den Bildschirm schreibt oder sogar grafische Ausgaben produziert (GEM-Programme). In diesem Fall zerstört die Ausgabe des Debuggers ständig den Bildschirmaufbau des Programms.

Schwierig ist das Verfolgen von Interrupts. Deshalb muß man oft zu einer anderen Methode greifen als zum Debugger, um Informationen über den Zustand einen Programms zu erhalten. Eine recht praktische Möglichkeit ist es, sich einige Prozeduren zur Ausgabe von Dezimalzahlen und von Strings zu schreiben. Wenn man einen Makroassembler hat, nimmt man dazu am besten Makros, ansonsten muß man sich mit Subroutinen behelfen. Wichtig ist, daß diese Prozeduren den Zustand des Prozessors nicht verändern, damit die Programmabarbeitung nicht beeinflußt wird, wobei man auch bedenken sollte, daß die meisten Systemroutinen einige Register verändern. Deshalb stellt man an den Anfang der Ausgabeprozeduren am besten den Befehl

```
MOVEM.L D0-D7/A0-A6,-(SP)
```

um alle Registerinhalte zu sichern. Am Ende der Prozedur steht entsprechend

```
MOVEM.L (SP)+,D0-D7/A0-A6
```

Wenn der momentane Zustand des CCR von Bedeutung ist, legen Sie auch dieses auf dem Stack ab. So kann eigentlich kaum etwas schiefgehen.

Besonderheiten des Prozessors MC68000

Als fortschrittlicher Prozessor bietet der MC68000 noch einige zusätzliche Möglichkeiten. Zunächst wäre da die Trennung zwischen User- und Supervisormodus.

Der Supervisormodus

Jedes Programm auf dem ATARI ST wird zunächst einmal im USER-Modus gestartet. Das bedeutet, daß Sie einige Dinge nicht dürfen. Einerseits ist es im Usermodus nicht erlaubt, auf die Systemvariablen zuzugreifen, sei es lesend oder schreibend. Der Speicherbereich von Adresse 0 bis $800 (2048) wird von der MMU, der Speicherverwaltungseinheit des ATARI ST, geschützt. Bei einem Zugriff auf diese Adressen im USER-Modus tritt ein Busfehler auf. Ein weiteres Handicap ist, daß einige Prozessorbefehle nur im Supervisormodus ausgeführt werden dürfen; andernfalls gibt es einen "privilege violation"-Fehler. Die kritischen Befehle, die größtenteils noch nicht erwähnt wurden, sind:

STOP	Prozessor stoppen
RESET	Hardware initialisieren
RTE	Rückkehr von einer Exception
MOVE to SR	Wert ins SR schreiben
AND (word) to SR	Bits im SR löschen
OR (word) to SR	Bits im SR setzen
EOR (word) to SR	Bits im SR invertieren
MOVE from USP	User-Stackpointer lesen

Natürlich ist es im USER-Modus nicht erlaubt, den Inhalt des Systembytes zu verändern, denn dadurch könnte ein Programm ja das Supervisor-Bit setzen und sich dadurch selbst das Privileg des Supervisor-Modus verschaffen. Dies würde dem Konzept der Einschränkung der Möglichkeiten im USER-Modus widersprechen. Ansonsten finden Sie die Erklärung dieser teilweise recht ausgefallenen Befehle im nächsten Kapitel.

Ein Befehl ist noch etwas kritisch:

MOVE from SR	Inhalt des SR lesen

Dieser Befehl ist zwar auf dem MC68000 uneingeschränkt verwendbar und sogar notwendig, wenn man den Inhalt des User-Bytes auslesen will, doch der 68010-Prozessor, der große Bruder des MC68000, gestattet ihn nur im Supervisormodus. Deshalb ist es empfehlenswert, diesen Befehl möglichst nur im Supervisor-Modus zu verwenden, wenn Sie darauf Wert legen, daß ihre Programme auch auf zukünftigen Modellen noch problemlos laufen.

Wie man in den Supervisormodus (und wieder hinaus) gelangt, erfahren Sie in Kapitel 4 unter "Das GEMDOS".

Das Systembyte

Das Systembyte besteht genau wie das Userbyte aus 8 Bits, von denen nur 5 genutzt werden. Die anderen ergeben immer null, wenn sie ausgelesen werden. Das Systembyte belegt die oberen 8 Bits des 16 Bits umfassenden Statusregisters und ist nur im Supervisormodus beschreibbar. Abbildung 2.14 zeigt die Anordnung der Flags im Systembyte.

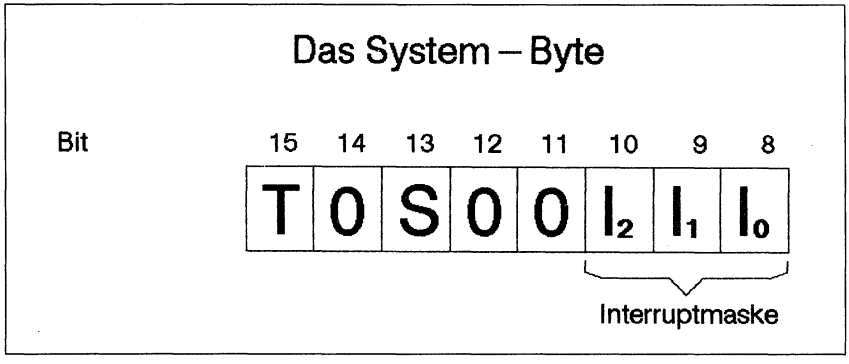


Abb. 2.14: Aufbau des System-Bytes (Bits 8 – 15 des Statusregisters)

Die Interruptmaske

Diese 3 Bits regeln das Auftreten von Interrupts, Unterbrechungen, die den Prozessor veranlassen, für kurze Zeit etwas anderes zu tun. Bei 3 Bits ergeben sich 8 Kombinationen. Davon stehen die Werte 0 bis 7 für Prioritäten von Interrupts. Auf dem ATARI ST werden folgende Prioritätsebenen benutzt:

Ebene	Interrupt
2	Horizontal Blank Interrupt
4	Vertical Blank Interrupt
6	MFP 68901 Interrupt

Genauer wird darauf im Kapitel 6 unter "Programmierung von Interrupts" eingegangen.

Die drei Bits der Interruptmaske legen nun fest, welche Interrupts erlaubt sind und welche nicht. Es können nur jene Interrupts auftreten, deren Priorität höher ist als der Wert der Interruptmaske. Üblicherweise beträgt der Wert der Interruptmaske auf dem ATARI ST 3, wodurch nur die Interrupts der Ebenen 4 und 6 gestattet werden. Die Interruptebene 7 hat die Besonderheit, daß sie auch dann nicht maskierbar ist, wenn die Interruptmaske auf 7 steht; ein Interrupt der Ebene 7 würde also in jedem Fall durchkommen. Auf dem ATARI ST wird allerdings von dieser Möglichkeit kein Gebrauch gemacht. Wenn ein Interrupt auftritt, dann wird die Interruptmaske automatisch auf den Wert der Priorität des Interrupts gesetzt. Dadurch kann die Ausführung der Interruptroutine nur durch Interrupts höherer Priorität unterbrochen werden, aber nicht durch solche gleicher oder niedrigerer Priorität.

Das Supervisor-Flag (S-Bit)

Das S-Bit zeigt an, ob sich der Prozessor im Supervisormodus befindet. 1 steht für Supervisormodus, 0 für Usermodus. Bekanntlich gibt es keine direkte Möglichkeit, vom User- in den Supervisormodus zu gelangen; dafür wird ein Betriebssystemaufruf verwendet. Umgekehrt sollte man nicht direkt das S-Bit manipulieren, um vom Supervisor- in den Usermodus zu gelangen, da es fatale Folgen haben kann, wenn das Betriebssystem von diesem Übergang nichts mitbekommt.

Das Trace-Flag (T-Bit)

Das Trace-Flag ist eine Besonderheit des MC68000. Hilfsmittel zur Fehlersuche, sogenannte Debugger, bieten oft eine Möglichkeit, dem Prozessor bei der Abarbeitung eines Programms genau auf die Finger zu schauen, indem nach jedem ausgeführten Befehl das entsprechende Mnemonic angezeigt wird und die Registerinhalte ausgegeben werden. Um die Programmierung eines solchen Debuggers zu erleichtern, bietet der MC68000 nun den Trace-Modus. Ist

das T-Bit gesetzt, dann wird nach jedem von der CPU ausgeführten Befehl eine bestimmte Exception ausgelöst, in der der Zustand der Register ausgegeben oder etwas ähnlich Nützliches getan werden kann. Erst nach der Beendigung dieser Exception wird der nächste Befehl ausgeführt. Siehe auch in diesem Kapitel unter "Die Exceptions".

Der 68000 bietet einige Möglichkeiten, Exceptions zu behandeln. Es handelt sich dabei um Ausnahmebedingungen, die entweder vom Prozessor selbst kommen – meist durch Programmabstürze – oder von der angeschlossenen Hardware ausgelöst werden. Von außen können dabei Interrupts (Unterbrechungen) oder ein Busfehler ausgelöst werden.

Eine ankommende Exception wird vom Prozessor folgendermaßen behandelt: Die Abarbeitung des gerade laufenden Befehls wird unterbrochen. Dann begibt sich der Prozessor in den Supervisor-Modus und legt den Inhalt des Programmzählers und des Statusregisters auf dem Supervisor-Stack ab. Danach wird eine von der Ursache der Exception abhängige Routine aufgerufen, deren Einsprungpunkt in einer Liste von Adressen in den ersten 1024 Bytes des Speichers festgelegt wird. Wenn die Exception beendet ist und das unterbrochene Programm normal weitergeführt werden soll, so kann das mit dem Befehl RTE (ReTurn from Exception) geschehen, der Statusregister und Programmzähler vom Stack wiederherstellt. Sofern es sich um eine Fehlerbedingung handelt, wird eine Betriebssystemroutine ausgeführt, die je nach Ursache eine bestimmte Anzahl der gefürchteten Bomben auf den Bildschirm bringt. Das auslösende Programm wird abgebrochen, und es wird ein Warmstart des Systems versucht, der leider nicht immer gelingt. Es ist für Programme prinzipiell auch möglich, hier eigene Routinen zur Exception-Behandlung zu verwenden. Debugger machen von dieser Möglichkeit Gebrauch.

An den absoluten Adressen 0 und 4 befinden sich der Stackpointer und der Programmzähler, die beim Start des Systems oder bei einem Reset geladen werden. Deshalb beginnen die Exception-Vektoren erst bei Adresse 8 (Vektornummer 2).

Adresse	Art der Exception
$08	Busfehler – 2 Bomben Dieser Fehler tritt auf, wenn – auf einen nicht existenten Speicherbereich zugegriffen wird – versucht wird, in einen ROM-Bereich zu schreiben – im USER-Modus auf die Systemvariablen ($0000-$0800) oder auf die Hardwareregister (ab $FF7FFF) zugegriffen wird

Adresse	Art der Exception
$0C	Adreßfehler – 3 Bomben Ein Adreßfehler tritt auf, wenn bei einer Wort- oder Langwortoperation auf eine ungerade Adresse zugegriffen wird.
$10	Illegaler Befehl – 4 Bomben Der Prozessor traf auf ein Befehlswort, dem keine Bedeutung zugeordnet ist.
$14	Division durch Null (keine Bomben) kann bei DIVU oder DIVS auftreten. Die Division durch Null führt nicht zu Bomben; die Exception wird zwar ausgeführt, aber der zugeordnete Vektor zeigt direkt auf ein RTE, wodurch sofort mit dem nächsten Befehl fortgefahren wird. Der Zustand des Prozessors ist dadurch zwar teilweise undefiniert, aber das laufende Programm wird wenigstens nicht abgebrochen.
$18	CHK-Befehl – 6 Bomben Beim CHK-Befehl wurde eine Bereichsüberschreitung festgestellt
$1C	TRAPV-Befehl – 7 Bomben Bei der Ausführung eines TRAPV-Befehls war das Overflow-Bit gesetzt, d.h. ein Überlauf bei arithmetischen Operationen ist aufgetreten.
$20	Privilegverletzung – 8 Bomben Ein nur im Supervisor-Modus zulässiger Befehl sollte im User-Modus ausgeführt werden.

Auch wenn man sicher nur selten Programme schreibt, die diese Vektoren selbst benutzen, hilft diese Aufstellung, bei einem Programmabsturz aus der Anzahl der Bomben auf die Ursache der Katastrophe zu schließen. In Anhang E wird noch genauer darauf eingegangen, wie man die Exception-Meldungen zur Fehlersuche nutzen kann.

Kapitel 3

Die Befehle des MC68000 in systematischer Reihenfolge

In diesem Kapitel werden die Befehle des 68000 im einzelnen behandelt. Dabei wird die Beschreibung eines jeden Befehls folgendermaßen gegliedert:

- Zuerst wird das Mnemonik des Befehls mit einer stichwortartigen Beschreibung angegeben.

- Danach ist die Funktion des Befehls in symbolischer Schreibweise dargestellt. Dabei gilt folgende Legende:

Q	Adresse des Quelloperanden
(Q)	Inhalt des Quelloperanden
Z	Adresse des Zieloperanden
(Z)	Inhalt des Zieloperanden
<-	wird zu (Zuweisung)
+-*/	die üblichen mathematischen Operationen
&|	bitweises UND und ODER
~	bitweise Negation (Invertieren)
<a:b>	bezeichnet die Bits a bis b eines Operanden; etwa (Q)<7:0> für das untere Byte des Quelloperanden
Dn	ein beliebiges Datenregister von D0 bis D7
An	ein beliebiges Adreßregister von A0 bis A7
SP	Stackpointer (A7)
USP	User-Stackpointer
SSP	Supervisor-Stackpointer
PC	Programmzähler (Program Counter)
CCR	User-Byte (Condition Code Register)
SR	Status Register (System- und User-Byte)
-(SP)	Ein Wert wird auf dem Stack abgelegt, nachdem der SP um 2 oder 4 verringert wurde.
(SP)+	ein Wert wird vom Stack heruntergeholt. Danach wird der SP um 2 oder 4 erhöht
N,Z,V,B,X	Die Systemflags. Werden sie in Ausdrücken verwendet, so ist ihr Wert mit 1 gleichzusetzen, wenn sie gesetzt sind, mit 0, wenn sie gelöscht sind.

Die Beeinflussung des Programmzählers wird nicht gesondert beschrieben, da sie implizit in jedem Befehl enthalten ist.

- Darunter folgt die Angabe der möglichen Adressierungsarten für diesen Befehl. Die Bedeutung der Adressierungsarten ist in Kapitel 2, Seite 66 beschrieben.

- Es wird angegeben, welche der fünf Bits des Statusregisters beeinflußt werden. Die Bits sind:

  N: Negative (negativ, oberstes Bit gesetzt)
  Z: Zero (Null-Flag)
  V: Overflow (Überlauf-Flag)
  C: Carry (Übertrag)
  X: Extend (Erweiterungsflag, entspricht oft Carry)

Soweit nicht anders angegeben, haben die Flags folgende Standardbedeutung:

Z wird gesetzt, wenn der Zieloperand nach der Ausführung einer arithmetisch/logischen Operation gleich null ist; sonst wird es gelöscht.

N wird gesetzt, wenn der Zieloperand nach der Ausführung einer arithmetisch/logischen Operation negativ ist. Das heißt, das höchstwertige Bit in der verwendeten Verarbeitungsbreite ist gesetzt.

V wird gesetzt, wenn bei einer arithmetischen Operation ein Überlauf bei vorzeichenbehafteter Betrachtung der Zahlen auftritt und das Ergebnis nicht mehr stimmt. Im allgemeinen belanglos bei vorzeichenlosen Zahlen.

C wird gesetzt, wenn bei einer arithmetischen Operation bei vorzeichenbehafteter Betrachtung ein Überlauf (Übertrag) auftritt. C wird auch von den Verschiebebefehlen beeinflußt.

X wird oft genauso wie C gesetzt. Es gibt allerdings einige Operationen, die C, nicht jedoch X beeinflussen.

- Danach folgt eine genaue Beschreibung des Befehls.

- Es kann noch ein Abschnitt über Besonderheiten des Befehls folgen (etwa spezielle Anwendungen des Befehls).

- Wenn die Wirkungsweise des Befehls unklar sein könnte, werden Beispiele angegeben.

Bei den Adressierungsarten gibt es allgemein folgendes zu beachten:

Die Adressierungsarten "Adreßregister indirekt mit Displacement" und "Adreßregister indirekt mit Index und Displacement" werden abgekürzt mit "indirekt mit Displacement" bzw. "indirekt mit Index und Displacement".

Wenn nichts anderes angegeben wird, sind Operationen immer wahlweise in Byte-, Wort- oder Langwortbreite ausführbar.

Bei Zugriffen auf Adreßregister sind nur Wort- und Langwortbreite erlaubt. Wird bei einem Adreßregister als Ziel Wortbreite verwendet, so wird das Wort automatisch auf Langwortformat erweitert, während bei Datenregistern die oberen 16 Bits nicht beeinflußt werden.

Bei den arithmetisch/logischen Befehlen muß einer der Operanden immer ein Datenregister sein. Eine Ausnahme bilden nur die Befehlsvarianten, bei denen man ein "A" oder "I" an das Mnemonik anhängt. Dabei steht "A" für Adreßregister und "I" für Immediate (Direktoperand). Ein "A" hängt man an, wenn Quelle oder Ziel ein Adreßregister ist. Das "I" wird verwendet, wenn die Quelle ein Direktoperand, aber das Ziel kein Datenregister ist. Diese Abwandlungen werden als eigenständige Befehle jeweils nach den Grundbefehlen aufgeführt, da sie sich teilweise in der Beeinflussung der Systemflags unterscheiden.

Die meisten Assembler setzen die "A"- und "I"-Abwandlungen, wenn sie nötig sind, selbständig ein. Wenn man also etwa schreibt

```
ADD 4(A0),A5
```

verläßt man sich einfach auf die Fähigkeit des Assemblers, denn eigentlich meint man folgendes:

```
ADDA.W 4(A0),A5
```

Daneben gibt es noch die "Q"-Abwandlung (Quick). Diese Abwandlung ist nur erlaubt, wenn der Quelloperand direkt angegeben wird und nicht länger als 3 Bits ist. (Einzige Ausnahme: Bei MOVEQ darf der Operand mit Vorzeichen 8 Bit lang sein.) Die "Q"-Variante wird allerdings vom Assembler nicht automatisch erzeugt.

Ein-Operand-Befehle

Bei den Ein-Operand-Befehlen ist der einzige Operand gleichzeitig Eingangswert und Ausgabewert der Operation, die mit ihm durchgeführt werden soll.

Schieben und Rotieren

Die Verschiebebefehle sind hier der Übersichtlichkeit halber nur unter den Ein-Operand-Befehlen aufgeführt, obwohl sie eigentlich jeweils zwei Varianten haben: Die eine hat nur einen Operanden, die andere zwei.

Adressierungsarten:

Typ 1:

ZIEL Bei dem Typ mit nur einem Operanden sind folgende Adressierungsarten erlaubt:

Adreßregister indirekt
indirekt mit Displacement
indirekt mit Index und Displacement
postinkrement/predekrement indirekt
absolut kurz/lang

Bei diesem Typ ist nur der Wort-Modus erlaubt (16 Bits).

Typ2: (zwei Operanden)

QUELLE Datenregister direkt
unmittelbar

ZIEL Datenregister direkt

Es gibt acht Verschiebebefehle:

LSL
LSR
ASL
ASR
ROL
ROR
ROXL
ROXR

logical shift left
Logisches Verschieben nach links

LSL

Operation: (Z) <- (Z) um n Stellen nach links verschoben

Flags:
N : gesetzt, wenn Ergebnis < 0
Z : gesetzt, wenn Ergebnis = 0
V : gesetzt, wenn sich das Vorzeichen ändert
C : enthält das letzte links herausgeschobene Bit
X : wie C

Beschreibung: Besprechen wir zuerst, was den beiden Typen gemeinsam ist. Der Operand wird nach links verschoben (ein Verschieben um ein Bit nach links entspricht einer Multiplikation mit 2, egal ob der Operand vorzeichenbehaftet ist oder nicht). Dabei werden von rechts, also im niedrigsten Bit des Operanden, Nullen nachgeschoben.

Typ 1: Es wird nur um ein Bit nach links geschoben. Das links herausfallende Bit kann man danach im Carry- und X-Flag wiederfinden.

Typ 2: Der Quelloperand wird interpretiert als Anzahl der Stellen, um die nach links verschoben werden soll. Es gibt zwei Möglichkeiten, diese Anzahl anzugeben: Als unmittelbarer Operand, wobei Werte von 1 bis 8 erlaubt sind, oder als Inhalt eines Datenregisters. Bei letzterer Möglichkeit werden nur die unteren 6 Bit des Datenregisters beachtet; man kann also um 0 bis 63 Stellen verschieben. Eine Verschiebung um n Stellen nach links entspricht einer Multiplikation des Operanden mit 2^n. Nach der Operation steht das zuletzt herausgeschobene Bit in Carry- und X-Flag. Das V-Flag wird gesetzt, wenn durch die Verschiebung das Vorzeichen des Operanden geändert wurde.

LSL

logical shift left
Logisches Verschieben nach links

Besonderheit: LSL hat genau die gleiche Wirkung wie ASL.
Beispiele:

zu Typ 1

–1 nach links verschoben ergibt –2 und C = 1,
+2 nach links verschoben ergibt +4 und C = 0,
–3 nach links verschoben ergibt –6 und C = 1

zu Typ 2

Folgender Befehl wird ausgeführt:

```
LSL.W #3,D0
```

Wenn D0 vorher 3 enthielt, ist das Ergebnis 24 und C=0, bei –1 ergibt sich –8 und C=1, und bei +8192 ergibt sich 0 und C=1

logical shift right
logisches Schieben nach rechts

LSR

Operation: (Z) <- (Z) um n Stellen nach rechts verschoben

Flags:
N : gesetzt, wenn Ergebnis < 0
Z : gesetzt, wenn Ergebnis = 0
V : gesetzt, wenn sich das Vorzeichen ändert
C : enthält das letzte rechts herausgeschobene Bit
X : wie C

Beschreibung: Ein vorzeichenloser Operand wird nach rechts verschoben (ein Verschieben nach rechts entspricht einer abrundenden Division durch 2). Dabei rücken von links, also im obersten Bit des Operanden, Nullen nach.

Typ 1: Es wird nur um ein Bit nach rechts geschoben. Das rechts herausfallende Bit steht nach der Operation im Carry- und X-Flag.

Typ 2: Der Quelloperand wird interpretiert als Anzahl der Stellen, um die nach rechts verschoben werden soll. Die Anzahl der Stellen wird entweder direkt angegeben (1 – 8) oder steht in einem Datenregister (0 – 63). Eine Verschiebung um n Stellen nach rechts entspricht einer ganzzahligen Division des Operanden durch 2^n. Nach der Operation steht das zuletzt herausgeschobene Bit im Carry- und X-Flag. Das V-Flag wird gesetzt, wenn durch die Verschiebung das Vorzeichen des Operanden geändert wurde.

Beispiele:

zu Typ 1 +7 nach rechts verschoben ergibt +3 und C = 1,
+8 nach rechts verschoben ergibt +4 und C = 0,
–8 (=65528) nach rechts verschoben ergibt 32764

zu Typ 2 Folgender Befehl wird ausgeführt:

```
LSR.W     #3,D0
```

Wenn D0 vorher 8 enthielt, ist das Ergebnis 1 und C = 0, bei +36 ergibt sich +4 und C = 0

ASL

arithmetic shift left
arithmetisches (vorzeichenbehaftetes) Verschieben nach links

siehe unter LSL (die Befehle sind identisch)

arithmetic shift right
arithmetisches (vorzeichenbehaftetes) Verschieben nach rechts

ASR

Operation: (Z) <- (Z) um n Stellen nach rechts verschoben

Flags:
N : gesetzt, wenn Ergebnis < 0
Z : gesetzt, wenn Ergebnis = 0
V : gesetzt, wenn sich das Vorzeichen ändert
C : enthält das letzte rechts herausgeschobene Bit
X : wie C

Beschreibung: Ein vorzeichenbehafteter Operand wird nach rechts verschoben (ein Verschieben nach rechts entspricht einer abrundenden Division durch 2). Dabei wird das Vorzeichen des Operanden bewahrt.

Typ 1: Es wird nur um ein Bit nach rechts geschoben. Das rechts herausfallende Bit steht nach der Operation im Carry- und X-Flag. Das oberste Bit des Operanden (Vorzeichen) bleibt erhalten und wird in das zweitoberste kopiert.

Typ 2: Der Quelloperand wird interpretiert als Anzahl der Stellen, um die nach rechts verschoben werden soll. Die Anzahl der Stellen wird entweder direkt angegeben (1 – 8) oder steht in einem Datenregister (0 – 63). Eine Verschiebung um n Stellen nach rechts entspricht einer ganzzahligen Division des Operanden durch 2^n. Nach der Operation steht das zuletzt herausgeschobene Bit in Carry- und X-Flag. Das oberste Bit des Operanden (Vorzeichen) bleibt erhalten und wird in die neu hineingeschobenen Bits kopiert.

Beispiele:

zu Typ 1 –1 nach rechts geschoben ergibt –1 und C = 1,
+5 nach rechts geschoben ergibt +2,
–5 nach rechts geschoben ergibt –3

zu Typ 2 Folgender Befehl wird ausgeführt:

```
ASR     #3,D0
```

Wenn D0 vorher +36 enthält, ergibt sich +4 und C = 0, bei –25 ergibt sich –4 und C = 1

ROL

rotate left
rotiere Bits nach links

Operation: (Z) <- (Z) um n Stellen nach links rotiert

Flags:
N : gesetzt, wenn Ergebnis < 0
Z : gesetzt, wenn Ergebnis = 0
V : gesetzt, wenn sich das Vorzeichen ändert
C : enthält das letzte links herausgeschobene Bit
X : unberührt

Beschreibung: Die Bits des Operanden werden nach links rotiert. Das heißt, beim Verschieben nach links herausfallende Bits werden gleichzeitig rechts wieder hineingeschoben.

Typ 1: Es wird nur um ein Bit nach links rotiert. Das links herausfallende Bit steht nach der Operation im niederwertigsten Bit des Operanden und außerdem im Carry (nicht im X-Bit!).

Typ 2: Der Quelloperand wird interpretiert als Anzahl der Stellen, um die nach links rotiert werden soll. Die Anzahl der Stellen wird entweder direkt angegeben (1 – 8) oder steht in einem Datenregister (0 – 63). Nach der Operation steht das zuletzt herausgeschobene Bit im Carry- (nicht im X-Flag!).

Beispiel: Diese Befehlsfolge wird ausgeführt:

```
MOVE    #$1234,D0
ROL     #4,D0
```

Danach enthält D0 $2341

rotate right
rotiere Bits nach rechts

ROR

Operation: (Z) <- (Z) um n Stellen nach rechts rotiert

Flags:
N : gesetzt, wenn Ergebnis < 0
Z : gesetzt, wenn Ergebnis = 0
V : gesetzt, wenn sich das Vorzeichen ändert
C : enthält das letzte rechts herausgeschobene Bit
X : unberührt

Beschreibung: Die Bits des Operanden werden nach rechts rotiert. Das heißt, beim Verschieben nach rechts herausfallende Bits werden gleichzeitig links wieder hineingeschoben.

Typ 1: Es wird nur um ein Bit nach rechts rotiert. Das rechts herausfallende Bit steht nach der Operation im obersten Bit (Bit 15) des Operanden und außerdem im Carry (nicht im X-Bit!).

Typ 2: Der Quelloperand wird interpretiert als Anzahl der Stellen, um die nach rechts rotiert werden soll. Die Anzahl der Stellen wird entweder direkt angegeben (1 – 8) oder steht in einem Datenregister (0 – 63). Nach der Operation steht das zuletzt herausgeschobene Bit im Carry-, aber nicht im X-Flag.

Besonderheit: Mathematisch gesehen entspricht ROR – im Gegensatz zu LSR und ASR – nichts besonderem!

Beispiel: Diese Befehlsfolge wird ausgeführt:

```
MOVE    #$1234,D0
ROR     #4,D0
```

Danach enthält D0 $4123

ROXL

rotate left through extend-flag
Rotieren nach links über X-Flag

Operation: (Z) <- (Z) um n Stellen nach links verschoben

Flags:
N : gesetzt, wenn Ergebnis < 0
Z : gesetzt, wenn Ergebnis = 0
V : gesetzt, wenn sich das Vorzeichen ändert
C : enthält das letzte links herausgeschobene Bit
X : wie C

Beschreibung: Der Zieloperand wird nach links verschoben. Das links herausfallende Bit wird im Carry- und X-Flag abgelegt, während rechts, also im niederwertigsten Bit, der alte Inhalt des X-Flags nachrückt.

Typ 1: Es wird nur um ein Bit nach links geschoben. Das links herausfallende Bit steht nach der Operation im Carry- und X-Flag. In Bit 0 wird der alte Inhalt des X-Flags geschrieben.

Typ 2: Der Quelloperand wird interpretiert als Anzahl der Stellen, um die nach links verschoben werden soll. Die Anzahl der Stellen wird entweder direkt angegeben (1 – 8) oder steht in einem Datenregister (0 – 63). Nach der Operation steht das zuletzt herausgeschobene Bit im Carry- und X-Flag. Der alte Inhalt des X-Flags wird in das zuerst hineingeschobene Bit geschrieben, also in Bit 1 – n, wenn um n Stellen verschoben wurde und l die Verarbeitungsbreite (8, 16, 32) ist. Alle anderen hineingeschobenen Bits sind gelöscht.

Besonderheit: ROXL ist als Erweiterung von LSL für beliebig lange Zahlen gedacht, nicht als Erweiterung von ROL.

rotate right through extend-flag
Rotieren nach rechts über X-Flag

ROXR

Operation: (Z) <- (Z) um n Stellen nach rechts verschoben

Flags:
N : gesetzt, wenn Ergebnis < 0
Z : gesetzt, wenn Ergebnis = 0
V : gesetzt, wenn sich das Vorzeichen ändert
C : enthält das letzte rechts herausgeschobene Bit
X : wie C

Beschreibung: Der Zieloperand wird nach rechts verschoben. Das rechts herausfallende Bit wird im Carry- und X-Flag abgelegt, während links der alte Inhalt des X-Flags nachrückt.

Typ 1: Es wird nur um ein Bit nach rechts geschoben. Das rechts herausfallende Bit steht nach der Operation im Carry- und X-Flag. In Bit 15 wird der alte Inhalt des X-Flags geschrieben.

Typ 2: Der Quelloperand wird interpretiert als Anzahl der Stellen, um die nach rechts verschoben werden soll. Die Anzahl der Stellen wird entweder direkt angegeben (1 – 8) oder steht in einem Datenregister (0 – 63). Nach der Operation steht das zuletzt herausgeschobene Bit im Carry- und X-Flag. Der alte Inhalt des X-Flags wird in das zuerst hineingeschobene Bit geschrieben, also in Bit n – 1, wenn um n Stellen verschoben wurde. Alle anderen hineingeschobenen Bits sind gelöscht.

Besonderheit: ROXR ist als Erweiterung von LSR oder ASR für beliebig lange Zahlen zu sehen, nicht als Erweiterung von ROR, wie man meinen könnte!

Arithmetische und logische Ein-Operand-Befehle

Unter diese Gruppe fallen:

NEG
NEGX
NBCD
NOT
CLR
TST
TAS

Alle diese Befehle bieten folgende Adressierungsmöglichkeiten:

ZIEL	Datenregister direkt
	Adreßregister indirekt
	indirekt mit Displacement
	indirekt mit Index und Displacement
	postinkrement/predekrement indirekt
	absolut kurz/lang

Alle Befehle außer NBCD erlauben Byte-, Wort- oder Langwortbreite.

Invertieren eines Operanden

NOT

Operation: (Z) <- ~(Z)

Flags:
N : gesetzt, wenn Ergebnis < 0
Z : gesetzt, wenn Ergebnis = 0
V : gelöscht
C : gelöscht
X : unberührt

Beschreibung: Der Operand wird invertiert. Das bedeutet, daß jede binäre 0 durch eine 1 ersetzt wird und jede 1 durch eine 0. Diese Operation ist vergleichbar mit dem EOR mit einem Operanden aus lauter Einsen. Die Flags C und V werden immer auf 0 gesetzt. Wenn man den Operanden als vorzeichenbehaftet betrachtet, entspricht NOT dem Invertieren des Vorzeichens und der Subtraktion von 1.

NEG

negate
Vorzeichenumkehr

Operation: (Z) <- ~(Z)+1

Flags:
N : gesetzt, wenn Ergebnis < 0
Z : gesetzt, wenn Ergebnis = 0
V : gesetzt, wenn ein Vorzeichenwechsel stattfand
C : gesetzt, wenn ein Borgen in der obersten Stelle stattfand
X : wie C

Beschreibung: Das Zweierkomplement des Operanden wird gebildet. Intern geht das so vor sich, daß der Prozessor wie bei NOT jedes Bit invertiert und dann 1 zu dem Ergebnis addiert. C und X zeigen an, ob bei der Addition der 1 ein Übertrag entstand (gleichbedeutend damit, daß eine 1 aus einer imaginären höheren Stelle geliehen wurde).

Negate with X-Flag
Vorzeichenwechsel mit X-Flag

NEGX

Operation: (Z) <- ~(Z)+1-X

Flags:
N : gesetzt, wenn Ergebnis < 0
Z : gesetzt, wenn Ergebnis = 0
V : gesetzt, wenn ein Vorzeichenwechsel stattfand
C : gesetzt, wenn ein Borgen in der obersten Stelle stattfand
X : wie C

Beschreibung: Der Operand und das X-Flag werden addiert und das Ergebnis wird negiert. NEGX ist als eine Erweiterung des NEG-Befehls auf Zahlen gedacht, die mehr als 32 Bits lang sind. Um eine solche Zahl zu negieren, wendet man NEG auf die niederwertigsten Bytes an und danach beliebig oft den Befehl NEGX nacheinander auf die höherwertigeren Bytes.

NBCD

negate binary coded decimal with x-flag
negiere BCD mit X-Flag

Operation: (Z) <- 0-(Z)-X im BCD-Verfahren

Flags:
N : undefiniert
Z : gesetzt, wenn Ergebnis = 0
V : undefiniert
C : gesetzt, wenn ein Borgen aus einer höheren BCD-Stelle stattfand
X : wie C

Beschreibung: Das BCD-Format stellt je eine Dezimalziffer durch eine Gruppe von vier Bits dar. Die Null wird durch das Bitmuster 0000 dargestellt, 9 durch 1001 und die Werte dazwischen entsprechend. Die Bitmuster 1010 bis 1111 sind nicht belegt und werden manchmal für die Darstellung von Vorzeichen verwendet. Der 68000er bietet Befehle für den Umgang mit dieser selten verwendeten Zahlendarstellung. NBCD addiert den Wert des X-Flags zum Zieloperanden und negiert ihn anschließend. C und X zeigen dabei an, ob ein Borgen aus einer imaginären höheren Dezimalstelle erfolgte. Dieser Befehl ist nur bei Byte-Operanden erlaubt.

Besonderheit: Es gibt kein NBCD ohne Berücksichtigung des X-Flags. Deshalb vor der Benutzung immer das X-Flag löschen!

Clear
Nullsetzen eines Operanden

CLR

Operation: (Z) <- 0

Flags:
N : gelöscht
Z : gesetzt
V : gelöscht
C : gelöscht
X : unberührt

Beschreibung: Der Operand wird gelöscht. CLR ist eigentlich nur eine Abkürzung für

```
MOVE.x    #0,Ziel
```

Die Flags N, V und C werden auf null gesetzt, während Z auf 1 gesetzt wird.

TST

Test
Operand testen

Operation: keine

Flags:
N : gesetzt, wenn Operand < 0
Z : gesetzt, wenn Operand = 0
V : gelöscht
C : gelöscht
X : unberührt

Beschreibung: Dieser Befehl setzt nur N und Z entsprechend dem Wert des Operanden, läßt diesen aber unverändert. TST wird oft verwendet, um etwa den Wert programmeigener Flags zu überprüfen und abhängig von N oder Z zu verzweigen.

test and set
testen und Bit 7 setzen

TAS

Operation: (Z) <- (Z) mit gesetztem Bit 7

Flags:
N : gesetzt, wenn Operand < 0
Z : gesetzt, wenn Operand = 0
V : gelöscht
C : gelöscht
X : unberührt

Beschreibung: Dieser sehr selten verwendete Befehl ist nur auf Bytes anwendbar. Der Operand wird getestet, und die Flags werden entsprechend gesetzt. Erst dann wird Bit 7 gesetzt. Dieser Befehl ist nur bei Time-Sharing-Verfahren interessant und wird auf dem ST kaum verwendet.

Besonderheit: Die Flags werden vor der Operation gesetzt.
TAS ist durch keine Exception abzubrechen.

EXT und SWAP

Hier sind die Befehle aufgeführt, die in die beiden anderen Gruppen der Ein-Operand-Befehle nicht hineinpassen: EXT und SWAP.

Sign Extend
Operand vorzeichenrichtig auf Wort- oder Langwortformat erweitern

EXT

Adressierungsart: Datenregister direkt

Flags:
N : gesetzt, wenn Operand < 0
Z : gesetzt, wenn Operand = 0
V : gelöscht
C : gelöscht
X : unberührt

Beschreibung: Der Operand in Byte- oder Wortbreite wird vorzeichenrichtig auf Wort- bzw. Langwortbreite erweitert. Der Befehl

```
EXT.W Ziel
```

überträgt den Inhalt von Bit 7 des Operanden in die Bits 8 bis 15.

```
EXT.L Ziel
```

überträgt Bit 15 in die Bits 16 bis 31.

SWAP

Hochwertiges und niederwertiges Wort vertauschen

Adressierungsart: Datenregister direkt

Flags:
N : gesetzt, wenn Operand < 0
Z : gesetzt, wenn Operand = 0
V : gelöscht
C : gelöscht
X : unberührt

Beschreibung: Der Prozessor tauscht die oberen 16 Bits des angegebenen Datenregisters gegen die unteren 16 Bits aus.

Zwei-Operand-Befehle

Allen Befehlen dieser Gruppe ist gemeinsam, daß der Quelloperand mit dem Zieloperanden verknüpft und das Ergebnis im Zieloperanden abgelegt wird. Einzige Ausnahme: EXG

Die MOVE-Befehle

Es gibt gleich eine ganze Gruppe von MOVE-Befehlen, die sich durch die Art der ansprechbaren Operanden und die Zugriffsmöglichkeit auf spezielle Prozessorregister unterscheiden:

MOVE
MOVEA
MOVEM
MOVEP
MOVEQ
MOVE to CCR
MOVE to SR
MOVE from SR
MOVE USP
EXG

MOVE

Wert in den Zieloperanden schreiben

Operation: (Z) <- (Q)

Flags:
N : gesetzt, wenn Quelle < 0
Z : gesetzt, wenn Quelle = 0
V : gelöscht
C : gelöscht
X : unberührt

Adressierungsarten:

QUELLE
Datenregister direkt
Adreßregister direkt
Adreßregister indirekt
indirekt mit Displacement
indirekt mit Index und Displacement
indirekt/PC-relativ mit Displacement
indirekt/PC-relativ mit Index und Displacement
postinkrement/predekrement indirekt
absolut kurz/lang
unmittelbar

ZIEL
Datenregister direkt
Adreßregister indirekt
indirekt mit Displacement
indirekt mit Index und Displacement
postinkrement/predekrement indirekt
absolut kurz/lang

Beschreibung: Der Wert des Quelloperanden wird in den Zieloperanden geschrieben. Dabei werden N und Z entsprechend dem übertragenen Wert gesetzt, während C und V gelöscht werden.

Besonderheit: Bei diesem Befehl gilt die Beschränkung nicht, daß entweder Ziel oder Quelle ein Datenregister sein muß.

move to address register
Laden ins Adreßregister

MOVEA

Operation: An <- (Q)

Flags:
N : unberührt
Z : unberührt
V : unberührt
C : unberührt
X : unberührt

Adressierungsarten:

QUELLE
Datenregister direkt
Adreßregister direkt
Adreßregister indirekt
indirekt/PC-relativ mit Displacement
indirekt/PC-relativ mit Index und Displacement
postinkrement/predekrement indirekt
absolut kurz/lang
unmittelbar

ZIEL Adreßregister direkt

Beschreibung: Der angegebene 32-Bit-Operand wird in ein Adreßregister geladen. Es ist auch der Wort-Modus erlaubt, wobei das Wort vorzeichenrichtig auf 32 Bit erweitert wird. Im Gegensatz zum normalen MOVE läßt MOVEA die Flags unverändert.

Besonderheit: Die meisten Assembler bilden MOVE auf MOVEA ab, wenn die Adressierungsarten es verlangen.

MOVEM

move multiple registers
mehrere Register in/aus Speicher laden

Operation: <Registergruppe> <- (Q) (Q+4) (Q+8) ...
oder
(Z) (Z+4) (Z+8) ... <- <Registergruppe>

Flags:
N : unberührt
Z : unberührt
V : unberührt
C : unberührt
X : unberührt

Adressierungsarten:

QUELLE
Adreßregister indirekt
indirekt/PC-relativ mit Displacement
indirekt/PC-relativ mit Index und Displacement
postinkrement indirekt
absolut kurz/lang
Registerliste

ZIEL
Adreßregister indirekt
indirekt/PC-relativ mit Displacement
indirekt/PC-relativ mit Index und Displacement
predekrement indirekt
absolut kurz/lang
Registerliste

Beschreibung: Dieser Befehl dient dazu, ausgewählte Daten- und Adreßregister schnell aufeinanderfolgend in den Speicher zu schreiben oder sie wieder zu lesen. Die ausgewählten Register werden dabei in einem 16-Bit-Wort gespeichert, wobei eine 1 dem Prozessor sagt, daß das entsprechende Register bewegt werden soll. Der Assembler bietet folgende Möglichkeit, eine Registerliste anzugeben:

- es kann ein Registername angegeben werden (etwa D2)
- es können zwei Registernamen, getrennt durch einen Strich (–) angegeben werden. Die Datenregister D0 bis D7 werden

move multiple registers
mehrere Register in/aus Speicher laden

MOVEM

auf die Zahlen 0 bis 7, die Adreßregister A0 bis A7 auf 8 bis 15 abgebildet. Durch die Bereichsangabe werden alle Register angesprochen, die den Zahlen zwischen dem Wert des ersten und dem Wert des zweiten angegebenen Registers entsprechen. D4 – A2 beispielsweise bezeichnet die Register D4 bis D7 und A0 bis A2; D0 – A7 bezeichnet alle Daten- und Adreßregister. Mehrere Operanden der obigen zwei Arten können durch "/" miteinander kombiniert werden.

Normalerweise werden beim Schreiben in den Speicher zuerst die Datenregister in aufsteigender Reihenfolge und dann die Adreßregister abgelegt. Eine Ausnahme bildet die Predekrement-Adressierungsart, bei der die Reihenfolge genau umgekehrt ist. Der Grund ist, daß oft Register mit dem Prädekrement-Modus auf dem Stack abgelegt werden, die man später mittels Postinkrement wieder einlesen will. Um die Symmetrie zu erhalten, muß beim Ablegen im Prädekrement-Modus die umgekehrte Reihenfolge eingehalten werden. Wichtig ist auf jeden Fall, daß das Ablegen auf und Lesen vom Stack mit MOVEM funktioniert.

Die Flags werden nicht beeinflußt.

MOVEP

move peripheral
schreiben/lesen mit peripheren Geräten

Operation:

(Z) <- Dn<31:24> oder (Z) <- Dn<15:8>
(Z+2) <- Dn<23:16> (Z) <- Dn<7:0>
(Z+4) <- Dn<15:8>
(Z+6) <- Dn<7:0>
oder in umgekehrter Richtung

Flags:

N : unberührt
Z : unberührt
V : unberührt
C : unberührt
X : unberührt

Adressierungsarten:

QUELLE Adreßregister indirekt mit Displacement
Datenregister direkt

ZIEL Adreßregister indirekt mit Displacement
Datenregister direkt

Beschreibung: Dieser Befehl dient zum Schreiben auf oder Lesen aus bestimmten Hardwareregistern. Aus Hardwaregründen ist es einfach, 8-Bit-Chips so mit der CPU zu verdrahten, daß an jeder Wortadresse nur ein 8-Bit-Register des Chips erreichbar ist. (Die Register des Videochips Shifter werden etwa auf diese Weise angesprochen.) Deshalb teilt MOVEM den Operanden in einzelne Bytes auf, die dann jeweils mit einem Byte Abstand geschrieben werden. Das erste Byte wird an die angegebene Adresse geschrieben (die auch ungerade sein darf), das nächste 2 Bytes weiter und so fort. Beim Lesen aus dem Speicher geht das Ganze genau umgekehrt vor sich. Es sind nur zwei Modi erlaubt, der Transfer von einem Register zu einem Adreßregister indirekt mit Displacement oder umgekehrt. Beide können im Wort-Modus (2 Bytes) oder Langwort-Modus (4 Bytes) verwendet werden. Die Flags werden von diesem Befehl nicht beeinflußt.

move quick
Laden eines Datenregisters mit einem kurzen unmittelbaren Operanden

MOVEQ

Operation: Dn <- (Q) vorzeichenerweitert

Flags:
N : gesetzt, wenn Operand < 0
Z : gesetzt, wenn Operand = 0
V : gelöscht
C : gelöscht
X: unberührt

Adressierungsarten:

QUELLE 8 Bit unmittelbar

ZIEL Datenregister direkt

Beschreibung: Der 8-Bit-Wert wird in das unterste Byte des angegebenen Datenregisters geschrieben. Dann wird der Wert vorzeichenrichtig auf die gewünschte Länge erweitert, indem Bit 7 in alle höheren Bits übertragen wird.

MOVEQ ist besonders schnell, da der Wert gleich im Befehlswort untergebracht wird und so keine weiteren Speicherzugriffe erfolgen müssen.

Besonderheit: MOVEQ ist der einzige "Quick"-Befehl, bei dem der Wert 8 Bits lang sein darf (sonst sind es nur 3 Bits). Dafür ist man beim Ziel auf ein Datenregister beschränkt.

MOVE to CCR

move to condition code register
Wert ins Flag-Register schreiben

Operation: CCR <- (Q)

Flags:
N : Bit 3 des Quelloperanden
Z : Bit 2 des Quelloperanden
V : Bit 1 des Quelloperanden
C : Bit 0 des Quelloperanden
X : Bit 4 des Quelloperanden

Adressierungsarten:

QUELLE
Datenregister direkt
Adreßregister indirekt
indirekt mit Displacement
indirekt mit Index und Displacement
postinkrement/predekrement indirekt
absolut kurz/lang
unmittelbar

Beschreibung: Dieser Befehl ist nur im Wort-Modus erlaubt. Die unteren 8 Bits des Quelloperanden werden in das CCR geschrieben, wobei die Flags in der oben beschriebenen Weise ihre Werte erhalten. Die Bits 5 bis 7 werden ignoriert; sie sind im CCR immer null.

Besonderheit: Es gibt kein "MOVE from CCR", dazu muß der Befehl "MOVE from SR" verwendet werden.

move to status register
Wert ins Status-Register (User-Byte und System-Byte) schreiben

MOVE to SR

Operation: SR <- (Q)

Flags:
N : Bit 3 des Quelloperanden
Z : Bit 2 des Quelloperanden
V : Bit 1 des Quelloperanden
C : Bit 0 des Quelloperanden
X : Bit 4 des Quelloperanden

Adressierungsarten:

QUELLE
Datenregister direkt
Adreßregister indirekt
indirekt/PC-relativ mit Displacement
indirekt/PC-relativ mit Index und Displacement
postinkrement/predekrement indirekt
absolut kurz/lang
unmittelbar

Beschreibung: Auch dieser Befehl existiert nur im Wort-Modus. Der Quelloperand wird ins Statusregister geschrieben. Dadurch können sowohl die Systemflags als auch Interruptebene, Prozessormodus und Trace-Bit geändert werden. Deshalb ist dieser Befehl nur im Supervisor-Modus erlaubt.

Besonderheit: Privilegierter Befehl

MOVE from SR

move from status register
Inhalt des Status-Registers (User-Byte und System-Byte) lesen

Operation: (Z) <- SR

Flags:
N : unberührt
Z : unberührt
V : unberührt
C : unberührt
X : unberührt

Adressierungsarten:

ZIEL
Datenregister direkt
Adreßregister indirekt
indirekt mit Displacement
indirekt mit Index und Displacement
postinkrement/predekrement indirekt
absolut kurz/lang

Beschreibung: Der Wert des Statusregisters (16 Bit) wird ausgelesen und im Zieloperanden abgelegt. So können Systemflags, Interruptebene, Prozessormodus und Trace-Bit überprüft werden. Die Flags werden durch diesen Befehl nicht verändert.

Besonderheit: Auf dem Prozessor 68000 ist "MOVE from SR" frei verwendbar. Nur auf dem 68010 darf dieser Befehl ausschließlich im Supervisormodus benutzt werden, da es dort einen Befehl "MOVE from CCR" gibt, der das Auslesen der Userflags ermöglicht. Um die Lauffähigkeit Ihrer Programme auch auf zukünftigen Prozessorgenerationen zu sichern, sollten Sie deshalb "MOVE from SR" möglichst nur im Supervisormodus verwenden.

move user stack pointer
Zugriff auf den USP vom Supervisormodus aus

MOVE USP

Operation: (Z) <- USP
oder
USP <- (Q)

Flags: N : unberührt
Z : unberührt
V : unberührt
C : unberührt
X : unberührt

Adressierungsarten:

QUELLE Adreßregister direkt

ZIEL Adreßregister direkt

Beschreibung: Der Stackpointer A7 besteht eigentlich aus zwei Registern: eines für den Supervisormodus und eines für den Usermodus. Das jeweils andere ist normalerweise nicht erreichbar. MOVE USP bietet nun die Möglichkeit, im Supervisormodus auf den "User Stack Pointer" zuzugreifen. Dabei kann der Inhalt des USP in ein Adreßregister geschrieben oder aus einem gelesen werden. Die Flags werden nicht beeinflußt. Da dieser Befehl im Usermodus unsinnig wäre, ist er nur im Supervisormodus erlaubt.

Besonderheit: Privilegierter Befehl

EXG

exchange registers
Inhalte zweier 32-Bit-Register vertauschen

Operation: (Z) <- (Q), (Q) <- (Z)

Flags:
N : unberührt
Z : unberührt
V : unberührt
C : unberührt
X : unberührt

Adressierungsarten:

QUELLE
Datenregister direkt
Adreßregister direkt

ZIEL
Datenregister direkt
Adreßregister direkt

Beschreibung: Die vollen 32 Bit der beiden angegebenen Register werden vertauscht. Dabei dürfen Daten- und Adreßregister beliebig gemischt werden.

Besonderheit: EXG ist der einzige Befehl, bei dem der Quelloperand verändert wird.

Arithmetische Befehle

Bei den arithmetischen Befehlen berücksichtigt der 68000-Befehlssatz die vier Grundoperationen Addition, Subtraktion, Multiplikation und Division. Auch die CMP-Befehle werden zu den arithmetischen gerechnet, da sie im Grunde eine Subtraktion durchführen. Unter diese Gruppe fallen:

ADD
ADDX
ADDA
ADDI
ADDQ
ABCD
SUB
SUBX
SUBA
SUBI
SUBQ
SBCD
CMP
CMPA
CMPI
CMPM
MULU
MULS
DIVU
DIVS

ADD

Addition

Operation: (Z) <- (Z)+(Q)

Flags:
N : gesetzt, wenn Ergebnis < 0
Z : gesetzt, wenn Ergebnis = 0
V : gesetzt, wenn der Zahlenbereich der vorzeichenbehafteten Zahlen verlassen wird
C : gesetzt, wenn ein Übertrag auftritt
X : wie C

Adressierungsarten:

QUELLE
Datenregister direkt
Adreßregister direkt
Adreßregister indirekt
indirekt/PC-relativ mit Displacement
indirekt/PC-relativ mit Index und Displacement
postinkrement/predekrement indirekt
absolut kurz/lang
unmittelbar

ZIEL
Datenregister direkt
Adreßregister indirekt
indirekt mit Displacement
indirekt mit Index und Displacement
postinkrement/predekrement indirekt
absolut kurz/lang

Beschreibung: Der Quelloperand wird zum Zieloperanden addiert. Nach der Operation werden die Flags entsprechend dem Ergebnis gesetzt. Bei der Adressierung gilt es zu beachten, daß mindestens einer der Operanden in einem Datenregister stehen muß. Andernfalls handelt es sich um eine der Varianten ADDI oder ADDA.

add with extend-flag
Addition mit X-Flag

ADDX

Operation: (Z) <- (Z)+(Q)+X

Flags:
N : gesetzt, wenn Ergebnis < 0
Z : gesetzt, wenn Ergebnis = 0
V : gesetzt, wenn der Zahlenbereich der vorzeichenbehafteten Zahlen verlassen wird
C : gesetzt, wenn ein Übertrag auftritt
X : wie C

Adressierungsarten:

QUELLE Datenregister direkt
Adreßregister indirekt

ZIEL Datenregister direkt
Adreßregister indirekt

Beschreibung: Der Quelloperand wird unter Berücksichtigung früher aufgetretener Überträge zum Zieloperanden addiert. Nach der Operation werden die Flags entsprechend dem Ergebnis gesetzt.

ADDX bietet nur zwei Möglichkeiten, die Operanden zu adressieren: Entweder befinden sich beide im Speicher und werden indirekt über ein Datenregister angesprochen, oder beide befinden sich in Datenregistern. Ein Mischen der beiden Adressierungsarten ist nicht erlaubt.

ADDX dient zur Erweiterung des ADD-Befehls auf beliebig lange Zahlen, da dieser Befehl den Übertrag berücksichtigt und ihn auch selbst neu setzt. Man addiert zuerst die niederwertigsten Bytes mittels ADD, dann aufsteigend die höherwertigeren Bytes mit beliebig vielen ADDX-Befehlen.

ADDA

add address register
Addieren zu einem Adreßregister

Operation: An <- An+(Q)

Flags:
N : unberührt
Z : unberührt
V : unberührt
C : unberührt
X : unberührt

Adressierungsarten:

QUELLE
Datenregister direkt
Adreßregister direkt
Adreßregister indirekt
indirekt/PC-relativ mit Displacement
indirekt/PC-relativ mit Index und Displacement
postinkrement/predekrement indirekt
absolut kurz/lang
unmittelbar

ZIEL Adreßregister direkt

Beschreibung: Der Operand, der 16 oder 32 Bits lang ist, wird zum Inhalt des angegebenen Adreßregisters addiert. Wenn der Quelloperand nur 16 Bits lang ist, wird er vorher intern auf 32 Bit erweitert. Im Gegensatz zu ADD ändert ADDA die Flags nicht.

Besonderheit: ADD wird von den meisten Assemblern nach ADDA übersetzt, wenn die Kombination der Adressierungsarten es verlangt.

add immediate
unmittelbaren Operanden addieren

ADDI

Operation: (Z) <- (Z)+(Q)

Flags:
N : gesetzt, wenn Ergebnis < 0
Z : gesetzt, wenn Ergebnis = 0
V : gesetzt, wenn der Zahlenbereich der vorzeichenbehafteten Zahlen verlassen wird
C : gesetzt, wenn ein Übertrag auftritt
X : wie C

Adressierungsarten:

QUELLE unmittelbar

ZIEL Datenregister direkt
Adreßregister indirekt
indirekt mit Displacement
indirekt mit Index und Displacement
postinkrement/predekrement indirekt
absolut kurz/lang

Beschreibung: Die Wirkungsweise ist genau wie beim ADD-Befehl: Der Quelloperand wird zum Zieloperanden addiert. ADDI bietet nur eine Erweiterung der Adressierungsarten, denn bei ADD ist man mit einem unmittelbaren Operanden als Quelle auf ein Datenregister als Ziel beschränkt. ADDI erlaubt hingegen auch ein Ansprechen des Speichers im Zusammenhang mit unmittelbaren Daten.

Besonderheit: ADD wird von den meisten Assemblern nach ADDI übersetzt, wenn die Kombination der Adressierungsarten es verlangt.

ADDQ

add quick
addition eines kurzen unmittelbaren Operanden

Operation: (Z) <- (Z)+(Q)

Flags:
N : gesetzt, wenn Ergebnis < 0
Z : gesetzt, wenn Ergebnis = 0
V : gesetzt, wenn der Zahlenbereich der vorzeichenbehafteten Zahlen verlassen wird.
C : gesetzt, wenn ein Übertrag auftritt
X : wie C

Adressierungsarten:

QUELLE unmittelbar

ZIEL
Datenregister direkt
Adreßregister direkt
Adreßregister indirekt
indirekt mit Displacement
indirekt mit Index und Displacement
postinkrement/predekrement indirekt
absolut kurz/lang

Beschreibung: Ein 3-Bit-Operand wird zum Zieloperanden addiert. Die Flags werden genau wie beim ADD-Befehl gesetzt. Dabei ist ein Zahlenbereich von 1 bis 8 erlaubt; die 8 wird dabei durch das Bitmuster 000 codiert.

add binary coded decimal
BCD-Zahlen mit X-Flag addieren

ABCD

Operation: (Z) <- (Z)+(Q)+X im BCD-Verfahren

Flags:
N : undefiniert
Z : gesetzt, wenn Ergebnis = 0
V : undefiniert
C : gesetzt, wenn ein Übertrag entsteht
X : wie C

Adressierungsarten:

QUELLE Datenregister direkt
Adreßregister indirekt

ZIEL Datenregister direkt
Adreßregister indirekt

Beschreibung: Wie alle BCD-Befehle existiert ABCD nur im Byte-Modus. Zwei Bytes und der Inhalt des X-Flags werden nach dem BCD-Verfahren addiert (siehe NBCD). Es sind nur die Adressierungsarten "Datenregister direkt" und "Adreßregister indirekt" erlaubt, die nicht gemischt werden dürfen. X, Z und C werden dem Ergebnis der Addition entsprechend gesetzt, während der Wert von N und V nicht definiert ist.

Besonderheit: Es gibt kein ABCD ohne Berücksichtigung des X-Flags, deshalb sollte vor der ersten Verwendung das X-Flag mittels "MOVE to CCR" gelöscht werden.

SUB

subtract
Subtraktion des Quelloperanden vom Zieloperanden

Operation: (Z) <- (Z)-(Q) (genauer (Z)+~(Q)+1)

Flags:
N : gesetzt, wenn Ergebnis < 0
Z : gesetzt, wenn Ergebnis = 0
V : gesetzt, wenn der Zahlenbereich der vorzeichenbehafteten Zahlen verlassen wird
C : gesetzt, wenn ein Borgen in der obersten Stelle auftritt
X : wie C

Adressierungsarten:

QUELLE
Datenregister direkt
Adreßregister direkt
Adreßregister indirekt
indirekt/PC-relativ mit Displacement
indirekt/PC-relativ mit Index und Displacement
postinkrement/predekrement indirekt
absolut kurz/lang
unmittelbar

ZIEL
Datenregister direkt
Adreßregister indirekt
indirekt mit Displacement
indirekt mit Index und Displacement
postinkrement/predekrement indirekt
absolut kurz/lang

Beschreibung: Der Quelloperand wird vom Zieloperanden abgezogen. Intern wird dazu jedes Bit des Quelloperanden invertiert, 1 hinzugezählt und das Ergebnis zum Zieloperanden addiert. Nach der Operation werden die Flags entsprechend dem Ergebnis gesetzt. Bei der Adressierung gilt es zu beachten, daß mindestens einer der Operanden in einem Datenregister stehen muß. Andernfalls handelt es sich um eine der Varianten SUBI oder SUBA. Beachten Sie, daß die Reihenfolge der Operanden genau umgekehrt ist, als man es gewohnt ist:

```
SUB A,B
```

berechnet B-A und legt das Ergebnis in B ab.

subtract with extend-flag
Subtraktion mit X-Flag

SUBX

Operation: (Z) <- (Z)-(Q)-X
(genauer (Z)+~(Q)+1-X)

Flags:
N : gesetzt, wenn Ergebnis < 0
Z : gesetzt, wenn Ergebnis = 0
V : gesetzt, wenn der Zahlenbereich der vorzeichenbehafteten Zahlen verlassen wird
C : gesetzt, wenn ein Borgen in der obersten Stelle auftritt
X : wie C

Adressierungsarten:

QUELLE Datenregister direkt
Adreßregister indirekt

ZIEL Datenregister direkt
Adreßregister indirekt

Beschreibung: Der Quelloperand wird unter Berücksichtigung früher aufgetretener Überträge vom Zieloperanden abgezogen. Nach der Operation werden die Flags entsprechend dem Ergebnis gesetzt. Entweder befinden sich beide Operanden im Speicher und werden indirekt über ein Datenregister angesprochen, oder sie befinden sich in Datenregistern. Ein Mischen der beiden Adressierungsarten ist nicht erlaubt.

SUBX dient zur Erweiterung des SUB-Befehls auf beliebig lange Zahlen, da dieser Befehl den Übertrag berücksichtigt und ihn auch selbst neu setzt. Man subtrahiert zuerst die niederwertigsten Bytes mittels SUB, dann aufsteigend die höherwertigeren Bytes mit beliebig vielen SUBX-Befehlen.

SUBA

subtract from address register
Subtrahieren von einem Adreßregister

Operation: An <- An-(Q)
(genauer An+~(Q)+1)

Flags: N : unberührt
Z : unberührt
V : unberührt
C : unberührt
X : unberührt

Adressierungsarten:

QUELLE Datenregister direkt
Adreßregister direkt
Adreßregister indirekt
indirekt/PC-relativ mit Displacement
indirekt/PC-relativ mit Index und Displacement
postinkrement/predekrement indirekt
absolut kurz/lang
unmittelbar

ZIEL Adreßregister direkt

Beschreibung: Der Operand, der 16 oder 32 Bits lang ist, wird vom Inhalt des angegebenen Adreßregisters subtrahiert. Wenn der Quelloperand nur 16 Bits lang ist, wird er vorher intern auf 32 Bits erweitert. Im Gegensatz zu SUB ändert SUBA die Flags nicht.

Besonderheit: SUB wird von den meisten Assemblern in SUBA verwandelt, wenn die verwendeten Adressierungsarten es verlangen.

subtract immediate
unmittelbaren Operanden subtrahieren

SUBI

Operation: (Z) <- (Z)-(Q)
(genauer (Z)+~(Q)+1)

Flags:
N : gesetzt, wenn Ergebnis < 0
Z : gesetzt, wenn Ergebnis = 0
V : gesetzt, wenn der Zahlenbereich der vorzeichenbehafteten Zahlen verlassen wird
C : gesetzt, wenn ein Borgen in der höchsten Stelle auftritt
X : wie C

Adressierungsarten:

QUELLE unmittelbar

ZIEL Datenregister direkt
Adreßregister indirekt
indirekt mit Displacement
indirekt mit Index und Displacement
postinkrement/predekrement indirekt
absolut kurz/lang

Beschreibung: Die Wirkungsweise ist genau wie beim SUB-Befehl: Der Quelloperand wird vom Zieloperanden abgezogen. SUBI bietet nur eine Erweiterung der Adressierungsarten, denn bei SUB ist man mit einem unmittelbaren Operanden als Quelle an ein Datenregister als Ziel gebunden. SUBI erlaubt hingegen auch ein Ansprechen des Speichers im Zusammenhang mit unmittelbaren Daten.

Besonderheit: SUB wird von den meisten Assemblern wie SUBI behandelt, wenn die verwendeten Adressierungsarten es verlangen.

SUBQ

substract quick
Subtraktion eines kurzen unmittelbaren Operanden

Operation: (Z) <- (Z)-(Q)

Flags:
N : gesetzt, wenn Ergebnis < 0
Z : gesetzt, wenn Ergebnis = 0
V : gesetzt, wenn der Zahlenbereich der vorzeichenbehafteten Zahlen verlassen wird.
C : gesetzt, wenn ein Borgen in der obersten Stelle auftritt
X : wie C

Adressierungsarten:

QUELLE unmittelbar

ZIEL Datenregister direkt
Adreßregister direkt
Adreßregister indirekt
indirekt mit Displacement
indirekt mit Index und Displacement
postinkrement/predekrement indirekt
absolut kurz/lang

Beschreibung: Ein 3-Bit-Operand wird vorzeichenlos auf die Breite des Zieloperanden erweitert und dann von diesem abgezogen. Die Flags werden genau wie beim SUB-Befehl gesetzt. Ein Zahlenbereich von 1 bis 8 ist erlaubt; die 8 wird dabei durch das Bitmuster 000 codiert.

subtract binary coded decimal
BCD-Zahlen mit X-Flag subtrahieren

SBCD

Operation: (Z) <- (Z)-(Q)-X im BCD-Verfahren
(genauer (Z)+~(Q)+1-X)

Flags: N : undefiniert
Z : gesetzt, wenn Ergebnis = 0
V : undefiniert
C : gesetzt, wenn ein Borgen in der obersten Stelle auftritt
X : wie C

Adressierungsarten:

QUELLE Datenregister direkt
Adreßregister indirekt

ZIEL Datenregister direkt
Adreßregister indirekt

Beschreibung: SBCD existiert nur im Byte-Modus. Ein Byte wird von einem anderen im BCD-Verfahren subtrahiert (siehe NBCD). Es sind nur die Adressierungsarten "Datenregister direkt" und "Adreßregister indirekt" erlaubt, die nicht gemischt werden dürfen. X, Z und C werden dem Ergebnis der Subtraktion entsprechend gesetzt, während der Wert von N und V nicht definiert ist.

Besonderheit: Es gibt kein SBCD ohne Berücksichtigung des X-Flags, deshalb sollte vor der ersten Verwendung das X-Flag mittels "MOVE to CCR" gelöscht werden.

CMP

compare
Vergleich zweier Operanden

Operation: (Q)-(Z)
(genauer (Q)+~(Z)+1)

Flags:
N : gesetzt, wenn Quelle < Ziel
Z : gesetzt, wenn Quelle = Ziel
V : gesetzt, wenn bei der internen Subtraktion der Zahlenbereich der vorzeichenbehafteten Zahlen verlassen wird
C : gesetzt, wenn Quelle ≤ Ziel
X : unberührt

Adressierungsarten:

QUELLE
Datenregister direkt
Adreßregister direkt
Adreßregister indirekt
indirekt/PC-relativ mit Displacement
indirekt/PC-relativ mit Index und Displacement
postinkrement/predekrement indirekt
absolut kurz/lang
unmittelbar

ZIEL
Datenregister direkt

Beschreibung: Das Zieldatenregister wird mit dem Quelloperanden verglichen. Beide Operanden werden nicht verändert, nur die Flags werden entsprechend dem Ergebnis des Vergleiches gesetzt. Intern geht der Vergleich so vor sich, daß das Ziel von der Quelle abgezogen wird (genau umgekehrt gegenüber dem SUB-Befehl!) und die Flags entsprechend dem Ergebnis gesetzt werden. Normalerweise folgt dem CMP-Befehl bald eine bedingte Verzweigung, damit je nach dem Ergebnis des Vergleiches verschiedene Befehle ausgeführt werden können. Auch hier ist die Reihenfolge der Operanden genau andersherum als bei der mathematischen Schreibweise:

compare
Vergleich zweier Operanden

CMP

```
CMP A,B
```

entspricht

```
B relop A
```

wobei "relop" (Relations-Operator) einer von >, <, ≥, ≤ ist, entsprechend den Bedingungskürzeln HI, CS, CC, LS für vorzeichenlose Zahlen oder GT, LT, GE, LE für Zweierkomplementzahlen. Beim Test auf Gleichheit oder Ungleichheit spielt die Reihenfolge der Operanden natürlich keine Rolle.

CMPA

compare with address register
Vergleich mit einem Adreßregister

Operation: (Q)-An
(genauer (Q)+~An+1)

Flags:
N : gesetzt, wenn Quelle < An
Z : gesetzt, wenn Quelle = An
V : gesetzt, wenn bei der internen Subtraktion der Zahlenbereich der vorzeichenbehafteten Zahlen verlassen wird
C : gesetzt, wenn Quelle ≤ An
X : unberührt

Adressierungsarten:

QUELLE
Datenregister direkt
Adreßregister direkt
Adreßregister indirekt
indirekt/PC-relativ mit Displacement
indirekt/PC-relativ mit Index und Displacement
postinkrement/predekrement
indirekt absolut kurz/lang
unmittelbar

ZIEL Adreßregister direkt

Beschreibung: Ein Adreßregister wird mit der Quelle verglichen, und die Flags werden entsprechend gesetzt, während beide Operanden nicht verändert werden. Intern wird dafür das Adreßregister von der (vorzeichenerweiterten) Quelle abgezogen. Bei der Quelle sind nur Wort- und Langwortbreite erlaubt. Beachten Sie auch hier die Reihenfolge der Operanden (vergleiche CMP).

Besonderheit: CMP wird von den meisten Assemblern als CMPA betrachtet, wenn die Adressierungsarten es erfordern.

compare immediate
Ziel mit unmittelbarem Operanden vergleichen

CMPI

Operation: (Q)-(Z)
(genauer (Q)+~(Z)+1)

Flags:
N : gesetzt, wenn Quelle < Ziel
Z : gesetzt, wenn Quelle = Ziel
V : gesetzt, wenn bei der internen Subtraktion der Zahlenbereich der vorzeichenbehafteten Zahlen verlassen wird
C : gesetzt, wenn Quelle ≤ Ziel
X : unberührt

Adressierungsarten:

QUELLE unmittelbar

ZIEL Datenregister direkt
Adreßregister indirekt
indirekt mit Displacement
indirekt mit Index und Displacement
postinkrement/predekrement indirekt
absolut kurz/lang

Beschreibung: CMPI vergleicht einen Operanden aus dem Speicher oder in einem Datenregister mit einem unmittelbaren Wert. Die Ausführung entspricht der von CMP; CMPI bietet nur andere Adressierungsmöglichkeiten. Zur Reihenfolge der Operanden siehe CMP.

Besonderheit: CMP wird von den meisten Assemblern als CMPI behandelt, wenn die verwendeten Adressierungsarten es verlangen.

CMPM

compare memory
Vergleich zweier Operanden aus dem Speicher

Operation: (Q)-(Z)
(genauer (Q)+~(Z)+1)

Flags:
N : gesetzt, wenn Quelle < Ziel
Z : gesetzt, wenn Quelle = Ziel
V : gesetzt, wenn bei der internen Subtraktion der Zahlenbereich der vorzeichenbehafteten Zahlen verlassen wird
C : gesetzt, wenn Quelle ≤ Ziel
X : unberührt

Adressierungsarten:

QUELLE Adreßregister indirekt

ZIEL Adreßregister indirekt

Beschreibung: Die Inhalte zweier Operanden, die in der Adressierungsart "Adreßregister indirekt" angegeben werden müssen, werden miteinander verglichen und die Flags entsprechend gesetzt. CMPM ist nur eine Erweiterung von CMP hinsichtlich der Adressierungsarten. Weitere Informationen siehe unter CMP.

Besonderheit: CMP wird nach CMPM übersetzt, wenn die Adressierungsarten es verlangen.

multiply unsigned
Vorzeichenlose 16-Bit-Multiplikation

MULU

Operation: Dn <- Dn * (Q)

Flags:
N : gesetzt, wenn Ergebnis (32 Bit) < 0
Z : gesetzt, wenn Ergebnis = 0
V : gelöscht
C : gelöscht
X : unberührt

Adressierungsarten:

QUELLE
Datenregister direkt
Adreßregister indirekt
indirekt/PC-relativ mit Displacement
indirekt/PC-relativ mit Index und Displacement
postinkrement/predekrement indirekt
absolut kurz/lang
unmittelbar

ZIEL Datenregister direkt

Beschreibung: Zwei vorzeichenlose Wort-Operanden werden miteinander multipliziert. Das Ziel der Operation muß dabei ein Datenregister sein. Das 32 Bit lange Ergebnis wird im angegebenen Datenregister gespeichert. Beachten Sie, daß das Produkt von zwei 16-Bit-Operanden immer in 32 Bit Platz findet.

Besonderheit: Da der Prozessor für die Multiplikation ein relativ aufwendiges Microcode-Programm durchläuft, braucht die Multiplikation ein Mehrfaches einer normalen Befehlsausführungszeit.

MULS

multiply signed
vorzeichenbehaftete 16-Bit-Multiplikation

Operation: Dn <- Dn * (Q)

Flags:
N : gesetzt, wenn Ergebnis (32 Bit) < 0
Z : gesetzt, wenn Ergebnis = 0
V : gelöscht
C : gelöscht
X : unberührt

Adressierungsarten:

QUELLE
Datenregister direkt
Adreßregister indirekt
indirekt/PC-relativ mit Displacement
indirekt/PC-relativ mit Index und Displacement
postinkrement/predekrement indirekt
absolut kurz/lang
unmittelbar

ZIEL
Datenregister direkt

Beschreibung: Zwei Wort-Operanden werden unter Beachtung der Vorzeichen miteinander multipliziert. Das 32 Bit lange Ergebnis wird in dem angegebenen Datenregister abgelegt. Auch bei der Multiplikation mit Vorzeichen wird das Ergebnis immer in 32 Bit Platz finden. Beachten Sie, daß das Ergebnis von MULS nur dann mit dem Ergebnis von MULU übereinstimmt, wenn beide Operanden positiv sind.

Besonderheit: Auch hier eine deutlich längere Befehlsausführungszeit als üblich.

divide unsigned
Vorzeichenlose Division 32 Bit durch 16 Bit

DIVU

Operation: Dn <- Dn / (Q)

Flags:
N : gesetzt, wenn Ergebnis (16 Bit) < 0
Z : gesetzt, wenn Ergebnis = 0
V : gesetzt, wenn Ergebnis länger als 16 Bit
C : gelöscht
X : unberührt

Adressierungsarten:

QUELLE
Datenregister direkt
Adreßregister indirekt
indirekt/PC-relativ mit Displacement
indirekt/PC-relativ mit Index und Displacement
postinkrement/predekrement indirekt
absolut kurz/lang
unmittelbar

ZIEL Datenregister direkt

Beschreibung: Eine 32-Bit-Zahl in einem Datenregister (Dividend) wird durch den 16 Bits umfassenden Quelloperanden (Divisor) geteilt. Das Ergebnis wird im Ziel-Datenregister abgelegt, wobei die unteren 16 Bits das Ergebnis der Division und die oberen 16 Bits den Divisionsrest darstellen. Die Flags werden entsprechend dem Ergebnis, also den unteren 16 Bits, gesetzt. Der Divisionsrest entspricht "Dividend Modulo Divisor". Bei einer Division durch null wird eine Exception ausgelöst und zum Vektor ab $14 gesprungen. Da dieser normalerweise direkt auf ein RTE zeigt, wird das Programm nicht abgebrochen, aber der Prozessor befindet sich in einem etwas undefinierten Zustand. Das Ergebnis der Division findet leider nicht immer in 16 Bits Platz. Sollte es größer sein, so bricht der Prozessor die Division ab und setzt das V-Flag auf 1. Da der Prozessor keine Kommazahlen darstellen kann, wird das Ergebnis immer abgerundet.

DIVU

divide unsigned
Vorzeichenlose Division 32 Bit durch 16 Bit

Besonderheit: Beachten Sie, daß die Reihenfolge der Operanden genau umgekehrt ist, als man es von der mathematischen Schreibweise gewohnt ist:

```
DIVU a,b
```

berechnet b/a (b muß ein Datenregister sein).

DIVU benötigt ein Vielfaches der Ausführungszeit eines durchschnittlichen Befehls.

divide signed
Vorzeichenbehaftete Division 32 Bits durch 16 Bits

DIVS

Operation: Dn <- Dn / (Q)

Flags:
N : gesetzt, wenn Ergebnis (16 Bit) < 0
Z : gesetzt, wenn Ergebnis = 0
V : gesetzt, wenn Ergebnis länger als 16 Bit
C : gelöscht
X : unberührt

Adressierungsarten:

QUELLE
Datenregister direkt
Adreßregister indirekt
indirekt/PC-relativ mit Displacement
indirekt/PC-relativ mit Index und Displacement
postinkrement/predekrement indirekt
absolut kurz/lang
unmittelbar

ZIEL Datenregister direkt

Beschreibung: Eine 32-Bit-Zahl in einem Datenregister (Dividend) wird unter Beachtung der Vorzeichen durch den 16 Bits umfassenden Quelloperanden (Divisor) geteilt. Das Ergebnis wird im Ziel-Datenregister abgelegt, wobei die unteren 16 Bits das vorzeichenbehaftete Ergebnis der Division und die oberen 16 Bits den Divisionsrest darstellen. Die Flags werden entsprechend dem Ergebnis, also den unteren 16 Bits, gesetzt. Der Divisionsrest entspricht "Dividend Modulo Divisor". Bei einer Division durch null wird eine Exception ausgelöst und zum Vektor $14 gesprungen. Da dieser normalerweise direkt auf ein RTE zeigt, wird das Programm nicht abgebrochen, aber der Prozessor befindet sich in einem etwas undefinierten Zustand. Das Ergebnis der vorzeichenbehafteten Division findet leider nicht immer in 16 Bits Platz. Sollte es größer sein, so bricht der Prozessor die Division ab und setzt das V-Flag auf 1. Da der Prozessor keine Kommazahlen darstellen kann, wird das

DIVS

divide signed
Vorzeichenbehaftete Division 32 Bits durch 16 Bits

Ergebnis immer abgerundet. Der Rest hat immer das Vorzeichen des Quelloperanden, also des Wertes, durch den geteilt wird.

Besonderheit: Beachten Sie, daß die Reihenfolge der Operanden genau umgekehrt ist, als man es von der mathematischen Schreibweise gewohnt ist:

```
DIVU a,b
```

berechnet b/a (b muß ein Datenregister sein).

DIVS benötigt ein Vielfaches der Ausführungszeit eines durchschnittlichen Befehls.

Logische Befehle

Der MC68000 beherscht die drei wichtigsten logischen Operationen mit zwei Operanden: ODER, UND und EXKLUSIV-ODER. Alle Operationen werden mit allen Bits der beiden Operanden durchgeführt. Mit den Abwandlungen ergeben sich folgende Befehle:

OR
ORI
AND
ANDI
EOR
EORI

OR

bitweises ODER

Operation: (Z) <- (Z) (Q)

Flags:
N : gesetzt, wenn Ergebnis < 0
Z : gesetzt, wenn Ergebnis = 0
V : gelöscht
C : gelöscht
X : unberührt

Adressierungsarten:

QUELLE
Datenregister direkt
Adreßregister indirekt
indirekt/PC-relativ mit Displacement
indirekt/PC-relativ mit Index und Displacement
postinkrement/predekrement indirekt
absolut kurz/lang
unmittelbar

ZIEL
Datenregister direkt
indirekt mit Displacement
indirekt mit Index und Displacement
postinkrement/predekrement indirekt
absolut kurz/lang

Beschreibung: Jedes Bit des Zieloperanden wird mit dem entsprechenden Bit des Quelloperanden ODER-verknüpft. Das heißt, daß ein Bit im Zieloperanden genau dann gesetzt wird, wenn dieses Bit vorher entweder im Zieloperanden oder im Quelloperanden gesetzt war oder in beiden. Die Flags werden entsprechend dem Ergebnis gesetzt. Der OR-Befehl wird selten zum Berechnen von Wahrheitswerten benutzt; vielmehr dient er dazu, ausgewählte Bits im Zieloperanden zu setzen. Jedes Bit, das im Quelloperanden gesetzt ist, wird nach der Ausführung auch im Zieloperanden gesetzt sein; die anderen Bits des Zieloperanden bleiben erhalten. Bei der Adressierung ist zu beachten, daß entweder die Quelle oder das Ziel ein Datenregister sein muß.

or immediate
bitweises ODER mit
einer unmittelbaren Quelle

ORI (ORI to SR)

Operation: (Z) <- (Z) (Q)

Flags:
N : gesetzt, wenn Ergebnis < 0
Z : gesetzt, wenn Ergebnis = 0
V : gelöscht
C : gelöscht
X : unberührt

Adressierungsarten:

QUELLE unmittelbar

ZIEL
Datenregister direkt
Adreßregister indirekt
indirekt mit Displacement
indirekt mit Index und Displacement
postinkrement/predekrement indirekt
absolut kurz/lang
Statusregister

Beschreibung: Jedes Bit des Zieloperanden wird mit dem entsprechenden Bit des unmittelbar angegebenen Quelloperanden ODER-verknüpft. Das heißt, daß ein Bit im Zieloperanden genau dann gesetzt wird, wenn dieses Bit vorher entweder im Zieloperanden oder im Quelloperanden gesetzt war oder in beiden. ORI ist eine Erweiterung des OR-Befehls hinsichtlich der Adressierungsarten. ORI bietet die Besonderheit, daß als Ziel auch das Statusregister (SR) verwendet werden kann. Im Byte-Modus können ausgewählte Systemflags gesetzt werden. Im Wort-Modus kann das Systembyte verändert werden, weshalb dieser Befehl nur im Supervisormodus verwendet werden darf.

Besonderheit: ORI.W to SR ist ein privilegierter Befehl. Die meisten Assembler übersetzen OR nach ORI, wenn die verwendete Adressierungsart es verlangt.

AND

bitweises UND

Operation: (Z) <- (Z) (Q)

Flags:
N : gesetzt, wenn Ergebnis < 0
Z : gesetzt, wenn Ergebnis = 0
V : gelöscht
C : gelöscht
X : unberührt

Adressierungsarten:

QUELLE
Datenregister direkt
Adreßregister indirekt
indirekt/PC-relativ mit Displacement
indirekt/PC-relativ mit Index und Displacement
postinkrement/predekrement indirekt
absolut kurz/lang
unmittelbar

ZIEL
Datenregister direkt
indirekt mit Displacement
indirekt mit Index und Displacement
postinkrement/predekrement indirekt
absolut kurz/lang

Beschreibung: Jedes Bit des Zieloperanden wird mit dem entsprechenden Bit des Quelloperanden UND-verknüpft. Das heißt, daß ein Bit im Zieloperanden genau dann gesetzt wird, wenn dieses Bit vorher im Zieloperanden und im Quelloperanden gesetzt war. Die Flags werden entsprechend dem Ergebnis gesetzt. Der AND-Befehl wird kaum zum Berechnen von Wahrheitswerten benutzt; seine Aufgabe ist es vielmehr, ausgewählte Bits im Zieloperanden zu löschen. Jedes Bit, das im Quelloperanden gelöscht ist, wird nach der Ausführung auch im Zieloperanden gelöscht sein; die anderen Bits des Zieloperanden bleiben erhalten. Bei der Adressierung ist zu beachten, daß entweder die Quelle oder das Ziel ein Datenregister sein muß.

and immediate
bitweises ODER mit
einer unmittelbaren Quelle

ANDI
(ANDI TO SR)

Operation: (Z) <- (Z) (Q)

Flags:
N : gesetzt, wenn Ergebnis < 0
Z : gesetzt, wenn Ergebnis = 0
V : gelöscht
C : gelöscht
X : unberührt

Adressierungsarten:

QUELLE unmittelbar

ZIEL
Datenregister direkt
Adreßregister indirekt
indirekt mit Displacement
indirekt mit Index und Displacement
postinkrement/predekrement indirekt
absolut kurz/lang
Statusregister

Beschreibung: Jedes Bit des Zieloperanden wird mit dem entsprechenden Bit des unmittelbar angegebenen Quelloperanden UND-verknüpft. Das heißt, daß ein Bit im Zieloperanden genau dann gesetzt wird, wenn dieses Bit vorher im Zieloperanden und im Quelloperanden gesetzt war. ANDI ist eine Erweiterung des OR-Befehls hinsichtlich der Adressierungsarten. ANDI bietet die Möglichkeit, als Ziel das Statusregister (SR) zu adressieren. Im Byte-Modus können so ausgewählte Systemflags gelöscht werden. Im Wort-Modus kann das Systembyte verändert werden, weshalb dieser Befehl nur im Supervisormodus verwendet werden darf.

Besonderheit: ANDI.W to SR ist ein privilegierter Befehl. Die meisten Assembler übersetzen AND nach ANDI, wenn die verwendete Adressierungsart es verlangt.

EOR

exclusive or
bitweises EXKLUSIV-ODER

Operation: (Z) <- (Z) EOR (Q)

Flags:
N : gesetzt, wenn Ergebnis < 0
Z : gesetzt, wenn Ergebnis = 0
V : gelöscht
C : gelöscht
X : unberührt

Adressierungsarten:

QUELLE
Datenregister direkt
Adreßregister indirekt
indirekt/PC-relativ mit Displacement
indirekt/PC-relativ mit Index und Displacement
postinkrement/predekrement indirekt
absolut kurz/lang
unmittelbar

ZIEL
Datenregister direkt
indirekt mit Displacement
indirekt mit Index und Displacement
postinkrement/predekrement indirekt
absolut kurz/lang

Beschreibung: Jedes Bit des Zieloperanden wird mit dem entsprechenden Bit des Quelloperanden EXKLUSIV-ODER verknüpft. Das heißt, daß ein Bit im Zieloperanden genau dann gesetzt wird, wenn dieses Bit vorher entweder im Zieloperanden oder im Quelloperanden gesetzt war, aber nicht in beiden. Die Flags werden entsprechend dem Ergebnis gesetzt. EOR dient selten zum Berechnen von Wahrheitswerten. Meist wird es dazu verwendet, ausgewählte Bits im Zieloperanden zu invertieren. Jedes Bit, das im Quelloperanden gesetzt ist, führt dazu, daß das entsprechende Bit des Zieloperanden invertiert wird. Die anderen Bits des Zieloperanden bleiben erhalten. Bei der Adressierung ist zu beachten, daß entweder die Quelle oder das Ziel ein Datenregister sein muß.

exclusive or immediate
bitweises EXKLUSIV-ODER
mit einer unmittelbaren Quelle

EORI

Operation: (Z) <- (Z) EOR (Q)

Flags:
N : gesetzt, wenn Ergebnis < 0
Z : gesetzt, wenn Ergebnis = 0
V : gelöscht
C : gelöscht
X : unberührt

Adressierungsarten:

QUELLE unmittelbar

ZIEL Datenregister direkt
Adreßregister indirekt
indirekt mit Displacement
indirekt mit Index und Displacement
postinkrement/predekrement indirekt
absolut kurz/lang

Beschreibung: Jedes Bit des Zieloperanden wird mit dem entsprechenden Bit des unmittelbar angegebenen Quelloperanden EXKLUSIV-ODER-verknüpft. Das heißt, daß ein Bit im Zieloperanden genau dann gesetzt wird, wenn dieses Bit vorher entweder im Zieloperanden oder im Quelloperanden gesetzt war, aber nicht in beiden. EORI ist eine Abwandlung des EOR-Befehls hinsichtlich der Adressierungsarten.

Besonderheit: Die meisten Assembler übersetzen EOR nach EORI, wenn die verwendete Adressierungsart es verlangt.

Bit-Befehle

Der MC68000 bietet Befehle, mit denen auf einzelne Bits des Zieloperanden zugegriffen werden kann:

BSET
BCLR
BCHG
BTST

Da diese Bit-Befehle über große Gemeinsamkeiten verfügen, hier zunächst einmal die Adressierungsarten für alle Bit-Befehle:

QUELLE	Datenregister direkt unmittelbar
ZIEL	Datenregister direkt Adreßregister indirekt indirekt mit Displacement indirekt mit Index und Displacement postindecrement/predecrement indirekt absolut kurz/lang

Für die vier Befehle gilt folgendes: Der Inhalt des Quelloperanden bestimmt das Bit des Zieloperanden, das bearbeitet werden soll. Es gibt keine Varianten mit verschiedenen Verarbeitungsbreiten; vielmehr ist die Länge eines Operanden im Speicher auf 8 Bit festgelegt, die eines Datenregisters auf 32 Bit. Wenn also das Ziel ein Datenregister ist, werden nur die letzten fünf Bits der Quelle beachtet, bei Speicherzellen nur die letzten drei. Die übrigen Bits der Quelle werden ignoriert. Wie üblich ist Bit 0 das niederwertigste, Bit 7 bzw. 31 das höchstwertige des Operanden.

bit set
Bit testen und auf 1 setzen

BSET

Operation: (Z) <- (Z) mit Bit n gesetzt
n=(Q)<4:0> oder n=(Q)<2:0>

Flags:
N : unberührt
Z : gesetzt, wenn Bit = 0
V : unberührt
C : unberührt
X : unberührt

Beschreibung: Zunächst wird das durch den Quelloperanden angegebene Bit im Zieloperanden getestet; ist es null, so wird das Z-Flag gesetzt, andernfalls gelöscht. Dann wird das Bit im Zieloperanden gesetzt. Wenn n wie oben angegeben der Wert der ersten 3 oder 5 Bits des Quelloperanden ist, entspricht die Operation einer ODER-Verknüpfung mit 2^n. Alle anderen Flags außer Z bleiben unverändert.

BCLR

bit clear
Bit testen und auf 0 setzen

Operation: (Z) <- (Z) mit Bit n gelöscht
n=(Q)<4:0> oder n=(Q)<2:0>

Flags:
N : unberührt
Z : gesetzt, wenn Bit = 0
V : unberührt
C : unberührt
X : unberührt

Beschreibung: Zunächst wird das durch den Quelloperanden angegebene Bit im Zieloperanden getestet; ist es null, so wird das Z-Flag gesetzt, andernfalls gelöscht. Erst nach dieser Abfrage wird das Bit im Zieloperanden gelöscht. Wenn n wie oben angegeben der Wert der ersten 3 oder 5 Bits des Quelloperanden ist, entspricht die Operation einer UND-Verknüpfung mit minus 2^n. Alle anderen Flags außer Z bleiben unverändert.

bit change
Bit testen und invertieren

BCHG

Operation: (Z) <- (Z) mit Bit n invertiert
n=(Q)<4:0> oder n=(Q)<2:0>

Flags:
N : unberührt
Z : gesetzt, wenn Bit = 0
V : unberührt
C : unberührt
X : unberührt

Beschreibung: Zunächst wird das durch den Quelloperanden angegebene Bit im Zieloperanden getestet; ist es null, so wird das Z-Flag gesetzt, andernfalls gelöscht. Dann wird das Bit im Zieloperanden invertiert. Wenn n wie oben angegeben der Wert der ersten 3 oder 5 Bits des Quelloperanden ist, entspricht die Operation einer EXKLUSIV-ODER-Verknüpfung mit 2^n. Alle anderen Flags außer Z bleiben unverändert.

BTST

bit test
Bit testen

Operation: keine

Flags:
N : unberührt
Z : gesetzt, wenn Bit = 0
V : unberührt
C : unberührt
X : unberührt

Beschreibung: Das durch den Quelloperanden angegebene Bit im Zieloperanden wird getestet; ist es null, so wird das Z-Flag gesetzt, andernfalls gelöscht. Der Wert des Bits wird also invers ins Z-Flag geschrieben. Alle anderen Flags bleiben unverändert.

Bedingte Befehle

Die bedingten Befehle führen zu unterschiedlichen Resultaten, je nachdem, welche Bedingungsflags gerade gesetzt sind. Es gibt 16 sogenannte Befehlscodes (condition codes), die jeweils für eine abprüfbare Bedingung stehen. Zunächst eine Liste der Bedingungscodes und ihrer Bedeutung:

T	true	T ist immer wahr
F	false	F ist niemals wahr
HI	higher	A > B (ohne Vorzeichen)
LS	lower or same	A <= B (ohne Vorzeichen)
CC	carry clear	C=0 oder A ≥ B (ohne Vorzeichen)
CS	carry set	C=1 oder A < B (ohne Vorzeichen)
NE	not equal	Z=0 oder A <> B
EQ	equal	Z=1 oder A = B
VC	overflow clear	V=0
VS	overflow set	V=1
PL	plus	N=0
MI	minus	N=1
GE	greater or equal	A ≥ B (mit Vorzeichen)
LT	less than	A < B (mit Vorzeichen)
GT	greater than	A > B (mit Vorzeichen)
LE	less or equal	A ≤ B (mit Vorzeichen)

Dabei beziehen sich die Relationen zwischen A und B auf einen vorangegangenen Vergleich

```
CMP B,A
```

Alle Relationen werden auf den Zustand eines oder mehrerer Flags zurückgeführt; manchmal ist das so offensichtlich wie bei CC, wo das Carry-Flag gleichzeitig für die Bedingung "A < B" steht, in anderen Fällen handelt es sich um recht komplizierte logische Verknüpfungen mehrerer Flags.

Weitere Informationen über bedingte Befehle und die Bedeutung der Bedingungscodes finden Sie in Kapitel 2, Abschnitt "Bedingte Verzweigungen". Die Systemflags werden von keinem der bedingten Befehle beeinflußt. Bei den folgenden bedingten Befehlen steht "cc" für condition code (Bedingungscode) und kann durch jedes der 16 oben genannten Kürzel ersetzt werden. Die Befehle sind:

Bcc
DBcc
Scc

Bcc

branch if...
bedingter relativer Sprung

Operation: Wenn Bedingung: PC <- PC + Adreßdistanz

Flags:
N : unberührt
Z : unberührt
V : unberührt
C : unberührt
X : unberührt

Adressierungsarten:

8 Bit PC-relativ
16 Bit PC-relativ

Beschreibung: Der Prozessor testet, ob eine bestimmte Bedingung erfüllt ist. Trifft die Bedingung nicht zu, so fährt er mit der nächsten Anweisung fort. Ist sie jedoch erfüllt, so verzweigt der Prozessor zur angegebenen Adresse. Gewöhnlich wird die Adresse dem Assembler in Form eines Labels überreicht, der daraus den Abstand der Adresse des Branch-Befehls und der Zieladresse errechnet. So sind die Branch-Befehle automatisch relozierbar. Der Adreßabstand kann wahlweise eine Länge von 8 Bit haben, womit ein Bereich von –126 bis +129 von der Adresse des Branch-Befehls gerechnet angesprochen werden kann, oder 16 Bit lang sein, wobei sich ein Bereich von –32766 bis +32769 ergibt. Die Standardform ist der 16-Bit-Abstand; die 8-Bit-Form wird durch das Anhängsel ".S" am Befehlscode gekennzeichnet. Manche Assembler generieren allerdings automatisch die kurze Form, wann immer es möglich ist, da diese nicht nur kürzer, sondern auch schneller ist. Für BT, branch if true, die bedingungslose Verzweigung, die man eigentlich unter den Sprungbefehlen einordnen müßte, kann man auch BRA für "branch always" schreiben. BF wird niemals verzweigen und ist daher vergleichbar mit NOP. Der Befehl BSR, der manchmal auch unter den bedingten Verzweigungen aufgeführt wird, ist hier unter den Sprungbefehlen beschrieben.

decrement and branch until ...
Zählschleife mit zusätzlicher Abbruchbedingung

DBcc

Operation: Wenn Bedingung: nächster Befehl
sonst: Dn <- Dn - 1
Wenn Dn=-1: nächster Befehl
sonst: Verzweigung

Flags:
N : unberührt
Z : unberührt
V : unberührt
C : unberührt
X : unberührt

Adressierungsarten:

PC-relativ 16 Bit

Beschreibung: Als Operanden zu DBcc werden in dieser Reihenfolge ein Datenregister und eine relative Adresse angegeben. Diese recht komplexe Anweisung ist dafür gedacht, an das Ende einer Zählschleife gestellt zu werden, wobei das angegebene Datenregister den Zähler enthält.

Zusätzlich kann die Schleife noch durch eine bestimmte Bedingung verlassen werden. Zunächst wird die Bedingung abgeprüft. Ist sie wahr, so geschieht nichts weiter, und es wird sofort mit dem nächsten Befehl fortgefahren. Andernfalls wird der Inhalt des angegebenen Datenregisters in Wortbreite um eins verringert. Der neue Inhalt des Datenregisters wird überprüft: Ist er –1, so wird die Schleife als beendet betrachtet, und der Prozessor fährt mit dem folgenden Befehl fort. Andernfalls muß die Schleife ein weiteres Mal durchlaufen werden. Es wird zur relativ angegebenen Adresse verzweigt.

Bei DBcc wird die Sprungadresse relativ als 16-Bit-Offset angegeben. Allerdings kann DBcc nur rückwärts, also zu einer niedrigeren Adresse verzweigen. Dadurch ergibt sich ein Adreßbereich bis zu –65534 Bytes. Der am häufigsten ver-

DBcc

decrement and branch until ...
Zählschleife mit zusätzlicher Abbruchbedingung

wendete Befehl dieser Gruppe ist DBF, denn dieser macht von der Möglichkeit, eine besondere Bedingung als außergewöhnliche Abbruchbedingung der Schleife zu wählen, keinen Gebrauch; Er verzweigt also nur, solange der Zähler im Datenregister nicht zu –1 wird. Statt DBF kann oft auch DBRA (decrement and branch) verwendet werden.

Zwei Dinge gibt es bei der Verwendung von DBcc zu beachten:

- DBcc verhält sich genau entgegengesetzt zu Bcc: Bei DBcc kann nur verzweigt werden, wenn die Bedingung nicht wahr ist.

- Der Zähler muß eins niedriger als die Zahl der gewünschten Schleifendurchläufe gewählt werden, da erst bei –1 abgebrochen wird. Sinnvoll ist diese Festlegung dann, wenn der Zähler in der Schleife etwa als Index verwendet wird und alle Werte einschließlich der Null durchlaufen soll.

set if ...
Setzen eines Programmflags nach einer Bedingung

Scc

Operation: Wenn Bedingung: (Z) <- 11111111 (binär)
sonst: (Z) <- 00000000 (binär)

Flags:
N : unberührt
Z : unberührt
V : unberührt
C : unberührt
X : unberührt

Adressierungsarten:

ZIEL
Datenregister direkt
Adreßregister indirekt
Indirekt mit Displacement
Indirekt mit Index und Displacement
Postinkrement/Predekrement indirekt
absolut kurz/lang

Beschreibung: Dieser Befehl kann nur in Byte-Breite verwendet werden. Wenn die abgeprüfte Bedingung erfüllt ist, wird das adressierte Byte mit binären Einsen gefüllt, also auf dezimal 255 gesetzt. Trifft die Bedingung nicht zu, so wird der Operand auf null gesetzt. Dieser Befehl dient dazu, den aktuellen Wahrheitswert einer Bedingung zu speichern, damit er zu einem späteren Zeitpunkt abgefragt werden kann.

Sprungbefehle

Alle Sprungbefehle beeinflussen in irgendeiner Form den Befehlszähler. Es handelt sich dabei um direkte Sprünge, Aufruf von Unterprogrammen und Rückkehr aus denselben:

JMP
JSR
BSR
TRAP
RTS
RTR
RTE

Keiner der Sprungbefehle beeinflußt die Flags.

jump
Direkter Sprung

JMP

Operation: PC <- Z

Flags:
N : unberührt
Z : unberührt
V : unberührt
C : unberührt
X : unberührt

Adressierungsarten:

ZIEL
Adreßregister indirekt
Indirekt/PC-relativ mit Displacement
Indirekt/PC-relativ mit Index und Displacement
absolut kurz/lang

Beschreibung: Es wird ein Programmsprung zur angegebenen Adresse ausgeführt. Bei der PC-relativen Adressierungsart bleibt der Programmsprung relozierbar wie bei den bedingten Verzweigungen. Bei den anderen Adressierungsarten finden jedoch absolute Sprünge statt. Als eine Alternative siehe auch BT (BRA).

JSR

Jump to subroutine
Unterprogrammaufruf

Operation: -(SP) <- PC
PC <- Z

Flags: N : unberührt
Z : unberührt
V : unberührt
C : unberührt
X : unberührt

Adressierungsarten:

ZIEL Adreßregister indirekt
Indirekt/PC-relativ mit Displacement
Indirekt/PC-relativ mit Index und Displacement
absolut kurz/lang

Beschreibung: Der Prozessor legt den Inhalt des PC auf dem Stack ab und verzweigt zur angegebenen Adresse. Somit wird ein Unterprogramm aufgerufen, das mit RTS die Kontrolle dem aufrufenden Programm zurückgeben kann. Wie bei JMP kann die Adresse entweder absolut oder PC-relativ angegeben werden. JSR ist in den Adressierungsarten flexibler als die Alternative BSR, aber langsamer.

Branch to subroutine
relativer Unterprogrammaufruf

BSR

Operation: -(SP) <- PC
PC <- PC+Adreßdistanz

Flags: N : unberührt
Z : unberührt
V : unberührt
C : unberührt
X : unberührt

Adressierungsarten:

PC-relativ 8 Bit
PC-relativ 16 Bit

Beschreibung: Der Prozessor legt den aktuellen Stand des Befehlszählers auf dem Stack ab und verzweigt zur angegebenen Adresse. Die Adresse wird wie bei den bedingten Verzweigungsbefehlen angegeben als Differenz der Adresse des BSR und der Zieladresse. Es können entweder 8 Bit oder 16 Bit Adreßdistanz angegeben werden, wobei sich ein Bereich von –126 bis +129 oder –32766 bis +32769 ergibt. Standard ist die 16-Bit-Variante; die kürzere und schnellere 8-Bit-Form wird durch ein angehängtes ".S" kenntlich gemacht. Natürlich können in dem aufgerufenen Unterprogramm die Flags verändert werden. Der Unterschied zu JSR besteht darin, daß BSR nur PC-relativ verzweigen kann.

TRAP

Programmgesteuerte Exception

Operation: -(SSP) <- SR (16 Bit)
-(SSP) <- PC
PC <- Trapvektor n (n=0...15)

Flags: N : unberührt
Z : unberührt
V : unberührt
C : unberührt
X : unberührt

Adressierungsarten:

Angabe einer unmittelbaren Vektornummer 0 bis 15

Beschreibung: Zunächst legt der Prozessor den SR und PC in dieser Reihenfolge auf dem Stack ab. Anschließend wird der unmittelbar angegebene Wert von 0 bis 15 als Index in einer Vektortabelle benutzt und ein Programmsprung zur dort angegebenen Adresse ausgeführt. Dabei geht der Prozessor automatisch in den Supervisor-Modus. Der TRAP-Befehl ist für den Aufruf von Betriebssystemroutinen gedacht: Er bietet eine genormte Einsprungstelle. Folgende Trap-Vektoren sind beim ATARI ST belegt:

TRAP #1	GEMDOS
TRAP #2	GEM (VDI und AES)
TRAP #13	BIOS
TRAP #14	XBIOS

Alle anderen TRAP-Vektoren können vom Programmierer frei verwendet werden. Die Trap-Vektoren belegen ab der Adresse $80 16 Langworte. Sie können nur im Supervisor-Modus geändert werden.

return from subroutine
Rückkehr von einem Unterprogramm

RTS

Operation: PC <- (SP)+

Flags:
N : unberührt
Z : unberührt
V : unberührt
C : unberührt
X : unberührt

Adressierungsarten:

keine

Beschreibung: Der Prozessor holt den alten Wert des PC (Langwort) vom Stack und springt so zu der aufrufenden Adresse des Unterprogramms zurück. Voraussetzung für eine korrekte Ausführung ist natürlich, daß der Wert des Stackpointers im Unterprogramm nicht verändert wird. Im Gegensatz zu RTR und RTE werden die Flags nicht verändert.

RTR

return from subroutine and restore CCR
Rückkehr von einem Unterprogramm mit Wiederherstellung des CCR

Operation: CCR <- (SP)+
PC <- (SP)+

Flags: N : Bit 3 des vom Stack geholten Wortes
Z : Bit 2 des vom Stack geholten Wortes
V : Bit 1 des vom Stack geholten Wortes
C : Bit 0 des vom Stack geholten Wortes
X : Bit 4 des vom Stack geholten Wortes

Adressierungsarten:

keine

Beschreibung: Der Prozessor holt zunächst einen 16-Bit-Wert vom Stack und schreibt die unteren 8 Bit in das CCR. Danach wird der alte Stand des PC vom Stack wiederhergestellt. Beachten Sie, daß der Prozessor bei einem Unterprogrammaufruf mit JSR oder BSR die Flags nicht automatisch auf dem Stack ablegt. Wenn Sie Ihr Unterprogramm mit RTR verlassen wollen, sollte der erste Befehl des Unterprogramms folgender sein:

```
MOVE SR,-(SP)
```

return from exception
Rückkehr von Exception
mit Wiederherstellung des SR

RTE

Operation: SR <- (SP)+
PC <- (SP)+

Flags: N : Bit 3 des vom Stack geholten Wortes
Z : Bit 2 des vom Stack geholten Wortes
V : Bit 1 des vom Stack geholten Wortes
C : Bit 0 des vom Stack geholten Wortes
X : Bit 4 des vom Stack geholten Wortes

Adressierungsarten:

keine

Beschreibung: Der Prozessor holt zunächst einen 16-Bit-Wert vom Stack und schreibt ihn in das SR. Danach wird der alte Stand des PC vom Stack wiederhergestellt. RTE ist für die Rückkehr von einer Exception-Routine gedacht. Da das gesamte SR wieder auf den Stand vor der Exception gestellt wird, wird auch sichergestellt, daß der Prozessor in den gleichen Zustand – Supervisor- oder Usermodus – zurückkehrt, in dem er sich vor der Exception befand. Da der Befehl das Systembyte verändert, darf er nur im Supervisormodus ausgeführt werden. Exceptions werden jedoch ohnehin im Supervisormodus ausgeführt.

Besonderheit: Privilegierter Befehl

Sonstige Befehle

In dieser Kategorie sind jene Befehle aufgeführt, die sich in die anderen Kategorien nicht einordnen lassen. Es handelt sich dabei einerseits um recht komplexe Befehle, die die Implementierung von Compilern begünstigen, andererseits um sehr hardwarenahe Befehle:

LINK
UNLK
TRAPV
CHK

LEA
PEA

STOP
RESET

NOP

Die Befehle LINK und UNLK sind recht schwer zu verstehen; sie sind allerdings für den Assemblerprogrammierer auch kaum von Bedeutung, da sie in erster Linie für den Einsatz durch Compiler ausgelegt sind.

Stackbereich vorübergehend reservieren

LINK

Operation:
-(SP) <- An
An <- SP
SP <- SP + (Z)

Flags:
N : unberührt
Z : unberührt
V : unberührt
C : unberührt
X : unberührt

Adressierungsarten:

QUELLE Adreßregister direkt

ZIEL unmittelbar 16 Bit

Beschreibung: Dieser Befehl ist dazu gedacht, am unteren Ende des Stacks einen Raum für lokale Variablen am Anfang eines Unterprogramms zu reservieren.

Als Operanden werden unmittelbar die Länge des Bereiches angegeben, um den der Stack erweitert werden soll, und ein Adreßregister, der sogenannte "frame pointer".

Zunächst wird der Inhalt des Adreßregisters auf dem Stack gesichert. Als nächstes wird der Inhalt des Stackpointers (nach dem Ablegen des Registerinhalts) in das Adreßregister übertragen. Nun folgt der entscheidende Schritt: Der angegebene Wert wird vorzeichenrichtig zum Stackpointer addiert und das Ergebnis wieder im Stackpointer abgelegt. Wenn der Programmierer einen negativen Wert angegeben hat, wird der Stackpointer entsprechend nach unten verschoben und so Speicherplatz für lokale Variablen angelegt.

Gewöhnlich wird auf diese Variablen mit "-n(SP)" zugegriffen. Damit das Unterprogramm auch auf etwa vorher auf dem

LINK

Stackbereich vorübergehend reservieren

Stack abgelegte Argumente zurückgreifen kann, bleibt der alte Wert des SP im "frame pointer" erhalten. Und da Unterprogramme sich ja auch verschachtelt aufrufen können, muß der alte Wert des "frame pointers" auf dem Stack abgelegt werden. So kann in allen Unterprogrammen das gleiche Register als "linkage pointer" dienen.

Ein Unterprogramm, das mit LINK Platz reserviert, sollte diesen vor der Rückkehr ins aufrufende Programm mit UNLK wieder freigeben.

unlink
mit LINK reservierten Stackbereich freigeben

UNLK

Operation: SP <- An
An <- (SP)+

Flags: N : unberührt
Z : unberührt
V : unberührt
C : unberührt
X : unberührt

Adressierungsarten:

ZIEL Adreßregister direkt

Beschreibung: Dieser Befehl ist das Gegenstück zur LINK-Anweisung und dient dazu, mit LINK auf dem Stack reservieren Platz wieder freizugeben. Im allgemeinen findet das am Ende des Unterprogramms, unmittelbar vor der RTS-Anweisung statt. Zunächst wird der Stackpointer mit dem Inhalt des angegebenen Adreßregisters geladen. Dann wird der Inhalt des Adreßregisters vom Stack geholt. Wenn bei UNLK das gleiche Adreßregister wie bei LINK angegeben wird und dieses zwischendurch nicht verändert wird, dann wird der alte Wert des Stackpointers und des Adreßregisters wiederhergestellt: Alles ist wieder, wie es vor dem LINK-Befehl war.

TRAPV

trap on overflow
Auslösung einer Exception, wenn ein Überlauf auftrat

Operation: Wenn V=1: TRAPV-Exception

Flags:
N : unberührt
Z : unberührt
V : unberührt
C : unberührt
X : unberührt

Adressierungsarten:

keine

Beschreibung: Wenn das V-Flag gesetzt ist, führt der Prozessor eine Exception aus. Dazu werden SR und PC auf den Stack gesichert, und es wird durch den Vektor in $1C gesprungen. Sofern die Exception nicht abgefangen wird, werden dadurch 7 Bomben auf den Bildschirm gebracht. Wenn das V-flag jedoch nicht gesetzt ist, geschieht nichts weiter, und der Prozessor fährt mit dem nächsten Befehl fort. Die Flags werden nicht beeinflußt. Der Befehl wird praktisch nur von Compilern verwendet, um einen Überlauf bei Integer-Berechnungen aufzuspüren.

check data register against boundaries
Überprüfen, ob der Inhalt eines Datenregisters in einem gültigen Bereich liegt

CHK

Operation: Wenn Dn > (Q) (vorzeichenlos): CHK-Exception

Flags:
N : gesetzt, wenn Dn < 0
Z : undefiniert
V : undefiniert
C : undefiniert
X : unberührt

Adressierungsarten:

QUELLE
Datenregister direkt
Adreßregister indirekt
indirekt/PC-relativ mit Displacement
indirekt/PC-relativ mit Index und Displacement
postinkrement/predekrement indirekt
absolut kurz/lang
unmittelbar

ZIEL Datenregister direkt

Beschreibung: Das angegebene Datenregister wird vorzeichenlos und in Wortbreite mit dem Quelloperanden verglichen. Ist der Inhalt des Datenregisters größer, so wird eine Exception ausgeführt. Dazu legt der Prozessor SR und PC auf dem Stack ab und ruft die Routine auf, deren Adresse im Exception-Vektor ab $18 steht. Normalerweise sind 6 Bomben das Ergebnis. Ist der Inhalt des Datenregisters jedoch im zulässigen Bereich, so wird mit der Programmausführung normal fortgefahren. Es gilt zu beachten, daß der Zustand von C, Z und V nach der Ausführung von CHK nicht festgelegt ist. Diese Anweisung wird eigentlich nur von Compilern eingesetzt, etwa um zu überprüfen, ob ein Index in ein Feld im zulässigen Bereich liegt.

LEA

load effective address
Lade effektive Adresse in ein Adreßregister

Operation: An <- Q

Flags:
N : unberührt
Z : unberührt
V : unberührt
C : unberührt
X : unberührt

Adressierungsarten:

QUELLE Adreßregister direkt
indirekt/PC-relativ mit Displacement
indirekt/PC-relativ mit Index und Displacement
absolut kurz/lang

ZIEL Adreßregister direkt

Beschreibung: Der Prozessor berechnet die Quelladresse und lädt sie in ein Adreßregister. Beachten Sie, daß die Adresse geladen wird, nicht der Inhalt der Adresse. Flags werden nicht beeinflußt. Diese Anweisung ist hauptsächlich dazu gedacht, jene Lücken zu füllen, die sich durch beschränkte Adressierungsarten bei den meisten Befehlen auftun. So ist es etwa meist nicht möglich, den Zieloperanden einer arithmetischen oder logischen Operation PC-relativ zu adressieren. Hier kann man sich mit LEA behelfen: Man lädt erst mittels LEA die gewünschte Adresse in ein Adreßregister und verwendet dann die Adressierungsart "Adreßregister indirekt", die bei fast allen Befehlen erlaubt ist. Dazu ein Beispiel:

```
LEA $5000(PC),A0 * Adresse des Zieloperanden
ADD D0,(A0)      * D0 zum Ziel addieren
```

Als zusätzliches Bonbon erlaubt LEA in einigen Fällen, mit Adreßregistern zu rechnen:

```
LEA 0(A0,A1),A2
```

load effective address
Lade effektive Adresse in ein Adreßregister

LEA

berechnet die Summe aus A0 und A1 und legt sie in A2 ab. Beliebt ist es auch, die Adressierungsart "indirekt mit Displacement" zum Rechnen zu verwenden:

```
LEA $100(A0),A0
```

Dieser Befehl ist etwas schneller als das entsprechende

```
ADDA #$100,A0
```

PEA

push effective address onto stack
effektive Adresse auf dem Stack ablegen

Operation: -(SP) <- Q

Flags:
N : unberührt
Z : unberührt
V : unberührt
C : unberührt
X : unberührt

Adressierungsarten:

QUELLE
Adreßregister direkt
indirekt/PC-relativ mit Displacement
indirekt/PC-relativ mit Index und Displacement
absolut kurz/lang

Beschreibung: Der Prozessor berechnet die Quelladresse (richtig – hier gibt es nur einen Quelloperanden; das Ziel ist der Stack) und legt sie als Langwort auf dem Stack ab. Wie bei LEA gilt, daß die Adresse des Operanden abgelegt wird, nicht der Inhalt der Adresse. Nützlich ist diese Anweisung dann, wenn die Adressen von Datenobjekten als Argumente an ein Unterprogramm übergeben werden sollen.

stop processor
Prozessor anhalten und auf Interrupt warten

STOP

Operation: SR <- (src)
HALT-Zustand

Flags: N : Bit 3 des Quelloperanden
Z : Bit 2 des Quelloperanden
V : Bit 1 des Quelloperanden
C : Bit 0 des Quelloperanden
X : Bit 4 des Quelloperanden

Adressierungsarten:

QUELLE unmittelbar 16 Bit

Beschreibung: Zuerst wird der unmittelbar angegebene Wort-Operand ins SR übertragen. Dann geht der Prozessor in den HALT-Zustand. Das heißt, daß die CPU so lange inaktiv ist, bis ein Interrupt oder Reset-Signal eintrifft. Sinn dieser Anweisung ist es, daß die CPU bei einigen sehr hardwarenahen Operationen den Bus freigibt und so andere Bausteine nicht in ihren Buszugriffen behindert, wie sie es in einer Warteschleife täte. Wenn der Baustein die Kontrolle wieder der CPU übergeben will, löst er einen Interrupt aus. Auch das Reset-Signal, ausgelöst durch einen Druck auf den Reset-Knopf, beendet den HALT-Zustand. Diese Anweisung ist nur im Supervisor-Modus ausführbar.

Besonderheit: privilegierter Befehl

RESET

Hardware-Initialisierung

Operation: RESET-Pin auf High setzen

Flags:
N : unberührt
Z : unberührt
V : unberührt
C : unberührt
X : unberührt

Adressierungsarten:

keine

Beschreibung: Der Prozessor setzt eine bestimmte Leitung auf High. Dadurch werden alle Bausteine in einen definierten Grundzustand versetzt. Beim ST wird diese Anweisung nur beim Einschalten des Rechners oder nach dem Drücken des Reset-Knopfes ausgeführt. RESET darf nur im Supervisormodus ausgeführt werden.

Besonderheit: privilegierter Befehl

no operation
nichts tun

NOP

Operation: keine

Flags:
N : unberührt
Z : unberührt
V : unberührt
C : unberührt
X : unberührt

Adressierungsarten:

keine

Beschreibung: Es geschieht nichts. Der Prozessor fährt einfach mit der nächsten Anweisung fort. Dieser Befehl kann z.B. dazu verwendet werden, unerwünschte Befehle in einem Programm zu überschreiben. Es ist vielleicht interessant, den Befehlscode von NOP zu wissen, damit man etwa in einem Debugger Befehle "wegstreichen" kann: Der Code ist $8771.

Line A Emulator/ Line F Emulator

frei belegbare Opcodes

Operation: -(SSP) <- SR
-(SSP) <- PC
PC <- Inhalt von Ausnahmevektor 10/11

Flags: N : unberührt
Z : unberührt
V : unberührt
C : unberührt
X : unberührt

Adressierungsarten:

keine

Beschreibung: Bei einem nicht implementierten Opcodes wird normalerweise die Exception 4 durchgeführt. Alle Opcodes die die Form $Axxx oder $Fxxx haben (xxx steht für drei beliebige Hexadezimalziffern) sind jedoch davon ausgenommen. Der erste wird als ein Line-A-Befehl bezeichnet und führt zu einem Sprung durch den Vektor ab Adresse $28, der zweite, ein Line-F-Befehl, durch $2C. Vom Hersteller der CPU ist diese Funktion dazu gedacht, nicht implementierte Opcodes durch einen solchen Line-A- oder Line-F-Befehl zu ersetzen. Auf dem ATARI ST ist man einen Schritt weiter gegangen: Die Line-A-Befehle mit den Opcodes $A000 bis $A00E werden für grundlegende Grafikroutinen verwendet (siehe auch Kapitel 4) und sind auch für den Programmierer interessant. Die Line-F-Befehle werden intern vom GEM benutzt und sollten besser nicht in eigenen Programmen verwendet werden. In einem Assemblerprogramm werden diese Opcodes etwa folgendermaßen untergebracht:

```
[Programmcode]
    .
    .
DC.W $A000
```

Kapitel 4

Zusammenarbeit mit dem Betriebssystem

In diesem Kapitel werden die Aufrufkonventionen aller auf dem ST vorhandenen Routinen erläutert, die mehr oder weniger eng zum Betriebssystem gehören.

Der ATARI ST verfügt über eine fast unüberschaubare Menge von Betriebssystemroutinen. Allein der Betriebssystemkern bietet 104 Funktionen, die sich aus 53 GEMDOS-, 12 BIOS- und 39 XBIOS-Aufrufen zusammensetzen. Dazu kommen noch die Line-A-Routinen und die über 100 GEM-Aufrufe. Aber keine Angst, das Ganze vereinfacht sich dadurch etwas, daß das Betriebssystem hierarchisch organisiert ist (Abb. 4.1).

Um keine Begriffsverwirrung aufkommen zu lassen: TOS umfaßt das GEMDOS, BIOS und XBIOS. TOS steht für "Tramiel Operating System", benannt nach dem Chef der Firma ATARI, und ist der Name des Betriebssystems der ST-Computer. GEMDOS, BIOS, XBIOS sind nur Gruppen von Betriebssystemroutinen.

Die oberste Ebene des TOS bildet das GEMDOS. Es benutzt die Dienste von BIOS und XBIOS; sie sind für die Ausführung der Operationen – etwa in Form von Hardware-Ansteuerung – zuständig.

Das GEM – namentlich VDI und AES – ist nicht unmittelbarer Teil des Betriebssystems. Wenn die GEM-Routinen benutzt werden, stehen sie noch eine Ebene höher, denn sie benutzen ihrerseits die GEMDOS-Aufrufe. Auf unterster Ebene verfügt GEM noch über die sogenannten Line-A-Routinen, die grundlegende Grafikoperationen bieten. Dem Programmierer stehen sämtliche auf Abb. 4.1 eingezeichneten Routinengruppen zur Verfügung.

Durch diese hierarchische Ordnung ergibt sich, daß oft mehr als ein Weg zum Ziel führt. Es gibt oft mehr als eine mögliche Funktion, um eine bestimmte Operation vorzunehmen.

Logisch, daß eine Funktion um so schneller ausgeführt wird, je niedriger sie in der Hierarchie steht. Diese Vielfalt kann durchaus nützlich sein, da man als Programmierer mitunter die Wahl zwischen schneller ausführenden und einfacher zu benutzenden Funktionen hat. Natürlich muß von Fall zu Fall entschieden werden, worauf man mehr Wert legt.

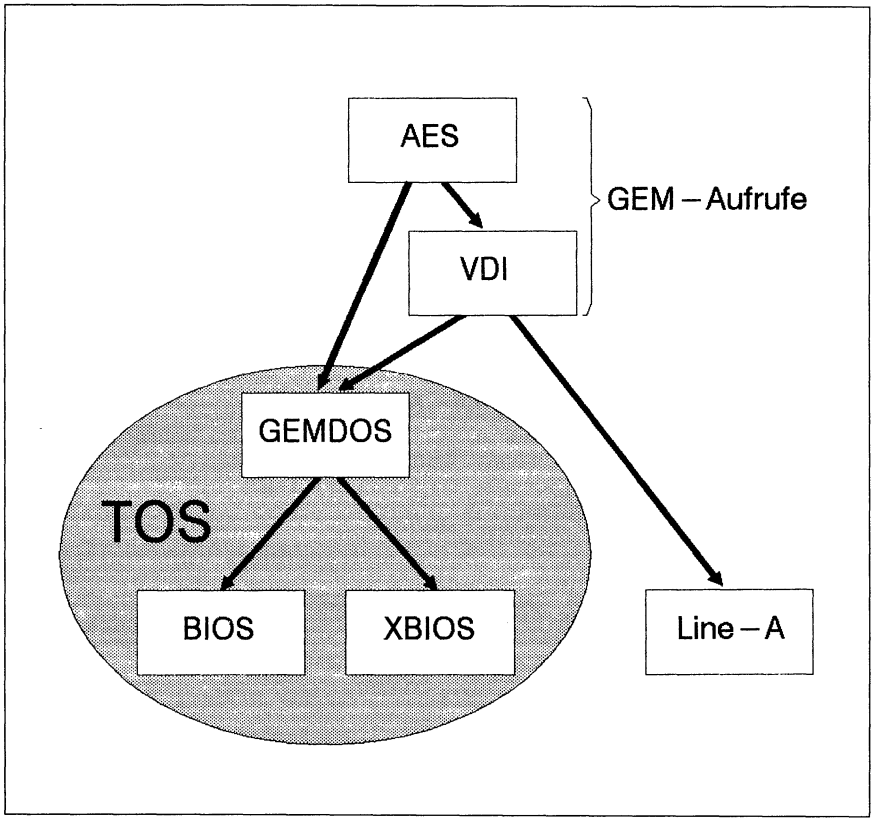


Abb. 4.1: Hierarchie der GEM- und Betriebssystemroutinen
Die Pfeile stehen für die Benutzt-Beziehung

Das GEMDOS

Die GEMDOS-Funktionen orientieren sich stark an den Funktionen des Betriebssystems MS-DOS. Insbesondere die Funktionsnummern entsprechen sich bei beiden Betriebssystemen. Allerdings wurden nicht alle Funktionen des MS-DOS übernommen; besonders Hardware-abhängige Funktionen, die den Prozessor Intel 8086/88 betreffen, wurden natürlich fortgelassen. Daraus erklären sich die Lücken in der Vergabe der Funktionsnummern.

GEMDOS steht für "GEM Disk Operation System", also GEM-Disketten-Betriebssystem. Wie schon der Name andeutet, bietet das GEMDOS Funktionen für den Zugriff auf Geräte und für die Verwaltung von Disketten- oder Plattendateien, darüber hinaus für den Aufruf und die Beendigung von Programmen, Datumsabfrage und das Erlangen des Supervisor-Status.

Sämtliche Parameter für GEMDOS-Routinen werden auf dem Stack übergeben; darunter fällt auch die Funktionsnummer. Dabei sollten Sie darauf achten, ob es sich bei den Parametern um Worte oder Langworte handelt. Der Sprung ins Betriebssystem erfolgt schließlich mit einem TRAP-Befehl, wobei dem GEMDOS der Vektor 1 zugeordnet ist. Sofern die Funktion einen Wert zurückliefert, wird dieser im unteren Wort D0 zurückgegeben. Wie üblich ist das aufrufende Programm dafür verantwortlich, die Parameter nach dem Aufruf wieder vom Stack abzuräumen. Dazu addiert es die Gesamtlänge der Parameter (inklusive Funktionsnummer) zum Stackpointer, wobei für jedes Wort 2 und für jedes Langwort 4 angerechnet werden.

Vielleicht möchten Sie jetzt einwenden, daß die Benutzung des Stacks doch eine recht unpraktische Art der Parameterübergabe ist, wo die Übergabe in Registern – der 68000 hat schließlich genügend davon – mit Sicherheit viel effizienter wäre. Tatsache ist ja auch, daß es bei anderen Betriebssystemen oft so gemacht wird. Allerdings ist dieser Einwand leicht zu kontern: Das Betriebssystem des ATARI ST wurde nicht speziell für Assemblerprogrammierer ausgelegt, sondern vielmehr auf eine einfache Benutzung von C aus. So unterscheidet sich der Aufrufmechanismus der GEMDOS-Funktionen von dem in C eigentlich nur dadurch, daß statt des JSR (oder BSR) ein TRAP den eigentlichen Sprung zur Routine übernimmt.

Bei fast allen Funktionen bedeutet die Rückgabe eines negativen Worts in D0, daß ein Fehler aufgetreten ist; D0 enthält dann den negativen TOS-Fehlercode. Dem GEMDOS sind die Fehlercodes von –32 bis –49 zugeordnet; ihre Beschreibung finden Sie am Ende dieses Abschnitts. Meistens werden Rückgabewerte zwar im Langwortformat in D0 abgelegt, aber bei manchen Routinen kommt es vor, daß nur ein Wort den Fehlercode enthält. Untersuchen Sie deshalb sicherheitshalber immer nur wortweise, ob in D0 eine Fehlernummer steht.

Bei der Programmierung sollten Sie beachten, daß die GEMDOS-Routinen nicht nur das Register D0, sondern auch A0 verändern. A0 wird in den meisten Fällen die Adresse der Funktionsnummer auf dem Stack enthalten. Wenn also der Inhalt von A0 oder D0 nach dem Aufruf noch gebraucht wird, sollten Sie diesen vorher in Sicherheit bringen.

Allgemein formuliert kann ein GEMDOS-Aufruf also folgendermaßen aussehen:

```
    MOVE.x      ...,-(SP)       * Parameter n . . .
    MOVE.x      ...,-(SP)       * Parameter 2
    MOVE.x      ...,-(SP)       * Parameter 1
    MOVE.W      #Nummer,-(SP)   * GEMDOS-Funktionsnummer
    TRAP        #1              * eigentlicher Aufruf
    ADD.W       #Länge,SP       * Stack aufräumen
* Dieser Teil folgt nur, wenn die Funktion tatsächlich
* einen Fehlercode zurückgeben kann
    TST.W       D0              * D0 negativ?
    BMI         Fehler          * Ja, Fehler auswerten
```

Beachten Sie, daß die Parameter im Vergleich zu ihrer Reihenfolge in einer Hochsprache hier genau umgekehrt auf den Stack geschoben werden.

Natürlich bleibt es dem Programmierer überlassen, zur Stack-Korrektur statt des ADD.W-Befehls das schnellere ADDQ zu verwenden, sofern der zu addierende Wert kleiner als 8 ist, oder auch

```
LEA Länge(SP),SP
```

was auch etwas schneller als ADD.W ist.

Ein Beispiel für GEMDOS-Aufrufe wird hier nicht angegeben, da ein solches schon als im Kapitel 2 Abschnitt "Das erste lauffähige Programm" auftauchte.

Eine GEMDOS-Funktion ist für Assembler-Programmierer besonders interessant: Es handelt sich um die Funktion Nummer 32 mit dem Namen SUPER. Mit dieser Funktion wechselt man in den Supervisor-Modus, und man kann ihn mit derselben Funktion auch wieder verlassen. SUPER wirkt wie ein Umschalter; es wird immer in den Modus gewechselt, in dem man sich gerade nicht befindet.

Als Parameter erhält die Funktion den Initialisierungswert des Stackpointers, den man im Supervisor- bzw. User-Modus wünscht. Wird jedoch beim Umschalten in den Supervisormodus eine 0 (Langwort) angegeben, so benutzt die Funktion den alten User-Stackpointer auch für den Supervisor-Modus. Dies ist der Normalfall. Rückgabewert ist dann der alte Wert des Supervisor-Stackpointers. Diesen Wert sollte sich das Programm merken, da er bei der Wieder-

herstellung des User-Modus in jedem Fall gebraucht wird. Unter diesen Voraussetzungen sieht ein Wechsel in den Supervisor-Modus und zurück so aus:

```
* hier läuft das Programm noch im USER-Modus
*
   CLR.L      -(SP)            * 0: User- als Supervisor-Stack
   MOVE.W     #32,-(SP)        * Funktionsnummer von SUPER
   TRAP       #1               * GEMDOS-Aufruf
   ADDQ.L     #6,SP            * Stack-Korrektur
   MOVE.L     D0,old_ssp       * alten Supervisor-Stackpointer
                               * sichern . . .
* hier kann das Programm mit allen Privilegien
* des Supervisor-Modus etwas anstellen . . .
* Rückkehr in den USER-Modus
*
   MOVE.L     old_ssp,-(SP)    * alter Supervisor-Stackpointer
   MOVE.W     #32,-(SP)        * SUPER-Funktionsnummer
   TRAP       #1               * GEMDOS-Trap
   ADDQ.L     #6,SP            * und den Stack korrigieren
* hier läuft das Programm wieder im User-Modus
```

Sämtliche Betriebssystemaufrufe funktionieren auch im Supervisormodus, es sei denn, man ist durch die XBIOS-Funktion SUPEXEC hineingelangt. Allerdings muß ein Programm vor seiner Beendigung wieder in den User-Modus schalten, sonst gibt es bei der Rückkehr zum Desktop Bomben...

Es ist überhaupt empfehlenswert, den Supervisor-Modus nur dann einzuschalten, wenn man ihn wirklich braucht, denn bei einem Programmabsturz im Supervisormodus ist die Chance geringer, daß das System wieder "hochgezogen" werden kann. Der Usermodus ist ja gerade dazu da, wichtige Speicherbereiche des Betriebssystems vor unkontrollierten Zugriffen zu schützen.

SUPER hat eine Besonderheit: Es ist die einzige GEMDOS-Funktion, bei der auch die Register A1 und D1 verändert werden können. Wenn also deren Inhalte von Bedeutung sind, sollten diese Register vorher gesichert werden.

Nun zur Beschreibung der GEMDOS-Aufrufe im einzelnen:

Die hexadezimale Zahl vor jeder Funktionsbeschreibung ist die Funktionsnummer, die beim Aufruf als oberster Wert auf dem Stack stehen muß. Die restlichen Parameter werden in C-ähnlicher Syntax angegeben. Bei Verwendung der Funktionsaufrufe von Assembler aus müssen also zuerst die eigentlichen Parameter in der umgekehrten Reihenfolge wie hier angegeben, also von rechts nach links, auf dem Stack abgelegt werden, dann folgt die Funktionsnummer. Nach der Ausführung des TRAP #1 muß der Stack korrigiert werden; der Wert, der dafür addiert werden muß, wird in eckigen Klammern hinter dem C-Funktionsaufruf angegeben. Danach werden die Typen der einzel-

nen Parameter angegeben. Es ist zu beachten, daß mit dem Typ Byte gemeint ist, daß nur das untere Byte eines Wortes von Bedeutung ist; es muß aber trotzdem ein Wort auf dem Stack abgelegt werden.

Langworte haben meist die Bedeutung von Zeigern, größtenteils auf Zeichenketten. Zeichenketten müssen immer mit einem Nullbyte abgeschlossen werden.

Rückgabewerte werden immer in D0 übergeben. –1 zeigt dabei gewöhnlich an, daß die Funktion korrekt ausgeführt worden ist.

Um es noch einmal deutlich zu machen, setzen wir einmal die GEMDOS-Funktion Dfree um. Beschrieben ist sie folgendermaßen:

```
$36   Dfree(buf,drv) [8]
      buf:       Langwort
      drv:       Wort
```

In Assembler sieht der Aufruf so aus:

```
MOVE.W     drv,-(SP)        * 2. Parameter
MOVE.L     buf,-(SP)        * 1. Parameter
MOVE.W     #$36,-(SP)       * Funktionsnummer
TRAP       #1               * ins GEMDOS
ADDQ.L     #8,SP            * Stack korrigieren
```

$00 Pterm0() [2]
Programm beenden; die Kontrolle wird an das aufrufende Programm zurückgegeben (mit Rückgabewert 0).

$01 Cconin() [2]
Ein Zeichen vom Standardeingabekanal (Tastatur) lesen und gleichzeitig auf dem Standardausgabekanal (Bildschirm) ausgeben. Rückgabewert ist ein Langwort, wobei im untersten Byte das gelesene ASCII-Zeichen steht, während das untere Byte des oberen Wortes den Tastaturcode enthält.

$02 Cconout(chr) [4]
chr: Byte
Schreibt ein Zeichen auf den Standardausgabekanal (Bildschirm). Escape-Sequenzen werden richtig interpretiert.

$03 Cauxin() [2]
Liest ein Zeichen von der seriellen Schnittstelle (RS232-Schnittstelle).

$04 Cauxout(chr) [4]
chr: Byte
Gibt ein Zeichen auf der seriellen Schnittstelle aus.

$05 Cprnout(chr) [4]
chr: Byte
Gibt ein Zeichen auf dem Drucker aus. Wenn das Zeichen nicht gedruckt werden kann, erhält man den Rückgabewert 0, andernfalls –1.

$06 Crawio(wrd) [4]
wrd: Wort
Wenn wrd den Wert $00FF hat, wird ein Zeichen ohne Echo vom Standardeingabekanal gelesen, sofern gerade eins im Puffer ansteht; ist gerade keines verfügbar, dann ist der Rückgabewert 0. Bei allen anderen Werten außer $00FF wird "wrd" als ASCII-Zeichen interpretiert und auf dem Standardausgabekanal ausgegeben.

$07 Crawin() [2]
Ein Zeichen vom Standardeingabekanal ohne Echo lesen. Steuerzeichen wie <Ctrl>-<C>, <Ctrl>-<S> oder <Ctrl>-<Q> werden nicht interpretiert.

$08 Cnecin() [2]
Ein Zeichen vom Standardeingabekanal ohne Echo lesen. Hier werden Steuerzeichen wie <Ctrl>-<C>, <Ctrl>-<S> oder <Ctrl>-<Q> interpretiert.

$09 Cconws(str) [6]
str: Langwort
Eine Zeichenkette, deren Adresse in "str" übergeben wird, wird auf dem Standardausgabekanal ausgeben. Die Zeichenkette muß mit einem Nullbyte abgeschlossen sein.

$0A Cconrs(buf) [6]
buf: Langwort
Editierbare Zeichenkette vom Standardeingabekanal lesen. "buf" zeigt auf einen reservierten Bereich; das erste Byte dieses Bereichs enthält die maximale Länge der einzugebenden Zeichenkette, das zweite nach der Rückkehr die tatsächliche Anzahl der eingegebenen Zeichen. Die gelesenen Zeichen werden ab dem dritten Byte abgelegt.

$0B Cconis() [2]
Gibt null zurück, wenn ein Zeichen am Standardeingabekanal verfügbar ist, andernfalls einen von null verschiedenen Wert.

$0E Dsetdrv(drv) [4]
Aktuelles Laufwerk festlegen; "drv" ist die Laufwerksnummer (0 = A:, 1 = B:, ...). Rückgabewert ist ein Bitvektor der momentan angeschlossenen Laufwerke (Bit 0 = A:, Bit 1 = B:, ...)

$10 Cconos() [2]
Gibt null zurück, wenn die Console bereit ist, Zeichen auszugeben, andernfalls einen von null verschiedenen Wert.

$11 Cprnos() [2]
Gibt null zurück, wenn der Drucker bereit ist, Zeichen auszugeben, andernfalls einen von null verschiedenen Wert.

$12 Cauxis() [2]
Gibt null zurück, wenn ein Zeichen an der seriellen Schnittstelle verfügbar ist, andernfalls einen von null verschiedenen Wert.

$13 Cauxos() [2]
Gibt null zurück, wenn die serielle Schnittstelle bereit ist, Zeichen auszugeben, andernfalls einen von null verschiedenen Wert.

$19 Dgetdrv() [2]
Gibt die Nummer des aktuellen Laufwerks zurück (0 = A:, 1 = B:, ...)

$20 Super(stack) [6]
stack: Langwort
Wechselt zwischen User- und Supervisormodus. Wenn sich der Prozessor beim Aufruf von Super im Usermodus befindet, schaltet er den Supervisor-Modus ein und benutzt "stack" als den neuen Supervisor-Stackpointer. Wenn "stack" den Wert Null hat, wird der User-Stack als Supervisorstack weiterbenutzt. Wenn der Prozessor sich im Supervisormodus befindet, schaltet er in den Usermodus um und benutzt fortan "stack" als Supervisor-Stackpointer. Wenn "stack" den Wert –1 hat, wird nur 0 oder 1 zurückgegeben, je nachdem, ob der Prozessor sich im User- oder Supervisormodus befindet.

$2A Tgetdate() [2]
Gibt das Systemdatum zurück. Rückgabewert ist ein Bitfeld, das folgendermaßen interpretiert werden muß:

Bits	
0..4	Tag 1..31
5..8	Monat 1..12
9..15	Jahr 0..119, gerechnet ab 1980

$2B Tsetdate(date) [4]
date: Wort
Systemdatum in obigem Format setzen.

$2C Tgettime() [2]
Gibt die Systemzeit zurück. Rückgabewert ist ein Bitfeld, das folgendermaßen interpretiert werden muß:

Bits	
0...4	Sekunden 1..59
5...8	Minuten 1..59
9...15	Stunden 0..23

$2D Tsettime(time) [4]
time: Wort
Systemzeit in obigem Format setzen.

$2F Fgetdta() [2]
Ermittelt die Adresse des aktuellen DTA (Disk Transfer Address). Siehe Funktionen Fsetdta, Fsnext und Fsfirst.

$30 Sversion() [2]
Gibt die Versionsnummer des GEMDOS zurück.

$31 Ptermres(keep,ret) [8]
keep: Langwort
ret: Wort
Beendet einen Prozeß, reserviert aber den von ihm belegten Speicher. "ret" ist der Rückgabewert des Prozesses, "keep" die Anzahl der Bytes ab Anfang der Basepage, die reserviert werden sollen.

$36 Dfree(buf,drv) [8]
buf: Langwort
drv: Wort
Liefert Informationen über das Laufwerk, dessen Nummer in "drv" übergeben wird (0=aktuelles Laufwerk, 1=A:, 2=B:, ...). "buf" zeigt auf mindestens 16 reservierte Bytes, in denen vier Langworte zurückgegeben werden:

buf	b_free	Anzahl der freien Cluster
buf+4	b_total	Gesamtanzahl Cluster
buf+8	b_secsiz	Anzahl der Bytes in einem Sektor
buf+12	b_clsiz	Anzahl Sektoren pro Cluster

$39 Dcreate(path) [6]
path: Langwort
Erzeugt einen Ordner. "path" zeigt auf eine Zeichenkette, die den Pfadnamen des anzulegenden Ordners enthält und mit einem Nullbyte abgeschlossen ist.

$3A Ddelete(path) [6]
path: Langwort
Löscht einen Ordner. "path" zeigt auf eine Zeichenkette, die den Pfadnamen des zu löschenden Ordners enthält und mit einem Nullbyte abgeschlossen ist.

$3B Dsetpath(path) [6]
path: Langwort
Legt das aktuelle Directory fest. "path" zeigt auf eine Zeichenkette mit dem Pfadnamen.

$3C Fcreate(name,attr) [8]
name: Langwort
attr: Wort
Legt eine Datei an; "name" zeigt auf den Dateinamen (Pfadnamen). Wenn eine Datei mit dem angegebenen Namen schon existiert, wird sie auf Länge Null abgeschnitten. Im Parameter "attr" können Datei-Attribute festgelegt werden:

Bit	
0	nur Lesezugriff auf Datei erlaubt
1	versteckte Datei; wird nicht angezeigt
2	Systemdatei; ebenfalls versteckt
3	Diskettenkennung (11 Zeichen lang)

Rückgabewert ist eine Handle-Nummer oder eine negative Fehlernummer.

$3D Fopen(name,mode) [8]
name: Langwort
mode: Wort
Öffnet eine Datei mit dem angegebenen Namen. "mode" bestimmt die Zugriffsart:

mode	
0	nur lesen
1	nur schreiben
2	lesen und schreiben

Rückgabewert ist eine Handle-Nummer oder eine negative Fehlernummer. Die kleinste mögliche Handle-Nummer ist 6, da die ersten 5 Nummern bereits für die Standard-Kanäle vergeben sind, nämlich

0 für Standard-Eingabe (Tastatur)
1 für Standard-Ausgabe (Bildschirm)
2 für Standard-Fehlerkanal (Bildschirm)
3 für die serielle Schnittstelle (AUX:)
4 für den Drucker (PRN:, LST:)

$3E Fclose(handle) [4]
handle: Wort
Schließt die Datei mit der Nummer handle.

$3F Fread(handle,count,buf) [12]
handle: Wort
count: Langwort
buf: Langwort
Bytes aus einer Datei lesen. Aus der Datei mit der Nummer "handle" werden count Bytes in den Puffer "buf" gelesen. Rückgabewert ist die Anzahl der tatsächlich gelesenen Bytes oder eine negative Fehlernummer.

$40 Fwrite(handle,count,buf) [12]
handle: Wort
count: Langwort
buf: Langwort
Bytes in eine Datei schreiben. In die Datei mit der Nummer "handle" werden "count" Bytes aus dem Puffer "buf" geschrieben. Rückgabewert ist die Anzahl der tatsächlich geschriebenen Bytes oder eine negative Fehlernummer.

$41 Fdelete(name) [6]
name: Langwort
Löscht die Datei mit dem angegebenen Namen.

$42 Fseek(offset,handle,mode) [10]
offset: Langwort
handle: Wort
mode: Wort
Setzt den Dateizeiger für Schreib/Lesezugriffe auf einen bestimmten Wert. "offset" wird je nach dem Wert von "mode" unterschiedlich interpretiert:

mode
0 ab Dateianfang positionieren
1 ab aktueller Position positionieren
2 ab Dateiende positionieren (offset muß negativ sein)

$43 Fattrib(path, mode, attr) [10]
path: Langwort
mode: Wort
attr: Wort
Wenn "mode" null ist, liefert die Funktion die Attribute der mit "path" bezeichneten Datei. Hat "mode" den Wert 1, so werden die Attribute der Datei auf den Wert 1 gesetzt. Die Bits in "attr" haben folgende Bedeutung:

Bit
0 nur Lesezugriff erlaubt
1 versteckte Datei
2 Systemdatei (auch versteckt)
3 Diskettenkennung
4 Ordner
5 Archiv-Bit, z.Z. unbenutzt

$45 Fdup(stdhandle) [4]
stdhandle: Wort
Liefert eine zweite Handle-Nummer für einen Standard-Kanal, der mit "stdhandle" ausgewählt wird (siehe Fopen). Man hat somit zwei Handle-Nummern, die sich auf das gleiche Gerät beziehen.

$46 Fforce(stdhandle, nonstdhandle) [6]
stdhandle: Wort
nonstdhandle: Wort
Leitet den Kanal "stdhandle" (siehe Fopen) in den Kanal "nonstdhandle" um. Wenn "nonstdhandle" beispielsweise eine Datei bezeichnet, kann damit die Ausgabe, die normalerweise auf dem Bildschirm erfolgen würde, in diese Datei umgelenkt werden.

$47 Dgetpath(pathbuf, drv) [8]
pathbuf: Langwort
drv: Wort
Schreibt den aktuellen Zugriffspfad für das angegebene Laufwerk in den Puffer "pathbuf". Es sollten mindestens 64 Bytes reserviert werden.

$48 Malloc(amount) [6]
amount: Langwort
Reserviert Speicherplatz. "amount" gibt die Anzahl der benötigten Bytes an; Rückgabewert ist ein Zeiger auf die (gerade) Anfangsadresse des reservierten Speicherbereiches oder 0, falls ein Fehler auftrat.

$49 Mfree(addr) [6]
addr: Langwort
Gibt einen mit Malloc reservierten Speicherblock wieder frei. Dessen Adresse wird in "addr" übergeben.

$4A Mshrink(zero, mem, size) [12]
zero: Wort
mem: Langwort
size: Langwort
Verkürzt einen mit Malloc reservierten Speicherblock. "zero" muß null sein, "mem" ist die Adresse des Speicherblocks, "size" die Anzahl der Bytes, die im Block gehalten werden sollen.

$4B Pexec(mode, path, commandline, environment) [16]
mode: Wort
path: Langwort
commandline: Langwort
environment: Langwort
Lädt und/oder startet ein anderes Programm. "path" gibt den Pfadnamen des Programms an, "commandline" eine Kommandozeile, die dem Programm übergeben wird, "environment" den MS-DOS-kompatiblen Environment-String; ist "environment" null, dann erbt das Programm den Environment-String des aufrufenden Prozesses. Zulässige Werte für "mode" sind:

mode
0 laden und starten
3 nur laden
4 nur Basepage für das Programm einrichten
5 Programm starten

Das aufrufende Programm bleibt in jedem Fall resident. Im Modus Null wird der Rückgabewert des aufgerufenen Programms geliefert; ein negativer Funktionswert signalisiert in jedem Fall einen Fehler.

$4C Pterm(code)
code: Wort
Programm beenden und den angegebenen Wert an das aufrufende Programm zurückgeben.

$4E Fsfirst(spec, attr)[8]
spec: Langwort
attr: Wort
Directory nach einer Datei durchsuchen, deren Name auf das Muster paßt, auf das "spec" zeigt. Mit "attr" können nur Dateien mit bestimmten Attributen ausgewählt werden (siehe Fattrib). Das Ergebnis wird in der aktuellen DTA abgelegt:

Bytes	
0..20	uninteressant
21	Dateiattribut
22 – 23	Dateierzeugungszeit
24 – 25	Dateierzeugungsdatum
26 – 29	Dateigröße (Langwort)
30 – 43	Dateiname

$4F Fsnext() [2]
Nächste Datei suchen, die auf die Angaben von Fsfirst paßt.

$56 Frename(zero, old, new) [12]
zero: Wort
old: Langwort
new: Langwort
Ändert einen Dateinamen von "old" nach "new". Zero ist reserviert und muß den Wert 0 haben.

$57 Fdatime(handle, buf, set) [10]
handle: Wort
buf: Langwort
set: Wort
Wenn "set" den Wert Null hat, wird Datum und Zeit der Erzeugung der Datei mit der Nummer "handle" im Puffer "buf" abgelegt; wenn "set" 1 ist, werden Datum und Zeit aus dem Puffer "buf" festgelegt.

Fehlermeldungen

Beim GEMDOS können folgende negative Fehlernummern auftreten:

–32 Ungültige Funktionsnummer
–33 Datei nicht gefunden
–34 Pfadname nicht gefunden
–35 Zu viele Dateien geöffnet
–36 Zugriff nicht möglich

Gelegentlich kann es auch vorkommen, daß das GEMDOS BIOS-Fehlermeldungen zurückgibt. Die BIOS-Fehlermeldungen finden Sie am Ende des nächsten Abschnitts.

Das BIOS

BIOS steht für "Basic Input/Output System". Diese Routinensammlung stellt zusammen mit dem XBIOS die Schnittstelle zwischen dem GEMDOS und der Hardware dar.

Wie der Name schon sagt, ist es Aufgabe des BIOS, sich um grundlegende Ein- und Ausgabefunktionen zu kümmern. Dazu gehört das Schreiben von Zeichen auf den Bildschirm und deren Übertragung zum Drucker, das Einlesen von Zeichen von der Tastatur und der Zugriff auf Diskettendateien.

Das BIOS ist aber nicht nur "Sklave des GEMDOS für niedere Tätigkeiten", sondern kann auch vom Programmierer aufgerufen werden. Der Aufruf erfolgt wie beim GEMDOS: Zuerst werden die Parameter der Reihe nach auf dem Stack abgelegt, dann folgt ein Wort mit der Funktionsnummer, und der Einsprung ins Betriebssystem wird schließlich mit TRAP #13 ausgeführt. Es folgt noch die unvermeidliche Stackkorrektur, und der Rückgabewert des Aufrufs befindet sich im Register D0.

Einen Unterschied zu den GEMDOS-Aufrufen gibt es: Beim BIOS können die Register D0 – D2 und A0 – A2 verändert werden.

Es ist vielleicht manchmal von Bedeutung, daß die Funktionsnummer auf dem Stack vom BIOS auf null gesetzt wird; Sie sollten also nicht versuchen, diese in einem darauffolgenden BIOS-Aufruf zu "recyclen". (Durch diese Maßnahme werden bei jedem BIOS-Aufruf einige Taktzyklen gespart).

Für besonders trickreiche Programmierung kann es von Bedeutung sein, daß das BIOS bis zu dreifach reentrant ist; das heißt, daß man etwa aus einem Interrupt heraus problemlos eine BIOS-Funktion ausführen kann, während das Vordergrundprogramm gleichzeitig ebenfalls einen BIOS-Aufruf ablaufen läßt. Diese Eigenschaft erlaubt es auch, aus eigenen BIOS-Routinen heraus weitere BIOS-Aufrufe auszuführen.

Als Beispiel soll hier ein Programmteil dienen, der ein Zeichen von der Tastatur einliest – eine Alternative zur GEMDOS-Funktion CNECIN (Nummer 8). Dazu dient die BIOS-Funktion BCONIN (Nummer 2), nicht zu verwechseln mit der GEMDOS-Funktion CONIN. BCONIN erhält als Parameter einen Code für das Gerät, von dem ein Zeichen gelesen werden soll. Dabei steht 2 für Eingabe von der Tastatur.

```
CON     EQU 2                   * Code für Console=Tastatur
BCONIN  EQU 2                   * Funktionscode . . . liestaste
        MOVE.W  #CON,-(SP)      * Gerätenummer
        MOVE.W  #BCONIN,-(SP)   * Funktionscode
        TRAP    #13             * BIOS-Aufruf
        ADDQ.L  #4,SP           * Stack korrigieren
*
* Jetzt steht der ASCII-Wert des gelesenen Zeichens
* im untersten Byte von D0
```

Nun zu den einzelnen BIOS-Aufrufen:

$00 getmpb(p_mpb) [6]
p_mpb: Langwort
Füllt einen Puffer mit Informationen über die Speicheraufteilung des Systems. MPB steht für "Memory Parameter Block".

$01 Bconstat(dev) [4]
dev: Wort
Liefert den Eingabestatus eines zeichenorientierten Geräts. Rückgabewert ist 0, wenn gerade kein Zeichen verfügbar ist und –1 ($FFFF), wenn mindestens ein Zeichen ansteht. Die Werte von "dev" haben folgende Bedeutungen:

0	PRT:	(Drucker; Parallelport)
1	AUX:	(RS232-Schnittstelle)
2	CON:	(Tastatur)
3	MIDI:	(Midi-Schnittstelle)
4	KBD:	(Tastaturprozessor)

$02 Bconin(dev) [4]
dev: Wort
Liest ein Zeichen von dem angegebenen Gerät (Bedeutung von "dev" siehe Bconstat). Die Funktion wartet so lange, bis ein Zeichen eintrifft, und gibt das Zeichen dann in D0 zurück, wobei das obere Wort von D0 null ist.

Bei der Console-Eingabe (Gerätenummer 2) wird zusätzlich der Tastatur-Scancode im unteren Byte des oberen Wortes von D0 zurückgegeben.

$03 Bconout(dev, c) [6]
dev: Wort
c: Byte
Gibt das Zeichen c auf dem angegebenen Gerät aus. Die Funktion kehrt nicht zurück, bevor das Zeichen ausgegeben wurde, also unter Umständen nie, wenn das entsprechende Gerät nicht bereit war.

$04 Rwabs(rwflag, buf, count, recno, dev) [12]
rwflag: Wort
buf: Langwort
count: Wort
recno: Wort
dev: Wort
Liest oder schreibt logische Sektoren auf ein Laufwerk. "rwflag" darf folgende Werte annehmen:

0 lesen
1 schreiben
2 lesen, Diskettenwechsel ignorieren
3 schreiben, Diskettenwechsel ignorieren

"buf" zeigt auf einen Puffer, aus dem gelesen bzw. in den geschrieben werden soll. Ungerade Werte von "buf" sind zwar erlaubt, verlangsamen aber die Übertragung. "count" gibt an, wie viele Sektoren übertragen werden sollen. "recno" ist die logische Sektornummer, bei der die Übertragung beginnt. "dev" ist die Gerätenummer des Laufwerks:

0 Diskettenlaufwerk A:
1 Diskettenlaufwerk B:
2 und größer: Festplatten, RAM-Disks, ...

Rückgabewert 0 signalisiert eine erfolgreiche Operation; negative Werte zeigen einen Fehler an.

$05 Setexc(vecnum, vec) [8]
vecnum: Wort
vec: Langwort
Setzt den Exception-Vektor mit der Nummer "vecnum" auf den Wert "vec". Wenn "vec" allerdings den Wert –1 hat, wird lediglich der momentane Inhalt des Vektors zurückgegeben.

$06 tickcal() [2]
Liefert die zwischen zwei Aufrufen des System-Timers verstrichene Zeit auf Millisekunden gerundet.

$07 getbpb(dev) [4]
dev: Wort
"dev" ist eine Laufwerksnummer (0 = A:, 1 = B:, ...). Die Funktion gibt einen Zeiger auf den BIOS-Parameter-Block des Geräts zurück (siehe Abschnitt Kapitel 6 unter "Eine RAM-Disk").

$08 Bcostat(dev) [4]
dev: Wort
"dev" ist eine Gerätenummer wie in Bconstat. Die Funktion liefert 0 zurück, wenn das Gerät bereit ist, Zeichen zu empfangen, andernfalls –1.

$09 mediach(dev) [4]
dev: Wort
"dev" ist eine Laufwerksnummer. Rückgabewert ist einer der folgenden:

0 mit Sicherheit kein Diskettenwechsel
1 Diskettenwechsel könnte stattgefunden haben
2 Diskettenwechsel hat stattgefunden.

$0A Drvmap() [2]
Liefert einen Bitvektor, in dem ein gesetztes Bit signalisiert, daß ein entsprechendes Laufwerk angeschlossen ist (0..31). Der Aufruf liefert den Inhalt der Systemvariable _drvbits. Für das Eintragen der Bits in dieser Variablen sind die Laufwerkstreiber selbst verantwortlich.

$0B Kbshift(mode) [4]
mode: Wort
Setzt die Shift-Bits der Tastatur auf den Wert "mode". Wenn allerdings "mode" negativ ist, werden nur die aktuellen Shift-Bits zurückgegeben. Die einzelnen Bits haben folgende Bedeutung:

0 rechte Shift-Taste
1 linke Shift-Taste
2 Control-Taste
3 Alternate-Taste
4 Caps-Lock-Taste
5 rechter Mausknopf (CLR/HOME)
6 linker Mausknopf (INSERT)
7 (momentan nicht benutzt; enthält 0)

Fehlermeldungen

Das BIOS kann eine der folgenden Fehlernummern zurückgeben:

–1 Kein Fehler; alles in Ordnung
–2 Laufwerk spricht nicht an (Timeout)
–3 Unbekannter Befehl
–4 CRC Fehler; Prüfsumme falsch
–5 Ungültiger Befehl

–6	Track nicht gefunden
–7	Falscher Bootsektor
–8	Sektor nicht gefunden
–9	Kein Papier im Drucker
–10	Schreibfehler
–11	Lesefehler
–12	Allgemeiner Fehler
–13	Diskette schreibgeschützt
–14	Diskette wurde gewechselt
–15	Unbekanntes Gerät
–16	Defekter Sektor (Verify)
–17	Andere Diskette einlegen

Das XBIOS

XBIOS steht für "extended BIOS", eine Routinensammlung, die Funktionen zur Ausnutzung der speziellen Hardwaremöglichkeiten des ATARI ST bietet. GEMDOS und BIOS bieten Routinen, die in ähnlicher Form auf allen Computern verfügbar sein müssen; XBIOS ist dagegen speziell auf den ST zugeschnitten.

Das XBIOS bietet zunächst einmal Funktionen zur Ansteuerung der Chips des ATARI ST: Grafikchip, Soundchip, Tastaturprozessor, Schnittstellen-Controller und Floppy-Controller. Darüber hinaus gibt es noch Funktionen für Zufallszahlen, Einstellung des Druckertreibers und Ausführung einer Bildschirm-Hardcopy.

Der dem XBIOS zugeordnete TRAP-Vektor ist Nummer 14. Ansonsten erfolgt der Aufruf einer XBIOS-Funktion genau wie beim BIOS. Auch in XBIOS-Aufrufen können die Register D0 – D2 und A0 – A2 verändert werden.

Als Beispiel soll unser Programmausschnitt eine Einstellung am Tastaturprozessor vornehmen: Die Tastenwiederholrate soll verringert werden. Dazu wird der XBIOS-Aufruf KBRATE (35) verwendet, der zwei Parameter erhält: Der erste ist die Verzögerung zwischen dem Niederdrücken einer Taste und dem Einsetzen der Wiederholfunktion, der zweite gibt die Zeitspanne zwischen zwei Wiederholungen bei gedrückter Taste an. Beide Werte werden in Vielfachen einer fünfzigstel Sekunde angegeben. Wenn einer der beiden Werte nicht verändert werden soll, wird statt der Zeitspanne –1 ($FFFF) übergeben. Wenn wir die Wiederholrate auf fünf Wiederholungen pro Sekunde, also 10/50 Sekunden für eine Wiederholung setzen wollen, sieht das folgendermaßen aus:

```
KBRATE EQU 35                   * XBIOS-Funktionsnummer
  .
  .
  .
MOVE.W     #10,-(SP)            * Wiederholung alle 10/50 s
MOVE.W     #-1,-(SP)            * Verzögerung nicht ändern
MOVE.W     #KBRATE,-(SP)        * XBIOS-Funktionsnummer
TRAP       #14                  * XBIOS-Aufruf
ADDQ.L     #6,SP                * Stack korrigieren
```

Hier wieder eine Beschreibung sämtlicher Aufrufe:

$00 Initmous(type, param, vec) [12]
type: **Wort**
param: **Langwort**
vec: **Langwort**
Intitialisiert den Maus-Handler. Dieser Aufruf ist in erster Linie von betriebssysteminterner Bedeutung und wird deshalb hier nicht näher beschrieben.

$01 Ssbrk(amount) [4]
amount: **Wort**
Reserviert "amount" Bytes am oberen Ende des Speichers. Zurückgegeben wird die Anfangsadresse des reservierten Speicherbereiches. Leider funktioniert diese Funktion nur, bevor das Betriebssystem initialisiert ist. Deshalb ist sie kaum von praktischer Bedeutung.

$02 _physBase() [2]
Liefert die physikalische Bildschirmadresse (wartet bis zum nächsten VBI).

$03 _logBase() [2]
Liefert die logische Bildschirmadresse.

$04 _getRez() [2]
Liefert die momentane Bildschirmauflösung. 0 steht für niedrige Auflösung, 1 für mittlere, 2 für hohe.

$05 _set Screen(logLoc, physLoc, rez) [12]
logLoc: **Langwort**
physLoc: **Langwort**
rez: **Wort**
Legt die logische Bildschirmadresse, die physikalische Bildschirmadresse und die Auflösung fest. Negative Parameter werden ignoriert; so braucht man nicht alle drei Dinge auf einmal festzulegen. Die logische Bildschirmadresse wird sofort geändert; die physikalische erst nach dem nächsten VBI. Wenn die Auflösung geändert wird, wird der Bildschirm

gelöscht, der Cursor in die linke obere Ecke gesetzt und der VT52-Emulator neu initialisiert.

$06 _setPallete(palettePtr) [6]
palettePtr: Langwort
Belegt alle 16 Farbregister des Grafikchips von den 16 Worten, auf die palettePtr zeigt, neu. Die Farben werden erst im nächsten VBI geändert.

$07 _setColor(colorNum, color) [6]
colorNum: Wort
color: Wort
Setzt die Farbe "colorNum" (0..15) auf den angegebenen Wert. Wenn der Farbwert negativ ist, wird er ignoriert. In D0 wird der alte Farbwert zurückgegeben.

$08 _floprd(buf, filler, devno, sectno, trackno, sideno, count) [20]
buf: Langwort
filler: Langwort
devno: Wort
sectno: Wort
trackno: Wort
sideno: Wort
count: Wort
Liest einen oder mehrere Sektoren von einer Diskette. "buf" zeigt auf einen Puffer an Wortadresse, in den die Daten geschrieben werden. "filler" ist ein unbenutztes Langwort. "devno" ist die Laufwerksnummer (0 oder 1), "sectno" die sektornummer (normalerweise 1 – 9), "trackno" die Tracknummer, "sideno" die Diskettenseite (0 oder 1). "count" gibt an, wie viele Sektoren gelesen werden sollen (nicht mehr, als sich auf einem Track befinden). Ein negativer Rückgabewert in D0 zeigt einen Fehler an.

$09 _flopwr(buf, filler, devno, sectno, trackno, sideno, count) [20]
buf: Langwort
filler: Langwort
devno: Wort
sectno: Wort
trackno: Wort
sideno: Wort
count: Wort
Schreibt einen oder mehrere Sektoren auf eine Diskette. "buf" zeigt auf einen Puffer an gerader Adresse, der die zu schreibenden Daten enthält. "filler" ist ein unbenutztes Langwort. "devno" ist die Laufwerksnummer (0 oder 1), "sectno" die Sektornummer (normalerweise 1 – 9), "trackno"

die Tracknummer (normalerweise 0 – 79), "sideno" die Diskettenseite (0 oder 1). "count" gibt an, wie viele Sektoren geschrieben werden sollen (nicht mehr, als sich auf einem Track befinden). Ein negativer Rückgabewert in D0 zeigt einen Fehler an.

$0A _flopfmt(buf, filler, devno, spt, trackno, sideno, interlv, magic, virgin) [26]

buf: **Langwort**
filler: **Langwort**
devno: **Wort**
spt: **Wort**
trackno: **Wort**
sideno: **Wort**
interlv: **Wort**
magic: **Langwort**
virgin: **Wort**

Formatiert einen Track auf einer Diskette. "buf" zeigt auf einen Puffer an gerader Adresse, der die gesamten Trackdaten aufnehmen kann (8K für 9 Sektoren pro Track). "filler" ist ein ungenutztes Langwort. "devno" gibt die Laufwerksnummer an (0 oder 1), "spt" die Anzahl der Sektoren pro Track (1 – 11, üblicherweise 9), "trackno" die Tracknummer (normalerweise 0 – 79), "sideno" die Diskettenseite (0 oder 1). "magic" ist eine magische Zahl, die den Wert $87654321 haben muß. "virgin" ist ein Wort Füllwert, mit dem die neuen Sektoren beschrieben werden (üblicherweise $E5E5). Wenn defekte Sektoren auftreten, wird eine Liste im Puffer "buf" abgelegt, die von einer Null beendet wird. Gab es keine defekten Sektoren, so ist gleich das erste Wort null. Durch das Formatieren eines Tracks wird der Diskettenwechselstatus auf "Wechsel hat stattgefunden" gesetzt.

$0B used-by-BIOS()
interne Funktion

$0C midiws(cnt,ptr) [8]

cnt: **Wort**
ptr: **Langwort**

Schreibt eine Zeichenkette zur MIDI-Schnittstelle. "cnt" gibt die Anzahl der zu übertragenden Zeichen minus 1 an, "ptr" die Adresse der Zeichenkette.

$0D _mfpint(interno, vector) [8]

interno: **Wort**
vector: Langwort

Setzt den MFP-Interrupt-Vektor Nummer "interno" (0-15) auf den Wert "vektor".

$0E iorec(devno) [4]
devno: Wort
Gibt einen Zeiger auf einen "input buffer record" eines seriellen Geräts. Diese Funktion ist hauptsächlich von interner Bedeutung und wird deshalb nicht weiter erklärt.

$0F rsconf(speed, flowctl, ucr, rsr, tsr, scr) [14]
speed: Wort
flowctl: Wort
ucr: Wort
rsr: Wort
tsr: Wort
scr: Wort
Konfiguriert die RS232-Schnittstelle. Jeder Parameter, der den Wert –1 ($FFFF) hat, wird ignoriert. "speed" legt die Übertragungsrate fest, etwa 9 für 300 bps oder 7 für 1200 bps. "flow" bestimmt das Übertragungsprotokoll, "ucr", "rsr", "tsr" und "scr" werden in die entsprechenden Register des MFP 68901 geschrieben.

$16 keytbl(unshift, shift, capslock) [14]
unshift: Langwort
shift: Langwort
capslock: Langwort
Setzt Zeiger auf die Tabellen, mit denen der Tastatur-Scancode in ASCII umgewandelt wird. "unshift", "shift" und "capslock" setzen Zeiger für Tasten ohne Shift, Tasten mit Shift und Tasten im Caps-Lock-Modus. Jede der Tabellen ist 128 Bytes lang. Zurückgegeben wird ein Zeiger auf die Struktur "keytab": keytab: unshift, keytab+4: shift, keytab+8: capslock

$11 random() [2]
Gibt eine 24-Bit-Zufallszahl zurück. Bits 24..31 sind null. Bei jedem Systemstart wird eine neue Sequenz von Zufallszahlen geliefert.

$12 _protobt(buf, seralno, disktype, execflag) [14]
buf: Langwort
serialno: Langwort
disktype: Wort
execflag: Wort
Erzeugt einen Bootsektor im Speicher, der dann auf eine Diskette geschrieben werden kann. "buf" zeigt auf einen 512-Byte-Puffer mit beliebigem Inhalt. "serialno" ist die Seriennummer, die der Bootsektor erhalten soll. Wenn "serialno" größer oder gleich $01000000 ist, wird eine zufällige Seriennummer erzeugt. "disktype" gibt die Art der Diskette an:

0 40 Tracks, einseitig (180K)
1 40 Tracks, zweiseitig (360K)
2 80 Tracks, einseitig (360K)
3 80 Tracks, zweiseitig (720K)

Wenn "disktype" –1 ist, werden die in "buf" vorhandenen Informationen nicht überschrieben. Wenn "execflag" den Wert 1 hat, wird der Bootsektor ausführbar gemacht, bei 0 nicht. Bei –1 wird er so gelassen, wie er ist.

$13 _flopver(buf, filler, devno, sectno, trackno, sideno, count) [20]

buf: Langwort
filler: Langwort
devno: Wort
sectno: Wort
trackno: Wort
sideno: Wort
count: Wort

Überprüft den Zustand von Sektoren, indem sie einfach gelesen werden. "buf" zeigt auf einen 1024 Bytes langen Puffer an Wortadresse, in den die Daten geschrieben werden. "filler" ist ein unbenutztes Langwort. "devno" ist die Laufwerksnummer (0 oder 1), "sectno" die Sektornummer (normalerweise 1 – 9), "trackno" die Tracknummer, "sideno" die Diskettenseite (0 oder 1). "count" gibt an, wie viele Sektoren überprüft werden sollen (nicht mehr, als sich auf einem Track befinden). Ein negativer Rückgabewert in D0 zeigt einen Fehler an. Nach der Ausführung befindet sich in "buf" eine Liste der zerstörten Sektoren, die mit einer Null abgeschlossen wird.

$14 scrdmp() [2]

Druckt eine Bildschirm-Hardcopy aus.

$15 cursconf(function, operand) [6]

function: Wort
operand: Wort

Stellt den Cursor ein. Zulässige Werte für "function" sind:

0 Cursor abschalten
1 Cursor einschalten
2 Cursor blinkend
3 Cursor nicht blinkend
4 Cursor-Blink-Intervall auf "operand" setzen
5 Cursor-Blink-Intervall zurückgeben

Das Cursor-Blink-Intervall wird von der Bildschirmfrequenz abgeleitet (50, 60 oder 70 Hz).

$16 settime(datetime) [6]
datetime: Langwort
Setzt Datum und Zeit im Tastaturprozessor. In "datetime" wird die Zeit im unteren, das Datum im oberen Wort angegeben. Das Format entspricht dem der GEMDOS-Funktionen Tsetdate() und Tsettime().

$17 gettime() [2]
Liefert Zeit und Datum aus dem Tastaturprozessor zurück (Zeit im unteren Wort, Datum im oberen Wort).

$18 bioskeys() [2]
Stellt die Standardtabellen zur Umrechnung von Tastaturcodes in ASCII wieder her.

$19 ikbdws(cnt,ptr) [8]
cnt: Wort
ptr: Langwort
Überträgt eine Zeichenkette zum Tastaturprozessor. "cnt" gibt die Anzahl der zu übertragenden Zeichen minus 1 an, "ptr" die Adresse der Zeichenkette.

$1A jdisint(intno) [4]
intno: Wort
Schaltet Interrupt Nummer "intno" auf dem MFP 68901 ab.

$1B jenabint(intno) [4]
intno: Wort
Schaltet Interrupt Nummer "intno" auf dem MFP 68901 ein.

$1C giaccess(data, regno) [6]
data: Byte
regno: Wort
Liest oder schreibt in Register des Soundchips. "regno" ist die Registernummer (0 – 15). Normalerweise wird gelesen; Wenn der Wert geschrieben werden soll, muß zusätzlich Bit 7 in "regno" gesetzt sein. "data" ist der hineinzuschreibende Wert.

$1D offgibit(bitno) [4]
bitno: Wort
Setzt das Bit "bitno" im PORT A-Register auf null.

$1E ongibit(bitno) [4]
bitno: Wort
Setzt das Bit "bitno" im PORT A-Register auf eins.

$1F xbtimer(timer, control, data, vec) [12]
timer: Wort
control: Wort
data: Wort
vec: Langwort
Schreibt in die Register der Timer A, B, C oder D. "timer" ist die Timer-Nummer (entsprechend 0, 1, 2 oder 3), "control" der Wert für das Kontrollregister, "data" der Wert für das Datenregister des Timers. "vec" ist der Vektor für den Timer-Interrupt.

$20 dosound(ptr) [6]
ptr: Langwort
Spielt im Hintergrund Töne ab; der Programmzähler des Sound-Programms wird auf "ptr" gesetzt. "ptr" zeigt auf eine Reihe von Tonbefehlen, die byteweise organisiert sind.

Befehle $00 bis $0F erhalten ein Byte-Argument, das entsprechend in das Soundchip-Register 0 – 15 geschrieben wird. $80 erhält ein Byte-Argument, das in ein internes Register gespeichert wird. $81 hat drei Byte-Argumente. Das erste Byte steht für ein Soundchip-Register, in das der Wert des internen Registers geladen wird. Das zweite Byte ist eine vorzeichenbehaftete Zahl, die zum Inhalt des internen Registers addiert wird, und der dritte Wert ist der Endwert des internen Registers. Die Anweisung wird ausgeführt, bis das interne Register den Endwert annimmt. $82 – $FF erhalten Ein-Byte-Parameter. Ist der Parameter null, dann ist die Tonausgabe beendet. Andernfalls wird der Parameter als Anzahl von 1/50 Sekunden interpretiert, die gewartet werden, bis die nächsten Werte interpretiert werden.

$21 setprt(config) [4]
config: Wort
Setzt oder liest das Drucker-Konfigurationsbyte. Wenn "config" den Wert –1 hat, wird der aktuelle Wert zurückgegeben, sonst wird das Byte gesetzt. Die einzelnen Bits haben folgende Bedeutung:

Bit	Wert 0	Wert 1
0	Matrix	Typenrad
1	Farbdrucker	nur schwarz/weiß
2	ATARI-Drucker	Epson-kompatibler Drucker
3	Entwurfsschrift	NLQ
4	Centronics	RS232
5	Endlospapier	Einzelblatt

$22 kbdvbase() [2]
Gibt einen Zeiger auf eine Struktur zurück, die Vektoren für verschiedene Routinen enthält, die für die Zusammenarbeit mit dem Tastaturprozessor sorgen. Die Tabelle ist folgendermaßen aufgebaut:

tab:	midivec	MIDI-Eingabe
tab+4:	vkbderr	Tastatur-Fehler
tab+8:	vmiderr	MIDI-Fehler
tab+12:	statvec	Tastaturprozessor Status
tab+16:	mousevec	Maus-Routinen
tab+20:	clockvec	Uhrzeit-Routine
tab+24:	joyvec	Joystick-Routine

Jeder dieser Vektoren ist mit der Adresse einer Betriebssystemroutine vorbelegt. "midivec" zeigt auf eine Routine, die die über die MIDI-Schnittstelle empfangenen Daten (Byte in D0) in den MIDI-Puffer schreibt.

"vkbderr" und "vmiderr" werden aufgerufen, wenn bei der Tastatur- oder MIDI-Eingabe der Puffer überläuft.

"statvec", "mousevec", "clockvec" und "joyvec" zeigen auf Routinen, die die vom Tastaturprozessor abgeschickten Datenpakete verarbeiten, wenn ein Tastatur-Status-, Maus-, Uhrzeit- oder Joystick-Event auftritt. Diesen Routinen wird in A0 ein Zeiger auf das empfangene Paket überreicht. Wenn Sie hier eigene Routinen installieren wollen, so müssen diese mit einem RTS abgeschlossen sein und dürfen nicht länger als eine Millisekunde Ausführungszeit benötigen.

$23 kbrate(initial, repeat) [6]
initial: Wort
repeat: Wort
Setzt oder liest die Tasten-Wiederholrate. Parameter mit dem Wert –1 werden ignoriert. Wenn eine Taste gedrückt wird, so wird zuerst die Zeit "initial" gewartet, bevor die Wiederholfunktion einsetzt; dann wird mit dem Intervall "repeat" wiederholt. Die Intervalle werden von der 50-Hz-Frequenz abgeleitet.

$24 _prtblk() [2]
Betriebssysteminterne Funktion.

$25 vsync() [2]
Wartet bis zum nächsten Vertical Blank Interrupt und kehrt dann zu-

rück. Diese Funktion ist nützlich, um etwa Grafikoperationen mit der Bildschirmfrequenz zu synchronisieren.

$26 supexec(codeptr) [6]
codeptr: Langwort
Führt ein Unterprogramm an der Adresse "codeptr" im Supervisormodus aus. Beendet wird der Code mit RTS. Im Unterprogramm dürfen keine GEMDOS-, BIOS- oder XBIOS-Aufrufe ausgeführt werden.

$27 puntaes() [2]
Wenn die AES-Routinen im RAM stehen, werden sie mit dieser Funktion entfernt, und der dadurch freigewordene Speicherplatz steht frei zur Verfügung. Wenn das AES sich nicht im Speicher befindet, kehrt die Funktion zurück. Wenn das AES allerdings entfernt werden kann, führt diese Funktion einen Neustart des Systems durch, nachdem das AES entfernt worden ist.

Die GEM-Aufrufe

Zweifellos ist die Programmierung einer GEM-Anwendung in Maschinensprache eine heikle Sache, zumal die Effizienz der Maschinensprache sich bei den doch recht zeitaufwendigen GEM-Routinen kaum auswirkt. Für den Fall, daß Sie Ihre Assemblerprogramme unter GEM laufen lassen wollen, rate ich Ihnen deshalb dazu, Maschinensprache in eine Hochsprache einzubinden und die GEM-Aufrufe in einer Sprache wie C, Pascal oder Modula II zu programmieren. Wenn Sie nicht die Absicht haben, GEM-Routinen in Assembler aufzurufen, können Sie dieses Kapitel überspringen. Ohnehin ist es nur im Zusammenhang mit einer umfassenderen GEM-Dokumentation verständlich.

Da es aber auch einmal sinnvoll sein kann, die GEM-Funktionen direkt von Maschinensprache aus aufzurufen, soll an dieser Stelle der Parameterübergabemechanismus erklärt werden. Natürlich ist es nicht Ziel dieses Buches, eine Einführung in die GEM-Programmierung zu geben; hier soll nur auf die speziellen Bedürfnisse des Assemblerprogrammierers eingegangen werden, die in der Literatur zu GEM oft nicht beachtet werden.

Es gibt einige geringe Unterschiede zwischen den Aufrufkonventionen von VDI und AES. Das Prinzip ist jedoch das gleiche: Die Parameter werden in globalen Feldern übergeben. Diese Felder werden vom Programm selbst angelegt; deshalb werden den GEM-Routinen beim Aufruf die Adressen dieser Felder übergeben.

Wichtig ist, daß ein GEM-Programm am Anfang den nicht benötigten Speicherplatz freigibt, da GEM einigen Platz zum Manövrieren braucht. Wie dies im einzelnen funktioniert, wird in Kapitel 2, Abschnitt "Organisation von ATARI ST-Programmen" beschrieben.

AES-Aufrufe (Application Environment Services)

Zu jedem AES-Aufruf gehören insgesamt 7 Felder:

- Der Control-Block (control array)
  Für den Kontrollblock sollten 12 Bytes reserviert werden. Er enthält folgende Informationen (jeweils 16-Bit-Werte):

Contrl	op_code, Funktionsnummer der gewünschten AES-Routine
Contrl+2	Anzahl der Worte, die im Intin-Feld übergeben werden
Contrl+4	Anzahl der Worte, die die Funktion im Intout-Feld zurückliefert
Contrl+6	Anzahl der Langworte, die im Addrin-Feld übergeben werden
Contrl+8	Anzahl der Langworte, die die Funktion im Addrout-Feld zurückliefert

- Das Global-Feld
  Jede unter GEM laufende Anwendung sollte genau ein GLOBAL-Feld haben. Hier befinden sich Verwaltungsinformationen der GEM-Routinen, die teilweise von der GEM-Implementierung abhängen, teilweise beim appl_init-Aufruf initialisiert werden. Es ist für den Programmierer kaum von Bedeutung. Reservieren Sie aber 30 Bytes für dieses Feld.

- Das Int_in-Feld
  In diesem Feld werden jene Eingabeparameter abgelegt, die Wortlänge haben (Integer). Deren Anzahl wird in control+2 vermerkt.

- Das Int_out-Feld
  Hier legt die AES-Funktion ihre Ausgabeparameter ab. Deren Anzahl schreibt sie in Contrl+4.

- Das Addr_in-Feld
  Hier werden Eingabeparameter abgelegt, die Langwort-Format haben (Adressen). Die Anzahl steht in Contrl+6.

- Das Addr_out-Feld
  Hier werden Adressen abgelegt, die die AES-Funktion an das aufrufende Programm zurückgibt. Die Anzahl kann man Contrl+8 entnehmen.

- Der Parameterblock
  Dieses Feld dient nur dazu, die anderen Felder dem AES zugänglich zu machen. Zu jedem Feld ist die Adresse als Langwort angegeben. Die Belegung ist folgendermaßen:

params	Adresse des Contrl-Feldes
params + 4	Adresse des Global-Feldes
params + 8	Adresse des Int_in-Feldes
params + 12	Adresse des Int_out-Feldes
params + 16	Adresse des Addr_in-Feldes
params + 20	Adresse des Addr_out-Feldes

Wenn die VDI-Routinen von Assembler aus aufgerufen werden sollen, muß zunächst einmal Platz für die sieben Felder reserviert werden. Nehmen Sie dabei 12 Worte für den Control-Block und 15 Worte für das Global-Feld. Die Länge der Int_in-, Int_out-, Addr_in- und Addr_out-Felder ist nicht festgelegt, sie hängt von den Funktionen ab, die aufgerufen werden sollen. 256 Bytes für jedes der vier Felder sollten aber in allen Fällen reichen.

Der Control-Block muß vom aufrufenden Programm jedesmal vollständig belegt werden. In einer ausführlichen Dokumentation zu GEM sollten Sie für jede Funktion den "Opcode" (Funktionsnummer) und die Anzahl der Ein- und Ausgabeparameter finden.

Natürlich müssen der Int_in- und Addr_in-Block entsprechend den Erfordernissen der gewünschten Funktion belegt werden. Die Int_out- und Addr_out-Felder brauchen hingegen nicht speziell initialisiert zu werden.

Nun zum eingentlichen GEM-Aufruf: Alle GEM-Funktionen werden mit TRAP #2 aufgerufen; vorher muß sich jedoch die Adresse des Parameterblocks im Register D1 befinden. Mit dieser Angabe kann sich die GEM-Routine über mehrere Zeiger zu den eigentlichen Parametern "durchhangeln".

Im unteren Wort von D1 wird ein spezieller Code verlangt, der das VDI oder AES anspricht. Für AES ist es $C8 (dezimal 200). Eine Stack-Korrektur ist nicht erforderlich, da ja keine Parameter auf dem Stack abgelegt werden. Nachdem Control-, Int_in- und Addr_in-Felder belegt worden sind, sähe also die Aufrufsequenz so aus:

```
        MOVE.L  #Aespara,D1  * Adresse des Parameterblocks
        MOVE.W  #$C8,D0      * wir wollen AES
        TRAP    #2           * Einsprung ins GEM
```

Die dazugehörigen Felder werden etwa so reserviert:

```
        DATA                 * jetzt initialisierte Daten
Aespara
        DC.L    Control      * Adresse des Control-Blocks
        DC.L    Global       * "          " Global-Feldes
        DC.L    Int_in       * "          " Int_in-Feldes
        DC.L    Int_out      * "          " Int_out-Feldes
        DC.L    Addr_in      * "          " Addr_in-Feldes
        DC.L    Addr_out     * "          " Addr_out-Feldes

        BSS                  * im BSS Platz reservieren
Control DS.W    12
Global  DS.W    15
Int_in  DS.W    128
Int_out DS.W    128
Addr_in DS.W    128
Addr_out DS.W   128
```

Nach diesem Aufruf können etwaige Ausgabewerte im Int_out- und Addr_out -Feld ausgelesen werden.

Wie in allen Programmiersprachen gilt, daß der erste AES-Aufruf ein appl_init() sein muß.

Das VDI (Virtual Device Interface)

Zu einem VDI-Aufruf gehören 6 Felder:

- Die Int_in- und Int_out-Felder haben die gleiche Funktion wie beim AES

- Pts_in ersetzt das Addr_in-Feld des AES. Statt einer Adresse wird hier ein Punktkoordinaten-Paar für grafische Operationen hineingeschrieben. Da jede Koordinate ein Wort beansprucht, ist ein Eintrag immer noch 2 Worte, also 4 Bytes lang.

- Pts_out ersetzt entsprechend Addr_out; hier werden Punktekoordinaten vom VDI zurückgegeben.

- Contrl entspricht dem Control-Feld beim AES. Es ist folgendermaßen belegt:

Contrl	Opcode (Funktionsnummer des VDI-Aufrufs)
Contrl + 2	Anzahl der Punkte in Pts_in
Contrl + 4	Anzahl der Punkte in Pts_out (Ausgabewert)
Contrl + 6	Anzahl der Worte in Int_in
Contrl + 8	Anzahl der Worte in Int_out (Ausgabewert)
Contrl + 10	Subfunktionsnummer (wird selten verwendet, nur z.B. bei Escapes)
Contrl + 12	"Device-Handle", Geräte-Identifikation. Sie wird beim ersten VDI-Aufruf "v_openwk" zurückgegeben.
ab Contrl+14	Ausgabewerte je nach Funktion

– Parameterblock
Hier sind die Adressen der oben genannten Felder verzeichnet, und zwar in folgender Reihenfolge:

Params	Adresse des Contrl-Blocks
Params + 4	Adresse des Int_in-Feldes
Params + 8	Adresse des Int_out-Feldes
Params + 12	Adresse des Pts_in-Feldes
Params + 16	Adresse des Pts_out-Feldes

Da die maximal gebrauchte Größe der einzelnen Felder wieder von den aufgerufenen Funktionen abhängt, dimensionieren Sie am besten alle Felder mit 256 Bytes (für alle Fälle).

Der Aufruf erfolgt ähnlich wie beim AES: Zuerst müssen Contrl-Int_in- und Pts_in-Feld entsprechend der Funktion vorbelegt werden. Die als Ausgabewerte im Contrl-Block gekennzeichneten Worte brauchen Sie natürlich nicht vorzubelegen.

Dann erfolgt der eigentliche Aufruf: Die Adresse des Parameterblocks wird ins Register D1 geladen, und in das untere Wort von D0 kommt die Identifikationsnummer des VDI: $73 oder dezimal 115. Der Einsprung erfolgt wieder mit TRAP #2. Danach können etwa auftretende Ausgabewerte in den entsprechenden Feldern ausgelesen werden.

Hier also die Aufrufsequenz, nachdem die Eingabe-Felder vorbelegt wurden, und gleich dazu die Dimensionierung der Felder:

```
            MOVE.L    #Vdipara,D1 * Adresse des Parameterblocks
            MOVE.W    #$73,D0     * ein VDI-Aufruf, bitte
            TRAP      #2          * Einsprung ins GEM
            BSS                   * im BSS Platz reservieren
Contrl      DS.W      128
Int_in      DS.W      128
```

```
   Int_out  DS.W     128
   Pts_in   DS.W     128
   Pts_out  DS.W     128
            DATA                    * jetzt initialisierte Daten
Vdipara
            DC.L      Contrl        * Adresse des Contrl-Blocks
            DC.L      Int_in        * " "         Int_in-Feldes
            DC.L      Pts_in        * " "         Pts_in-Feldes
            DC.L      Int_out       * " "         Int_out-Feldes
            DC.L      Pts_out       * " "         Pts_out-Feldes
```

Bei den VDI-Aufrufen gilt zu beachten, daß als erstes immer mit v_openwk oder v_openvwk ein "handle" reserviert wird.

Das folgende Programmbeispiel zeigt ein minimales GEM-Programm, das eine Alert-Box auf den Bildschirm bringt. Für Beispielprogramme sind Alert-Boxen nun einmal besonders beliebt, da GEM die Verwaltung einer Alert-Box völlig eigenständig ausführt.

Zunächst wird der vom Programm nicht unmittelbar benötigte Speicherplatz freigegeben. Bis zum Label "init" entspricht der Programmcode genau dem in Kapitel 2, Abschnitt "Organisation von ATARI ST-Programmen" angegebenen. Dann folgt ein "appl_init"-Aufruf, der das Programm bei GEM anmeldet. Als nächstes wird die AES-Funktion "graf_handle" ausgeführt, die dem Programm eine handle-Nummer zurückgibt, die für viele weitere AES-Aufrufe gebraucht wird und deshalb vom Programm an einer sicheren Stelle abgelegt wird. Nun wird der virtuelle Bildschirm geöffnet. Dazu dient der VDI-Aufruf "v_opnvwk".

Nun ist endlich die Initialisierung abgeschlossen, und das eigentliche Programm kann starten. Mit dem AES-Opcode für "alert_box" wird eine Alert Box auf den Bildschirm gebracht. Nach der Rückkehr dieser Routine wird je nach dem Rückgabewert verzweigt: Wurde Knopf 1 (Ja) angewählt, dann wird der Vorgang wiederholt, andernfalls wird das Programm einfach mittels TERM beendet.

```
*************************************************************
* GEM.S                                                     *
* Ein Beispiel für den Aufruf von GEM-Routinen in Assembler *
* Bringt eine Alert-Box auf den Bildschirm                  *
*************************************************************
*
* zuerst die Speicherfreigabe, damit GEM genügend Platz hat
start:
          move.l   4(sp),a5    * Basepageadresse in A5
          move.l   12(a5),d0   * Länge des Textsegments...
          add.l    20(a5),d0   * + Länge des Datensegments...
          add.l    28(a5),d0   * + Länge des BSS-Segments...
```

```
          add.l   #$1100,d0  * + 4K (=$1000) Für den Stack
*                            * + 256 (=$100) Bytes für die
                             * Basepage
          move.l  a5,d1      * neuer SP = Basepageadresse...
          add.l   d0,d1      * + berechnete Länge...
          and.l   #-2,d1     * auf gerade Adresse abrunden
          move.l  d1,sp      * in den Stackpointer damit
          move.l  d0,-(sp)   * Länge des reservierten Bereichs
          move.l  a5,-(sp)   * Anfangsadresse des Bereichs
          clr     -(sp)      * überflüssiger Parameter (Dummy)
          move.w  #$4a,-(sp) * GEMDOS-Funktion Setblock
          trap    #1          * Aufruf des GEMDOS
          add.l   #12,sp      * Stack wiederherstellen
*
init:     move    #10,contrl     * appl_init()
          clr     contrl+2       * keine Eingabeworte
          move    #1,contrl+4    * ein Ausgabewort
          clr     contrl+6       * keine Eingabeadressen
          clr     contrl+8       * keine Ausgabeadressen
          bsr     aes            * AES-Aufruf
*
          move    #77,contrl     * graf_handle()
          clr     contrl+2       * keine Eingabeworte
          move    #5,contrl+4    * 5 Ausgabeworte
          clr     contrl+6       * keine Eingabeadressen
          clr     contrl+8       * keine Ausgabeadressen
          bsr     aes            * AES-Aufruf
          move    intout,handle  * handle-Nummer merken
*
          move    #100,contrl    * v_opnvwk()
          clr     contrl+2       * keine Eingabepunkte
          clr     contrl+4       * keine Ausgabepunkte
          move    #11,contrl+6   * 11 Eingabeworte
          clr     contrl+8       * keine Ausgabeworte
          move    handle,contrl+12 * handle-Nummer angeben
          lea     intin,a0       * Im INTIN-Feld...
          move    #9,d0          * 10 Worte mit 1 füllen
initloop  move    #1,(a0)+       *
          dbra    d0,loop        *
          move    #2,(a0)        * elftes Wort mit 2
          bsr     vdi            * VDI-Aufruf
*
* hier geht es richtig los
loop      move    #52,contrl     * alert_box()
          move    #1,contrl+2    * ein Eingabewort
          move    #1,contrl+4    * ein Ausgabewort
          move    #1,contrl+6    * eine Eingabeadresse
          clr     contrl+8       * keine Ausgabeadresse
          move    #1,intin       * 1. Knopf als Standard
          move.l  #alert,addrin  * Text der Alert-Box
          jsr     aes            * ins AES
          move    intout,d0      * Feldnummer holen
          cmp     #1,d0          * noch einmal?
          beq.s   loop           * ja!
```

```
          clr     -(sp)          * TERM
          trap    #1             * Schluß!
*
* Hier wird ein AES-Aufruf durchgeführt
aes       move.l  #Aespara,d1    * Adresse des Parameterblocks
          move    #$c8,d0        * wir wollen AES
          trap    #2             * Einsprung ins GEM
          rts                    *
*
* Hier wird ein VDI-Aufruf durchgeführt
vdi       move.l  #Vdipara,d1    * Adresse des Parameterblocks
          move    #$73,d0        * wir wollen AES
          trap    #2             * Einsprung ins GEM
          rts                    *

          DATA
*
* Hier wird der Text für die Alert Box definiert
alert     DC.B    "[2][Wollen Sie diese Alert-Box|"
          DC.B    "noch einmal sehen ?]"
          DC.B    "[Ja| Nein ]",0
* Jetzt kommen die Parameterblöcke
Aespara   DC.L    contrl
          DC.L    global
          DC.L    intin
          DC.L    intout
          DC.L    addrin
          DC.L    addrout
Vdipara   DC.L    contrl
          DC.L    intin
          DC.L    ptsin
          DC.L    intout
          DC.L    ptsout

          BSS
handle    DS.W    1
*
* nur noch Platz für die Felder reservieren
contrl    DS.W    12
global    DS.W    15
intin     DS.W    128
intout    DS.W    128
addrin    DS.W    128
addrout   DS.W    128
ptsin     DS.W    128
ptsout    DS.W    128
          END
```

Die Line-A-Routinen

Die Line-A-Routinen stellen eine für Assemblerprogrammierer recht interessante Gruppe von Funktionen dar. Sie bilden das Grundgerüst der GEM-Rou-

tinen und bieten in der Hauptsache grundlegende Grafikroutinen wie Punkt setzen, Punkt abfragen, Linien ziehen und Sprite-Operationen.

Die Line-A-Routinen bilden einen Kurzschluß um die GEM-Routinen herum, denn durch den Aufruf dieser Routinen kann man sich die aufwendigen Initialisierungen und Parameterübergabemechanismen des GEM sparen, erreicht aber trotzdem ohne viel Aufwand Grafikoperationen. Außerdem sind die Line-A-Routinen merklich schneller als der Umweg über entsprechende VDI-Routinen.

Angesprochen werden die Line-A-Routinen mit den Opcodes $A00x, wobei x eine Hexadezimalziffer zwischen 0 und E ist. Jeder dieser Opcodes löst eine Exception aus, die mit einem Sprung durch den Vektor ab $28 bearbeitet wird.

Zunächst zur Parameterübergabe: Die Parameter für Line-A-Routinen werden teilweise in Int_in und Pts_in-Feldern des VDI übergeben (Beschreibung siehe Kapitel 4, "GEM-Aufrufe"), teilweise in einem globalen Variablen-Bereich. Die Adresse dieses Bereichs steht nicht fest; man kann allerdings mit der Funktion $A000 (Initialisierung) einen Zeiger darauf erhalten. Da sich unter den Line-A-Variablen auch Zeiger auf die 5 VDI-Felder befinden, sollte es kein Problem sein, die Parameter an die richtigen Stellen zu schreiben.

Wie beim BIOS gilt, daß die Line-A-Aufrufe die Register D0 – D2 und A0 – A2 verändern können.

Hier nun die einzelnen Line-A-Routinen:

$A000 Adressen der Datenbereiche holen

Dieser Opcode muß als erster ausgeführt werden, wenn die Line-A-Routinen benutzt werden sollen. Es werden verschiedene Adressen in den Registern D0 – D2 und A0 – A2 übergeben. Von Interesse ist hauptsächlich der Inhalt von D0 und A0, der die Anfangsadresse der Line-A-Variablen darstellt. A1 zeigt auf eine Struktur von drei Adressen, die ihrerseits Zeiger auf die Startadressen der Systemzeichensätze sind; in A2 steht die Startadresse einer Tabelle der Adressen sämtlicher Line-A-Routinen.

$A001 Put_pixel

Ein Pixel wird an die Koordinaten gesetzt, die durch Pts_in (X-Wert) und Pts_in+2 (Y-Wert) festgelegt werden. Die Nummer der Farbe wird in Int_in übergeben. Der Farbindex kann je nach Auflösung 0 – 1, 0 – 3 oder 0 – 15 sein. Im Y-Bereich werden unzulässige Koordinaten ignoriert, im X-Bereich

wird jedoch keine Überprüfung vorgenommen und der Punkt entsprechend an eine falsche Position gesetzt.

$A002 Get_pixel

Der Farbwert eines Pixels, dessen Koordinaten in Pts_in und Pts_in + 2 abgelegt werden, wird im Register D0 zurückgegeben.

$A003 Line

Diesmal befinden sich die Parameter ausschließlich in den Line-A-Variablen. Es wird eine Linie von den Koordinaten x1, y1 nach x2, y2 gezogen. Die Farbe wird je nach Auflösung nur durch _fg_bp1, durch _fg_bp1 und _fg_bp2 oder durch _fg_bp1-_fg_bp4 bestimmt (siehe Beschreibung der Line-A-Variablen). Beim Linienziehen richtet sich diese Funktion außerdem nach dem Muster der Linie _ln_mask und dem Schreibmodus _wrt_mod. Bereichsüberschreitungen werden genauso behandelt wie bei Put_pixel.

$A004 Horizontal Line

Diese Funktion zieht eine horizontale Linie von x1, y1 nach x2, y1. Es werden die gleichen Parameter wie bei Line ($A003) berücksichtigt, nur daß anstatt des Linienmusters _ln_mask das durch die Variablen _patptr und _patmsk bestimmte Füllmuster benutzt wird. _patptr zeigt auf eine bestimmte Anzahl Worte, in denen das Muster codiert ist; _patmsk gibt die Anzahl der Worte des Füllmusters minus eins an. Jeder Bildschirmzeile wird eines dieser Worte zugeordnet; so wird in Zeile n eine Linie mit dem Muster des Wortes n modulo (_patmsk + 1) gezogen. Werden mehrere dieser Linien untereinandergesetzt, dann läßt sich das vollständige Füllmuster erkennen.

$A005 Filled Rectangle

Diese Funktion erzeugt ein gefülltes Rechteck. Die Koordinaten der linken oberen und rechten unteren Ecke werden in x1, y1 und x2, y2 angegeben. Das Füllmuster wird wieder durch _patmsk und die Werte, auf die _patptr zeigt, bestimmt. Ansonsten finden die gleichen Variablen Verwendung wie bei Line oder Horizontal Line. Hinzugekommen ist allerdings die Möglichkeit des Clipping: Man kann in _XMN_CLIP, _YMN_CLIP, _XMX_CLIP und _YMX_CLIP zwei Koordinatenpaare eintragen, die einen rechteckigen Ausschnitt des Bildschirms festlegen, auf den die Grafikoperation beschränkt ist.

Zusätzlich muß noch die Variable _CLIP auf 1 gesetzt werden, womit angezeigt wird, daß Clipping angewandt werden soll.

$A006 Filled Polygon

Mit dieser Funktion können beliebig geformte Flächen gefüllt werden. Die Eckpunkte des Polygons werden im PTSIN-Array eingetragen, wobei das letzte Koordinatenpaar dem ersten entsprechen sollte, damit eine geschlossene Fläche entsteht. Die Anzahl der Koordinatenpaare wird in CONTRL(1) angegeben. Es gelten die üblichen Variablen für Schreibmodus, Farbe, Muster und Clipping. Zu beachten ist, daß dieser Opcode nur eine Zeile füllt; die zu füllende Y-Coordinate wird in _y1 angegeben. Um also eine Fläche vollständig zu füllen, muß die Funktion in einer Schleife mit allen Y-Werten vom kleinsten zum größten aufgerufen werden.

$A007 Bitblt

Nur von interner Bedeutung.

$A008 Textblt

Mit diesem Opcode können Texte in allen Variationen auf den Bildschirm gebracht werden. Da der Aufruf jedoch sehr kompliziert ist, würde die Beschreibung hier zu weit führen.

$A009 Show Mouse

Dieser Opcode schaltet den Mauszeiger ein. Dabei hat der Wert in INTIN(0) eine spezielle Bedeutung: Normalerweise unterhält die Routine zur Verwaltung des Mauszeigers einen Zähler, der bei jedem Aufruf von Show Mouse um eins erhöht, bei jedem Aufruf von Hide Mouse ($A00A, nächste Funktion) um eins verringert wird. Der Mauszeiger wird nur eingeschaltet, wenn der Zähler größer als null ist; wenn man den Mauszeiger also zweimal einschaltet, muß man ihn auch zweimal abschalten. Mit INTIN(0) kann dieses Verhalten übergangen werden; ist dieser Wert null, dann wird der Mauszeiger auf jeden Fall eingeschaltet bzw. bei Hide Mouse abgeschaltet.

$A00A Hide Mouse

Schaltet den Mauszeiger ab. Siehe $A009, Show Mouse.

$A00B Transform Mouse

Mit dieser Funktion können Sie sich ihren eigenen Mauszeiger programmieren. Sämtliche Parameter werden als Worte im INTIN-Array abgelegt:

INTIN+6	Maskenfarbindex, normalerweise 0
INTIN+8	Datenfarbindex, normalerweise 1
INTIN+10	bis INTIN+40 16 Worte Maskenform
INTIN+42	bis INTIN+72 16 Worte Mauszeigerdaten

An jeder Stelle, an der in den Mauszeigerdaten ein Bit gesetzt ist, erscheint ein schwarzer Punkt auf dem Bildschirm (wie bei allen Grafikoperationen erscheint das niederwertigste Bit rechts, das höchstwertige links). Wenn ein Bit in den Maskendaten gesetzt ist, dann erscheint statt des Hintergrunds ein weißer Punkt. Auf diese Art wird der weiße Rahmen um den normalen Mauszeiger erzeugt.

$A00C Undraw Sprite

Dieser Opcode dient dazu, ein mit $A00D, Draw Sprite, gezeichnetes Sprite wieder zu löschen. Dazu wird in A2 die Adresse des Puffers angegeben, in den der Hintergrund von der Funktion Draw Sprite gerettet wurde.

$A00D Draw Sprite

Mit dieser Funktion wird eine 16 x 16 Pixel große Figur auf dem Bildschirm gezeichnet. D0 und D1 enthalten die X- und Y-Position eines ausgezeichneten Punktes des Sprites auf dem Bildschirm, Hot Spot genannt. A0 muß auf den Sprite Definition Block zeigen, in dem sämtliche Informationen über das Sprite stehen:

Wort 1	X-Offset von der linken oberen Ecke zum Hot Spot
Wort 2	Y-Offset von der linken oberen Ecke zum Hot Spot
Wort 3	Format-Flag
Wort 4	Hintergrund-Farbnummer des Sprites
Wort 5	Vordergrund-Farbnummer des Sprites
Wort 6 – 37	32 Worte Sprite-Daten

Bei den Sprite-Daten erscheint beginnend mit der obersten Zeile immer abwechselnd zuerst ein Wort Vordergrundmuster, dann ein Wort Hintergrundmuster. Das Format-Flag bestimmt nun, wie Vordergrund- und Hintergrunddaten zu interpretieren sind. Ist es null, dann wird im sogenannten VDI-Format gearbeitet:

Vg.	Hg.	Ergebnis
0	0	Der Hintergrund erscheint
0	1	Die Farbe aus Wort 4 erscheint
1	0	Die Farbe aus Wort 5 erscheint
1	1	Die Farbe aus Wort 5 erscheint

Hat das Format-Flag hingegen den Wert 1, dann wird das XOR-Format verwendet:

Vg.	Hg.	Ergebnis
0	0	Der Hintergrund erscheint
0	1	Die Farbe aus Wort 4 erscheint
1	0	Hintergrundpixel wird mit Vg.-Bit EXKLUSIV-ODER-verknüpft
1	1	Die Farbe aus Wort 5 erscheint

Das Register A2 muß die Adresse eines Puffers enthalten, in dem der Hintergrund abgespeichert wird. Der Puffer braucht für jede Bitebene 64 Bytes; zusätzlich werden in jedem Fall 10 Bytes für Verwaltungsinformationen benutzt. Man braucht also für hohe Auflösung 74 Bytes, für mittlere 138 und für niedrige 266.

$A00E Copy Raster Form

Dieser Opcode kann Bildschirmausschnitte kopieren. Er ist jedoch in erster Linie von GEM-interner Bedeutung, etwa für die Fensterverwaltung.

Die Line-A-Variablen

Da die Lage der Line-A-Variablen im Speicher nicht festliegt, wird zu jeder Variable nur die Adreßdistanz zum Anfang des Variablenblocks angegeben. Mit dem Opcode $A000 kann man die Anfangsadresse der Variablen erhalten. Sofern es sich nicht um Adressen handelt, haben alle Variablen Wortlänge (16 Bit).

Dist.	Name	Beschreibung
0	v_planes	Anzahl der Bitebenen des Grafikmodus (1 für 640x400, 2 für 640x200, 4 für 320x200)
4	contrl	Zeiger auf das Contrl-Feld (Langwort)
8	intin	Zeiger auf das Int_in-Feld (Langwort)
12	ptsin	Zeiger auf das Pts_in-Feld (Langwort)

16	intout	Zeiger auf das Int_out-Feld (Langwort)
20	ptsout	Zeiger auf das Pts_out-Feld (Langwort)
24	_fg_bp1	Farbwert der Bitebene 0 (alle drei Auflösungen)
26	_fg_bp2	Farbwert der Bitebene 1 (620x200, 320x200)
28	_fg_bp3	Farbwert der Bitebene 2 (nur 320x200)
30	_fg_bp4	Farbwert der Bitebene 3 (nur 320x200) _fg_bp1 bis _fg_bp4 werden vom Opcode Line ($A003) benutzt. Sie sollten nur 1 oder 0 sein.
34	_ln_mask	Linienmuster bei Line ($A003) etwa %1111 1111 1111 1111: durchgezogene Linie. %1111 0000 1111 0000: unterbrochene Linie Beachten Sie, daß das oberste Bit von _ln_mask sich immer oben links bei der Linie auswirkt.
36	_wrt_mod	Schreibmodus für Linienziehen und Musterzeichnen 0: Überschreiben Der Hintergrund wird vom gezeichneten Muster (Linie) einfach ersetzt. 1: Transparent Nur gesetzte Bits des gezeichneten Musters werden überschrieben; sind Bits im Schreibmuster gelöscht, so bleibt dort der Hintergrund erhalten (ODER-Verknüpfung). 2: XOR-Modus Das einzuzeichnende Muster wird mit dem Hintergrund durch EXKLUSIV-ODER verknüpft. 3: Invers Transparent Sind Bits im zu schreibenden Muster gelöscht, so überschreiben sie den Hintergrund; sind sie gesetzt, so bleibt der Hintergrund erhalten.
38	_x1	x-Koordinate des ersten Punktes, etwa bei Line
40	_y1	erste y-Koordinate
42	_x2	zweite x-Koordinate
44	_y2	zweite y-Koordinate
46	_patptr	Zeiger auf die Füllmuster-Daten
50	_patmsk	Anzahl der Worte des Füllmusters minus 1
54	_CLIP	0: Clipping abgeschaltet Ungleich 0: Clipping eingeschaltet
56	_XMN_CLIP	linker Rand des sichtbaren Bereiches
58	_YMN_CLIP	oberer Rand des sichtbaren Bereiches
60	_XMX_CLIP	unterer Rand des sichtbaren Bereiches
62	_YMX_CLIP	rechter Rand des sichtbaren Bereiches

Kapitel 5

Einige nützliche Routinen

In diesem Kapitel sollen einige Routinen zur Ein- und Ausgabe von Zahlen und Zeichenketten vorgestellt werden. Die eigentlichen Routinen sind zum Einbinden in Ihre Assemblerprogramme mit Hilfe eines Editors durchaus geeignet; sie können natürlich Ihren eigenen Bedürfnissen beliebig angepaßt werden (etwa an eine Bildschirmmasken-Eingabe). Experimentieren Sie ruhig mit einigen Veränderungen, schließlich lernt man die Assemblerprogrammierung in erster Linie durch Praxis. Außerdem könnte die eine oder andere Routine noch einige Verbesserungen gebrauchen.

Um die Routinen austesten zu können, werden sie in den meisten Fällen in Form eines vollständigen Programms abgedruckt. Beim Einbinden in eigene Programme werden Sie natürlich nur die eigentliche Routine verwenden.

Wenn Sie einen Macroassembler benutzen, ist sicher auch eine Implementierung dieser Routinen als Macros interessant.

Ausgabe von Zeichenketten

Die Ausgabe von Zeichenketten (Strings) auf den Bildschirm wird man in praktisch jedem Programm brauchen, etwa für Menüs oder Hinweise. Glücklicherweise gibt es dafür schon eine Betriebssystemfunktion, nämlich PRINTLINE, GEMDOS-Funktion Nummer 9. Als Parameter erhält sie die Adresse der Zeichenkette, die ausgegeben werden soll. Die Zeichenkette wird durch ein Null-Byte beendet. Die Ausgabe funktioniert so, als ob jedes Zeichen des Strings bis zum Nullbyte mit der Funktion CONOUT (Nummer 2) ausgegeben würde. Das heißt, daß auch sämtliche Kontrollzeichen und Escape-Sequenzen richtig verarbeitet werden. So können Sie mit dieser Routine auch den Cursor positionieren, an- und abschalten, den Bildschirm löschen und noch einige andere nützliche Dinge tun.

Die hier vorgestellte Routine tut eigentlich nichts anderes, als die Parameterübergabe etwas zu vereinfachen. Es wird einfach die Adresse des auszugebenden Strings in D0 übergeben. Wie üblich werden durch den GEMDOS-Aufruf D0 und A0 verändert.

```
*************************************************************
* PRINT.S                                                   *
* Routine zur Ausgabe von Strings                           *
* Die Adresse des Strings wird in D0 übergeben              *
*************************************************************

start     move.l  #botschaft,d0     * Adresse laden
          bsr     print             * und aufrufen
          move    #8,-(sp)          * GEMDOS CNECIN
          trap    #1                *
          addq.l  #2,sp             *
          clr     -(sp)             * GEMDOS-TERM
          trap    #1                *

print     move.l  d0,-(sp)          * Adresse des Strings
                                    * ist Parameter
          move    #9,-(sp)          * GEMDOS Funktion PRINTLINE
          trap    #1                * Aufruf
          addq.l  #6,sp             *
          rts                       * das war's schon

          DATA
botschaft DC.B "CPU MC68000 damaged. Please contact Motorola.",13,10,0
*
* 13 ist der ASCII-Code für Carriage Return, 10 für Newline
* beides zusammen gibt das, was sonst immer das Ergebnis der
* Return-Taste ist.
          END
```

Der Testaufruf gibt eine "wichtige" Botschaft aus und wartet dann noch auf einen Tastendruck, damit man die Botschaft auch dann mitbekommt, wenn man das Programm vom Desktop aus startet.

Übrigens müssen Sie hier darauf achten, daß unter TOS immer zwei ASCII-Zeichen notwendig sind, einen vollständigen Zeilenrücklauf durchzuführen: Carriage Return (ASCII 13) setzt den Cursor an den Anfang der aktuellen Zeile, und Linefeed (ASCII 10) führt einen Zeilenvorschub aus. Die Reihenfolge, in der diese Steuerzeichen ausgegeben werden, spielt dabei keine Rolle. Dieses Verhalten gilt für den Bildschirm genauso wie für den Drucker.

Das nächste Programmbeispiel gibt eine Zeichenkette auf dem Drucker aus. Für diese Aufgabe gibt es keine so freundliche Betriebssystemroutine. Statt dessen kann der Aufruf WRITE verwendet werden, der normalerweise eine beliebige Anzahl von Bytes in eine Datei schreibt. Um die Datei zu identifizieren, gibt man einen sogenannten Datei-Handle (wörtlich: Griff) an, der beim Öffnen der Datei vom Betriebssystem geliefert wird. Das Besondere daran ist nun folgendes: Überall, wo das Betriebssystem einen Datei-Handle erwartet,

kann man auch die Konsole, den Drucker oder die RS232-Schnittstelle ansprechen, indem man eine bestimmte Handle-Nummer im Bereich 0-5 angibt. Die Zuordnung ist folgende:

Handle	Name	Gerät
0	CON:	Console-Eingabe (Tastatur mit Echo auf dem Bildschirm)
1	CON:	Console-Ausgabe (Bildschirm)
2	AUX:	RS-232-Schnittstelle
3	PRN:	Drucker

Wie Sie sehen, findet diese Möglichkeit in den Gerätebezeichnungen ihre Entsprechung. Tatsächlich kann man in vielen Fällen, wo ein Dateiname verlangt wird, auch eines dieser Geräte ansprechen.

Der WRITE-Systemaufruf bekommt als Parameter zunächst den Datei-Handle (in unserem Fall 3 für den Drucker)und danach die Anzahl der auszugebenden Bytes und die Adresse des ersten Bytes. Letzteres ist bei uns einfach die Adresse der Zeichenkette, doch die Anzahl der auszugebenden Zeichen, also die Länge der Zeichenkette, muß erst festgestellt werden. Dabei geht unsere Routine davon aus, daß die Zeichenkette wie üblich mit einem Nullbyte beendet wird. Ein Zähler in D0 wird so lange erhöht, bis ein Nullbyte erreicht ist. Dann erst erfolgt der WRITE-Aufruf.

Leider gibt der WRITE-Aufruf bei der Ausgabe auf den Drucker keine Meldung, ob er erfolgreich ist. Tatsächlich wird D0 durch diesen Aufruf gar nicht verändert. So hat man also keine Möglichkeit, festzustellen, ob der Drukker überhaupt empfangsbereit ist – wenn er es nicht ist, gibt es nach etwa 30 Sekunden einen Timeout, und WRITE kehrt unverrichteter Dinge zurück. Aus diesem Grund sollte man vor dem ersten Ansprechen des Druckers immer mit der Funktion PRTOUT STAT (GEMDOS $11) testen, ob der Drucker überhaupt existiert und zur Zusammenarbeit bereit ist.

```
*************************************************************
* PRINTER.S                                                 *
* Routine zur Ausgabe von Strings                           *
* Die Adresse des Strings wird in D0 übergeben              *
*************************************************************

start     move.l  #botschaft,d0       * Adresse laden
          bsr     printer             * und aufrufen
          clr     -(sp)               * GEMDOS TERM
          trap    #1                  *
*
*
```

```
printer   move.l  d0,a0              * Adresse der Zeichenkette
          clr.l   d0                 * D0: Länge der Zeichenkette
p_loop    tst.b   0(a0,d0.w)         * Testen auf Nullbyte
          beq.s   print2             * gefunden -> weiter
          addq    #1,d0              * nächstes Byte testen
          bra.s   p_loop             * und nächster Schleifendurchlauf
print2    move.l  a0,-(sp)           * 3. Parameter: Adresse des Strings
          move.l  d0,-(sp)           * 2. Parameter: Länge in Bytes
          move    #3,-(sp)           * handle für Drucker
          move    #$40,-(sp)         * GEMDOS-Funktion WRITE
          trap    #1                 * Aufruf
          lea     12(sp),sp          * Stack korrigieren
          rts                        * das war's

          DATA
botschaft DC.B    "Hallo Drucker!",13,10,13,10,0
*
* 13 ist der ASCII-Code für Carriage Return, 10 der von Line Feed
          END
```

Eingabe von Zeichenketten

Irgendwann wird Ihr Programm den Benutzer auch nach irgend etwas fragen wollen. Dazu ist eine Routine zur Eingabe von Zeichenketten notwendig.

Zunächst einmal gibt es auch dafür eine GEMDOS-Funktion: READLINE (Nummer 10) bekommt als Parameter die Adresse eines Eingabepuffers. Die eingegebenen Zeichen werden erst ab dem dritten Byte des Eingabepuffers abgelegt, während den ersten beiden Bytes (0 und 1) eine besondere Bedeutung zukommt: Das erste Byte enthält die maximale Anzahl der Zeichen, die eingegeben werden darf, das zweite gibt nach der Ausführung die tatsächliche Anzahl der eingegebenen Zeichen an. Diese Zahl findet sich auch in D0. READLINE bietet die folgenden einfachen Editiermöglichkeiten:

<Ctrl><B>/<Backspace>	letztes Zeichen löschen
<Ctrl><I>/<Tab>	Tabulator
<Ctrl><J>/<Ctrl><M>/ <Return>/<Enter>	Eingabe beenden
<Ctrl><R>	Eingabe in neuer Zeile ausgeben
<Ctrl><U>	ungültig, in neuer Zeile beginnen
<Ctrl><X>	Zeile löschen, Cursor an Zeilenanfang
<Ctrl><C>	Programm beenden (!)

Das Zeichen für Carriage Return ist nicht Teil der eingegebenen Zeichenkette.

Bevor ich diese Routine kritisiere, sollten Sie sich erst einmal das Beispiel-Listing ansehen. Die Routine "input" vereinfacht die Aufrufsequenz: Die Adresse des Puffers wird in D0 übergeben, die maximale Anzahl der Zeichen in D1; sie braucht nicht noch extra im ersten Byte des Puffers zu stehen – dafür sorgt "input" selbst. Nach der Ausführung wird die Anzahl der tatsächlich gelesenen Zeichen in D0 zurückgegeben. Wie üblich wird durch den GEMDOS-Aufruf in "input" A0 in Mitleidenschaft gezogen.

"input" übernimmt es auch, an den eingegebenen String ein Nullbyte anzuhängen, damit die Eingabe wie alle Strings behandelt werden kann. Man muß nur darauf achten, daß die Anfangsadresse des Strings um 2 größer ist als die Anfangsadresse des Puffers.

Um zu zeigen, wie man mit "input" umgeht, ist ein vollständiges Programm abgedruckt, das auch die "print"-Routine aus dem vorigen Programmbeispiel verwendet.

```
*************************************************************
* STRINGIN.S                                                *
* input - Routine zur Eingabe von Strings                   *
* Die Adresse des Strings wird in D0 übergeben              *
*************************************************************
start     move.l  #prompt,d0      * Adresse der Meldung laden
          bsr.s   print           * und aufrufen
          move.l  #antwort,d0     * Eingabe einlesen
          move    #30,d1          * höchstens 30 Zeichen
          bsr.s   input           *
          move.l  #hallo,d0       * ersten Text ausgeben
          bsr.s   print           *
          move.l  #antwort+2,d0   * Eingegebenen Text ausgeben
          bsr.s   print           *
          move.l  #hallo2,d0      * und noch etwas anhängen
          bsr.s   print           *
          move    #8,-(sp)        * GEMDOS CNECIN: auf Taste
                                  * warten
          trap    #1              *
          addq.l  #2,sp           *
          clr     -(sp)           * GEMDOS TERM
          trap    #1              *

print     move.l  d0,-(sp)        * Adresse des Strings ist
                                  * Parameter
          move    #9,-(sp)        * GEMDOS-Funktion PRINTLINE
          trap    #1              * Aufruf
          addq.l  #6,sp           *
          rts                     * das war's schon

input     move.l  d0,a0           * in ein Adreßregister
          move.b  d1,(a0)         * maximale Zeichenanzahl
                                  * eintragen
```

```
            move.l  d0,-(sp)        * Adresse des reservierten
                                    * Bereichs
            move    #10,-(sp)       * GEMDOS-Funktion READLINE
            trap    #1              * Aufruf
            addq.l  #2,sp           * Nur Funktionsnummer weg
                                    * vom Stack
            move.l  (sp)+,a0        * alten D0-Wert wiederholen
            clr.b   2(a0,d0.w)      * Nullbyte anhängen
            rts                     *

            DATA
prompt      DC.B    "Hallo, mein Name ist Jack.",10,13
            DC.B    "Was ist Ihrer?",10,13,0
hallo       DC.B    10,13,"Hallo, ",0
hallo2      DC.B    " !",10,13,0
antwort     DS.B    33
            END
```

Die Funktion READLINE können Sie auch ausprobieren, indem Sie den Kommandointerpreter COMMAND.TOS laden. Dort ist genau diese Eingaberoutine verwendet worden.

COMMAND.TOS ist unseres Wissens das einzige größere Programm, das READLINE für die Zeicheneingabe benutzt – und das aus gutem Grund. Denn READLINE hat einige Nachteile:

- Durch Tastenkombinationen wie <Ctrl><R> und <Ctrl><U> kann der ganze Bildschirm durcheinandergebracht werden

- Andere Ctrl-Kombinationen werden nicht korrekt behandelt.

- Es besteht jederzeit die Gefahr eines Programmabbruchs mit<Ctrl><C>, was besonders unangenehm ist, wenn das Programm Routinen installiert hat, die es bei der Beendigung wieder rückgängig machen sollte (etwa Interruptroutinen).

- READLINE akzeptiert keine Umlaute und kein "ß".

Zusammenfassen kann man das Ganze so: Sie werden kaum umhinkommen, sich für größere Anwendungen selbst eine Eingaberoutine zu schreiben. Deshalb will ich Ihnen hier eine mögliche Eingaberoutine vorstellen.

"input2" wird genauso aufgerufen wie "input": Die Adresse des Puffers wird in D0 übergeben, die maximale Anzahl der Zeichen in D1. Im Puffer gibt es jedoch keine reservierten Werte, die eingegebenen Zeichen werden direkt ab dem Anfang des Puffers hineingeschrieben. Abgeschlosssen wird der eingege-

bene String mit einem Nullbyte. Sie sollten also immer für ein Byte mehr Platz vorsehen, als tatsächlich eingegeben werden kann.

Beim Aufruf von "input2" wird zunächst die Länge des Puffers auf 0 gesetzt, da noch keine Zeichen eingelesen worden sind. Die eigentliche Eingabe wird mit der GEMDOS-Funktion CNECIN vorgenommen, die ein Zeichen von der Tastatur liest, aber nichts auf dem Bildschirm ausgibt. Als erstes wird untersucht, ob die Return-Taste gedrückt wurde (ASCII 13). In diesem Fall ist die Eingabe beendet, und es wird nur noch ein Null-Byte an die eingegebene Zeichenkette angehängt. Dann wird überprüft, ob es sich um die Backspace-Taste (ASCII 8) handelt. Wenn dies der Fall ist, wird die Länge des Puffers um eins verringert, sofern überhaupt noch Zeichen im Puffer sind. Um das Zeichen auch auf dem Bildschirm zu löschen, muß Backspace in die Zeichenfolge "Backspace, Leerzeichen, Backspace" übersetzt werden, da das ASCII-Zeichen Backspace zwar den Cursor zurückbewegt, aber kein Zeichen löscht. Bei allen anderen ankommenden ASCII-Zeichen wird zunächst beim Label "in_normal" noch ein Filter dazwischengeschaltet, der alle Steuerzeichen aussondert. Erst wenn das Zeichen diese Tests bestanden hat, wird es ausgegeben, in den Puffer eingetragen und die Pufferlänge um eins erhöht – es sei denn, die maximal zulässige Anzahl von Zeichen war schon erreicht. An diesem Punkt wird wieder zum Anfang der Eingabeschleife verzweigt.

Natürlich können Sie den Filter, der hier nur alle Steuerzeichen hinauswirft, beliebig für eigene Verwendungen erweitern. So ist es etwa bei der Eingabe von Zahlen sinnvoll, nur Ziffern zu akzeptieren und alles andere zu ignorieren.

```
*************************************************************
* STRINGI2.S                                                *
* input2 - konfortablere Routine zur Eingabe von Strings    *
* Die Adresse des Strings wird in D0 übergeben              *
*************************************************************

start   move.l  #prompt,d0      * Adresse der Meldung laden
        bsr.s   print           * und aufrufen
        move.l  #antwort,d0     * Eingabe einlesen
        move    #30,d1          * höchstens 30 Zeichen
        bsr.s   input2          *
        move.l  #hallo,d0       * ersten Text ausgeben
        bsr.s   print           *
        move.l  #antwort,d0     * Eingegebenen Text ausgeben
        bsr.s   print           *
        move.l  #hallo2,d0      * und noch etwas anhängen
        bsr.s   print           *
        move    #8,-(sp)        * GEMDOS CNECIN: auf Taste
                                * warten
```

```
          trap    #1          *
          addq.l  #2,sp       *
          clr     -(sp)       * GEMDOS TERM
          trap    #1          *

print     move.l  d0,-(sp)    * Adresse des Strings ist Parameter
          move    #9,-(sp)    * GEMDOS-Funktion PRINTLINE
          trap    #1          * Aufruf
          addq.l  #6,sp       *
          rts                 * das war's schon

************************************************************
*    Registerbelegung                                      *
*    D0.L      nur Übergabewert                            *
*    D1.W      maximale Anzahl der Zeichen (inklusive)     *
*    D2.W      aktuelle Anzahl der Zeichen im Puffer       *
*    A1        Adresse des Puffers                         *
************************************************************
input2    move.l  d0,a1       * in ein Adreßregister
          clr     d2          * Anzahl der gelesenen Zeichen = 0
in_loop   move    #8,-(sp)    * CNECIN Funktionscode
          trap    #1          * zum GEMDOS
          addq.l  #2,sp       *
          cmp     #13,d0      * Return-Taste?
          beq.s   in_end      * ja, Ende der Eingabe
          cmp     #8,d0       * Backspace?
          bne.s   in_normal   * nein, weiter
          tst     d2          * sind überhaupt Zeichen da?
          beq.s   in_loop     * nein, also ignorieren
          pea     bs          * sonst Backspace-String ausgeben
          move    #9,-(sp)    * PRINTLINE
          trap    #1          *
          addq.l  #6,sp       *
          subq    #1,d2       * ein Zeichen weniger
          bra.s   in_loop     *
in_normal cmp     #32,d0      * Steuerzeichen (ASCII<32)?
          bcs.s   in_loop     * ja, ignorieren
          cmp     d1,d2       * Zeichenanzahl erreicht?
          beq.s   in_loop     * ja, ignorieren
          move.b  d0,0(a1,d2)
                              * Zeichen ablegen
          addq    #1,d2       * Zeichenzähler erhöhen
          move    d0,-(s)     * Zeichen ausgeben
          move    #2,-(sp)    * CONOUT
          trap    #1          *
          addq.l  #4,sp       *
          bra.s   in_loop     * nächstes Zeichen
in_end    clr.b   0(a1,d2.w)* Nullbyte anhängen
          rts                 *

bs        dc.b    8,' ',8,0
* Statt Backspace wird Backspace, Leerzeichen und Backspace
* ausgegeben, da Backspace nur den Cursor bewegt, aber kein
* Zeichen löscht
```

```
          DATA
prompt    DC.B    "Hallo, mein Name ist Jack.",10,13
          DC.B    "Was ist Ihrer?",10,13,0
hallo     DC.B    10,13,"Hallo, ",0
hallo2    DC.B    " !",10,13,0
antwort   DS.B    33
          END
```

Ausgabe von hexadezimalen Zahlen

Das folgende Programm gibt ein in D0 übergebenes Langwort als Hexadezimalzahl auf dem Bildschirm aus.

Die Ausgabe einer hexadezimalen Zahl ist eigentlich recht einfach. Es geht nur darum, sich die Bits von oben bis unten jeweils in Vierergruppen vorzunehmen und das entsprechende Zeichen auszugeben. Die Vierergruppen erhält man jeweils dadurch, daß man den ursprünglichen Wert des Operanden um 28, 24, 20 ... 0 Stellen nach rechts schiebt und alle Bits außer den vier untersten ausmaskiert. Die Umrechnung der Werte 0 – 15 (oder 0 – F) in die entsprechenden ASCII-Werte erfolgt am einfachsten mit einer Tabelle. Dann müssen die einzelnen Zeichen nur noch ausgegeben werden. Es sollte nicht schwierig sein, diese Routine für die Ausgabe eines Wortes umzuschreiben, falls Sie so etwas brauchen.

```
*************************************************************
* PRHEX.S                                                   *
* Routine zum Ausgeben eines Langwortes als Hexazimalzahl   *
*                                                           *
* Die auszugebende Zahl wird im Register D0 übergeben       *
*                                                           *
* Registerbelegung:                                         *
* D0.L  nur Übergabewert, wird sonst von CONIN verändert    *
* D1.L  noch auszugebender Restwert                         *
* D2.W  zum Berechnen des auszugebenden Zeichens            *
* D3.W  Stellenzähler                                       *
*************************************************************
* Hauptprogramm
start     move.l  #$AFFE1987,d0 * beliebige Zahl zum Testen
          bsr     printhex      * und Routine aufrufen
          move    #8,-(sp)      * CONOUT-Funktionscode
          trap    #1            * ins GEMDOS
          addq.l  #2,sp         * Stack aufräumen
          clr     -(sp)         * TERM-Funktionscode
          trap    #1            * Schluß jetzt!

printhex  move    #28,d3        * Stellenzähler initialisieren
          move.l  d0,d1         * Wert in D1 retten
h_loop    move.l  d1,d2         * Ziffer wird in d2 berechnet
          lsr.l   d3,d2         * um 4*n Stellen nach links
```

```
                                    * schieben
            and     #$F,d2          * nur die unteren 4 Bits
                                    * bleiben
            lea     ziffern,a0      * Adresse der Ziffern-Tabelle
            move.b  0(a0,d2.w),d2   * ASCII-Wert laden
            move    d2,-(sp)        * auf den Stack damit
            move    #2,-(sp)        * CONOUT-Funktionscode
            trap    #1              *
            addq.l  #4,sp           *
            subq    #4,d3           * jetzt um 4 Bits weniger
                                    * verschieben
            bpl     h_loop          * nächste Ziffer
            rts                     * fertig, wenn Stellen-
                                    * zähler < 0

ziffern     DC.B    "0123456789ABCDEF"
            END
```

Eingabe von hexadezimalen Zahlen

Die hier vorgestellte Routine liest nicht selbst Zeichen von der Tastatur, sondern erwartet als Parameter in D0 die Adresse einer Zeichenkette, aus der eine Hexadezimalzahl gelesen werden soll. Diese Maßnahme läßt Ihnen die Möglichkeit offen, jede beliebige Routine für die Eingabe der Zeichenkette zu verwenden. Die Zahl wird als beendet betrachtet, sobald ein Zeichen gefunden wird, das nicht in eine Hexadezimalzahl gehört. Nach der Ausführung steht in D0 die gelesene Zahl und in D1 die Anzahl der verarbeiteten Zeichen.

Und so arbeitet die Routine: Zunächst werden die Zahl (D0) und der Zeichen-Index (D1) auf Null gesetzt. Dann folgt der Anfang der Schleife: Es wird immer ein Zeichen eingelesen und der ASCII-Wert in den Wert der Hexadezimalziffer umgewandelt, wobei auch Kleinbuchstaben berücksichtigt werden. Hier wird die Eigenschaft des ASCII-Codes benutzt, daß alle Ziffern von 0 bis 9 in aufsteigender Reihenfolge hintereinanderliegen; um das Zeichen für eine der Ziffern also in ihren Wert umzurechnen, braucht man nur den ASCII-Wert des Zeichens für 0 abzuziehen.

Genauso liegen auch alle Buchstaben des Alphabets in der üblichen Reihenfolge hintereinander; um die Großbuchstaben A – F in die Ziffernwerte 10 – 15 umzurechnen, zieht man den um 10 verringerten ASCII-Wert des Zeichens A ab. Wenn das gelesene Zeichen keine gültige Hexadezimalziffer ist, dann wird die Ziffernfolge als beendet betrachtet und die bisher gelesene Zahl zurückgegeben. Ist es jedoch eine gültige Ziffer, dann wird die bisher gelesene Zahl um eine Hexadezimalstelle nach links verschoben, was einer Multiplikation mit 16

oder einem Verschieben um 4 Bits nach links entspricht, und der Wert der neu gelesenen Ziffer wird addiert. Dann wird nur noch der Zeichenindex erhöht und zum Anfang der Schleife zurückgesprungen.

Aus der Struktur des Programms kann man leicht ablesen, daß eine Null zurückgeliefert wird, wenn schon das erste gelesene Zeichen keine gültige Ziffer darstellt.

Übrigens handelt es sich diesmal nicht um ein vollständiges Programm, es wird nur eine Routine abgedruckt.

```
*************************************************************
* INPHEX.S                                                  *
* Einlesen einer Hexadezimalzahl (kein vollständiges Programm)*
* Die Adresse einer Zeichenkette wird in D0 überrreicht;    *
* nach der Ausführung steht in D0.L die Zahl und in D1.W    *
* die Anzahl der gelesenen Zeichen                          *
*                                                           *
* Registerbelegung                                          *
* D0    bisher gelesene Zahl                                *
* D1    Anzahl der gelesenen Zeichen                        *
* D2    aktuzelles Zeichen                                  *
* A0    Adresse der Zeichenkette                            *
*************************************************************

inphex    move.l  d0,a0          * Adresse der Zeichenkette in
                                 * A0
          clr.l   d0             * Bisher gelesene Zahl = 0
          clr     d1             * Anzahl der gelesenen
                                 * Zeichen = 0
ih_loop   move.b  0(a0,d1.w),d2  * aktuelles Zeichen einlesen
          cmp.b   #'0',d2        * größer/gleich '0'?
          bcs.s   ih_end         * Nein, Ende
          cmp.b   #'9',d2        * kleiner/gleich '9'?
          bhi.s   ih_af          * nein, noch auf A-F testen
          sub.b   #'0',d2        * ASCII '0'-'9' auf Zahl
                                 * umrechnen
          bra.s   ih_mul16       * Test auf A-F überspringen
ih_af     cmp.b   #'A',d2        * größer gleich 'A'?
          bcs.s   ih_end         * Nein, keine Ziffer
          cmp.b   #'F',d2        * kleiner/gleich 'F'?
          bhi.s   ih_aflow       * nein, noch auf 'a'-'f'
                                 * testen
          sub.b   #'A'-10,d2     * ASCII 'A'-'F' auf
                                 * 10-15 umrechnen
          bra.s   ih_mul16       * Test auf a-f überspringen
ih_aflow  cmp.b   #'a',d2        * größer gleich 'a'?
          bcs.s   ih_end         * Nein, keine Ziffer
          cmp.b   #'f',d2        * kleiner/gleich 'F'?
          bhi.s   ih_end         * nein, keine Ziffer
```

```
            sub.b   #'a'-10,d2          * ASCII 'a'-'f' auf 10-15
                                        * umrechnen
ih_mul16    lsl.l   #4,d0               * D0.L=D0.L*16  ->
                                        * D0.L=D0.L >> 4
            ext.w   d2                  * d2 von Byte- auf
                                        * Langwortbreite
            ext.l   d2                  *
            add.l   d2,d0               * D0.L = D0.L + neue
                                        * Ziffer
            addq    #1,d1               * ein Zeichen mehr
                                        * gelesen
            bra.s   ih_loop             *
ih_end      rts                         * fertig!
```

Ausgabe von Dezimalzahlen

Bei Dezimalzahlen muß man etwas anders vorgehen als bei Hexadezimalzahlen. Die Methode, die der Computer hier verwendet, entspricht der, die Sie verwenden würden, wenn Sie mit Bleistift und Papier eine Dezimalzahl in ein fremdes Zahlensystem verwandeln sollten. Für den Computer ist dabei das Dezimalsystem das "fremde" System, er ist ja im Binärsystem zu Hause. Sehen wir uns dazu beispielsweise einmal an, wie man die Dezimalzahl 1000 in eine Hexadezimalzahl umwandeln kann:

Die höchste Potenz von 16, die nicht größer als 1000 ist, ist 16 hoch 2, also 256. Wir fangen deshalb mit 256 an:

256	geht	3	mal in	1000	Rest	1000 – 3*256	=	232	Ziffer 3
16	geht	14	mal in	232	Rest	232 – 14*16	=	8	Ziffer E
1	geht	8	mal in	8	Rest	8 – 8*1	=	0	Ziffer 8

Wir erhalten 1000 = $3E8.

Die folgende Routine zur Ausgabe einer Dezimalzahl bekommt den auszugebenden Wert in D0 übergeben. Es werden die Register D0 – D5 und A0 verändert.

Um festzustellen, wie oft eine Zehnerpotenz in die noch zu verarbeitende Zahl hineingeht, wird jedesmal getestet, ob die Zehnerpotenz noch kleiner ist als die restliche Zahl. Ist dies der Fall, dann wird die Zehnerpotenz von der Restzahl abgezogen und dafür die aktuelle Ziffer um eins erhöht. Das geschieht so lange, bis die Restzahl tatsächlich kleiner ist als die Zehnerpotenz. Dann wird die errechnete Ziffer ausgegeben und der Rest mit der nächst niedrigeren Zehnerpotenz weiterbearbeitet.

Die einfachste Methode, die verschiedenen Zehnerpotenzen zu erhalten, ist eine Liste. Bei der Ausgabe eines Langwortes muß mit 10^9 = 1.000.000.000 begonnen werden, da die höchste in einem Langwort darstellbare Zahl etwa 4,29 Milliarden entspricht.

Einem Problem muß man noch Beachtung schenken: dem Entfernen von führenden Nullen. Hier wird das so gelöst, daß in D5 ein Flag eingerichtet wird, das angibt, ob schon eine andere Ziffer als 0 vorkam. Solange Nullen errechnet werden und dieses Flag noch nicht gesetzt ist, erfolgt keine Ausgabe, da es sich mit Sicherheit um führende Nullen handelt. Leider muß man hier noch auf einen Sonderfall eingehen, denn bei dieser Behandlung würde bei einer Null in D0 überhaupt nichts ausgegeben werden. Deshalb wird das Kriterium, daß nach dem Berechnen aller Ziffern das fragliche Flag noch immer nicht gesetzt ist, als Anzeichen dafür betrachtet, daß es sich um die Null handelt, und das ASCII-Zeichen "0" wird noch einmal extra ausgegeben.

```
*************************************************************
* PRDECP.S                                                  *
* Routine zum Ausgeben eines Langwortes als Dezimalzahl     *
* (nur positiv)                                             *
*                                                           *
* Die auszugebende Zahl wird im Register D0 übergeben       *
*                                                           *
* Registerbelegung:                                         *
* D0.L  nur Übergabewert, wird sonst von CONIN verändert    *
* D1.L  noch auszugebender Restwert                         *
* D2.W  Index in der Zehnerpotenzen-Tabelle                 *
* D3.L  Zehnerpotenz mit dem Stellenwert der aktuellen Ziffer*
* D4.W  Wert der aktuellen Ziffer                           *
* D5.B  Flag; 1 wenn schon eine andere Ziffer als 0         *
*       vorkam ,sonst 0                                     *
* A1    Adresse der Zehnerpotenzentabelle                   *
*************************************************************

* Hauptprogramm
start     move.l  #123456789,d0    * beliebige Zahl zum Testen
          bsr     printdec         * und Routine aufrufen
          move    #8,-(sp)         * CONOUT-Funktionscode
          trap    #1               * ins GEMDOS
          addq.l  #2,sp            * Stack aufräumen
          clr     -(sp)            * TERM-Funktionscode
          trap    #1               * Schluß jetzt!

printdec  clr.b   d5               * Nullen-Flag löschen
          move.l  #4*9,d2          * Index von 1.000.000.000
          lea     zehner,a1        * Adresse der Zehnerpoten-
                                   * zen-Tabelle
          move.l  d0,d1            * Wert in D1 retten

d_loop    move.l  0(a1,d2.w),d3    * Zehnerpotenz nach D3
```

```
          moveq   #48,d4          * Ziffer=ASCII(0)=48
d_verglei cmp.l   d3,d1           * Rest schon kleiner
                                  * als Zehnerpotenz?
          bcs.s   d_ausgabe       * ja, zur Ausgabe
          sub.l   d3,d1           * Rest=Rest-Zehnerpotenz
          addq    #1,d4           * Ziffer um eins erhöhen
          bra.s   d_verglei       * und noch einmal

d_ausgabe cmp.b   #48,d4          * Ziffer='0'
          bne.s   d_aus           * nein, normal ausgeben
          tst.b   d5              * Null schon ausgeben?
          beq.s   d_next          * nein, nächste Ziffer
d_aus     st      d5              * ab jetzt Nullen ausgeben
          move    d4,-(sp)        * Ziffer ausgeben
          move    #2,-(sp)        * CONOUT Funktionsnummer
          trap    #1              * und hinein ins GEMDOS
          addq.l  #4,sp           * immer schön aufräumen
d_next    subq    #4,d2           * Index verringern
          bpl.s   d_loop          * noch mehr Ziffern ausgeben
d_end     tst.b   d5              * Wurde schon eine Ziffer
                                  * ausgegeben?
          bne.s   d_rts           * ja, alles ok
          move    #'0',-(sp)      * nein, es muß sich um die 0
                                  * handeln
          move    #2,-(sp)        * CONOUT-Funktionsnummer
          trap    #1              * ins GEMDOS
          addq.l  #4,sp           *
d_rts     rts                     * Fertig wenn Index<0
* Tabelle der Zehnerpotenzen von 1 bis eine Milliarde
zehner    DC.L    1,10,100,1000,10000,100000,1000000
          DC.L    10000000,100000000,1000000000
          END
```

Diese Routine gibt Zahlen immer positiv aus. Die folgende Routine hingegen trägt dem Umstand Rechnung, daß man ein Langwort auch als Zweierkomplementzahl betrachten kann. Es handelt sich nur um eine kleine Abwandlung des vorherigen Programms: Zum Anfang der Routine wird getestet, ob die Zahl negativ ist. Ist das der Fall, dann wird das Minus-Zeichen (–) ausgegeben und die Zahl negiert, also in eine positive verwandelt. Der Rest entspricht der vorherigen Routine.

```
*************************************************************
* PRDECN.S                                                  *
* Routine zum Ausgeben eines Langworts als Dezimalzahl (+/-) *
*                                                           *
* Die auszugebende Zahl wird im Register D0 übergeben.      *
*                                                           *
* Registerbelegung:                                         *
* D0.L  nur Übergabewert, wird sonst von CONIN verändert    *
* D1.L  noch auszugebender Restwert                         *
* D2.W  Index in der Zehnerpotenzen-Tabelle                 *
```

```
* D3.L  Zehnerpotenz mit dem Stellenwert der aktuellen Ziffer *
* D4.W  Wert der aktuellen Ziffer                             *
* D5.B  Flag; 1 wenn schon eine andere Ziffer als 0 vorkam,   *
*       sonst 0                                               *
* A1    Adresse der Zehnerpotenzentabelle                     *
*************************************************************

* Hauptprogramm
start     move.l  #-123456789,d0     * negative Zahl zum Testen
          bsr.s   printdecn          * und Routine aufrufen
          move    #8,-(sp)           * CONOUT-Funktionscode
          trap    #1                 * ins GEMDOS
          addq.l  #2,sp              * Stack aufräumen
          clr     -(sp)              * TERM-Funktionscode
          trap    #1                 * Schluß jetzt!

printdecn clr.b   d5                 * Null-Flag löschen
          move.l  #4*9,d2            * Index von 1.000.000.000
          lea     zehner,a           * Adresse der Zehner-
                                     * potenzen-Tabelle
          move.l  d0,d1              * Wert in D1 retten
          bpl.s   d_loop             * Wert positiv -> nichts
                                     * weiter
          move    #$2d,-(sp)         * negativ, '-' ausgeben
          move    #2,-(sp)           * CONOUT-Funktionsnummer
          trap    #1                 * ins GEMDOS damit
          addq.l  #4,sp              *
          neg.l   d1                 * Operand positiv machen
d_loop    move.l  0(a1,d2.w),d3      * Zehnerpotenz nach D3
          moveq   #48,d4             * Ziffer=ASCII(0)=48
d_verglei cmp.l   d3,d1              * Rest schon kleiner als
                                     * Zehnerpotenz?
          bcs.s   d_ausgabe          * ja, zur Ausgabe
          sub.l   d3,d1              * Rest = Rest-Zehnerpotenz
          addq    #1,d4              * Ziffer um eins erhöhen
          bra.s   d_verglei          * und noch einmal

d_ausgabe cmp.b   #48,d4             * Ziffer="0"
          bne.s   d_aus              * nein, normal ausgeben
          tst.b   d5                 * Null schon ausgeben?
          beq.s   d_next             * nein, nächste Ziffer
d_aus     st      d5                 * ab jetzt Nullen ausgeben
        move    d4,-(sp)             * Ziffer ausgeben
          move    #2,-(sp)           * CONOUT-Funktionsnummer
          trap    #1                 * und hinein ins GEMDOS
          addq.l  #4,sp              * immer schön aufräumen
d_next    subq    #4,d2              * Index verringern
          bpl.s   d_loop             * noch mehr Ziffern aus-
                                     * geben
d_end     tst.b   d5                 * Wurde schon eine Ziffer
                                     * ausgegeben?
          bne.s   d_rts              * ja, alles ok
          move    #'0',-(sp)         * nein, es muß sich um die
                                     * 0 handeln
```

```
          move    #2,-(sp)          * CONOUT-Funktionsnummer
          trap    #1                * ins GEMDOS
          addq.l  #4,sp             *
d_rts     rts                       * Fertig wenn Index<0
* Tabelle der Zehnerpotenzen von 1 bis eine Milliarde
zehner    DC.L    1,10,100,1000,10000,100000,1000000
          DC.L    10000000,100000000,1000000000
          END
```

Es sollte keinen großen Aufwand machen, diese Routinen für die Ausgabe eines Wortes umzuschreiben. In diesem Fall müßte man mit der Zehnerpotenz 10000 beginnen, da in einem Wort Zahlen bis 65535 dargestellt werden können.

Eingabe von Dezimalzahlen

Um das Prinzip des Einlesens einer Dezimalzahl zu verstehen, erinnern wir uns, wie das bei einer Hexadezimalzahl funktioniert. Dort haben wir immer eine Ziffer eingelesen und ihren Wert berechnet, sofern es eine gültige Ziffer war. Daraufhin wurde der bisher gelesene Wert um eine Ziffer nach links verschoben, was in diesem Fall gerade einer Multiplikation mit 16 entsprach – eben weil 16 die Basis des Zahlensystems war. Zum Schluß wurde die neue Ziffer zum Ergebnis dieser Multiplikation (oder Verschiebung, wie man es eben betrachtet) addiert. Genau aus dem gleichen Grund werden wir bei Dezimalzahlen das bisherige Ergebnis jedesmal mit 10 multiplizieren.

Bei der folgenden Routine wird beim Aufruf in D0 die Adresse einer Zeichenkette übergeben, aus der die Zahl gelesen werden soll. Bei der Rückkehr steht in D0 die gelesene Zahl (Langwort) und in D1 die Anzahl der verarbeiteten Zeichen. Zwischendurch werden noch die Register D2 – D4 und A0 verändert.

Für die Multiplikation mit 10 muß man sich einen kleinen Trick einfallen lassen, da der MC68000 ja nur Worte mittels MULU multiplizieren kann, wir jedoch eine Langwortmultiplikation brauchen. Deshalb setzt man die Multiplikation mit 10 aus einer Multiplikation mit 8 und einer mit 2 zusammen – die sich ja direkt aus Verschiebeoperationen ergeben. Übrigens ist diese Methode mitunter auch bei Wort-Multiplikationen sinnvoll, da sie noch immer schneller ist als ein einziger MULU-Befehl. Logischerweise ist dieser Trick nur anwendbar, wenn einer der Operatoren eine Konstante ist.

Die Routine berücksichtigt auch ein führendes Minuszeichen, so daß der Benutzer bei großen Zahlen die Wahl hat, ob er sie lieber als positive oder negati-

ve Zahlen eingibt. So wird am Anfang der Routine getestet, ob das erste Zeichen ein Minus-Zeichen ist und ein Flag entsprechend gesetzt. Erst nach dem Einlesen der Ziffern wird darauf noch einmal zugegriffen: Ist es gesetzt, dann wird das Ergebnis negiert.

Diese Routine achtet nicht darauf, ob zu große Zahlen eingegeben werden; in diesem Fall stimmt das Ergebnis nur noch modulo 2^{32}.

```
*************************************************************
* INPDEC.S                                                  *
* Einlesen einer Dezimalzahl (kein vollständiges Programm)  *
* Die Adresse einer Zeichenkette wird in D0 überreicht;     *
* nach der Ausführung steht in D0.L die Zahl und in D1.W    *
* die Anzahl der gelesenen Zeichen                          *
*                                                           *
* Registerbelegung                                          *
* D0    bisher gelesene Zahl                                *
* D1    Anzahl der gelesenen Zeichen                        *
* D2    aktuelles Zeichen                                   *
* D3    zum Rechnen                                         *
* D4    Minus-Flag; gesetzt: Zahl ist negativ               *
* A0    Adresse der Zeichenkette                            *
*************************************************************

inpdec    move.l   d0,a0           * Adresse der Zeichenkette
                                   * in A0
          clr.l    d0              * bisher gelesene Zahl = 0
          clr      d1              * Anzahl der gelesenen
                                   * Zeichen = 0
          clr.b    d4              * Minus-Flag löschen
          move.b   (a0),d2         * erstes Zeichen auf '-'
                                   * testen
          cmp.b    #'-',d2         *
          bne.s    id_loop         * kein "-", weiter
          st       d4              * Minus-Flag setzen
          addq     #1,d1           * ein gelesenes Zeichen
id_loop   move.b   0(a0,d1.w),d2   * aktuelles Zeichen einle-
                                   * sen
          sub.b    #'0',d2         * ASCII auf Ziffernwert
                                   * umrechnen
          cmp.b    #9,d2           * größer als 9?
          bhi.s    id_end          * keine Ziffer
                                   * (ASCII 0-47 / 58-255)
          move.l   d0,d3           * D0.L = D0.L * 10 berech-
                                   * nen
          lsl.l    #1,d3           *      = D0.L*8 + D0.L*2
          lsl.l    #3,d0           *
          add.l    d3,d0           *
          ext.w    d2              * d2 von Byte- auf Lang-
                                   * wortbreite
          ext.l    d2              *
```

```
          add.l   d2,d0           * D0.L = D0.L + neue Ziffer
          addq    #1,d1           * ein Zeichen mehr gelesen
          bra.s   id_loop         *
id_end    tst     d4              * negative Zahl?
          beq.s   id_rts          * nein, Rückkehr
          neg.l   d0              * Zahl = -Zahl
id_rts    rts                     * fertig!
```

Die Langwortdivision

Bekanntlich sind Multiplikation und Division auf dem MC68000 nur für Worte implementiert. Bei der Multiplikation stellt das kein großes Problem dar, da man diese Operation mit größeren Einheiten auf die Wortmultiplikation zurückführen kann. Leider ist das bei der Division nicht so einfach – dort muß ein ganz anderer Algorithmus her, wenn man mit größeren Einheiten als der Division eines Langwortes durch ein Wort operieren will.

Die folgende Routine führt die Division eines Langwortes durch ein Langwort aus; sie kann auch leicht für noch längere Operanden erweitert werden.

Um den Algorithmus zu verstehen, betrachten wir zunächst einmal, eine schriftliche Division zweier Dezimalzahlen:

```
    52325 : 17 = 3077 Rest 16
-   51             3 * 17 = 51
  -------
     1325
-     0            0 * 17 = 0
    ------
     1325
-    119           7 * 17 = 119
    ------
      135
-     119          7 * 17 = 119
    ------
       16
```

In diesem Fall wird 5325 als Dividend bezeichnet, 17 als Divisor.

Die binäre Division geht ganz ähnlich vor sich:

```
   10101010 : 1101 = 01011 Rest 1
- 0000              0*1101
  --------
  10101010
- 1101              1*1101
```

```
   --------
    1000010
-   0000          0*1101
    -------
    1000010
-    1101         1*1101
    -------
       1110
-      1101       1*1101
    -------
           1
```

Wieder einmal ist die binäre Operation wesentlich einfacher durchzuführen als die dezimale: Der Divisor wird mit der höchsten Stelle des Operanden verglichen; ist der obere Teil des Dividenden größer als der Divisor, dann wird der Divisor an dieser Stelle vom Dividenden abgezogen und eine Eins im Ergebnis angerechnet. Andernfalls geschieht nichts weiter, und im Ergebnis erscheint eine Null. Das Ganze wird so lange wiederholt, bis die niederwertigste Stelle des Dividenden erreicht ist.

Der folgende Algorithmus geht im Prinzip genauso vor. Allerdings wurde ein kleiner Trick verwendet, der Befehle und Rechenzeit spart: Der Dividend teilt sich mit dem Ergebnis das gleiche Register, da beide Werte in jedem Durchlauf nach links verschoben werden müssen. Wenn alle 32 Bit abgearbeitet sind, ist der Dividend ganz aus dem Register herausgeschoben, und nur noch das Ergebnis steht dort.

Der Dividend wird in D0 überreicht, der Divisor in D1. Zurückgegeben wird in D0 das Ergebnis der Division (wie üblich abgerundet) und der Rest in D2. Der Divisor in D1 wird nicht verändert.

```
*************************************************************
* DIVI.S                                                    *
* Langwortdivision in Maschinensprache                      *
* Die Operanden werden in D0.L (Divisor) und D1.L (Dividend) *
* überreicht; Nach dem Aufruf steht in D0 das Ergebnis D0/D1 *
* und in D2 der Rest D0 mod D1                              *
*                                                           *
* Registerbelegung:                                         *
* D0: Operand 1 (Divisor) und Ergebnis                      *
* D1: Operand 2 (Dividend)                                  *
* D2: Teil des aktuellen Divisors bzw. Rest                 *
* D3: Zähler für 32 Bitstellen                              *
*************************************************************

divi      moveq   #0,d2              * Rest auf Null setzen
          moveq   #31,d3             * Stellenzähler
divi1     asl.l   #1,d0              * Operand und Rest
```

```
          roxl.l   #1,d2              * nach links schieben
          cmp.l    d2,d1              * Rest > Dividend?
          bhi.s    divi2              * ja, nichts weiter tun
          sub.l    d1,d2              * Rest=Rest-Dividend
          addq     #1,d0              * Ergebnis 1 erhöhen
divi2     dbf      D3,divi1           * 32mal wiederholen
          rts                         * fertig
```

Kapitel 6

Maschinennahe Programmierung

In diesem Kapitel stürzen wir uns Hals über Kopf in die Praxis. Es werden einige Beispiele gezeigt, wie Programme mit der Hardware des ATARI ST umgehen können – so ziemlich das einzige Gebiet, das allein der Assemblerprogrammierung vorbehalten bleibt. Natürlich ist dieses Kapitel weit davon entfernt, einen vollständigen Überblick über die Hardwaremöglichkeiten des ATARI ST zu vermitteln – dafür sind sie einfach zu umfangreich. Die folgenden Beispiele sollen nur mit einigen Grundlagen der maschinennahen Programmierung vertraut machen und Anregungen zu eigenen Experimenten bieten.

Setzen eines Punktes in hoher Auflösung

Eine elementare Grafikroutine ist die zum Setzen oder Löschen eines Pixels. Natürlich gibt es im Betriebssystem schon Routinen für diesen Zweck (etwa die Line-A-Routinen). Es ist jedoch in vielen Fällen nützlich, selbst auf den Grafikspeicher zugreifen zu können, da man so Routinen entwickeln kann, die schneller sind als die Betriebssystemroutinen, da sie nicht so flexibel sein müssen. Außerdem kann man auch Grafikoperationen implementieren, die nicht im Betriebssystem vorgesehen sind.

Bevor wir eine Routine zum Setzen oder Löschen eines Punktes entwickeln können, müssen wir zunächst einiges über den Aufbau des Bildschirmspeichers wissen.

Der Grafikspeicher des ATARI ST liegt direkt im Hauptspeicher und umfaßt in jeder Auflösungsstufe 32000 Bytes. Normalerweise wird der Bildschirmspeicher direkt vor dem Ende des physikalischen RAMs liegen, also bei einer 512K-Maschine von $78000 bis $78FFF, bei einer Megabyte-Maschine von $F8000 bis $FFFFF. Es ist allerdings kein guter Programmierstil, direkt eine dieser Adressen zu benutzen, da die Bildschirmadresse von einigen residenten Programmen – wie beispielsweise einer RAM-Disk – verschoben werden kann. Zu diesem Zweck gibt es zwei XBIOS-Funktionen mit den Namen PHYSBASE (Nummer 2) und LOGBASE (Nummer 3). Beide haben keine Parameter und liefern in D0 eine Adresse zurück. Erstere liefert dabei die Anfangsadresse des Bildschirmspeichers, den man in diesem Moment tatsächlich sieht, also die physikalische Bildschirmadresse. LOGBASE gibt die logische Bildschirmadresse an, auf der alle Grafikoperationen wie Zeichenausgabe und

Line-A-Routinen arbeiten. Im Normalfall sind beide identisch. Da jedoch beide Adressen getrennt verändert werden können, hat ein Programm die Möglichkeit, einen Bildschirm zu zeigen, während ein anderer für den Benutzer unsichtbar gerade aufgebaut wird.

Nun zunächst zur Organisation des Bildschirmspeichers für den höchstauflösenden Modus (640 x 400). Da in dieser Auflösung jeder Punkt nur zwei Farben, nämlich Schwarz oder Weiß, annehmen kann, entspricht jeder Pixel genau einem Bit. Dabei steht normalerweise 0 für einen weißen, 1 für einen schwarzen Punkt. 80 aufeinanderfolgende Bytes oder besser gesagt 40 Worte entsprechen einer Bildschirmzeile. Das Wort mit der niedrigsten Adresse wird am linken Bildschirmrand abgebildet, das mit der höchsten rechts. Innerhalb eines Wortes wird das höchstwertigste Bit links abgebildet, das niederwertigste rechts. Die Bildschirmzeilen werden nacheinander von oben nach unten abgespeichert. Somit werden die Worte genau in der Reihenfolge abgespeichert, in der sie für den Bildschirmaufbau synchron zur Bewegung des Elektronenstrahls abgetastet werden müssen. Abb. 6.1 verdeutlicht diese Zusammenhänge. Um also den Punkt links oben in der Ecke zu setzen, müßte Bit 15 des ersten Wortes des Bildschirmspeichers auf 1 gesetzt werden.

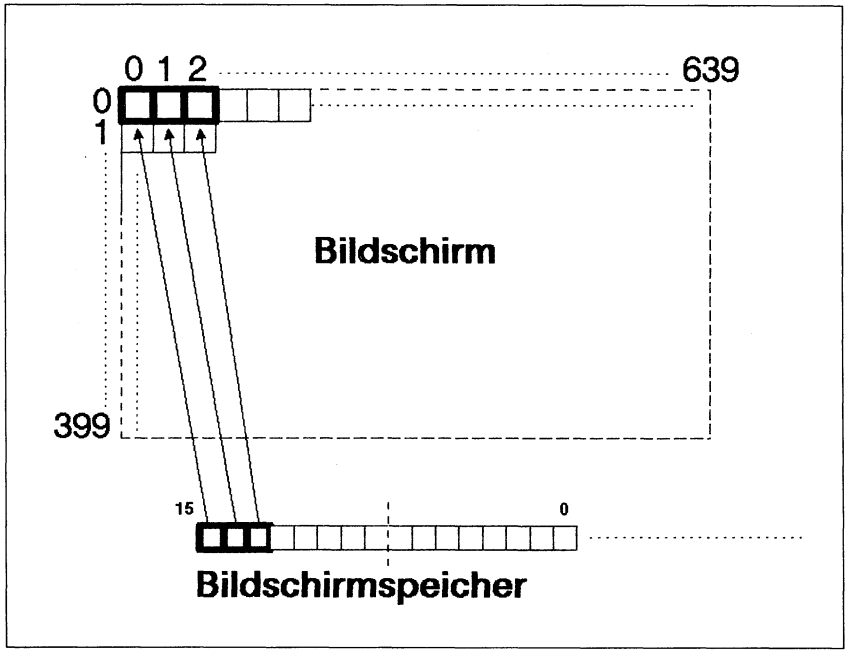


Abb. 6.1: Organisation des Bildschirmspeichers in hoher Auslösung

Für die Programmierung von Grafikroutinen ist es von Bedeutung, daß man den hochauflösenden Modus sowohl byteweise als auch als wortweise organisiert betrachten kann. Die Linearität bleibt in beiden Fällen erhalten, da etwa Bit 15 des ersten Worts das gleiche physikalische Bit anspricht wie Bit 7 des ersten Bytes. Erinnern wir uns, daß beim MC68000 immer die hochwertigen vor den niederwertigen Einheiten abgespeichert werden.) Sinnvoll ist das besonders deshalb, da für die Pixeloperationen die Bit-Befehle wie BSET oder BCLR gut anwendbar sind, diese jedoch die Eigenheit haben, daß im Speicher nur Bytes angesprochen werden können.

Überlegen wir uns nun, wie man aus gegebenen X- und Y-Koordinaten die Adresse und das zu setzende Bit berechnen kann. Dabei soll, wie auf Computern allgemein üblich, die linke obere Ecke des Bildschirms als Koordinatenursprung dienen. Beginnen wir mit der Adresse des Bildschirmspeichers. Um nun zur Anfangsadresse der gewünschten Bildschirmzeile zu kommen, addieren wir das 80-fache der Y-Koordinate (schließlich entspricht eine Bildschirmzeile 80 Bytes). Um nun auf die Adresse des Bytes zu kommen, addiert man X/8, da ein Byte 8 Punkten in der Horizontalen entspricht. (Die Division ist hier als die abrundende ganzzahlige Division aufzufassen, wie sie von DIVU realisiert wird). Somit kommen wir also auf die Formel:

adr = Bildschirmadresse + Y*80 + X/8

Nun geht es nur noch darum, die Bitposition in diesem Byte zu bestimmen. Wir erhalten sie einfach, indem wir X modulo 8 berechnen, also die unteren 3 Bits der X-Koordinate herausgreifen. Allerdings müssen diese Bits noch invertiert werden, da das Bit 7 links dargestellt wird, während dieses Bit bei X modulo 8 rechts steht.

Nun sind wir auch schon so weit, daß wir ein Programm zum Setzen oder Löschen eines Punktes schreiben könnten. "plothi" erwartet in D0 die X-Koordinate, in D1 die Y-Koordinate und in D2 den Farbwert, 0 oder 1. Vor dem ersten Aufruf dieser Routine muß allerdings die logische Bildschirmadresse festgestellt werden, die hier der Einfachheit halber im Register D6 aufbewahrt wird. Übrigens werden keine Bereichsüberprüfungen vorgenommen; dafür ist das aufrufende Programm selbst verantwortlich.

```
*************************************************************
* PLOTHI.S                                                  *
* Routine zum Plotten eines Punktes in hoher Auflösung      *
* Die X-Koordinate wird in D0.W übergeben,                  *
* Y in D1.W und die Farbe in D2.B (0:löschen, 1: setzen).   *
* In A6 wird die Anfangsadresse des Bildschirmspeichers     *
* erwartet.                                                 *
*                                                           *
* Registerbelegung:                                         *
*************************************************************
```

```
CNECIN    EQU     8               * GEMDOS-Funktion
LOGBASE   EQU     3               * XBIOS-Funktion

start     bsr     initscr         * Bildschirmadresse fest-
                                  * stellen
          move    #100,d0         * X-Koordinate
          move    #50,d1          * Y-Koordinate
          moveq   #1,d2           * Pixel setzen
          bsr     plothi          * in die Routine
          move    #CNECIN,-(sp)   * auf Taste warten
          trap    #1              * ins GEMDOS
          addq.l  #2,sp           *
          clr     -(sp)           * TERM
          trap    #1              * Ende

initscr   move    #LOGBASE,-(sp) * logische Bildschirmadresse
 holen
          trap    #14             * ins XBIOS
          addq.l  #2,sp           *
          move.l  d0,a6           * Bildschirmadresse in A6
                                  * merken
          rts

plothi    move.l  a6,a0           * Bildschirmadresse
          move    d0,d3           * berechne scradr+X/8+Y*80
          lsr     #3,d3           * X/8
          add     d3,a0           * zum Ergebnis
          move    d1,d3           * berechne
          mulu    #80,d3          * Y*80
          add     d3,a0           * zum Ergebnis
          move    d0,d3           * berechne Bitstelle
          and     #$FFF8,d3       * nur Bits 0-2 von X
          eor     #7,d3           * berechne 7-Bitstelle
          tst     d2              * Setzen oder löschen?
          bne.s   setzen          * Pixel setzen
loeschen  bclr    d3,(a0)         * Bit löschen
          rts                     * fertig
setzen    bset    d3,(a0)         * Bit setzen
ph_end    rts                     * fertig
          END
```

Setzen eines Punktes in niedriger Auflösung

Nachdem wir nun den hochauflösenden Grafikmodus gemeistert haben, wenden wir uns dem niedrigauflösenden (320 x 200) zu. Dort ist die Darstellung des Bildschirmrasters im Speicher etwas komplizierter, da jeder Pixel eine von 16 Farben annehmen kann. Somit müssen für jeden Bildschirmpunkt 4 Bits gespeichert werden. Diese sind allerdings nicht, wie man vielleicht erwarten könnte, einfach hintereinander abgelegt. Vielmehr werden die Pixel einer

Bildschirmzeile in Gruppen zu 16 aufgeteilt, deren Farbwerte von insgesamt 4 Worten bestimmt werden. Bezeichnen wir die Adresse des Bildschirmspeichers mit "scr" für engl. "screen". Um die Farbe des Punktes in der oberen linken Ecke bestimmen zu können, müssen also die obersten Bits der Worte scr, scr + 2, scr + 4 und scr + 6 herhalten; die Farbe des Punktes rechts daneben wird von den Bits 14 der entsprechenden Worte bestimmt, und so weiter (Abb. 6.2). Wenn man für einen Pixel diese Bits zusammenfügt, und zwar das Bit aus scr als unterstes, das aus scr + 2 als nächsthöheres und so fort, erhält man einen Index von 0 bis 15 in die Farbregister des Videochips, wo die tatsächliche Farbe des Bildschirmpunkts in Rot-, Grün- und Blauanteil festgelegt ist.

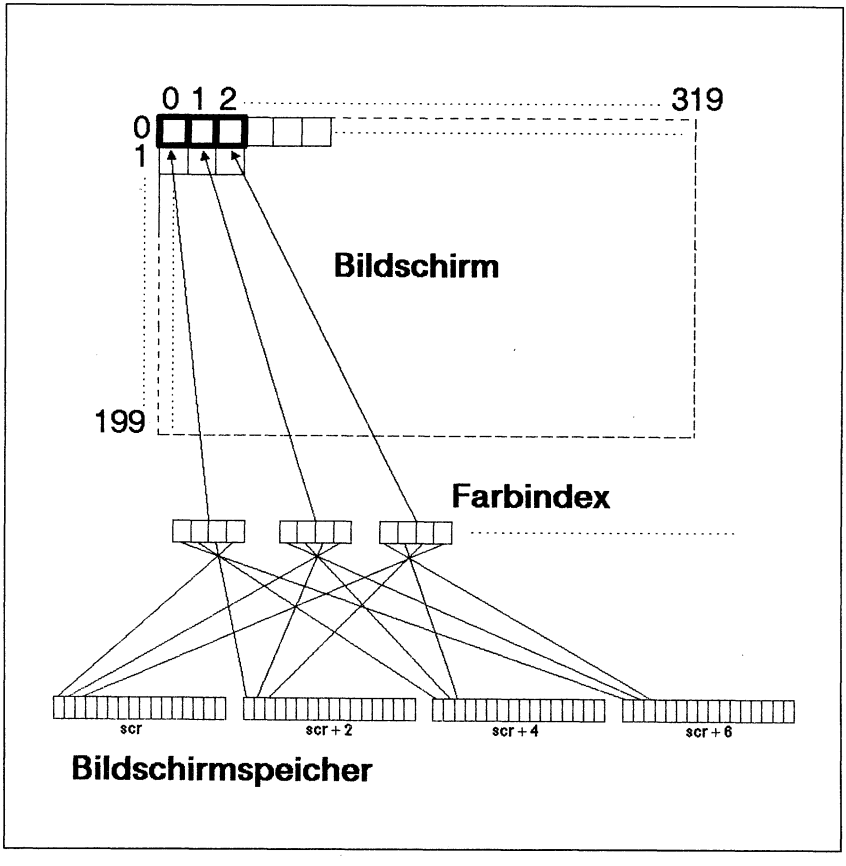


Abb. 6.2: Organisation des Bildschirmspeichers im Lo-Res-Modus

Bei der Berechnung der Adresse eines Pixels gilt es noch zu beachten, daß nun eine Bildschirmzeile 160 Bytes umfaßt (man könnte auch sagen: 20 16-Pixel-Gruppen zu je 4 Worten). Dafür ist die Anzahl der Zeilen in der Vertikalen im Vergleich zum hochauflösenden Modus von 400 auf 200 gesunken, wodurch sich wieder die gleiche Länge des Bildschirmspeichers von 32K ergibt.

Versuchen wir nun, eine Formel für die Berechnung der Adresse eines Pixels mit den Koordinaten X, Y zu finden. Zur Anfangsadresse des Bildschirmspeichers addieren wir zunächst Y * 160, um die Anfangsadresse unserer Zeile zu finden. Die Nummer der vier-Wort-Gruppe in dieser Zeile ergibt sich, wenn man X / 16 berechnet (wieder ganzzahlig), da ja 16 Pixel in einer solchen Gruppe dargestellt werden. Die Distanz zum Anfang der Zeile erhält man, wenn man diesen Wert mit 8 multipliziert, weil eine solche Gruppe jeweils 8 Bytes umfaßt. Somit kommen wir auf die Formel

adr = Bildschirmadresse + Y*160 + (X/16) * 8

Das Bit, das in den vier Worten ab "adr" gesetzt werden muß, errechnet sich einfach nach X modulo 16, das heißt, nur die unteren 4 Bits werden übriggelassen. An dieser Stelle plagt es den Programmierer, daß die Bit-Befehle im Speicher nur Bytes adressieren können. Sicher wäre es eine Möglichkeit, die vier fraglichen Worte nacheinander in ein Register zu holen, dort das Bit zu setzen und sie wieder zurückzuschreiben (erinnern wir uns, daß Register von den Bit-Befehlen auf voller Länge angesprochen werden). Effizienter ist allerdings ein kleiner Trick: Er beruht darauf, daß es aufs gleiche herauskommt, ob man im Wort "adr" Bit n + 8 setzt oder im Byte "adr" Bit n. In unserer Routine wird diese Tatsache indirekt so verwendet: Wenn das Ergebnis von X modulo 16 kleiner oder gleich 7 ist, wird das untere Byte des Wortes angesprochen, das eine um eins größere Adresse hat. Ein "Abschneiden" des Bits 3 ist beim Bereich 8 – 15 nicht nötig, da bei BSET und BCLR ohnehin nur die Bits 0 – 2 beachtet werden.

```
*************************************************************
* PLOTLO.S                                                  *
* Routine zum Plotten eines Punktes in niedriger Auflösung  *
* Die X-Koordinate wird in D0.W übergeben, Y in D1.W und    *
* die Farbe in D2.B (0-15).                                 *
* In A6 wird die Anfangsadresse des                         *
* Bildschirmspeichers erwartet.                             *
*                                                           *
* Registerbelegung:                                         *
*************************************************************
CNECIN    EQU     8             * GEMDOS-Funktion
LOGBASE   EQU     3             * XBIOS-Funktion

start     bsr     initscr       * Bildschirmadresse feststellen
          move    #100,d0       * X-Koordinate
          move    #50,d1        * Y-Koordinate
```

```
          moveq   #15,d2          * Pixel in Farbe 15 (schwarz)
          bsr     plotlo          * in die Routine
          move    #CNECIN,-(sp)   * auf Taste warten
          trap    #1              * ins GEMDOS
          addq.l  #2,sp           *
          clr     -(sp)           * TERM
          trap    #1              * Ende

initscr   move    #LOGBASE,-(sp)  * logische Bildschirmadresse
                                  * holen
          trap    #14             * ins XBIOS
          addq.l  #2,sp           *
          move.l  d0,a6           * Bildschirmadresse in A6
                                  * merken
          rts

plotlo    move.l  a6,a0           * Bildschirmadresse
          move    d0,d3           * berechne
                                  * scradr+(X DIV 16)*8+Y*160
          lsr     #1,d3           * X/2
          and     #$FFF8,d3       * (X DIV 16) * 8
          add     d3,a0           * zum Ergebnis
          move    d1,d3           *
          mulu    #160,d3         * Y*160
          add     d3,a0           * zum Ergebnis
          move    d0,d3           * berechne Bitstelle
          and     #$FFF0,d3       * nur Bits 0-3 von X
          cmp     #7,d3           * Bitstelle kleiner gleich 7?
          bls.s   pl0             * nein, bleibt so
          and     #7,d3           * 8-15 auf 0-7 abbilden
          addq.l  #1,a0           * Adresse um eins erhöhen
pl0       eor     #7,d3           * berechne 7-Bitstelle
          btst    #0,d2           * Farbe Bit 0 setzen
                                  * oder löschen?
          bne.s   pl0set          *
          bclr    d3,(a0)         * Bit löschen
          bra.s   pl1             * nächstes Bit
pl0set    bset    d3,(a0)         * Bit setzen
pl1       btst    #1,d2           * Bit 1 setzen oder löschen?
          bne.s   pl1set          *
          bclr    d3,2(a0)        * Bit löschen
          bra.s   pl2             * nächstes Bit
pl1set    bset    d3,2(a0)        * Bit setzen
pl2       btst    #2,d2           * Bit 2 setzen oder löschen?
          bne.s   pl2set          *
          bclr    d3,4(a0)        * Bit löschen
          bra.s   pl3             * nächstes Bit
pl2set    bset    d3,4(a0)        * Bit setzen
pl3       btst    #3,d2           * Bit 3 setzen oder löschen?
          bne.s   pl3set          *
          bclr    d3,6(a0)        * Bit löschen
          rts                     * fertig
pl3set    bset    d3,6(a0)        * Bit setzen
          rts                     * fertig
          END
```

Obwohl die mittlere Auflösung (640 x 200) hier keine Verwendung findet, soll sie auch noch beschrieben werden. Bei dieser Auflösung wird jedes Pixel durch 2 Bits dargestellt, da es ja 4 mögliche Farben gibt. Die Abspeicherung erfolgt ähnlich wie die der niedrigen Auflösung: Auch hier werden die Pixel in horizontale Gruppen von 16 aufgeteilt, deren Farbe aber nur durch 2 Worte bestimmt ist. Die Farbe der oberen linken Ecke wird also durch Bit 15 der Speicherstellen scr und scr+2 bestimmt. Versuchen Sie ruhig einmal, obigen Algorithmus für die mittlere Auflösung umzuschreiben.

Linien ziehen in hoher und niedriger Auflösung

Besonders bei Programmen mit viel Grafik, ist ein schneller Algorithmus zum Linien ziehen erwünscht. Hier soll deshalb eine von vielen möglichen Methoden vorgestellt werden, die immerhin um einiges schneller als der Line-A-Algorithmus ist.

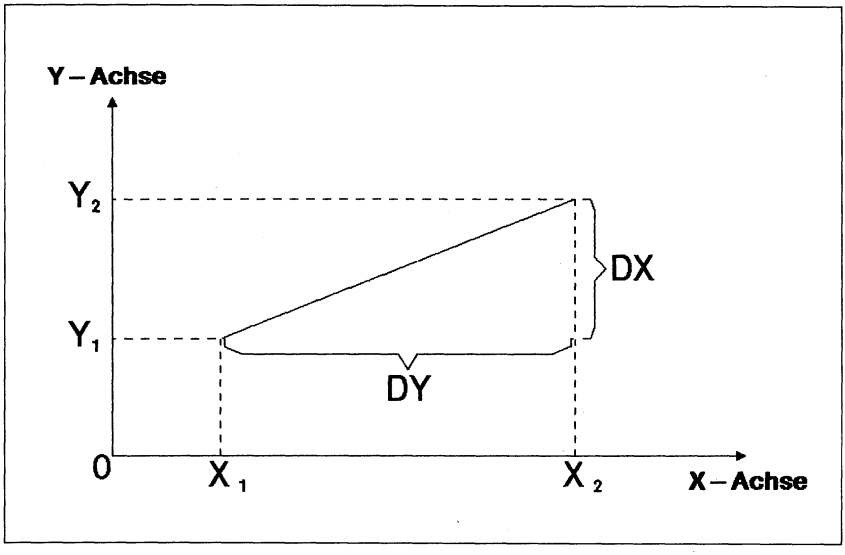


Abb. 6.3: Linie mit positiver Steigung

Das Problem ist also folgendes: Es soll eine möglichst gut angenäherte Linie von X1, Y1 nach X2, Y2 gezogen werden. Versuchen wir zunächst, das etwas einfachere Problem zu knacken, wie eine solche Linie im ersten Quadranten des Koordinatensystems zu zeichnen wäre (Abb. 6.3). Es handelt sich dabei um

eine Linie mit positiver Steigung, bei der X1<X2 gilt. Eine recht einleuchtende Lösung wäre, die Steigung der Gerade nach der Formel

$$S = \frac{(Y2 - Y1)}{(X2 - X1)}$$

zu berechnen. Dann setzt man zwei Variablen, die die momentanen Koordinaten verkörpern (nennen wir sie X und Y) auf die Anfangswerte X1 und Y1. Nun zählt man X1 in Einerschritten hoch und addiert zu Y jedesmal die Steigung, wobei bei jedem Schritt ein Punkt gesetzt wird. Sobald X den Wert von X2 erreicht hat, ist man fertig.

Das klingt recht plausibel, hat aber leider einen Haken: Man kann für jede X-Koordinate nur einen Punkt setzen. Bei sehr steil ansteigenden Linien werden deshalb die Punkte recht dünn gesät sein, bis zum grotesken Extremfall der vertikalen Linie, die nur durch einen kümmerlichen Punkt in Erscheinung tritt. Deshalb muß man hier eine Fallunterscheidung einführen: Ist der Abstand der beiden Y-Koordinaten größer als der der X-Koordinaten, also die Steigung größer als 1, dann werden die Rollen von X und Y vertauscht. Das heißt, daß in diesem Fall Y bei jedem Schritt um eins erhöht wird, während zu X der Kehrwert von S addiert wird. Die X-zu-Y-Steigung errechnet sich also folgendermaßen:

$$SXY = \frac{(X2 - X1)}{(Y2 - Y1)}$$

Vorteilhaft ist auch, daß die Steigung S bzw. SXY in jedem Fall kleiner oder gleich 1 ist, da immer die kleinere Differenz durch die größere geteilt wird. Diese Tatsache werden wir uns noch zu Nutze machen.

Nun haben wir eine Lösung, die wohlgemerkt nur für Geraden im ersten Quadranten gilt. Jetzt geht es darum, alle anderen Fälle mit diesem in Beziehung zu bringen.

Zuerst wird geprüft, ob Y1 kleiner als Y2 ist. Ist das der Fall, geschieht nichts weiter. Andernfalls werden die beiden Endpunkte der Gerade ausgetauscht, so daß die Relation Y1 <= Y2 auf jeden Fall gewahrt ist. Als nächstes wird die X- und Y-Differenz berechnet. Die Y-Differenz (DY) muß ja aufgrund obigen Verhältnisses in jedem Fall positiv sein, während die X-Differenz (DX) auch negativ sein kann. Ist sie es, so wird sie positiv gemacht und statt dessen ein Flag gesetzt. Immer wenn nun X verändert wird, muß der Wert dieses Flags beachtet werden. Ist es gesetzt, so wird nicht addiert, sondern subtrahiert. Ein Flußdiagramm dieses Algorithmus finden Sie in Abbildung 6.4.

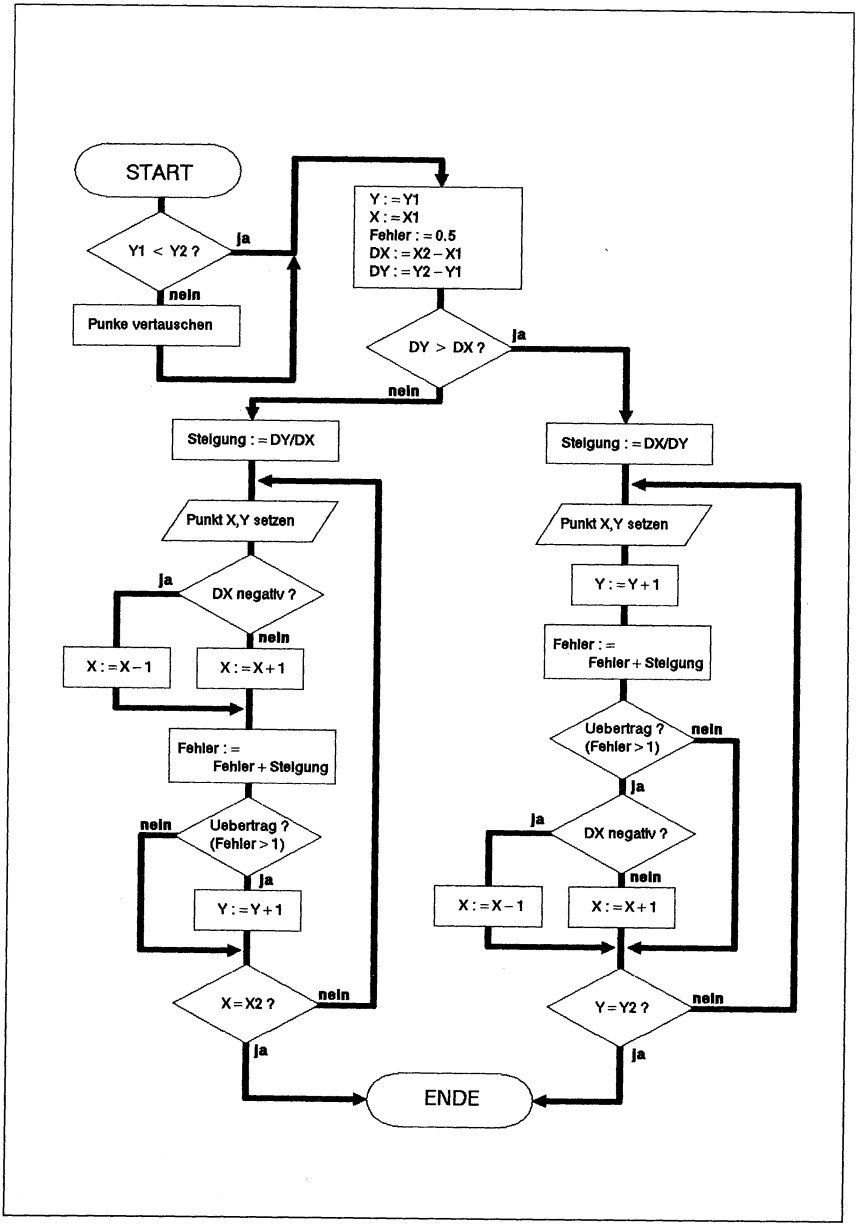


Abb. 6.4: Flußdiagramm des Line-Algorithmus

Soweit die Beschreibung des Algorithmus auf höherer Ebene; wenden wir uns nun der Umsetzung in Assembler zu. Zunächst wäre es vielleicht naheliegend, nach diesem Algorithmus die Koordinaten der Punkte, die zu einer Linie gehören, zu berechnen und diese dann mit einer der obigen Routinen plotten zu lassen. Bei genauerer Betrachtung erweist sich diese Methode allerdings als recht ineffizient, da für jeden Punkt von neuem dessen Adresse berechnet werden muß. Geschickter wäre es, sich von den X- und Y-Koordinaten zu lösen und direkt mit Bildschirmadressen zu arbeiten, zumal bei der Methode, die hier vorgestellt wird, immer nur Schritte von einem Pixel nach rechts, links, oben oder unten auftreten können.

Bei der folgenden Implementierung wurde die Sache so angegangen: Es gibt zwei Zahlen, die die Position auf dem Bildschirm festlegen: die Adresse (steht in A0) und das dort angesprochene Bit (D6). Diese beiden Werte werden nur für den Anfangspunkt der Linie genau wie bei PLOTHI errechnet; danach werden sie nur noch um Pixeleinheiten nach rechts, links, unten oder oben verschoben. Ein Pixelschritt nach links entspricht der Addition von 1 zu D6; wird dabei 7 überschritten, dann wird D6 auf 0 gesetzt und dafür A0 um eins verringert. Ein Schritt nach rechts verläuft umgekehrt. Ein Schritt nach unten kommt der Addition von 80 zu A0 gleich; ein Schritt nach oben der Subtraktion von 80.

Es wird Ihnen wahrscheinlich aufgefallen sein, daß wir eigentlich für den oben beschriebenen Algorithmus Kommazahlen brauchen. Da aber die Werte der Steigung nur zwischen 0 und 1 liegen können, nehmen wir dafür einfach ein Langwort, dessen Bits nur als Nachkommastellen betrachtet werden. Deshalb wird auch bei der Berechnung der Steigung nur deren Nachkommateil betrachtet bzw. der Zähler wird vorher um 16 Stellen nach links verschoben. Entsprechend gibt es auch ein Nachkomma-Langwort für X und Y. Um nun etwa festzustellen, ob beim Hochzählen von X ein Schritt in Y-Richtung erfolgen muß, wird die Steigung zum Nachkomma-Langwort addiert. Tritt dabei ein Übertrag auf, so müßte Y um eins erhöht werden, und der entsprechende Schritt in Y-Richtung wird ausgeführt.

Man kann diesen Nachkommawert als Fehler-Wert betrachten, denn er stellt die Differenz zwischen dem tatsächlichen Y-Wert und dem durch den Rasterbildschirm auf ganze Zahlen beschränkten dargestellten Y-Wert dar.

Man könnte einwenden, daß die 1 in dieser Form nicht dargestellt werden kann, obwohl ja die Steigung sicher eins sein kann. Tatsächlich muß dieser Fall eine Sonderbehandlung erfahren, da er bei unserer (mathematisch nicht sehr exakten, aber schnellen) Berechnung sonst eine Steigung von 0 liefern würde. Die 1 wird durch $FFFF angenähert, was für unsere Zwecke hinreichend genau ist. Die folgende Line-Routine arbeitet nur im EXKLUSIV-ODER-Mo-

dus; es sollte jedoch ein leichtes sein, sie für andere Schreibmodi abzuändern. Die Koordinaten X1, Y1, X2, Y2 werden in den Registern D0, D1, D2 und D4 übergeben. Im Gegensatz zu den anderen bisher vorgeführten Routinen werden hier alle Registerinhalte bewahrt. Die folgende Routine arbeitet im hochauflösenden Modus:

```
*************************************************************
*                                                           *
*    LINEHI.S                                               *
*    Linie ziehen in 640x400;                               *
*    alle Registerinhalte bleiben erhalten                  *
*    Parameter:                                             *
*         d0        X-Wert Anfangspunkt                     *
*         d1        Y-Wert Anfangspunkt                     *
*         d2        X-Wert Endpunkt                         *
*         d3        Y-Wert Endpunkt                         *
*                                                           *
*    interne Belegung:                                      *
*         d4        Niederwertiges Register für             *
*                   X- bzw. Y-Schritt                       *
*         d5,d7     zum Rechnen                             *
*         d6        Bit-Position für den nächsten Punkt     *
*                                                           *
*************************************************************
draw
          movem    d0-d7/a0,-(sp)   * Register retten
          clr      minus            * Initialisierung
          move     #$8000,d4        * Fehler auf 1/2
                                    * initialisieren
          cmp      d1,d3            * y1<y2?
          bhi.s    noexch           * scheint so, also nicht
                                    * austauschen
          exg      d0,d2            * x1 <-> x2
          exg      d1,d3            * y1 <-> y2
noexch    sub      d0,d2            * d2: X-Differenz
          bpl.s    notneg           * Ist X-Different positiv?
          neg      d2               * X-Differenz negativ!
          st       minus            * Minus-Flag setzen
notneg    sub      d1,d3            * d3: Y-Differenz
          move     d0,d6            * berechne erste Bitposition
          and      #7,d6            * nur untere 4 Bits übrig-
                                    * lassen
          eor      #7,d6            * und invertieren
          move     d0,d7            * berechne Adresse
          lsr      #3,d7            * des ersten Punktes:
          move     d1,d5            * adr = screenadr + X/8 + Y*80
          mulu     #80,d5           *
          add      d5,d7            *
          ext.l    d7               *
          add.l    screenadr,d7     *
          move.l   d7,a0            * Adresse nach A0
          cmp      d2,d3            * Y-Differenz > X-Differenz?
          bhi.s    county           * ja, verzweige
```

```
          bne.s   countx          *
          move    #$ffff,d3       * Fall DX=DY kann mit DIVU
                                  * nicht
          bra.s   x_loop          * korrekt behandelt werden
countx    move    #16,d5          *
          lsl.l   d5,d3           * verschiebe Y-Differenz um
                                  * 16 Bits
          divu    d2,d3           * d3 = (DY << 16) / DX
x_loop    bchg    d6,(a0)         * Punkt setzen
          tst.b   minus           * X-Increment negativ?
          bne.s   x_sub           * ja...
x_add     subq    #1,d6           * einen Pixel nach rechts
          bpl.s   countx_1        * OK, wenn Bitzähler posi-
                                  * tiv ist,
          move    #7,d6           * sonst Bitzähler auf 7
                                  * verringern
          addq.l  #1,a0           * und 1 addieren
          bra.s   countx_1        * und weiter geht's
*
* hierher wenn X-Increment negativ
x_sub     addq    #1,d6           * ein Pixel nach links
          cmpi    #8,d6           * 7 überschritten?
          bne.s   countx_1        * nein, weiter
          clr     d6              * Bitzähler auf Null setzen
          subq.l  #1,a0           * und Adresse herunterzählen
*
countx_1  add     d3,d4           * low:= low + Y-Steigung
          bcc.s   countx_2        * kein Carry: kein Y-Schritt
          add.l   #80,a0          * Y-Schritt
countx_2  dbra    d2,x_loop       * wiederhole DX+1 mal
          bra.s   draw_exit       * Fertig!
*
county    move    #16,d5          * Diesmal wird Y gezählt
          lsl.l   d5,d2           *
          divu    d3,d2           * d2: ( DX << 16 ) / DY
y_loop    bchg    d6,(a0)         * Punkt setzen
          add.l   #80,a0          * ein Pixel nach unten
          add     d2,d4           * low:=low+Increment
          bcc.s   county_2        * kein X-Schritt
          tst.b   minus           * rechts oder links?
          bne.s   y_sub           * links!
y_add     subq    #1,d6           * ein Pixel nach rechts
          bpl.s   county_2        * Bitzähler OK
          move    #7,d6           * Bitzähler auf 7 setzen
          addq.l  #1,a0           * und Adresse um 1 erhöhen
          bra.s   county_2        * und weiter
*
y_sub     addq    #1,d6           * ein Pixel nach links
          cmpi    #8,d6           * Bitzähler > 7?
          bne.s   county_2        * nein, weiter
          clr     d6              * ja, Bitzähler auf 0
                                  * setzen...
          subq.l  #1,a0           * und Adresse um 1 erhöhen
county_2  dbra    d3,y_loop       * DY mal wiederholen
draw_exit movem   (sp)+,d0-d7/a0  * Register wiederherstellen
```

```
          rts                    * Rückkehr
*
minus     DS.W    1              * Flag für Richtung oben/links
screenadr DS.L    1              * logische Bildschirm-Adresse
```

Es folgt noch eine Abwandlung für die niedrige Auflösung. Beim eigentlichen Algorithmus gibt es keine Unterschiede; nur der Teil zum Setzen eines Punktes ist etwas abgewandelt worden. Er funktioniert so ähnlich wie die PLOTLO-Routine, nur daß hier aus Effizienzgründen zwei verschiedene Befehlssequenzen für die Fälle Bitnummer > 7 und Bitnummer <= 7 vorhanden sind.

```
*************************************************************
*    LINELO.S                                               *
*    Linie ziehen in 320 x 200;                             *
*    alle Registerinhalte bleiben erhalten                  *
*    Parameter:                                             *
*         d0        X-Wert Anfangspunkt                     *
*         d1        Y-Wert Anfangspunkt                     *
*         d2        X-Wert Endpunkt                         *
*         d3        Y-Wert Endpunkt                         *
*                                                           *
*    interne Belegung:                                      *
*      d4      Niederwertiges Register für X- bzw. Y-Schritt *
*      d5,d7   zum Rechnen                                    *
*         d6   Bit-Position für den nächsten Punkt          *
*                                                           *
*************************************************************
draw
          movem   d0-d7/a0,-(sp)  * Register retten
          clr     minus           * Initialisierung
          move    #$8000,d4       * Fehler auf 1/2 initia-
                                  * lisieren
          cmp     d1,d3           * y1<y2 ?
          bhi.s   noexch          * scheint so, also nicht
                                  * austauschen
          exg     d0,d2           * x1 <-> x2
          exg     d1,d3           * y1 <-> y2
noexch    sub     d0,d2           * d2: X-Differenz
          bpl.s   notneg          * Ist X-Different positiv?
          neg     d2              * X-Differenz negativ!
          st      minus           * Minus-Flag setzen
notneg    sub     d1,d3           * d3: Y-Differenz
          move    d0,d6           * berechne erste Bitposition
          and     #15,d6          * nur untere 4 Bits übrig-
                                  * lassen
          eor     #15,d6          * und invertieren
          move    d0,d7           * berechne Adresse
          lsr     #1,d7           * des ersten Punktes:
          and     #$FFF8,d7       * Bits 0-2 löschen
          move    d1,d5           * adr = screenadr + (X DIV 16)
                                  * 8 + Y * 160
          mulu    #160,d5         *
```

```
          add     d5,d7           *
          ext.l   d7              *
          add.l   screenadr,d7    *
          move.l  d7,a0           * Adresse nach A0
          cmp     d2,d3           * Y-Differenz > X-Differenz?
          bhi.s   county          * ja, verzweige
          bne.s   countx          *
          move    #$ffff,d3       * Fall DX = DY kann mit DIVU
                                  * nicht
          bra.s   x_loop          * korrekt behandelt werden
countx    move    #16,d5          *
          lsl.l   d5,d3           * verschiebe Y-Differenz um
                                  * 16 Bits
          divu    d2,d3           * d3 = (DY << 16) / DX
x_loop    cmp.b   #7,d6           * welches Byte ansprechen?
          bls.s   x_ungerad       *
x_gerade  move.b  d6,d5           * Gerade Adresse
          and.b   #7,d5           * 8-15 auf 0-7 abbilden
          bchg    d5,(a0)         * Punkt setzen
          bchg    d5,2(a0)        *
          bchg    d5,4(a0)        *
          bchg    d5,6(a0)        *
          bra.s   x_test          * weiter
x_ungerad bchg    d6,1(a0)        * Punkt setzen
          bchg    d6,3(a0)        *
          bchg    d6,5(a0)        *
          bchg    d6,7(a0)        *
x_test    tst.b   minus           * X-Increment negativ?
          bne.s   x_sub           * ja...
x_add     subq    #1,d6           * ein Pixel nach rechts
          bpl.s   countx_1        * OK, wenn Bitzähler posi-
                                  * tiv ist,
          move    #15,d6          * sonst Bitzähler auf 15
                                  * verringern
          addq.l  #8,a0           * und 8 zur Adresse addieren
          bra.s   countx_1        * und weiter geht's
*
* hierher wenn X-Increment negativ
x_sub     addq    #1,d6           * ein Pixel nach links
          cmpi    #16,d6          * 15 überschritten?
          bne.s   countx_1        * nein, weiter
          clr     d6              * Bitzähler auf Null setzen
          subq.l  #8,a0           * und Adresse herunterzählen
*
countx_1  add     d3,d4           * low:= low + Y-steigung
          bcc.s   countx_2        * kein Carry: kein Y-Schritt
          add.l   #160,a0         * Y-Schritt
countx_2  dbra    d2,x_loop       * wiederhole DX+1 mal
          bra.s   draw_exit       * Fertig!
*
*
county    move    #16,d5          * Diesmal wird Y gezählt
          lsl.l   d5,d2           *
          divu    d3,d2           * d2: ( DX << 16 ) / DY
```

```
y_loop      cmp.b   #7,d6           * welches Byte ansprechen?
            bls.s   y_ungerad       *
y_gerade    move.b  d6,d5           * Gerade Adresse
            and.b   #7,d5           * 8-15 auf 0-7 abbilden
            bchg    d5,(a0)         * Punkt setzen
            bchg    d5,2(a0)        *
            bchg    d5,4(a0)        *
            bchg    d5,6(a0)        *
            bra.s   y_schritt       * weiter
y_ungerad   bchg    d6,1(a0)        * Punkt setzen
            bchg    d6,3(a0)        *
            bchg    d6,5(a0)        *
            bchg    d6,7(a0)        *
y_schritt   add.l   #160,a0         * ein Pixel nach unten
            add     d2,d4           * low:=low+Increment
            bcc.s   county_2        * kein X-Schritt
            tst.b   minus           * rechts oder links?
            bne.s   y_sub           * links!
y_add       subq    #1,d6           * ein Pixel nach rechts
            bpl.s   county_2        * Bitzähler OK
            move    #15,d6          * Bitzähler auf 15 setzen
            addq.l  #8,a0           * und Adresse um 1 erhöhen
            bra.s   county_2        * und weiter
*
y_sub       addq    #1,d6           * ein Pixel nach links
            cmpi    #16,d6          * Bitzähler > 7?
            bne.s   county_2        * nein, weiter
            clr     d6              * ja, Bitzähler auf 0 setzen
            subq.l  #8,a0           * und Adresse um 1 erhöhen
county_2    dbra    d3,y_loop       * DY mal wiederholen
draw_exit   movem   (sp)+,d0-d7/a0  * Register wiederherstellen
            rts                     * Rückkehr
*
minus       DS.W    1               * Flag für Zählrichrung
                                    * oben/links
screenadr   DS.L    1               * Logische Bildschirm-Adresse
```

Um die Line-Algorithmen zu demonstrieren, zeigen wir eine kleine Grafik-Spielerei, die auf den speziellen Eigenschaften der XOR-Verknüpfung beruht und einen recht interessanten visuellen Effekt erzeugt. Stellen Sie sich zwei Punkte vor, die gleichzeitig in schrägen Sprüngen von einigen Pixeln über den Bildschirm wandern. Wenn sie an eine Grenze des Bildschirms stoßen, werden sie reflektiert wie ein Lichtstrahl an einem Spiegel. Nun zieht man zwischen zwei Sprüngen immer die Verbindungslinie zwischen den beiden Punkten. Damit der Bildschirm nicht irgendwann zugemalt ist, werden früher gezeichnete Linien wieder entfernt, wenn inzwischen, sagen wir, 50 neue Linien gezeichnet worden sind. Hier kommt die interessante Eigenschaft der XOR-Verknüpfung ins Spiel: Wenn man die gleiche Linie zweimal hintereinander im XOR-Modus zeichnet, ist alles so, wie es vorher war. Das funktioniert auch dann, wenn nacheinander zwei sich schneidende Linien zweifach gezeichnet werden.

So brauchen wir uns keine aufwendigen Operationen auszudenken, um Linien wieder vom Bildschirm zu entfernen. Durch das kontinuierliche Zeichnen und Löschen von Linien entsteht der Effekt einer dreidimensionalen linierten Fläche, die sich in den wildesten Verzerrungen über den Bildschirm bewegt.

Nun stellt sich die Frage, wie man das realisiert. Die Geschichte mit den zwei Punkten, die sich geradlinig über den Bildschirm bewegen und am Rand abgestoßen werden, stellt sicher kein großes Problem dar. Doch wir müssen uns die Koordinaten von früher gezeichneten Linien, die noch nicht gelöscht worden sind, irgendwie merken, da man wohl kaum die Koordinaten 50 Schritte zurückverfolgen könnte. Dazu nimmt man am besten eine sogenannte Schlange (engl. queue). Eine Schlange ist ein FIFO- Stapel ("first in, first out" – was zuerst hineinkommt, wird auch zuerst wieder herausgelesen). In Assembler besteht eine Schlange einfach aus einem reservierten Speicherbereich und zwei Zeigern, die auf Werte in diesem Bereich weisen. Der erste zeigt auf die Stelle, an die der nächste Eintrag in die Schlange geschrieben wird, der zweite auf die, an der etwas herausgelesen werden kann. Wichtig ist, daß die Zeiger beim Schreiben und Lesen innerhalb des zulässigen Bereichs bleiben. Wird etwas in den physisch obersten Teil der Schlange geschrieben, so weist der entsprechende Zeiger danach nicht auf den Eintrag dahinter, sondern auf den ersten physischen Eintrag.

Jedesmal, wenn eine neue Gerade gezeichnet wird, werden deren Koordinaten in einer solchen Schlange vermerkt. Wenn insgesamt schon mehr als 50 Geraden gezeichnet worden sind, werden die letzten Koordinaten (also die der Geraden, die vor 50 Schritten gezeichnet wurde) wieder herausgelesen und aus der Schlange entfernt, und die Gerade wird vom Bildschirm gelöscht, indem sie noch einmal mit XOR gezeichnet wird.

Damit das Programm lauffähig wird, kopieren Sie mit einem Editor je nach der Auflösung, in der das Programm laufen soll, an der angegebenen Stelle den Line-Algorithmus in den Quelltext. Die Voreinstellung der Parameter des folgenden Listings ist für die hohe Auflösung gedacht; ändern Sie für die hohe Auflösung die Konstanten XMAX auf 319 und YMAX auf 199. Wenn Sie wollen, können Sie zusätzlich den Wert des Symbols ANZ ändern, der die Anzahl der gleichzeitig auf dem Bildschirm sichtbaren Geraden angibt. Natürlich steht es Ihnen auch frei, die Additionswerte für die beiden Koordinatenpaare in den Registern D4 – D7 zu verändern.

```
**************************************************************************
* LINES.S                                                                *
* Grafikspielerei mit einer Gruppe von Linien, die über den Bildschirm   *
* wandert. Damit es lauffähig ist, muß noch eine Line-Routine            *
* eingefügt werden, je nach Auflösung entweder LINESHI.S oder            *
```

```
* LINESLO.S                                                              *
* Für niedrige Auflösung noch XMAX und YMAX ändern!                      *
*************************************************************************
CONSTAT   EQU     11                  * GEMDOS-Funktionsnummer
LOGBASE   EQU     3                   * XBIOS-Funktionsnummer
CURSCONF  EQU     21                  * XBIOS-Funktionsnummer

ANZ       EQU     50                  * Anzahl der Linien

XMAX      EQU     639                 * maximale X-Koordinate
YMAX      EQU     399                 * maximale Y-Koordinate

*************************************************************************
* Registerbelegung der Hauptschleife                                    *
* D0      1. X-Koordinate X1                                            *
* D1      1. Y-Koordinate Y1                                            *
* D2      2. X-Koordinate X2                                            *
* D3      2. Y-Koordinate Y2                                            *
* D4      Additionswert für X1 (DX1)                                    *
* D5      Additionswert für Y1 (DY1)                                    *
* D6      Additionswert für X2 (DX2)                                    *
* D7      Additionswert für Y1 (DY2)                                    *
*************************************************************************
start     bsr     initscreen          * Bildschirm löschen etc.
          lea     schlange,a2         * Anfangs-Zeiger der Schlange
          lea     schlange,a3         * Ende-Zeiger der Schlange
          move    #20,d0              * Beliebiger Anfangswert für X1
          move    #50,d1              * Y1
          move    #100,d2             * X2
          move    #150,d3             * Y2
          move    #3,d4               * X1-Additionswert
          move    #4,d5               * Y1-Additionswert
          move    #-5,d6              * X2-Additionswert
          move    #2,d7               * Y2-Additionswert
drawloop  movem   d0-d3,(a2)          * neue Werte in Schlange ablegen
          tst.b   undraw              * alte Gerade löschen?
          beq.s   drawline            * Nein, weiter
          movem   (a3)+,d0-d3         * alte werte aus Schlange holen
          bsr     draw                * und Linie zeichnen (löschen)
          cmp.l   #schlange+8*ANZ,a3  * am Ende der Schlange?
          bne.s   drawline            * Nein, weiter
          lea     schlange,a3         * auf Anfang der Schlange setzen
drawline  movem   (a2),d0-d3          * neue Werte wiederholen
          addq.l  #8,a2               * und Zeiger erhöhen
          cmp.l   #schlange+8*ANZ,a2  * Ende der Schlange erreicht?
          bne.s   godraw              * Nein, weiter
          lea     schlange,a2         * auf Anfang der Schlange setzen
godraw    bsr     draw                * und neue Linie zeichnen
x1add     add     d4,d0               * X1=X1+DX1
          cmp     #XMAX,d0            * X1 > XMAX oder X1 < 0 ?
          bcs.s   y1add               * nein, weiter
          neg     d4                  * ja, DX1=-DX1
          add     d4,d0               * X1=X1+2*DX1
          add     d4,d0               *
```

```
y1add     add     d5,d1              * Y1 entsprechend
          cmp     #YMAX,d1
          bcs.s   x2add
          neg     d5
          add     d5,d1
          add     d5,d1
x2add     add     d6,d2              * X2 entsprechend
          cmp     #XMAX,d2
          bcs.s   y2add
          neg     d6
          add     d6,d2
          add     d6,d2
y2add     add     d7,d3              * Y2 entsprechend
          cmp     #YMAX,d3
          bcs.s   loopend
          neg     d7
          add     d7,d3
          add     d7,d3
loopend   movem   d0/a0,-(sp)        * Register sichern
          move    #CONSTAT,-(sp)     * GEMDOS-Funktionsnummer
          trap    #1                 * Einsprung ins GEMDOS
          addq.l  #2,sp              * Stack aufräumen
          tst     d0                 * Zeichen Eingegeben?
          bne.s   exit               * Ja, Programmende
          movem   (sp)+,d0/a0        * Register wiederholen
          tst.b   undraw             * Flag schon gesetzt?
          bne     drawloop           * Ja, nächster Durchlauf
          add     #1,zaehler         * Eine Gerade mehr
          cmp     #ANZ-1,zaehler     * bis ANZ-1 geraden gezeichnet sind
          bcs     drawloop           * zaehler<ANZ-1 -> Schleife
          st      undraw             * Zaehler=ANZ-1, ab jetzt...
          bra     drawloop           * Geraden wieder löschen
exit      movem   (sp)+,d0/a0        * Programmende; Register wiederholen
          clr     -(sp)              * GEMDOS TERM, Funktionsnummer 0
          trap    #1                 * Das war's!
*
* Bildschirm löschen, Cursor abschalten und Bildschirmadresse feststellen
initscreen
          move    #0,-(sp)           * Cursor aus
          move    #CURSCONF,-(sp)    * XBIOS Funktionscode
          trap    #14                * zum XBIOS
          addq.l  #4,sp              *
          move    #LOGBASE,-(sp)     * logische Bildschirmadresse holen
          trap    #14                * zum XBIOS
          addq.l  #2,sp              *
          move.l  d0,screenadr       * logische Bildschirmadr. speichern
          move.l  d0,a0              * und nach A0
          move    #7999,d0           * 32000/4 = 8000 Langworte löschen
clr_loop  clr.l   (a0)+              *
          dbra    d0,clr_loop        *
          rts                        *
*
* Hier die Line-Routine einfügen
*
```

```
          DATA
zaehler   DC.W    0              * Zähler für gezeichnete Geraden
undraw    DC.W    0              * Löschen von Geraden
          BSS
schlange  DS.W    ANZ*4          * für die Koordinaten von ANZ Geraden
          END
```

Programmierung von Interrupts

Bevor es losgeht, zunächst eine Warnung: Wenn Sie noch nicht viel Erfahrung mit Assembler haben, empfehle ich Ihnen, sich zuvor durch praktische Programmierung einige Übung zu verschaffen.

Erinnern wir uns, daß Interrupts Unterbrechungen der normalen Programmabarbeitung des Prozessors sind, der seine Aufmerksamkeit kurzzeitig anderen, meist systemspezifischen Aufgaben widmet, um danach genau an der Stelle fortzufahren, an der er unterbrochen worden ist.

Der von Programmen wohl am häufigsten benutzte Interrupt ist der Vertical-Blank-Interrupt, abgekürzt VBI. Er wird vom Grafikchip jedesmal dann ausgelöst, wenn der Elektronenstrahl gerade die unterste Bildzeile beendet hat und sich unsichtbar auf den Weg von der rechten unteren zur linken oberen Ecke macht, um das nächste Bild zu zeichnen. Aus der Sicht des Computers ist das eine ziemlich lange Zeitspanne; er kann in der Zwischenzeit bequem einige tausend Befehle ausführen. Wenn man noch die Zeit hinzunimmt, die der Elektronenstrahl für das Zeichnen der Zeilen über der ersten Zeile mit Pixelinformationen braucht, kann man sich sogar eine ganze Menge Zeit lassen. Die wichtigste Aufgabe des VBI ist die Synchronisation von Grafikoperationen zum Bildschirmaufbau. Nur wenn Operationen wie Änderung der physikalischen Bildschirmadresse, Änderung der Farbregister oder das Neuzeichnen von Figuren während der Vertical-Blank-Phase ausgeführt werden, läßt sich das Flackern verhindern. Deshalb wird etwa die Änderung einer Bildschirmfarbe vom Betriebssystem nicht sofort ausgeführt, sondern erst während des nächsten VBI.

Die zweite wichtige Aufgabe des VBI ist das Timing, also die zeitliche Abstimmung von Programmen. So wird etwa die Zeit für ein Drucker-Timeout mit Hilfe des VBI gemessen.

Der VBI wird zunächst von einer Betriebssystemroutine behandelt, doch es ist auch eine Möglichkeit vorgesehen, daß ein Programm mehrere Routinen an

diese anhängen kann. Dafür wird ein Feld von Vektoren (Zeiger auf Routinen) zur Verfügung gestellt, in denen Programme die Adressen von Unterprogrammen eintragen können. Diese Unterprogramme werden dann bei jedem VBI ausgeführt.

Bevor wir weiter in die Interrupts einsteigen, befassen wir uns zunächst eingehender mit dem Grafikchip "Shifter". Wie Sie wissen, kann man auf einem Farbmonitor je nach Auflösung 4 oder 16 Farben gleichzeitig aus einer Palette von 512 Farben darstellen. Wie wird diese Palette nun festgelegt?

Jedem Bitmuster eines Pixels entspricht ein Farbregister. So spricht etwa bei der mittleren Auflösung die Kombination 00 das erste Farbregister an, 01 das zweite, 10 das dritte und 11 schließlich das vierte. Erst in den Farbregistern steht nun die eigentliche Farbe des Punktes. Auf dem ST wird sie nach Rot-, Grün-, und Blauanteil getrennt festgelegt. Für jeden der drei Anteile stehen drei Bits zur Verfügung, mit denen man Werte von 0 bis 7 darstellen kann. 0 bedeutet, daß die entsprechende Grundfarbe überhaupt nicht zur Geltung kommt, 7 steht für volle Helligkeit einer Grundfarbe. Die drei Farbwerte mischen sich additiv.

Jedes der Farbregister belegt ein Wort. Bitweise sind die Register so aufgeteilt:

```
-----rrr-ggg-bbb
```

In Worten: Bits 0 – 2 enthalten den Blauanteil, Bits 4 – 6 den Grünanteil und Bits 8 – 10 den Rotanteil. Alle anderen Bits sind nicht belegt. Diese Darstellung hat den Vorteil, daß man Farbwerte anschaulich als Hexadezimalzahlen mit drei Ziffern darstellen kann; $777 etwa ergibt ein sattes Weiß.

Die Farbregister können nicht nur beschrieben, sondern auch gelesen werden (bei Hardware-Registern ist das nicht selbstverständlich). Es gilt allerdings zu beachten, daß die oben durch Striche gekennzeichneten Bits keine Informationen speichern können und beim Auslesen immer Null liefern.

Ein für die Grafikprogrammierung auch sehr interessanter Interrupt ist der Horizontal-Blank-Interrupt (HBI). Er wird auch vom Grafikchip ausgelöst, und zwar nach jeder beendeten Bildschirmzeile. Das ist der Moment, wenn der Elektronenstrahl sich vom rechten zum linken Rand des Bildschirms auf den Weg macht, um den Anfang der nächsten Zeile zu zeichnen. Allerdings geht das so schnell, daß diese Zeitspanne sogar für den Prozessor relativ kurz ist. Während des wirklichen HBI's können nicht mehr als etwa 3 oder 4 Befehle

ausgeführt werden. Je nachdem, was diese Befehle tun, können sie allerdings auch nach der echten Horizontal-Blank-Phase ausgeführt werden, während eine neue Zeile schon gezeichnet wird.

Ein HBI ist in erster Linie dazu gedacht, die Farbregister synchron zum Bildschirmaufbau zu ändern. Mit diesem Trick ist man nicht mehr auf 4 oder 16 Farben beschränkt, sondern es lassen sich praktisch beliebig viele Farben gleichzeitig darstellen – nur nicht auf einer Zeile.

Um so etwas zu programmieren, betrachten wir erst einmal die Organisation der Interrupts im ST. Der Prozessor bietet sieben Interruptebenen, wobei ein Interrupt mit höherer Ebene immer Proirität vor einem mit niedrigerer Ebene hat. Mit den ersten 3 Bits des Systembytes kann man bestimmen, ab welcher Ebene Interrupts erlaubt werden. Auf dem ST werden aber nur drei der sieben möglichen Interrupts benutzt. Die folgende Aufstellung zeigt die Zuordnung zu den Interruptebenen:

Ebene 1	nicht belegt
Ebene 2	HBI, Horizontal-Blank-Interrupt
Ebene 3	nicht belegt
Ebene 4	VBI, Vertical-Blank-Interrupt
Ebene 5	nicht belegt
Ebene 6	MFP 68901 – Interrupts
Ebene 7	nicht belegt

Jeder Interruptebene ist ein Vektor zugeordnet, über den beim Auftreten eines Interrupts automatisch gesprungen wird:

Ebene	Vektornummer	Vektoradresse
1	25	$64
2	26	$68
3	27	$6C
4	28	$70
5	29	$74
6	30	$78
7	31	$7C

Normalerweise ist die Interruptmaske im Systembyte auf 3 gesetzt, damit die HBIs gesperrt sind, denn sie würden einen beachtlichen Teil der Rechenzeit verbrauchen.

Sobald ein Interrupt auftritt, wird die Interruptmaske im Systembyte automatisch auf die Nummer des Interrupts gesetzt, damit die Ausführung des Inter-

rupts nicht von Interrupts gleicher oder niedrigerer Priorität unterbrochen werden kann. Um das Zurücksetzen der Interruptmaske braucht sich aber die Interrupt-Routine nicht zu kümmern, da ja beim Auftreten eines Interrupts automatisch das Statusregister auf dem Stack abgelegt wird. Somit wird bei der Beendigung eines Interrupts mit RTE automatisch die alte Interruptmaske wiederhergestellt.

Die oben genannten MFP-Interrupts sind eine genauere Betrachtung wert. MFP steht für "Multifunction Peripheral". Es handelt sich dabei um einen Chip, der im ST eine Art "Mädchen für alles" darstellt: Er ist u. a. für den größten Teil der Interrupterzeugung und die Steuerung der Centronics-Schnittstelle verantwortlich. Uns sollen hier nur seine Interruptmöglichkeiten interessieren.

Zunächst einmal verfügt der MFP 68901 über vier Timer mit den Bezeichnungen A, B, C und D. Sie sind in erster Linie dazu gedacht, immer nach bestimmten Zeitspannen Interrupts auszulösen. So gehört zu jedem Timer ein Zähler und ein Datenregister. Der Zähler wird mit einer in Schritten einstellbaren Frequenz (bis ca. 250 kHz) heruntergezählt. Sobald er Null erreicht, kann der MFP einen Interrupt auslösen. Im gleichen Moment wird der Zähler mit dem Wert des Datenregisters initialisiert und fängt wieder an, herunterzuzählen. Es handelt sich dabei um den Delay-Modus (engl. delay: Verzögerung). Auf diese Art wird immer nach einer bestimmten Zeitspanne ein Interrupt ausgelöst. Nach diesem Prinzip wird etwa mit Timer A ein 200 Hz-Systemtakt und mit Timer D die Baudrate für die RS-232-Schnittstelle erzeugt. Jeder der vier Timer kann auf diese Art zur Erzeugung eines regelmäßig auftretenden Signals benutzt werden. Mit Timer B hat es jedoch etwas besonderes auf sich: Er kann wahlweise auch im sogenannten Event-Count-Mode verwendet werden. Das bedeutet, daß der Timer nicht mit einer vom Taktsignal abgeleiteten Frequenz verringert wird, sondern immer dann um eins heruntergezählt wird, wenn eine bestimmte Eingangsleitung des MFP auf High geht. Geschickterweise ist nun der MFP im ST so verdrahtet, daß diese Eingangsleitung für den Timer B mit dem Zeilen-Synchronisations-Signal des Shifters verbunden ist. Daraus ergibt sich folgendes: Immer wenn eine Bildschirmzeile beendet ist (also immer dann, wenn ein HBI auftreten kann), wird der Zähler des Timers B um eins verringert, sofern sich dieser im Event-Count-Mode befindet. So bietet sich also eine Alternative, wie man HBIs in jeder soundsovielten Zeile erzeugen kann. Klarer wird das, wenn wir uns die Register des Timers B ansehen:

$FFFA1B Timer B Control (8 Bit)
Hier wird der Betriebsmodus des Timers B festgelegt. Nur die ersten fünf Bits dieses Registers sind belegt. Für uns sind dabei nur die ersten drei Bits von Bedeutung:

Bit 2	1	0	
0	0	0	Timer Stop, nichts ausführen
0	0	1	Delay-Modus, Taktsignal durch 4 teilen
0	1	0	Delay-Modus, Taktsignal durch 10 teilen
0	1	1	Delay-Modus, Taktsignal durch 16 teilen
1	0	0	Delay-Modus, Taktsignal durch 50 teilen
1	0	1	Delay-Modus, Taktsignal durch 100 teilen
1	1	0	Delay-Modus, Taktsignal durch 200 teilen
1	1	1	Event-Count-Mode

Bei den Einstellungen 1 – 6 gilt es zu beachten, daß der MFP nicht mit der normalen Systemfrequenz von 8 MHz getaktet ist, sondern nur mit 1 MHz. Daraus ergibt sich, daß die höchste erzeugbare Signalfrequenz bei 250 kHz liegt (Einstellung 1).

$FFFA21 Timer B Data (8 Bit): Hier wird der Wert eingetragen, mit dem der Zähler des Timers B beim Erreichen von 0 wieder initialisiert wird. Es können Werte von 1 bis 255 eingetragen werden. Im Delay-Modus wird also die Taktfrequenz von 1 MHz zunächst durch den im Control-Register festgelegten Wert geteilt, um dann noch einmal durch die hier eingetragene Zahl geteilt zu werden. Das Ergebnis ist die Frequenz, mit der der MFP einen Interrupt auslösen kann. Beim Event-Count-Mode wird die Anzahl der eintreffenden Events direkt durch die hier angegebene Zahl geteilt.

Wie löst nun der MFP einen Interrupt aus? Tatsächlich ist ja dem MFP nicht nur ein einziger Interrupt zugeordnet, sondern bis zu 16, davon 4 alleine für die Timer. Die Behandlung eines MFP-Interrupts geht so vor sich: Der MFP löst einen Interrupt der Ebene 6 aus. Daraufhin springt der MC68000 in eine Routine, die in einem bestimmten Register des MFP die Ursache des Interrupts abfragt und durch einen Vektor in die wirkliche Interrupt-Routine verzweigt. Dort werden als Ursache des Interrupts die entsprechenden Aktionen ausgeführt, und der Vorgang wird schließlich mit einem RTE abgeschlossen.

Die Vektoren der MFP-Interrupts stehen ab Adresse $100 im Speicher, was der Vektornummer 64 entspricht. Die für uns interessanten Vektoren sind folgende:

Nummer	Adresse	Beschreibung
4	$110	Timer D, RS232 Baudraten-Generator
5	$114	Timer C, 200Hz Systemtakt-Generator

6	$118	Tastatur- und Midi-Interrupt
8	$120	Timer B, HBI-Zähler
13	$134	Timer A, vom System nicht benutzt

Die obige Nummer gibt die Nummer des MFP-Interrupts an, die von 0 bis 15 reichen kann. Auch diese Interrupts sind untereinander priorisiert: Um so größer die Nummer desto größer die Priorität. Im MFP gibt es vier Gruppen von Registern, mit denen diese Interrupts kontrolliert werden können:

$FFFA07 Interrupt-Enable-Register A (IERA): Mit diesem Register lassen sich Interruptquellen des MFP gesondert an- oder abschalten. Bit 0 steht dabei für Timer B: Ist es gesetzt, so ist ein Timer-B-Interrupt erlaubt, sonst nicht. Im gleichen Sinne steht Bit 5 für Timer A.

$FFFA09 Interrupt-Enable-Register B (IERB): Dieses Register enthält die Fortsetzung des eben beschriebenen Registers. Bit 4 erlaubt Timer-D-Interrupts, Bit 5 Timer-C-Interrupts. Die anderen Bits dieser Register sind für uns nicht von Bedeutung, da sie für die Steuerung des Centronics-Ports verwendet werden.

Übrigens haben die Bezeichnungen A und B dieser Register nichts mit den Timern A und B zu tun; sie sollen nur die Aufteilung von Bits auf zwei 8-Bit-Register verdeutlichen.

$FFFA0B Interrupt-Pending-Register A (IPRA): Sobald ein Interrupt auftritt, wird das zugeordnete Bit im Interrupt-Pending-Register auf 1 gesetzt. Es gilt die gleiche Zuordnung der Bits wie bei IERA.

$FFFA0D Interrupt-Pending-Register B (IPRB): siehe IPRA

$FFFA0F Interrupt-In-Service-Register A (ISRA): Auch in diesem Register wird beim Auftreten eines Interrupts das zugehörige Bit gesetzt. Solange hier ein Bit gesetzt ist, sind alle MFP-Interrupts gleicher und niedrigerer Priorität gesperrt. Deshalb sollte am Ende einer Interrupt-Routine das entsprechende Bit in ISRA gelöscht werden.

$FFFA11 Interrupt-In-Service-Register B (ISRB): siehe ISRA

$FFFA13 Interrupt-Mask-Register A (IMRA): Wenn ein bestimmtes Bit in IERA gesetzt ist, aber nicht im Mask-Register, so wird

die Interrupt-Ursache nur im Interrupt Pending Register angezeigt, aber kein Interrupt ausgeführt. Erst wenn Bits in beiden Registern gesetzt sind, wird der Interrupt tatsächlich ausgeführt.

$FFFA15 Interrupt-Mask-Register B (IMRB): siehe IMRA

Soviel zur Benutzung des MFP: Inzwischen werden Sie sich sicher fragen, wozu ich Ihnen all das erzähle, denn um HBIs zu verwenden, braucht man doch eingentlich nur eine Routine zu schreiben, die die Bildschirmfarben ändert, deren Adresse im Interruptvektor der Ebene 2 einzutragen und die Prozessor-Interruptmaske auf 1 zu verringern – und schon hätte man HBIs. Leider werden bei der direkten Benutzung des HBI-Vektors die Interrupts erst ausgelöst, nachdem etwa ein Viertel der neuen Bildschirmzeile bereits gezeichnet ist, was bei einer Farbänderung zu einem flackern des Bildschirms führt. Außerdem ergeben sich Probleme dadurch, daß die HBIs die niedrigste Priorität haben und deshalb von sämtlichen anderen Interrupts durcheinandergebracht werden. So führt also nichts am Umweg über den MFP vorbei.

Jetzt sind wir so weit, daß Sie sich das folgende Listing ansehen können. Das Programm erzeugt in jeder Bildschirmzeile zwei neue Farben, wobei die Farbstreifen einer Farbe nach oben, die der anderen nach unten rollen.

```
*************************************************************
*       COLORS.S                                            *
*       Demonstration von Horizontal-Blank-Interrupts       *
*       läuft mit Farbmonitor, in mittlerer oder niedriger  *
*       Auflösung. Am besten als GEM-Programm starten!      *
*************************************************************
ZEILEN     EQU      1                * in jeder Zeile Interrupt
*
* System-Vektoren
hbivec     EQU      $120             * Vektor zur HBI-Routine
vbivec     EQU      $70              * Vektor zur VBI-Routine
mkbvec     EQU      $118             * Maus- u. Tastatur-Interrupt
*
* MFP-Register
iera       EQU      $FFFA07          * Interrupt-Enable-Register A
ierb       EQU      $FFFA09          * Interrupt-Enable-Register B
isra       EQU      $FFFA0F          * Interrupt-Service-Regi-
                                     * ster A
imra       EQU      $FFFA13          * Interrupt-Mask-Register A
tbcr       EQU      $FFFA1B          * Timer B Control-Register
tbdr       EQU      $FFFA21          * Timer B Data-Register
*
* Shifter-Register
color_0    EQU      $FF8240          * Register für Hintergrund-
```

```
                                        * farbe
color_2   EQU     $FF8244               * 2. Vordergrundfarbregister
*
* XBIOS-Funktionscode
SUPEXEC   EQU     38                    * Routine im SUPER-Modus
                                        * ausführen
*
* Programmstart
* zunächst eine Routine, die die Farbwerte von 0 bis 511 auf
* die Shifter-Darstellung 00000rrr0ggg0bbb umrechnet
start     lea     farbtab,a0            * Adresse der Farbliste
                                        * nach a0
          clr     d0                    * mit Farbe 0 anfangen
ci_loop   move    d0,d1                 * Farbe nach D1,D2 und D3
                                        * kopieren
          move    d0,d2                 *
          move    d0,d3                 *
          and     #%000000111,d1        * B-Bits aussondern
          and     #%000111000,d2        * G-Bits aussondern
          and     #%111000000,d3        * R-Bits aussondern
          lsl     #1,d2                 * eine Stelle frei vor G-Bits
          lsl     #2,d3                 * noch eine Stelle frei von
                                        * R-Bits
          or      d2,d1                 * und wieder alles zusam-
                                        * menfügen
          or      d3,d1                 *
          move    d1,(a0)+              * Farbwert in Tabelle
                                        * abspeichern
          addq    #1,d0                 * nächster Farbwert
          cmp     #512,d0               * Ende erreicht?
          bcs.s   ci_loop               * nein, nächster Durchlauf
*
* Jetzt wird die "gefährliche" supinit-Routine im
* Supervisor-Modus gestartet
          pea     supinit               * Adresse der Routine auf
                                        * Stack
          move    #SUPEXEC,-(sp)        * Funktionscode
          trap    #14                   * ins XBIOS
          addq.l  #6,sp                 *
          clr     -(sp)                 * Programm beenden
          trap    #1                    * GEMDOS TERM
*
* zunächst muß allerhand initialisiert werden
supinit   move.l  vbivec,oldvbi+2       * alten VBI-Vektor merken
          move.l  mkbvec,oldmkb+2       * alter Maus- und Tastatur-
                                        * Vektor
          move.l  #hbi,hbivec           * neuen HBI-Vektor eintragen
          move.l  #vbi,vbivec           * neuen VBI-Vektor eintragen
          move.l  #mkb,mkbvec           * neuen MKB-Vektor eintragen
          and.b   #$DF,ierb             * Timer C Interrupt ab-
                                        * schalten
          or.b    #1,iera               * HBI-Interrupt erlauben
          or.b    #1,imra               * HBI-Interrupt erlauben
*
* OK, das war die Initialisierung.* Jetzt wird einige Sekunden
```

```
gewartet...
          move.l   #2000000,d0      * warte ein Weilchen
wait      subq.l   #1,d0            * ...
          bne.s    wait             * ...
*
* So. Damit das System zu normalen Zuständen zurückkehren
* kann, muß alles wiederhergestellt werden.
          move.l   oldvbi+2,vbivec * alten VBI-Vektor wieder-
                                   * herstellen
          move.l   oldmkb+2,mkbvec * alten MKB-Vektor wieder-
                                   * herstellen
          and.b    #$FE,iera       * HBI-Interrupt ausschalten
          or.b     #$20,ierb       * Timer C Interrupt ein-
                                   * schalten
          move     #$777,color_0   * Standard-Hintergrund-
                                   * farbe
          move     #$070,color_2   * Standard-Vordergrund-
                                   * farbe
          rts                      * Fertig!
*
* Hier muß die Routine zur Behandlung von Maus- und Tastatur-
* interrupts durch. Es wird dafür gesorgt, daß HBIs auch
* während dieses Interrupts erlaubt sind, damit die Farben
* nicht durcheinandergeraten.
mkb       move.w   #$2500,SR          * HBIs erlauben
oldmkb    jmp      $FFFFFFFF          * zur alten Routine
                                      * springen
*
* Diese Routine wird vor die Systemroutine für VBI geschaltet.
* Es werden neue Farbwerte berechnet und danach HBIs
* ermöglicht.
vbi       movem.l d0/a0,savereg       * A0 und D0 sichern
          move.b   #0,tbcr            * Timer B anhalten,
                                      * keine HBIs
          lea      farbtab,a0         * Adresse der Farbwert-
                                      * Tabelle
          move     colind0,d0         * Anfangsindex der Hin-
                                      * tergrundfarbe
          move     d0,colind0a        * in HBI-Farbindex
                                      * schreiben
          move     0(a0,d0.w),color_0 * Farbwert ins Shifter-
                                      * Register
          move     0(a0,d0.w),nextcol0 * ...und als nächster
                                      * HBI-Farbwert
          addq     #2,d0              * Index erhöhen (Wort)
          and      #1023,d0           * zwischen 0 und 1023
                                      * (= 2 * 512 - 1)
          move     d0,colind0         * Index zurückschreiben
          move     colind2,d0         * Anfangsfarbindex für
                                      * Vordergrund
          move     d0,colind2a        * in HBI-Farbindex
                                      * schreiben
          move     0(a0,d0.w),color_2 * Farbwert ins Shifter-
                                      * Register
```

```
          move    0(a0,d0.w),nextcol2 * ...und als nächster
                                      * HBI-Farbwert
          addq    #2,d0               * Index erhöhen (Wort)
          and     #1023,d0            * zwischen 0 und 1023
          move    d0,colind2          * Index zurückschreiben
          move.b  #ZEILEN,tbdr        * alle n Zeilen HBI
          move.b  #8,tbcr             * Timer B im Event-
                                      * Count-Mode
          movem.l savereg,d0/a0       * Register wiederher-
                                      * stellen
oldvbi    jmp     $FFFFFFFF           * zur alten VBI-Routine
*
* Hier endlich die eigentliche HBI-Routine
hbi       move    nextcol0,color_0    * vorher errechnete
                                      * Farbwerte...
          move    nextcol2,color_2    * in Shifter-Register
                                      * schreiben
* Das Eilige ist erledigt. Jetzt können wir in Ruhe die
* nächsten Farbwerte errechnen.
          movem.l d0/a0,savereg       * Register sichern
          lea     farbtab,a0          * Adresse der Farbtabelle
          move    colind0a,d0         * Hintergrund-Farbindex
          addq    #2,d0               * Index erhöhen (Wort)
          and     #1023,d0            * zwischen 0 und 1023
          move    d0,colind0a         * zurückschreiben
          move    0(a0,d0.w),nextcol0* neuer Farbwert für
                                      * nächsten HBI
          move    colind2a,d0         * Vordergrund-Farbindex
          subq    #2,d0               * Index erhöhen (Wort)
          and     #1023,d0            * zwischen 0 und 1023
          move    d0,colind2a         * zurückschreiben
          move    0(a0,d0.w),nextcol2* neuer Farbwert für
                                      * nächsten HBI
          and.b   #$FE,isra           * ISRA Bit 0 löschen
          movem.l savereg,d0/a0       * Register wiederher-
                                      * stellen
          rte                         * Rückkehr

          DATA
colind0   dc.w    0                   * Anfangsfarbindex Hin-
                                      * tergrund
colind2   dc.w    0                   * Anfangsfarbindex Vor-
                                      * dergrund
colind0a  dc.w    0                   * HBI-Farbindex Hinter-
                                      * grund
colind2a  dc.w    0                   * HBI-Farbindex Vorder-
                                      * grund
nextcol0  dc.w    0                   * Farbwert für nächsten
                                      * HBI
nextcol2  dc.w    0                   * Farbwert für nächsten
                                      * HBI

          BSS
savereg   ds.l    4                   * Platz für gesicherte
                                      * Register
```

```
farbtab   ds.w    512                 * Farbtabelle
          END
```

Nun die Beschreibung:

Damit sämtliche 512 Farben des ST gleichzeitig angezeigt werden können, wird zunächst eine Tabelle angelegt, die zu jeder Zahl von 0 bis 511 die Shifter-Darstellung liefert. Man erhält sie, indem man in der binären Darstellung an den Stellen 3 und 7 ein Nullbit dazwischenschiebt. Als nächstes werden – natürlich im Supervisor-Modus – sämtliche Interruptvektoren eingetragen. Außer dem HBI-Vektor brauchen wir noch folgende Vektoren:

- Den VBI-Vektor: Der VBI wird dazu benutzt, die Farbwerte auf einen Anfangswert zurückzusetzen und außerdem für das Rollen der Farben zu sorgen. Damit dies rechtzeitig vorgenommen wird, wird diese Routine nicht wie üblich in die oben erwähnte Vektoren-Tabelle eingetragen, sondern vor die System-Routine gehängt. Das heißt, daß bei Initialisierung die Adresse der System-Routine direkt in den Adreßteil eines JMP-Befehls am Ende dieser Routine geschrieben wird. Dadurch wird sichergestellt, daß nach unseren Befehlen die Routine ausgeführt wird, die vorher für diesen Interrupt zuständig war. Dieses Verfahren nennt sich "vector stealing", auf deutsch auch "Vektoren verbiegen" genannt.

- Der MFP-Vektor für Maus- und Tastaturinterrupts: Auch dieser Interrupt muß umgelenkt werden, da während seiner Abarbeitung normalerweise keine HBIs möglich wären. Das Ergebnis ist ein Flackern, das jedesmal auftritt, wenn eine Taste gedrückt oder die Maus bewegt wird. Der vorgeschaltete Befehl verringert die Interruptmaske auf 5, so daß HBIs weiterhin auftreten können.

Außerdem muß noch der vom Timer C erzeugte 200Hz-Systemtakt abgeschaltet werden, da er die HBIs stören würde.

Nun zu den Interrupts selbst: Die VBI-Routine sorgt dafür, die ersten Farbwerte zu berechnen und in den Variablen nextcol0 und nextcol2 abzulegen. Es werden die Hintergrundfarbe (Farbe 0) und die zweite Vordergrundfarbe verändert. Außerdem werden die den Farben zugeordneten Indizes jeweils um eins hochgezählt, um den Rollo-Effekt zu erreichen. Es wird hier das Timer-B-Datenregister mit 1 initialisiert, damit in jeder Zeile ein Interrupt auftritt. Wenn dieser Wert verändert wird, können Interrupts mit einem beliebigen Abstand auftreten.

Im HBI werden erst die zuvor berechneten Farbwerte in die Shifter-Register übertragen, dann erst werden mit Hilfe der Indizes die neuen Farbwerte berechnet und in nextcol0 und nextcol2 abgelegt. Dies wird deshalb so gemacht, damit schon die ersten beiden Befehle der HBI-Routine die Farben ändern. Geschieht dies erst später, so ist inzwischen schon ein Teil der Zeile in den alten Farben gezeichnet worden, wodurch die Farbe erst mitten in der Zeile verändert würde. Sie sehen also, daß die Farbänderung recht zeitkritisch ist.

Im Vordergrundprogramm wird während des Zeigens der Grafik einfach eine Warteschleife ausgeführt. Wenn sie beendet ist, ist es Zeit, alles wieder so herzustellen, wie es war: Die alten Interruptvektoren werden wiederhergestellt, HBIs werden abgeschaltet, und der 200Hz-Interrupt wird wieder gestattet. Dann ist das Programm fertig und kann wieder zum Desktop zurückkehren.

Es ist wichtig, bei der Beendigung eines Programms alle selbstinstallierten Interruptroutinen wieder zu entfernen, da das nächste Programm, das geladen wird, sicherlich den Speicherbereich überschreibt, in dem die Interruptroutine steht. Falls nicht sämtliche Interruptroutinen wieder "abgehängt" werden, führt dies wahrscheinlich zu einem Systemabsturz, wenn der fragliche Interrupt zum nächsten Mal auftritt.

Es ist wohl überflüssig, zu betonen, daß das Programm nur mit dem Farbmonitor vernünftig läuft. Sie sollten es am besten als GEM-Programm (Endung PRG) ausführen, da so der größte Teil des Bildschirms mit der Farbe 2 gefüllt ist. An den Rändern sieht man außerdem die Hintergrundfarbe 0.

Leider tritt bei dem Programm noch eine kleine ungewollte Nebenerscheinung auf, denn wenn eine Taste betätigt oder die Maus bewegt wird, entsteht ein leichtes Flackern. Es kommt dadurch zustande, daß eine geringe Zeitspanne zwischen der Auslösung des Maus- und Tastatur-Interrupts und dem Zurücksetzen der Interruptmaske auf 5 vergeht. Ein HBI, der in diesem Moment auftreten will, kommt nicht zum Zuge. Die Lösung dieses Problems ist möglich, aber alles andere als einfach, wenn man Tastatur und Maus nicht ganz abschalten will: Man müßte Maus- und Tastaturvorkommnisse nicht durch Interrupts, sondern durch kontinuierliche Abfrage des Interrupt-Pending-Registers selbst ausführen (polling). Dieser Aufwand lohnt sich nur bei umfangreichen Programmen, die von HBIs Gebrauch machen, etwa bei Spielen.

Klangerzeugung durch direkte Amplitudensteuerung

Der Soundchip des ST bietet einige interessante Möglichkeiten. Der normale Weg besteht darin, daß man dem Soundchip eine Frequenz und eventuell noch

einen Lautstärkeverlauf (Hüllkurve) mitteilt und dann erwartet, daß er den Rest alleine tut. Hier soll statt dessen die Möglichkeit beschrieben werden, einen Klang "zu Fuß" zu erzeugen. Damit ist gemeint, daß die CPU selbst die einzelnen Amplitudenwerte eines Klangs berechnet und über den Soundchip ausgibt. Dieses Prinzip findet man auch bei der Wiedergabe von digitalisierten Klängen; Nachteil der Methode ist, daß die CPU damit voll oder zumindest zu einem beachtlichen Teil ausgelastet ist.

Der Soundchip YM-2149 bietet drei unabhängige Tonkanäle A, B und C. Für jeden Tonkanal gibt eine 12-Bit-Zahl die Periodendauer an, also den Kehrwert der Frequenz. Dabei wird eine Frequenz von 500 KHz durch diese 12-Bit-Zahl geteilt, um die Frequenz des Tons zu liefern.

Natürlich ist es unumgänglich, zuerst die Register des Soundchips YM-2149 kennenzulernen. Von außen betrachtet verfügt der Soundchip über nur zwei Register, die jedoch zu nichts anderem da sind, als den Durchgriff auf die wirklichen 16 Register des Soundchips zu ermöglichen. Die beiden äußeren Register haben folgende Bedeutung:

$FF8800 Read Data/Register select (8 Bit)
In diese Adresse wird die Nummer eines der Datenregister des Soundchips (0 – 15) geschrieben. Erst dann hat man Zugriff auf das so angewählte Datenregister. Wenn man dieses Register allerdings ausliest, erhält man den Inhalt des zuletzt angewählten Datenregisters. So greift man beim Lesen auf ein anderes Register zu als beim Schreiben. Bei Hardware-Registern ist dieses Verhalten gar nicht so ungewöhnlich. Tatsächlich kann man dadurch den einen oder anderen Transistor einsparen.

Von Bedeutung ist auch die Tatsache, das der hier hineingeschriebene Wert so lange erhalten bleibt, bis er überschrieben wird; er wird also nicht durch einen Zugriff auf eines der Datenregister geändert. Wenn man mehrmals hintereinander auf das gleiche Datenregister zugreifen möchte, braucht man "Register Select" nicht jedesmal neu zu schreiben.

$FF8802 Write Data (8 Bit)
Hier wird der Wert hingeschrieben, der in das zuletzt ausgewählte Datenregister gelangen soll.

Übrigens bietet der Soundchip nebenbei noch Möglichkeiten zur Steuerung von Ports, die aber in diesem Zusammenhang nicht weiter von Interesse sind. Für uns sind deshalb nur folgende Register von Bedeutung:

0, 1 Diese Register bestimmen die Periodendauer und somit den Kehrwert der Frequenz des Tonkanals A. Dabei werden nur die ersten 4 Bits von Register 0 benutzt. Die 8 Bits des Registers 0 bilden die unteren 8 Bits der 12-Bit-Periodendauer, die 4 Bits des Registers 1 rücken an die Stellen 8 – 11.

2, 3 Entsprechend 0, 1 für Tonkanal B

4, 5 Entsprechend 0, 1 für Tonkanal C

6 Hier bestimmen die Bits 0 – 4 die Periodendauer des Rauschgenerators

7 Mit diesem Register werden alle Tonkanäle kontrolliert. Die einzelnen Bits haben folgende Bedeutung:

0	Tonkanal A	0:ein/1:aus
1	Tonkanal B	0:ein/1:aus
2	Tonkanal C	0:ein/1:aus
3	Rauschen zu Kanal A	0:ein/1:aus
4	Rauschen zu Kanal B	0:ein/1:aus
5	Rauschen zu Kanal C	0:ein/1:aus

8 Die Lautstärke eines Tonkanals kann Werte von 0 bis 15 annehmen. 0 bedeutet abgeschaltet, 15 steht für maximale Lautstärke. Dieser Wert wird in die Bits 0 – 3 geschrieben. Bit 4 hat eine besondere Bedeutung: Ist es gesetzt, dann wird nicht dieser Lautstärkewert benutzt, sondern der Lautstärkeverlauf des Tons wird vom (hier nicht beschriebenen) Hüllkurvenregister bestimmt.

9 entsprechend 8 für Tonkanal B

10 entsprechend 8 für Tonkanal C

Normalerweise erzeugt der YM-2149 also ein Rechtecksignal mit einer angegebenen Frequenz; bestenfalls kann noch über die Hüllkurven eine Dreiecks- oder Sägezahnschwingung erzeugt werden. Was muß man nun tun, um eine Amplitude direkt auszugeben?

- Nur ein Tonkanal wird eingeschaltet (natürlich ohne Rauschen), alle anderen werden abgeschaltet.

- Die Periodendauer des Tonkanals wird auf 0 gesetzt. Dadurch wird eine Schwingung erzeugt, die nicht nur jenseits der menschlichen Wahrnehmung, sondern auch jenseits der Bandbreiten sämtlicher Lautsprecher liegt.

Deshalb wirkt sich diese Schwingung nicht aus, und nur die Lautstärke erscheint am Ausgang.

- Der Amplitudenwert wird in das Lautstärkeregister des Tonkanals geschrieben, wobei das Hüllkurven-Bit nicht gesetzt wird.

Durch diese Maßnahmen wird der Soundchip zum Digital/analog-Wandler degradiert, der nur die hineingeschriebenen Lautstärkewerte in Spannungen am Lautsprecher des Monitors umzuwandeln hat.

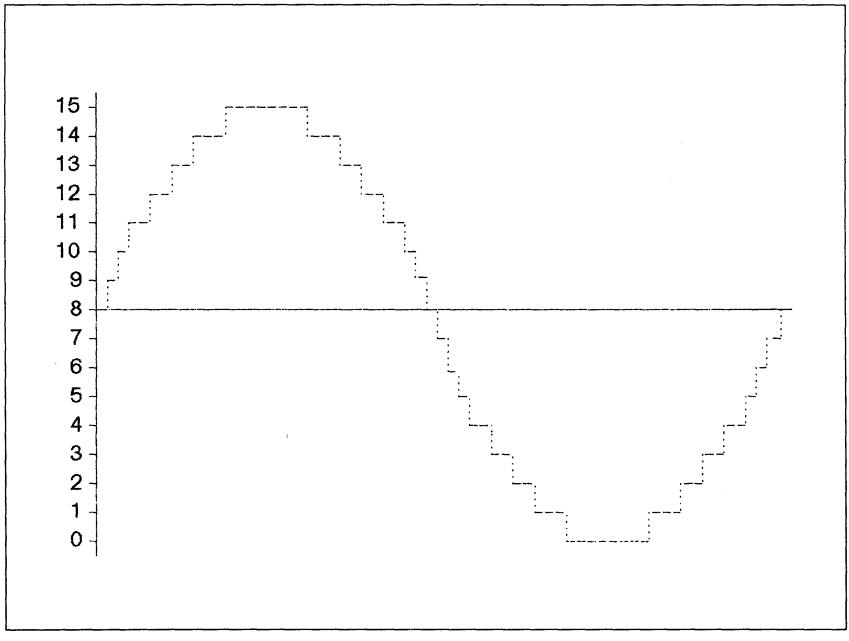


Abb. 6.5: Umsetzung einer Sinusschwingung in digitale Daten

Um das Prinzip zu demonstrieren, muß man allerdings auch noch irgend etwas haben, was man ausgeben kann. Als Beispiel soll hier eine angenäherte Sinusschwingung dienen (Abb. 6.5). Das dort gezeigte Diagramm wird für das folgende Programm einfach in eine Werteliste von 64 Bytes umgewandelt – Digitalisierung per Hand.

Eines gilt es bei dieser Art der Tonerzeugung zu beachten, wenn man eine annehmbare Klangqualität erreichen will: Es müssen alle Interrupts radikal ab-

geschaltet werden, da sonst die Tonerzeugung ständig unterbrochen würde, besonders vom VBI. Deshalb wird am Anfang die Interruptmaske auf 7 gesetzt. Probieren Sie ruhig einmal aus, wie es sich auswirkt, wenn man die Interrupts nicht abschaltet.

Zunächst wird in den Supervisor-Modus geschaltet, und die Register des Soundchips werden initialisiert. Dabei achtet das Programm darauf, die Bits 6 und 7 des Soundchip-Registers 7 nicht zu verändern, da sie wichtigen Portsteuerungszwecken dienen.

Nun zur eigentlichen Tonschleife: Um die momentane Stelle in der Werteliste festzuhalten, bewahrt das Programm in D0 einen Index auf. Dieser wird mit der Anweisung "AND #63,D0" nach Additionen immer im richtigen Wertebereich gehalten, wodurch die Sinusschwingung zyklisch ausgegeben wird. Wollte man nun in jedem Schleifendurchlauf ganze Zahlen zu D0 addieren, so erhielte man eine sehr hohe und auch nur in großen Schritten änderbare Frequenz – was übrigens die Geschwindigkeit zeigt, mit der die einzelnen Werte ausgegeben werden. Deshalb findet hier wieder das schon beim Line-Algorithmus benutzte Festkomma-Prinzip Verwendung: Der Nachkommateil des Index (16 Bit) wird in D1 gespeichert. So hat man auch die Möglichkeit, den Index jedesmal um kleinere Werte als 1 zu erhöhen. In der Schleife wird gleichzeitig noch ein Zähler mitgeführt, damit das Programm auch irgendwann endet.

```
*************************************************************************
*  SOUND.S                                                              *
*  Demo für Sounderzeugung durch direkte Amplitudensteuerung            *
*  erzeugt eine "weiche" Schwingung                                     *
*    Registerbelegung                                                   *
*    D0.W      Index in Wertetabelle (0-63)                             *
*    D1.W      Nachkommateil des Index D2.W                             *
*    D2.W      Additionswert des Index D0                               *
*    D3.W      Nachkommateil des Additionswertes D2                     *
*    D4.L      Zähler der Schleifendurchläufe, damit das Programmm      *
*              auch terminiert                                          *
*************************************************************************

nreg      EQU     $FFFF8800          * Soundchip-Registernummer
value     EQU     $FFFF8802          * hier Wert des Registers schreiben

SUPER     EQU     $20

start     clr.l   -(sp)             * Userstack als Supervisorstack
          move.w  #SUPER,-(sp)      * GEMDOS-Funktionsnummer
          trap    #1                * Sprung ins GEMDOS
          addq.l  #6,sp             *
          move.l  d0,save_ssp       * SUPER-Stack merken
          bsr.s   reginit           * Register des Soundchips init.
```

```
          or      #$700,SR            * alle Interrupts abschalten
          lea     sintab,a0           * Adresse der Werteliste
          move.b  #8,nreg             * Registernummer 8: Lautstärke A
          move    #0,d0               * Index auf 0
          move    inch,d2             * Additionswert unteres Wort
          move    incl,d3             * Additionswert oberes Wort
          move.l  #1000000,d4         * 1.000.000mal durch die Schleife
outloop   move.b  0(a0,d0.w),value    * Lautstärkewert ausgeben
          add     d3,d1               * d1: niederwertiger Teil des Index
          addx    d2,d0               * d0: hochwertiger Teil des Index
          and     #63,d0              * immer in der Liste bleiben
          subq.l  #1,d4               * Schleifendurchläufe herunter-
                                      * zählen
          bne.s   outloop             * bis Null erreicht ist
loopend   move.b  #0,value            * Lautstärke auf 0
          and     #$FBFF,SR           * Maske 3 (Bit 10 des SR auf 0)
          move.l  save_ssp,-(sp)      * alten SUPERVISOR-Stack holen
          move    #SUPER,-(sp)        * Funktionscode
          trap    #1                  * ins GEMDOS
          addq.l  #6,sp               *
          clr     -(sp)               * TERM Funktionscode 0
          trap    #1                  * zurück zum Desktop!
*
* hier werden die Register des Soundchips initialisiert
reginit   move.b  #0,nreg             * Frequenz A Low auf 0
          move.b  #0,value            *
          move.b  #0,nreg             * Frequenz A High auf 0
          move.b  #0,value            *
          move.b  #7,nreg             * Multifunktionsregister Nummer 7
          move.b  nreg,d0             * Wert auslesen
          and.b   #%11000000,d0       * Bits 6 und 7 erhalten
          or.b    #%00111110,d0       * Nur Kanal A ein, Rauschen aus
          move.b  d0,value            * und zurückschreiben
          rts                         *

          DATA
incl      dc.w    20000               * Inkrement Low: 20000/65536
inch      dc.w    0                   * Increment Hi:  0/1
save_ssp  dc.l    0                   * zum Speichern des SSP
* und die Wertetabelle
sintab    dc.b    08,09,10,11,11,12,12,13,13,14,14,14,15,15,15,15
          dc.b    15,15,15,15,14,14,14,13,13,12,12,11,11,10,09,08
          dc.b    07,06,05,04,04,03,03,02,02,01,01,01,00,00,00,00
          dc.b    00,00,00,00,01,01,01,02,02,03,03,04,05,06,07,08
          END
```

Wenn Sie mit diesem Prinzip etwas herumexperimentieren wollen, so ändern Sie doch einmal die Frequenz, oder fügen Sie in der Tonschleife irgendwo folgenden Befehl ein:

```
ADDQ #1,D0
```

Tatsächlich erzeugt der Soundchip mit diesen Klangdaten keine echte Sinusschwingung; in Wirklichkeit ergibt sich eine etwas komplexere Schwingungs-

form, da die Lautstärkestufen des Soundchips nicht einfach linear gestuft sind, sondern logarithmisch. Das heißt, daß etwa der tatsächliche Lautstärkeabstand zwischen den Lautstärken 14 und 15 viel größer ist als der zwischen 1 und 2. Die Grundidee dabei ist, daß das menschliche Gehör die Lautstärken auch logarithmisch wahrnimmt; Ihnen werden also – rein subjektiv – diese Lautstärkenunterschiede gleich vorkommen. In unserem Beispiel wirkt sich das kaum störend aus, da trotzdem noch ein recht weich klingender Ton entsteht. Für die Wiedergabe von digitalisierten Klängen ist es allerdings empfehlenswert, eine Liste für die Übersetzung der linear angegebenen Lautstärkewerte in die logarithmischen Werte des Soundchips zu verwenden. Somit wäre für eine größere Klangtreue gesorgt.

Natürlich könnte man an diesem Beispiel noch einige Verbesserungen vornehmen:

- Statt nur eines Tonkanals könnten drei für ein einziges Signal verwendet werden, um die Wiedergabetreue zu verbessern. Besonders für das Abspielen von digitalisierten Klängen empfiehlt sich diese Methode. Es ist sinnvoll, beim Ansteuern der drei Kanäle die schon genannte Tatsache zu berücksichtigen, daß die Lautstärkewerte des Soundchips logarithmisch gestuft sind.

- Statt einer Schleife könnte man den Timer-A-Interrupt verwenden (Beschreibung siehe in diesem Kapitel unter "Programmierung von Interrupts"). So wäre die CPU nicht restlos damit ausgelastet, ein paar Amplitudenwerte auszugeben, sondern könnte im Vordergrund andere sinnvolle Dinge tun. Für diesen Zweck könnte die Klangausgaberoutine auch noch etwas auf Geschwindigkeit optimiert werden. Probleme würden allerdings mit anderen Interrupts auftreten, die eine höhere Priorität als der Timer-A-Interrupt haben.

Eine RAM-Disk

Dieser Abschnitt soll das Interessante mit dem Nützlichen verbinden, denn eine RAM-Disk kann den Umgang mit einem Assembler sehr erleichtern, sofern man genug Speicherplatz hat. Das Beispiel liefert gleichzeitig einen Anhaltspunkt dafür, wie man dem Betriebssystem Gerätetreiber zugänglich machen kann.

Um eine RAM-Disk installieren zu können, muß man zunächst einmal wissen, wie der Zugriff auf Laufwerke im Betriebssystem organisiert ist, denn

schließlich hat eine RAM-Disk ja den Status eines Laufwerks. Nun, die Dateiverwaltung bleibt völlig dem GEMDOS überlassen und soll uns deshalb hier nicht weiter interessieren. Interessant wird es erst in dem Moment, wo das GEMDOS auf die einzelnen Sektoren des Laufwerks zugreifen will. Auf dem ATARI ST sind Disketten normalerweise auf jeder Seite in 80 Spuren (Tracks) zu jeweils 9 Sektoren organisiert; die Diskettenlaufwerke können jeweils nur einen vollständigen Sektor auf einmal lesen oder schreiben. Die recht komplizierte Aufgabe, diese Sektoren nun so zu verwalten, daß Dateien mit einer beliebigen Anzahl von Bytes gespeichert werden können, obliegt dem GEMDOS. Für den Zugriff auf einzelne Sektoren benutzt das GEMDOS die BIOS-Funktion Nummer 4 mit der Bezeichnung "rwabs" für "read/write absolute sectors", die direkt auf Sektoren eines Laufwerks zugreift. Aus einer höheren Programmiersprache wie C würde "rwabs" folgendermaßen aufgerufen:

```
error=rwabs(rwflag,puffer,anzahl,sektor,dev)
```

Betrachten wir nun die Parameter, die allesamt außer "puffer" Worte sind:

"rwflag" gibt an, ob Sektoren geschrieben oder gelesen werden sollen. Der Parameter kann folgende Werte annehmen:

0 Sektoren lesen
1 Sektoren schreiben
2 Sektoren lesen, Diskettenwechsel ignorieren
3 Sektoren schreiben, Diskettenwechsel ignorieren

Bekanntlich sind die Diskettenlaufwerke des ATARI ST in der Lage, einen Diskettenwechsel zu erkennen und ihn dem Computer zu signalisieren. Auf diese Möglichkeit wurde hier Rücksicht genommen.

Schreiben und lesen werden hier in eine Routine zusammengepackt, da beide Operationen oft einen großen Teil ihres Codes gemeinsam haben.

"puffer" ist ein Langwort, das die Adresse des Bereiches angibt, aus dem gelesen bzw. in den geschrieben wird. Er sollte an einer geraden Adresse beginnen, da sonst der Zugriff auf das Laufwerk etwas verlangsamt wird.

"anzahl" gibt an, wie viele Sektoren nacheinander geschrieben oder gelesen werden sollen.

"sektor" gibt die Nummer des ersten Sektors an, der geschrieben oder gelesen werden soll.

"dev" schließlich gibt das Laufwerk an, das bei der Operation benutzt werden soll. Dabei steht 0 für Laufwerk A, 1 für B, 2 für C und so weiter.
Der Rückgabewert ist null, wenn der Zugriff erfolgreich war, andernfalls erhält man eine negative Fehlernummer.

Für die Verwaltung eines Laufwerks sind noch zwei weitere BIOS-Funktionen von Bedeutung: getbpb (Nummer 7) und mediach (Nummer 9).

```
getbpb(dev)
```

"dev" hat die gleiche Bedeutung wie der entsprechende Parameter bei rwabs(). Diese Funktion bewirkt nichts anderes, als einen Zeiger auf eine Datenstruktur zurückzugeben, die alle wichtigen Informationen über das Laufwerk bzw. die Diskette enthält, den BIOS-Parameter-Block. Unter anderem steht dort, wie viele Bytes ein Sektor enthält (512), wie viele Sektoren in einen Cluster gehören (2) und wo die Verwaltungsinformationen auf der Diskette stehen. Das meiste kann bei der RAM-Disk einfach vom BIOS-Parameter-Block einer normalen Diskette übernommen werden. Interessant ist für uns in erster Linie der Eintrag im BIOS-Parameter-Block, der die Anzahl der insgesamt auf dem Laufwerk verfügbaren Cluster angibt. Ein Cluster ist ein logischer Block, der auf dem ST 2 Sektoren mit insgesamt 1024 Bytes umfaßt; an dieser Stelle wird also die Kapazität des Laufwerks in Kilobytes eingetragen.

Die letzte für ein Laufwerk zu realisierende Funktion wird so aufgerufen:

```
ergebnis=mediach(dev)
```

"dev" gibt wieder die Laufwerksnummer an. Der Rückgabewert dieser Funktion zeigt an, ob bei dem angegebenen Laufwerk ein Diskettenwechsel stattgefunden hat. Diese Information ist folgendermaßen codiert:

0 Diskette wurde nicht gewechselt
1 Diskette könnte gewechselt worden sein
2 Diskette wurde gewechselt

Logischerweise kann bei einer RAM-Disk kein Diskettenwechsel auftreten.

Jetzt wissen wir also, welche Funktionen wir für einen Gerätetreiber implementieren müssen. Es geht nun nur noch darum, dem Betriebssystem mitzuteilen, wo unsere Routinen zu finden sind.

Unter den Systemvariablen befinden sich drei Vektoren mit den Namen hdv_bpb, hdv_rw und hdv_mediach, die Vektoren zu den Routinen getbpb(),

rwabs() und mediach() enthalten. Man hat also nur noch die Adressen der eigenen Routinen dort einzutragen. Doch halt: Natürlich darf man die Routinen für die normalen Diskettenlaufwerke und eventuell für eine Festplatte nicht einfach abhängen. Wir benutzen also wieder das Verfahren des "vector stealing" und verzweigen zu den alten Routinen, wenn ein anderes Laufwerk als die RAM-Disk angesprochen wird.

Eines gibt es noch zu tun, bevor unser Laufwerk offiziell beim Betriebssystem angemeldet ist: Das der Laufwerksnummer entsprechende Bit in der Systemvariablen "drvbits" muß gesetzt werden, sonst wird die verwendete Laufwerkskennung nicht anerkannt.

Um die Verwaltung der Dateien und freien Blocks auf der RAM-Disk brauchen wir uns nicht weiter zu kümmern; Eine RAM-Disk zu "formatieren" heißt lediglich, sämtliche Bytes auf 0 zu setzen.

Das folgende Programm installiert bei seiner Initialisierung zunächst einmal die drei besprochenen Vektoren und meldet das Laufwerk in "drvbits" an, was natürlich alles im Supervisor-Modus ablaufen muß. Dann wird die Gesamtlänge des Programms berechnet. Das Programm wird nicht wie üblich mit der Funktion TERM beendet, sondern mit dem ähnlichen KEEPTERM. Diese Funktion sorgt zwar auch für eine Rückkehr zum Desktop, reserviert aber einen Speicherbereich ab der Startadresse des Programms, dessen Länge KEEPTERM als Parameter überreicht wird. Hier wird dafür einfach die Gesamtlänge des Programms und der RAM-Disk berechnet, indem die Längen der Basepage, der drei Programmsegmente und des Speicherbereichs für die RAM-Disk addiert werden.

Allerdings prüft das Programm nicht, ob noch genügend Speicher vorhanden ist; wenn Sie mehr Kilobytes für die RAM-Disk reservieren wollen, als tatsächlich noch frei sind, ist das Resultat ein Busfehler.

Nun zu den drei Routinen selbst: Bei jeder Routine wird zunächst einmal abgefragt, ob überhaupt das Laufwerk der RAM-Disk angesprochen wird. Ist das nicht der Fall, dann wird gleich zu der Adresse gesprungen, die ursprünglich im entsprechenden Vektor stand. Wenn Sie nichts daran ändern, wird die RAM-Disk automatisch auf Laufwerk C installiert. Wollen Sie sie unter einer anderen Laufwerkskennung benutzen, so brauchen sie nur den Wert des Symbols RAMDISK entsprechend abzuändern.

Da die Anordnung der Parameter bei rwabs() vielleicht etwas unübersichtlich ist, hier eine Aufstellung ihrer Positionen auf dem Stack nach dem Aufruf:

Adresse	Länge	Name
14(SP)	Wort	dev, Gerätenummer
12(SP)	Wort	sektor, Nummer des ersten Sektors
10(SP)	Wort	anzahl, Anzahl der Sektoren
6(SP)	Langwort	puffer, Adresse des Datenpuffers
4(SP)	Wort	rwflag, Flag für lesen/schreiben
0(SP)	Langwort	Rückkehradresse

Und das hat die rwabs-Funktion zu tun: Sie muß aus der angegebenen Sektornummer die Adresse in der RAM-Disk berechnen, und zwar nach der Formel

adr = Ramdiskadresse + Sektornummer * 512

Dann muß nur noch die Anzahl der zu kopierenden Bytes ausgerechnet werden:

Bytes = anzahl * 512

Um den Zugriff zu beschleunigen, findet der Kopiervorgang langwortweise statt. In dem Fall gilt die Formel

Langworte = anzahl * 512 / 4 = anzahl * 128

Nur der seltene Fall, daß die Adresse des Datenpuffers ungerade ist, muß wie oben byteweise gehandhabt werden.

Wenn Sie sich die Routine ansehen, werden Sie feststellen, daß Schreib- und Lesevorgang weitgehend gleich gehandhabt werden. Tatsächlich besteht der einzige Unterschied darin, daß für das Schreiben die Inhalte des Quell- und Zielregisters ausgetauscht werden.

Wie benutzt man die RAM-Disk nun? Zunächst muß die Laufwerkskennung der RAM-Disk beim Desktop angemeldet werden. Klicken Sie dazu ein beliebiges Laufwerkssymbol an und fahren Sie auf den Menüpunkt "Floppy anmelden". Tippen Sie nun die Laufwerkskennung (C) ein, und klicken Sie das Feld "anmelden" an. Wenn Sie wollen, können Sie vorher in das Feld mit der Laufwerksbezeichnung "RAM-Disk" oder etwas ähnliches eintragen; es schadet aber auch nichts, wenn Sie es nicht tun.

Natürlich sollten Sie die Größe der RAM-Disk (KBytes) an Ihre eigenen Bedürfnisse anpassen (das heißt, eigentlich an die Gegebenheiten Ihres Systems).

Wenn das RAM-Disk-Programm in einem AUTO-Ordner abgelegt wird, wird es bei jedem Systemstart oder Reset automatisch in den Speicher geladen.

Sicherlich kann es ärgerlich sein, daß bei einem schweren Systemabsturz – der ja bei der Assemblerprogrammierung nicht gerade schwierig zu erzeugen ist – der Inhalt der RAM-Disk verloren ist. Es gibt eine Lösung für dieses Problem: Einige aufwendigere RAM-Disk-Programme sind in der Lage, den Inhalt der RAM-Disk über ein Reset hinwegzuretten. Leider ist der dazu notwendige Aufwand so groß, daß er den Rahmen dieses Buchs sprengen würde. Wenn Sie auf die hier abgedruckte RAM-Disk angewiesen sind, kann ich nur empfehlen, häufiger Sicherheitskopien auf einem weniger flüchtigen Medium anzulegen.

```
*****************************************************************************
*    RAMDISK.S                                                              *
*    Ramdisk-Programm                                                       *
*****************************************************************************

DRIVE     EQU     2                 * 2 für Laufwerk C:
KBYTES    EQU     512               * Größe in 1K-Blöcken

* Base-Page-Adressen
textlen   EQU     12                * Länge des Textsegments in Bytes
datalen   EQU     20                * Länge des Datensegments
bsslen    EQU     28                * Länge des BSS-Segments

* absolute Adressen einiger Systemvariablen
hdv_bpb   EQU     $472              * Vektor für BIOS getbpb()
hdv_rw    EQU     $476              * Vektor fur BIOS rwabs()
hdv_media EQU     $47E              * Vektor für BIOS mediach()
drvbits   EQU     $4c2              * Bits für angeschlossene Laufwerke

* GEMDOS Funktionsnummer
KEEPTERM  EQU     49                * Programmende mit reservieren

* XBIOS Funktionsnummer
SUPEXEC   EQU     38                * Routine im SUPER-Modus ausführen

start     move.l  4(sp),a0          * Basepage-Adresse nach A0
          moveq   #0,d1             * Ramdisk mit Nullen füllen
          lea     rdstart,a0        * Anfangsadresse
          move.l  #KBYTES*1024/4,d0 * Zähler für Langworte
clrloop   move.l  d1,(a0)+          * Langworte löschen
          subq.l  #1,d0             *
          bne     clrloop           *
          move.l  #$100,d7          * 256 Bytes für Basepage...
          add.l   textlen(a0),d7    * + Länge des Textsegments...
          add.l   datalen(a0),d7    * + Länge des Datensegments
          add.l   bsslen(a0),d7     * + Länge des BSS in D7
          add.l   #KBYTES*1024,d7   * + Länge der RAM-Disk
          pea     supinit           * Adresse der Init-Routine auf Stack
```

```
          move    #SUPEXEC,-(sp)      * im Supervisor-Modus ausführen
          trap    #14                 * zum BIOS
          addq.l  #6,sp               *
          clr     -(sp)               * exit code = 0 (OK)
          move.l  d7,-(sp)            * Gesamtlänge auf Stack
          move    #KEEPTERM,-(sp)     * Programmende mit Reservieren des
          trap    #1                  * Speicherbereiches

supinit   move.l  hdv_bpb,oldbpb+2    * alten getbpb()-Vektor merken
          move.l  #bpb,hdv_bpb        * eigene Routine dazwischenschalten
          move.l  hdv_rw,oldrw+2      * alten rwabs()-Vektor merken
          move.l  #rw,hdv_rw          * eigene Routine dazwischenschalten
          move.l  hdv_media,oldmedia+2 * alten mediach()-Vektor merken
          move.l  #media,hdv_media    * eigene Routine dazwischenschalten
          move.l  drvbits,D0          * Laufwerk "C:" anmelden
          bset    #DRIVE,D0           * Bit setzen
          move.l  D0,drvbits          * und zurückschreiben
          rts                         *

bpb       cmp     #DRIVE,4(sp)        * ist die RAM-Disk gemeint?
          bne.s   oldbpb              * nein, zur alten Routine
          move.l  #rdbpb,d0           * Adresse des BIOS Parameter Blocks
          rts
oldbpb    jmp     $FFFFFFFF           * Hierhin den alten hd_bpb-Vektor

rw        cmp     #DRIVE,14(sp)       * "C:" angesprochen?
          bne.s   oldrw               * nein, springe durch alten Vektor
          move.l  6(sp),a0            * zu schreibender/lesender Bereich
          lea     rdstart,a1          * Anfang des RAM-Disk-Speichers
          clr.l   d0                  * D0 säubern
          move    12(sp),d0           * erster Block
          moveq   #9,d1               *
          lsl.l   d1,d0               * Bytes=Blocks*512
          add.l   d0,a1               * zur RAM-Disk-Anfangsadr. addieren
          clr.l   d0                  * d0 noch einmal säubern
          move    10(sp),d0           * Anzahl der Blocks nach D0
          move.l  a0,d1               *
          btst    #0,d1               * test auf ungerade Startadresse
          bne.s   bytes               * zur langsameren Byte-Schleife
          asl.l   #9-2,d0             * Bytes=Blocks*512; Longs=Bytes/4
          btst    #0,5(sp)            * schreiben oder lesen?
          beq.s   longs               * lesen!
          exg     a0,a1               * in umgekehrter Richtung
                                      * übertragen
longs     move.l  (a1)+,(a0)+         * je ein Langwort übertragen...
          subq.l  #1,d0               * bis der Zähler...
          bne.s   longs               * auf 0 steht
          rts                         *

bytes     moveq   #9,d1               * langsameres Byte-übertragen
          lsl.l   d1,d0               * Bytes=Blocks/512
          btst    #0,5(sp)            * lesen oder schreiben?
          beq.s   byteloop            * lesen!
          exg     a0,a1               * in umgekehrter Richtung
```

```
                                        * übertragen
byteloop  move.b  (a1)+,(a0)+           * byteweise kopieren
          subq.l  #1,d0                 *
          bne.s   byteloop              *
          rts
oldrw     jmp     $FFFFFFFF             * zur alten rwabs()-Routine

media     cmp     #DRIVE,4(sp)          * unsere Drive-Kennung?
          bne.s   oldmedia              * nein, zu alten mediach()-Routine
          moveq   #0,d0                 * 0: die RAM-Disk kann natürlich
          rts                           *    nicht gewechselt werden
oldmedia  jmp     $FFFFFFFF             * zur alten mediach()-Routine

          DATA
* BIOS-Parameter-Block der Ramdisk
rdbpb     DC.W    512                   * Bytes pro Record(Block)
clsiz     DC.W    2                     * Blocks pro Cluster
clsizb    DC.W    1024                  * Bytes pro Cluster
rdlen     DC.W    7                     *
fsiz      DC.W    5                     *
fatrec    DC.W    6                     * Blockanzahl der FAT
datrec    DC.W    18                    * Blockanzahl des Directory
numcl     DC.W    KBYTES-18/2           * Gesamtgröße in Clusters
flags     DC.W    8                     *

rdstart   BSS
* Mit dem BSS wird Platz für die RAM-Disk reserviert
          END
```

Kapitel 7

Tips und Tricks für schnellere Programme

Manchmal kommt es vor, daß Programme oder Programmteile trotz der Implementierung in Assembler nicht schnell genug sind. Wenn der Such- oder Sortieralgorithmus Minuten braucht, wenn das Grafikprogramm den Bildschirm nicht schnell genug aufbaut, oder wenn sich die Figuren für das neue Computerspiel nur wiederwillig und mit Flackern bewegen, dann ist ein schnelleres Programm nötig.

Tatsächlich gibt es auch in Assembler noch eine ganze Menge Möglichkeiten, Programme schneller zu machen. Deshalb sollen hier die wichtigsten und allgemein anwendbaren Methoden vorgestellt werden.

Es wird ausschließlich auf die Geschwindigkeitsoptimierung eingegangen, da Speicherplatz auf einem Computer wie dem ATARI ST zumindest in Assembler wohl kaum ein Problem darstellt. In manchen Fällen wird sogar Speicherplatz für Geschwindigkeit geopfert.

Die verschiedenen Möglichkeiten werden auf drei Ebenen aufgeteilt:

- Die Befehlsebene
  Dies ist die unterste Ebene. Die hier angebotenen Optimierungen betreffen einzelne Maschinensprachebefehle, denn oft bietet der MC68000 Dank seines reichhaltigen Befehlssatzes mehr als eine Möglichkeit, eine bestimmte Aktion durchzuführen. Oft unterscheiden sich diese Möglichkeiten in ihrer Geschwindigkeit. Hierbei handelt es sich allerdings oft um rein mechanische Optimierungen, die auch von Compilern vorgenommen werden können.

- Die Implementierungsebene
  Jetzt sind wir schon eine Stufe höher. Hier geht es um Gruppen von Befehlen, um Schleifen und Programmorganisation. Trotzdem ist es noch eine sehr maschinenabhängige Optimierung. Allerdings erfordert sie schon einige menschliche Intelligenz; für Compiler sind diese Methoden oft schon zu kompliziert darzustellen.

- Die Algorithmenebene
  Dies ist die höchste Ebene des Programmierens – wenn man einmal von der Modellebene absieht, die eher von theoretischer Bedeutung ist. Natürlich

kann auf dieser Ebene nur eine Auswahl von allgemein anwendbaren Optimierungsmethoden vorgestellt werden, denn die hier denkbaren Methoden sind so vielfältig wie die Probleme, die auf Computern gelöst werden.

Bevor man mit dem Optimieren beginnt, ist es das Wichtigste, den Blick für das Wesentliche zu haben. Natürlich hat es keinen Sinn, Befehlssequenzen zu optimieren, die während des Programmablaufs nur einige oder auch einige hundert Male durchlaufen werden. So kommt es zunächst einmal darauf an, zu wissen, welche Teile des Codes den Hauptanteil der Rechenzeit benötigen. Den meisten Programmen sieht man das schon von vornherein an; wenn Sie jedoch ein besonders kompliziertes Programm mit vielen geschachtelten Unterprogrammaufrufen geschrieben haben, können Sie in die fraglichen Routinen Zähler einbauen, die Ihnen Aufschluß darüber geben, wie oft sie in einem Programmlauf aufgerufen wurden.

Bevor es richtig losgeht, noch ein Wort der Warnung: Die Optimierung von Programmen bringt auch Nachteile mit sich. Man muß die Verbesserung einer Eigenschaft fast immer mit Nachteilen auf anderen Gebieten bezahlen – dieses Gesetz gilt nicht allein für Computerprogramme, sondern auch für fast alle anderen Bereiche der Technik. Leider verhält es sich nun einmal so, daß "schnelles Programm" und "strukturiertes Programm" oft Gegensätze sind. Sie müssen von Fall zu Fall selbst entscheiden, ob der Gewinn an Geschwindigkeit wirklich den Mehraufwand bei späteren Änderungen des Programms wert ist.

Tatsächlich ist ein perfekt optimiertes größeres Programm für irgend jemand anderen außer dem Autor kaum noch verständlich. Deshalb ist die Optimierung besonders dann mit Vorsicht zu genießen, wenn später vielleicht ein anderer Programmierer das Programm verstehen soll. Aber auch, wenn Sie das Programm nur selbst verstehen müssen, empfehle ich Ihnen, ungewöhnliche Methoden der Optimierung zu dokumentieren, damit Sie Ihre eigenen Tricks auch einen Monat später noch verstehen.

Optimierungen auf Befehlsebene

Die folgenden Optimierungen sind vom Standpunkt der Klarheit aus noch relativ harmlos, da leicht verständlich. Es kann nichts schaden, wenn Sie sich mit der Zeit angewöhnen, diese Optimierungen gleich beim Niederschreiben von Programmen zu verwenden. Allerdings sind unter den folgenden Methoden einige schwarze Schafe, die dazu neigen, Programme unübersichtlich zu machen; auf sie wird gesondert hingewiesen.

Allgemeine Optimierungen

Wenn Sie es mit der Optimierung auf Befehlsebene ernst meinen, sollten Sie zunächst einmal einen Blick auf die Ausführungszeiten der Befehle im Anhang F werfen. Alle Ausführungszeiten sind in Taktzyklen angegeben. Beim ST, der ja mit 8 MHz getaktet ist, dauert der Ablauf eines Taktzyklus 125 Nanosekunden. Bald wird Ihnen auffallen, daß es auf dem MC68000 keinen Befehl gibt, der weniger als 4 Taktzyklen benötigt. Im Vergleich zu einfacheren Prozessoren wie etwa dem 6502 ist das relativ viel. Der Grund liegt in der komplexen Architektur des MC68000.

Ein weiteres Merkmal fällt sofort ins Auge: Sobald sich der Befehl nicht allein in Prozessorregistern abspielt, erhöht sich die Ausführungszeit drastisch auf mindestens 8 Taktzyklen. Die CPU kann ihre eigenen Register sehr viel schneller erreichen als Speicherstellen. Daraus läßt sich gleich eine wichtige Regel ableiten: Oft benutzte Variablen sollten möglichst in Registern aufbewahrt werden. Es empfiehlt sich schon deshalb, weil die meisten Operationen verlangen, daß sich mindestens einer der Operanden in einem Datenregister befindet.

Nachteilig ist, daß es bei der Vielzahl der Register des MC68000 recht unübersichtlich werden kann, was nun eigentlich in welchem Register steht. Deshalb bieten einige Assembler die nicht vom Motorola-Standard vorgeschriebene Direktive REG, die ein Label mit einem Register identifiziert:

```
zaehler REG D5
```

Wann immer man nun das Label "zaehler" verwendet, wird dafür D5 eingesetzt. Im Extremfall, wenn Sie sehr viele Registervariablen verwenden, ist das allerdings auch nicht ganz unproblematisch. Dann verdecken die Label, in welchem Register sich eine Variable nun wirklich befindet, wann welche Register zur Verfügung stehen und ob sich nicht etwa zwei Registervariablen überschneiden. Es ist also in jedem Fall Aufmerksamkeit geboten, wenn viele Variablen gleichzeitig in Registern aufbewahrt werden.

Aus den Ausführungszeiten ergibt sich noch folgendes: Um so komplizierter die verwendete Adressierungsart ist, desto länger dauert auch die Befehlsausführung. Die einfachste Adressierungsart von Speicherzellen ist "Adreßregister indirekt". Nur mit dieser Variante läßt sich die minimale Ausführungszeit von 8 Taktzyklen erreichen. Deshalb werden alle anderen Speicher-Adressierungsarten mit der genannten verglichen. Die zusätzliche Ausführungszeit wird in Taktzyklen angegeben:

Adressierungsart	Zusätzliche Taktzyklen
(An)	0
(An)+	0
-(An)	2
d16(An)/d16(PC)	4
d8(An,i)/d8(PC,i)	6
absolut 16 Bit	4
absolut 32 Bit	8

Mit "d16" wird dabei ein 16-Bit-Displacement bezeichnet, mit "d8" ein 8-Bit-Displacement. "i" steht für ein Indexregister.

Wenn man Listen bearbeiten will, sollte man sie also besser von unten nach oben mit der Postinkrement-Adressierungsart durchgehen als umgekehrt mit Predekrement. Ersteres ist ohnehin naheliegender.

Aus dieser Tabelle kann man auch ablesen, daß es günstiger ist, ein Feld mit Hilfe eines Zeigers durchzugehen als mit einem Index unter Verwendung der Adressierungsart "Adreßregister indirekt mit Index und Displacement".

Noch ein Wort zur Verarbeitungsbreite: Byte- und Wort-Befehle nehmen sich in der Ausführungsgeschwindigkeit nichts. Das rührt daher, daß der Datentransport seine Zeit braucht, egal ob der 16 Bit breite Datenbus vollständig oder nur zur Hälfte genutzt wird. Nur die Langwortoperationen fallen aus der Reihe, denn für sie muß der Datenbus zweimal bemüht werden. Eine Langwortoperation ist allerdings immer schneller als zwei gleichwertige Wortoperationen. Deshalb sollte man für Berechnungen und Datentransporte immer die höchste anwendbare Verarbeitungsbreite wählen.

Optimierung von MOVE-Befehlen

Betrachten wir zunächst, wie man am schnellsten einen konstanten Wert in ein Register bringt. Die mit Abstand effizienteste Methode bietet der MOVEQ-Befehl. Er benötigt nur 4 Taktzyklen, um die vollen 32 Bit eines Datenregisters mit einem vorzeichenerweiterten 8-Bit-Wert zu füllen. Natürlich ist seine Verwendung begrenzt, denn MOVEQ läßt als Ziel nur ein Datenregister zu.

Es kommt oft vor, daß eine Adresse in ein Adreßregister gebracht werden soll. Häufig geschieht das in dieser Form:

```
MOVE.L #adr,An
```

Genau die gleiche Zeit benötigt folgende Form (Beachten Sie das fehlende Doppelkreuz):

```
LEA adr,An
```

Nach der zuvor gezeigten Tabelle werden jedoch 4 Taktzyklen gespart, wenn das Label PC-relativ adressiert wird (Voraussetzung ist natürlich, daß die Adreßdistanz in 16 Bit darstellbar ist):

```
LEA adr(PC),An
```

Manche Assembler verwenden allerdings (zumindest optional) automatisch die PC-relative Adressierungsart, wenn sie anwendbar ist.

Der häufigste unmittelbare Wert, den man in ein Register bringen will, ist sicherlich 0. Naheliegenderweise würde man dafür den CLR-Befehl verwenden:

```
CLR.L Dn
```

Tatsächlich ist aber hier MOVEQ um zwei Taktzyklen schneller:

```
MOVEQ #0,Dn
```

Dies gilt allerdings nur für Langworte; bei Worten und Bytes nehmen sich die beiden Varianten nichts.

Genauso häufig dürfte es vorkommen, daß man Speicherzellen auf 0 setzen will. Will man ganze Speicherbereiche "ausnullen", so ist der CLR-Befehl auch nicht optimal. Man spart bei Wortbreite 4 und bei Langwortbreite sogar 8 Taktzyklen pro Speicherzugriff, wenn man vorher ein Datenregister löscht und die Null aus dem Register in den Speicher schreibt. Unerwarteterweise ist also der Befehl

```
CLR.L (a0)
```

um vier Taktzyklen langsamer als die beiden folgenden Befehle zusammen:

```
MOVEQ  #0,D0
MOVE.L D0,(a0)
```

Wenn es häufiger vorkommt, daß man mehrere Variablen auf einmal vom Speicher in Register laden will, um mit ihnen zu rechnen, dann kann man sich des MOVEM-Befehls bedienen: Die Variablen bekommen aufeinanderfolgende Speicheradressen, und mit einem MOVEM werden sie auf einmal in mehre-

re Prozessorregister geladen, wobei als Quelloperand die Adresse der ersten Variablen angegeben wird. Dazu ein Beispiel:

```
        MOVEM.L var1,D0-D2
           .
           .
           .
        BSS
var1    DS.L 1
var2    DS.L 1
var3    DS.L 1
```

Nach der Ausführung des MOVEM-Befehls steht var1 in D0, var2 in D1 und var3 in D2. (Vorsicht! Beachten Sie die Reihenfolge, in der die Register bei MOVEM behandelt werden). Genauso schnell können die Variablen nach der Berechnung wieder zurückgeschrieben werden:

```
MOVEM.L D0-D2,var1
```

Zum Vergleich: Der erste MOVEM-Befehl benötigt 44 Taktzyklen, während drei einfache MOVE-Befehle zusammen 60 brauchen; bei Verwendung der PC-relativen Adressierungsart würde sich der Vorteil auf 40 zu 48 verringern. Natürlich ist dies kein optimaler Trick, denn er beeinträchtigt die Lesbarkeit des Programms.

Optimierung von arithmetischen Befehlen

Zunächst einmal ist es empfehlenswert, Berechnungen in Datenregistern auszuführen, nicht in Adreßregistern. Dies bietet sich ja schon deshalb an, weil arithmetische Operationen auf Adreßregister sehr eingeschränkt sind und logische überhaupt nicht vorhanden sind. Bei Addition und Subtraktion bieten Datenregister einen leichten Geschwindigkeitsvorteil. Dieser tritt allerdings nur dann in Erscheinung, wenn der Quelloperand in einem Register steht und in Wortbreite verarbeitet wird. Diese Aufgabe wird bei einem Datenregister um 4 Taktzyklen schneller erledigt. Bedenken Sie aber, daß ja beim Datenregister der Wert auf Langwortbreite vorzeichenerweitert werden muß und so in Wirklichkeit eine Langwortoperation durchgeführt wird.

Klar, daß man immer ADDQ bzw. SUBQ verwenden sollte, wenn ein unmittelbarer Operand verwendet werden soll und dieser genügend klein ist (1 – 8). Was aber, wenn der Wert größer ist? Bei einem Adreßregister könnte das dann so aussehen:

```
ADD.W #300,A0
```

Beachten Sie, daß man bei einer in 16 Bit darstellbaren Konstanten auch Wortbreite verwenden kann, da ja der Operand vor der Addition auf Langwortbreite vergrößert wird. Um 4 Taktzyklen schneller ist allerdings folgender Befehl:

```
LEA 300(A0),A0
```

Zur Addition einer 16-Bit-Konstanten zu einem Adreßregister ist also immer die Adreßberechnung mittels LEA sinnvoll.

Wie schon an anderer Stelle erwähnt, kann man die Multiplikation mit einer Konstanten in einigen Fällen durch Verschiebeoperationen ersetzen. Im einfachen Fall, wo mit einer Zweierpotenz multipliziert werden soll, wäre es viel aufwendiger, dafür MULU/MULS zu bemühen, wo man doch den Operanden nur um eine bestimmte Anzahl von Bits mittels LSL zu verschieben braucht. Das gleiche Prinzip läßt sich natürlich auch auf eine Division durch Zweierpotenzen anwenden.

Betrachten wir die Multiplikationsbefehle einmal genauer. Aus der Tabelle im Anhang G kann man ablesen, daß eine Multiplikation von zwei Datenregistern, unabhängig davon, ob mit MULU oder MULS, höchstens 70 Taktzyklen benötigt werden. Was heißt das nun genau? Die Multiplikation wird vom MC68000 nicht so geradlinig ausgeführt wie die anderen Befehle; vielmehr durchläuft der Prozessor im Mikrocode ein kleines Programm, das die Multiplikation auf Additionen und Subtraktionen zurückführt. Da in diesem Mikroprogramm auch bedingte Verzweigungen enthalten sind, hängt die Ausführungszeit von den zu multiplizierenden Werten ab. Tatsächlich kann man feststellen, daß sich je nach den Werten beträchtliche Unterschiede in der Ausführungszeit ergeben.

Bei MULU hängt die Ausführungszeit nur vom Quelloperanden ab und steigt linear mit der Anzahl der gesetzten Bits in diesem Operanden. Ist die Quelle 0, dann ergeben sich ca. 45 Taktzyklen, beim "worst case" $FFFF (lauter binäre Einsen) sind es hingegen 76 Taktzyklen (alle Werte wurden durch Versuche ermittelt). Letzterer Wert zeigt, daß die vom Hersteller angegebenen maximal 70 Taktzyklen eigentlich ein wenig untertrieben sind. Im Durchschnitt, bei der Verwendung von Zufallszahlen ergeben sich etwa 59 Taktzyklen für MULU. Bis auf eine kleine Ungenauigkeit kann man also sagen, daß für jedes im Quelloperanden gesetzte Bit MULU 2 Taktzyklen länger braucht. Somit kann man MULU etwas beschleunigen, indem man den Wert, der voraussichtlich weniger binäre Einsen enthält, als Quelloperanden benutzt.

Bei MULS ist der Zusammenhang zwischen Einsen im Quelloperanden und der Ausführungszeit nicht so einfach. Hier läßt sich nur sagen, daß die durchschnittliche Ausführungszeit von MULS auch bei 59 Taktzyklen liegt.

In manchen Fällen ist es möglich, eine Multiplikation durch eine Addition zu ersetzen. Nehmen wir als Beispiel eine Routine, die in hoher Auflösung eine vertikale Linie über den Bildschirm ziehen will. Ihre Aufgabe ist es also, im Bildschirmspeicher alle 80 Bytes ein Bit zu setzen (siehe in Kapitel 6 unter "Setzen eines Punktes in hoher Auflösung"). Wenn beim Eintritt in die Routine die Bildschirmadresse in A0 steht, könnte man dies so programmieren:

```
linie
      CLR     D0                * Zeile:=0
loop  MOVE    D0,D1             * Zeile*80 berechnen
      MULU    #80,D1            * in D1
      BSET    #0,40(A0,D1.L)    * Punkt setzen
      ADDQ    #1,D0             * Zeile erhöhen
      CMP     #400,D0           * Ende erreicht?
      BNE     loop              * nein, weiter
      RTS                       * Fertig
```

Hier wird also für jeden Punkt der Ausdruck

Bildschirmadresse + 40 + Zeile * 80

berechnet. Bei so regelmäßig aufgebauten Schleifen läßt sich jedoch leicht die Multiplikation durch die Addition ersetzen, wie das folgende Beispiel zeigt:

```
linie2
      MOVE.L  A0,A1             * Adresse in A1
      ADD     #40,A1            * Mitte einer Zeile
      CLR     D0                * Zeile:=0
loop  BSET    #0,(A1)           * Punkt setzen
      ADD     #80,A1            * eine Zeile weiter
      ADDQ    #1,D0             * und mitzählen
      CMP     #400,D0           * Ende erreicht?
      BNE     loop              * nein, weiter
      RTS
```

In dieser Version wird der Zeilenzähler D0 nur noch zum Mitzählen benutzt; die eigentliche Berechnung der Pixeladressen wird nur noch ausgeführt, indem A1 bei jedem Durchlauf um 80 erhöht wird. Natürlich könnte man obige Schleife durch die Verwendung des DBF-Befehls etwas beschleunigen.

Durch die Verwendung eines Zeigers auf die aktuelle Pixeladresse wird hier einige Rechenzeit gespart. Nach dem gleichen Prinzip wird man auch Zeichenketten oder Felder allgemein durchgehen; nicht, indem man die Adresse jedes neuen Elements einzeln berechnet, sondern mit Hilfe eines Zeigers, der nur durch die Addition einer Konstanten über das gesamte Feld bewegt wird.

Die Verschiebebefehle sollen hier auch als arithmetische Operationen behandelt werden. So kann man sie möglichst effizient einsetzen:

Oft ärgert man sich darüber, daß zwei Befehle nötig sind, um einen Operanden um mehr als 8 Stellen zu verschieben. Wenn die Anzahl der Stellen zwischen 8 und 16 liegt, gibt es zwei Möglichkeiten: Man kann das Verschieben auf zwei Befehle aufteilen, etwa

```
ASL.L #8,D0
ASL.L #8,D0
```

oder man kann die Anzahl der Verschiebungen vorher in ein Register laden:

```
MOVEQ #16,D1
ASL.L D1,D0
```

Dabei ist die letztere Variante zu empfehlen, da sie um 4 Taktzyklen schneller ist.

Beim Programmieren von Verschiebungen sollten Sie beachten, daß die Befehlsausführungszeit linear mit der Anzahl der Verschiebungen steigt. So berechnet sich die Anzahl der Taktzyklen eines Verschiebebefehls nach einer Formel wie etwa

$$8 + 2n$$

wobei n die Anzahl der Verschiebungen ist. Die erste Konstante (hier 8) variiert mit der Verarbeitungsbreite und Adressierungsart.

Achtung! Wenn man ein Wort um mehr als 8 Stellen nach links verschieben will, kann man auch so vorgehen: Statt das Register um n Stellen nach links zu verschieben, tauscht man zuerst die obere und untere Registerhälfte mit dem SWAP-Befehl aus und schiebt dann um 16-n Stellen nach rechts. Voraussetzung für diese Methode ist allerdings, daß der obere Teil des Registers vorher auf Null stand. Da diese Methode selten anwendbar und auch recht zweifelhaft ist, soll hier kein Beispiel gebracht werden.

Optimierung von Verzweigungsbefehlen

Die wohl auffälligste Optimierung der Verzweigungsbefehle der Form "Bcc" ist es, die kurze Form "Bcc.S" zu verwenden, wann immer sie anwendbar ist. Allerdings bietet die 8-Bit-Variante tatsächlich nur dann einen Vorteil, wenn die Verzweigung nicht stattfindet. In diesem Fall werden 4 Taktzyklen gespart.

Wie sonst auch ist die PC-relative Adressierung schneller als die absolute; bei JMP und JSR macht der Unterschied 2 Taktzyklen aus. Wenn man bei JMP die PC-relative Adressierung benutzt, dauert die Ausführung genauso lange wie bei BRA. Für Unterprogrammaufrufe gilt hingegen, daß JSR bei dieser Adressierungsart sogar um zwei Taktzyklen schneller ist als das entsprechende BSR (nur 18 statt 20 Taktzyklen).

Alle hier genannten Optimierungen werden von einigen Assemblern automatisch vorgenommen. Dazu kann ich nur auf das Handbuch Ihres Assemblers verweisen.

Selbstmodifizierender Code

Am Schluß dieses Abschnitts soll noch auf eine besonders unsaubere, aber manchmal recht nützliche Art der Optimierung eingegangen werden: selbstmodifizierender Code. Als selbstmodifizierend bezeichnet man Programme, die ihren eigenen Programmcode verändern. Damit müssen nicht unbedingt Befehlscodes gemeint sein; es können auch unmittelbar angegebene Operanden verändert werden. Klar, daß diese Methode zu besonders unübersichtlichen Programmen führt.

Ein gutes Beispiel für selbstmodifizierenden Code liefert das Programm, welches die Horizontal Blank Interrupts erzeugte. Dort wurde sogar an zwei Stellen eine Adresse in den Operandenteil eines JMP-Befehls geschrieben. In Assembler sieht das so aus:

```
        MOVE.L     adr,jump+2
            .
            .
            .
jump:   JMP        $FFFFFF
```

Die wirkliche Zieladresse des JMP-Befehls wird also erst zur Laufzeit des Programms bekannt. Beim Hineinschreiben der Adresse wird berücksichtigt, daß der JMP-Befehl aus einem Befehlswort besteht, dem 2 Worte Adresse folgen. Natürlich muß zur Assemblierzeit des Programms schon irgendein Wert als Operand des JMP-Befehls eingetragen werden; hier wurde willkürlich $FFFFFF gewählt. Achten Sie aber darauf, dort nicht etwa 0 einzutragen, da manche optimierenden Assembler auf die Idee kommen könnten, die 16-Bit-Variante von JMP zu verwenden – womit wieder einmal ein Systemabsturz fällig wäre.

Natürlich hätte man das auch so lösen können:

```
        MOVE.L      adr,jumpadr
           .
           .
           .
jump:   MOVE.L      jumpadr,A0
        JMP         (A0)
           .
           .
           .
        DATA
        jumpadr     DS.L 1
```

Diese Variante wäre allerdings um einiges langsamer als die erste.

Das gleiche Prinzip läßt sich auch auf Direktoperanden von arithmetischen oder MOVE-Operationen anwenden. Stellen Sie sich vor, in einem besonders zeitkritischen Programmteil wird eine bestimmte Variable nur an einer Stelle benutzt, um sie an eine bestimmte Stelle zu schreiben oder mit einem anderen Wert zu verknüpfen. In diesem Fall könnte ein weniger zeitkritischer Programmteil den Wert vorher berechnen und in den Quelloperandenteil eines Befehls der kritischen Routine schreiben.

Sinnvoll ist das zum Beispiel, wenn man das HBI-Programm dahingehend verändern möchte, daß in einem Interrupt alle Farbregister auf einmal verändert werden. Das würde dann etwa so aussehen:

```
hbi:    MOVE.W      #0,color_0
c1:     MOVE.W      #0,color_1
           .
           .
           .
        MOVE.W      x,hbi+2
        MOVE.W      y,c1+2
```

(color_0, color_1, x und y sind woanders definiert.)

Durch diese Befehlssequenz wird sichergestellt, daß am Anfang der Routine, so schnell es geht, neue Werte in die Farbregister geschrieben werden. Erst wenn das erledigt ist, können neue Werte für den nächsten Durchlauf der Routine berechnet werden.

Hier wurde die Tatsache benutzt, daß Erweiterungswerte für den Quelloperanden immer direkt hinter dem Opcode abgespeichert werden. Der Befehl

```
hbi:    MOVE.W      #0,color_0
```

hat also im Speicher folgenden Aufbau:

- ein Wort Opcode für "Bewege einen unmittelbaren 16-Bit-Operanden an eine absolute Adresse"

- ein Wort für den unmittelbaren Quelloperanden, also Null

- zwei Worte für die Adresse "color_0"

Also liegt man richtig, wenn man den neuen Quelloperanden an die Adresse hbi + 2 schreibt.

Es gibt Situationen, in denen selbstmodifizierender Code nicht nur sinnvoll, sondern sogar notwendig ist. Außerdem kann man diese Methode damit entschuldigen, daß sie auf einigen – heute veralteten – Prozessoren aufgrund deren sehr beschränkten Befehlssatzes sogar notwendig war. Eines sollte man noch bedenken: Logischerweise kann selbstmodifizierender Code nicht in ROMs untergebracht werden.

Optimierung auf der Realisierungsebene

Die Tricks, die Programme auf der Realisierungsebene schneller machen, sind schon problematischer als die des vorhergehenden Abschnitts: Sie können Programmteile ziemlich unlesbar machen. Deshalb sollten Sie die im folgenden beschriebenen Methoden am besten erst dann anwenden, wenn ein Programm schon weitgehend fertiggestellt ist und nur noch das Laufzeitverhalten verbessert werden soll. So ersparen Sie sich zusätzliche Schwierigkeiten beim Ändern des unübersichtlichen optimierten Codes.

Alle Programme verbrauchen einen gewissen Teil ihrer Rechenzeit damit, sich mit Organisationsaufgaben zu beschäftigen. Diese Aktivitäten tragen zur tatsächlichen Lösung des Problems nichts bei und sind deshalb eigentlich unproduktiv. Konkreter geht es dabei um Programmsprünge, Unterprogrammaufrufe, die Übergabe von Parametern und die Organisation von Programmschleifen. In diesem Abschnitt werden deshalb einige Möglichkeiten gezeigt, wie man diesen Organisationsaufwand verringern kann.

Wenden wir uns zunächst einmal dem Befehl JSR und seinem Verwandten BSR zu. Tatsache ist, daß Unterprogrammaufrufe in manchen Fällen die Programmabarbeitung merklich verlangsamen können. In zeitkritischen oder besonders häufig durchlaufenen Codesequenzen ist es deshalb empfehlenswert, Unterprogrammaufrufe durch den tatsächlichen Code zu ersetzen. Immerhin

beträgt der zeitliche Aufwand für einen Unterprogrammaufruf im Bestfall 26 Taktzyklen – die Summe aus der Ausführungszeit eines BSR und RTS. Hinzu kommen vielleicht noch Befehle für die Übergabe von Parametern.

Eine gute Möglichkeit, Unterprogrammaufrufe zu vermeiden, bieten Makros. Wenn Ihr Assembler Makros beherrscht, sollten Sie von dieser Möglichkeit Gebrauch machen. Makros haben gegenüber Unterprogrammen auch oft den Vorteil, daß Parameter direkt in den Code des Makros eingebaut werden und deshalb keine Befehle für eine Parameterübergabe erforderlich sind.

In manchen Fällen lassen sich jedoch Unterprogramme nicht vermeiden. Falls die Unterprogramme Parameter erhalten, sollten Sie diese am besten in Prozessorregistern übergeben. Im Vergleich zur Übergabe über den Stack spart das nicht nur Zeit, sondern auch den einen oder anderen Befehl für einen Stackzugriff. Eine Ausnahme ist es natürlich, wenn Unterprogramme sich mehrfach verschachtelt oder sogar rekursiv aufrufen sollen. In diesem Fall gibt es keine Alternative zum Stack.

Nun zu den verschiedenen Schleifenarten. Betrachten wir zunächst, wie man eine Zählschleife (entsprechend der FOR-NEXT-Schleife in BASIC) in Assembler implementieren würde. Ein Wert in D0 soll von einem Anfangswert bis zu einem Endwert hochgezählt werden, wobei für jeden der Werte die Schleife durchlaufen wird:

```
        MOVE        anfangswert,D0
loop:     .
        [ hier steht der Rumpf der Schleife]
          .
          .
        ADDQ        #1,D0
        CMP         endwert,D0
        BLE         loop
```

Wenn der Zähler D0 ebensogut von einem Anfangswert bis null hinuntergezählt werden kann, ist natürlich die Verwendung des DBF-Befehls angebracht, der sinngemäß die letzten drei Befehle des obigen Beispiels ersetzt:

```
        MOVE        anfangswert,D0
loop:     .
        [ hier steht der Rumpf der Schleife]
          .
          .
        DBF         D0,loop
```

Beachten Sie aber, daß die Anzahl der Durchläufe der Schleife um eins höher ist als der Wert in "anfangswert", da D0 in der Schleife auch den Wert Null annimmt.

Gemeinsam ist den beiden gezeigten Schleifen der Nachteil, daß sie nicht abweisend sind. Sie müssen also mindestens einmal durchlaufen werden. Wenn beim ersten Beispiel der Anfangswert gleich dem Endwert ist, geschehen sogar recht unerwünschte Dinge. Für abweisende Schleifen, die auch diesen Fall richtig behandeln, wird deshalb oft folgende Form gewählt, die die Abfrage, ob die Schleife beendet werden soll, an den Anfang stellt:

```
        MOVE        anfangswert,D0
loop:   CMP         endwert,D0
        BGT         loopend
          .
        [hier wieder der Rumpf der Schleife]
          .
          .
        BRA         loop
loopend:  .
          .
          .
```

Diese Methode hat jedoch den kleinen Mangel, daß sie am Ende der Schleife einen zusätzlichen Verzweigungsbefehl benötigt. Eleganter ist es deshalb, die Abfrage wieder ans Ende der Schleife zu stellen, aber genau an diesem Punkt in die Schleife hineinzuspringen:

```
        MOVE        anfangswert,D0
        BRA         entry
loop:
          .
          .
          .
entry:  ADDQ        #1,D0
        CMP         endwert,D0
        BLE         loop
```

Sehen Sie, wie dadurch ein Verzweigungsbefehl eingespart wird? Das heißt, genaugenommen sind es immer noch genauso viele Verzweigungsbefehle, nur wird "BRA entry" nur einmal ausgeführt und nicht in jedem Schleifendurchlauf.

Natürlich kann man auch hier den DBF-Befehl einsetzen:

```
        MOVE        anfangswert,D0
        ADDQ        #1,anfangswert
        BRA         entry
loop:     .
          .
          .
entry:  DBF         D0,loop
```

Vorsicht! Damit sich diese Schleife genauso verhält wie die oben gezeigte DBF-Schleife, muß man den Anfangswert um eins erhöhen, da sonst die Schleife nur mit den Werten von anfangswert–1 bis 0 durchlaufen würde.

Es kommt recht oft vor, daß man Speicherbereiche kopieren oder auf einen Anfangswert setzen möchte. Um etwa den Bildschirm zu löschen, wird meistens eine Schleife wie die folgende verwendet:

```
        MOVE.L      scradr,A0
        MOVE        #7999,D0
loop    CLR.L       (A0)+
        DBF         D0,loop
```

In der Variablen scradr soll die Anfangsadresse des Bildschirmspeichers stehen. Der Anfangswert 7999 für den Schleifenzähler errechnet sich daraus, daß der Bildschirmspeicher 32000 Bytes lang ist, also aus 32000 / 4 = 8000 Langworten besteht. Davon wird wegen der schon beschriebenen Eigenschaften von DBF noch eins abgezogen. Die Langwortbreite wird benutzt, da ein byteweises Löschen fast dreimal so lange dauern würde.

In dieser Form benötigt ein Schleifendurchlauf 30 Taktzyklen. Wie jedoch im vorigen Abschnitt beschrieben, ist es praktisch, CLR zu ersetzen:

```
        MOVE.L      scradr,A0
        MOVE        #7999,D0
        MOVEQ.L     #0,D1
loop    MOVE.L      D1,(A0)+
        DBF D0,loop
```

In dieser Form braucht ein Schleifendurchlauf nur noch 22 Taktzyklen – immerhin eine Zeitersparnis von mehr als einem Viertel. Doch eins ist an dieser Schleife immer noch nicht perfekt: Von den 22 Taktzyklen entfallen 10 allein auf den DBRA-Befehl, der ja zur eigentlichen Aufgabe nichts beiträgt, sondern nur der Programmorganisation dient. Ein teilweise vermeidbarer Aufwand, wie das folgende Programmbeispiel zeigt:

```
        MOVE.L      scradr,A0
        MOVE        #3999,D0
        CLR.L       D1
loop    MOVE.L      D1,(A0)+
        MOVE.L      D1,(A0)+
        DBF D0,loop
```

Die gezeigte Methode ist eine direkte Anwendung des Prinzips "Platz opfern, um Zeit zu sparen". Hier werden bei einem Schleifendurchlauf immer zwei Langworte auf einmal gelöscht. Die Anzahl der Schleifendurchläufe errechnet sich nach 32000 / (4 * 2) – 1 = 3999. So entfallen auf ein Langwort nur noch

17 Taktzyklen. Natürlich kann man diese Vergrößerung der Schleife beliebig fortsetzen, auf 4, 8, 16, ja bis zu 8000 Langworte. Zum Vergleich ist es vielleicht interessant, daß bei 32 Langworten, die in einer Schleife behandelt werden, nur noch durchschnittlich 12,31 Taktzyklen auf ein Langwort entfallen, wobei der Organisationsaufwand in Form des DBRA-Befehls nur noch 2,6% der Rechenzeit ausmacht – ein akzeptabler Kompromiß.

Beachten Sie, daß die zuletzt vorgeschlagene Implementierung des Algorithmus fast 2,5 mal schneller als die erste Version ist – bei einem Programm, daß etwa für bewegte Grafik mehrmals in der Sekunde den Bildschirm löschen muß, kann das eine ganze Menge ausmachen.

Natürlich läßt sich dieses Prinzip auch auf andere Schleifen anwenden. Zum Beispiel lohnt es sich, die Kopierschleife des RAM-Disk-Programms aus Kapitel 6 auf diese Weise zu optimieren. Logischerweise ist der Nutzen dieser Methode bei Schleifen mit einem größeren Rumpf sehr begrenzt, denn welchen Sinn hat es, den Verwaltungsaufwand zu verringern, wenn dieser von Anfang an ohnehin nur einen geringen Anteil der Rechenzeit verbraucht?

Eine Voraussetzung für die bisher gezeigte Art der Optimierung ist es, daß die Anzahl der Schleifendurchläufe von vornherein bekannt ist. Wir wollen jetzt das Prinzip auf eine beliebige Anzahl von Schleifendurchläufen erweitern. Einfach ist die Behandlung dann, wenn man von vornherein weiß, daß die Anzahl der Schleifendurchläufe in jedem Fall das Vielfache einer bestimmten Zahl sein muß. Doch was kann man tun, wenn diese Anzahl völlig beliebig ist, wenn etwa Speicherbereiche kopiert werden müssen, deren Länge jede beliebige Anzahl von Bytes annehmen kann? Solange die Anzahl der Schleifendurchläufe ziemlich gering ist, ist sicherlich die konventionelle Methode angebracht, immer nur die kleinste Einheit auf einmal zu behandeln, also etwa ein Byte. Wenn allerdings mehrere hundert oder sogar Tausende von Einheiten (etwa Bytes) behandelt werden müssen, lohnt es sich, darüber nachzudenken, ob das nicht etwas schneller ginge.

Es gibt tatsächlich eine Möglichkeit, die Sache zu beschleunigen: Man teilt dazu die Anzahl der Schleifendurchläufe in zwei Teile auf. Nennen wir der Einfachheit halber die Anzahl der Einheiten (Bytes oder was immer), die in einem Schleifendurchlauf behandelt werden, "n". Dann ist der eine Teil das größte Vielfache von n, das gerade noch kleiner oder gleich der gewünschten Anzahl Schleifendurchläufe ist; der zweite Teil bildet den Rest. Anders ausgedrückt, wenn man mit a die gewünschte Anzahl der Einheiten bezeichnet, wird eine große Schleife, in der je n Einheiten behandelt werden, a / n (ganzzahlige Division) mal durchlaufen, während eine kleine Schleife, in der jeweils nur eine Einheit behandelt wird, a modulo n mal durchlaufen wird.

Ein Beispiel verdeutlicht das Prinzip: Die folgende Routine kopiert einen Speicherbereich, dessen Länge jede beliebige Bytezahl annehmen kann. Die Anfangsadresse, von der kopiert werden soll, wird in A0 übergeben, jene, auf die kopiert werden soll in A1. Die Anzahl der zu kopierenden Bytes wird in D0 übergeben (Langwort). In der großen Schleife werden immer 8 Langworte, also 32 Bytes, in einem Durchlauf kopiert. In der Praxis wären vielleicht etwas mehr sinnvoll, aber schließlich würde es Ihnen ja auch nicht viel nützen, wenn hier Seite um Seite mit dem Befehl "MOVE.L (A0)+,(A1)+" gefüllt würde. Damit auch der Fall Länge = 0 richtig behandelt wird, wurden alle Schleifen abweisend gemacht.

```
fastcop
        MOVE.L      D0,D1           * D0 nicht ändern
        LSR.L       #5,D1           * durch 32
        BRA.S       entry1          * abweisende Schleife
loop1
        MOVE.L      (A0)+,(A1)+     * 32 Bytes kopieren
        MOVE.L      (A0)+,(A1)+     *
        MOVE.L      (A0)+,(A1)+     *
        MOVE.L      (A0)+,(A1)+     *
        MOVE.L      (A0)+,(A1)+     *
        MOVE.L      (A0)+,(A1)+     *
        MOVE.L      (A0)+,(A1)+     *
        MOVE.L      (A0)+,(A1)+     *
entry1
        DBF         D1,loop1        * a / 32 mal wiederholen
        MOVE        D0,D1           * D0 nicht ändern
        AND         #31,D1          * a modulo 32
        BRA.S       entry2          * abweisende Schleife
loop2
        MOVE.B      (A0)+,(A1)+     * einzelne Bytes kopieren
entry2
        DBF         D1,loop2        * a modulo 32 mal
        RTS                         * Ende
```

Bei dieser Schleife sind übrigens bis zu 16 + 5 = 21 Bits für die Anzahl der zu kopierenden Bytes erlaubt. Wenn volle Langwortbreite verarbeitet werden soll, müßte der erste DBF-Befehl durch folgende Sequenz ersetzt werden (die natürlich langsamer ist):

```
        SUBQ.L      #1,D1
        BCC.S       loop1
```

Übrigens erwartet obige Routine, daß beide Adressen an einer Wortgrenze liegen. Man könnte sie noch dahingehend abändern, daß der Fall, daß Quell- und Zieladresse ungerade sind, gesondert behandelt wird. Für den Fall jedoch, daß eine der Adressen gerade und die andere ungerade ist, gibt es auf dem MC68000 keine elegante Möglichkeit, den Vorgang zu beschleunigen: Da bleibt nur die altbewährte Möglichkeit, jedes Byte einzeln zu kopieren.

Optimierung auf der Algorithmenebene

In diesem Abschnitt zeigen wir einige allgemein anwendbare Methoden, wie man Programmabläufe beschleunigen kann. Auf der Algorithmenebene sind wir allerdings schon so weit von der Hardware entfernt, daß die Methoden dieses Abschnitts auch in höheren Programmiersprachen und auf gänzlich anderen Prozessoren als dem MC68000 anwendbar sind. Natürlich kann hier nur ein kleiner Ausschnitt der möglichen Optimierungen auf dieser Ebene vorgestellt werden, denn die meisten Möglichkeiten lassen sich nicht schematisieren.

Das erste Prinzip kann man so formulieren: Entscheidungen zu einem möglichst frühen Zeitpunkt treffen. Gemeint ist damit folgendes: Wenn in einer zeitintensiven Schleife in jedem Durchlauf eine Abfrage stattfindet, deren Ergebnis schon vor dem Eintritt in die Schleife feststeht und sich nicht mehr ändert, dann ist es effizienter, die Abfrage nur einmal durchzuführen, woraufhin dann für jeden Fall in eine spezielle Schleife verzweigt wird. Ein Diagramm drückt dies besser aus als viele Worte; betrachten Sie deshalb Abb. 7.1.

Denken Sie an den bereits vorgestellten Line-Algorithmus. Dort treten zwei Schleifen auf, die die Fälle DX >= DY und DX < DY gesondert behandeln. In jeder der Schleifen findet bei jedem Durchlauf eine Abfrage statt, ob DX negativ ist oder nicht, obwohl dies ja von vornherein feststeht. Statt dessen könnte zunächst eine Abfrage erfolgen, ob DX negativ ist und für jede der beiden Möglichkeiten – DX negativ oder DX positiv– eine eigene Schleife eingerichtet werden. Auf diese Weise läßt sich immerhin ein Geschwindigkeitsgewinn von rund 18 % erreichen; die Quellcode-Dateien LINESHOP.S (für hohe Auflösung) und LINESLOP.S (für niedrige Auflösung) auf der beiliegenden Diskette zeigen, wie man diese Idee anwenden kann.

Das Prinzip läßt sich natürlich auch dann anwenden, wenn mehr als nur zwei Möglichkeiten unterschieden werden. So hätte man beim Line-Algorithmus etwa noch die Möglichkeit einsetzen können, Linien wahlweise zu zeichnen oder zu löschen, wobei dann schon vier verschiedene Schleifen für jeden der beiden oben beschriebenen Fälle nötig gewesen wären. Das läßt sich beliebig fortsetzen. Natürlich hängt die gewonnene Rechenzeit von der Anzahl der Befehle in der Schleife ab; bei mehreren Dutzend Befehlen ist die sehr geringe Beschleunigung kaum den Aufwand wert.

Um es noch einmal ganz deutlich zu sagen: Das Prinzip funktioniert nur, wenn das Ergebnis der Abfrage(n) schon vor dem Eintritt in die Schleife bekannt ist. Sobald auf Ergebnisse Bezug genommen wird, die erst während eines Schleifendurchlaufs anfallen, ist es nicht mehr anwendbar.

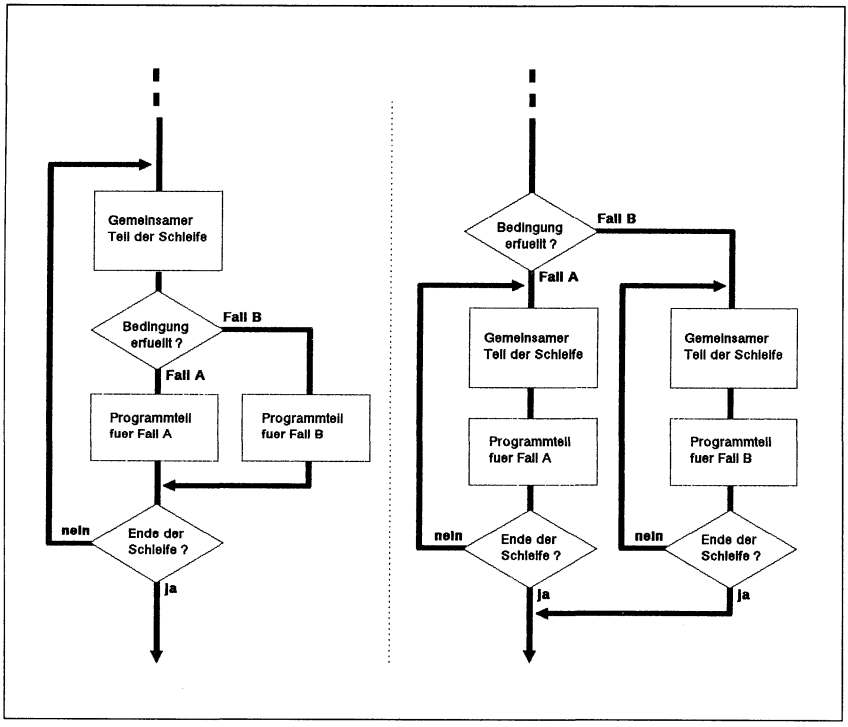


Abb. 7.1: Implementierung einer Schleife mit Fallunterscheidung – links die normale / rechts die optimierte Version

Das zweite wichtige Prinzip lautet folgendermaßen: Ergebnisse, die mehrmals hintereinander benötigt werden, brauchen nur einmal berechnet zu werden.

Dazu ein Beispiel: Bei Stringoperationen, bei denen die Länge eines bestimmten Strings mehrmals benötigt wird, reicht es natürlich, wenn diese Länge nur einmal festgestellt und an einem sicheren Platz zwischengespeichert wird. So gesehen bringt dieses Prinzip nichts neues, denn in Assembler würden es ohnehin die meisten Programmierer so machen. Interessanter ist aber folgendes Prinzip, das eine Erweiterung des vorangegangenen darstellt:

Wenn in einem Programm häufig eine bestimmte Funktion von Werten benötigt wird, kann eine Wertetabelle die ständige Berechnung von Funktionswerten ersparen.

Mit Funktion ist hier irgendeine Operation gemeint, die man auf einen begrenzten Bereich von Werten anwenden will. Erinnern wir uns an den Plot-Algorithmus zum Setzen eines Punktes in hoher Auflösung. Seine Aufgabe bestand hauptsächlich darin, folgende Formel zu berechnen:

adr = Bildschirmadresse + Y*80 + X/8

Man könnte diese Berechnung nun im mathematischen Sinne als eine Funktion zweier beliebiger Werte x und y auffassen:

f (x, y) = Bildschirmadresse + y * 80 + x / 8

Das Prinzip besagt nun, daß man ein Feld von Funktionswerten für x von 0 bis 639 und y von 0 bis 399 berechnen könnte, aus dem der Funktionswert direkt hervorgeht. Leider scheitert das in diesem Fall an technischen Gegebenheiten, denn mit einem 640 x 400-Feld von Langworten wäre fast ein Megabyte belegt. Realistischer ist folgende Betrachtungsweise:

g(y) = Bildschirmadresse + y * 80

Die Adresse errechnet sich dann aus

adr = g(y) + x / 8

In diesem Fall brauchen nur 640 Langworte für eine Tabelle aller Werte von g(y) abgespeichert zu werden. Wie baut man diese Tabelle nun auf? Eine Möglichkeit wäre es sicherlich, alle Funktionswerte mit einem Taschenrechner selbst auszurechnen und in den Quellcode des Assemblerprogramms zu schreiben. Doch wozu ist ein Computer schließlich programmierbar? Einfacher ist es, die Tabelle der Funktionswerte bei der Initialisierung des Programms zu füllen. In unserem Beispiel könnte das so aussehen:

```
        CLR         D0          * Wert auf 0
        LEA         tabelle,A0  * Adresse der Wertetabelle
tabinit
        MOVE        D0,D1       * D0 nicht ändern
        MULU        #80,D1      * erzeugt Langwort
        ADD.L       scradr,D1   * plus Bildschirmadresse
        MOVE.L      D1,(A0)+    * in die Tabelle
        ADDQ        #1,D0       * nächster Wert
        CMP         #400,D0     * fertig?
        BNE.S       tabinit     * nein, nächster Durchlauf
          .
          .
          .
        BSS
tabelle
        DS.L 400
```

Diese Routine setzt voraus, daß in der Variablen "scradr" die Bildschirmadresse steht. Zugreifen würde man auf einen Wert der Tabelle folgendermaßen, wenn Y in D1 steht:

```
LEA         tabelle,A0      * Wertetabelle
LSL         #2,D1           * Langworte -> Bytes
MOVE.L      0(A0,D1.W),A1   * Funktionswert lesen
```

Diese Befehlssequenz ersetzt folgende:

```
MULU        #80,D1
ADD.L       scradr,D1
MOVE.L      D1,A1
```

In diesem Fall ist die Listenadressierung jedoch wesentlich schneller, da besonders die Bearbeitung des MULU-Befehls einige Zeit braucht. Im Prinzip könnte man ja auch den Ausdruck x / 8 mit einer Tabelle berechnen, doch in diesem Fall lohnt sich der Aufwand ganz gewiß nicht, da ja schon der Zugriff auf einen Tabelleneintrag zwei bis drei Befehle erfordert, während obiger Ausdruck mit einem simplen LSR #3,D0 zu berechnen ist. Eine Tabelle lohnt sich also nur bei hinreichend komplizierten Berechnungen. Ziehen Sie im Zweifelsfalle die Taktzyklentabelle in Anhang G zu Rate.

In diesem Buch ist das Prinzip sogar schon einmal angewandt worden: Bei dem HBI-Programm aus Kapitel 6. Dort wurde am Anfang des Programms eine Tabelle sämtlicher Shifter-Farben aufgebaut. Theoretisch könnte man ja auch aus jeder Zahl von 0 bis 511 die entsprechende Shifter-Darstellung (mit dazwischengeschobenen Nullbits an Stellen 3 und 7) errechnen, sobald sie gebraucht wird. Nur ist das leider relativ aufwendig, wie Sie in der Initialisierungsroutine sehen können, und daher in der begrenzten Zeit in einem HBI nicht durchzuführen. Deshalb mußte eine Tabelle her.

Als besonders nützlich erweist sich das Prinzip, wenn die Berechnung von Werten sehr kompliziert ist, wie etwa bei der Sinusfunktion, die ja von vielen Grafikprogrammen benötigt wird. Dort kann man eine Tabelle von Sinuswerten mit nur vom Speicherplatz begrenzter Genauigkeit errechnen, was komplizierte Grafikberechnungen sicherlich auf ein Vielfaches der ursprünglichen Geschwindigkeit beschleunigt.

Soviel zur Verwendung von Tabellen. Man kann das Prinzip aber noch mehr erweitern, auf die "mundgerechte" Vorberechnung von beliebigen Objekten, nicht nur von Funktionswerten. Kommen wir noch einmal auf das anfangs in diesem Kapitel erwähnte Spiel mit bewegten Objekten zurück: Um dort Objekte an jedem beliebigen Punkt im Bildschirmspeicher kopieren zu können, sind recht aufwendige Berechnungen erforderlich, denn die Bitmap-Daten der Ob-

jekte müssen erst an die Pixeldarstellung des Bildschirmspeichers angepaßt werden, das heißt, eventuell um eine bestimmte Anzahl von Bits verschoben werden. Man kann dies nun wesentlich beschleunigen, wenn man die Daten für jedes Objekt nicht nur einmal abspeichert, sondern sechzehnfach, wobei jedes Abbild gegenüber seinem Vorgänger um ein Bit verschoben ist. Die Routine zum Darstellen der Objekte braucht nur noch zu berechnen, welches Objekt gerade benutzt werden muß, und kann dann die Daten direkt in den Bildschirmspeicher hineinkopieren. Einige professionell programmierte Spiele machen von dieser Möglichkeit Gebrauch.

Eine weitere interessante Möglichkeit besteht darin, bestimmten Sonderfällen auch eine besondere Behandlung zukommen zu lassen, wenn sie häufig genug auftreten. Wenn etwa in einem Programm häufig vertikale oder horizontale Linien auftreten, ist es sinnvoll, im Line-Algorithmus zu überprüfen, ob es sich um eine solche Linie handelt. Ist das der Fall, dann wird zu einem spezialisierten Algorithmus für eine vertikale oder horizontale Linie verzweigt, der sicher um ein Mehrfaches schneller sein kann als eine allgemeine Funktion zum Linienziehen. Dies kann man auf folgendes Prinzip zurückführen: Ein Algorithmus ist gewöhnlich um so langsamer, je flexibler er ist; ein spezialisierter Algorithmus kann wesentlich schneller sein.

Es gibt zweifellos noch eine Vielzahl von Möglichkeiten, wie man Abwandlungen dieses Prinzips nutzen kann. Zum Schluß wünschen wir Ihnen viel Spaß beim Experimentieren mit den Programmen dieses Buches.

Anhang A

Darstellung von Zahlen im Rechner

Ein Zahlensystem hat die Aufgabe, Zahlen auf eine möglichst überschaubare Weise darzustellen. Jedermann ist es gewohnt, mit dem dezimalen Zahlensystem umzugehen. Dabei ist unser Zehnersystem nur eines unter vielen; seine Entstehung erklärt sich bekanntlich hauptsächlich daraus, daß ein Mensch an jeder Hand fünf Finger hat. Aber es hätte nicht so kommen müssen, denn in vielen Kulturen waren Zahlensysteme zu allen möglichen Basen verbreitet, wobei sich besonders das Zwanziger- und Fünfersystem hervortaten. Selbst heute begegnet man gelegentlich noch den Auswirkungen anderer Zahlensysteme. So etwa bei der Einteilung der Stunden in Minuten und Sekunden, die auf dem Sechzigersystem der alten Ägypter beruht, oder dem französichen Wort für achtzig, quatre-vingt, was wörtlich "vier mal zwanzig" bedeutet. Betrachten wir zunächst einmal, wie die Zahlendarstellung in unserem Dezimalsystem eigentlich funktioniert:

$$
\begin{aligned}
327 &= 300 + 20 + 7 \\
&= 3 * 100 + 2 * 10 + 7 * 1 \\
&= 3 * 10^2 + 2 * 10^1 + 7 * 10^0
\end{aligned}
$$

Der Wert, mit dem eine Ziffer in obigem Beispiel multipliziert wird, ist der Stellenwert der Ziffer. Der Stellenwert der am weitesten rechts stehenden Ziffer ist 1; um so weiter eine Ziffer links steht, desto größer ist ihr Stellenwert. In diesem Beispiel ist 100 der Stellenwert der Ziffer 3, 10 der der Ziffer 2, und der Stellenwert der 7 ist 1. Im Dezimalsystem bilden die Potenzen der Zahl 10 die Stellenwerte. Deshalb wird 10 als Basis des dezimalen Zahlensystems bezeichnet.

Allerdings ist diese Methode, bei der jeder Ziffer eine Potenz der Basis als Stellenwert zugewiesen wird, nicht die einzig mögliche. Die römischen Zahlen beispielsweise zeigen, daß es auch anders geht. Wie wir noch sehen werden, hat unsere Darstellung beträchtliche Vorteile, wenn es darum geht, mit Zahlen zu rechnen. Wie gesagt, man kann genauso gut statt der 10 eine beliebige natürliche Zahl als Basis des Zahlensystems wählen. Da der Computer nur zwei Ziffern (oder genauer gesagt, 2 Zustände eines Bits) darstellen kann, wählt man praktischerweise das binäre Zahlensystem. Dort stehen nicht mehr Ziffern von 0 bis 9, sondern nur noch die Ziffern 0 und 1 zur Verfügung. Be-

trachten wir einige natürliche Zahlen im Binärsystem und im Dezimalsystem:

binär	dezimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8

Im Prinzip funktioniert das Zählen im Binärsystem genauso wie im Dezimalsystem: Will man eine Zahl um eins erhöhen, so wird zuerst die letzte Ziffer erhöht. Wird dabei der Bereich der erlaubten Ziffern (Wert 0 bis Basis–1) überschritten, dann wird diese Ziffer auf null gesetzt und dafür die nächste Ziffer genau auf die gleiche Art um eins erhöht.

Sehen wir uns nun an, wie man eine binäre Zahl in eine dezimale umrechnen kann:

$$\begin{aligned} 1010 &= 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 \\ &= 1*8 + 0*4 + 1*2 + 0*2 \\ &= 10 \end{aligned}$$

Halt, da kann doch etwas nicht stimmen! Letztendlich steht dort

$$1010 = 10$$

Natürlich ist hier gemeint, daß die binäre Zahl 1010 der dezimalen Zahl 10 entspricht. Wie Sie sehen, können Zahlen verschiedener Systeme leicht miteinander verwechselt werden. In der mathematischen Schreibweise kann man deshalb das Zahlensystem einer Ziffernfolge dadurch festlegen, daß man die Basis rechts heranschreibt. Praktischerweise schreibt man die Basis immer als Dezimalzahl. Korrekter wäre also

$$1010_2 = 10_{10}$$

Aus diesem Beispiel kann man mit etwas Überlegung folgendermaßen verallgemeinern: Um so kleiner die Basis eines Zahlensystems ist, desto mehr Ziffern braucht man für die Darstellung einer bestimmten Zahl. Eine Folge dieser Tatsache ist, daß die binäre Schreibweise für größere Zahlen praktisch un-

lesbar ist. So entspricht etwa der dezimalen Zahl 1000 folgende Binärdarstellung:

$$1000_{10} = 11111101000_2$$

Zweifellos ist die Binärdarstellung recht unpraktisch, wenn ein Mensch mit den Zahlen umgehen soll. Andererseits hat das Dezimalsystem bei Computern den Nachteil, daß die Umrechnung in Binärzahlen und zurück recht aufwendig ist. Deshalb wurde das Hexadezimalsystem eingeführt, das Zahlensystem zur Basis 16 (gelegentlich auch als Sedezimalsystem bezeichnet). Hier mußte man sich etwas einfallen lassen, um 16 verschiedene Ziffern für die Werte 0 bis 15 zusammenzubekommen, da ja bekanntlich nur 10 Ziffern vorgesehen sind. Man behilft sich einfach damit, daß die Ziffern 0 bis 9 ihre Werte behalten, während für die Werte 10 bis 15 die ersten 6 Buchstaben des Alphabets, also A bis F verwendet werden. Der Vorteil dieses Zahlensystems besteht darin, daß einer Hexadezimalziffer genau 4 Bits entsprechen, wodurch die Umwandlung binär nach hexadezimal oder umgekehrt zu einer Textersetzung vereinfacht wird. Darüber hinaus paßt die hexadezimale Darstellung immer zur Wortbreite eines Rechners, seien es 8, 16 oder 32 Bits.

Die folgende Tabelle zeigt die binären und dezimalen Entsprechungen der hexadezimalen Ziffern:

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

Noch einmal zurück zur Schreibweise der Zahlen anderer Systeme: Da die mathematische Schreibweise auf Computern nun einmal schlecht zu verwirkli-

chen ist, werden statt dessen bestimmte Sonderzeichen benutzt, die vor einer Ziffernfolge das Zahlensystem kennzeichnen. So werden binäre Zahlen meistens durch ein vorangestelltes Prozent-Zeichen (%) gekennzeichnet, hexadezimale durch das Dollar-Zeichen ($). Dezimale Zahlen werden nicht besonders hervorgehoben. Natürlich gibt es beliebig viele andere Möglichkeiten der Bezeichnung, die genannte ist jedoch am häufigsten anzutreffen. Will man nun etwa die Hexadezimalzahl $BAFF in eine binäre Zahl umwandeln, so braucht man nur die Binärcodes der einzelnen Ziffern nach obiger Liste hintereinanderzuschreiben:

$BAFF = %1011 1010 1111 1111

Die Umwandlung binär nach hexadezimal ist genauso einfach.

Es stellt auch kein großes Problem dar, eine hexadezimale oder binäre Zahlendarstellung in die dezimale umzurechnen. Das allgemeine Prinzip für eine beliebige Basis B ist folgendes:

– Jede Ziffer bekommt als Stellenwert B hoch s zugewiesen, wobei s die Stelle der Ziffer von rechts aus gezählt ist. Die rechte Ziffer erhält die Stelle 0, die davorstehende 1 und so weiter.

Nun wird jede Ziffer mit ihrem Stellenwert multipliziert, und alle sich so ergebenden Werte werden addiert. Das Ergebnis ist die dezimale Darstellung der Zahl.

Bei der Umwandlung von $BAFF sieht das etwa so aus:

$$\begin{aligned} \$BAFF &= \$B * 16^3 + \$A * 16^2 + \$F * 16^1 + \$F * 16^0 \\ &= 11*4096 + 10*256 + 15*16 + 15*1 \\ &= 47871 \end{aligned}$$

Die Umwandlung einer dezimalen Zahl in ein anderes Zahlensystem ist schon schwieriger. Eigentlich wiederspricht das der Tatsache, daß alle Zahlensysteme gleichwertig sind, denn demzufolge sollte es keinen Unterschied machen, ob man von einem Zahlensystem A in ein System B konvertiert oder umgekehrt. Tatsächlich besteht der Unterschied nur in den menschlichen Gewohnheiten. Der eben angegebene Algorithmus erscheint nur deshalb recht einfach, da Berechnungen nur im Dezimalsystem ausgeführt werden müssen, was nun einmal jedem geläufig ist. Er ließe sich sinngemäß genauso bei der Umwandlung vom dezimalen in ein fremdes System verwenden, nur würde das verlan-

gen, daß Berechnungen im fremden Zahlensystem durchgeführt werden – was schon deshalb schwierig ist, da man das Einmaleins nur in der dezimalen Schreibweise beherrscht. Deshalb wird hierfür eine etwas andere Methode verwendet:

- Zunächst wird die zu konvertierende Dezimalzahl durch die Basis des neuen Zahlensystems ganzzahlig geteilt.

- Der entstehende Rest ergibt die letzte Ziffer der Zahl im neuen Zahlensystem.

- Auf das Ergebnis der Division wird nun das gleiche Verfahren angewandt. Als nächstes ergibt sich die Ziffer, die links von der eben erzeugten liegt. Das Verfahren wird solange wiederholt, bis das Ergebnis der Division null ist.

Gehen wir das einmal an einem Beispiel durch: Wir wollen die Zahl 44 in eine Binärzahl umwandeln:

```
44 : 2 = 22   Rest 0
22 : 2 = 11   Rest 0
11 : 2 =  5   Rest 1
 5 : 2 =  2   Rest 1
 2 : 2 =  1   Rest 0
 1 : 1 =  0   Rest 1
```

Das Ergebnis ist also %101100

Um zu zeigen, daß das Verfahren auch mit anderen Systemen als dem Binärsystem funktioniert, überprüfen wir einmal das Ergebnis der oben durchgeführten Umwandlung von $BAFF in 47871:

```
47871 :16 = 2991   Rest 15 = $F
 2991 :16 =  186   Rest 15 = $F
  186 :16 =   11   Rest 10 = $A
   11 :16 =    0   Rest 11 = $B
```

Das Ergebnis ist tatsächlich 47871 = $BAFF

Konzentrieren wir uns nun ganz auf das Binärsystem und die Operationen, die man darin ausführen kann.

Die binäre Addition

Die binäre Addition wird genauso durchgeführt wie die schriftliche Addition unter Beachtung der Überträge. Für die Addition einer Stelle gelten dabei folgende Regeln:

```
0 + 0     ergibt 0
0 + 1     ergibt 1
1 + 0     ergibt 1
1 + 1     ergibt 0 und 1 Übertrag
1 + 1+1   ergibt 1 und 1 Übertrag
```

Als ein Beispiel mag folgende Addition dienen:

```
  1101101110
+ 0011000111
 10000110101
```

Die Subtraktion

Auch die Subtraktion von Binärzahlen entspricht dem üblichen Vorgehen mit Bleistift und Papier. Wenn die abzuziehende Ziffer größer ist als jene, von der sie abgezogen werden soll, so wird zum tatsächlichen negativen Ergebnis die Basis addiert und dafür eins mehr von der nächsten Stelle abgezogen (geborgt). Folgende Regeln sind gültig:

```
0 – 0     ergibt 0
0 – 1     ergibt 1 und 1 geborgt
1 – 0     ergibt 1
1 – 1     ergibt 0
0 – 1–1   ergibt 0 und 1 geborgt
1 – 1–1   ergibt 1 und 1 geborgt
```

Hierzu wieder ein Beispiel:

```
  1101101110
– 0011000111
  1010100111
```

Die Darstellung von negativen Zahlen

Es gibt mehrere Möglichkeiten, negative Zahlen darzustellen. Hier soll allerdings nur die am häufigsten verwendete behandelt werden: die sogenannte Zweierkomplementdarstellung. Normalerweise lassen sich mit einem Wort von n Bits genau 2^n verschiedene Zahlen darstellen, nämlich von 0 bis 2^{n-1}. Bei 8 Bits beispielsweise ergibt sich ein Bereich von 0 bis 255, was $2^8 = 256$ darstellbaren Zahlen entspricht. Wenn dieser Bereich – etwa durch eine Addition – überschritten wird, reichen die verfügbaren Stellen nicht mehr aus, um die Zahl korrekt darzustellen: Bei 8 Bit ist das Ergebnis nur noch modulo 256 richtig. Die Grundidee der Zweierkomplementdarstellung ist nun folgende: Man zieht die Grenze nicht mehr bei 0 bzw. 2^n, sondern betrachtet die 0 als Mitte des darstellbaren Zahlenbereichs. Die "Überlaufsgrenze" wird bei der positiven Zahl 2^{n-1} gezogen, bei 8 Bit also bei 128; alle Zahlen, die kleiner sind, werden als positiv betrachtet, alle die größer oder gleich sind, als negativ. Um -a darzustellen, benutzt man das Bitmuster der positiven Zahl $2^n - a$. In unserem Beispiel hieße das, daß man Zahlen von –128 bis +127 darstellen könnte, wobei der Bereich –128 bis –1 bei der Betrachtung als positive Zahlen in dieser Reihenfolge +128 bis 255 entspricht, während 0 bis 127 sich selbst entspricht.

Die Operation 2^n–a wird als Komplementbildung bezeichnet. Bei binären Zahlen ist die Komplementbildung auch ohne eine wirkliche Subtraktion auszuführen. Und so geht sie vor sich:

- Jedes Bit der zu komplementierenden Zahl wird invertiert; d.h. jede 0 wird in eine 1 verwandelt und umgekehrt.

- Zum Ergebnis der Invertierung wird eins addiert.

Probieren wir das einmal mit der binären Darstellung von 36 in 8 Bit aus:

36 =	% 00100100
invertieren	11011011
1 addieren	11011100

Der besondere Vorteil der Zweierkomplementdarstellung ist nun der, daß die Addition genauso vorgenommen werden kann, als wären beide Zahlen positiv. Der einzige Unterschied ist der, daß bei Zweierkomplementzahlen ein Übertrag auftreten kann, der nicht die gleiche Bedeutung hat wie ein Übertrag bei positiven Zahlen. Wenn das stimmt, müßte also die Summe einer Zahl und ihres Komplements immer 0 ergeben. Zurück zu unserem Beispiel:

```
   00100100      36
+  11011100  +(-36)
(1) 00000000      0
```

Nehmen wir noch einmal zwei andere Zahlen:

```
   00011110      30
+  11111011   +(-5)
 (1)00011001     25
```

Dieses Beispiel zeigt auch, daß es keinen Unterschied macht, ob man eine Zahl B von einer Zahl A abzieht oder das Komplement von B zu A addiert. Der MC68000 nimmt Subtraktionen nach diesem Prinzip vor, nicht nach der früher besprochenen direkten Subtraktionsmethode. Tatsächlich braucht man sich also bei Subtraktionen und Additionen überhaupt nicht darum zu kümmern, ob man mit positiven oder Zweierkomplementzahlen umgeht; es ist nur eine Frage der Interpretation.

Bei der Multiplikation und Division ist das leider nicht so unproblematisch: Dort müssen ganz verschiedene Algorithmen benutzt werden, je nachdem, ob die Zahlen als vorzeichenlos oder vorzeichenbehaftet betrachtet werden. Übrigens kann man das Vorzeichen einer Zweierkomplementzahl leicht erkennen: Ist das höchstwertigste Bit gesetzt, dann ist die Zahl negativ, andernfalls ist sie positiv.

Da der MC68000 Befehle zum Umgang mit BCD-Zahlen bietet, soll hier kurz darauf eingegangen werden.

BCD steht für "Binary Coded Decimal", also binär codierte Dezimalzahl. Bei einer BCD-Zahl wird eine Folge von Dezimalziffern abgespeichert, bei der jede Ziffer vier Bits belegt. Dabei wird der 0 die Bitfolge 0000 zugeordnet, der 1 0001, und so fort bis zur 9, die durch 1001 dargestellt wird. Die Bitfolgen 1010 bis 1111 werden nicht benutzt; einige von ihnen werden gelegentlich zur Darstellung eines Vorzeichens verwendet. Ein Beispiel:

4321 = %0100 0011 0010 0001

Der MC68000 bietet nun Möglichkeiten, BCD-Zahlen zu addieren, zu subtrahieren und zu negieren. Wann ist nun die Verwendung einer BCD-Zahl angebracht?

BCD-Zahlen haben den Vorteil, daß die Umwandlung vom und ins Dezimalsystem besonders einfach ist. Außerdem garantieren sie im Gegensatz zu binären

Zahlen Genauigkeit bis auf die letzte Dezimalstelle, da keine Umwandlungsfehler auftreten können. Aus diesen Gründen werden BCD-Zahlen gelegentlich für kaufmännische Anwendungen eingesetzt, wenn auch nicht gerade auf dem ATARI ST. Nachteilig ist, daß das Rechnen mit BCD-Zahlen sehr aufwendig sein kann. Addition und Subtraktion stellen zumindest auf dem MC68000 kein Problem dar, da sie schon als Maschinenbefehle implementiert sind. Multiplikaton, Division und noch kompliziertere Operationen verlangen hingegen einen viel größeren Programmieraufwand als die gleichen Operationen mit Binärzahlen und brauchen dementsprechend auch ein Vielfaches an Rechenzeit. Bei großen Zahlenmengen macht es sich auch bemerkbar, daß BCD-Zahlen mehr Speicherplatz verbrauchen als Binärzahlen mit gleicher Genauigkeit. Deshalb spielen BCD-Zahlen eine sehr untergeordnete Rolle gegenüber den üblichen Binärzahlen.

Anhang B

Unterschiede verschiedener Assembler

Ziel dieses Anhangs ist es in erster Linie, Ihnen zu ermöglichen, die Programme aus diesem Buch auch mit anderen Assemblern als dem aus dem ATARI-Entwicklungspaket zu verwenden. Nebenbei gibt es noch den einen oder anderen nützlichen Tip zum Umgang mit den angesprochenen Assemblern.

Bekanntlich führt es meistens zu Problemen, wenn ein Programm für einen Assembler an einen anderen angepaßt werden soll. Oftmals handelt es sich nur um kleine Details wie etwa die Verwendung von Semikolons statt Sternchen zur Einleitung einer Bemerkung, doch ist ihre Beseitigung oftmals zeitraubend. Damit Sie sich mit diesen Problemen so wenig wie möglich herumärgern müssen, liegen die angepaßten Quellcodes für die gängigsten Assembler auf einer beim SYBEX-Verlag erhältlichen Programmdiskette vor. Leider führten einige Programme bei manchen Assemblern zu grundsätzlichen Problemen, so daß diese oftmals aus ungeklärten Gründen überhaupt nicht angepaßt werden konnten. Für diese Fälle befinden sich im Ordner "PROGRAMS" die assemblierten ausführbaren Programme, die mit dem ATARI-Assembler AS68 erzeugt wurden.

Die eine oder andere Schwäche der Assembler, die hier angesprochen wird, kann natürlich in neueren Versionen schon längst behoben sein. Deshalb wird immer die Versionsnummer der getesteten Programme angegeben. Bei Programmen, die keine Versionsnummer vorzeigen, ist die Länge in Bytes angegeben. Man kann davon ausgehen, daß spätere Versionen meistens länger sind.

Der Metacomco-Assembler (Version 10.203)

Um die Programme auf der Diskette mit dem Metacomco-Assembler zu verwenden, brauchen Sie nur sämtliche "xxxxxxxx.S"-Dateien in "xxxxxxxx.ASM" umzunennen. Weitere Anpassungen sind nicht notwendig.

Beim Metacomco-Assembler handelt es sich um einen Assembler der konventionellen Sorte. Er besteht aus unabhängigem Editor, eigentlichem Assembler und Linker. Die Syntax entspricht praktisch vollständig der des Assemblers aus dem ATARI-Entwicklungspaket, der im folgenden als ATARI-Assembler bezeichnet wird; es wurden allerdings noch einige Erweiterungen aufgenom-

men, insbesondere Makros. Einige Besonderheiten gibt es trotzdem:

- Die etwas merkwürdige Art des ATARI-Assemblers, mit Zeichenkonstanten umzugehen, wurde nicht kopiert. Nehmen wir folgende Anweisung:

  ```
  MOVE.W    #'A',D0
  ```

  Der ATARI-Assembler macht daraus folgendes:

  ```
  MOVE.W    #$4100,D0
  ```

  ($41 ist der ASCII-Wert des Zeichens A)

  Der Metacomco-Assembler verhält sich hingegen so, wie man es erwarten würde und erzeugt folgenden Code:

  ```
  MOVE.W    #$0041,D0
  ```

  Bei den Programmen dieses Buches wurde diese Operandenart allerdings nicht benutzt.

- Die Standard-Endung für Assembler-Quellcode ist ASM. Leider ist der Metacomco-Assembler in dieser Beziehung genauso starrköpfig wie der ATARI-Assembler. Will man ihm Dateien mit anderen Endungen unterjubeln, dann ist das Resultat eine ruinierte Quellcodedatei und eine mitleidlose Fehlermeldung. Also nur ASM-Dateien verwenden!

- Dieser Assembler beachtet bei Symbolnamen bis zu 32 Zeichen, nicht nur die ersten 8. Natürlich führt das bei der Verwendung von mit dem ATARI-Assembler entwickelten Programmen nicht zu Problemen; man sollte es nur beachten, wenn man Programme auf diesem Assembler schreibt und sie irgendwann auf einen anderen Assembler übertragen will.

- Der Metacomco-Assember kennt die Direktive EVEN nicht; statt dessen garantiert er, daß nach Byte-Daten in jedem Fal auf die nächste gerade Adresse gerundet wird, sofern die folgenden Daten nicht ebenfalls Byte-Daten sind. Lassen Sie die Direktive EVEN also einfach weg.

- Ungewöhnlich ist die Behandlung des BSS-Segments: Der Metacomco-Assembler speichert es zusammen mit den restlichen Daten auf der Diskette ab. Auch vom Linker wird das BSS nicht aus der ausführbaren Datei getilgt, weshalb die Programme etwas länger werden, als sie eigentlich sein sollten. Insbesondere kann man keine riesigen Felder von mehreren 100K im BSS anlegen, da sonst das Programm auf der Diskette auch entsprechend vergrö-

ßert wird, sofern der Platz überhaupt zum Assemblieren ausreicht. Arbeiten Sie statt dessen mit dem MALLOC-Mechanismus.

Die Anleitung bietet zwar im Prinzip genügend Informationen, scheint aber für einen etwas anderen Assembler geschrieben worden zu sein. Der Hauptunterschied ist, daß der Assembler nicht wie angegeben Objekt-Dateien produziert, die von LINK68 weiterverarbeitet werden können, sondern solche für den GST- Linker. Leider funktioniert die Assembler-Option OPT S nicht – sie soll normalerweise in der Objektdatei eine vollständige Symboltabelle erzeugen, wird jedoch ignoriert. Ein symbolischer Debugger bringt also mit diesem Assembler nicht viel.

Der Metacomco-Assembler verwendet übrigens den Linker von GST. Daraus ergibt sich, daß die vom Assembler erzeugten Objektdateien (Endung BIN) zusammen mit sämtlichen von GST angebotenen Compilersprachen problemlos genutzt werden können, etwa GST C. Außerdem kann Code mit Lattice C (auch von Metacomco) zusammengebunden werden.

Für die RAM-Disk werden folgende Dateien benötigt:

ASSEM.TTP	der eigentliche Assembler
LINK.TTP	der GST-Linker
MENU+.PRG	die Benutzeroberfläche
MENU.INF	editierbare Einstellung von MENU+

Dazu kommt noch ein beliebiger Editor.

Die Programme aus diesem Buch brauchen mit keiner anderen Datei zusammengebunden zu werden.

Der GST-Assembler (die Angaben gelten für Version A 246 V 040)

Auch der GST-Assembler besteht aus den drei Teilen Editor, Assembler und Linker. Leider weist seine Syntax besonders bei den Assembler-Direktiven eine ganze Reihe von Unterschieden zum Motorola-Standard und zur Syntax des ATARI-Assemblers auf, obwohl er ansonsten ein leistungsfähiger Makroassembler ist. Auf der im SYBEX-Verlag erhältlichen Diskette zum Buch befinden sich im Ordner GSTASM die Programme dieses Buches in an den GST-Assembler angepaßter Syntax. Leider werden die Programme COLORS aus Abschnitt 6.4 und RAMDISK aus 6.6 vom GST-Assembler nicht korrekt übersetzt und stürzen ab; wenn Sie diese beiden Programme laufen sehen wollen, können Sie jedoch die mit dem ATARI-Assembler erzeugten ausführbaren

Programme COLORS.PRG und RAMDISK.PRG aufrufen. Hier nun die Unterschiede des GST-Assemblers zum ATARI-Assembler:

- Der GST-Assembler kennt die Direktiven TEXT, DATA und BSS nicht. Statt dessen benutzt man die SECTION-Direktive folgendermaßen:

  ```
  SECTION TEXT
  SECTION DATA
  SECTION BSS
  ```

  Der GST-Assembler verlangt auch, daß vor dem ersten Befehl die Direktive SECTION TEXT benutzt wird; er steht also nicht standardmäßig am Anfang im TEXT-Section-Modus.

- Zeichenketten werden nicht in Anführungszeichen, sondern in Hochkommas eingeschlossen. Also statt

  ```
  DS.B      "Hallo Welt",0
  ```

  heißt es beim GST-Assembler

  ```
  DS.B      'Hallo Welt',0
  ```

- Wird der Assembler ohne spezielle Optionen aufgerufen, so erzeugt er beim Assemblieren der Programme dieses Buches Massen von Warnungen. Dabei handelt es sich eigentlich nur um zwei Arten von Warnungen:

  ```
  **** WARNING 51 -- size missing, W assumed
  ```

  Das heißt nur, daß der Assembler standardmäßig Wortbreite angenommen hat, da hinter dem Mnemonik nichts weiter angegeben war. Eigentlich sollte ein Assembler sich darüber nicht weiter aufregen. Wenn die vielen Warnungen Sie stören, schreiben Sie die Endung ".W" hinter alle arithmetischen und logischen Befehle, die keine Endung haben.

  ```
  **** WARNING 5F -- run-time-relocation is required for this expression
  ```

  Nur der dezente Hinweis, daß eine Speicherstelle in die Relozierungsliste aufgenommen wurde. Zum Glück verhindern selbst Hunderte von Warnungen nicht, daß das Programm assembliert wird. Also kümmern Sie sich am besten nicht weiter um diese Marotte des GST-Assemblers.

- Der GST-Assembler hat etwas gegen Symbolnamen, die mit einem Unterstrich beginnen. Da solche Namen in einigen der Programme vorkommen, ersetzen Sie sie durch andere.

- Im Gegensatz zu anderen Assemblern haben die eckigen Klammern [] auch innerhalb von Zeichenketten für den Assembler eine besondere Bedeutung im Zusammenhang mit Makros. Dies sollte man bei der Definition von Zeichenketten beachten.

Der GST-Assembler arbeitet mit dem GST-Linker (Version R132 V039) zusammen. Wie schon beim Metacomco-Assembler erwähnt, hat dieser Linker die Eigenheit, das BSS-Segment mit dem ausführbaren Programm zusammen abzuspeichern.

Die vom GST-Assembler erzeugten BIN-Dateien können in sämtlichen Compilersprachen von GST, in Lattice C und dem Metacomco-Assembler eingebunden werden. Wenn Sie den GST-Assembler in einer RAM-Disk benutzen wollen, werden folgende Dateien gebraucht:

GSTC.LNK	editierbare Anweisung für den Linker
LINKASM.OVR	gehört zu Assembler + Linker
ASM.PRG	der eigentliche Assembler
GSTASM.PRG	die Benutzeroberfläche
LINK.PRG	der GST-Linker
GSTASM.RSC	gehört zu GSTASM.PRG

Dazu kommt natürlich noch ein beliebiger Editor, der den Namen EDIT.PRG haben sollte, damit er von der Benutzeroberfläche angesprochen werden kann. Die Programme dieses Buches brauchen auch beim GST-Assembler mit nichts anderem zusammengebunden zu werden.

Der Seka-Assembler (Version 1.1)

Der Seka-Assembler ist ein integriertes Assembler-Editor-Debugger-Paket. Zu bemängeln ist nur, daß die Direktiven des Assemblers nicht so reichlich wie bei den anderen Assemblern angeboten werden; der eingebaute Editor ist ein Zeileneditor nach alter Machart und für mehr als kleine Änderungen äußerst unhandlich. Ich empfehle deshalb, zum Schreiben des Quelltextes einen unabhängigen Editor zu verwenden und nur zum Assemblieren und Austesten den Seka-Assembler zu laden. Praktisch ist dabei, daß der Seka-Assembler äußerst schnell assembliert. Als ein Nachteil muß es allerdings aufgefaßt werden, daß der Seka-Assembler über keinen Linker verfügt.

Es erwies sich als unpraktisch, daß die Programme aus diesem Buch mit der GEMDOS-Funktion TERM beendet werden, weshalb sich nach einem Testlauf jedesmal der Seka-Assembler verabschiedet. Deshalb ist in jeder Quellcodedatei für den Seka-Assembler auf der beim SYBEX-Verlag erhältlichen Programmdiskette zu diesem Buch das Programmende mit dem Label "bp:"

gekennzeichnet. Wenn Sie beim Starten des Programms 'bp' als Breakpoint angeben, erhält der Assembler nach dem Ablauf wieder die Kontrolle.

In der Syntax weist der Seka-Assembler eine ganze Reihe von Unterschieden zum ATARI-Assembler auf:

- Labels müssen immer in Spalte Null beginnen und mit einem Doppelpunkt abgeschlossen werden. Bemerkungen dürfen nur mit einem Semikolon (;) eingeleitet werden; das gilt auch dann, wenn es sich um Bemerkungen nach einem Assemblerbefehl handelt. Der Stern (*) wird nicht anerkannt.

- Der Assembler kennt die Direktiven TEXT, DATA und BSS nicht, da er normalerweise direkt in den Speicher assembliert.

- Der Seka-Assembler verarbeitet keine Abwandlungen wie ADDA oder CMPI. Benutzen Sie statt dessen die Grundform ADD bzw. CMP. Die Quick-Varianten werden allerdings erkannt.

- Statt EQU wird für das Zuweisen eines Wertes an ein Symbol das Gleichheitszeichen verwendet:

```
SYMBOL = 42
```

- Die Direktive DS (define Storage) muß durch BLK (Block) ersetzt werden. Die Anhängsel ".B", ".W" und ".L" werden allerdings genauso verwendet.

- Auch dieser Assembler verabscheut Symbolnamen, die mit einem Unterstrich (_) beginnen.

Die an die Syntax des Seka-Assemblers angepaßten Programme befinden sich auf der beiliegenden Diskette im Ordner "SEKA". Aus undurchsichtigen Gründen verarbeitet dieser Assembler die Programme zum Linienziehen für hohe und niedrige Auflösung (LINEHI.S und LINELO.S) nicht korrekt; ebenso das GEM-Beispiel GEM.S. Hier bleibt Ihnen nur, direkt die mit dem ATARI-Assembler erzeugten ausführbaren Programme LINESL.TOS, LINESH.TOS und GEM.PRG zu starten. Überhaupt benahm sich die mir vorliegende Version etwas merkwürdig; es kam öfter zu unmotivierten Abstürzen.

Die Verwendung eines separaten Editors kann die Arbeit mit dem SEKA-Assembler sehr erleichtern. Für den Fall, daß Sie das anpassungsfähige MENU+ von Metacomco besitzen, liegt im Ordner SEKA die Datei MENU.INF bei, die für den SEKA-Assembler und den Editor EDIT.TTP gedacht ist. Wenn Sie einen anderen Editor verwenden, so ändern sie den Editoraufruf in MENU.INF entsprechend.

Data Becker Profimat ST (Länge 126674 Bytes)

Beim Profimat handelt es sich um ein integriertes Paket aus Editor, Assembler und Debugger mit durchgehend sehr anwenderfreundlicher GEM-Benutzeroberfläche. Leider gilt auch für den Profimat, daß kein linkbarer Code erzeugt wird. Mit den Direktiven ILABEL/IBYTES ist zwar prinzipiell ein Einfügen von fertig assemblierten Programmen möglich, doch ist die Benutzung unhandlich, und die Einschränkung auf PC-relativen Code macht diese Möglichkeit oft unbrauchbar. Wenden wir uns zunächst einmal dem erzeugten Code zu: Wenn Sie die Programme aus diesem Buch verwenden wollen, sollten Sie die Option "PC-relativ" (im Assembler-Menue) abschalten und "relozierbar" einschalten. Damit verhält sich der Profimat genauso wie etwa der ATARI-Assembler: Es werden nicht automatisch PC-relative Adressierungsarten erzeugt, aber die üblichen Relozierungs-Daten angelegt. Wollen sie die Option "PC-relativ" verwenden, so müssen Sie sich bei den Adressierungsarten etwas einschränken, denn bei den meisten Befehlen kann nur der Quelloperand PC-relativ adressiert werden. Natürlich kann man sich beim Zieloperanden mit LEA behelfen, aber das Programm wird dadurch ineffizienter.

Bei der Anpassung des Quellcodes aus diesem Buch gibt es folgendes zu beachten:

- Der Profimat akzeptiert keine Sternchen als Einleitung einer Bemerkung. Statt dessen muß das Semikolon verwendet werden, und zwar nicht nur am Anfang von Bemerkungszeilen, sondern auch dann, wenn einem Assemblerbefehl eine Bemerkung folgt.

- Der Profimat unterscheidet im Gegensatz zu allen anderen Assemblern zwischen Symbolen und Konstanten. Einem Symbol kann man jeden beliebigen Text zuweisen; An der Stelle der Verwendung dieses Symbols wird dann dieser Text eingesetzt – was übrigens auch die erwähnte Direktive REG des ATARI-Assemblers ersetzt. Eine Konstante kann hingegen nur einen numerischen Wert erhalten. Eine Konstante ist also genau das, was bei anderen Assemblern als Symbol bezeichnet wird. Symbole werden mit der Direktive EQU initialisiert, Konstanten mit dem Gleichheitszeichen (=). Da nun im Quellcode für den ATARI-Assembler immer EQU verwendet wird, werden somit nur Symbole verwendet. Natürlich funktioniert das auch, nur muß der Assembler durch die textuelle Ersetzung unnötigen Aufwand treiben, was die Assemblierzeit erhöht. Deshalb sollte "EQU" durch "=" ersetzt werden.

- Nach der Reservierung von Platz für Byte-Daten muß unbedingt der Programmzähler begradigt werden, da dies nicht automatisch geschieht. Dazu

dient nicht EVEN wie beim ATARI-Assembler, sondern die Direktive ALIGN.

- Der Profimat akzeptiert keine Labels oder Makro-Namen, die reservierten Namen wie Opcodes oder Direktiven gleichen (etwa INPUT, START). Benennen Sie solche Labels um.

- Die mit Vorliebe benutzte Endung für Quellcode ist .Q (für Quellcode, schließlich kommt der Profimat aus Deutschland). Andere Endungen werden aber beim Laden und Speichern auch akzeptiert.

- Der Makro-Mechanismus funktioniert etwas anders als etwa beim Metacomco-Assembler. Hier kann ich nur auf das sehr ausführliche Handbuch verweisen.

Assembler von Eckhard Kruse (Public Domain Software)

Wenn Sie dem Buch ausgerechnet mit diesem Assembler folgen wollen, kann ich Ihnen nur einen Rat geben: Lassen Sie die Finger davon. Die Assemblerprogramme dieses Buches an diesen Assembler anpassen zu wollen, ist so gut wie aussichtslos. Die Probleme fangen schon damit an, daß der Assembler keine Zuweisung von Werten an Symbole zuläßt (Labels hat er immerhin). Außerdem fehlt die eine oder andere nützliche Direktive.

Omicron IDEAL

IDEAL steht für "Integrierter Debugger-Editor-Assembler-Linker", also handelt es sich auch hier um ein einziges Programm, das alle Funktionen vereinigt. Leider ist der Assembler äußerst dürftig; viele wichtige Direktiven fehlen, und nicht einmal Zuweisungen von Werten an Symbole werden zugelassen. Deshalb scheidet der IDEAL-Assembler für die Benutzung mit den Programmen aus diesem Buch aus. Empfehlenswert ist hingegen der integrierte Debugger, der viele nützliche Funktionen zur Fehlersuche bietet.

Anhang C

Tips für Umsteiger

Dieser Anhang richtet sich speziell an jene, die mit der Maschinensprache eines 8-Bit-Prozessors wie 6502/6510 oder der 8086-Serie vertraut sind, aber jetzt zum MC68000 "aufsteigen". Die folgenden kleinen Tips sollen helfen, Fehler zu vermeiden, die dadurch entstehen, daß man – vielleicht unbewußt – Konzepte vom einen zum anderen Prozessor übertragen will. Zunächst die Unterschiede zum 6502/6510:

- Worte und Langworte dürfen auf dem MC68000 nur an geraden Adressen angesprochen werden, Bytes an jeder beliebigen Adresse.

- Wird ein Byte oder ein Wort in einem Datenregister manipuliert, so bleiben die nicht zum Byte bzw. Wort gehörenden Bits unverändert.

- Die Adreßregister arbeiten nur mit Wort- oder Langwortoperationen zusammen

- Wird ein Wort in ein Adreßregister geschrieben oder mit dessen Inhalt verknüpft, so wird das Wort vorher auf Langwortbreite erweitert.

- Der Stack wächst auf dem MC68000 nach unten, nicht nach oben. Der Stackpointer zeigt immer auf das unterste beschriebene Byte des Stacks.

- Denken Sie immer an den Unterschied zwischen User- und Supervisormodus. Im Usermodus dürfen nicht alle Befehle ausgeführt werden, und es dürfen nicht alle Speicherbereiche benutzt werden. Jeder der Modi hat seinen eigenen Stackpointer.

- Das Statusregister wird in Systembyte und Userbyte unterteilt. Im Usermodus darf nur auf das Userbyte zugegriffen werden, in dem die Flags stehen. Es gibt keine Befehle, um Flags direkt einzeln zu setzen oder zu löschen; statt dessen wird MOVE to CCR, ANDI to CCR oder ORI to CCR verwendet.

- Adressen sind im Normalfall 32 Bit lang.

- Den 6502-Befehlen ROL und ROR entsprechen nicht die gleichnamigen 68000-Befehle, sondern ROXL und ROXR.

Einige dieser Tips gelten ebenfalls für Umsteiger von der Intel-8086-Serie. Hinzu kommt noch folgendes:

- Bei den Assemblern des 8086/88 und der Prozessoren der 80xxx-Serie werden Quelle und Ziel in umgekehrter Reihenfolge angegeben wie beim 68000, also zuerst das Ziel, dann die Quelle.

- Beachten Sie, daß bei den Intel-Prozessoren genau wie beim 6502 immer zuerst niederwertige, dann höherwertige Einheiten (Bytes oder Worte) im Speicher liegen. Beim 68000 ist es umgekehrt.

- Die gesamte aufwendige Speicherverwaltung der Intel-Prozessoren fällt auf dem 68000 weg. Es gibt keine Segmente, sondern absolute Adressen.

Anhang D

Tips zum Einbinden von Assembler in höhere Programmiersprachen

Nur selten werden umfangreiche Programme auf einem System wie dem ATARI ST noch vollständig in Assembler implementiert, denn ein schneller Prozessor kann so mancher müden Programmiersprache Beine machen, und mit dem Speicherplatz braucht man auch nicht zu sparen. So entschließt man sich oft zu einem Kompromiß: Nur die Prozeduren, bei denen es wirklich nicht anders geht oder die viel Rechenzeit erfordern, werden in Assembler programmiert. Nun ist eine gute Verbindung gefragt, denn natürlich müssen die Assemblerroutinen mit dem restlichen Programm Informationen austauschen können. Wie das gemacht wird, zeigt Ihnen dieser Anhang.

Wenn Sie Ihre Assemblerprogramme zusammen mit einer Compilersprache verwenden wollen, gibt es zunächst einmal zwei Möglichkeiten: Assembler und Compiler verwenden das gleiche Format für ihren Objektcode, oder eben nicht. Im ersten Fall können Sie den vom Compiler erzeugten Code einfach mit dem Assemblercode zusammenbinden und erhalten so eine einzige ausführbare Datei. Wenn Assembler und Compiler nicht den gleichen Linker verwenden oder Sie gar Assemblercode in einer Interpretersprache aufrufen wollen, ist das nicht ganz so einfach, aber auch machbar. Nun stellt sich für Sie die Frage, welche Teile dieses Anhangs für Sie überhaupt interessant sind. Nun, wenn Sie auf Zusammenarbeit mit C aus sind, sollten Sie in jedem Fall den Abschnitt über Digital Research C lesen, egal, welchen Compiler Sie verwenden. Das gleiche gilt für Pascal-Anhänger, da die Prozedurkonventionen von C und Pascal sich weitgehend gleichen. Sogar für die Anwender von GFA-BASIC ist der Abschnitt interessant, da dieser BASIC-Interpreter die C-Konvention beherrscht. Am Ende dieses Anhangs finden Sie Tips für den Fall, daß Ihre Programmiersprache hier nicht aufgeführt sein sollte.

1. Verwendung eines gemeinsamen Linkers

Digital Research C aus dem ATARI-Entwicklungspaket

Dieser C-Compiler verwendet nicht nur den gleichen Linker LINK68 wie der ATARI-Assembler, sondern auch den ATARI-Assembler selbst, denn der C-

Compiler erzeugt aus dem C-Quellcode zunächst reinen mnemonischen Assemblercode, der dann von AS68 assembliert werden kann. Wer will und genügend Zeit hat, kann vorher etwas am Assemblercode ändern, etwa um das fertige Programm zu beschleunigen. Nehmen wir nun an, man wollte von C aus auf die Line-A-Routinen zugreifen. Man kommt nicht darum herum, den eigentlichen Line-A-Aufruf in Assembler zu formulieren, da es keine Möglichkeit gibt, den Compiler dazu zu bringen, die notwendigen Befehlscodes $A00x zu erzeugen. In unserem Beispiel sollen der Einfachheit halber nur zwei Aufrufe implementiert werden: "line" und "put pixel", im Beispiel "plot" genannt (siehe Dokumentation der Line-A-Aufrufe im Abschnitt 4.5). Damit die Aufrufe von C aus einfach einzusetzen sind, sollen die Assemblerroutinen genau wie C-Prozeduren aufgerufen werden, speziell in der Form

```
line(x1,y1,x2,y2)
```

und

```
plot(x,y,farbe)
```

wobei es sich bei sämtlichen übergebenen Werten um Integer-Werte handelt, also um 16-Bit-Worte.

Zu diesem Zweck muß man zunächst über die C-Prozedurkonvention auf dem MC68000 Bescheid wissen. Wenn eine C-Prozedur eine andere aufruft, werden zuerst die Parameter in umgekehrter Reihenfolge auf dem Stack abgelegt. Der im C-Quellcode am weitesten rechts stehende Wert wird zuerst abgelegt, der links stehende zuletzt. Dann erfolgt der eigentliche Unterprogrammaufruf mittels JSR oder BSR. Danach ist die aufrufende Prozedur noch für das Zurücksetzen des Stackpointers verantwortlich. Genauer finden Sie diesen Vorgang in Kapitel 2 erklärt.

Auf dem Stack belegt ein "int" 2 Bytes und ein "long" 4 Bytes, wobei es egal ist, ob es sich um vorzeichenlose oder vorzeichenbehaftete Zahlen handelt. Da auf dem Stack nur Einheiten von mindestens Wortgröße abgelegt werden dürfen, wird ein "char" auf Wortgröße erweitert; der eigentliche Wert befindet sich dann im niederwertigen Byte des abgelegten Wortes. Ein "float" belegt den gleichen Platz wie ein Langwort, ein "double" erstreckt sich über zwei aufeinanderfolgende Langworte. Bei Pointertypen werden immer die absoluten Adressen der Objekte, auf die sie zeigen, in Langwortbreite übergeben. Bedenken Sie, daß niemals Zeichenketten selbst, sondern immer nur Zeiger auf mit einem Nullbyte abgeschlossene Zeichenketten übergeben werden.

Beim Eintritt in die aufgerufene Routine geschieht jedoch etwas anderes als bei der früher vorgestellten Aufrufkonvention: Am Anfang der Routine steht ein

Befehl der Form

```
LINK  A6,#n
```

Damit wird ein Bereich auf dem Stack, ein sogenannter Stackframe, angelegt. Rufen wir uns noch einmal ins Gedächtnis, was der LINK-Befehl tut:

- Der Inhalt des angegebene Adreßregisters wird auf dem Stack abgelegt.
- Der alte Wert des Stackpointers wird in das Adreßregister übertragen.
- Zum Stackpointer wird der angegebenen Wert (16 Bit) addiert. Normalerweise ist n ein kleiner negativer Wert. So sorgt dieser Befehl dafür, daß am unteren Ende des Stacks ein Bereich von n Bytes angelegt wird, der in C für die Aufbewahrung von lokalen Variablen dient. Damit der vorherige Wert des Stackpointers noch erreichbar ist (etwa zur einfacheren Adressierung der Parameter) wird er in A6, dem sogenannten Frame Pointer, erhalten. Damit dieser Wert bei mehreren geschachtelten Aufrufen nach dieser Konvention nicht verlorengeht, wird vorher der alte Wert von A6 auf dem Stack gesichert. Abbildung 24 zeigt das untere Ende des Stacks, nachdem der LINK-Befehl ausgeführt wurde.

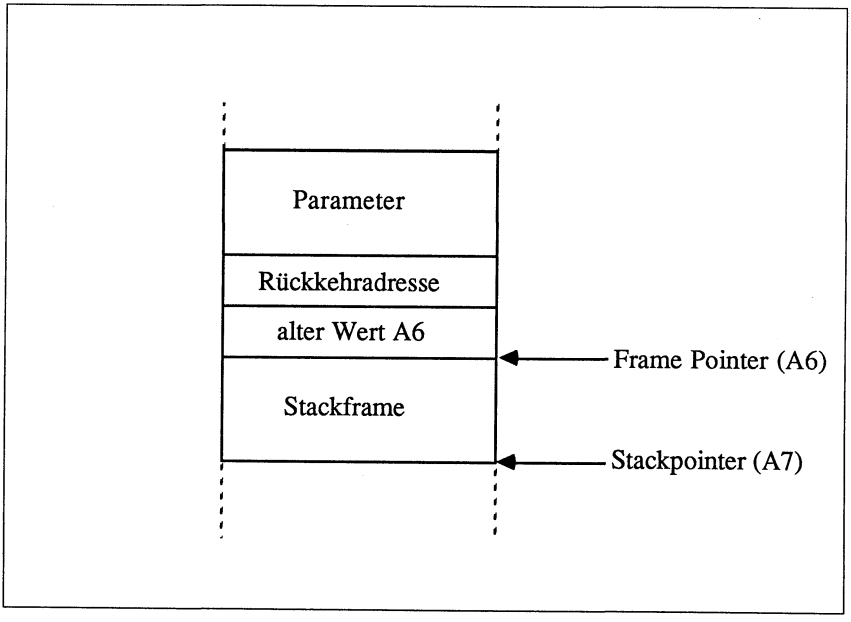


Abb. A.1: Daten auf dem Stack nach Ausführung des LINK-Befehls

Am Ende der Prozedur müssen die Auswirkungen des LINK-Befehls wieder rückgängig gemacht werden. Dies geschieht mit dem Befehl

```
UNLK  A6
```

Damit wird der alte Wert des Stackpointers wieder aus A6 in den Stackpointer kopiert und dann der frühere Wert aus A6 vom Stack geholt. Damit ist das Stackframe gelöscht.

Nach diesem Befehl erfolgt mit RTS der Rücksprung in die aufrufende Prozedur. Vorher werden allerdings noch etwa auftretende Rückgabewerte ins Register D0 geladen. Dabei spielt es keine Rolle, ob ein Rückgabewert 8, 16 oder 32 Bit lang ist. Die einzige Ausnahme muß bei "double"-Werten gemacht werden: Sie werden in den Registern D0 und D1 zurückgegeben.

Für den Assemblerprogrammierer ist die Geschichte mit den Stackframes nicht von so großer Bedeutung – außer wenn es darum geht, rekursiv aufrufbare Assembler-Prozeduren zu schreiben, die lokale Variablen verwenden. Wichtig ist nur, daß das Register A6 – und natürlich der Stackpointer – innerhalb der Prozedur nicht verändert werden dürfen. Das gleiche gilt für die Datenregister D3 bis D7 und die Adreßregister A2 bis A5, die möglicherweise Registervariablen enthalten. Sollten Sie nicht ohne diese Register auskommen, so können Sie die Inhalte der benötigten Register am Anfang der Routine mit einer MOVEM-Anweisung sichern und vor dem RTS entsprechend zurückholen. Nun wissen wir genug, um die Assemblerroutinen zu formulieren. Betrachten wir das folgende Listing. Zunächst werden die Symbole "_line" und "_plot", die später nach außen hin sichtbar sein sollen, mit XDEF global bekannt gemacht. Im Gegensatz zum C-Aufruf müssen die Funkionsnamen mit einem Unterstrich versehen werden, da der C-Comiler alle externen Lables mit einem Unterstrich versieht. Darauf folgt eine Reihe von Konstantendefinitionen, die die relativen Adressen einiger Line-A-Variablen bekanntmachen. Die Funktion dieser Variablen ist hier nicht weiter von Bedeutung, wird aber im Kapitel 4 eingehend erläutert.

```
************************************************************
* AS68C.S                                                  *
* Beispielprogramm zum Einbinden von Assembler             *
* in Hochsprachen                                          *
* für Metacomco-Assembler oder ATARI-Assembler mit         *
* Digital Research C,                                      *
* Megamax C oder GST c mittels exec()                      *
* ermöglicht Zugriff auf Line-A-Routinen line und          *
* put_pixel (plot)                                         *
************************************************************
```

```
* zuerst die von außen erreichbaren Symbole
          XDEF      _line
          XDEF      _plot
*
* relative Adressen der benutzten Line-A Variablen
intin     EQU       8
ptsin     EQU       12
fg_bp_1   EQU       24
fg_bp_2   EQU       26
fg_bp_3   EQU       28
fg_bp_4   EQU       30
wrt_mod   EQU       36
x1        EQU       38
y1        EQU       40
x2        EQU       42
y2        EQU       44
*
* Line-A-Opcodes
A_INIT    EQU       $A000
A_PUT_PIX EQU       $A001
A_LINE    EQU       $A003
*
* Aufruf: line(x1,y1,x2,y2)
_line      link      a6,#0           * SP nach a6
          DC.W      A_INIT          * Variablenadressen holen
          move      8(a6),x1(a0)    * Parameter in die
          move      10(a6),y1(a0)   * Line-A-Variablen
          move      12(a6),x2(a0)   * schreiben
          move      14(a6),y2(a0)   *
          move      #1,fg_bp_1(a0)  * Farbe 1
          clr       fg_bp_2(a0)     *
          clr       fg_bp_3(a0)     *
          clr       fg_bp_4(a0)     *
          move      #2,wrt_mod(a0)  * Schreibmodus: XOR
          DC.W      A_LINE          * Line-A-Aufruf
          unlk      A6              * SP wiederherstellen
          rts                       *
*
* Aufruf: plot(x,y,color)           *
_plot      link      a6,#0          *
          DC.W      A_INIT          * Variablenadressen holen
          move.l    ptsin(a0),a1    * Adresse des ptsin-
                                    * Feldes holen
          move      8(a6),(a1)      * Koordinaten ins
          move      10(a6),2(a1)    * ptsin-Feld schreiben
          move.l    intin(a0),a1    * Adresse des intin-
                                    * Feldes holen
          move      12(a6),(a1)     * Farbe ins intin-
                                    * Feld schreiben
          DC.W      A_PUT_PIX       * Line-A-Aufruf
          unlk      a6              *
          rts                       *
*
          END
```

Interessant wird es ab dem Label _"line", dem Einsprungpunkt. Zunächst wird der Befehl LINK A6,#0 ausgeführt, der ein Stackframe der Länge 0 anlegt. Eigentlich könnte man darauf verzichten; LINK und UNLK werden hier nur verwendet, um ein Beispiel für ihre Verwendung zu geben und außerdem deshalb, weil es nun einmal Konvention ist. Beachten Sie jedoch, daß alle relativen Stackadressen um 4 verringert werden müssen, wenn Sie diesen Befehl weglassen. Der erste Parameter würde in diesem Fall nicht mehr mit 8(SP), sondern mit 4(SP) erreicht. Nun weiter im Programm: Als nächstes holt sich die Routine mit dem Opcode A_INIT die Anfangsadresse der Line-A-Variablen ins Register A0. Daraufhin werden die Parameter 1 bis 4 der Reihe nach in die entsprechenden Line-A-Variablen kopiert. Es wird nur noch der Farbindex auf 1 und der Schreibmodus auf XOR gesetzt, und der Einsprung in die Line-A-Routinen kann erfolgen. Schließlich wird noch der UNLK-Befehl ausgeführt, und das war's dann auch schon. Ein Aufruf von "plot" läuft ähnlich ab; eine zweite Prozedur wurde hier nur aufgenommen, um zu zeigen, daß die Methoden des Einbindens auch mit anderen Einsprungadressen als der logischen Adresse Null funktionieren. Dies wird hauptsächlich bei späteren Beispielen von Bedeutung sein.

Natürlich sind diese beiden Prozeduren noch nicht perfekt; man könnte sich noch mehr sinnvolle Parameter vorstellen, bei "line" etwa den Schreibmodus oder die Farbe. Die Routinen sind natürlich nur als Beispiele gedacht; wenn Sie sie allerdings nützlich finden, können Sie sie natürlich noch verbessern und erweitern.

Sehen wir uns nun an, wie man diese Prozeduren von C aus aufruft:

```
/* AUFRUF1.C
   erster Beispielaufruf von AS68_LC aus C heraus
   für jeden C-Compiler, der den gleichen Linker wie der
   verwendete Assembler benutzt */

main()
{ int i;                    /* Zählvariable              */
  for(i=0;i<320;i++)        /* ein paar Linien ziehen    */
    line(i,0,319-i,199);
  gemdos(8);                /* auf Taste warten          */
}
```

Wie Sie sehen, braucht eine Assemblerprozedur einfach nur aufgerufen zu werden. Der C-Compiler nimmt automatisch an, daß in einem Modul nicht definierte Prozeduren sich in einem anderen befinden. Aber Vorsicht: Wenn die Prozeduren einen anderen Rückgabewert als "int" haben, etwa "long", dann müssen sie als extern deklariert werden. Nehmen wir an, "line" gäbe ein Langwort zurück. In diesem Fall müßte man an den Anfang des C-Codes schreiben:

```
extern long line();
```

oder, völlig gleichwertig

```
long line();
```

So weiß der C-Compiler, daß diese Prozedur in einem anderen Modul definiert ist und ein Langwort zurückgibt.

Natürlich können nicht nur Prozeduren, sondern auch Variablen aus Assemblerprozeduren global zugänglich gemacht werden, einfach indem ihr Name mittels XDEF exportiert wird. Um vom C-Programm darauf zugreifen zu können, müssen diese Variablen allerdings als extern deklariert werden, etwa in der Form

```
extern int asmvar;
```

Um umgekehrt von Assembler aus auf globale C-Variablen zugreifen zu können, müssen ihre Namen im Assemblerprogramm nur mittels XREF bekanntgemacht werden. Sie können dann damit arbeiten, als wären sie in Ihrem Assemblerprogramm definiert, denn globale Variablen werden vom C-Compiler automatisch exportiert.

Es ist auch möglich, C-Prozeduren von Assembler aus aufzurufen. Auch hier müssen sie nur die Prozedurnamen mit XDEF bekanntmachen. Legen Sie einfach die Parameter in der rechts- nach-links-Reihenfolge auf dem Stack ab, und rufen Sie die C-Prozedur mit "JSR name" auf. Nicht vergessen, den Stack zu korrigieren!

Wie wird das Ganze nun compiliert und gebunden? Um von den beiden Quellcodedateien AUFRUF1.C und AS68C.S zum fertigen Programm AUFRUF1.TOS zu gelangen, sind im einzelnen folgende Kommandos nötig (der rechts abgesetzte Teil gehört natürlich nicht dazu, sondern dient der Dokumentation):

```
as68 -l as68c.s                                          assemblieren
cp68 aufruf1.c aufruf1.i                                 C-Preprozessor
c068 aufruf1.i aufruf1.1 aufruf1.2 aufruf1.3 -f
                                                         C Pass 1
c168 aufruf1.1 aufruf1.2 aufruf1.s
                                                         C Pass 2
as68 -l -u aufruf1.s                                     Compiler-Output assemblieren
link68 aufruf1.68k=gems,apstart,aufruf1,as68c,osbind,gemlib
                                                         der große Linker-Aufruf
relmod aufruf1.68k aufruf1.tos                           das unvermeidliche Relmod
```

Danach können natürlich die Zwischendateien AUFRUF1.I, AUFRUF1.1, AUFRUF1.2, AUFRUF1.3, AUFRUF1.S, AUFRUF1.68K und eventuell AUFRUF1.O und AS68C.O gelöscht werden. Dies nur für den Fall, daß Sie keine Batch-Datei zur Bedienung des Compilers haben.

Die gesamte hier vorgestellte Prozedurkonvention gilt nicht nur für den C-Compiler aus dem ATARI-Entwicklungspaket, sondern für alle C-Compiler auf dem ATARI ST. Schließlich gibt es für C nicht nur einen Standard, sondern auch Leute, die sich daran halten. Geringfügige Abweichungen treten natürlich trotzdem auf. Betrachten wir deshalb andere C-Compiler.

Lattice C (von Metacomco)

Lattice C fällt unter den C-Compilern für den ST etwas aus der Reihe, denn bei ihm ist ein "int" nicht 16, sondern 32 Bit lang und somit gleichbedeutend mit "long". 16 Bit lange Variablen gibt es allerdings auch: Sie werden mit "short" bezeichnet. (Bei den anderen Compilern ist "short" gleichbedeutend mit "int", also 16 Bit.) Ungewöhnlich ist das Verfahren, alle Parameter auf dem Stack auf 32 Bit zu erweitern, egal, ob es sich um "long", "int", "short" oder "char" handelt. Kürzere Einheiten als 32 Bit befinden sich in den unteren Bytes des übergebenen Langwortes. Dies gilt es bei der Adressierung der Parameter zu beachten. Auf "int"-Parameter wird also (nach Verwendung des LINK-Befehls) nicht mehr mit den Adressen 8(sp), 10(sp), 12(sp) usw. zugegriffen, sondern mit 10(sp), 14(sp), 18(sp) usw. Beachten Sie, daß das erste Langwort zwar bei 8(sp) beginnt, der uns interessierende 16-Bit-Wert jedoch erst bei 10(sp). Es folgt das entsprechend abgeänderte Listing:

```
************************************************************
* AS68LC.S                                                 *
* Beispielprogramm zum Einbinden von Assembler in          *
* Hochsprachen                                             *
* für Metacomco-Assembler oder ATARI-Assembler             *
* mit Lattice C Compiler                                   *
*                                                          *
* ermöglicht Zugriff auf Line-A-Routinen line und          *
* put_pixel (plot)                                         *
************************************************************
* zuerst die von außen erreichbaren Symbole
          XDEF      line
          XDEF      plot
*
* relative Adressen der benutzten Line-A-Variablen
intin     EQU       8
ptsin     EQU       12
```

```
fg_bp_1    EQU       24
fg_bp_2    EQU       26
fg_bp_3    EQU       28
fg_bp_4    EQU       30
wrt_mod    EQU       36
x1         EQU       38
y1         EQU       40
x2         EQU       42
y2         EQU       44
*
* Line-A-Opcodes
A_INIT     EQU       $A000
A_PUT_PIX  EQU       $A001
A_LINE     EQU       $A003
*
* Aufruf: line(x1,y1,x2,y2)
line       link      a6,#0              * SP nach a6
           DC.W      A_INIT             * Variablenadressen
                                        * holen
           move      10(a6),x1(a0)      * Parameter in die
           move      14(a6),y1(a0)      * Line-A-Variablen
           move      18(a6),x2(a0)      * schreiben
           move      22(a6),y2(a0)      *
           move      #1,fg_bp_1(a0)     * Farbe 1
           clr       fg_bp_2(a0)        *
           clr       fg_bp_3(a0)        *
           clr       fg_bp_4(a0)        *
           move      #2,wrt_mod(a0)     * Schreibmodus: XOR
           DC.W      A_LINE             * Line-A-Aufruf
           unlk      A6                 * SP wiederherstellen
           rts                          *
*
* Aufruf: plot(x,y,color)              *
plot       link      a6,#0              *
           DC.W      A_INIT             * Variablenadressen
                                        * holen
           move.l    ptsin(a0),a1       * Adresse des ptsin-
                                        * Feldes holen
           move      10(a6),(a1)        * Koordinaten ins
           move      14(a6),2(a1)       * ptsin-Feld schreiben
           move.l    intin(a0),a1       * Adresse des intin-
                                        * Feldes holen
           move      18(a6),(a1)        * Farbe ins intin-Feld
                                        * schreiben
           DC.W      A_PUT_PIX          * Line-A-Aufruf
           unlk      a6                 *
           rts                          *
*
           END
```

In allen anderen Punkten gleicht die Prozedurkonvention von Lattice C genau der des C-Compilers aus dem Entwicklungspaket. Für den Aufruf können Sie obiges C-Beispielprogramm verwenden.

Um ein Programm aus Metacomco-Assembler und Lattice C zusammenzubinden, gehen Sie am besten so vor:

- Assemblieren Sie zunächst AS68LC.ASM. Es wird die Datei AS68LC.BIN erzeugt.

- Sofern es noch nicht dort steht, kopieren Sie AS68LC.BIN in das Verzeichnis, in dem Lattice C arbeitet.

- Editieren Sie die Datei C.LNK. Schreiben Sie unter die Zeile "INPUT *" folgendes:

```
INPUT AS68LC.BIN
```

  Dadurch wird der Linker dazu veranlaßt, dieses Modul mit zu linken.

- Compilieren Sie nun AUFRUF1.C und binden Sie es. Damit wird ein lauffähiges Programm AUFRUF1.TOS oder AUFRUF1.PRG erzeugt, je nachdem, welche Linker-Optionen eingestellt sind.

Im Prinzip bietet der Lattice C Compiler auch die Möglichkeit, Objektcode im LINK68-Format zu erzeugen. Man braucht dazu nur die Option "-t" anzugeben. Und es funktioniert tatsächlich: Sie können dann Objektcode von AS68 dazu linken. Leider gibt es dabei ein Problem: Die C-Bibliotheken von Lattice C liegen nur im Format des (von Lattice C verwendeten) GST-Linkers vor, und ohne die C-Standardbibliothek wird man nicht weit kommen. Deshalb bringt diese Option nicht viel, es sei denn, Sie finden eine Möglichkeit, die Bibliotheken ins LINK68-Format zu konvertieren.

Megamax C

Die Prozedurkonvention gleicht der des Compilers aus dem ATARI-Entwicklungssystem aufs Haar. Leider benutzt dieser Compiler einen Linker eigener Bauart (der Extender ".O" hat nichts zu sagen), so daß sie ohne weitere Tricks nur vom Inline-Assembler Gebrauch machen können. Leider läßt dieser nur PC-relativen Code zu und ist auch sonst kein vollwertiger Assembler, läßt sich aber für kleine Prozeduren mitunter gut gebrauchen.

Sollten Sie größere Dinge vorhaben, so wird Ihnen im zweiten Teil dieses Anhangs eine Möglichkeit vorgestellt.

CCD ST Pascal plus (Version 1.10)

Die Prozedurkonvention dieses Pascal-Compilers gleicht der C-Konvention bis auf ein Detail: Die Parameter werden anders herum auf dem Stack abgelegt, also diesmal in der Reihenfolge, wie man sie beim Aufruf hinschreibt, so daß der rechts stehende Parameter beim Aufruf unten auf dem Stack liegt. Ansonsten gilt, daß ein INTEGER 2 Bytes auf dem Stack belegt, ein LONG_INTEGER 4 Bytes. Eine Eigenheit des Compilers gilt es noch zu beachten: Alle von Pascal erzeugten Symbole werden in Großschrift verwandelt. Da Assembler und Linker hingegen zwischen Klein- und Großbuchstaben unterscheiden, kommt es zu Problemen, wenn das Pascal-Programm eine Prozedur "LINE" aufrufen will, jedoch nur "line" vorfindet. Deshalb müssen alle vom Assemblercode exportierten Symbole ebenfalls groß geschrieben werden. Das ist auch schon alles, was es an Abweichungen gibt. Nun das entsprechend angepaßte Listing:

```
*************************************************************
* AS68PAS.S                                                 *
* Beispielprogramm zum Einbinden von Assembler in           *
* Hochsprachen                                              *
* für ATARI-Assembler mit CCD ST Pascal plus                *
* ermöglicht Zugriff auf Line-A-Routinen line und           *
* put_pixel (plot)                                          *
*************************************************************
* zuerst die von außen erreichbaren Symbole
          XDEF      LINE
          XDEF      PLOT
*
* relative Adressen der benutzten Line-A Variablen
intin     EQU       8
ptsin     EQU       12
fg_bp_1   EQU       24
fg_bp_2   EQU       26
fg_bp_3   EQU       28
fg_bp_4   EQU       30
wrt_mod   EQU       36
x1        EQU       38
y1        EQU       40
x2        EQU       42
y2        EQU       44
*
* Line-A-Opcodes
A_INIT    EQU       $A000
A_PUT_PIX EQU       $A001
A_LINE    EQU       $A003
*
* Aufruf: line(x1,y1,x2,y2)
LINE      link      a6,#0               * SP nach a6
          DC.W      A_INIT              * Variablenadressen
                                        * holen
```

```
            move        14(a6),x1(a0)       * Parameter in die
            move        12(a6),y1(a0)       * Line-A-Variablen
            move        10(a6),x2(a0)       * schreiben
            move        8(a6),y2(a0)        *
            move        #1,fg_bp_1(a0)      * Farbe 1
            clr         fg_bp_2(a0)         *
            clr         fg_bp_3(a0)         *
            clr         fg_bp_4(a0)         *
            move        #2,wrt_mod(a0)      * Schreibmodus: XOR
            DC.W        A_LINE              * Line-A-Aufruf
            unlk        A6                  * SP wiederherstellen
            rts                             *
*
* Aufruf: plot(x,y,color)                  *
PLOT        link        a6,#0               *
            DC.W        A_INIT              * Variablenadressen
                                            * holen
            move.l      ptsin(a0),a1        * Adresse des ptsin-
                                            * Feldes holen
            move        12(a6),(a1)         * Koordinaten ins
            move        10(a6),2(a1)        * ptsin-Feld schreiben
            move.l      intin(a0),a1        * Adresse des intin-
                                            * Feldes holen
            move        8(a6),(a1)          * Farbe ins intin-Feld
                                            * schreiben
            DC.W        A_PUT_PIX           * Line-A-Aufruf
            unlk        a6                  *
            rts                             *
*
            END
```

ST Pascal plus verfügt über einen zu LINK68 kompatiblen Linker. Kopieren Sie deshalb die von AS68 erzeugte Datei AS68PAS.O in das Arbeitsverzeichnis des Pascal-Compilers, und geben Sie beim Formular der Linkeroptionen als zusätzlich zu linkende Datei eben jenes AS68PAS.O an. Nur noch Compiler und Linker durchlaufen lassen, und schon haben Sie die ausführbare Datei.

Es kann übrigens durchaus nützlich sein, den Pascal-Linker an Stelle von LINK68 zu benutzen, denn ersterer ist kürzer, schneller und benötigt kein RELMOD. Parameter werden ihm genauso übergeben wie LINK68. Allerdings beinhaltet er nicht alle Optionen von LINK68, doch meistens soll ein Linker ja ohnehin nichts anderes tun als eben linken.

2. Einbinden ohne Linker

Bei den meisten Kombinationen Compiler/Assembler gibt es keinen gemeinsamen Linker, bei Interpretern natürlich erst recht nicht. Nun, es gibt zwei Methoden, den Assemblercode der gewünschten Programmiersprache zugänglich

zu machen. Beide verlangen, daß der Code assembliert und durch den Linker geschickt wird, so daß er den Status eines ausführbaren Programms hat. Wenn es sich um Routinensammlungen handelt, wird der Code natürlich nicht tatsächlich ausführbar sein, aber es kommt ja nur darauf an, daß es für das Betriebssystem so aussieht. Noch eines ist wichtig: Wenn Sie mehrere Routinen einbinden wollen, ist es unerläßlich, ihre Startadressen relativ zum Programmanfang zu kennen, denn in irgendeiner Form müssen Sie die Aufgaben des Linkers nun von Hand ausführen. Benutzen Sie dazu am besten ein Assemblerlisting, bei dem eine Symboltabelle vorhanden sein sollte. Dort finden Sie die Adressen relativ zum Programmanfang, der auf die logische Adresse Null festgelegt ist. Diese Informationen finden Sie wahlweise auch im Linkerprotokoll. Problematisch ist dabei folgendes: Die relativen Adressen verschieben sich, sobald Sie auch nur einen einzigen Befehl zu ihrer Routinensammlung hinzufügen. Da diese Adressen im aufrufenden Programm verwendet werden, müßten Sie bei jeder Erweiterung ihrer Assemblerroutinen dieses verändern. Es geht natürlich auch anders: Zu diesem Zweck gibt es bei den meisten Assemblern die Direktive ORG oder etwas Vergleichbares. Als Parameter bekommt ORG einen absoluten Wert, der in den "location counter" des Assemblers geladen wird, das heißt, der nächste Befehl wird an die dort angegebene logische Adresse assembliert. Dies gibt Ihnen die Möglichkeit, hinter jeder Routine genügend Platz für Erweiterungen zu lassen. Wenn Sie nun Befehle hinzufügen, ändern sich die logischen Adressen der Einsprungpunkte nicht; die Lücken zwischen den Routinen werden nur kleiner. So könnte man beispielsweise vor die Routine "plot" die Anweisung

```
ORG  100
```

setzen. So wissen Sie, daß diese Routine an der logischen Adresse 100 beginnt, und es sind noch genügend Bytes für Erweiterungen der "line"-Routine frei. Mit ORG sind sinnvollerweise nur Verschiebungen des "location counters" nach oben erlaubt; wenn also die "line"-Routine mehr als 100 Bytes lang wird, gibt es eine Fehlermeldung. Die erste Möglichkeit besteht nun darin, das Assemblerprogramm von vornherein PC-relativ zu schreiben. Um keine Begriffsverwirrung aufkommen zu lassen: Ein Programm ist PC-relativ, wenn jede Variable und jedes Label PC-relativ adressiert wird. PC-relativer Code braucht also nicht reloziert zu werden. Ein relozierbares Programm hingegen darf mit absoluten Adressierungsarten auf seine Variablen und Label zugreifen; die Programmdatei ist jedoch mit einer Relozierungstabelle versehen, die das Programm trotzdem an jeder Stelle im Speicher lauffähig macht. Wenn Sie also ein Programm PC-relativ schreiben, brauchen Sie den Code nur an eine beliebige Stelle im Speicher zu laden und können ihn dann aufrufen. Nachteil dieser Methode ist, daß bei den meisten Maschinensprachebefehlen nur die Quelle PC-relativ adressiert werden kann; wollen Sie auf den Zieloperanden

PC-relativ zugreifen, so hilft oft nur ein vorheriges Ermitteln der absoluten Adresse mittels LEA, etwa in der Form

```
LEA label(pc)
```

Natürlich ist dies etwas umständlich und verlangsamt das Programm auch geringfügig. Deshalb zur zweiten Methode, die diese Nachteile nicht aufweist:

Sehen wir uns die GEMDOS-Prozedur PEXEC ($4B) einmal etwas genauer an. Normalerweise übergibt man ihr den Namen einer Programmdatei zusammen mit einigen anderen Parametern, wodurch die Programmdatei geladen, reloziert und gestartet wird. Erst wenn das so gestartete Programm terminiert, kehrt der PEXEC-Aufruf zurück, und das aufrufende Programm macht dort weiter, wo es aufgehört hat. Rückgabewert von PEXEC ist der vom aufgerufenen Programm beim Aufruf von PTERM (GEMDOS $4C) angegebene Wert; wurde das Programm mit TERM (GEMDOS Nummer 0) beendet, so wird 0 zurückgegeben. Tief versteckt im ATARI findet sich jedoch eine interessante Option: Wenn der erste Parameter, "load" genannt, den Wert 3 hat, wird das Programm nicht gestartet, sondern nur geladen und reloziert. Rückgabewert von PEXEC ist dann die Adresse der Basepage des geladenen Programms oder eine negative Fehlermeldung, falls etwas schiefgegangen ist. Zu dieser Adresse brauchen Sie nun nur noch 256 hinzuzuzählen, um auf die Adresse des ersten Bytes des Programmcodes zu kommen – in unserem Beispiel ist das der Einsprungpunkt in die "line"-Routine. Um auf die Adressen der anderen Routinen zu kommen, zählt man deren logische Adressen zur Anfangsadresse des Codes hinzu.

Übrigens brauchen Sie sich keine Sorgen darum zu machen, daß der vom Code belegte Speicherplatz überschrieben werden kann, denn dieser Bereich wird vom Betriebssystem als belegt vermerkt und kann vorerst nicht zu anderen Zwecken vergeben werden. Diese Methode erfordert also von der benutzten Programmiersprache zwei Fähigkeiten:

- Zugriff auf die GEMDOS-Aufrufe, speziell auf PEXEC.

- Die Möglichkeit, zu einer absoluten Adresse einen Unterprogrammaufruf durchzuführen.

Leider bieten nicht alle Programmiersprachen diese beiden Möglichkeiten. Sehen wir uns nun Anwendungen der beiden Methoden in den wichtigsten Programmiersprachen an:

Sämtliche C-Compiler

Die Sprache C erfüllt die beiden oben genannten Voraussetzungen, auch wenn für die zweite einige Pointerakrobatik notwendig ist. Der Assemblercode bleibt der gleiche wie der weiter oben verwendete, also die Datei AS68LC.S, falls Sie Lattice C verwenden, in allen anderen Fällen AS68C.S. Assemblieren und linken Sie diese Dateien mit dem Assembler ihrer Wahl. (Es macht natürlich auch nichts, wenn Ihr Assembler ohne Linker arbeitet, es geht nur darum, eine ausführbare Datei zu erzeugen.) Damit niemand auf die Idee kommt, eine solche Routinensammlung ausführen zu wollen, sollten nicht die Endungen PRG oder TOS verwendet werden, sondern etwas Neutrales wie OVR (für "overlay"). Benennen Sie deshalb die ausführbare Datei in AS68C.OVR um (auch dann, wenn sie vorher AS68LC.PRG hieß). Der Aufruf sieht für alle C-Compiler gleich aus:

```
/* AUFRUF2.C
   zweiter Beispielaufruf von AS68C aus C heraus
   Laden des Codes mittels PEXEC
   für jeden beliebigen Compiler!                */

#define pexec(a,b,c,d) gemdos(0x4b,a,b,c,d)

long gemdos();  /* gemdos() ist externer Funktionsaufruf */

main()
{ int i;          /* Zählvariable              */
  long prog_adr;                          /* Rückgabewert von exec    */
  int (*line)();                          /* Pointer auf Funktion     */
  int (*plot)();                          /* Pointer auf Funktion     */
  prog_adr=pexec(3,"as68c.ovr","","");    /* nur laden                */
  if (prog_adr<0)                         /* Fehler!                  */
  { printf("grausamer Fehler!\nas68c.ovr läßt sich nicht laden!\n");
    gemdos(8);                            /* auf Taste warten         */
    exit();                               /* Programm beenden         */
  }
  line=(int(*)()) (prog_adr+256);         /* Anfangsadresse des Codes */
  plot=(int(*)()) (prog_adr+256+60);      /* plus Offset für 'plot'   */
  for(i=0;i<200;i++)                      /* ein paar Punkte setzen   */
  { (*plot)(i,i,1);
    (*plot)(199-i,i,1);
  }
  gemdos(8);                              /* auf Taste warten         */
}
```

Zunächst wird dem Compiler mitgeteilt, daß die Prozedur gemdos() ein Langwort zurückgibt ("long gemdos()" ist gleichbedeutend mit "extern long gemdos()"). Innerhalb von main() erfolgen zunächst die Variablendeklarationen, wobei zwei Pointer auf Funktionen "line" und "plot" vereinbart werden. Nun

folgt der PEXEC-Aufruf mit dem Flag load=3. Ist der Rückgabewert negativ, dann wird das Programm mit einer Fehlermeldung beendet – zur Vermeidung von Bomben. Was folgt, ist ein Stück interessanter Pointerakrobatik: Die Adresse der Line-Routine errechnet sich aus der Adresse der Basepage plus 256. Da dieser Wert jedoch immer noch vom Typ "long" ist, muß ein Cast angewandt werden: (int(*)()) verwandelt das Ergebnis in den Typ "Pointer auf Funktion, die int zurückgibt" (ja, das funktioniert tatsächlich!). Eigentlich gibt unsere Prozedur "line" ja gar keinen Wert zurück, doch eine entsprechende Konstruktion mit "void" anstelle von "int" wird von einigen Compilern nicht verdaut. Man braucht ja den Rückgabewert nicht auszuwerten.

Die Adresse der Routine "plot" liegt noch 60 Bytes höher, da das Label "plot" die logische Adresse 60 hat, wie aus dem Assemblerlisting hervorging. Nun endlich können die Assemblerroutinen aufgerufen werden, indem die Pointer auf Funktion dereferenziert und mit den Parametern versehen werden. Hier werden beispielhaft einige Punkte mit "plot" gesetzt.

GFA-BASIC (Version 2.0)

Auch GFA-BASIC erfüllt beide oben genannten Bedingungen – sogar auf recht komfortable Weise. Es gestattet den Aufruf nach der üblichen C-Konvention. Deshalb kann auch hier der Code aus AS68C.S verwendet werden. Erzeugen Sie daraus eine Programmdatei, und nennen Sie sie in AS68C.OVR um. Beim GFA-BASIC gilt es noch eine Besonderheit zu beachten: Da der Interpreter normalerweise den gesamten verfügbaren Speicher bis auf einige Kilobytes für den Stack belegt, ist es notwendig, einen Speicherbereich freizugeben, in den das Assemblerprogramm geladen werden kann. Mit der Anweisung

```
Reserve Fre(0)-10000
```

werden 10000 Bytes am oberen Ende des Speichers freigegeben.

Der vollständige Aufruf sieht so aus:

```
' GFA_AUFR.BAS
' Beispielaufruf von AS68_C.S aus GFA-BASIC 2.0
' Laden der Programmdatei mittels EXEC (GEMDOS $4B)
'
Reserve Fre(0)-10000
Adr%=Exec(3,"as68c.ovr","","")
If Adr%<0
  Print "Fataler Fehler!"
  Print "Datei as68c.ovr läßt sich nicht öffnen!"
```

```
  End
Endif
Let Line%=Adr%+256
Let Plot%=Adr%+256+60
For I%=0 To 320
  Dummy=C:Line%(I%,0,320-I%,199)
Next I%
```

GFA-BASIC 2.0 stellt PEXEC als Funktion mit dem Namen "Exec" zur Verfügung. Nach dem Aufruf wird überprüft, ob der Rückgabewert negativ ist; wenn ja, wird das Programm mit einer Fehlermeldung abgebrochen. Andernfalls werden die Adressen der Routinen Line und Plot berechnet. Nun kann der Aufruf erfolgen: Die Funktion "C:" führt einen Aufruf nach der C-Konvention durch. Alle Parameter werden als Worte übergeben, es sei denn, man stellt ihnen ein "L:" für Langwort voran. GFA-BASIC verlangt, daß der Rückgabewert einer Variablen zugewiesen wird, deshalb wird hier die Variable "Dummy" verwendet, die natürlich nicht ausgewertet wird.

Nun zur zweiten Möglichkeit: Da bei unseren Routinen keine Zugriffe auf Variablen erfolgen, ist der Code ohnehin PC-relativ, braucht also nicht reloziert zu werden. Nun macht man sich die Tatsache zu nutze, daß eine ausführbare Datei einen Header von 28 Bytes Länge hat, auf den sofort das Textsegment folgt. Der Zugriff auf PC-relative Routinen läuft also so ab: Die gesamte Datei wird en bloc in einen reservierten Speicherbereich geladen, etwa in eine Zeichenkette. Die Anfangsadresse des Textsegments errechnet sich aus der Adresse dieses Bereichs plus 28; die einzelnen Routinen erreicht man, indem man wiederum zu jenem Wert ihre logischen Adressen addiert. Ein Aufruf sieht also folgendermaßen aus (es wird wieder die Datei AS68C.OVR benötigt):

```
' GFA_AUF2.BAS
' Beispielaufruf von AS68_C.S aus GFA-BASIC 2.0
' Binäres Laden des PC-relativen Programmcodes mittels BLOAD
'
Code$=Space$(1000)
Start%=Varptr(Code$)
Bload "as68c.ovr",Start%
Let Line%=Start%+28
Let Plot%=Start%+28+60
For I%=0 To 320
  Dummy=C:Line%(I%,0,320-I%,199)
Next I%
```

In diesem Beispiel wird eine Zeichenkette der Länge 1000 benutzt, um den Programmcode aufzunehmen. Es ist wichtig, daß die Zeichenkette vorher tatsächlich lang genug ist, da der Speicherplatz für Zeichenketten in GFA-BASIC dynamisch verteilt wird.

ST BASIC (Länge 138 944 Bytes)

Leider scheidet beim ST BASIC die komfortablere Möglichkeit des Ladens mit PEXEC aus, da es keinen Weg gibt, die GEMDOS-Routinen zu erreichen. ST BASIC verwendet auch eine eigene Aufrufkonvention für Assemblerroutinen. Diese ist im Handbuch leider nicht vollständig beschrieben; als Ausgleich dafür sind aber die wenigen Informationen, die darüber zu finden sind, schlichtweg falsch. Die tatsächliche Konvention beim Aufruf mit CALL sieht so aus: Auf dem Stack wird als erster Wert über der Rückkehradresse ein Wort abgelegt, das die Anzahl der Parameter angibt, die der CALL-Funktion mitgegeben worden sind. Das folgende Langwort gibt die Adresse eines Arrays an, indem die Parameter abgelegt sind. Dort belegt jeder Parameter 4 Bytes (nicht 8, ATARI!). Alle Parameter werden als Langworte abgelegt; für uns sind jedoch wieder nur die niederwertigen Worte interessant. Eines wird im Handbuch leider verschwiegen: Die Prozedur des BASIC-Interpreters, die den Aufruf mittels CALL durchführt, arbeitet mit dem Frame-Pointer A6 (Bekanntlich wurde ATARI-BASIC in einer Compilersprache geschrieben, was einen der Gründe für die eher gemächliche Geschwindigkeit darstellt). Deshalb darf dieses Register in unserer Prozedur nicht verändert werden, da sonst wieder einmal Bomben angesagt sind. Aus mysteriösen Gründen schien der Aufruf der Routine A_INIT plötzlich genau dieses Register zu verändern, weshalb im folgenden Listing am Anfang der Routinen der Inhalt von A6 gesichert wird.

```
*************************************************************
* AS68BAS.S                                                 *
* Beispielprogramm zum Einbinden von Assembler in           *
* ATARI-BASIC                                               *
* für Metacomco-Assembler oder ATARI-Assembler              *
* ermöglicht Zugriff auf Line-A-Routinen line und           *
* put_pixel (plot)                                          *
*************************************************************
*
* relative Adressen der benutzten Line-A Variablen
intin     EQU       8
ptsin     EQU       12
fg_bp_1   EQU       24
fg_bp_2   EQU       26
fg_bp_3   EQU       28
fg_bp_4   EQU       30
wrt_mod   EQU       36
x1        EQU       38
y1        EQU       40
x2        EQU       42
y2        EQU       44
*
* Line-A-Opcodes
A_INIT    EQU       $A000
A_PUT_PIX EQU       $A001
```

```
A_LINE     EQU       $A003
*
* Aufruf: line(x1,y1,x2,y2)
line       move.l    a6,-(sp)          * Register a6 sichern
           DC.W      A_INIT            * Variablenadressen
                                       * holen
           move.l    10(sp),a6         * Zeiger auf
                                       * Parametertabelle
           move      2(a6),x1(a0)      * Parameter in die
           move      6(a6),y1(a0)      * Line-A-Variablen
           move      10(a6),x2(a0)     * schreiben
           move      14(a6),y2(a0)     *
           move      #1,fg_bp_1(a0)    * Farbe 1
           clr       fg_bp_2(a0)       *
           clr       fg_bp_3(a0)       *
           clr       fg_bp_4(a0)       *
           move      #2,wrt_mod(a0)    * Schreibmodus: XOR
           DC.W      A_LINE            * Line-A-Aufruf
           move.l    (sp)+,a6          * a6 wiederherstellen
           rts                         *
*
* Aufruf: plot(x,y,color)
plot       move.l    a6,-(sp)          * A6 sichern
           DC.W      A_INIT            * Variablenadressen
                                       * holen
           move.l    10(sp),a6         * Adresse der
                                       * Parametertabelle
           move.l    ptsin(a0),a1      * Adresse des ptsin-
                                       * Feldes holen
           move      2(a6),(a1)        * Koordinaten ins
           move      6(a6),2(a1)       * ptsin-Feld schreiben
           move.l    intin(a0),a1      * Adresse des intin-
                                       * Feldes holen
           move      10(a6),(a1)       * Farbe ins intin-Feld
                                       * schreiben
           DC.W      A_PUT_PIX         * Line-A-Aufruf
           move.l    (sp)+,a6          *
           rts                         *
*
           END
```

Der Aufruf erfolgt ähnlich wie bei obigem GFA-BASIC-Programm: Eine initialisierte Zeichenkette stellt den notwendigen Platz zur Verfügung, in den die gesamte ausführbare Datei mittels BLOAD geladen wird. Nun brauchen nur noch die Einsprungadressen berechnet zu werden. Diesmal ist die logische Adresse der Plot-Routine 62, da der Code der Line-Routine durch die Änderungen 2 Bytes länger geworden ist.

Falls die Programmiersprache Ihrer Wahl sich nicht unter den hier aufgeführten befinden sollte, so hoffe ich dennoch, daß die vorgeführten Methoden und Prozedurkonventionen Ihnen weiterhelfen. Die Methoden des Einbindens soll-

ten sich recht einfach auf andere Programmiersprachen oder neue Versionen übertragen lassen. Problematischer ist es schon mit der Prozedurkonvention. Leider dokumentieren nicht alle Softwarefirmen ihre Compiler oder Interpreter in diesem Punkt so ausführlich, wie es wünschenswert ist. Doch es gibt einen Weg, mit Hilfe eines Debuggers Ihrer Programmiersprache auf die Schliche zu kommen:

- Schreiben Sie eine Testprozedur in Asssembler, die nur aus einem Befehl zu bestehen braucht, aber etwas illegales tut, damit eine Exception auftritt, etwa folgendes:

```
test:     MOVE   1,D0
```

- Schreiben Sie ein Programm in Ihrer Programmiersprache, das einen Testaufruf der Assemblerprozedur ausführt, etwa in der Art

```
test(1,2,3,4)
```

  Wenn Sie einen Compiler verwenden und dieser Code erzeugt, der Exceptions abfängt, so sollten Sie dies abschalten. Ist das nicht möglich, so wird die Prozedurkonvention dieses Compilers wohl ein ewiges Geheimnis bleiben.

- Laden Sie nun zunächst den Debugger. Von diesem aus laden und starten Sie nun das lauffähige Programm, den Interpreter, das Laufzeitsystem oder was immer Sie zuerst laden, wenn Sie Ihre Programmiersprache benutzen wollen. Starten Sie Ihr Programm. Sollte es sich bei dem Programm, das Sie vom Debugger aus starten, um eine GEM-Anwendung handeln, so rufen Sie auch den Debugger als GEM-Anwendung auf. Dadurch gerät der Bildschirmaufbau zwar etwas durcheinander, aber das Ganze funktioniert. Wollten Sie etwa von SID aus ST-BASIC starten, so nennen Sie SID.TTP in SID.PRG um und starten den Debugger. Dann geben Sie folgendes ein:

```
e basic.prg
g
```

  Damit wird der Interpreter gestartet.

- Sobald der Testaufruf erfolgt, gibt es einen Adreßfehler. Nun kommt die große Stunde des Debuggers: Er meldet sich mit der Ausgabe der Registerwerte und der Adresse, an der die Exception aufgetreten ist. Jetzt können Sie in aller Ruhe untersuchen, wo Ihre Parameter gelandet sind. Sehen Sie zuerst in den Registern und auf dem Stack nach. Sollten Sie da noch nicht fündig werden, so untersuchen Sie, ob man Langworte auf dem Stack oder Registerinhalte als Zeiger auf Parametertabellen interpretieren kann. So sollten Sie eigentlich jede Prozedurkonvention knacken können.

Anhang E

Tips zur Fehlersuche

Nun ist es geschehen. Das Programm benimmt sich aus äußerst mysteriösen Gründen völlig anders, als es soll. Sie sollten von diesem Anhang keine Wunderdinge erwarten, denn die Anzahl der möglichen Fehler in einem Programm ist unüberschaubar. Hier wird nur auf die Fehler der Befehlsebene eingegangen – für die Logik Ihrer Algorithmen sind Sie natürlich selbst verantwortlich. Die Erfahrung lehrt aber, daß die meisten Programmfehler in Assembler auf der falschen Verwendung einzelner Maschinensprachebefehle beruhen – so etwas passiert dem Profi ebenso wie dem Anfänger.

Im folgenden werden wir die erfahrungsgemäß häufigsten Fehler aufführen, in der Hoffnung, Ihren Blick für die bekanntesten Patzer zu schärfen und Ihnen so vielleicht die eine oder andere frustrierende Stunde der Fehlersuche zu ersparen. Einige Vorschläge mögen vielleicht recht trivial erscheinen, doch meist sind es Trivialitäten, die einen recht lange aufhalten. Die meisten der hier aufgeführten Fehler sind mir schon einmal selbst zugestoßen.

Zunächst noch ein Wort zur Fehlersuche allgemein: Wenn Sie überhaupt keine Vorstellung davon haben, was die Ursache eines Absturzes oder eines merkwürdigen Verhaltens sein könnte, so geht es zunächst einmal darum, den Fehler einzukreisen. Wie in Kapitel 2 beschrieben, kann zu diesem Zweck ein Debugger von großem Nutzen sein, um etwa die Abarbeitung eines Programms zu verfolgen oder die genauen Umstände eines Absturzes zu untersuchen.

In manchen Fällen kann es hingegen nützlicher sein, sich nur vom Programm an bestimmten Stellen die Werte der einen oder anderen Variablen ausgeben zu lassen. Zu diesem Zweck könnten Sie etwa die Routinen zur Ausgabe von dezimalen und hexadezimalen Zahlen aus Kapitel 5 als Makros implementieren. Wichtig ist dabei, daß diese Routinen nichts am Zustand des Prozessors ändern; legen Sie deshalb am besten vor der Ausführung alle Prozessorregister auf dem Stack ab, um sie nach der Abarbeitung der Routine wiederherzustellen. Wenn es nötig sein sollte, tun Sie das auch mit dem Userbyte.

Nun zu den eigentlichen Fehlern. Betrachten wir zunächst einmal die verschiedenen Arten von Exceptions und Ihre Ursachen.

Busfehler – 2 Bomben

Wenn Sie nicht gerade vergessen haben, für einen Zugriff auf die Systemvariablen oder die Hardwareregister den Supervisormodus einzuschalten, so bedeutet die Exception, daß das Programm auf einen Speicherbereich zugreift, auf den es eigentlich gar nicht zugreifen will. Einige mögliche Ursachen:

- Bei einer Adreßberechnung wurde versehentlich eine Operation nur in Wort- statt in Langwortbreite durchgeführt.

- Beim Ansprechen des Bildschirms wurde eine etwas zu hohe Adresse verwendet. Bedenken Sie, daß meist direkt hinter dem Bildschirmspeicher das physikalische RAM endet.

- In einem Unterprogramm wird der Stackpointer verändert, wodurch nach der nächsten Ausführung eines RTS der Programmzähler plötzlich ins Nirwana zeigt.

Adreßfehler – 3 Bomben

Dies ist das Anzeichen dafür, daß in einer Wort- oder Langwortoperation auf eine ungerade Adresse zugegriffen wird. Mögliche Ursachen:

- Aus irgendeinem Grund hat der Assembler Wort- oder Langwortvariablen an einer ungeraden Adresse abgelegt. Verwenden Sie nach der Angabe von Byte-Daten sicherheitshalber die EVEN-Direktive.

- Bei Adreßberechnungen wurde nicht beachtet, daß Indizes in Felder von Worten mit 2 multipliziert werden müssen, bei Feldern von Langworten mit 4.

- Es kann auch ein Folgefehler sein, wenn vorher an irgendeiner Stelle auf einem Stack – sei es ein eigener oder der Systemstack – Bytes abgelegt werden und später Worte oder Langworte darauf abgelegt werden.

Illegaler Befehl – 4 Bomben

Mögliche Ursachen:

- Durch eine falsche Zieladresse bei einem Sprungbefehl landete der Prozessor an einer falschen Stelle im Speicher.

- Durch eine falsche Adreßberechnung oder die Angabe eines falschen Labels schreibt das Programm – möglicherweise an einer völlig anderen Stelle – in seinen eigenen Code. Das fällt natürlich erst auf, wenn der überschriebene Code ausgeführt wird.

- In einem Unterprogramm wird der Stackpointer verändert, wodurch das RTS den Programmzähler auf einen völlig falschen Wert setzt.

Andere Fehlerquellen

Bei der Division durch null geschieht normalerweise nichts weiter; die CHK- und TRAPV-Exception dürften in einem in Assembler geschriebenen Programm wohl kaum auftauchen, da man dafür die Befehle CHK bzw. TRAPV einsetzen muß. Die Privilegverletzung (8 Bomben) kann gelegentlich auftreten. Ihre Ursache ist klar: Vor der Benutzung eines privilegierten Befehls wurde nicht in den Supervisormodus geschaltet.

Die Ursache einer Exception der oben beschriebenen Arten ist im allgemeinen recht leicht einzukreisen, zumal ein Debugger Auskunft geben kann über die Speicherstelle, an der die Exception aufgetreten ist und den Zustand der Prozessorregister. Schwieriger zu identifizieren sind dagegen die folgenden allgemeinen Fehler, die zumindest nicht sofort zu einer Exception, sondern eher zu einem merkwürdigen Verhalten des Programms führen:

- Der OR-Befehl zum Setzen von Bits wurde mit AND verwechselt, etwa nach folgendem Gedankengang: Ich will die im Zielregister gesetzten Bits erhalten und die Bits zusätzlich auf 1 setzen, also benutze ich den AND-Befehl. Das ist natürlich falsch; hier muß der OR-Befehl verwendet werden.

- Es kann leicht vergessen werden, daß bei Adreßberechnungen Werte, die kürzer sind als ein Langwort, als vorzeichenbehaftet betrachtet werden. Nehmen wir etwa folgenden Befehl:

```
MOVE        D0,0(A0,D1)
```

  Damit kann man kein 64 KByte langes Feld verwalten! Wenn etwa D1 den Wert 40000 hat, so greift man damit tatsächlich auf Speicherplatz –25536(A0) zu! Entsprechendes gilt auch für die Adressierungsart "Adreßregister indirekt mit Displacement":

```
MOVE        D0,40000(A0)
```

Auch dieser Befehl greift tatsächlich auf –25536(A0) zu.

– Ein verbreiteter Fehler besteht darin, das Doppelkreuz vor einem unmittelbaren Operanden zu vergessen. Da der Prozessor dann natürlich auf die entsprechende Adresse zugreift und unmittelbare Operanden größtenteils recht klein sind, ist das Resultat meistens ein Busfehler, es sei denn, das Programm läuft gerade im Supervisormodus. Achten Sie also auch auf vergessene Doppelkreuze, wenn sich ein Programm merkwürdig benimmt.

Anhang F

Befehlstabelle mit Adressierungsarten und Ausführungszeiten

Zur folgenden Tabelle:

Spalte "Breite":

B	Bytelänge, 8 Bit
W	Wortlänge, 16 Bit
L	Langwortlänge, 32 Bit

Spalte "Adressierungsart"

q	Quelle
z	Ziel

Bei Befehlen mit 2 Operanden wird in dieser Spalte die Adressierungsart des einen festgelegt; die folgenden 12 Spalten geben die verschiedenen Adressierungsarten des anderen Operanden an.

i	Indexregister
Abs. W	absolut kurz
Abs. L	absolut lang
unm.	unmittelbar
unm.3	bei ADDQ/SUBQ unmittelbar 1–8

Ausführungszeiten:

Alle Ausführungszeiten sind in Taktzyklen angegeben;

1 Taktzyklus entspricht 125 Nanosekunden

<	Maximalwert
n	bei Verschiebefehlen die Anzahl der Stellen, um die verschoben wird, bei MOVEM die Anzahl der Register.

Mnemonic	Breite	Adressierungsart		Dn	An	(An)	(An)+	
ABCD	B	q=Dn	z	6				
		q=(An)	z					
ADD	B/W	q=Dn	z		ADDA	12	12	
		z=Dn	q	4	4	8	8	
	L	q=Dn	z		ADDA	20	20	
		z=Dn	q	8	8	14	14	
ADDA	W	z=An	q	8	8	12	12	
	L	z=An	q	8	8	14	14	
ADDI	B/W	q=unm.	z	8	ADDA	16	16	
	L	q=unm.	z	16	ADDA	28	28	
ADDQ	B/W	q=unm.3	z	4	4	12	12	
	L	q=unm.3	z	8	8	20	20	
ADDX	B/W	q=Dn	z	4				
		q=(An)	z			18		
	L	q=Dn	z	8				
		q=(An)	z			30		
AND	B/W	q=Dn	z			12	12	
		z=Dn	q	4		8	8	
	L	q=Dn	z			20	20	
		z=Dn	q	8		14	14	
ANDI	B/W	q=unm.	z	8		16	16	
	L	q=unm.	z	16		28	28	
ASL,	B/W	q=Dn	z	6+2n				
ASR		q=#1–8	z	6+2n				
	L	q=Dn	z	8+2n				
		q=#1–8	z	8+2n				
Speicher	W		z			12	12	
Bcc	B(S)	entfällt		verzweigt 10				
	L			verzweigt nicht 8				
	W	entfällt		verzweigt 10				
				verzweigt nicht 14				
BCHG,	B	q=Dn	z			12	12	
BCLR,		q=unm.	z			16	16	
BSET	L	q=Dn	z	<10				
		q=unm.	z	<12				
BTST	B	q=Dn	z			8	8	
		q=unm.	z			12	12	
	L	q=Dn	z	6				
		q=unm.	z	10				
BSR	B(S)	entfällt				20		
	W	entfällt				20		
CHK	W	z=Dn trap 9		<40		<44	<44	
		kein trap 9		8		12	12	
CLR	B/W		z	4		12	12	

	-(An)	d(An)	d(An)	Abs.W	Abs.L	D(PC)	D(P)	q=unm. z=SR/ CCR	Bedin-gungs-codes XNZVC
	18								*u*u*
	14 10 22 16	16 12 24 18	18 14 26 20	16 12 24 18	20 16 28 22	12 18	14 20	8 14	*****
	14 16	16 18	18 20	16 18	20 22	16 18	18 20	12 14	-----
	18 30	20 32	22 34	20 32	24 36				*****
	14 22	16 24	18 26	16 24	20 28				*****
									*****
	14 10 22 16	16 12 24 18	18 14 26 20	16 12 24 18	20 16 28 22	12 18	14 20	8 14	-**00
	18 30	20 32	22 24	20 32	24 36			20	-**00
	14	16	18	16	20				*****
									-----
	14 18	16 20	18 22	17 20	20 24				--*--
	10 14	12 16	14 18	12 16	16 20	12 16	14 18		--*--

	<46 14	<48 16	<50 18	<48 16	<52 20	<48 16	<50 18	<44 12	-*uuu
	14	16	18	16	20				-0100

Mnemonic	Breite	Adressierungsart		Dn	An	(An)	(An)+
	L		z	6		20	20
CMP	B/W	z=Dn	q	4	4	8	8
	L	z=Dn	q	6	6	14	14
CMPA	W	z=An	q	6	6	10	10
	L	z=An	q	6	6	14	14
CMPI	B/W	q=unm.	z	8	CMPA	12	12
	L	q=unm.	z	14	CMPA	20	20
CMPM	B/W	q=(An)+	z				12
	L	q=(An)+	z				20
	W	z=Dn		10 wenn cc=false, Zähler ≠–1 12 wenn cc=true, Zähler ≠–1 14 wenn cc=false, Zähler =–1			
DIVS	W	z=Dn	q	<158		<162	<162
DIVU	W	z=Dn	q	<140		<144	<144
EOR	B/W	q=Dn	z	4		12	12
	L	q=Dn	z	8		20	20
EORI	B/W	q=unm.	z	8		16	16
	L	q=unm.	z	16		28	28
EXG	L	q=Dn		6			
		q=An		6	6		
EXT	W		z	4			
	L		z	4			
JMP			z			8	
JSR			z			16	
LEA	L	z=An	q			4	
LINK		z=unm.	q		16		
LSL,	B/W	q=Dn	z	6+2n			
LSR		q=#1–8	z	6+2n			
	L	q=Dn	z	8+2n			
		q=#1–8	z	8+2n			
Speicher	W		z			12	12
MOVE	B/W	q=Dn	z	4	MOVEA	8	8
		q=An	z	4	MOVEA	8	8
		q=(An)	z	8	MOVEA	12	12
		q=(An)+	z	8	MOVEA	12	12
		q=–(An)	z	10	MOVEA	14	14
		q=d(An)	z	12	MOVEA	16	16
		q=d(An,i)	z	14	MOVEA	18	18
		q=Abs.w	z	12	MOVEA	16	16
		q=Abs.L	z	16	MOVEA	20	20
		q=d(PC)	z	12	MOVEA	16	16
		q=d(PC,i)	z	14	MOVEA	18	18
		q=unm.	z	8	MOVEA	12	12
	L	q=Dn	z	4	MOVEA	12	12

	-(An)	d(An)	d(An)	Abs.W	Abs.L	D(PC)	D(P)	q=unm. z=SR/ CCR	Bedin-gungs-codes XNZVC
	22	24	26	24	28				
	10	12	14	12	16	12	14	8	-****
	16	18	20	18	22	18	20	14	
	12	14	16	12	18	14	16	10	-****
	16	18	20	18	22	18	20	14	
	14	16	18	16	20				-****
	22	24	26	24	28				
									-****

	<164	<166	<168	<166	<170	<166	<168	<162	-***0
	<146	<148	<150	<148	<152	<148	<150	<144	-***0
	14	16	18	16	20				-**00
	22	24	26	24	28				
	18	20	22	20	24				-**00
	30	32	34	32	36				

									-**00
		10	14	10	12	10	14		-----
		18	22	18	20	18	22		-----
		8	12	8	12	8	12		-----

									***0*
	14	16	18	16	20				
	8	12	14	12	16				
	8	12	14	12	16				
	12	16	18	16	20				
	12	16	18	16	20				
	14	18	20	18	22				
	16	20	22	20	24				
	18	22	24	22	26				
	16	20	22	20	24				
	20	24	26	24	28				
	16	20	22	20	24				
	18	22	24	22	26				
	12	16	18	16	20				
	12	16	18	16	20				

Mnemonic	Breite	Adressierungsart		Dn	An	(An)	(An)+
MOVE	L	q=An	z	4	MOVEA	12	12
		q=(An)	z	12	MOVEA	20	20
		q=(An)+	z	12	MOVEA	20	20
		q=-(An)	z	14	MOVEA	22	22
		q=d(An)	z	16	MOVEA	24	24
		q=d(An,i)	z	18	MOVEA	26	26
		q=Abs.W	z	16	MOVEA	24	24
		q=Abs.L	z	20	MOVEA	28	28
		q=d(PC)	z	16	MOVEA	24	24
		q=d(PC,i)	z	18	MOVEA	26	26
		q=unm.	z	12	MOVEA	20	20
MOVE to CCR	W	z=CCR	q	12		16	16
MOVE SR	W	z=SR	q	12		16	16
		q=SR	z	6		12	12
MOVE USP	L	q=USP	z		4		
		z=USP	q		4		
MOVEA	W	z=An	q	4	4	8	8
	L	z=An	q	4	4	12	12
MOVEM	W	q=Rn	z			8+4n	
		z=Rn	q			12+4n	12+4n
	L	q=Rn	z			8+8n	
		z=Rn	q			12+8n	12+8n
MOVEP	W	q=Dn	z				
		q=d(An)	z	16			
	L	q=Dn	z				
		q=d(An)	z	24			
MOVEQ	L	q=unm.	z	4			
MULS	W	z=Dn	q	<70		<74	<74
MULU	W	z=Dn	q	<70		<74	<74
NBCD	B		z	6		12	12
NEG	B/W		z	4		12	12
	L		z	6		20	20
NEGX	B/W		z	4		12	12
	L		z	6		20	20
NOP				4			
NOT	B/W		z	4		12	12
	L		z	6		20	20
OR	B/W	q=Dn	z			12	12
		z=Dn	q	4		8	8
	L	q=Dn	z			20	20
		z=Dn	q	8		14	14
ORI	B/W	q=unm.	z	8		16	16
	L	q=unm.	z	16		30	30

	-(An)	d(An)	d(An)	Abs.W	Abs.L	D(PC)	D(P)	q=unm. z=SR/ CCR	Bedin-gungs-'codes XNZVC
	12	16	18	16	20				
	20	24	26	24	28				
	20	24	26	24	28				
	22	26	28	26	30				
	24	28	30	28	32				
	26	30	32	30	34				
	24	28	30	28	32				
	28	32	34	32	36				
	24	28	30	28	32				
	26	30	32	30	34				
	20	24	26	24	28				
									*****
	18	20	22	20	24	20	22	16	*****
	18	20	22	20	24	20	22	16	
	14	16	18	16	20				

	10	12	14	12	16	12	14	8	-----
	14	16	18	16	20	16	18	12	
	8+4n	12+4n	14+4n	12+4n	16+4n				-----
		16+4n	18+4n	16+4n	20+4n	16+4n	18+4n		
	+8n	12+8n	14+8n	12+8n	16+8n				
		16+8n	18+8n	16+8n	20+8n	16+8n	18+8n		
		16							-----
		24							
									-**00
	<76	<78	<80	<78	<82	<78	<80	<84	-**00
	<76	<78	<80	<78	<82	<78	<80	<84	-**00
	14	16	18	16	20				*u*u*
	14	16	18	16	20				*****
	22	24	26	24	28				
	14	16	18	16	20				*****
	22	24	26	24	28				

	14	16	18	16	20				
	22	24	26	24	28				
	14	16	18	16	20				-**00
	10	12	14	12	16	12	14	8	
	22	24	26	24	28				
	16	18	20	18	22	18	20	14	
	18	20	22	20	24				-**00
	32	34	36	34	38				

Mnemonic	Breite	Adressierungsart		Dn	An	(An)	(An)+
PFA	L		q			14	
RESET				132			
ROR,	B/W	q=Dn	z	6+2n			
ROL		q=#1–8	z	6+2n			
	L	q=Dn	z	8+2n			
		q=A1–8	z	8+2n			
Speicher	W		z			12	12
ROXR,	B/W	q=Dn	z	6+2n			
ROXL		q=#1–8	z	6+2n			
	L	q=Dn	z	8+2n			
		q=#1–8	z	8+2n			
Speicher	W		z			12	12
RTE				20			
RTR				20			
RTS				16			
SBCD	B	q=Dn	z	6			
		q=(An)	z				
STOP				4			
SUB	B/W	q=Dn	z		SUBA	12	12
		z=Dn	q	4	4	8	8
	L	q=Dn	z		SUBA	20	20
		z=Dn	q	8	8	14	14
SUBA	W	z=An	q	8	8	12	12
	L	z=An	q	8	8	14	14
SUBI	B/W	q=unm.	z	8	SUBA	16	16
	L	q=unm.	z	16	SUBA	28	28
SUBQ	B/W	q=unm.3	z	4	4	12	12
	L	q=unm.3	z	8	8	16	16
SUBX	B/W	q=Dn	z	4			
		q=(An)	z			18	
	L	q=Dn	z	8			
		q=(An)	z			30	
SWAP			z	4			
TAS	B		z	4		14	14
TRAP				34			
TRAPV				34	wenn TRAP ausgeführt		
				4	wenn TRAP nicht ausgeführt		
TST	B/W		z	4		8	8
	L		z	4		12	12
UNLK					12		

	-(An)	d(An)	d(An)	Abs.W	Abs.L	D(PC)	D(P)	q=unm. z=SR/ CCR	Bedin-gungs-codes XNZVC
		18	22	18	22	18	22		-----

	14								-**00
	14	16 16	18 18	16 16	20 20				-**00
									*****
									*****
									-----
									*u*u*
									*****
	14 10 22 16	16 12 24 18	18 14 26 20	16 12 24 18	20 16 28 22	12 18	14 20	8 14	*****
	14 16	16 18	18 20	16 18	20 22				-----
	18 30	20 32	22 34	20 32	24 36				*****
	14 22	16 20	18 26	16 24	20 28				*****
									*****
									-**00
	16	18	20	18	22				-**00

									-----
	10 14	12 16	14 18	12 16	16 20				-**00

Anhang G

Glossar

Adreßbus
Der Teil des Bussystems, auf dem die CPU signalisiert, auf welche Adresse des Speichers sie zugreifen will. Die eigentlichen Daten werden gleichzeitig auf dem Datenbus transportiert. Siehe auch Bus, Datenbus.

Adreßdistanz
Differenz zwischen zwei Adressen.

Adreßregister
Register des MC68000, die in erster Linie Adressen enthalten. Sie werden mit A0 bis A7 bezeichnet. A7 nimmt eine Sonderstellung ein; es handelt sich dabei um den sogennannten Stackpointer.

ASCII
American Standard Code for Information Interchange, ein allgemein anerkannter Code für Buchstaben, Ziffern, Sonder- und Steuerzeichen. ASCII existiert in vielen verschiedenen systemspezifischen Abwandlungen.

Assembler
Ein Programm, das vom Menschen geschriebene Mnemoniks in Maschinensprache übersetzt. Zudem bietet ein Assembler oft noch viele weitere Möglichkeiten, wie etwa Symbole oder Adreßberechnungen. Andererseits wird mit Assembler auch die Programmiersprache bezeichnet, die von einem Assembler-Programm übersetzt wird.

Ausnahmebedingung siehe Exception.

BASIC
Abkürzung für "Beginners All Purpose Symbolic Instruction Code", eine sehr verbreitete Interpretersprache, die ursprünglich als Lernsprache konzipiert war, doch in stark erweiterter Form heute auch in den professionellen Bereich vordringt. BASIC existiert in einer unüberschaubaren Anzahl von Dialekten. Vorherrschende Eigenschaften dieser Sprache sind Einfachheit der Bedienung, besonders bei älteren Versionen leider auch Unstrukturiertheit und geringe Geschwindigkeit.

Batchdatei
Eine Textdatei, die mehrere Systemkommandos enthält, die bei einem Aufruf

von einem soganannten Batchprogramm nacheinander abgearbeitet werden, als wären sie über die Tastatur eingegeben worden.

Betriebssystem
Ein oft fest in den Computer eingebautes Programm, das systemnahe Aufgaben wie Ansteuerung der Hardware, Ein/Ausgabe, Initialisierung des Systems und Starten von Anwendungsprogrammen übernimmt.

BIOS
Abkürzung für "Basic Input/Output System". Das BIOS ist in erster Linie für Ein- und Ausgabe auf Laufwerke und sonstige Peripheriegeräte zuständig.

Bitmap
Ein anderes Wort für einen Bildschirmspeicher, in dem für jeden Punkt auf dem Bildschirm einige Bits stehen.

Byte
Eine Einheit von 8 Bits. Ein Byte kann positive Zahlen von 0 bis 255 enthalten.

Bug
(engl. Wanze) Bezeichnung für einen Programmfehler.

Bus
Ein System von Leitungen und dazugehöriger Verwaltungseinheit, die für den korrekten Datenaustausch zwischen den Komponenten eines Computers sorgt, etwa CPU, Hauptspeicher und sonstigen Bausteinen.

C
Eine strukturierte Compilersprache, die sich besonders durch die Möglichkeiten zur maschinennahen Programmierung auszeichnet. C ist auf dem ATARI ST zu Hause, da große Teile des Betriebssystems in C geschrieben wurden.

Cluster
Logische Organisationsform von Blöcken auf einem Laufwerk. Auf dem ST umfaßt ein Cluster zwei physikalische Sektoren, also 1024 Bytes.

CPU
Central Processing Unit, Zentraleinheit eines Computers, die gewöhnlich sämtliche Berechnungen ausführt und Kontrolle über das gesamte System hat.

Carry
(engl. Übertrag) Beim MC68000 ist damit eines der Flags aus dem User-Byte gemeint, das einen Übertrag bei Addition, Subtraktion und Schiebeoperationen anzeigt.

Condition code
(engl. Bedingungscode) Auf dem MC68000 eine Gruppe von 1 oder 2 Buchstaben, die eine Bedingung ausdrücken, die sich aus dem Zustand der 4 Flags N, Z, V und C ergeben. Die Bedingung kann nur wahr oder falsch sein.

Datenbus
Der Teil des Bussystems, auf dem Daten zwischen Speicher, CPU und anderen Bausteinen ausgetauscht werden können. Siehe auch Bus, Adreßbus.

Datenregister
Register des MC68000, mit denen Rechenoperationen durchgeführt werden können. Sie werden mit D0 bis D7 bezeichnet.

Debugger
(engl. "Entwanzer") Ein Hilfsmittel zur Fehlersuche in ausführbaren Programmen.

Dekrementieren
(engl. to decrement) Einen Wert um eins verringern.

Directory
Englisch für Disketteninhaltsverzeichnis. Eine Liste, in der Name, Länge, Entstehungsdatum und etliche andere Verwaltungsinformationen zu sämtlichen Dateien eines Laufwerks aufgeführt werden.

Direktive
In diesem Zusammenhang eine Anweisung im Quellcode, die den Assembler direkt anspricht, aber im allgemeinen nicht direkt Code erzeugt.

Disassembler
Ein Programm, das Maschinensprache in den lesbaren mnemonischen Assemblercode zurückverwandelt.

Displacement
Bei einigen Adressierungsarten ein konstanter Wert, der zu einer indirekt ermittelten Adresse addiert wird, um die tatsächliche Adresse zu liefern.

dummy
(engl. Atrappe) Diese Bezeichnung wird oft für Variablen oder Prozeduren gewählt, die keine Bedeutung haben, aber vorhanden sein müssen.

Extender
Bei den Mnemoniks der MC68000-Maschinensprache bezeichnet man die Anhängsel an Befehle, die die Verarbeitungsbreite oder eine Abwandlung eines

Befehls anzeigen, ".B", ".W" und ".L" sind als Extender für arithmetische und logische Befehle, die die Verarbeitungsbreiten Byte, Wort und Langwort angeben; bei den Branch-Befehlen zeigt ".S" die 8-Bit-Variante an.

Exception
Eine Ausnahmebedingung des Prozessors MC68000. Unter Exceptions fallen Interrupts, vom MC68000 aufgespürte Programmfehler und vom Programm absichtlich ausgelöste Ausnahmebedingungen.

Flag
Eine Variable, die nur zwei Werte annehmen kann (Boolesche Variable). Ein Flag zeigt einem Programm an, ob eine bestimmte Bedingung zutrifft oder nicht zutrifft.

frame pointer
Ein Zeiger auf einen auf dem Stack angelegten lokalen Adreßbereich eines Unterprogramms.

GEMDOS
Ein Teil des Betriebssystems des ATARI ST. GEMDOS übernimmt die weniger hardwarenahen Aufgaben wie Dateiverwaltung und Programmkontrolle.

High
(engl. hoch) Einerseits bezeichnet man damit den Zustand einer Leitung; High heißt, daß die Leitung Strom führt. Andererseits wird High oft im Zusammenhang mit der Wertigkeit von Bytes, Worten oder Bits benutzt: Mit "High Byte" bezeichnet man etwa das hochwertige Byte eines Wortes.

HBI
Abkürzung für "Horizontal Blank Interrupt", ein vom Grafikchip ausgelöster Interrupt, der jedesmal auftreten kann, nachdem der Elektronenstrahl auf dem Monitor eine Zeile fertiggezeichnet hat.

Index
Eine Zahl, die für die Nummer eines Elementes aus einem Feld von Elementen steht.

Integer
Eine vorzeichenbehaftete ganze Zahl; auf dem ATARI ST umfassen Integers gewöhnlich 16 Bits, womit ein Zahlenbereich von –32768 bis +32767 darstellbar ist.

Interrupt
Eine Unterbrechung der normalen Programmabarbeitung. Auf ein Signal der

restlichen Hardware hin tut die CPU kurzzeitig etwas anderes, wobei es sich meist um systemnahe Aufgaben handelt. Danach fährt sie an dem Punkt fort, an dem das laufende Programm unterbrochen wurde, als wäre nichts geschehen.

Inkrementieren
(engl. to increment) Einen Wert um eins erhöhen.

Kaltstart
Komplettes Initialisieren des Systems. Nichts von dem, was vorher im Speicher des Computers installiert war, wird bewahrt. Wird gewöhnlich nur beim Einschalten des Systems ausgeführt.

Kilobyte
Einheit von 2 hoch 10 = 1024 Bytes. Abgekürzt K.

Konsole
Einheit von Tastatur und Bildschirm.

Langwort
Eine Einheit von 32 Bits. Ein Langwort entspricht 2 Worten oder 4 Bytes. Es kann positive Werte von 0 bis 4.294.967.295 enthalten.

LISP
LISP steht für LISt Processing. LISP ist eine Programmiersprache mit recht ungewöhnlichem Konzept, die in erster Linie im Gebiet der künstlichen Intelligenz eingesetzt wird. Meistens wird LISP als Interpreter verwirklicht; es gibt jedoch auch LISP-Compiler.

Logo
Von LISP abgeleitete Interpretersprache, die etwas einfacher zu bedienen ist und oft als Lernsprache dient. Logo eignet sich jedoch durchaus auch für die Programmierung von Künstlicher Intelligenz.

Low
(engl. niedrig) Einerseits bezeichnet man damit den Zustand einer Leitung; Low heißt, daß die Leitung keinen Strom führt. Andererseits wird Low oft im Zusammenhang mit der Wertigkeit von Bytes, Worten oder Bits benutzt: Mit "Low Byte" bezeichnet man etwa das niederwertige Byte eines Wortes.

Maschinensprache
Befehle, die von der CPU eines Rechners direkt ausgeführt werden können.

Maske
Auf Bitgruppen bezogen: Ein Bitmuster, das nur ausgewählte Bits eines Wer-

tes erhält, andere hingegen auf 0 oder 1 setzt, also ausmaskiert. Beispiel: Nehmen wir die Binärzahl %10101010. Nun wenden wir darauf die Maske %00001111 an, indem wir die AND-Verknüpfung benutzen:

```
      %10101010
AND   %00001111
      ----------
      %00001010
```

Man sagt, daß Bits 4 – 7 ausmaskiert worden sind.

Megabyte
Einheit von 1024 Kilobyte oder 1.048.576 Bytes. Abgekürzt MB.

Mikrocode
In der CPU befindet sich Code, der genau angibt, wie die einzelnen Maschinensprachebefehle aus kleineren Operationen zusammengesetzt werden sollen. Der Mikrocode stellt also die Ebene unter der Maschinenspracheebene dar. Für den Assembler ist dies aber kaum von Belang, da zumindest beim MC68000 der Mikrocode nicht modifizierbar ist.

Mnemonic
Bezeichnung, die hinsichtlich der Assoziation mit bekannten Dingen gewählt wurde. Meistens handelt es sich dabei um Abkürzungen.

Modul
In irgendeiner Form abgegrenzter Teil eines Programms.

monadisch
Eine Operation ist monadisch, wenn sie nur einen Operanden hat.

Motorola
Herstellerfirma des Prozessors MC68000.

Objektcode
Aus dem Quellcode vom Compiler oder Assembler erzeugter Maschinensprachecode. Der Objektcode muß allerdings noch nicht unbedingt ausführbar sein.

Overflow
Siehe Überlauf.

Overlay
(engl. Überlagerung) Ein Programmteil, der nur dann geladen wird, wenn er

gebraucht wird, und nach der Benutzung von einem anderen Overlay überschrieben werden kann.

Pascal
Verbreitete strukturierte Compilersprache, benannt nach dem französichen Mathematiker Blaise Pascal.

PC-relativ
Kurz für Programmzähler-relativ. Code ist Programmzähler-relativ, wenn er bei der Adressierung von Variablen oder Labels auf absolute Adressierungsarten verzichtet und somit ohne weitere Vorkehrungen an jeder Stelle im Speicher lauffähig ist. Siehe auch relozierbar.

physikalisch
Real vorhanden.

pointer
Siehe Zeiger.

Quellcode
Programmtext für einen Compiler oder Assembler, wie er direkt vom Programmierer geschrieben wird.

RAM
Ramdom Access Memory, Speicherbausteine mit beliebigem Zugriff. Damit ist Speicher gemeint, den man sowohl beschreiben also auch auslesen kann.

Register
Eine Speicherzelle, die sich in der CPU oder einem anderen Chip befindet. Register dienen zum Ausführen von Rechenoperationen oder enthalten Informationen über den Zustand und die Funktionsweise des Chips, in dem sie sich befinden.

relokatibel
Von englisch relocatable. Siehe relozierbar.

relozierbar
Ein Programm ist relozierbar, wenn es an jeder beliebigen Adresse im Speicher laufen kann. Es ist also frei verschiebbar. Auf dem ATARI ST hat dieses Wort eine spezielle Bedeutung: Eine ausführbare Datei im TOS-Format enthält gewöhnlich eine Relozierungs-Tabelle. Mit Hilfe dieser Informationen kann das Betriebssystem Programme an jeder Stelle im Speicher lauffähig machen, ohne daß das Programm sich in irgendeiner Form in den Adressierungsarten einschränken müßte.

ROM
Read Only Memory, Nur-Lese-Speicher. Speicher, den man nur auslesen kann, aber dessen Inhalt nicht beliebig verändert werden kann.

Routine
Anderes Wort für Unterprogramm.

Schlange
(engl. queue) Eine Datenstruktur, die nach dem LIFO-Prinzip funktioniert (Last In – First Out), etwa entsprechend dem Verhalten einer Menschenschlange vor einer Theaterkasse. Werte können immer nur an einem Ende an eine Schlange angehängt werden und am anderen wieder herausgeholt.

Sektor
Organisationseinheit einer Diskette oder Festplatte. In der Regel kann ein Laufwerk immer nur Operationen mit vollständigen Sektoren durchführen. Auf dem ATARI ST umfaßt ein Sektor gewöhnlich 512 Bytes.

Shell
(engl. wörtlich Schale) Benutzeroberfläche für ein Betriebssystem oder ein anderes Programm.

Shifter
(wörtlich Verschieber) Grafik-Chip des ATARI ST, der dafür sorgt, daß die Informationen aus dem Bildschirmspeicher zu Signalen für den Monitor "verschoben" werden.

Sprite
(engl. Kobold) Bezeichnung für ein kleines Objekt, das beliebig auf dem Bildschirm umherbewegt werden kann.

Symbol
Im Zusammenhang mit Assemblern ist mit einem Symbol eine Zeichenkette gemeint, die für eine bestimmte Speicherzelle oder eine Konstante steht.

Taktzyklus
Um alle Komponenten eines Computers miteinander zu synchronisieren, wird das gesamte System einem bestimmten Takt unterworfen. Ein Taktzyklus ist die kleinste Zeiteinheit, innerhalb der sich der abstrakte Zustand des Systems sich in irgendeiner Form ändern kann.

Timing
Zeitgebundene Ausführung bestimmter Operationen.

TOS
Betriebssystem der ATARI ST. TOS steht für "Tramiel Operating System", nach Jack Tramiel, dem Chef der Firma ATARI. TOS setzt sich zusammen aus GEMDOS, BIOS und XBIOS.

Überlauf
(engl. Overflow) Ein Überlauf tritt auf, wenn bei Berechnungen der darstellbare Zahlenbereich verlassen wird. Bei Prozessoren bezieht sich der Überlauf gewöhnlich nur auf vorzeichenbehaftete Zahlen.

Variante
Beim MC68000 Abwandlung eines Befehls hinsichtlich der Adressierungsarten. Etwa bilden ADDQ, ADDA und ADDI Varianten von ADD.

VBI
Abkürzung für "Vertical Blank Interrupt", ein vom Grafikchip ausgelöster Interrupt, der jedesmal auftritt, nachdem der Elektronenstrahl auf dem Monitor ein Bild fertiggezeichnet hat.

Vektor
Ein Zeiger auf eine bestimmte Routine, meist auf eine Betriebssystemroutine, deren Position im Speicher feststeht.

Warmstart
Ein Zurücksetzen des Systems in einen definierten Zustand. Im Gegensatz zum Kaltstart muß das System jedoch nicht völlig neu initialisiert werden. Das vorher laufende Programm wird aber auf jeden Fall radikal abgebrochen. Ein Warmstart tritt auf, wenn die Reset-Taste gedrückt wird.

Wort
Auf dem ATARI ST eine Einheit von 16 Bits. Ein Wort entspricht zwei Bytes; es kann positive Zahlen von 0 bis 65535 enthalten.

XBIOS
Abkürzung für "eXtended Basic Input/Output System". Eine Sammlung von Routinen für die Nutzung der speziellen Hardwareeigenschaften des ATARI ST.

Zeiger
Eine Variable oder ein Speicherplatz, der die Adresse eines beliebigen Datenobjekts enthält.

Zweierkomplement
Das Zweierkomplement einer binären Zahl erhält man, wenn man jedes Bit invertiert und zum Ergebnis 1 hinzuzählt.

Zweierkomplementzahl
Eine bestimmte Art, eine vorzeichenbehaftete Zahl darzustellen.

Stichwortverzeichnis

von Frank Aumann, Peter Maier, Ralf Stöpper

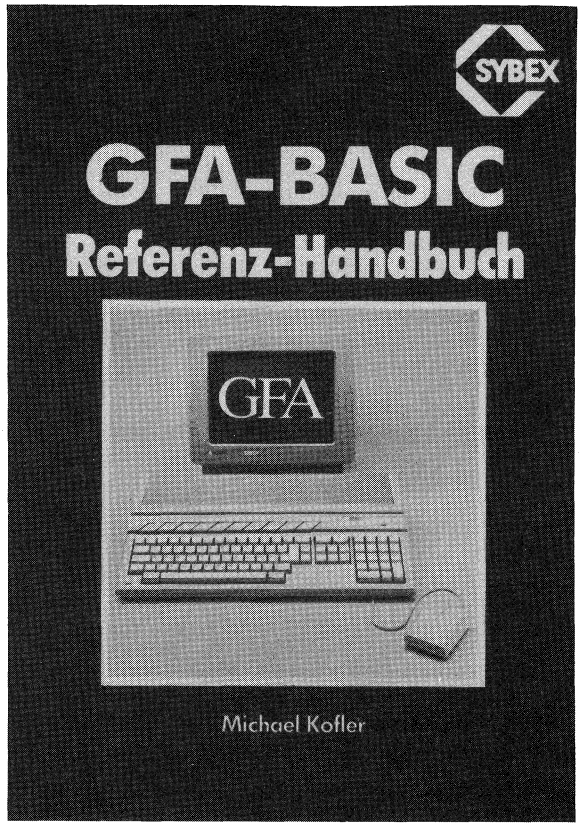
mit Programm-Diskette – Wichtige Hintergrund-Informationen über die Floppy-Laufwerke und deren Funktionen. Die Power-Disk enthält eine Fülle nützlicher unter GEM ablauffähiger Programme. Die wichtigsten Module der Programme sind als Quell-Listings (in C oder Assembler) im Buch enthalten.

168 Seiten, ca. 24 Abb., plus Diskette, Best.-Nr. **3642** (1986)

von Michael Kofler

Der Autor erläutert die grafischen Fähigkeiten der ATARI ST-Computer und führt anhand einer Vielzahl von BASIC-Programmen in die Programmierung zwei- und dreidimensionaler Grafiken sowie bewegter grafischer Objekte ein. Alle Programmbeispiele werden durch Original-Bildschirmkopien dokumentiert.

272 Seiten, ca. 120 Abb.,+ 8 Vierfarb-Seiten, mit integrierter Programm-Diskette, Best.-Nr. **3673** (1987)



von Michael Kofler

In diesem umfangreichen Arbeits- und Nachschlagewerk finden Sie wirklich alle zum Programmieren notwendigen Informationen gebündelt. Dabei ist die logisch geordnete Befehlsliste nur ein Bestandteil des Buches; zusätzlich werden schwer bedienbare Befehle und Funktionen sehr ausführlich beschrieben, wobei der Autor sinnvolle Beispiele hinzugefügt hat. Außerdem gibt er zusätzliche Informationen über Programmiertechniken; damit Probleme bei der GEM-Programmierung (die auch beschrieben werden) Ihre Arbeit nicht unnötig verzögern, bietet Michael Kofler Ihnen direkt die zur Lösung erforderlichen Kenntnisse über das Betriebssystem. Highlights sind u. a. die Verwendung der RSC-Dateien und eine Fensterverwaltung mit Beispielprogramm. Erfahren Sie, was Ihr Rechner mit GFA-BASIC tatsächlich leisten kann!

536 Seiten, zahlr. Abb. Best.-Nr. **3555** (1987).

Die SYBEX-Bibliothek

Atari

ARBEITEN MIT DEM ATARI ST
von Karl-Heinz Hauer vermittelt Ihnen notwendige Kenntnisse zum Umgang mit den ATARI ST-Computern, z. B. System-Hardware, Betriebssystem-Adressen, TOS, Kernel-Routinen, ATARI-BASIC, ATARI-Logo. 432 Seiten, 172 Abb. Best.-Nr. **3623** (1986)

ATARI ST – ARBEITEN MIT GEM, Bd. 1: DIE AES-BIBLIOTHEK
von Gerd Sender – Anhand einer Vielzahl von Beispielen wird gezeigt, wie der unter der Sprache C programmierende ATARI-Besitzer sich die AES-Bibliothek eröffnen und zunutze machen kann. 320 Seiten, 36 Abb., Best.-Nr. **3626** (1987). Eine Programm-Diskette ist im Buch integriert und enthält die vorgestellten Programme und Unterroutinen.

ATARI ST – ARBEITEN MIT GEM, Bd. 2: DIE VDI-BIBLIOTHEK
von Holger Danielsson/Andreas Volkmann – Der ATARI-ST-Nutzer wird anhand einer Vielzahl kleiner C-Routinen mit dem Aufruf der VDI-Bibliothek von GEM und der Einbindung in eigene Programme bekannt gemacht. 240 Seiten, ca. 48 Abb., Best.-Nr. **3627** (1986), Mit integrierter Programm-Diskette, die Programme und Unterroutinen enthält.

ATARI ST – ARBEITEN MIT CP/M
von Bernhard Bachmann – Für ATARI ST-Nutzer, die auf ihrem Rechner Standardsoftware (z.B. WordStar) unter dem Betriebssystem CP/M nutzen möchten. Mit allen notwendigen Hinweisen für die Arbeit mit CP/M und die Übertragung für andere Systeme vorliegender CP/M-Programme auf den ST: CP/M-Emulatoren, CP/M-Dienstprogramme, CP/M-Controlcodes u.v.m. 256 Seiten, ca. 50 Abb., Best.-Nr. **3665** (1987)

ATARI ST – EINFÜHRUNG IN WORDSTAR
von Arthur Naiman – Das Originalwerk „Einführung in WordStar" ist seit Erscheinen 1983 ein SYBEX-Bestseller. Um der Arbeit in der speziellen System-Umgebung des ATARI ST unter Kontrolle der CP/M-2.2-Emulatoren gerecht zu werden, wurde das Buch für ST-Nutzer überarbeitet und durch Zusatz-Informationen ergänzt. 280 Seiten, mit Abb., Best.-Nr. **3666** (1986)

ATARI ST STARFILE
Dateiverwaltung plus Bildverarbeitung von Heino Hansen/Elmar Sonnenschein – Ein Karteikarten-orientiertes Dateiverwaltungs-Programm für den ATARI ST, das Ihnen etwas ganz Besonderes bietet: die Verarbeitung grafischer Informationen – selbst digitalisierter Bilder. Egal, ob diese mit einem handelsüblichen ST-Grafikprogramm oder mit dem StarFile-Editor erstellt wurden. Das Programm stellt beliebige Masken für die Arbeit mit Dateien oder deren Ausgabe zur Verfügung. Weitere Spezialitäten von StarFile: ISAM-Dateiverwaltung nach dem B-Tree-Verfahren voll dokumentiert; flexible Drucker-Anpassung für 9/24-Nadeldrucker bzw. Laserdrucker (soweit lieferbar); Analog-/Digital-Uhr mit Datum; eigenes Snapshot. Das leicht bedienbare Programm ist voll unter GEM oder optional über die Tastatur zu steuern. Diskette + Trainingsbuch, Best.-Nr. **4006** (1987)

ATARI ST STARCOMM
von Arnd Beißner – Das universelle Telekommunikations-Programm zur Übertragung beliebiger Daten zwischen Computern oder über die Netze der Bundespost. StarComm arbeitet unter GEM, wobei alle Funktionen dialog- und menüunterstützt sind. Eine Besonderheit für ST-Systeme mit Echtzeit-Uhr: Das zeitgesteuerte Upload von Dateien mit automatischer Herstellung der Telefonverbindung zu programmierten Sendezeiten (automatische Wiederwahl). Außerdem: Übertragung von Texten, Programm-Infos und Grafiken; 16 Übertragungsraten zwischen 50 und 19600 Baud; Softscrolling der Bildschirm-Ausgabe; Telefonregister zur automatischen Nummernwahl; komfortabler Editor; jederzeit abrufbare Hilfstexte u.v.m. Telekommunikationsprogramm mit Handbuch, Best.-Nr. **4039** (1987)

ATARI ST STARPAINTER
von Heino Hansen/Elmar Sonnenschein – Erfahren Sie, was Sie – zusammen mit Ihrem ATARI ST – als Grafiker drauf haben. Mit StarPainter ist das einfacher, als Sie denken. Die Kreation und der Ausdruck von einfachen Strichzeichnungen über Körper wie Prisma und Würfel bis hin zu Polygomen werden Ihnen mit diesem Programm leicht gemacht. Wenn Ihnen ein bestimmter Bildausschnitt besonders gefällt: Ausschneiden, verschieben, vergrößern, abspeichern und drucken – das bereitet StarPainter keine Probleme. Weitere Extras: Mehrere Bildschirm-Ebenen; UNDO-Funktion; Lesen diverser Bildformate; spezielle Anpassung für Grafik-Tableau; Zeichen-Editor; Füllmuster-Editor. Grafikprogramm mit Trainingsbuch, Best.-Nr. **3424** (1987)

Andere Programmiersprachen

ERFOLGREICH PROGRAMMIEREN MIT C
von J. A. Illik – ein unentbehrliches Handbuch für jeden, der mit der universellen Sprache C erfolgreich programmieren will. Aussagekräftige Beispiele, auf verschiedenen Mini- und Mikrocomputern getestet. 408 Seiten, Best.-Nr.: **3055** (1984)

C – EINE EINFÜHRUNG
von Bruce H. Hunter – Das ideale Buch für den Einsteiger in die Programmiersprache C, speziell für Anwender, die von BASIC auf den leistungsfähigen Compiler umsteigen wollen. 296 Seiten, ca. 12 Abb., Best.-Nr. **3632** (1986)

Commodore

COMMODORE 64 STARTEXTER
Textverarbeitung mit Diskette und Handbuch – StarTexter ist die Textverarbeitung mit Doppelnutzen: das Buch führt Sie in die Textverarbeitung mit Ihrem C64 ein, die Diskette bietet Ihnen ein exzellentes Programm – komplett zu einem erstaunlichen Preis! Version 5.0 mit Schnittstelle zum C64 StarPainter. 160 Seiten, Handbuch und Diskette, Best.-Nr.: **4038** (1987)

COMMODORE 64 STARDATEI
von Toni Schwaiger – Der universelle Karteikasten für den C 64, mit dem sich beliebige Daten speichern und wie bei einem Karteikasten bearbeiten lassen. Voll kompatibel zu StarTexter mit echten MailMerge-Funktionen – und ebenso komfortabel wie auch bedienerfreundlich. Diskette und ausführliches Trainingsbuch (96 Seiten) Best.-Nr. **3413** (1985)

DAS GROSSE COMMODORE BASIC HANDBUCH
von Michael Orkim – BASIC komplett für alle Commodore-Rechner von VC 20 bis C128. BASIC-Versionen 2.0, 3.5, 4.0, 7.0. Mit Tips für die Programmübertragung zwischen den einzelnen Modellen und für Befehls-Simulation sowie BASIC-Erweiterungen. 640 Seiten, Best.-Nr. **3615** (1986)

C 128 STARTEXTER
von Toni Schwaiger – Die Textverarbeitung der Spitzenklasse auch für professionelle Anwender mit dem Commodore C 128. Außergewöhnliche features, die den C 128 zum Textverarbeitungs-Star werden lassen – zum kleinen Preis. Diskette + Trainingsbuch (120 Seiten), Best.-Nr. **3415** (1986)

COMMODORE 64 STARPAINTER
von H. Hansen/E. Sonnenschein – Das Grafikprogramm (plus Trainingsbuch) der Spitzenklasse, mit dem Sie sogar professionell arbeiten können. Und das komfortabel, gut verständlich und zum kleinen Preis. Diskette mit ausführlichem Handbuch, Best.-Nr. **3421** (1986)

DAS C 128 BUCH
von Larry Greenly u. a. – Commodores Originalbuch-Handbuch für Programmierer. Mehr brauchen Sie nicht, um den leistungsfähigen Commodore PC 128 schnell kennen zu lernen und direkt sicher für Ihre Aufgabenstellungen nutzen zu können. 880 Seiten, Best.-Nr. **3618** (1986)

COMMODORE 128 STARDATEI
von Toni Schwaiger, dem Autor des Textverarbeitungs-Pakets Commodore 128 StarTexter. Ein leistungsfähiges und komfortables Dateiverwaltungs-Programm der Profiklasse mit Trainingsbuch, natürlich voll kompatibel zu StarTexter. Diskette + Trainingsbuch, Best.-Nr. **3420** (1987)

COMMODORE 128 STARPAINTER
von Heino Hansen und Elmar Sonnenschein. Das bedienerfreundliche Grafikprogramm der vielen Möglichkeiten, mit dem Sie professionelle Grafiken auf Ihrem C 128 erstellen. Den reibungslosen Einstieg ermöglicht das ausführliche Trainingsbuch. Diskette + Trainingsbuch, Best.-Nr. **3422** (1987)

SYBEX

ATARI ST

Programmieren in Maschinensprache

Der Autor führt in leicht verständlicher Art in die Entwicklung von Maschinensprache-Routinen für die ATARI ST Systeme ein. Er erläutert den Aufbau und die Funktionsweise des Mikroprozessors MC 68000, dessen Befehlssatz und Adressierungsarten.

Alle wichtigen Funktionen werden anhand von kleinen, in der Assemblersymbolik entworfenen Routinen erläutert, die nach Einbindung in die Benutzeroberfläche des ATARI ST lauffähig sind.

Besonders wertvoll sind Informationen über die Verknüpfung zwischen Programm-Modulen des Betriebssystems und vom Anwender entwickelten Assemblerroutinen.

Aus dem Inhalt:

- Einführung in die Maschinensprache
- Der Befehlssatz des MC 68000
- Grafik- und Soundprogrammierung
- Zusammenarbeit mit dem Betriebssystem
- Die GEMDOS, BIOS- und XBIOS-Routinen

ISB N 3-88745-678-5

DM 48,—
sfr 44,20
öS 374,—

9 783887 456788