



Autoren: Sören Hellwig
Markus Fritze

Anleitung: Christoph Pagalies
Markus Fritze

Der Assembler für den Atari ST
Version 1.8

Programmdesign und -entwicklung von Σ -soft für OMIKRON.Software.

Designer: Markus Fritze

Sören Hellwig

Schreiberlinge: Christoph Pagalies

Markus Fritze

Vertrieb: OMIKRON.Soft- & Hardware GmbH

Erlachstraße 15, D-7534 Birkenfeld 2

Telefon: 07082/5386

Dank an: AssAge Entertainment Software

Thomas Hertzler — Lothar Schmitt

Oberstraße 31, D-4330 Mülheim/Ruhr

sowie alle unsere unermüdlichen Tester:

Harald, Volker, Thomas, u. v. a. m.

Die Anleitung wurde mit Stefan Lindner's TeX-Implementation auf einem Atari ST (was sonst...) erstellt.

Alle Rechte, insbesondere das Recht auf Vervielfältigung, Verbreitung und Übersetzung vorbehalten. Kein Teil des Werkes darf in irgendeiner Form ohne ausdrückliche schriftliche Genehmigung der OMIKRON.Software GmbH reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

Das vorliegende Handbuch und die dazugehörigen Programme wurden mit der größten Sorgfalt erstellt. Trotzdem sind Fehler nie ganz auszuschließen. Daher möchten wir Sie darauf hinweisen, daß wir weder eine Garantie für die Fehlerfreiheit geben, noch die Haftung für irgendwelche Folgen, gleich ob durch Fehler im Handbuch, in der Software oder in der Hardware verursacht, übernehmen können.

Wenn Sie einen Fehler in der Software gefunden haben: Schreiben Sie bitte an Σ -soft, z. Hd. Markus Fritze, Birkhahnkamp 38, D-2000 Norderstedt 1. Besitzer eines Modems bzw. Akustikkopplers können mich auch unter dem Namen „Markus“ in der SSB-Mailbox (Telefon 040/6046266, 300/1200 Baud, 8N1) erreichen.

Inhaltsverzeichnis

1 Vorwort	14
2 Inhalt der Diskette	17
3 Installation	19
4 Gemeinsames von Assembler und Debugger	20
4.1 Die Benutzeroberfläche	20
4.2 Die Dialogboxen	20
4.3 Die Environment-Variable	22
4.4 Blitterunterstützung	22
5 Der Assembler	23
5.1 Zur Geschichte des Assemblers	23
5.2 Einführung in den Assembler	25
5.3 Die Drop-Down-Menüs	28
5.4 Die Funktionen der Maus	28
5.5 Der Editor	30
5.5.1 Besonderheiten des Assemblers	31
Konvertierungen in den Motorola-Standard	32
Lokale Label	32

Optimierungen	34
5.5.2 Formelrechnungen	34
5.5.3 Symbole, Label und Konstanten	37
Rechnen mit Symbolen	38
5.5.4 Tastaturkommandos	40
5.5.5 Die Drop-Down-Menüs	45
Das Menü „Atari“	47
Das Menü „Datei“	48
Das Menü „Assembler“	55
Das Menü „Editor“	61
Das Menü „Suchen“	66
Das Menü „Block“	73
Das Menü „Einstellungen“	73
Das Menü „Hilfe“	80
Die ASCII-Anzeige	81
Die Uhrzeit	81
5.6 Die Pseudo-Opcodes	81
5.6.1 Definition von Datenbereichen	82
5.6.2 Daten-Strukturen mit RS	85
5.6.3 Segmentierung	86
5.6.4 Symbole	88
5.6.5 Assembleroptionen	89
5.6.6 Dateioperationen	90
5.6.7 Allgemeines	92
5.6.8 Bedingte Assemblierung	97
5.7 Fehlermeldungen des Assemblers	100
5.7.1 Optimierungen	100
5.7.2 Warnungen	100

5.7.3	Sofort auftretende Fehler	101
5.7.4	Fehler bei der Assemblierung	102
5.8	Das Zusammenspiel Assembler ↔ Debugger	104
6	Der Debugger	106
6.1	Vorwort	106
6.2	Starten des Debuggers	107
6.3	Für den Anfänger	108
6.4	Allgemeine Bedienung	108
6.4.1	Der Bildschirmaufbau	108
6.4.2	Die Tastaturbelegung	111
6.5	Die Funktionstasten	117
6.5.1	F1 — Trace	117
6.5.2	SHIFT-F1 — Tracesimulator 68 020	117
6.5.3	F2 — Do PC	117
6.5.4	SHIFT-F2 — Trace no subroutines	118
6.5.5	F3 — Trace until RTS	118
6.5.6	SHIFT-F3 — Trace until RTE	119
6.5.7	F4 — Trace Traps	119
6.5.8	SHIFT-F4 — Go PC	119
6.5.9	F5 — Skip PC	119
6.5.10	SHIFT-F5 — Insert/Overwrite	119
6.5.11	F6 — Directory	120
6.5.12	SHIFT-F6 — Marker anzeigen	120
6.5.13	F7 — Memorydump PC	120
6.5.14	SHIFT-F7 — Breakpoints anzeigen	121
6.5.15	F8 — Disassemble PC	121
6.5.16	SHIFT-F8 — Memoryinfo	121

6.5.17 F9 — List PC	122
6.5.18 SHIFT-F9 — Direct	122
6.5.19 F10 — Toggle Screen	122
6.5.20 SHIFT-F10 — Quit	123
6.6 Die Befehle	124
6.6.1 Allgemeines	124
6.6.2 Schreibweise der Befehle	124
6.6.3 Variablen des Debuggers	125
6.6.4 Die Formelauswertung	130
6.6.5 Trace-Funktionen	132
Breakpoints	132
Go	133
Trace	134
Showmemory	134
Do, Call	134
Untrace	135
If	136
Observe	138
Cacheclr	140
Cacheget	140
Cls Δ	141
Mouseon Δ	141
Mouseoff Δ	141
Getregister	141
Bssclear	142
Initregister	142
.	142
Sync50, Sync60	143

6.6.6	I/O-Befehle	143
	Lexecute Δ	143
	Labelbase	144
	Load Δ	144
	Save Δ	145
	Directory Δ	145
	Prn	146
	File Δ	146
	Fopen Δ	146
	Fclose Δ	147
	Line	147
	Cr	147
	“	147
	Erase, Kill Δ	148
	Free Δ	148
	Mkdir directory Δ	148
	Rmdir directory Δ	149
	Name Δ	149
	Fatattrib Δ	149
	Format Δ	150
	Type Δ	150
	Rwabs Δ	151
	Readsector Δ	151
	Writesector Δ	152
	Readtrack	152
6.6.7	Listende Befehle mit Scrolling	153
	Disassemble	153
	/	154

List	154
!	154
Symboltable	155
Dump, Memory	156
.W und .L	157
,	157
]	157
Ascii	158
)	158
6.6.8 Allgemeine Befehle	158
Find	158
Hunt	159
Ascfind	159
Continue	159
Info	160
?	160
Move, Copy	160
Fill	161
Clr	161
Compare	162
Checksumme	162
Resident Δ	163
Exit, Quit, System Δ	163
©	163
Set, Let	164
Key	164
Reset all Δ	165
Switch	166

Scrollon	166
Scrolloff	166
Cursor	167
Help	167
6.7 Internas zum Debugger	167
6.8 Die Datei „OM-DEBUG.INF“	168
6.9 Ein paar Tips...	170
6.10 Externe Abbruchmöglichkeiten	170
6.11 Meldungen des Debuggers	171
6.12 Der „Resident“-Befehl	174
6.13 Debugger-Demo	175
A Die Makro-Version	181
B Ein kleines internes Problem...	182
C Einige Begriffserklärungen	184
D Aufbau von Betriebssystemstrukturen	189
D.1 Boot-Sektor	191
D.2 Aufbau einer Symboltabelle	191
D.3 Die Symboltabelle des Assemblers	192
D.4 Relozierinformationen	192
D.5 Der Druckertreiber	194
E Probleme mit älteren TOS-Versionen	198
F Assembler und OMIKRON.BASIC	202
G Laden an beliebige Adressen	205

H Das Modulkonzept	206
H.1 Allgemeine Richtlinien:	206
H.2 Der Aufbau des Modulheaders:	207
H.3 Die Routine „init“:	208
H.4 Die Routine „disable“:	209
H.5 Die Routine „choose“:	209
H.6 Die vorhandenen Prozeduren:	210
H.7 Der RSC-Editor	214
H.7.1 Der Aufbau des Formats	214
Zur Beachtung	214
Allgemeiner Aufbau	214
Die verschiedenen Typen	215
Icons	216
H.7.2 Der Editor	216
Die Iconlibrary	217
I Die beiliegenden Dateien	218
I.1 AES_DEMO.SRC und AES_DEMO.PRG	218
I.2 AES_VDI.S	219
I.3 AUTOBOOT.SRC	219
I.4 BIOS.S, XBIOS.S und GEMDOS.S	219
I.5 BOOT_2.0.PRG	220
I.6 BREAK.S	220
I.7 CALL.SRC und CALL.PRG	220
I.8 DAY_CALC.SRC	221
I.9 DESKTOP.PRG und DESKTOP.SRC	221
I.10 EPSON.HEX	221
I.11 GAME.OMI	222

I.12 GEMDOS.S	223
I.13 GEM_VARS.S und GEM_VARS.SYM	223
I.14 HANOI.SRC	223
I.15 HEADER.S	224
I.16 L_XXXXXX.SRC und L_0C0000.PRG	224
I.17 MAKE_CFG.SRC, EPSON.HEX und NEC.HEX	224
I.18 NEC.HEX	225
I.19 OM_MODUL.SRC	225
I.20 SCRNDUMP.SRC und SCRNDUMP.PRG	225
I.21 SCRNERR.SRC	225
I.22 Σ-RSC	225
I.23 SHIP.ACC, SHIP.PRG und SHIP.SRC	226
I.24 SYS_VARS.S und SYS_VARS.SYM	226
I.25 TAKT_TAB.DOC	226
I.26 TATÜ.ACC und TATÜ.SRC	227
I.27 Think and Work	227
I.28 TOS_LOAD.SRC und TOS_LOAD.PRG	227
I.29 VDISK3.ACC und VDISK3.SRC	227
I.30 WPROT.SRC	227
I.31 XBIOS.S	228
I.32 XBRA.SRC und XBRA.TXT	228
J Literaturhinweise	229

Tabellenverzeichnis

4.1 Die Tasten in Dialogboxen	21
5.1 Konvertierungen in den Motorola-Standard	33
5.2 Die Rechenoperationen des Assemblers	36
5.3 Die Escape-Belegungen	46
5.4 Die Gründe fürs Disabeln	61
5.5 Das DX-Demo	85
6.1 Die Flags des Statusregisters	109
6.2 Allgemeine Tastenkommandos	115
6.3 Zusätzliche Tastenkommandos	116
6.4 Der Zehnerblock	116
6.5 Die Zahlensysteme	130
6.6 Die erlaubten Vorzeichen	130
6.7 Die erlaubten Rechenzeichen	131
6.8 Die Fileattribute	150
D.1 Der Programmheader	189
D.2 Die Basepage	190
D.3 Der Boot-Sektor	195
D.4 Die Symboltabelle	197
D.5 Die Symboltabelle des Assemblers	197

Abbildungsverzeichnis

5.1	Die Drop-Down-Menüs (eingeklappt)	45
5.2	Das Menü „Atari“	47
5.3	Die Info-Box	47
5.4	Die Dankesliste	48
5.5	Das Menü „Datei“	48
5.6	Die Sicherheitsabfrage bei „Neu anlegen...“	49
5.7	Die Passwordabfrage beim Laden	50
5.8	Eine Symboltabelle wurde erzeugt	52
5.9	Die Sicherheitsabfrage vor dem Überschreiben	53
5.10	Setzen des Passwords oder Seriennummerschutzes	54
5.11	Die Löschen-Dialogbox	54
5.12	Die Dialogbox beim Beenden	55
5.13	Das Menü „Assembler“	55
5.14	Die Dialogbox nach dem Assemblieren	57
5.15	Die Sicherheitsabfrage zum Löschen des Debuggers	61
5.16	Das Menü „Editor“	62
5.17	Der Taschenrechner	62
5.18	Die Zeichentabelle	63
5.19	Die Informationsbox	64
5.20	Der Reorganisations-Dialog	65

5.21 Die Dialogbox Block/Alles drucken	66
5.22 Der Drucker antwortet nicht...	66
5.23 Das Menü „Suchen“	67
5.24 Der Dialog „Symbol suchen“	68
5.25 Die Dialogbox „Symbol ersetzen“	69
5.26 Der Dialog „Text suchen“	70
5.27 Der Dialog „Text ersetzen“	71
5.28 Der Dialog „Sprung zu Zeile“	71
5.29 Das Menü „Block“	73
5.30 Das Menü „Einstellungen“	74
5.31 Die Darstellungs-Dialogbox	74
5.32 Die Dialogbox „Editor1“	76
5.33 Die Dialogbox „Editor 2“	77
5.34 Die Dialogbox „Editor 3...“	79
5.35 Die Dialogbox „Funktionstasten belegen“	81
5.36 Das Menü „Hilfe“	81
5.37 Die Dialogbox zum Uhrstellen	82
5.38 Abbruch bei OPT P+	90
5.39 Keine weiteren Fehler vorhanden	102
 6.1 Die Bildschirmseite	108
6.2 Die Funktionstastenleiste	117
6.3 Marker anzeigen im Debugger	120
6.4 Die Breakpointanzeige	121
6.5 Die Speicherinfoanzeige	122
6.6 Programmende des Debuggers	123
 B.1 Ein kleines internes Problem	183

D.1	ASCII-Tabelle	193
D.2	Scancode-Tabelle	193

Kapitel 1

Vorwort

Lieber Programmierer,

mit dem OMIKRON.Assembler haben Sie ein schnelles und komfortables Entwicklungssystem erworben, das Sie in jeder Phase der Programm-Entwicklung optimal unterstützt.

Der OMIKRON.Assembler ist für Profis entwickelt worden, d.h., es wurden viele neue Funktionen und Ideen implementiert, die noch kein anderer Assembler bietet; jeder Programmierer, der einmal mit diesem Assembler gearbeitet hat, wird diese Funktionen zu schätzen wissen.

Das soll aber nicht heißen, daß Anfänger mit diesem Assembler falsch beraten sind. Ganz im Gegenteil, gerade die sehr einfachen und zugleich sehr komfortablen Möglichkeiten des Debuggens in Verbindung mit einer extrem hohen Sicherheit gegen Fehlbedienung erlauben es dem Anfänger, diese Programmiersprache nach dem Trial-and-Error-Prinzip zu erlernen, was bisher nur bei BASIC-Interpretoren möglich war. Daß der Editor Fehler bereits bei der Eingabe bemerkte, ist dabei wohl selbstverständlich. Wir mußten in dieser Anleitung natürlich einige Fachbegriffe verwenden. Wenn Ihnen ein Wort nicht kennen, sehen Sie bitte im Anhang C ab Seite 184 nach.

Welcher andere Assembler erlaubt es z.B., ein Programm mit einem Tastendruck zu assemblen, das fertige Programm dem Debugger zu übergeben und das Programm automatisch zu starten? Und bei Sourcetexten mit weniger als 10 000 Zeilen dauert das ganze nicht einmal eine Sekunde. Wenn das Programm ordnungsgemäß beendet wurde, wird ebenfalls automatisch

in den Editor zurückgesprungen. Falls das Programm jedoch abstürzt, ist dies auch kein Beinbruch. Einfach CONTROL-HELP¹ drücken und schon befinden Sie sich wieder im Editor, und zwar in der Zeile, in welcher der Fehler auftrat. Eine Änderung, dann wieder ALTERNATE-A gedrückt, und schon läuft das Programm. Kein umständlicher Link-Vorgang, kein Diskettenzugriff. Die Turn-Around-Zeiten sind vernachlässigbar gering, es wird Sie begeistern!

Weitere neue Konzepte und unsere professionelle Umsetzung werden sicher einiges Erstaunen bei Ihnen hervorrufen:

- Der OMIKRON.Assembler übersetzt durchschnittlich 1 100 000 Zeilen pro Minute — er kann sogar bis zu 3 750 000 Zeilen pro Minute erreichen. Er ist damit zwischen 10 und 600 mal schneller als bisherige Assembler. Der Assembler übersetzt sich selbst in etwa 2 Sekunden — und das bei 30 000 Zeilen Sourcecode.
- Assembler und Debugger sind voll auf den Prozessoren 68 000 (was eigentlich zu erwarten), 68 010 und 68 020 lauffähig. (Allerdings können natürlich auch auf einem 68 020 nur die normalen 68 000-Befehle und -Adressierungsarten verwendet werden).
- Er meldet Tippfehler und falsche Adressierungsarten gleich bei der Eingabe. Sie können sofort bei der Eingabe korrigieren, erhalten also bei der Assemblierung keine ellenlangen Fehlerlisten. Ferner werden doppelt vergebene Symbolnamen — ein häufiger Fehler von Programmierern — bereits bei der Eingabe angemahnt.
- Sehr viele neue Funktionen des Editors erlauben es gerade dem Assemblerprogrammierer, noch effizienter und schneller als bisher zu arbeiten (z. B. Symbole suchen und ersetzen).

Möglich geworden ist dies alles durch einen „tokenisierenden Editor“: Der Assembler übersetzt den Quellcode schon während des Editierens.

Wir haben den OMIKRON.Assembler hinsichtlich der Geschwindigkeit perfektioniert, da die Programme auf dem ST wesentlich länger werden als z. B. auf dem C-64, der Programmierer aber trotzdem nicht auf den Assembler

¹In dieser Anleitung sind alle Tastenkombinationen in der Schriftart **TypeWriter** angegeben.

warten soll. Sie werden bei kürzeren Programmen (bis einige KByte) die Erfahrung machen, daß der OMIKRON.Assembler bereits fertig assembled hat, bevor Sie die Assemble-Taste losgelassen haben. Auf den OMIKRON. Assembler müssen Sie niemals warten!.

Wir haben uns, soweit es ging, an bestehenden Standards orientiert: Geübte Assembler-Programmierer müssen somit nicht umlernen, sondern können wie gewohnt weiterarbeiten. So versteht der Assembler nicht nur den Motorola-Standard, sondern auch diverse andere Befehls- und Pseudo-Opcode-Syntaxen . Unser Editor ist weitgehend Tempus-kompatibel (und das nicht nur in der Tastaturbelegung, auch in der Geschwindigkeit).

Wenn Sie bestehende Quellcodes in den OMIKRON.Assembler übernehmen, werden diese weitgehend automatisch konvertiert. Sie können also begonnene Projekte bequem mit dem OMIKRON.Assembler weiterführen.

Ach ja: Sie haben die Grundversion des OMIKRON.Assemblers erworben. Diese ist gegenüber der Makro-Version zwei Einschränkungen unterworfen:

1. Die Grundversion beherrscht keine Makros (hätte man sich fast denken können...). Makros (Befehls-Zusammenfassungen) sind sinnvoll, wenn Sie von Assembler aus viele Betriebssystemaufrufe machen wollen. Wir haben Libraries mit Betriebssystemaufrufen auf der Diskette mitgeliefert; dadurch können Sie im Regelfall auch hierbei auf Makros verzichten.
2. Die Grundversion besitzt keinen Linker. Linken (Programmteile zusammenbinden) müssen Sie bei normalen Assemblern z. B. deshalb, damit das Assemblieren nicht so lange dauert. Das ist beim OMIKRON.Assembler nicht nötig. Der OMIKRON.Assembler besitzt schon in der Grundversion eine begrenzte Linkfähigkeit, d. h. Assemblerprogramme können zu anderen Programmen zugelinkt werden. (Siehe auch Seite 60)

Falls Sie auf Makros bestehen, oder wenn Ihnen die Link-Möglichkeiten des OMIKRON.Assemblers nicht ausreichen, empfehlen wir die Makro-Version des OMIKRON.Assemblers; Sie erhalten ein Upgrade gegen den Differenzbetrag zwischen den Verkaufspreisen der Grund- und der Makro-Version.

Kapitel 2

Inhalt der Diskette

Achtung: Die Diskette ist doppelseitig formatiert. Die ganzen Dateien hatten auf einer einseitigen Diskette keinen Platz — da fast alle Atari ST-Besitzer ein doppelseitiges Laufwerk haben, wollten wir das Programm nicht durch eine zweite Diskette unnötig verteuern. Wenn Sie über kein doppelseitiges Laufwerk verfügen sollten, können Sie sich Ihre Originaldiskette bei Ihrem Händler oder einem Freund sicherlich auf zwei Disketten umkopieren.

Auf der beiliegenden Diskette befinden sich folgende Dateien:

- OM-ASSEM.PRG — Der Assembler (siehe Kapitel 5 auf Seite 23)
- OM-DEBUG.PRG — Der Debugger (siehe Kapitel 6 auf Seite 106)
- CALL .PRG — ruft den residenten Debugger auf (siehe Kapitel I.7 auf Seite 220)
- OM-ASSEM.CFG — Druckeranpassung des Assemblers (siehe Kapitel D.5 auf Seite 194)
- OM-ASSEM.DAT — Module des Assemblers (siehe Kapitel H auf Seite 206)
- README .TXT — Korrekturen der Anleitung, etc. (als erstes lesen!)

Der Ordner AUTO mit folgenden Dateien:

AUTO0300S.PRG — Kopiert Desktop-Infos, paßt Drucker an... (siehe Kapitel I.3 auf Seite 219)

DESKTOP .IN0 — Desktop-Info für niedrige Auflösung

DESKTOP .IN1 — Desktop-Info für mittlere Auflösung

DESKTOP .IN2 — Desktop-Info für hohe Auflösung

FSELECT .PRG — Ein neuer File-Selector

Die Ordner DEMOS.OMI, LIBRARYS.OMI und UTILITY.OMI, in denen einige nützliche Utilities u. ä. (wie eben die Namen sagen) enthalten — die Bedeutung der einzelnen Dateien sind im Anhang I erläutert.

Der Ordner GAME.OMI enthält das Spiel „Think and Work“. Er wird auch im Anhang I beschrieben.

Der Ordner Σ-RSC enthält einen eigenen RSC-Editor, der für das Erstellen eigener Module nützlich ist. Das Modulkonzept ist im Anhang H erklärt.

Kapitel 3

Installation

Zuerst sollten Sie den Assembler und den Debugger installieren; starten Sie dazu das Programm INSTALL.PRG auf Ihrer Originaldiskette.

Das Programm fragt Sie nach Namen und Adresse. Geben Sie zuerst Ihren Namen ein, und schließen Sie die Eingabe mit RETURN ab. Korrekturen können mit den Cursortasten und DELETE vorgenommen werden. Danach geben Sie bitte die Straße und Ihren Wohnort an. Wenn alle Eingaben richtig sind, können Sie die letzte Frage mit „J“ für „Ja“ beantworten, sonst geben Sie „N“ für „Nein“ ein — Sie erhalten dann die Möglichkeit, Ihre Eingaben noch einmal zu korrigieren.

Achtung: Geben Sie bitte wirklich Ihren korrekten Namen an. Wenn Sie einen falschen eingeben, haben Sie keinen Anspruch auf spätere Updates.

Sie müssen darauf achten, daß kein Dritter an Kopien von Ihrem OMIKRON. Assembler gelangen kann. Da Ihr Name im Programmcode steht (auch verschlüsselt), können wir bei jeder Raubkopie zurückverfolgen, wer sie in Umlauf gebracht hat. OMIKRON.Software behält sich gerichtliche Schritte gegen Raubkopierer vor.

Kapitel 4

Gemeinsames von Assembler und Debugger

4.1 Die Benutzeroberfläche

Die Benutzeroberfläche lehnt sich an GEM an, wurde aber vollkommen neu programmiert. Dies ist nötig, weil Abstürze von GEM-Programmen (die besonders durch Assemblerprogramme leicht zu erzeugen sind) häufig das gesamte GEM lahmlegen. Sie können mit dem OMIKRONAssembler jedoch auch bei zerstörtem GEM noch weiterarbeiten, bzw., was sicherlich besser ist: Ihren Sourcetext abspeichern¹, RESET drücken und das System neu starten.

4.2 Die Dialogboxen

Die Dialogboxen entsprechen denen des GEM, arbeiten aber auch vom Betriebssystem gänzlich unabhängig. Wie unter GEM gibt es Radio-Buttons (von denen immer nur einer angewählt (selected) ist) und Exit-Buttons. Mit letzteren wird ein Dialog verlassen. In ihnen stehen nur Großbuchstaben, damit sie sofort als Exit-Buttons erkannt werden können. Der Default-Button ist mit etwas breiterem Rahmen dargestellt.

¹Wenn das GEMdos beschädigt wurde, ist das eventuell nicht mehr möglich...

Tabelle 4.1: Die Tasten in Dialogboxen

RETURN	Betätigt den Default-Button.
↓ oder TAB ↑	Der Cursor springt zum nächsten Eingabefeld. Der Cursor springt ein Eingabefeld zurück.
CLR/HOME	Der Cursor springt zum ersten Eingabefeld.
ESC	Löscht das Eingabefeld, in dem der Cursor steht.
DELETE	Löscht das Zeichen unter dem Cursor.
BACKSPACE	Löscht das Zeichen links vom Cursor.
⇐ und ⇒	Bewegt den Cursor im Eingabefeld; Autoinsert ist immer an.
UNDO	Bricht die Eingabe ab. Alle Eingaben werden ungültig, kein Knopf wird gedrückt. Rücksprung in die Haupteingabe.
CONTROL-⇐	Springt an den Anfang des aktuellen Eingabefeldes.
CONTROL-⇒	Springt an das Ende des aktuellen Eingabefeldes.

Die Tasten mit besonderer Funktion in den Dialogboxen sind in Tabelle 4.1 aufgeführt.

Außerdem können Sie mit der linken Maustaste den Cursor in ein Eingabefeld setzen.

Oben links in jedem Button steht ein kleiner Buchstabe. Sie können einen Button auch anwählen, indem Sie ALT-Buchstabe drücken.

4.3 Die Environment-Variable

Falls die Environment-Variable „SIGMA“ definiert ist, versuchen sowohl der Assembler als auch der Debugger, dort ihre .INF-Dateien (beim Assembler auch die OM-ASSEM.CFG- und die OM-ASSEM.DAT-Datei) zu laden.

Es gibt verschiedene Möglichkeiten, die Environment-Variable zu setzen:

- unter NeoDesk, z. B. mit „SIGMA=D:\ASS\“
- unter einer Commandshell, z. B. unter der MISHELL in der Datei MISHELL.ENV
- mit dem Programm DESKTOP.PRG. Dieses Programm liegt dem Assembler bei; eine Kurzanleitung finden Sie im Anhang I.9 auf Seite 221. Die Environment-Variable können Sie im Sourcetext ändern. Das Programm muß als letztes im AUTO-Ordner liegen. Um die Programme in einem Ordner umzusortieren können Sie z. B. die MISHELL verwenden oder
 - alle Programme in einen anderen Ordner kopieren,
 - den ersten Ordner löschen,
 - ihn wieder neu anlegen
 - und die Programme in der gewünschten Reihenfolge zurückkopieren
 - (Danach natürlich den Zwischenordner wieder löschen)

4.4 Blitterunterstützung

Der OMIKRON Assembler benutzt zum Retten des Hintergrundes bei Drop-Down-Menüs, zum Scrollen u. ä. einen Blitter, wenn dieser eingeschaltet ist. Für den Fall, daß Ihr Blitter fehlerhaft arbeitet, kann er im Assembler abgeschaltet werden (siehe Kapitel 5.5.5 auf Seite 79).

Kapitel 5

Der Assembler

5.1 Zur Geschichte des Assemblers

Die Idee zum OMIKRON.Assembler ist etwa Ostern 1987 geboren worden, als wir (der Sören und ich, Markus) uns über die Assemblierzeiten von einigen Minuten bei den herkömmlichen Assemblern ärgerten. Es wurden einige Tage mit der Entwicklung des Konzepts verbracht und dann, tja, dann wurde das Projekt wegen zu komplizierter Strukturen wieder vergessen.

Bis Anfang 1988, dann bekam ich über einige Ecken das Angebot, für OMIKRON.Software einen Assembler zu entwickeln. Also setzten Sören und ich uns wieder zusammen und entwickelten in etwa zwei Monaten ein Konzept für einen tokenisierenden Assembler (welcher genaugenommen bei der Eingabe schon assembliert). Das Konzept stand, und wir steckten mitten im Abitur.

Sofort nach dem schriftlichen Abitur (kurz vor Ostern 1988) fingen wir an, zu programmieren. Sören versuchte sich an einem Editor, der bei der Eingabe bereits assembliert, und ich wandte mich einem Debugger zu, der für die Entwicklung dringend gebraucht wurde. Als der Debugger halbwegs funktionierte, kam die Benutzeroberfläche dran (eigene Dialogboxen, I/O-Routinen (erstmals auch mit eigenem Tastaturtreiber) und allem, was dazugehört). Daraus entstand die V0.1-Version des Debuggers, die vielleicht einigen als PD-Programm aus einer Hamburger Mailbox schon bekannt ist.

Erst nachdem die I/O-Routinen in den Assembler eingebaut waren (und funktionierten (stöhn)), gingen wir dazu über, die eigentliche Assemblierung zu schreiben. Unser Ziel war es, eine Geschwindigkeit von ca. 250 000 Zeilen pro Minute zu schaffen. Wie sich allerdings herausstellte, waren diese Schätzungen „etwas“ zu niedrig, der Assembler schaffte ca. 2 000 000 Zeilen pro Minute. Die Geschwindigkeit wurde durch nötige Abfragen (Reicht der Speicherplatz noch?) zwar noch etwas geringer, aber sie blieb doch hinreichend hoch.

Zu diesem Zeitpunkt verpasste Artur Södler (seines Zeichens Programmierer von OMIKRON.BASIC) unserem Assembler neue Zeichenausgaberroutinen, welche die Geschwindigkeit von 3 Seiten pro Sekunde auf über 10 beim seitenweisen Blättern brachte. Damit war die Darstellung des Assemblers ausreichend schnell, fehlte nur noch ein vernünftiger Editor (Block-Befehle, Suchen und Ersetzen, ...). Während Sören den Rest des Jahres mit solchen „Kleinigkeiten“ verbrachte, versuchte ich, den Debugger zu verbessern, indem ich fast alle Funktionen implementierte, die mir vorgeschlagen wurden. Besonders lange haben Sören und ich jedoch an der Benutzeroberfläche gefeilt; damit alle Funktionen, die häufig gebraucht werden einfach aufgerufen werden können. Das fängt mit Symbolsuche an und hört damit auf, daß man seine Einstellungen abspeichern kann. Dann haben wir nochmals alles auf Geschwindigkeit optimiert.

Weil wir bei der Eingabe bereits assemblieren, kam uns die Idee, daß es doch möglich sein müßte, den PC des Debuggers in eine Zeilennummer zurückzurechnen. Gesagt, getan, Sören machte das Unmögliche möglich und implementierte eine Umrechnung von PC in Zeilennummer und umgekehrt. Er setzte dem ganzen noch die Krone auf, als er auf die Idee kam, die Marker auch noch umzurechnen. Etwa 8 Stunden später (es muß so 5 Uhr Samstagmorgen gewesen sein) war die Schnittstelle fertig (sie wurde im Laufe der Zeit allerdings mehrfach erweitert und überarbeitet). Die Möglichkeit, auf einen Tastendruck ein Programm zu assemblieren, das Programm dem Debugger zu übergeben und zu starten, ist wohl einmalig und war vorher höchstens bei Interpretern vorhanden. Der Rücksprung in den Assembler geschieht bei Programmende automatisch oder bei Programmabbruch mit SHIFT-SHIFT an die entsprechende Stelle (Option ist ausschaltbar) — und das ohne nennenswerte Verzögerung.

So kommen wir also zur neuesten Version des Assemblers, welche seit neue-

stem endlich auch eigene Drop-Down-Menüs enthält (die sich fast genauso bedienen lassen, wie die Drop-Down-Menüs des GEM). Dies ermöglicht es uns, einfacher neue Funktionen zu implementieren, als dies bei 20 Buttons (Funktions-Tasten) möglich wäre.

Abschließend möchte ich mich bei allen unseren unermüdlichen Testern nochmals bedanken, welche nicht nur Fehler gefunden, sondern auch viele konstruktive Verbesserungsvorschläge gemacht haben.

Markus Fritze

5.2 Einführung in den Assembler

Der OMIKRONAssembler ist ein integriertes Entwicklungspaket für Assemblerprogramme, d. h., es gibt keinen separaten Assembler; der Assembler ist statt dessen im Editor mit integriert. Dies ermöglicht eine viel bessere Zusammenarbeit von Assembler und Editor — man denke nur an die Möglichkeit, mit CONTROL-J (siehe Kapitel 5.5.5 auf Seite 56) in eine fehlerhafte Zeile zu springen. Ganz nebenbei wird die Entwicklungszeit natürlich drastisch verkürzt.

Wie fange ich an? Zuerst sollten Sie sich eine Diskette mit den nötigen Programmen zusammenstellen. Harddisk-Besitzer reservieren sich dafür einen Ordner. Damit stellt sich gleich die nächste Frage:

Welche Programme sind für die Entwicklung nötig? Bei genügendem Speicher (1 MB oder mehr) empfiehlt es sich, den Debugger in den AUTO-Ordner zu legen, er wird damit automatisch resident und kann direkt vom Assembler oder vom Programm CALL.PRG aufgerufen werden. Ansonsten kann er jederzeit per Doppelklick vom Desktop aus gestartet werden.

Beispieldiskette:

AUTO\OM-DEBUG.PRG — Der Debugger liegt somit stets resident im RAM
OM-ASSEM.PRG — Der Assembler und Editor (nicht ganz unwichtig)

OM-ASSEM.CFG	— Die Druckeranpassung (siehe Kapitel D.5 auf Seite 194)
OM-ASSEM.DAT	— Eine Hilfsdatei für den Assembler (<i>wichtig!</i>)
OM-ASSEM.INF	— Die individuellen Einstellungen des Assemblers
OM-DEBUG.INF	— Die individuellen Einstellungen des Debuggers
CALL .PRG	— Den residenten Debugger vom Desktop starten

Alle Hilfsdateien (CFG, DAT und INF) können als hidden-Dateien vorliegen. Mit dem Befehl FATTRIBUT (siehe Kapitel 6.6.6 auf Seite 149) des Debuggers ist es ohne Probleme möglich, die Dateien als hidden zu deklarieren. Sie stören dann das Erscheinungsbild im Desktop nicht mehr.

OM-ASSEM.CFG wie auch OM-ASSEM.DAT werden vom Assembler nach folgendem Schema gesucht:

1. Wenn die Environment-Variable „SIGMA“ vorhanden ist, wird in diesem Pfad zuerst gesucht (siehe Kapitel 4.3 auf Seite 22)
2. Es wird im aktuellen Verzeichnis gesucht.
3. Es wird im Hauptverzeichnis gesucht.
4. Es werden alle Laufwerke von C bis P in deren aktuellen Pfaden durchsucht.
5. Es wird auf Laufwerk A gesucht.

OM-ASSEM.INF und OM-DEBUG.INF können vom Assembler bzw. Debugger aus erzeugt werden (siehe Kapitel 5.5.5 auf Seite 80 bzw. Kapitel 6.8 auf Seite 168).

Wie starte ich den Assembler, wenn ich eine Systemdiskette erstellt habe? Das geht an sich ganz einfach: Mit einem Doppelklick auf OM-ASSEM.PRG wird der Assembler (mit integriertem Editor) geladen. Man kann dann sofort loslegen. Allerdings kann ich nur empfehlen (wenn mindestens 1 MB vorhanden ist), einen residenten Debugger zu benutzen, da man erst dann die volle Leistungsfähigkeit des Assemblers ausschöpfen kann.

Was ist, wenn ich nur 512 KB RAM habe? Das macht auch nichts. Sie müssen dann allerdings ohne residenten Debugger arbeiten, indem Sie nach der Assemblierung Ihr erzeugtes Programm abspeichern, den Assembler verlassen und dann vom geladenen Debugger aus Ihr Programm erneut laden. Eine Symboleitabelle können Sie dabei natürlich auch übergeben.

Wie läuft eine Entwicklung ab? Beispielablauf einer Entwicklung mit residentem Debugger:

1. Den Assembler laden.
2. Den Sourcetext laden (Menüpunkt „Datei⇒Öffnen...“) oder Source- text eingeben (Als „Gerüst“ für ein Programm können Sie die Datei HEADER.S verwenden (siehe Kapitel I.15 auf Seite 224)).
3. Assemblieren anwählen (Mit dem Menüpunkt „Assemblieren⇒ Assem- blieren...“).
4. In der Dialogbox nach dem Assemblieren bei Symboleitabelle „normal“ anwählen.
5. Den Button „DEBUGGER“ anklicken.
6. Das Programm mit „GO“ starten (z. B. SHIFT-F4 oder GO eintippen und RETURN drücken).
7. Nach einem Abbruch mit SHIFT-SHIFT, oder wenn das Programm abstürzt, CONTROL-HELP drücken (⇒ Rücksprung in den Assembler).
8. Der Cursor steht an der fehlerhaften Stelle bzw. an der Abbruchstelle.
9. weiter bei 3...

Wahrscheinlich werden Sie die Befehle und Tastenkombinationen noch nicht kennen, aber das macht nichts, sie werden später noch einmal ausführlich erklärt. Das Beispiel soll nur zeigen, wie eine Entwicklung ungefähr ablaufen kann. Sie können dies mit einem der Sourcetexte aus dem DEMOS.OMI-Ordner nachvollziehen (z. B. AES_DEMO.SRC). Wenn es noch nicht klappt, macht das

auch nichts, denn schließlich soll diese Anleitung Sie ja in die Benutzung einführen.

Kommen wir nun zur genaueren Betrachtung des OMIKRON Assemblers.

5.3 Die Drop-Down-Menüs

Sie verhalten sich fast genauso, wie die Drop-Down-Menüs des GEM. Es gibt nur einen kleinen Unterschied: Falls Sie einen Eintrag aus Versehen berühren und ein Menü herunterklappt, brauchen Sie nur die rechte Maustaste zu drücken, und schon ist das Menü wieder eingeklappt¹. Eine Erklärung der einzelnen Menüpunkte folgt in der Erklärung des Editors (siehe Kapitel 5.5.5 auf Seite 45).

5.4 Die Funktionen der Maus

Die linke Maustaste: Mit der Maus können Sie den Cursor positionieren, indem Sie über der neuen Position kurz die linke Maustaste drücken.

Sie können mit der Maus auch einen Block markieren, indem Sie die die Maus mit gedrückter linker Taste nach oben bzw. unten bewegen. Wenn Sie den Bildschirmrand erreichen, fängt der Sourcetext an zu scrollen.

Mit einem Doppelklick mit der linken Taste auf ein Symbol führt der Editor einen Sprung zu der Zeile aus, in der das Symbol definiert wird. Wenn der Cursor bereits vor dem Doppelklick auf der Symboldefinition stand, wird das Symbols in die Funktion „Symbol suchen...“ übergeben, und die entsprechende Dialogbox erscheint (siehe Kapitel 5.5.5 auf Seite 67). Das bedeutet praktisch folgendes: Wenn Sie auf einen Symbolaufruf klicken, springen Sie zur Definition; wenn Sie auf eine Definition klicken, können Sie über die Dialogbox die Aufrufe suchen.

Die rechte Maustaste: Man kann auch mit der rechten Taste doppelklicken: Dann wird der Rechner aufgerufen und die Formel, auf die geklickt

¹Genauso verhält es sich in OMIKRON.BASIC.

wurde, in den Rechner übernommen (siehe Kapitel 5.5.5 auf Seite 62). Von dort kann dann die Formel nach dem Rechnen und eventuellen Veränderungen zurückkopiert werden.

Natürlich können Sie auch mit der Maus scrollen, ohne dabei einen Block zu markieren. Dazu drücken Sie die rechte Maustaste und bewegen die Maus zum oberen oder unteren Bildschirmrand. Während des Scrollings können Sie dann die Maustaste wieder loslassen: Der Editor stoppt erst dann das Scrollen, wenn er den Anfang oder das Ende des Sourcetextes erreicht hat, oder Sie die Maus wieder in Richtung Bildschirmmitte bewegen.

Wenn Sie den Cursor erst auf die Formel setzen (z. B. durch Linksklick) und dann auf den Cursor doppelklicken (mit rechts), wird die Formel auch in den Rechner übernommen. Wenn Sie dann den Rechner wieder verlassen und die Formel einsetzen lassen, wird sie nicht an der Cursorposition eingefügt, sondern durch das Ergebnis ersetzt (am besten, Sie probieren es mal aus!). Das Ersetzen klappt nur richtig in einem Einfügemodus (s. u.); beim Überschreibmodus können Formelreste stehenbleiben oder andere Textteile überschrieben werden.

Die Funktion der Statuszeile²: In dieser Zeile wird am linken Rand die Cursorposition im Sourcetext angegeben, und zwar erst die aktuelle Zeile „Z“ und dann die aktuelle Spalte „S“.

Ein Klick auf das „Z“ von „Zeile“ bringt sie abwechselnd an den Anfang bzw. das Ende des Sourcetextes. Ein Linksklick auf die Zeilennummer lässt eine Dialogbox erscheinen, in der Sie entweder eine Zeilennummer oder ein Symbol angeben können, zu der/dem gesprungen werden soll (siehe auch ALT-Z, die Tastaturkommandos sind im Kapitel 5.5.4 ab Seite 40 beschrieben). Ein Rechtsklick auf die Zeilennummer bringt Sie zur letzten geänderten Stelle (siehe auch CONTROL-Z).

Ein Linksklick auf die Spaltennummer setzt den Cursor an den Zeilenanfang (siehe auch CONTROL-←), ein Rechtsklick auf das Zeilenende (siehe auch CONTROL-→).

In der Mitte der Zeile steht entweder der Filename des aktuellen Programms oder eine Fehlermeldung, falls der Cursor in einer fehlerhaften Zeile steht.

²Das ist die Zeile, die direkt unter der Menüleiste steht

Daß ein Block markiert ist, sehen Sie an einem „B“ rechts oben. Wenn Sie mit der linken Maustaste auf dieses „B“ klicken, wird der Cursor an den Blockanfang, mit der rechten Taste an das Blockende gesetzt.

Zwischen dem „B“ und dem Einfügemodus kann ein „C“ stehen, falls CAPS LOCK aktiviert ist. Mit einem Mausklick dorthin kann man CAPS LOCK ein- bzw. ausschalten.

Ganz rechts steht der aktuelle Einfügemodus. Es gibt drei verschiedene Einfügemodi im OMIKRON Assembler:

Überschreiben: Eingegebener Text überschreibt den vorhandenen. Mit der Taste INSERT kann eingefügt werden.

Einfügen I: Eingegebener Text wird vor dem Cursor eingeschoben.

Einfügen II: Wie Einfügen I, jedoch wird nach RETURN auch gleich eine Zeile eingefügt.

Durch Anklicken dieses Textes können Sie diesen Modus ändern (siehe auch SHIFT-INSERT auf Seite 43)

Über dem Einfügemodus (links neben der Uhrzeit) sehen Sie den ASCII-Wert (dezimal) des Zeichens unter dem Cursor. Falls Sie diese Anzeige stört, kann sie abgeschaltet werden, indem Sie einmal auf sie klicken. Durch einen weiteren Klick auf diese (dann leere) Stelle kann sie wieder aktiviert werden.

5.5 Der Editor

Das Beste am Editor ist sicherlich, daß er in der Lage ist, einen Sourcetext automatisch zu formatieren: Sie können Ihre Programme unformatiert eingeben, und der Editor ist in der Lage, Symbole, Opcodes und Operanden voneinander zu unterscheiden. Nachdem Sie RETURN gedrückt haben, gibt er die Zeile formatiert aus.

Beispiele:

Eingabe: start move #5,d0;Remark

Ausgabe: start: move.w #5,d0 ;Remark

Alle unformatierte eingegebenen (oder als ASCII-Text eingelesenen) Texte werden also automatisch übersichtlich formatiert.

Normalerweise braucht der Assembler keine Doppelpunkte hinter Symboldefinitionen. Es gibt aber eine Ausnahme — wenn Sie unbedingt Opcodes als Symbolnamen verwenden wollen:

Eingabe: nop moveq #0,d0
Ausgabe: Syntax Fehler,
(da der Befehl "NOP" keine Parameter erlaubt)

abert:

Eingabe: nop:moveq #0,d0
Ausgabe: nop: moveq #0,d0

Opcodes sind also nur dann als Symbolnamen erlaubt, wenn man selbst dahinter einen Doppelpunkt setzt.

5.5.1 Besonderheiten des Assemblers

Wenn .w hinter einem Symbolreferenz (nicht hinter der Definition) steht, schreibt der Assembler die Adresse auf Wordbreite (das spart Speicherplatz und Rechenzeit). Das ist nur bei Adressen bis \$7FFF und Adresse größer gleich \$FFFF8800 möglich. So können Sie z. B. „move etv.timer.w, d0“ angeben. Dies ist bei relozierbaren Adressen nicht möglich, so daß also nur Systemvariablen so adressiert werden können.

Line-A-Routinen können direkt mit „LINEA #“ und der entsprechenden Nummer eingeben. Dahinter gibt der Assembler in eckigen Klammern automatisch den Funktionsnamen an, also z. B. „LINEA #0 [Init]“. Die eckigen Klammern werden vom Assembler in diesem Fall ignoriert.

Konvertierungen in den Motorola-Standard

Die Befehle werden, wie schon erwähnt, sofort vorassembliert und nicht als ASCII-Text gespeichert. Daher werden Sie gleich bei der Eingabe auf Syntax- und ähnliche Fehler hingewiesen. Die Zeile wird gleich wieder disassembliert und neu ausgegeben. Daher kann Ihre Zeile vom Assembler anders angezeigt werden, als Sie sie eingegeben haben. Ein Beispiel: Sie geben ein „move.l d3,a2“³. Laut Motorola-Standard ist diese Schreibweise nicht richtig: Wenn Sie etwas in ein Adreßregister schreiben, heißt der Befehl „movea“. Der Assembler korrigiert die Schreibweise dann automatisch — dabei ist er noch ziemlich tolerant; er konvertiert beispielsweise „movea“ zurück nach „move“ (für den Fall, daß Sie ein Adreßregister in ein Datenregister ändern). Die genauen Konvertierungen sind in Tabelle 5.1 angegeben.

Bei Conditions (B??, DB?? und S??) wird „Z“ (Zero) zu „EQ“ (EQual) und „NZ“ (Not Zero) zu „NE“ (Not Equal) konvertiert. „DBF“ wird zu „DBRA“ gewandelt.

Statt „CC“ und „CS“ können Sie auch „HS“ (Higher or Same) bzw. „LO“ (Lower) angeben. Dies wird nicht in den Standard konvertiert — es bleiben eventuell zwei unterschiedliche Befehle stehen, die aber den gleichen Code produzieren. *Achtung:* Dieses gilt nur für Branch- und Set-Befehle (also BHS, SLO usw.), DBHS und DBLO wird in DBCC bzw. DBCS.

Als Tipperleichterung können Sie das „#“ bei TRAP- und MOVEQ-Befehlen weglassen.

Wir haben uns bemüht, möglichst alle Formate von allen uns bekannten Assemblern zu konvertieren. Deshalb wird z. B. auch „LEA.L Adresse,Ax“ oder „LEA.W Adresse,Ax“ in „LEA xx,Ax“ gewandelt.

Lokale Label

Da der Editor von Grund- und Makroversion gleich ist, können Sie auch in der Grundversion lokale Label eingeben — diese werden mit einem Punkt am Anfang gekennzeichnet. Da der Assembler diese aber noch *nicht* richtig

³wobei <ea> nicht Dn sein darf

⁴Um das Zielregister einfach z. B. A0 in D7 ändern zu können

Tabelle 5.1: Konvertierungen in den Motorola-Standard

eingegebener Befehl:	Motorola-Befehl
moveq data,d0	moveq #data,d0
trap data	trap #data
cmp (ax)+,(ay)+	cmpm (ax)+,(ay)+
add #data,ea	addi #data,ea
and #data,ea	andi #data,ea
cmp #data,ea ³	cmpl #data,ea
eor #data,ea	eorl #data,ea
or #data,ea	orl #data,ea
sub #data,ea	subl #data,ea
add ea,An	adda ea,An
cmp ea,An	cmpa ea,An
sub ea,An	suba ea,An
move ea,An	movea ea,An
adda ea,Dn ⁴	add ea,Dn
cmpa ea,Dn ⁵	cmp ea,Dn
suba ea,Dn ⁵	sub ea,Dn
movea ea,Dn ⁵	move ea,Dn
clr An	suba An,An
xor	eor
asl/r Dn	asl/r #1,Dn
lsl/r Dn	lsl/r #1,Dn
rol/r Dn	rol/r #1,Dn
roxl/r Dn	roxl/r #1,Dn

verarbeiten kann, sollten Sie dies vermeiden! Symbole dürfen also nie mit einem Punkt beginnen.

Optimierungen

Der OMIKRON.Assembler beherrscht „halb-“automatische Optimierungen. Wenn bei der Assemblierung Optimierungsmöglichkeiten erkannt werden, werden diese mit entsprechenden Warnmeldungen versehen (siehe Kapitel 5.7.1 auf Seite 100). Über dem „ABBRUCH“-Button erscheint ein Button „ANPASSEN“. Wenn Sie darauf klicken, wird der Dialog beendet und alle Zeilen mit entsprechenden Warnmeldungen angepaßt. Dann müssen Sie nochmal neu Assemblieren. Eventuell müssen Sie diese Prozedur noch einmal wiederholen: Ein JMP kann in einem Durchgang zu einem BRA optimiert, in einem zweiten kann dieser zu einem BRA.S geändert werden (wenn der Sprung dadurch nicht zu weit wird, natürlich).

Dieses Verfahren mag jemandem im Vergleich zu anderen Assemblern etwas umständlich vorkommen, weil diese bei der Assemblierung gleich mit optimieren können. Durch unser Konzept ist dieses jedoch nicht möglich. Der Assembler wird durch die Optimierungsabfragen aber praktisch nicht langsamer, und das Doppelt-Assemblieren beim OMIKRON.Assembler geht wohl immer noch wesentlich schneller als das Einmal-Assemblieren bei einem anderen.

5.5.2 Formelrechnungen

Der OMIKRON.Assembler erlaubt, mit 16 verschiedenen Rechenoperationen zu arbeiten. Dabei handelt es sich um die Grundrechenarten, die booleschen Algebra sowie Vergleiche. Alle Operationen werden mit 32 Bit Genauigkeit ausgeführt. Er beherrscht natürlich Punkt-vor-Strich-Rechnung.

Fangen wir mit den Vorzeichen an:

+	Plus (wer hätte es gedacht)
-	Minus (auch nicht schlecht) (2er Komplement)
~	NOT (1er Komplement)
!	logisches NOT (Ergebnis ist 0 oder -1)

Der Assembler kann mit vier verschiedenen Zahlenbasen rechnen, die da wären:

.	bzw. nichts	Dezimal (Zahlenbasis 10)
\$		Hexadezimal (Zahlenbasis 16)
%		Binär (Zahlenbasis 2)
"oder '		ASCII (Zahlenbasis 256)

Man kann z. B. schreiben "A"*2-\$10+%11011-"soft"

ASCII-Zahlen dürfen (natürlich) nur aus maximal 4 Buchstaben bestehen, um im 32-Bit-Zahlenbereich zu bleiben. Die vorhandenen Rechenoperationen stehen in Tabelle 5.2.

Wenn ein Vergleich wahr ist, lautet das Ergebnis „TRUE“ (-1), sonst ist es „FALSE“ (0).

Der Assembler kennt dazu noch einige Systemvariablen, welche (bis auf *) alle durch „^“ am Anfang gekennzeichnet sind:

- * dient für relative Sprünge im Programm, ohne ein Label zu verwenden. Das spart Platz in der Symboltabelle und Assemblierzeit (da Ausdrücke wie „*+Wert“ sofort berechnet werden) — obwohl das beim OMIKRONAssembler wohl kaum ins Gewicht fällt. „JMP *+100“ springt beispielsweise 100 Byte hinter den Anfang des JMP-Befehls.
- ^^DATE ergibt das aktuelle Datum im TOS-Format (Bit 0-4: Tag; Bit 5-8: Monat; Bit 9-15: Jahr-1980). Damit ist es ohne Probleme möglich, das Datum in seine Programme einzubinden. Es kann dann von Ihrem Programm ausgegeben werden, und Sie sehen sofort, wann diese Version Ihres Programms erstellt wurde. So wissen Sie immer, ob Sie eine aktuelle Programmversion vor sich haben.
- ^^TIME liefert entsprechend die Uhrzeit im TOS-Format (Bit 0-4: Sekunden; Bit 5-10: Minuten; Bit 11-15: Stunden)

⁵Vergleiche sind im Taschenrechner nicht vorhanden

Tabelle 5.2: Die Rechenoperationen des Assemblers

Rechenzeichen	Beschreibung:
+	Addition
-	Subtraktion
	OR
-	EOR (XOR)
<<	Linksverschiebung (LSL-Befehl des Prozessors)
>>	Rechtsverschiebung (LSR-Befehl des Prozessors)
*	Multiplikation
/	Division
&	AND
%	Modulo
Vergleiche ⁵	
=	Gleichheit
<	Kleiner als
>	Größer als
<=	Kleiner gleich
>=	Größer gleich
<>	Ungleich

- ``RSCOUNT** ergibt den aktuellen Stand des RS-Counters (siehe bei der Erklärung der RS-Pseudo-Opcodes (siehe Kapitel 5.6.2 auf Seite 85)).
- ``SYMTAB** enthält einen Wert ungleich 0, wenn mit dem Pseudo-Opcode OPT (siehe Kapitel 5.6.5 auf Seite 89) eine Symboltabelle gewünscht wurde. Die ermöglicht z. B. folgende Konstruktion:

```
IF    ``SYMTAB
BRA  Debugging_input ;Während der Testphase
ELSE
BRA  Normal_input ;Wenn alles läuft
ENDC
```

Systemvariablen werden immer groß geschrieben. Näheres zur bedingten Assemblierung finden Sie im Kapitel 5.6.8 ab Seite 97.

5.5.3 Symbole, Label und Konstanten

Bei Symbol- und Labelnamen ist man folgenden „Einschränkungen“ unterworfen:

Ein Symbol darf aus maximal 23 Zeichen bestehen, wobei alle Zeichen signifikant sind, d. h., der Assembler unterscheidet Symbole bis zum 23. Zeichen voneinander (Bei anderen Assemblern sind nur die ersten 8 Buchstaben signifikant!).

Was (soweit uns bekannt ist) noch kein anderer Assembler erlaubt, aber gerade im deutschen Sprachraum nützlich ist: Der Assembler erlaubt auch Umlaute in Symbolnamen (z. B. „menüpunkt: EQU 1“). Ein Symbol darf beim OMIKRON.Assembler folgende Zeichen enthalten:

- Als erste Zeichen sind erlaubt: „A“-„Z“, „a“-„z“, „@_{}`“, sowie alle Zeichen mit einem ASCII-Code > 126 (Ausnahme: 255). In der Grundversion ist kein Punkt als erstes Zeichen erlaubt, da dieser als Kennzeichnung für lokale Label in der Makro-Version dient.

- Als Folgezeichen sind erlaubt: Alle oben angeführten Zeichen sowie „\$#“ und „0“-„9“.
- Als Anfangs- und Folgezeichen kann man auch alle Tasten mit einem ASCII-Code < 32 mit Ausnahme von 0, 10, 13 und 27 verwenden. Ob man diese Möglichkeit ausnutzt, sollte man allerdings gründlich überdenken: Die Symbole sind nicht nur schlecht einzugeben (über ALTERNATE-Zehnerblock), sie bereiten auch Probleme bei ASCII-Editoren, falls man einen Sourcetext im ASCII-Format abspeichert.

Der Assembler unterscheidet normalerweise Groß- und Kleinbuchstaben. Das bedeutet, daß die Symbole „Test“, „test“ oder „TEST“ drei unterschiedliche Symbole sind. Wer einen alten Sourcetext hat, der das Symbol „test“ und „Test“ für die gleiche Funktion benutzt, kann im Menü „Editor 2“ (siehe Kapitel 5.5.5 auf Seite 77) eine automatische Wandlung der eingegebenen Symbole in Groß- oder Kleinbuchstaben einschalten. Der Assembler wandelt dann bereits bei der Übernahme einer Zeile alle Symbole, wobei natürlich auch Umlaute gewandelt werden.

Rechnen mit Symbolen

Es gibt ja zwei Arten von Symbolen, zum einen relozierbare und zum anderen absolute Symbole. Wir haben auf eine sprachliche Unterscheidung beider Symboltypen in der Anleitung geachtet, damit es nicht zu Verwechslungen kommt: Konstanten sind bei uns absolute Symbole; d. h., Symbole, welchen mit dem Pseudo-Opcode EQU oder SET ein Wert zugewiesen wurde.

Beispiel:

```
wert: EQU 100
wert2: EQU wert*10+1 ;Da wert eine
           ;Konstante ist, ist
           ;wert2 auch eine.
wert1: RS.W 10          ;Eine Konstante, da RS
           ;fast so wie
           ;EQU funktioniert.
```

Label sind relozierbare Symbole, welche Adressen darstellen, die evtl. vom GEMDOS beim Laden reloziert werden müssen.

Beispiel:

```
label: NOP           ;Ein Symbol vor einem Opcode
label2: EQU label+100 ;Da label ein Label ist, ist label2
                  ;auch eins.
label3: DS.W 10      ;Auch ein Label
```

aber:

```
wert3: EQU label2-label ;Zwei Label voneinander subtrahiert
                  ;ergeben eine Konstante
wert4: EQU label2*label ;Das geht NICHT !!!
wert5: EQU label+label2 ;Das geht AUCH NICHT!!!
label3: EQU label+label2-label3 ;Das geht wieder, da
                  ;label2-label3 ein Offset ist,
                  ;welcher zu label addiert werden
                  ;kann!
```

Noch einige Erklärungen zu Remarks: Bei anderen Assemblern (z.B. dem Assembler des Entwicklungspaketes) beginnen Remarks stets mit einem „*“. Da aber der „*“ auch ein Rechenzeichen ist, bricht der Assembler mit einer Formel ab, wenn er auf ein Space trifft. Allerdings beginnt dann *nicht* automatisch ein Remark, sondern der Assembler erwartet vor einem Remark *zwingend* entweder ein „;“ oder ein „*“.

```
Beispiel: move.w #0,wert*2 *2
Ergebnis:      move.w #0,wert*2          *2

aber:      move.w #0,wert*2;2
Ergebnis:      move.w #0,wert*2          ;2
```

5.5.4 Tastaturkommandos

Der OMIKRON Assembler unterstützt eine Vielzahl von Tastaturkommandos. Die nun folgenden Kommandos können nicht über die Drop-Down-Menüs erreicht werden, da diese dadurch viel zu lang und unübersichtlich würden:

Auf ein oder zwei Zeilen bezogene Kommandos

BACKSPACE	Löscht das Zeichen links vom Cursor
DELETE	Löscht das Zeichen unter dem Cursor
CONTROL-T	Löscht das Wort unter dem Cursor (ähnlich bei TEMPUS)
CONTROL-SHIFT-T	Löscht ab dem Cursor den Zeilenrest
ALT-<	Löscht ab dem Cursor den Zeilenrest
INSERT	Fügt ein Space an der Cursorposition ein
TAB	Springt zur nächsten Tabulatorposition (Im Darstellungsmenü einstellbar). Dabei werden Spaces bis zu der nächsten Tabulatorposition eingefügt
SHIFT-TAB	Springt zur nächsten Tabulatorposition, ohne dabei Spaces einzufügen
CONTROL->	Sprung ans Zeilenende
SHIFT->	Sprung ans Zeilenende
CONTROL-←	Sprung an den Zeilenanfang
SHIFT-←	Sprung an den Zeilenanfang
CONTROL-DELETE	Löscht die aktuelle Zeile
CONTROL-Y	Löscht die aktuelle Zeile

CONTROL-INSERT	Fügt eine Leerzeile ein, die Cursorzeile rückt nach unten, und der Cursor steht in der eingefügten leeren Zeile
CONTROL-D	Dupliziert die Zeile, in der der Cursor steht. Dabei werden Remarks und Labeldefinitionen nicht dupliziert, da es sich in der Praxis gezeigt hat, daß dies gewöhnlich nicht sinnvoll ist (doppelte Labeldeklaration etc.)
CONTROL-M	Übernimmt eine Zeile in einen internen Buffer und löscht sie
ALT-M	Gibt den internen Buffer wieder aus. Das ist beliebig häufig möglich. In Verbindung mit CONTROL-M kann man sehr einfach Zeilen verschieben bzw. kopieren
SHIFT-DELETE	Verkettet zwei Zeilen (wenn syntaktisch möglich). Der Befehl erlaubt es beispielsweise, folgende Zeilen zu verketteten: ;Remark label: move.w #1000,d0
	Wenn Sie zweimal SHIFT-DELETE auf der Remarkzeile drücken, ergibt sich folgende Zeile: label: move.w \#1000,d0 ;Remark
SHIFT-RETURN	bewirkt den Gegensatz zu SHIFT-DELETE; mit dieser Kombination kann man eine Zeile auseinander trennen. Wenn man im obigen Beispiel zwischen „label“ und „move“ SHIFT-RETURN drückt, erhält man wieder zwei Zeilen. Bei DC-Zeilen (siehe bei den Pseudo-Opcodes) ist dies <i>nicht</i> möglich, hier hilft nur CONTROL-D und Löschen des Zeilenrestes mit ALT-<

Cursorbewegungen im Sourcetext

↑, ↓, →, ←	Bewegt den Cursor (wer hätte das gedacht?)
SHIFT-↑	Scrollt unabhängig von der Cursorposition nach oben
SHIFT-CONTROL-↑	Scrollt nach oben, Cursor bewegt sich mit (wie beim Mausscrolling)
CONTROL-↑	oder CONTROL-R Blättert eine Seite nach oben
ALT-↑	Blättert 10 Seiten nach oben
SHIFT-↓	Scrollt unabhängig von der Cursorposition nach unten
SHIFT-CONTROL-↓	Scrollt nach unten, Cursor bewegt sich mit (wie beim Mausscrolling)
CONTROL-↓	oder CONTROL-C Blättert eine Seite nach unten
ALT-↓	Blättert 10 Seiten nach unten

Einfache Suchkommandos bzw. Sprünge im Sourcetext

CLR/HOME	Springt abwechselnd zum Anfang bzw. Ende der Seite
SHIFT-CLR/HOME	oder ALT-T Springt abwechselnd zum Anfang bzw. Ende des Sourcetextes
ALT-E	Zum Ende des Sourcetexts springen
CONTROL-Z	Zur Position der letzten Änderung springen
ALT-SHIFT-↑	Springt zur vorigen Labeldefinition (aufwärts)
ALT-SHIFT-↓	Springt zur nächsten Labeldefinition (abwärts)

Marker: Der OMIKRON.Assembler kann sich maximal 10 frei definierbare Zeilenmarken merken, zusätzlich merkt er sich die Position des TEXT-, DATA-Segments und BSS selbständig.

ALT-ß	Sprung zum Anfang des TEXT-Segments
ALT-'	Sprung zum Anfang des DATA-Segments
ALT-#	Sprung zum Anfang des Block-Storage-Segments (BSS)
CONTROL-1 bis CONTROL-0 Merkt sich die aktuelle Zeile	
ALT-1 bis ALT-0	Sprung zu einer gemerkten Zeile
CONTROL-SHIFT-1 bis CONTROL-SHIFT-0	Löscht eine Positions-markierung. Dies ist meist unnötig, aber bei Sourcetexten von 20000 Zeilen und mehr, mit 10 Mar-kern, kann das Umrechnen der Marker und der PCs in eine Zeilennummer (Rücksprung vom Debugger in den Assembler) schon mal 1-2 Sekunden dauern; wem das zu lang ist (Zeit ist Geld), der kann die überflüssigen Markierungen löschen (siehe Kapitel 5.8 auf Seite 105).
Achtung:	Bei den obigen Funktionen sind die Zif-fern oberhalb des Alphateils gemeint, der Zehner-block wird für eine andere Funktion benötigt (s. u.)
SHIFT-INSERT	Schaltet die drei Editiermodi um (Überschreiben, Einfügen I/II) (siehe Kapitel 5.4 auf Seite 30)
CONTROL-W	Wandelt die Zahlenbasis. Falls der Cursor auf einer Zahl steht, kann deren Zahlenbasis mit CONTROL-W geändert werden (Dezimal / Hexadezimal / Binär / "ASCII" / 'ASCII'). Dies geht auch in Formeln; in DC-Zeilen (siehe Pseudo-Opcodes) kann, durch das interne Format bedingt, teilweise nur die ganze Zeile (oder Teile von ihr) gewandelt werden. Im

Zweifelsfall sollte man es ausprobieren, dadurch können jedenfalls keine Daten zerstört werden.

CONTROL-G

Löst eine sogenannte Garbage-Collection aus. Es werden intern überflüssige Symboleinträge wieder freigegeben. Diese Funktion ist normalerweise unnötig, da diese Funktion vor dem Speichern und beim Anzeigen des Sourcetextinfos (SHIFT-F4) automatisch aufgerufen wird. Wer jedoch mehrfach hintereinander „Symbol ersetzen“ aufruft und anschließend Warnungen zu Symbolen erhält, welche man, weil gerade ersetzt, gar nicht mehr im Sourcetext hat, sollte Abbrechen und CONTROL-G drücken (bei „Symbol ersetzen“ können freie Symboleinträge nicht automatisch erkannt werden)

CONTROL-U

Markiert alle Zeilen als fehlerhaft, in welchen ein Symbol definiert wird, das sonst nirgends im Programm verwendet wird. Damit lassen sich nun einfach alle Unterprogramme finden, die nicht benutzt werden. Sie können die „fehlerhaften“ Zeilen wie gewohnt mit CONTROL-J anspringen.

ALT-Zehnerblock

Man kann mit ALTERNATE und Ziffern auf dem Zehnerblock einen ASCII-Code eingeben (so wie bei IBM-kompatiblen). Ein Beispiel: ALTERNATE drücken (gedrückt halten) und 228 auf dem Zehnerblock eintippen, dann ALTERNATE loslassen. Es erscheint ein Σ an der Cursorposition. Weil die Kombination ALT-Zehnerblock im Tastaturtreiber verankert ist, funktioniert diese Methode auch in Dialogboxen (z. B. ALT-27 löscht das Eingabefeld, da ESC das Eingabefeld ebenfalls löscht).

ALT-SHIFT-HELP

Speichert den Inhalt des Bildschirms im DEGAS-Format auf dem Laufwerk ab, von dem der Assembler gestartet wurde.

CONTROL-ALT-DELETE Löst einen RESET aus (wie im TOS 1.4). Eine resetfeste Ramdisk bleibt dabei erhalten, der Assembler ist jedoch weg (und Ihr nicht gesicherter Sourcetext auch!). Das nennt man auch einen Warmstart des Rechners.

CONTROL-ALT-rechtes SHIFT-DELETE Löst einen Kaltstart aus, dabei wird der gesamte Hauptspeicher des Rechners gelöscht. Es ist somit vollkommen witzlos, sich vorher seinen Sourcetext in einer Ramdisk abzulegen: der ist dann weg! (Ebenso wie alle kleinen netten RESET-festen Viren, die Sie sich im Laufe der Zeit eingefangen haben).

ESC Nach ESC ändert sich die Cursorform. Wenn dann eine Buchstabetaste gedrückt wird, wird ein Befehl ausgeschrieben (man kann also mit ESC Befehle abkürzen). Die Belegung der Tasten nach ESC steht in Tabelle 5.3.

5.5.5 Die Drop-Down-Menüs

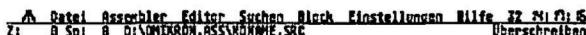


Abbildung 5.1: Die Drop-Down-Menüs (eingeklappt)

Die meisten Funktionen der Menüleiste sind auch auf Tastendruck zu erreichen. Welche Tastenkombination zu drücken ist, steht jeweils hinter den Einträgen. Dabei können drei Steuerzeichen verwendet werden:

1. ^ bedeutet die CONTROL-Taste, die mit einer anderen Taste zu drücken ist (z. B. ^I für CONTROL-I (Info aufrufen)).
2. Mit # ist die ALTERNATE-Taste gemeint (z. B. #Y für ALT-Y (Block löschen)).

Tabelle 5.3: Die Escape-Belegungen

Taste	ohne SHIFT	mit SHIFT
A	ADD.W	ADD.L
B	BRA	BSR
C	CMP	CLR
D	DBRA	DIVU
E	EOR	EXT
F	FAIL	
I	ILLEGAL	IBYTES
J	JSR	JMP
L	LEA	
M	MOVE	MULU
N	NOT	NEG
O	OR	OUTPUT
P	PEA	PATH
Q	ADDQ	MOVEQ
R	RTS	ROL
S	SUB	SWAP
T	TST	TRAP
U	UNLK	
X	XOR	

3. Mit \uparrow ist die SHIFT-Taste gemeint, sie wird zum Beispiel bei \uparrow^E für SHIFT-CONTROL-E (ASCII-Suchen-und-Ersetzen) benutzt.

Unter dem Menü „Hilfe“ können Sie sich im Zweifelsfall diese Tastencodes jederzeit nochmal ansehen (siehe Kapitel 5.5.5 auf Seite 80). Wenn hinter einem Eintrag noch drei Punkte stehen, heißt das, daß dem Anwählen dieser Funktion noch eine Abfrage folgt (z. B. eine Dialogbox, der Fileselector o. ä.).

Das Menü „Atari“

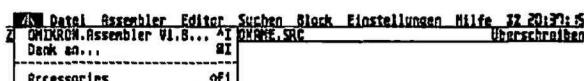


Abbildung 5.2: Das Menü „Atari“

Der erste Menüpunkt („OMIKRON.Assembler V1.7...“) zeigt (fast) die gleiche Dialogbox (Siehe Abbildung 5.3 auf Seite 47), die Sie schon beim Start des Assemblers gesehen haben. Sie enthält Informationen über die geladene Version des OMIKRON.Assemblers sowie Ihre Seriennummer. Diesen Menüpunkt können Sie mit CONTROL-I auch über Tastatur aufrufen.



Abbildung 5.3: Die Info-Box

Darunter können Sie mit „Dank an...“ die Liste der Personen ansehen, die irgendwie zur Förderung des OMIKRON.Assemblers beigetragen haben (Siehe Abbildung 5.4 auf Seite 48). Diese Dialogbox können Sie auch mit ALT-I aufrufen.



Abbildung 5.4: Die Dankesliste

An der Stelle, an der normalerweise die Accessories stehen, steht der Eintrag „Accessories“ (SHIFT-F1). Wenn Sie diesen Eintrag anwählen, schaltet der OMIXRONAssembler auf das GEM zurück und zeigt eine kurze GEM-Menüleiste, welche Ihnen die Möglichkeit gibt, auch aus dem OMIXRONAssembler heraus Accessories aufzurufen. Dort kommen Sie mit dem Menüpunkt „zurück in den Assembler“ wieder zur Benutzeroberfläche des Assemblers.

Das Menü „Datei“

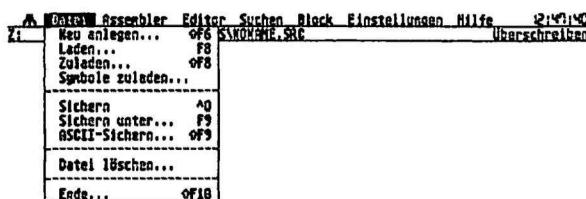


Abbildung 5.5: Das Menü „Datei“

Unter diesem Menüpunkt sind alle Disketten- bzw. Harddiskoperationen zusammengefaßt. (Ausnahmen sind „Block speichern“ im „Block“-Menü und

„Einstellungen sichern“ im „Einstellungen“-Menü).

Der Menüpunkt „Neu anlegen...“ (SHIFT-F6) erlaubt es Ihnen, den aktuellen Sourcetext aus dem Speicher zu entfernen. Dabei werden alle internen Pointer zurückgesetzt und ein leerer Sourcetext mit dem Namen NONAME.SRC erzeugt. Eine Sicherheitsabfrage warnt Sie (Siehe Abbildung 5.6 auf Seite 49), denn der Sourcetext ist nach dieser Funktion unwiderruflich verloren.

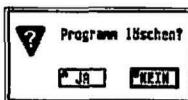


Abbildung 5.6: Die Sicherheitsabfrage bei „Neu anlegen...“

Der Menüpunkt „Laden...“ (F8) bringt den File-Selector auf den Bildschirm; Sie können dann einen Sourcetext laden. Der OMIKRONAssembler erkennt dabei automatisch das Format des Sourcetexts. Das eigene Format (zu erkennen an der SRC-Extension) wird im Gegensatz zum ASCII-Format erheblich schneller geladen und braucht weniger Speicherplatz auf der Diskette als ein ASCII-Sourcetext. Wenn sich der geladene Sourcetext im SRC-Format befindet, wird sofort eine Box angezeigt, welche das Erstellungsdatum (und -Uhrzeit), sowie das Datum der letzten Änderung (und Uhrzeit) enthält. Diese Box wird für ca. 2 Sekunden angezeigt, dann erst wird der Sourcetext geladen. Wird während der 2 Sekunden eine Taste gedrückt oder die Maus bewegt, lädt der Assembler sofort weiter.

Zum Erstellungsdatum: eine Datei gilt als neu erstellt, wenn der Sourcetext unter dem aktuellen Filennamen beim Speichern noch nicht existiert. Allerdings kann man eine Datei laden und unter anderem Namen abspeichern, ohne daß das Erstellungsdatum verloren geht.

Falls der Sourcetext mit einem Password geschützt ist, erwartet der Assembler die Eingabe (Siehe Abbildung 5.7 auf Seite 50) des bis zu 8 Zeichen langen Passwords (kann mit CONTROL-~ gesetzt werden). Bei fehlerhaftem Password wird der Ladevorgang sofort abgebrochen, sonst wird weitergeladen (siehe Kapitel 5.5.5 auf Seite 53). Das Password wird während der Eingabe übrigens *nicht* angezeigt (falls Ihnen jemand auf dem Monitor schaut...).

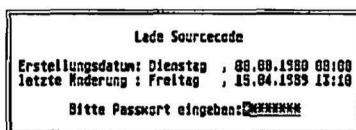


Abbildung 5.7: Die Passwordabfrage beim Laden

Ein ASCII-Sourcetext muß beim Laden erst tokenisiert werden, d. h., der Assembler muß jede eingelesene Zeile auf ihren Syntax hin überprüfen und die Zeile in das interne Format übertragen. Dieser Teil ist konzeptionsbedingt der langsamste Teil des Assemblers, trotzdem konvertiert der OMIKRON Assembler etwa 450 KB pro Minute in das interne Format⁶. Während des Einlesens von ASCII-Sourcetexten zeigt der Assembler die aktuelle Zeilennummer (in Zwanzigerschritten) links oben an. Der Ladevorgang kann jederzeit mit ESC abgebrochen werden. Nach dem Einlesen eines ASCII-Sourcetextes kann eine Dialogbox erscheinen, in der der Assembler die Anzahl der nicht tokenisierbaren Zeilen ausgibt. Solche Zeilen enthalten meist einen Syntax-Fehler, oder ein Symbol wurde zweimal deklariert. Nicht tokenisierbare Zeilen können Sie mit ALT-S anspringen. Dazu markiert der OMIKRON Assembler solche Zeilen mit „:>“ als Remarks, diese Markierung müssen Sie nach der Korrektur der Zeile löschen.

Wie bei allen Lade- und Speicheroperationen wird auch hier der Original-File-Selector verwendet. Das mag einige vielleicht stören, weil ihr altes Betriebssystem keinen sehr komfortablen zur Verfügung stellt und möglicherweise beim Zeichen „_“ abstürzt. Dafür können Sie aber eigene File-Selectoren einbinden. Es gibt eine ganze Reihe von luxuriösen File-Selectoren als Public-Domain-Programme. Man hätte natürlich auch einen erweiterten File-Selector in den Assembler integrieren können, der dann aber bestimmt wieder nicht jedermanns Wünschen gerecht werden würde. Weiterhin wird ab dem TOS 1.4 der Extended-File-Selector (AES 91) benutzt, d. h., oben im File-Selector steht die Funktion, die Sie gerade angewählt haben.

An dieser Stelle sei darauf hingewiesen, daß der Assembler die Command-

⁶Das entspricht ungefähr der Assembliergeschwindigkeit des Devpac V1.0

line auswertet. Sie können ihn also als Anwendung mit der SRC-Extension (natürlich auch mit S, ASM usw.) anmelden. Ein Doppelklick auf einen Source-Text lädt dann den Assembler und den Source.

Der Menüpunkt „Zuladen...“ (SHIFT-F8) erlaubt es Ihnen, in den bereits im Speicher stehenden Sourcetext weitere ASCII-Sourcetexte einzufügen. Dazu stellen Sie den Cursor an die Stelle, an die eingefügt werden soll (nur die Zeile ist entscheidend) und rufen dann diese Funktion auf. Danach werden wie beim normalen Laden von ASCII-Sourcen die nicht tokenisierbaren Zeilen gekennzeichnet. Da jedoch jede Zeile einzeln eingefügt werden muß, kann es beim Zuladen zu ganz erheblichen Wartezeiten kommen, wenn Ihr Sourcetext schon etwas größer ist. Diese Wartezeit können Sie drastisch verkürzen, wenn Sie stets ans Sourcetext-Ende zuladen und dann die zugeladenen Zeilen als Block markieren, um sie an die endgültige Stelle im Sourcetext zu verschieben. Da bei dieser Methode nur angefügt werden muß (und nicht eingefügt), geht das Laden erheblich schneller. Aber, wie bereits gesagt, lohnt sich dieser Trick erst bei längeren Sourcetexten.

Unter dem Menüpunkt „Symbole laden...“ können Sie nicht einen kompletten Sourcetext einladen, sondern nur eine Symboltabelle. Eine solche Symboltabelle besteht nur aus Konstanten-Zuweisungen (siehe Kapitel D.3 auf Seite 192). Eine Symboltabelle können Sie ganz einfach erzeugen, indem Sie ein Programm assemblyn, welches nur EQU-Befehle enthält (genaugenommen nur Befehle, die keinen Code produzieren. Kommentare, OUTPUT-Befehle usw. sind also auch möglich). Der Assembler meldet dann, daß er eine Symboltabelle mit n Symbolen erzeugt habe (Siehe Abbildung 5.8 auf Seite 52). Diese Symboltabelle können Sie dann mit der Extension SYM abspeichern und mit dieser Funktion wieder einladen. Diese Symboltabellen haben den Vorteil, daß Sie sich beim Blättern in Ihrem Listing nicht durch viele Seiten von EQUs wühlen müssen, denn diese Symboltabellen sind nicht sichtbar (als Beispiele seien die im LIBRARYS.OMI stehenden Dateien SYS_VARS.S und GEM_VARS.S genannt). Allerdings müssen Sie, um einen Wert zu ändern, den Sourcetext der Symboltabelle haben, diese neu assemblyn und dann wieder neu laden. Und das ist auch gleich die zweite Besonderheit der Symboltabellen: Sie können eine Symboltabelle beliebig häufig laden, bei jedem Laden werden die alten Werte eines bereits existierenden Symbols überschrieben.

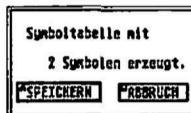


Abbildung 5.8: Eine Symboltabelle wurde erzeugt

Beispiel:

Test: EQU 1

Dieses Beispiel mit F1 assemblieren und als Symboltabelle abspeichern. Dann ändern Sie den Wert des Symbols Test auf 228.

Test: EQU 228

Wenn Sie nun die eben erstellte Symboltabelle wieder einladen, ergibt sich folgendes Bild:

Test: EQU 1

Noch eine Bemerkung zu den Symboltabellen: Die hier beschriebenen Symboltabellen haben nichts mit den Symboltabellen zu tun, die an ein assembledes Programm angehängt werden können.

Mit dem Menüpunkt „Sichern“ (CONTROL-Q) können Sie Ihren Sourcetext unter dem in der Statuszeile angegebenen Namen abspeichern. Es wird dazu kein File-Selector ausgegeben. Als Extension wird stets .SRC angehängt. Der Sourcetext wird dabei im eigenen Format abgespeichert. Wie bereits erwähnt, werden solche Sourcetexte nicht nur schneller gespeichert und geladen, sie sind auch kürzer als im ASCII-Format. Wenn der Source unter diesem Namen schon existiert, wird eine Dialogbox ausgegeben (Siehe Abbildung 5.9 auf Seite 53).

„Sichern unter...“ (F9) entspricht dem Menüpunkt „Sichern“, diesmal wird jedoch der File-Selector ausgegeben.

Noch etwas zum eigenen Sourcetextformat: Es werden (und das macht dieses Format so praktisch) alle aktuellen Einstellungen, die den Sourcetext betreffen, mit abgespeichert. Das sind: Die Cursorposition, die 10 Textmarker, Datum und Uhrzeit der Erstellung des Sourcetextes, Datum und Uhrzeit der letzten Änderung des Sourcetextes, die Stelle der letzten Änderung

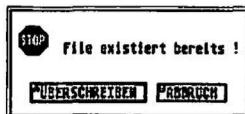


Abbildung 5.9: Die Sicherheitsabfrage vor dem Überschreiben

(CONTROL-Z), ein markierter Block, alle Einstellungen des Darstellungsmenüs (Tabulatoren, etc.), der Einfügemodus, Groß-, Kleinschrift-Konvertierung von Symbolen, der Remark-Merker, das Auto-Start-Flag des Debuggers, Länge des Zielcodespeichers sowie des freien Speichers für GEM. Durch einen Fehler in den Assembler-Versionen kleiner V1.1 sind dort wahrscheinlich beide Angaben fehlerhaft. Wenn Ihre Uhrzeit im Rechner nicht stimmt (keine Hardware-Uhr oder Mega ST), stimmen die Angaben natürlich auch nicht.

Und schließlich bietet das Format den Vorteil, den Sourcetext mit einem Password oder der Seriennummer des Assemblers schützen zu können. So kann z. B. jemand, der Ihren Sourcetext entwendet hat, nichts damit anfangen, solange er nicht auch Ihren Assembler mitgestohlen hat oder Ihr Password kennt. Das Password wird selbstverständlich kodiert abgespeichert.

Das Password können Sie vor dem Speichern mit CONTROL-~ festlegen. Die erscheinende Dialogbox (Siehe Abbildung 5.10 auf Seite 54) erlaubt Ihnen, ein Password zu definieren und den Sourcetext entweder mit einem Password zu schützen oder mit einem Seriennummernschutz zu versehen. Ein Ausschalten des Schutzes ist hier ebenfalls möglich. Der Seriennummernschutz verbietet es, den Sourcecode in einem anderen Assembler als mit Ihrer Seriennummer zu laden. Ein Password darf alle ASCII-Codes enthalten (Ausnahmen sind 0, 27 und 255). Das Password muß vor jedem Laden neu angegeben werden, das kann dann ganz schön lästig werden. Sie können deshalb statt des Password-Buttons auch den Button in der Mitte anklicken. Er enthält Ihre Seriennummer. Dann kann der Sourcetext nur noch auf einem Assembler mit dieser Seriennummer geladen werden — noch ein Grund, den OMIKRON-Assembler nicht weiterzugeben. Wenn jeder einen Assembler mit Ihrer Seriennummer hat, ist dieser Schutz ziemlich witzlos.

„ASCII-Sichern...“ (SHIFT-F9) erlaubt es Ihnen, den Sourcetext im nor-

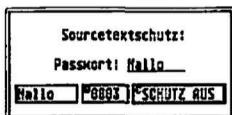


Abbildung 5.10: Setzen des Passwords oder Seriennummerschutzes

malen ASCII-Format zu speichern; dabei gehen dann natürlich alle Einstellungen, die im SRC-Format mitabgespeichert werden, verloren. Dadurch können Sie Ihre Programme auch an Leute weitergeben, die keinen OMI-KRONAssembler haben. Außerdem ist das nötig, wenn Sie Ihr Programm in ein anderes einfügen wollen (mit der Funktion „Zuladen...“). Falls Sie einen Block markiert haben, erscheint eine Dialogbox, die es Ihnen erlaubt, entweder nur diesen Block oder das ganze Programm zu speichern. Das Abspeichern eines Blockes ist nur im ASCII-Format möglich.

Mit „Datei löschen...“ können Sie Ihre gesamte Harddisk fileweise leeren, nur um Platz für die neuesten Sourcetexte zu schaffen. Dazu erscheint ein Fileselector, der Ihnen die Auswahl des zu löschenen Programms erlaubt; wenn Sie die darauffolgende Sicherheitsabfrage (Siehe Abbildung 5.11 auf Seite 54) mit „Ja“ beantworten, sind Sie das File los.

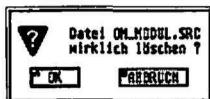


Abbildung 5.11: Die Löschen-Dialogbox

„Ende...“ (SHIFT-F10 oder CONTROL-X) gibt Ihnen die Möglichkeit, den Assembler ohne RESET zu verlassen und zum Desktop zurückzukehren. Wenn Sie Ihren aktuellen Sourcetext brav gesichert haben, erscheint nach CONTROL-X keine Abfrage mehr. SHIFT-F10 oder der Menüeintrag erzwingen jedoch eine Abfrage. Falls Sie den Sourcetext nicht gesichert haben, erscheint eine Dialogbox (Siehe Abbildung 5.12 auf Seite 55), welche Ihnen

die Möglichkeit gibt, dies nachzuholen. Falls dort beim Speichern ein Fehler auftritt, wird der Assembler nicht verlassen. Durch einen Klick auf den „Weiter“-Button verlassen Sie den Assembler endgültig.

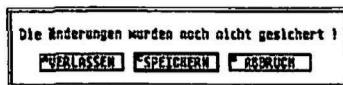


Abbildung 5.12: Die Dialogbox beim Beenden

Das Menü „Assembler“

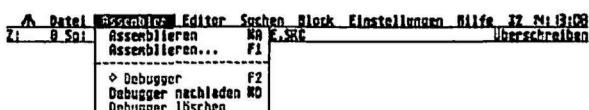


Abbildung 5.13: Das Menü „Assembler“

Hier kommen wir endlich zum wesentlichen Bestandteil des Assemblers (der hat ihm doch tatsächlich seinen Namen gegeben).

Der oberste Menüpunkt „Assemblerieren“ (ALT-A) assembliert den vorhandenen Sourcetext und löst den Default-Button aus.

Fehler bei der Assemblierung: Da der Editor natürlich nicht alle Fehler bereits bei der Eingabe finden kann, können natürlich auch Fehler bei der Assemblierung auftreten. Dabei gibt es vier Arten von Fehlern:

- Fatale Fehler (z. B. Datei bei IBYTES nicht gefunden, aber auch FAIL)
- Allgemeine Fehler (z. B. Division durch Null)
- Korrigierbare Fehler (Adresse wurde begradigt)

- Einfache Warnungen (Kann zu Bcc.S optimiert werden)

Eine vollständige Liste aller Fehlermeldungen steht weiter hinten (siehe Kapitel 5.7 auf Seite 100).

Bei fatalen Fehlern wird die Assemblierung sofort abgebrochen, eine Dialogbox mit der Fehlerursache erscheint, und nach RETURN springt der Editor in die entsprechende Zeile.

Bei allen anderen Fehlern assembliert der Assembler weiter und markiert alle fehlerhaften Zeilen. Falls bei der Assemblierung ein (oder mehrere) Fehler aufgetreten ist/sind, wird (solange der Sourcetext fehlerhaft ist) der Abbruch-Button default. Es wird bei Auslösen des Abbruch-Buttons dann automatisch zum ersten Fehler gesprungen. Wenn Sie den Dialog jedoch mit der UNDO-Taste abbrechen, bleibt der Cursor in der aktuellen Zeile. Erwähnenswert ist noch, daß der Editor sich *alle* fehlerhaften Zeilen merkt (und nicht nur 20 oder 30, wie andere Assembler), und daß z. B. das Duplizieren einer fehlerhaften Zeile auch die Fehlermarkierung verdoppelt. Sie können auch einen fehlerhaften Sourcetext abspeichern. Beim späteren Einladen sind alle Fehler (leider) noch vorhanden. Wenn sich der Cursor auf einer fehlerhaften Zeile befindet, wird in der Statuszeile dann die entsprechende Fehlermeldung angezeigt. Um solche fehlerhaften Zeilen einfach finden zu können, brauchen Sie lediglich ALT-J (Jump to error) drücken, und schon befinden Sie sich in der nächsten fehlerhaften Zeile. Mit SHIFT-ALT-J können Sie auch aufwärts suchen lassen. Wenn Sie eine Zeile korrigiert haben, sollten Sie mit CONTROL-J zum nächsten Fehler springen, dies hat den Vorteil, daß die eben noch als fehlerhaft gekennzeichnete Zeile aus der Fehlerliste entfernt wird und sie von ALT-J nicht mehr gefunden wird.

Der zweite Menüpunkt „Assemblieren...“ (F1 oder CONTROL-A) assembliert den Sourcecode, gibt aber im Gegensatz zu „Assemblieren“ eine Dialogbox aus (Siehe Abbildung 5.14 auf Seite 57), in der Sie zwei Dinge tun können:

1. Sie können Ihr Programm mit einer Symboltabelle versehen, d. h., alle im Sourcetext benutzten Label werden an das fertige Programm angehängt. Eine Symboltabelle erlaubt eine wesentlich bessere Orientierung im Programm, wenn Sie es mit dem Debugger auf Fehler testen, da statt Adressen eben Symbole angezeigt werden. Eine Symboltabelle

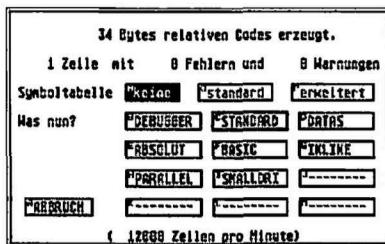


Abbildung 5.14: Die Dialogbox nach dem Assemblieren

kann das Programm manchmal erheblich verlängern, behindert das Programm aber nicht, da das GEMDOS die Symboltabelle gar nicht erst lädt. Der OMIKRON.Assembler unterstützt drei verschiedene Symboltabellenformate:

- Das Standard-Format von Digital Research: Die maximale Symbollänge beträgt dabei 8 Zeichen, d. h., der Assembler kürzt alle Symbole auf 8 Zeichen.
- Das Format der Firma GST: Dabei handelt es sich um eine Erweiterung des Standard-Formats, welches nun maximal 22 Zeichen lange Symbole kennt.
- Das ÜbergabefORMAT an den Debugger: Der Debugger bekommt vom Assembler, wenn eine Symboltabelle gewünscht wurde, nur die Zeiger auf die Symbolnamen übergeben, d. h., er ist in der Lage, Symbole bis maximal 23 Zeichen (Einschränkung durch den Editor des Assemblers) zu verwalten.

Das Format 1c kann *nicht* abgespeichert werden (im Gegensatz zu 1a und 1b); der Assembler erzeugt das Format automatisch, wenn eines der Formate 1a oder 1b gewählt wurde und in den Debugger gesprungen wird. Falls der zur Verfügung stehende Zielcodespeicher nicht für eine Symboltabelle reicht, erscheint eine Dialogbox, die eine entsprechende Meldung ausgibt. Der Assembler erzeugt dann eine Symbolta-

belle, soweit der Speicher reicht. Nähere Informationen zum Aufbau einer Symboltabelle finden Sie im Tabelle D.4 auf Seite 197.

2. Als zweiten Punkt haben Sie die Wahl zwischen verschiedenen Möglichkeiten, das erzeugte Programm abzuspeichern bzw. in den Debugger zu springen.

Der Assembler kann dabei maximal 12 verschiedene Auswahlmöglichkeiten verwalten, die vom Anwender vollkommen frei programmierbar sind. Dazu wurde eine ziemlich komfortable Schnittstelle implementiert, die einem den Assembler nach allen Seiten offen hält. Eine Dokumentation der Schnittstelle mit dokumentierten Beispielen, RSC-Editor für die internen Ressourcen des Assemblers sowie einem Spiel als Beispiel zur Programmierung der Schnittstelle finden Sie im Anhang H auf Seite 206.

Die Datei OM-ASSEM.DAT, die in der Grundversion mitgeliefert wird, enthält bereits einige Module, die im folgenden erklärt werden:

DEBUGGER: Dieses Modul tut nichts anderes, als das Programm dem Debugger zu übergeben; dabei wird wahlweise eine Symboltabelle mit übergeben sowie die gesamte Assembler↔Debugger-Schnittstelle (siehe Kapitel 5.8 auf Seite 104) mitverwaltet.

STANDARD: Hiermit kann das vorher assemblierte Programm als Programm (bzw. Accessory o. ä.) abgespeichert werden. Auch hier kann eine Symboltabelle zum späteren Debuggen an das Programm angefügt werden.

DATAS: Für BASIC-Programmierer ziemlich interessant, wenn man ein Assemblerprogramm einbinden will. Der Assembler erzeugt aus dem Programm automatisch DATA-Zeilen, welche er zudem auch gleich mit einer Initialisierungsschleife versieht (Speicher reservieren und Datas einlesen). Unter OMIKRON.BASIC können Sie die Zeilen einfach mit „BLOCK LOAD *.*“ einlesen. Für andere BASIC-Dialekte müssen Sie mit einem Texteditor die erste Zeile entfernen und eventuell alle „\$“-Zeichen entfernen.

ABSOLUT: Dieser Button bietet Ihnen die Möglichkeit, ein Programm schon im Assembler zu relozieren.

Dazu ein paar Erklärungen: Das Gemdos benötigt stets Programme, die ab Adresse 0 laufen und eine Tabelle aller absoluten Adressen beinhalten. Beim Laden eines Programms, welches ja im Atari *nicht*, wie bei den Homecomputern, an eine fixe Adresse geladen wird (siehe z. B. C-64 nach Adresse \$801), paßt das Gemdos das Programm dann automatisch an die Ladeadresse an, d. h., das Gemdos reloziert das Programm. Da nun aber z. B. Modulprogramme stets an einer fixen Adresse liegen und nicht reloziert werden können (ist halt ein ROM), kann man mit diesem Button das Programm bereits durch den Assembler relozieren lassen.

Wenn man diesen Button anklickt, erscheint eine Dialogbox, in der man zum einen die gewünschte Basisadresse des Programms eintragen kann (bei Modulen z. B. \$FA0000), und zum anderen drei weitere Buttons zur Auswahl hat:

ABBRUCH: Springt in den Editor zurück.

SPEICHERN: Speichert das relozierte Programm ab.

KOPIEREN: Reloziert und kopiert das Programm an die Adresse, für die es reloziert worden ist. *Achtung:* Da der Assembler hierbei keinerlei Sicherheitsabfragen hat, besteht die Chance, daß er bei falscher Bedienung zerstört wird! Sie sollten diese Funktion daher nur benutzen, wenn Sie ihre Wirkung voll verstanden haben.

BASIC: Mit diesem Button können Sie ein beliebiges Programm in BASIC einbinden (natürlich auch in andere Programmiersprachen). Dabei wird einem relozierbaren Programm eine Relozierroutine angefügt, welche das Programm automatisch beim ersten Aufruf an die entsprechende Adresse reloziert. Einzige Voraussetzung dabei ist, daß das Programm beim ersten Aufruf ab der ersten Adresse gestartet wird, danach löscht sich die Relozierroutine selbsttätig.

INLINE: Dieser Button erlaubt das Erzeugen einer INLINE-Zeile für OMIKRON.BASIC (also INLINE "Hexdaten").

PARALLEL: Dieses Modul überträgt das fertige Programm über die Parallel-Schnittstelle. Das dient in erster Linie für den Amiga.

Entsprechende Empfangsroutinen sind bei AssAge Entertainment Software erhältlich (Adresse steht auf der Titelseite).

SMALLDRI: Hiermit wird die Datei im DRI-Linker-Format abgelegt. Sie können Ihr Assemblerprogramm dann problemlos beispielsweise zu C-Programme linken. „Small“ bedeutet, daß noch keine Importe möglich sind. Alle Symbole, die in Ihrem Assemblerprogramm verwendet werden, müssen auch in diesem definiert sein. Wenn Sie Parameter übergeben wollen, können Sie das am einfachsten über den Stack.

Globale Symbole können durch einen 2. Doppelpunkt markiert werden; z. B. „main:: nop“. Durch den GLOBAL-Opcode können Symbole automatisch mit zwei Doppelpunkten versehen; sprich: als global definiert werden (siehe Kapitel 5.6.7 auf Seite 95).

Achtung: Zum Linken ist natürlich eine Symboltabelle nötig; Sie sollten daher an den Anfang eines Programms, das gelinkt werden soll, „OPT D+“ schreiben.

Der Beginn eines Opcodes (\$0007) im Relocinfo des DRI-Programms wird nicht gesetzt. Er ist jedoch für das Linken auch nicht erforderlich.

Die einzelnen Buttons sind nicht immer alle anwählbar. Die Gründe für das Disabeln der Buttons sind in Tabelle 5.4 angegeben.

Der Menüpunkt „⇒ Debugger“ (F2) ist enabled, wenn der residente Debugger vorhanden ist. Er erlaubt den Sprung in den Debugger, ohne ein Programm zu übergeben, z. B. um vom Debugger aus eine Diskette zu formatieren. Wenn Sie mit diesem Menüpunkt in den Debugger springen, können Sie auch ein Programm nachladen und dieses debuggen, dies ist ja nach einem Einsprung von „Assemblieren“ aus nicht möglich. Es ist allerdings zu beachten, daß 40 000 Bytes weniger Gmedos-Speicher zur Verfügung stehen als Sie bei „REORGANISIEREN“ angegeben haben (siehe Kapitel 5.5.5 auf Seite 65). Diese 40 000 Bytes stehen Ihnen wiederum nur nach „Assemblieren“ im Debugger zur Verfügung.

„Debugger nachladen“ (ALT-D ermöglicht es Ihnen, den Debugger nachträglich zu laden. Wenn Sie den Assembler verlassen, wird der Debugger auch wieder gelöscht. Um dieses Kommando ausführen zu können, müssen

Tabelle 5.4: Die Gründe fürs Disabeln

Modul:	Grund für ein Disabeln:
DEBUGGER	Debugger ist nicht resident vorhanden
STANDARD	Das Modul ist stets enabled
DATAS	Relocinfo vorhanden, BSS vorhanden, Prglänge > 99999 Bytes
ABSOLUT	BSS vorhanden
BASIC	BSS vorhanden
INLINE	Relocinfo vorhanden, BSS vorhanden, Prglänge > 124 Bytes
PARALLEL	Das Modul ist stets enabled
SMALLDRI	Das Modul ist stets enabled

mindestens 240 000 Byte freier Gemdos-Speicher zur Verfügung stehen (siehe Kapitel 5.5.5 auf Seite 65).

Wenn Sie den Speicher anderweitig verwenden wollen, können Sie ihn mit „Debugger löschen“ nach einer Sicherheitsabfrage (Siehe Abbildung 5.15 auf Seite 61) wieder freigeben. Wenn der Debugger bereits vor dem Start des



Abbildung 5.15: Die Sicherheitsabfrage zum Löschen des Debuggers

Assemblers resident im Speicher installiert war, können Sie ihn hiermit *nicht* mehr löschen — der Menüpunkt wird dann disabled

Das Menü „Editor“

Hier sind einige häufig gebrauchte Funktionen, die sich sonst nicht einordnen lassen, zusammengefaßt.

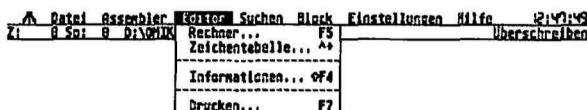


Abbildung 5.16: Das Menü „Editor“

Der erste Menüpunkt „Rechner...“ (F5) enthält einen vollständigen Taschenrechner (Siehe Abbildung 5.17 auf Seite 62). Dieser Rechner ist speziell auf den Assemblerprogrammierer zugeschnitten (wie alle anderen Features des Assemblers auch). Der Rechner erlaubt die Eingabe einer beliebigen Formel in der Schreibweise, die der Assembler sonst auch fordert. Vergleiche beherrscht der Taschenrechner jedoch nicht!

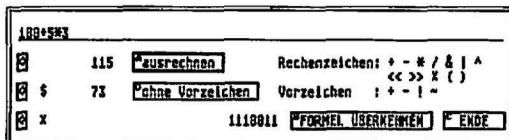


Abbildung 5.17: Der Taschenrechner

Diese Formel darf auch Symbole des Sourcetextes enthalten. Dabei gibt es allerdings zwei Punkte zu beachten:

- Konstanten, die mit EQU zugewiesen werden und keine Formel enthalten, können jederzeit benutzt werden.
- Label jedoch bekommen ihren Wert erst bei der Assemblierung, d. h., um z. B. die Länge einer Prozedur zu ermitteln, müssen Sie den Sourcecode erst einmal assemblieren, danach gehen Sie direkt in den Taschenrechner und geben z. B. ein: ENDE-ANFANG. Ein Druck auf die RETURN-Taste bzw. den „ausrechnen“-Button zeigt Ihnen dann die gewünschte Länge.

Da es in Assembler manchmal praktisch ist, eine Zahl vorzeichenlos zu betrachten, gibt es den Button „ohne Vorzeichen“. Das Ergebnis wird dann in

eine vorzeichenlose Zahl gewandelt (z. B. -123 in \$FFFFFF85). Eine Zurückwandlung in eine vorzeichenbehaftete Zahl ist durch erneutes Drücken von RETURN möglich. Um nun ein Ergebnis auch nutzen zu können, gibt es drei Buttons, die links von den Ergebnissen stehen. Mit ihnen kann das Ergebnis an die Cursorposition übernommen werden (gemäß ihrer Zahlenbasis). Ferner ist auch die Übernahme der gesamten Formel an die Cursorposition möglich, indem man auf den Button „Formel übernehmen“ klickt. Der „ENDE“-Button oder die Taste UNDO bringen Sie ohne Übernahme einer Formel in den Editor zurück. Es sei an dieser Stelle noch einmal daran erinnert, daß ein Doppelklick mit der rechten Maustaste auf eine Zahl, Formel etc. den Taschenrechner aufruft, und, daß die Zahl, Formel o. ä. in denselben übernommen wird. Noch eine nicht unwichtige Bemerkung am Rande: Der Taschenrechner kennt keine Fehlermeldungen; wenn ein Fehler gefunden wurde, ist das Ergebnis stets 0, d. h., $1/0+100$ ist 0, da bei $1/0$ ein Fehler auftrat.

Der nächste Menüpunkt ist „Zeichentabelle...“ (CONTROL-+). Falls Sie ein Zeichen eingeben wollen, das Sie nicht auf der Tastatur finden und dessen ASCII-Code Sie nicht auswendig wissen⁷, können Sie hiermit alle 255 möglichen Zeichen des STs auf einmal zu sehen bekommen (Siehe Abbildung 5.18 auf Seite 63). Sie können dann mit einfacherem Anklicken das entsprechende Zeichen auswählen. Es wird dann an die Cursorposition übernommen. Falls Ihnen keines der Zeichen zusagt, können Sie kurz die rechte Maustaste drücken, eine beliebige Taste der Tastatur oder mit der linken Maustaste außerhalb der Dialogbox klicken. Darauf verschwindet die Dialogbox wieder.

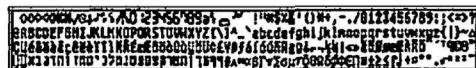


Abbildung 5.18: Die Zeichentabelle

⁷Mit Eingabe eines ASCII-Codes über ALTERBATE-Zehnerblock können beliebige Zeichen eingegeben werden. (siehe Kapitel 5.5.4 auf Seite 44).

Der Menüpunkt „Informationen...“ (SHIFT-F4) gibt eine kleine Statistik aus (Siehe Abbildung 5.19 auf Seite 64), wie der Speicher momentan belegt ist. Zur Speicherverwaltung des Assemblers noch einige Worte:



Abbildung 5.19: Die Informationsbox

Der Sourcetext besteht beim Assembler aus 4 Blöcken, die durch Pointer miteinander verkettet sind.

1. Der wohl wesentlichste Block (für den Editor) ist wohl die Zeilentabelle; jede Zeile kostet 10 Bytes Speicherplatz. In dieser Tabelle werden alle Informationen über eine Zeile abgelegt (Remark ja/nein, fehlerhafte Zeile ja/nein, ...).
2. Der Programmblock. Die angegebene Programmlänge müßte in den meisten Fällen der tatsächlichen recht nahe kommen (aber z. B. DS-Befehle fallen aus dem Rahmen).
3. Der Remarkblock. Auch hierzu noch eine Bemerkung: Der Editor packt mehrere gleiche Zeichen, d. h., 80 mal „~“ kostet nur drei Bytes Speicherplatz.
4. Der letzte Block enthält alle Symbolnamen, Formeln und sonstige speicherplatzfressende Informationen.

Da die Verwaltung der vier Blöcke sehr aufwendig ist, ist der Assembler der jetzigen Version nicht in der Lage, den Speicher selbst zu organisieren, wie z. B. ein Texteditor, der 8 unabhängige Texte im Speicher verteilen muß.

Die vom Assembler vorgegebene Verteilung entspricht jedoch der Praxis, und es kann normalerweise nur bei der Größe des Zielcodespeichers Probleme geben. Für einen solchen Fall gibt es den „REORGANISIEREN“-Button, der eine neue Dialogbox aufruft (Siehe Abbildung 5.20 auf Seite 65), in welcher man die Länge des zu erzeugenden Programms einstellen kann. Denken Sie dabei daran, daß eine Symboltafel zusätzlich Speicherplatz belegt. Bei Mammut-Programmen muß man also u. U. auf sie verzichten.

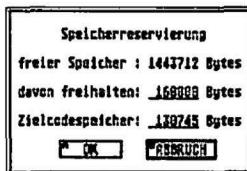


Abbildung 5.20: Der Reorganisations-Dialog

Für den Debugger benötigt man (leider) auch noch Speicher, dazu noch für das zu debuggende Programm, da es mit der Gemdos-Funktion Pexec() angemeldet werden muß. Der freie Speicher muß größer als die Länge aller drei Programmsegmente (TEXT, DATA und BSS) sein! Eine übergebene Symboltafel kostet allerdings keinen zusätzlichen Platz (ein Glück). Wenn Sie den Debugger nachladen wollen (siehe Kapitel 5.5.5 auf Seite 60), müssen hier mindestens 240 000 Bytes Gemdos-Speicher reserviert werden.

Machen Sie sich allerdings nicht allzuviel Gedanken über die Speicherverteilung; wer große Programme schreibt, hat meistens auch entsprechend Hauptspeicher zur Verfügung.

Der Sourcetext geht beim Reorganisieren *nicht* verloren. Wenn der Speicher nicht reicht, warnt eine Dialogbox, und man hat die Möglichkeit, den Source-Text zu löschen (um den Speicherplatz frei zu haben) oder abzubrechen.

Der letzte Menüpunkt lautet „Drucken...“ (F7). Von hier aus kann man

den gesamten Sourcetext ausdrucken. Wenn ein Block markiert ist, erscheint eine Dialogbox (Siehe Abbildung 5.21 auf Seite 66), in der Sie wählen können, ob Sie den gesamten Text oder nur den Block drucken wollen. Der Ausdruck

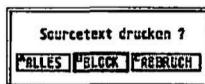


Abbildung 5.21: Die Dialogbox Block/Alles drucken

erfolgt stets über die Centronics-Schnittstelle. Eine Druckeranpassung ist ohne Probleme möglich. Im Ordner **UTILITY.OMI** befinden sich bereits zwei Anpassungen, eine für den NEC P6 und eine für einen EPSON-kompatiblen 9-Nadler. Die ASCII-HEX-Druckerdateien können mit dem als Sourcetext vorhandenen **MAKE_CFG.SRC** in das endgültige CFG-Format gebracht werden. Falls Sie Besitzer von 1st Word (Plus) sind, brauchen Sie dessen Druckertreiber nur in **OM-ASSEM.CFG** umzubenennen (Wegen des Bindestriches geht das am besten mit dem Debugger: NAME OLDNAME,NEWNAME). Der Aufbau des Druckertreibers kann dem Anhang (siehe Kapitel D.5 auf Seite 194) entnommen werden. Wenn der Drucker ausgeschaltet ist, wird eine Dialogbox ausgegeben (Siehe Abbildung 5.22 auf Seite 66).

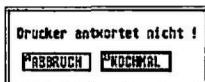


Abbildung 5.22: Der Drucker antwortet nicht...

Das Menü „Suchen“

Der OMIKRONAssembler beherrscht ebenfalls alle nötigen Such- und Ersetzbefehle; er enthält sogar erheblich mehr als jedes andere Programm, da der Assemblerprogrammierer viel spezieller suchen und ersetzen will, als es in



Abbildung 5.23: Das Menü „Suchen“

einem Texteditor möglich ist (Beispiel: Ersetzen Sie mal mit einem Textverarbeitungsprogramm das Symbol „move“ durch „label“, ohne alle „move“-Befehle oder Symbole namens „move2“ ebenfalls zu ersetzen).

Der erste Eintrag lautet „Symbol suchen...“ (F3 oder CONTROL-F). Hiermit kann man z. B. alle Stellen suchen, wo das Symbol „test“ benutzt wird. Um schnell von einer Stelle zur nächsten zu springen, geben Sie das gesuchte Symbol einmal an und drücken RETURN. Wenn das Symbol im Sourcetext vorhanden ist, springt der Cursor sogleich auf die entsprechende Stelle. Mit ALT-F können Sie dann zur nächsten Stelle springen, ohne die Dialogbox erneut zu bekommen. Es stehen zwei Platzhalter zur Verfügung: der Allquantor „*“, der für beliebig lange Textteile steht, und der Existenzquantor „?“, der ein beliebiges Zeichen ersetzt. Im Symbolnamen dürfen beliebig Existenzquantoren (also „?“) verwendet werden. Der Allquantor ist jedoch nur als Abschluß eines Symbols erlaubt (Beispiel: Sie suchen alle Symbole, die mit „A.*“ anfangen). Der Editor erstellt dann einen binären Baum von allen passenden Symbolen und kann so sehr schnell von Symbol zu Symbol springen. Die Buttons erlauben drei Möglichkeiten des Suchens:

- ab der aktuellen Cursorposition,
- ab dem Textanfang
- im markierten Block.

Ein Ändern des Allquantors bzw. des Existenzquantors ist bei Symbolen nicht nötig, da sowohl „*“, als auch „?“ in Symbolnamen nicht vorkommen dürfen.

Sie können „Symbol suchen...“ auch aktivieren, indem Sie mit der linken Maustaste auf ein Symbolnamen doppelklicken, wenn der Cursor auf der

Symboldefinition stand (siehe Kapitel 5.4 auf Seite 28). Dann wird dieser Dialog angezeigt (Siehe Abbildung 5.24 auf Seite 68) und der Symbolname in den Dialog übernommen.

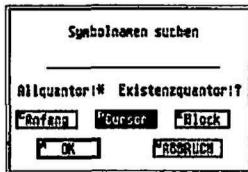


Abbildung 5.24: Der Dialog „Symbol suchen“

Der Eintrag „Symbol ersetzen...“ (SHIFT-F3 oder CONTROL-E) ist wohl die mächtigste Funktion des Assemblers. Mit ihr kann ein Symbol in Sekundenbruchteilen umbenannt werden, mehrere Symbole umbenannt werden oder aber (und das ist wirklich wichtig) nach einem „Block copy“ alle Symbole im Block umbenannt werden. Das spart viele Fehler aufgrund doppelter Deklarationen. Man kann aber auch global alle Zugriffe auf das Label „test“ durch Zugriffe auf das Label „sprung“ ersetzen.

Kommen wir aber nun zur Erklärung des Kommandos: In der Dialogbox stehen zwei Zeilen (Siehe Abbildung 5.25 auf Seite 69). In der obersten wird die gesuchte Symbolmaske, in der unteren die Symbolmaske eingetragen, durch die ersetzt werden soll. Der Button zwischen den beiden Eingabefeldern tauscht die beiden Strings aus. Damit kann man z. B. eine falsche Ersetzung rückgängig machen. Die Buttons „alles“ und „Block“, geben den Suchmodus an. Bei „alles“ wird ein Symbol global umbenannt, bei „Block“ wird ein neues Symbol mit neuem Namen eingerichtet. Falls bei der Symbolersetzung ein Symbol entsteht, welches bereits existiert, erscheint eine Warnbox; nun kann man entscheiden, ob man das Symbol ersetzen will, nicht ersetzen will, aber fortfahren, oder Abbrechen einem sinnvoll erscheint.

Zwei Beispiele:

Alle Symbole im Block sollen zusätzlich `block_` vorangestellt bekommen:
Suchbegriff : *

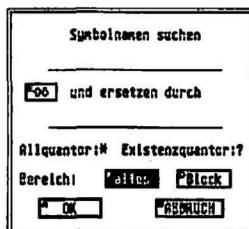


Abbildung 5.25: Die Dialogbox „Symbol ersetzen“

Ersetzbegriff : block_* mit dem Suchmodus „Block“

Das Symbol „unsinn“ soll im Sourcetext durch das bereits vorhandene Symbol „test“ ersetzt werden.

Suchbegriff : unsinn

Ersetzbegriff : test mit dem Suchmodus „alles“

Die Warnbox ist mit „Ersetzen“ zu beantworten.

Der nächste Eintrag in der Menüleiste ist „Text suchen...“ (CONTROL-L). Hiermit können Sie einen ASCII-Text suchen. Dazu wird der Sourcetext Zeile für Zeile disassembliert und dann mit dem Suchstring verglichen (wie beim ASCFIND-Befehl des Debuggers). Das geht natürlich nicht besonders schnell.

Mit den Quantoren (die Sie in beliebiger Anzahl benutzen dürfen; Ausnahme: Dem Allquantor darf kein Existenzquantor folgen, also „*?“ ist nicht erlaubt!) können Sie den Suchbegriff genau einschränken. Wenn Sie nach „*“ oder „?“ suchen wollen, können Sie allerdings andere Zeichen als Quantoren definieren. Quantoren am Anfang eines Suchbegriffs werden überlesen.

„Groß/klein“ legt fest, ob zwischen Groß- und Kleinschreibung unterschieden werden soll. Darunter können Sie einstellen, ab welcher Stelle der Text durchsucht werden soll. „Anfang“ bedeutet „ab dem Textanfang suchen“, „Cursor“ steht für „ab der aktuellen Cursorposition suchen“ und „Block“ sucht nur im Blockbereich.

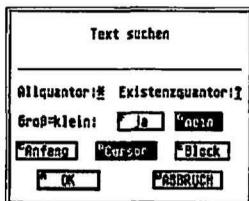


Abbildung 5.26: Der Dialog „Text suchen“

Drücken Sie, nachdem der Assembler etwas gefunden hat, ALT-L, wird die Suche fortgesetzt, ohne daß die Dialogbox wieder erscheint.

Wenn Sie genau wissen, daß der zu suchende Begriff in einem Remark steht, gibt es beim Suchen die Möglichkeit, nur die Remarks zu durchsuchen, indem Sie dem Suchbegriff ein „;“ voranstellen. Das Semikolon wird bei der Suche dann ignoriert; der Editor braucht aber die Zeilen nicht mehr zu disassemblyn, sondern kann gleich die Remarks durchsuchen (Achtung, mehr als 3 gleiche Zeichen werden gepackt und können mit der Remarksuche nicht gefunden werden!).

Beispiel:

Suchbegriff: ;Test

Gefunden : NOP ;Dies ist ein Test

Mit einem Druck auf die rechte Maustaste kann die Suche abgebrochen werden. Damit die Suche nicht unnötig verlangsamt wird, wird die Maustaste nicht ständig abgefragt. Halten Sie die Taste also gedrückt, bis der Suchvorgang abgebrochen worden ist.

Der vierte Eintrag heißt „Text ersetzen...“ (SHIFT-CONTROL-E). Hiermit können Texte gesucht und ersetzt werden. Die dazugehörige Dialogbox (Siehe Abbildung 5.27 auf Seite 71) wird genauso wie bei „Text suchen“ bedient, jedoch mit einigen Erweiterungen:

- Unter „Anzahl“ kann man angeben, ob alle gefundenen Texte ersetzt

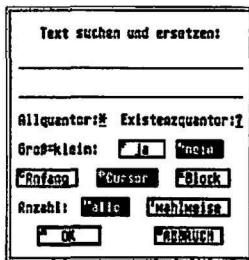


Abbildung 5.27: Der Dialog „Text ersetzen“

werden sollen, oder der Assembler vor jedem einzelnen Ersetzen fragen soll, ob dieser String ersetzt werden soll.

- Der Text, der den alten ersetzen soll, muß auch entsprechende Sternchen (keine Fragezeichen) enthalten. Wenn wir also z. B. alle „RTS“ durch „RTE“ ersetzen wollen, schreiben wir „RTS“ in die erste Zeile und „*RTE*“ in die zweite.

Der folgende Eintrag lautet „Sprung zu Zeile...“ (ALT-Z). Nach ihrem Aufruf erscheint dazu eine Dialogbox (Siehe Abbildung 5.28 auf Seite 71), welche die Eingabe einer Zeilennummer oder eines Symbolnamens erlaubt. Bei einem Symbolnamen wird die Zeile angesprungen, in der das Symbol definiert ist. Bei Symbolen sind Quantoren am Ende des Namens möglich!

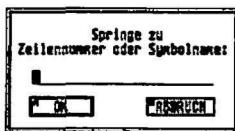


Abbildung 5.28: Der Dialog „Sprung zu Zeile“

Beispiel: A* springt zur nächsten Symboldefinition eines Symbols, das mit einem großen „A“ anfängt. Mit ALT-SHIFT-Z können Sie zum nächsten Symbol springen, ohne vorher die Dialogbox mit RETURN bestätigen zu müssen.

„Nächster Fehler“ (ALT-J) springt zur nächsten Zeile, welche bei einer Assemblierung als fehlerhaft markiert wurde. Falls keine Fehler vorhanden sind, erscheint kurz eine Box mit der Meldung „keine weiteren Fehler“ (siehe Kapitel 5.5.5 auf Seite 56).

Mit „Nächster SFehler“ (ALT-S) springt man zum nächsten Syntax-Fehler, genaugenommen zur nächsten nicht-tokenisierbaren Zeile. Nicht tokenisierbare Zeilen werden beim ASCII-Laden oder Zuladen ja als Remark gekennzeichnet. Man kann mit CONTROL-RETURN einzelne Zeilen als „mit Syntax-Fehler versehen“ kennzeichnen (s. u.).

ALT-SHIFT-S: Wie ALT-S, nur wird aufwärts gesucht.

CONTROL-RETURN: Zeile als nicht tokenisierbar übernehmen (d. h., als Remark, s. u.). Wenn man eine Zeile geändert hat, aber der Editor die Zeile nicht übernehmen will, weil z. B. der Operand noch fehlt (wie das Symbol hieß, muß man erst nachgucken), kann man CONTROL-RETURN drücken. Die Zeile wird dann als Remark übernommen. Zeilen, die so markiert worden sind, kann man mit ALT-S schnell wieder auffinden.

Mit „Nächster Merker“ (CONTROL-S) können Sie zum nächsten Remark-Merker springen. Im Editor-Menü können Sie ein Zeichen als Merker angeben (Voreinstellung ist „~“). Wenn Sie nun am Anfang von Remarks dieses Zeichen angeben, können Sie mit dieser bzw. der nächsten Tastenkombination sehr schnell solche Zeilen anspringen.

Ein Beispiel für Remark-Merker: Sie ändern in einem längeren Programm häufiger drei bestimmte Routinen, probieren dann aus, ob sie jetzt besser funktionieren, ändern dann wieder diese Routinen usw. Damit Sie sich nicht Zeilenummern, Labelnamen o. ä. merken müssen, um diese Routinen wiederzufinden, stellen Sie im Editor-Menü „*“ als Merker ein. Dann schreiben Sie an den Anfang jeder der drei Routinen: „*“ (und meinetwegen noch einen Erklärungstext). Dann können Sie die markierten Stellen mit CONTROL-S problemlos wiederfinden. Der Merker wird in jedem Sourcetext mit abgespeichert.

CONTROL-SHIFT-S sucht im Gegensatz zu CONTROL-S aufwärts.

Das Menü „Block“

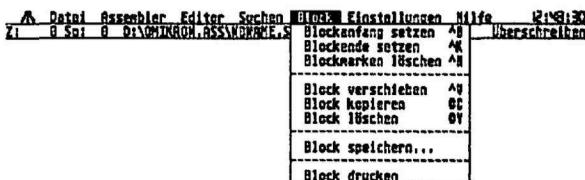


Abbildung 5.29: Das Menü „Block“

Das „Block“-Menü enthält alle Kommandos, die zur Sourcetextbearbeitung notwendig sind:

CONTROL-B: Markiert den Blockanfang (geht auch mit der Maus)

CONTROL-K: Markiert das Blockende (geht auch mit der Maus)

CONTROL-H: Löscht die Blockmarkierungen (Hide Block)

CONTROL-V: Verschiebt den Block an die Cursorposition

ALT-C: Kopiert den Block an die Cursorposition (Copy Block)

ALT-Y: Löscht den Block

Zudem kann man per Tastatur an die Blockgrenzen springen:

ALT-B: Sprung zum Blockanfang

ALT-K: Sprung zum Blockende

Ferner können Sie im „Block“-Menü einen Block speichern sowie einen Block ausdrucken.

Das Menü „Einstellungen“

Darstellung... (SHIFT-F5): Es erscheint eine Dialogbox, in der Sie zwei Dinge einstellen können:

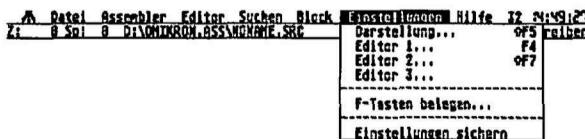


Abbildung 5.30: Das Menü „Einstellungen“

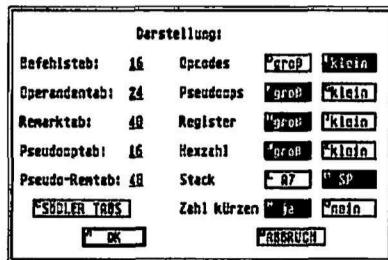


Abbildung 5.31: Die Darstellungs-Dialogbox

1. In der linken Spalte können Sie die Tabulatorpositionen festlegen. Die Labels werden immer am linken Rand (nicht eingerückt) ausgegeben. Alle anderen Einrückungen können frei definiert werden:

Befehlstab: Position der Opcodes

Operandentab: Position der Operanden

Remarktab: Position der Kommentare

Pseudoopstab: Position der Pseudoopcodes (ist normalerweise = Befehlstab)

Pseudo-Remtab: Position der Kommentare, die hinter Pseudoopcodes stehen. Da Pseudoopcodes meist kürzer als Befehle mit Operanden sind, wird dieser Wert meist kleiner als der Remarktab gesetzt.

2. In der rechten Spalte können Sie das Format für den Disassembler beeinflussen. Sie können getrennt für Opcodes (Befehle), Pseudoopcodes, Register (das „D“ bzw. „A“) und die Buchstaben von Hexzahlen einstellen, ob sie in Groß- oder Kleinbuchstaben ausgegeben werden sollen. Sie können wählen, ob „A7“ durch „SP“ ersetzt werden soll (wobei die Einstellung „Register groß/klein“ beeinflußt, wie „SP“ geschrieben wird) und ob führende Nullen bei Zahlen weggelassen werden. Wenn „Zahl kürzen“ ausgeschaltet ist, werden Zahlen entsprechend der Länge angezeigt, in der sie auch im Programmcode abgespeichert sind. Wenn Sie beispielsweise „move.w #\$124,d0“ eingeben, wird die Zahl als „\$0124“ (Wordbreite). Bei „move.l #\$124,d0“ oder „move.w \$124,d0“ wird daraus „\$00000124“. „Kürzen“ bedeutet für den Assembler, daß er die führenden Nullen auf Bytebreite kürzt. Also würde die Zahl in jedem der drei Beispiele als „\$0124“ angezeigt.

Diese Einstellungen haben keinerlei Auswirkungen auf die Funktion des Assemblers. Sie dienen lediglich dazu, die Ausgabe nach Ihrem Geschmack übersichtlicher zu gestalten.

Editor 1... (F4): In der Dialogbox (Siehe Abbildung 5.32 auf Seite 76) können verschiedene Einstellungen gemacht werden. Oben wird die Cursorform festgelegt. Die Ziffern bedeuten folgendes:

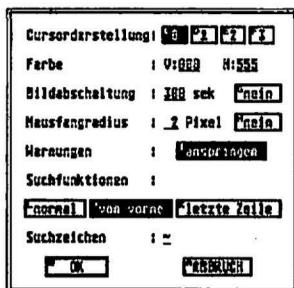


Abbildung 5.32: Die Dialogbox „Editor1“

0 = █, 1 = |, 2 = _, 3 = □.

Wenn Sie auf einem Farbmonitor arbeiten, können Sie die Bildschirmfarben frei wählen. „V“ steht dabei für die Vordergrund-(Zeichen-)Farbe, „H“ für die Hintergrundfarbe. Die einzelnen Ziffern stehen für den Rot-, Grün- und Blauanteil der Farbe. Es sind jeweils Werte zwischen 0 und 7 zulässig. Auf Monochrommonitoren können Sie mit der Hintergrundfarbe (Bit 0 setzen oder löschen) zwischen normaler und inverser „Farbe“ umschalten.

Damit das Bild nicht in Ihren Monitor einbrennt, wenn Sie eine längere Pause einlegen, ohne den Rechner auszuschalten, enthält der OMICKRON Assembler einen Bildschirmabschalter. Wenn er eingeschaltet ist und Sie innerhalb der eingestellten Zeit keine Taste gedrückt und die Maus nicht bewegt haben, wird der Bildschirm schwarz. Das Bild kommt zurück, sobald Sie eine Taste drücken (keine SHIFT- o. ä. Taste) oder die Maus bewegen. Abschaltzeiten unter 10 Sekunden sind nicht zugelassen, abgeschaltet wird diese Funktion, indem Sie auf den „nein“-Button klicken. Der Bildschirmabschalter schaltet übrigens nicht die Synchronisation ab.

Mit „Warnungen“ können Sie anwählen, ob bei Sprüngen zu Fehlern (mit CONTROL-ALT-J) auch Warnungen angesprungen werden sollen.

„Suchfunktionen“ gibt an, wie Fehler, nicht tokenisierbare Zeilen usw. angesprungen werden sollen. „normal“ ist, daß der nächste Fehler ab der Cursorposition angesprungen wird. Auf dem letzten Fehler im ganzen Source

bleibt der Cursor auf der fehlerhaften Zeile stehen. Bei „von vorne“ springt der Cursor nach dem letzten Fehler wieder auf den ersten Fehler im Source. Dann gibt es nicht die Einstellung „letzte Zeile“; der Cursor springt, wenn es keine weiteren Fehler im Source gibt, hinter die letzte Zeile des Sources.

Mit „Suchzeichen“ kann das Zeichen eingestellt werden, das auf das Semikolon folgen muß, um diese Zeile mit ALT-S anspringen zu können. Wenn Sie also „#“ einstellen und dann an einen Zeilenanfang „#“ schreiben, wird diese Zeile angesprungen, sobald Sie ALT-(SHIFT)-S drücken (siehe Kapitel 5.5.5 auf Seite 56).

Editor 2... (SHIFT-F7): In dieser Box (Siehe Abbildung 5.33 auf Seite 77) können Sie verschiedenes einstellen. „Sicherheitskopie“ bedeutet, daß beim

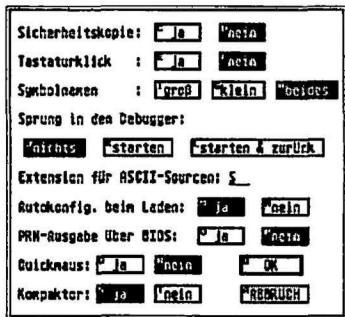


Abbildung 5.33: Die Dialogbox „Editor 2“

Abspeichern von SRC-Dateien eine Sicherheitskopie angelegt wird. Dazu wird erst der alte Source als BAK-Datei umbenannt, dann wird der neue mit SRC-Extension gespeichert. Dadurch kann auch nichts passieren, wenn beim Abspeichern der Strom ausfällt (dann haben Sie wenigstens noch den alten Source).

Unter „Tastaturklick“ können Sie den Tastaturklick abschalten, nicht aber den Ton, der bei Erstbenutzung von Symbolen erklingt.

„Symbolnamen“ ermöglicht Unterscheidungen zwischen Groß- und Klein-

buchstaben. Wenn „beides“ eingestellt ist, werden sowohl Groß- als auch Kleinbuchstaben zugelassen. Dann wird die Groß/Kleinschreibung auch zur Unterscheidung herangezogen (d. h., „Hallo“ ist nicht gleich dem Symbol „hallo“). Bei „groß“ und „klein“ werden alle Symbolnamen bei der *Eingabe* entsprechend konvertiert.

Mit „Druckerausgabe über BIOS“ können Sie den Assembler dazu zwingen, die Texte übers BIOS zu drucken. Normalerweise benutzt er eigene Routinen, die schneller als die des Betriebssystems sind. Dies kann störend sein, wenn Ihr Drucker nicht ganz mitkommt oder wenn Sie einen Laserdrucker haben.

Unter „Extension für ASCII-Sourcen“ können Sie die Default-Extension vorgeben. Diese wird dann im File-Selector automatisch verwendet, Sie können aber selbstverständlich auch andere benutzen.

„Sprung in den Debugger“ erklärt dem Debugger, ob ein Programm, das ihm nach dem Assemblieren übergeben wurde, automatisch gestartet werden soll. „starten & zurück“ bedeutet, das in den Assembler zurückgekehrt wird, sobald Ihr Programm beendet wurde (auch wenn Sie mit SHIFT-SHIFT abbrechen, nicht jedoch bei Breakpoints).

Der Assembler speichert bei jedem Source mit ab, wieviel GEMDOS-Speicher freigelassen werden soll (siehe Kapitel 5.5.5 auf Seite 64). Wenn Sie ein Source auf einem MEGA ST 4 erstellen, können Sie beispielsweise 2MB vorgeben. Diesen Source können Sie normalerweise auf einem 1040 ST nicht mehr laden. Haben Sie jedoch „Autokonfig beim Laden“, auf „nein“ gestellt, ignoriert der Assembler den Wert, der im Source gespeichert ist und behält den vor dem Laden eingestellten bei.

„Quickmaus“ bestimmt, ob die Maus überproportional („ja“) oder linear („nein“) beschleunigt werden soll. Das bedeutet, bei „ja“ wird die Maus sehr schnell über den Bildschirm bewegt, wenn Sie die Maus schnell schieben; kann aber genau positioniert werden, wenn Sie sie langsamer bewegen.

Der OMIKRON.Assembler hat einen Kompaktor, der größere Sourcetexte packt. Sie können dann natürlich nicht mehr von älteren Assembler-Versionen geladen werden. Deshalb kann er mit „Kompaktor“ abgeschaltet werden. Das Packen verkürzt die Sourcetexte um 10%–30%, damit sind sie ungefähr 40%–50% kürzer als ASCII-Texte. Bei kürzeren Sourcetexten bleibt der Kompaktor inaktiv.

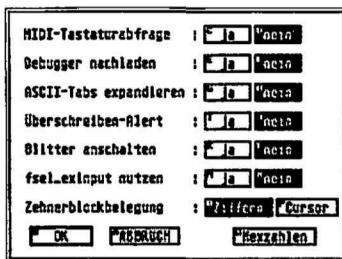


Abbildung 5.34: Die Dialogbox „Editor 3...“

Im der Dialogbox „Editor 3...“ (Siehe Abbildung 5.34 auf Seite 79) können Sie folgendes einstellen:

- Ob über die MIDI-Schnittstelle ankommende Daten als Scancodes ausgewertet werden sollen. Sie senden also von einem zweiten Computer, von einer externen Tastatur o. ä. die Scancodes der gedrückten Tasten, und der Assembler wertet sie so aus, als ob sie auf der normalen ST-Tastatur gedrückt worden wären.
- Ob der Debugger automatisch beim Laden des Assemblers nachgeladen werden soll.
- Ob Tabulator-Zeichen (ASCII-Zeichen 9) innerhalb von Anführungszeichen beim ASCII-Laden expandiert (durch entsprechend viele Spaces bis zur nächsten Tabulatorposition ersetzt) werden sollen. Außerhalb von Anführungszeichen (innerhalb von Remarks) werden sie immer expandiert — innerhalb von Anführungszeichen könnten sie beabsichtigt sein. Deshalb können Sie hier festlegen, ob sie expandiert werden sollen. *Achtung:* Wenn Sie hier „ja“ einstellen, werden auch Tabulatoren in Konstanten expandiert, was zu nicht tokenisierbaren Zeilen führen kann.
- Ob beim Abspeichern vor dem Überschreiben einer bestehenden Datei gewarnt werden soll (Siehe Abbildung 5.9 auf Seite 53).

- Ob ein Blitter genutzt werden soll.
- Ob beim TOS 1.4 die Funktion Fsel_exinput benutzt werden soll. Wenn Sie z. B. einen eigenen Fileselector verwenden, der diese Funktion nicht abfängt, können Sie den Assembler dazu zwingen, Fsel_input zu nehmen.
- Wie der Zehnerblock belegt ist. „Ziffern“ ist die normale, aufgedruckte Belegung. „Hexzahlen“ entspricht der Hex-Tastatur des Debuggers mit CAPS LOCK (siehe Kapitel 6.4 auf Seite 116). „Cursor“ entspricht der IBM-Belegung. Das heißt: Cursorbewegungen mit 8, 2, 4 und 6; Seite vor- und zurückblättern mit 3 und 9; an den Anfang springen mit 7; ans Ende mit 1, ein Zeichen löschen mit . und eins einfügen mit 0. +, - und ENTER sind normal; (,), / und * unbelegt.

F-Tasten belegen:

Sie können jede Funktionstaste noch viermal belegen: Wenn Sie eine Funktionstaste „ohne alles“ oder mit SHIFT drücken, wird eine Funktion aus einem Drop-Down-Menü ausgeführt (wie bei den Funktionen angegeben). Mit CONTROL können Sie nun einen frei definierbaren Text erreichen, einen weiteren mit CONTROL-SHIFT, eine andere Ebene mit ALTERNATE und die letzte mit ALT-SHIFT.

In der Dialogbox (Siehe Abbildung 5.35 auf Seite 81) wählen Sie zuerst die Taste aus, die belegt werden soll, indem Sie auf die Tastennummer in der Reihe mit oder ohne SHIFT klicken und „Control“ oder „Alternate“ einstellen. In der Zeile „Text“ erscheint nun die bisherige Belegung, die Sie ändern können. Abgeschlossen wird die Änderung entweder mit „OK“, oder indem Sie eine andere Taste anwählen.

Einstellungen sichern: Der Assembler speichert alle Einstellungen des Darstellungs- und der Editor-Menüs, die Speicheraufteilung, die Funktions-tastenbelegung und ob die Uhr an- oder abgeschaltet ist in einer Datei mit dem Namen OM-ASSEM.INF, die beim Start automatisch geladen wird.

Das Menü „Hilfe“

Die Einträge in diesem Menü haben keine Bedeutung; sie dienen nur als Gedächtnisstütze für die Zeichen der Shift-Tasten (^, ↑ und ☰).

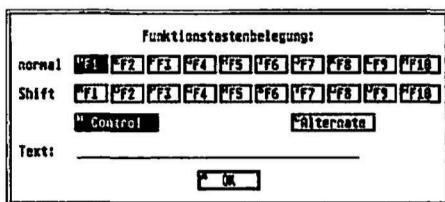


Abbildung 5.35: Die Dialogbox „Funktionstasten belegen“

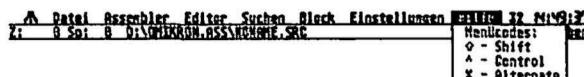


Abbildung 5.36: Das Menü „Hilfe“

Die ASCII-Anzeige

Links neben der Uhr (s. u.) wird in Dezimal der ASCII-Wert des Zeichens unter demm Cursor ausgegeben. Wenn Sie diese Anzeige stört, kann sie mit CONTROL-Ü oder einem Klick auf die Stelle an- und abgeschaltet werden.

Die Uhrzeit

Mit der Taste F10 oder einem Klick auf die Uhrzeitanzeige erscheint eine Dialogbox (Siehe Abbildung 5.37 auf Seite 82). In dieser kann die Uhrzeit und das Datum neu gesetzt werden (d. h., es wird auch die Uhr des Mega STs gesetzt). Außerdem kann man die Anzeige abschalten.

5.6 Die Pseudo-Opcodes

Beispiele zu den Pseudo-Opcodes lassen sich in den diversen Demos finden — insbesondere im „Think and Work“ (GAME.OMI-Ordner).



Abbildung 5.37: Die Dialogbox zum Uhrstellen

In Klammern sind andere Befehlssyntaxen angegeben, die der Assembler versteht, um kompatibel zu älteren Assemblern zu sein. Diese werden dann in den an erster Stelle stehenden Syntax konvertiert. Ein Punkt in Klammern vor einem Befehl bedeutet, daß aus Kompatibilitätsgründen ein Punkt vor dem Befehl ignoriert wird. Ein „.x“ bedeutet, daß sowohl „.B“, „.W“ als auch „.L“ erlaubt sind.

Einige Befehle sind unter verschiedenen Namen implementiert — so z. B. DS und DCB. Dadurch können Sie theoretisch sogar die unter dem OMI-KRON-Assembler geschriebenen Programme wieder auf anderen Assemblern verwenden.

5.6.1 Definition von Datenbereichen

(.)DC.x Wert{,Wert} fügt Konstanten in den Programmcode ein. „Wert“ kann ein beliebiger Ausdruck sein, der auch Formeln enthalten darf. Theoretisch sind sogar Conditions möglich, also z. B.:

`DC.B ^^TIME=$6000`

(Um 12 Uhr wird \$FF, sonst 0 eingefügt) (Daß das sinnvoll ist, hat ja keiner behauptet!). Wenn keine Länge angegeben wird, wird Wordbreite (.W) vorausgesetzt.

(.)DS.x Wert[,Füllwert] (BLK.x) reserviert „Wert“ Bytes (bzw. Words, Longwords) Speicher. „Füllwert“ gibt an, mit welchem Wert der Speicher gefüllt werden soll (wird keiner angegeben, ist der Speicher mit 0 gefüllt). Die Angabe eines Füllwerts ist natürlich nicht im Block-Storage-Segment (BSS) erlaubt. Wenn Sie den „Wert“ direkt ange-

ben, ist maximal 65535 zugelassen; bei Symbolen und Formeln wird natürlich das ganze Symbol bzw. die Formel berücksichtigt.

Ein Beispiel für DC und DS:

Der Text „Bitte geben Sie Ihren Namen ein.“ soll auf dem Bildschirm ausgegeben werden

```
pea    nam      ;Adresse des Textes im Speicher
move   #9,-(sp) ;Funktionsnummer von Cconws
trap   #1        ;Funktion aufrufen
addq.l #6,sp    ;Stackkorrektur,
                  ;6 Byte vom Stack entfernen

;Jetzt gibt der Benutzer seinen Namen ein

lea    buffer,a0 ;Adresse des Eingabebuffers
move.b #14,(a0) ;Maximal 15 Zeichen eingeben
move.l a0,-(sp) ;Adresse auf den Stack
move   #10,-(sp) ;Funktionsnummer von Cconrs
trap   #1        ;Funktion aufrufen
addq.l #6,sp    ;Stackkorrektur

.

.

nam: DC.B 27,'Y35Bitte geben Sie Ihren Namen ein'
;ESC Y positioniert den Cursor

;Der mit DC eingesetzte Text wird
;im Programmcode abgespeichert.

buffer: DS.B 15 ;15 Byte Namensbuffer

;Es wird keine 15 in den Programmcode eingeschoben,
;sondern 15 Byte Speicher reserviert.

DCB.x verhält sich genauso wie DS.x; ist nur aus Kompatibilitätsgründen enthalten.
```

DSBSS.x entspricht fast DS.x, und wird auch zu DS.x konvertiert. Der Unterschied ist nur, daß der GenST den DSBSS-Befehl ins BSS verschiebt, während der OMIKRON.Assembler ihn als einen normalen DS ansieht und den Platz an der Stelle in den Code einfügt, an dem der Befehl steht.

(.)**EVEN (ALIGN.W)** sorgt dafür, daß der nachfolgende Befehl bzw. die nachfolgenden Daten auf einer geraden Adresse liegen. Dazu wird, wenn der PC ungerade ist, ein Null-Byte eingefügt. Dieser Befehl ist meistens überflüssig, da der OMIKRON.Assembler automatisch alle Befehle und DC.W/L auf gerade Adressen legt. Dabei wird die Warnung „Adresse wurde begradiigt“ erzeugt.

DX.B 'String' fügt den String in den Programmcode ein. Im Unterschied zu DC wird der String bis zu einer vorgegebenen Länge aufgefüllt. Dies ist praktisch, wenn Sie eine Tabelle mit gleichlangen Einträgen anlegen wollen, da Sie nicht mehr jeden String einzeln verlängern müssen, bis er die vorgeschriebene Länge hat.

DXSET Länge,[Füllwert] legt die Länge für DX fest. Wird kein Füllwert angegeben, wird 0 genommen. Wenn ein String länger ist, als der bei DXSET angegebene Wert, wird bei der Assemblierung die Fehlermeldung „String zu lang“ erzeugt.

Ein Beispiel:

Von Ihrem Programm soll die gerade ausgeführte Operation oben rechts auf dem Bildschirm angezeigt werden. Dafür ist ein zehn Zeichen breites Feld vorgesehen. Die Texte dürfen nun einmal nicht länger als zehn Zeichen und sollen zweitens mit Spaces aufgefüllt sein, damit der ungenutzte Teil der Anzeige durch die Spaces gelöscht wird. Außerdem können Sie die Adresse eines jeden Textes ermitteln, indem Sie 10 * Nummer zur Adresse des ersten Texts addieren. Also legen Sie die Texte mit DX ab (siehe Tabelle 5.5 auf Seite 85).

Wenn Sie einen zu langen Text eingeben, bemängelt der Assembler das bei der Assemblierung.

Tabelle 5.5: Das DX-Demo

```

DXSET 10, ' ', 10 Zeichen, mit Spaces auffüllen

texte: DX.B  'Laden'
       DX.B  'Speichern'

.
.
```

5.6.2 Daten-Strukturen mit RS

RS.x reserviert Speicher wie DS. Das Label vor dem Befehl wird jedoch nicht auf die absolute Adresse gesetzt, sondern ihm wird der Wert des RS-Counters zugewiesen. Danach wird der RS-Counter entsprechend erhöht. Ein Beispiel: Sie möchten die X- und Y-Koordinate eines Cursors speichern (auf Wortbreite). Mit DS sieht das so aus:

```

XCOORD: DS.W 1
YCOORD: DS.W 1
```

und Sie greifen absolut darauf zu:

```

move #10,YCOORD ; 20 Taktzyklen 8 Bytes lang
clr XCOORD      ; 8 Taktzyklen 6 Bytes lang
```

Mit RS geht es so:

```

rsstart:
XCOORD: RS.W 1      ; XCOORD bekommt den Wert 0
YCOORD: RS.W 1      ; YCOORD bekommt den Wert 2
```

und der Zugriff sieht so aus:

```

lea rsstart,a6      ; Nur einmal erforderlich,
                     ; kostet 12 Taktzyklen
move #10,XCOORD(a6) ; 16 Taktzyklen 6 Bytes lang
clr YCOORD(a6)      ; 8 Taktzyklen 4 Bytes lang
```

Sie sparen bei MOVE und bei einigen anderen Befehlen je 4 Taktzyklen und bei jedem Befehl 2 Bytes. Außerdem wird Ihre Reloziertabelle kürzer, weil ein Adreßregister indirekt mit Offset ja nicht in die Reloziertabelle eingetragen werden muß. Dafür müssen Sie ein Adreßregister frei haben, das auf den Anfang Ihres Block-Storage-Segments (BSS) zeigt. Zum Beispiel OMIKRON.BASIC nutzt diese Adressierungsmöglichkeit sehr intensiv.

RSRESET setzt den RS-Counter auf 0. Damit können Sie z. B. mehrere relative Datensegmente anlegen.

RSSET Wert setzt den RS-Counter auf „Wert“.

RSEVEN begradigt den RS-Counter. Dies geschieht bei RS.W und RS.L automatisch; RSEVEN ist aber praktisch, wenn Sie z. B. zwei Buffer à 79 Byte anlegen wollen, die auf einer geraden Adresse beginnen müssen.

RSBSS addiert zur Länge des Block-Storage-Segments (BSS) den aktuellen RS-Counter. Dann wird der RS-Counter gelöscht. Sie sollten diesen Befehl also ans Ende des relativen Datensegments setzen, damit die BSS-Länge richtig in den Programmheader eingetragen werden kann. Im Gegensatz zu anderen Assemblern kann RSBSS beim OMIKRON. Assembler mehrfach im selben Programm verwendet werden.

Als Beispiel für die RS-Befehle sei auf „Think and Work“ verwiesen.

5.6.3 Segmentierung

(SECTION) (.).TEXT markiert den Anfang des Text-Segments. Da normalerweise jedes Programm mit dem Text-Segment beginnt, kann dieser Befehl auch entfallen. Es ist pro Programm nur ein Text-Befehl möglich. „SECTION TEXT“ wird zu „TEXT“ gekürzt; wir haben es nur aus Kompatibilitätsgründen eingebaut.

(SECTION) (.).DATA kennzeichnet entsprechend den Beginn des DATA-Segments. Das DATA-Segment wird mit dem Programmcode abgespeichert. Es sollte, wie der Name schon sagt, alle Daten des Programms enthalten. Dies ist jedoch nicht unbedingt nötig; Sie können Ihre Daten

auch ins Text-Segment packen. Die Unterteilung dient nur der Übersicht. Manchmal kann Sie aber auch von Nachteil sein: Die Daten im DATA-Segment liegen im Speicher hinter dem Text-Segment, wo durch sich eine große Distanz zwischen Befehl und angesprochenem Datum ergibt. Wenn Sie nun ein Datum mit der PC-relativen Adressierung ansprechen wollen, darf diese Distanz nicht größer als 32 768 bzw. -32 767 Byte sein. Sonst müßte das Datum in den Programmtext eingefügt werden.

(SECTION) (.BSS legt den Start des Block-Storage-Segments fest. Im BSS sind nur DS-Befehle ohne Füllwert, RS-Befehle und natürlich noch EVEN, RSEVEN u. ä. möglich. Das BSS wird nicht mit abgespeichert; es wird nur im Programm-Header (siehe Kapitel D.1 auf Seite 189) eingetragen, wie lang das BSS ist. Dadurch wird Ihr Programm kürzer, als wenn Sie die Variablen ins DATA- oder Text-Segment gelegt hätten. Nach dem Laden des Programms stellt nun das Betriebssystem fest, wie groß das BSS sein soll und reserviert einen entsprechend großen Speicherbereich. Dieser wird gelöscht (also mit 0 gefüllt).

Ein Beispiel:

```
TEXT      ;kennzeichnet den Anfang des Text-Segments,  
          ;kann auch weggelassen werden.  
.  
.        ;diverer Programmcode  
.        ;  
  
lea      buffend,a0  ;Ende des Buffers in A0  
movem.l d0-d7/a1-a6,-(a0) ;Register retten  
pea      copyright   ;Adresse der Copyrightmeldung  
          ;auf den Stack pushen  
move.w #9,-(sp)     ;Cconws - Funktionsnummer  
          ;(Zeile schreiben)  
trap    #1  
addq.l #6,sp        ;Stackkorrektur - 6 Byte addieren  
lea      buffend,a0  
movem.l (a0)+,d0-d7/a1-a6 ;Register wiederholen  
.        ;
```

DATA

;Damit die Daten vom
;Programm getrennt werden

copyright: DC.B "©1988 by Σ-soft",0

BSS	;Der folgende Teil wird ;nicht mit abgespeichert
buffer: DS.L 14	;14 Register Speicherplatz
buffend:	;Ende des Registerspeichers (wegen des ;Predekrents bei MOVEM)

Da der Speicherbereich für die Registerinhalte sowieso variiert, muß er nicht mit abgespeichert werden. Daher wird er in das BSS gelegt.

An dieser Stelle sei darauf hingewiesen, daß man mit ALT-**S** – ALT-**#** an den Anfang des Text-Segments – BSS springen kann.

(.)END kennzeichnet das Ende des Programms. Auch dieser Befehl ist nur aus Kompatibilitätsgründen vorhanden — er kann entfallen. Sämtlicher Code hinter END wird ignoriert.

5.6.4 Symbole

Symbol **(.)EQU Wert** **((.)=|==)** ordnet einem Label einen Wert zu. So können Sie z. B. am Anfang Ihres Programms definieren: etv.timer EQU \$400. Dann müssen Sie sich nicht bei jedem Zugriff auf diese Speicherstelle an die Adresse des Vektors erinnern, sondern können statt move \$400,d0 einfach move etv.timer, d0 schreiben. Wenn .w hinter einem **Symbolaufruf** (nicht hinter der Definition) steht, schreibt der Assembler die Adresse auf Wordbreite (das spart Speicherplatz und Rechenzeit). Das ist nur bei Adressen bis 32767 und Adresse größer gleich \$FFFF8800 möglich. Das obige Beispiel können Sie also auch als move etv.timer.w, d0 angeben.

Symbol (.SET Wert funktioniert ähnlich EQU, jedoch besteht bei mit SET gesetzten Symbolen die Möglichkeit, den Wert nachträglich mit einem weiteren SET-Opcode zu ändern. Als Beispiel sei auf das REPT-Beispiel auf Seite 96 verwiesen.

Symbol (.REG Registerliste hat eine ähnliche Funktion wie EQU, REG legt eine Registerliste für MOVEM fest. So können Sie z. B. definieren: stnrd_r REG D0-A3/A5; bei MOVEM stnrd_r,-(sp) werden dann die Registerinhalte D0, D1, ..., D6, D7, A0, A1, A2, A3 und A5 auf den Stack geschoben.

5.6.5 Assembleroptionen

OPT (X|D|W|P(+|-),... legt einige Optionen fest. Jede Option wird durch einen Buchstaben gekennzeichnet (s. u.), gefolgt durch ein „+“ für „einschalten“ oder ein „-“ für „ausschalten“. Danach können weitere Optionen, durch Kommata getrennt, angegeben werden. Die Buchstaben bedeuten:

- (-) **D** Debugging-Informationen einbinden. Wenn eingeschaltet, wird eine Symboltabelle an das Programm angehängt. Wenn Sie dieses Programm dann im Debugger LISTen, werden alle Symbole mit ausgegeben.
- (-) **X** eXtended Symboltable. Es wird eine erweiterte Symboltabelle im GST-Format eingebunden.
- (+) **W** Warnungen ausgeben. Der Assembler warnt Sie z. B., wenn eine Adresse begradigt wurde oder ein BRA.S zu einem NOP gewandelt werden mußte. Wenn Warnungen ausgegeben werden, heißt das noch nicht, daß sie bei CONTROL-J auch angesprungen werden. Dies ist im Menü „Editor 1“ einstellbar. Ferner warnt der Assembler alle Stellen an, die seiner Meinung nach optimiert werden können. Als optimierbar gelten:
 - bedingte Branches, die in ihre kurze Form gewandelt werden können (z. B. BEQ Label in BEQ.S Label)
 - absolute Sprünge, die in relative gewandelt werden können (z. B. JSR Label in BSR Label)

Dabei werden sowohl Vorwärts- wie auch Rückwärts-Referenzen erkannt. Weiteres zu den Optimierungen finden Sie auf Seite 34.

- (-) P PC-relativen Code überwachen. Wenn diese Option eingeschaltet ist, wird die Assemblierung abgebrochen (fataler Fehler), wenn der Assembler auf nicht PC-relativen Code stößt (Siehe Abbildung 5.38 auf Seite 90).

Das Plus bzw. Minus vor den einzelnen Optionen bedeutet, ob die Funktion normalerweise ein- oder ausgeschaltet ist (Plus bedeutet dabei eingeschaltet).

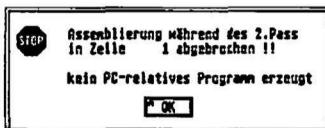


Abbildung 5.38: Abbruch bei OPT P+

5.6.6 Dateioperationen

OUTPUT 'Filename'[, 'Commandline'] gibt den Namen an, unter dem das fertig assemblierte Programm gespeichert werden soll. Wenn ein **OUTPUT**-Befehl im Programm vorkommt, werden der Name und der Pfad dem File-Selector übergeben, so daß Sie nach der Assemblierung nur einmal RETURN drücken müssen, um das Programm zu speichern. Wenn der Filename ohne Extension angegeben wird, wird die Extension vom Fileselector vorgegeben.

Die Commandline wird, wenn angegeben, an das zu debuggende Programm übergeben, wenn dies im Debugger gestartet wird. Der Commandline-Parameter ist z. B. sehr nützlich, wenn Sie ein TTP-Programm entwickeln. Die Environment-Variable wird unverändert weitergegeben — Ihr Programm erhält also die gleiche, die dem Assembler übergeben worden ist.

PATH 'Pfad' gibt den Pfad für IBYTES an. Der Pfad kann maximal 31 Zeichen lang sein.

IBYTES 'Filename'[,Anzahl[,Position]] (INCBIN) fügt eine Datei in den Programmcode ein. Diese Datei wird nicht assembled, sondern so übernommen. Damit können Sie zum Beispiel Graphikbilder oder fertig assembleden Programmcode (position-independent) einbinden. Der Filename darf nicht länger als 19 Zeichen sein. Da das bei längeren Pfaden zu Problemen führen kann, gibt es den Befehl PATH. „Anzahl“ gibt an, wieviele Byte eingefügt werden sollen. Fehlt dieser Parameter, wird die komplette Datei genommen. „Position“ legt fest, wieviele Byte der Datei übersprungen werden sollen. Wenn Sie also z. B. Position 5 angeben, wird die Datei erst ab dem 5. Byte gelesen.

Bei diesen Befehlen können die Fehler „Filename zu lang“ und „Illegaler Filename“ auftreten. Illegal ist ein Filename, wenn er Zeichen wie „*“ oder „?“ enthält.

Ein Beispiel: Wir wollen das DEGAS-Bild „TITEL.PI3“ ohne Farb- und Animationsinformationen einbinden. DEGAS-Bilder sind folgendermaßen abgespeichert:

- 1 Word Auflösung (bei uns 2 für hohe Auflösung) und Kompressions-Kennung
- 16 Words Farbpalette
- 32000 Byte Bild
- 16 Words Animationsinformationen, wenn nötig.

Wir fügen in unser Programm nun ein:

titel: IBYTES 'TITEL.PI3',32000,34

Die ersten 34 Byte werden überlesen, die folgenden 32000 Byte werden eingebunden. Unser Programm kann das Bild nun anzeigen, indem es 32000 Byte ab „titel“ in den Bildschirmspeicher kopiert:

```
OUTPUT  'GRAPHIK.PRG' ;Programmnamen vorgeben
move.w #2,-(sp)      ;Physbase()
trap    #14          ;Bildschirmsadresse ermitteln
```

```

addq.l #2,sp
move.l d0,a1
move.w #7999,d0      ;8000 Langworte = 32000 Byte
;kopieren
lea     titel,a0    ;Adresse des Grafikbildes
loop: move.l (a0)+,(a1)+ ;Die Grafik kopieren
      dbra   d0,loop

.

DATA
PATH  'C:\DEGAS\BILDER.PIC'
;In dem Ordner liegt das Bild
titel:IBYTE 'TITEL.PI3',32000,34
;jetzt wird das Bild eingebunden

```

5.6.7 Allgemeines

BASE (DC.(B|W))| (A0-A7),*| TEXT| DATA| BSS| Formel⁸|
OFF

Relative Adressierung ist zwar schneller als absolute Adressierung, die Berechnung der Offsets erweist sich jedoch als recht umständlich. Um sich die ganzen Formeln zu sparen, stellt der OMIKRON.Assembler einen ziemlich mächtigen Befehl zur Verfügung: BASE.

Die Beschreibung erfolgt am besten in zwei Abschnitten:

1. „BASE Adreßregister,...“ rechnet jedesmal, wenn Sie danach das Adreßregister benutzen und der Wert relokierbar ist, dieses relativ zum danach angegebenen Wert um, oder anders ausgedrückt, der Wert wird subtrahiert.

Beispiel 1. a) (konventionelle Methode):

```

lea     varbase,a4    ;Zeiger auf Variablenliste
move.w wert-varbase(a4),d0 ;Wert holen

```

⁸In diesem Fall muß die Formel mit einem Label beginnen

```

move.l    wert2-varbase(a4),a0 ;2.Wert holen
BSS
varbase:
wert2:ds.l   1
wert:ds.w    1

```

Beispiel 1.b) (Verwendung von BASE):

```

lea      varbase,a4    ;Zeiger auf Variablen-tabelle
base    a4,varbase    ;Bezugsadresse zu A4 setzen
move.w   wert(a4),d0    ;der Assembler setzt hier
          ;automatisch
move.l   wert2(a4),a0    ;den Offset wert-varbase bzw.
          ;wert2-varbase ein
varbase:BSS
wert2:ds.l   1
wert:ds.w    1

```

2. „BASE DC.(B|W)“: Bei jedem folgenden DC-Befehl entsprechender Breite wird der Wert subtrahiert, wenn er relozierbar ist. Zusätzlich steht hier noch als Bezugswert PC zur Verfügung. Damit ist es möglich, Tabellen anzulegen, die man auf folgende Weise ansprechen kann:

Beispiel 2.a) (konventionelle Methode):

```

move.w tabelle(pc,d0),d0    ;Sprungoffset holen
jmp    tabelle(pc,d0)       ;Sprung in Routine
tabelle: dc.w  routine-tabelle,routine2-tabelle,... 
          ;Tabelle mit Sprungoffsets
routine1: ....
routine2: ....

```

Mit dem BASE-Befehl kann die Sprungtabelle vereinfacht werden:

Beispiel 2.b):

```

BASE    DC.W,tabelle
          ;Bezugsadresse zu Label

```

```
;in DC.W-Zeilen
tabelle:dc.w    routine,routine2, ...
;Tabelle mit den Sprungadressen
```

Der Rest des Programms bleibt gleich; bei der Assemblierung werden die Adressen automatisch in den Offset umgerechnet.

Und nun ein Beispiel für BASE DC.W mit Bezugswert * (siehe Systemvariablen (siehe Kapitel 5.5.2 auf Seite 35)):

```
BASE DC.W,*      ;setzt Bezugswert zu den
                   ;Label in DC.W-Zeilen
tab: DC.W 'AB',jump1 ;Jump1: relativ zu tab+2
      DC.W 'CD',jump2 ;Jump2: relativ zu tab+6
      DC.W 'EF',jump3 ;Jump3: relativ zu tab+10
      DC.W 'GH',jump4 ;Jump4: relativ zu tab+14
;Die folgende Routine durchsucht die Tabelle
;nach der Buchstabenkombination in D0 und
;springt in die zugeordnete Routine
      move.w #'EF',D0 ;"Name" des Unterprogramms
      lea     tab-2(PC),A0 ;Zeiger auf den Anfang
                           ;der Tabelle minus 2
      moveq #3,D1      ;Anzahl der Texte in
                           ;der Tabelle
loop:addq.l #2,A0      ;Zeiger plus 2
      cmp.w  (A0)+,D0 ;Wert in D0 mit
                           ;Tabellenwert vergleichen
      dbeq   D1,loop   ;wenn die Bedingung
                           ;wahr oder D1 gleich
;Null ist wird die Schleife verlassen,
;ansonsten wird D1 um eins erniedrigt
;und zum Label "loop" gesprungen
      bne   error    ;der Wert wurde in der
                           ;Tabelle nicht gefunden
      adda.w (A0),A0 ;Berechnung der Sprungadresse
      jsr    (A0)    ;Aufruf des Unterprogramms
      usw.
```

```
jump1: . . . ;hier folgen die
jump2: . . . ;einzelnen Unterprogramme,
jump3: . . . ;die angesprungen werden
jump4: . . .
```

Dabei ist zu beachten, daß bei der Verwendung eines Labels als Offset zu einem Adressregister stets ein Bezugswert angegeben werden muß:

```
lea      record2(PC),A0
BASE    A0,record2      ;durf nicht fehlen
move.w  y(A0),D0
.
.
.

trecord:
x:  DC.W 0
y:  DC.W 24
l:  DC.W 80
record2:DC.W 5,12,70
```

Der Parameter OFF erlaubt es, die Wirkung des BASE-Befehls wieder auszuschalten.

GLOBAL Symbol{,Symbol} alle in der Liste enthaltenen Symbole werden als global markiert (was daran ersichtlich ist, daß sie zwei Doppelpunkte haben). Das ist für alle Symbole erforderlich, die von einem Linker berücksichtigt werden müssen. Wenn Sie also beispielsweise ein C-Programm schreiben, daß die Assembler-Routine „Test“ aufruft. Im Assembler müssen Sie „Test“, dann für den Linker den Status global geben.

CNOP Offset, Justierung „begradigt“ den Programmzähler, so daß er durch „Justierung“ teilbar ist, d. h.:

- Justierung=2: auf Wordgrenze;
- Justierung=4: auf Longwordgrenze usw.

Dazu wird noch „Offset“ addiert. Achtung: Justierung auf Longwordgrenze bedeutet nur, daß die Adreßdistanz zum Programmanfang durch 4 teilbar ist; oder auf Deutsch: der folgende Befehl liegt nur auf Longwordgrenze, wenn der Programmanfang auch auf einer Longwordgrenze liegt. Das ist aber nicht unbedingt gewährleistet. „Offset“ darf nicht größer als 65 535, „Justierung“ nicht größer als 255 sein.

ORG Adresse[,ⁿ] legt fest, für welche Adresse das Programm angepaßt werden soll. Wenn kein ORG-Befehl im Programm vorkommt, wird eine Reloziertabelle angelegt (sofern es nicht positionsunabhängig ist). Wenn Sie jedoch ein Programm für [EP]ROMs schreiben, also eins, das in einem Modul laufen soll, oder ein Betriebssystem, ist Relozieren natürlich nicht möglich. Dann können Sie mit ORG den endgültigen Adressbereich angeben — der Assembler reloziert dann schon vor dem Abspeichern. Wenn Sie „ⁿ“ hinter die Adresse schreiben, wird das fertige Programm unmittelbar nach dem Assemblieren dort hingelegt. Achtung: Der Assembler prüft nicht, ob der angegebene Adressbereich sinnvoll ist. Dadurch wird es beispielsweise möglich, ein Programm in RAM zu legen, das normalerweise nicht im ST existiert. Wenn Sie aber ORG 8,ⁿ eingeben, wird der Rechner natürlich abstürzen. Es ist nur ein ORG-Befehl pro Sourcetext möglich; wenn mehrere vorkommen, ist der letzte ausschlaggebend.

REPT Anzahl Anzahl darf nicht größer als 65 535 sein, wenn es als Zahl eingegeben wird. Als Symbol ist ein Longword erlaubt. Der in der REPT-Schleife stehende Code wird entsprechend „Anzahl“ eingefügt.

ENDR beendet eine REPT-Schleife.

Ein Beispiel:

```
Zahl: SET    0
      REPT   256
            DC.B  Zahl
Zahl: SET    Zahl+1
      ENDR
```

entspricht: DC.B 0, 1, 2, ..., 255

Ein zweites Beispiel:

```
        moveq   #15,d0
loop:  REPT    16
       move.b d1,(a0) +
       ENDR
       dbra   d0,loop
```

fügt zwischen den moveq und den dbra 16 mal move.b ein. Wenn das Programm ausgeführt wird, wird der Inhalt von D1 256 mal ab A0 in den Speicher kopiert.

ILLEGAL fügt \$4AFC in den Programmcode ein. Dieser Befehlscode ist nicht belegt; wenn der Prozessor versucht, ihn auszuführen, wird Exception #4 (Illegal Instruction) ausgeführt. Auf dem Desktop führt das zu 4 Bomben; der Debugger fängt sie ab. Das kann dazu benutzt werden, um einen „Breakpoint“ gleich in das Programm hineinzuassemblyn.

DEFAULT Nummer In der Dialogbox nach der Assemblierung ist der Button „Nummer“ stets default. Der Parameter „Nummer“ darf dabei einen Wert von 0 bis 12 annehmen, wobei der Wert 0 dem Abbruch-Button entspricht, 1 dem Debugger, 2 Standard, u. s. w. An dieser Stelle soll nochmals an die Tastenkombination ALT-A erinnert werden, die den Sourcecode assembliert und den Default-Button auslöst.

5.6.8 Bedingte Assemblierung

IF[NE] Bedingung Der folgende Teil wird nur assembliert, wenn die Bedingung erfüllt (ungleich 0) ist.

IFEQ Bedingung Der folgende Teil wird nur assembliert, wenn die Bedingung nicht erfüllt (= 0) ist.

IFLT Ausdruck Der Code wird assembliert, wenn der Wert des Ausdrucks < 0 ist.

IFLE Ausdruck Der Code wird assembliert, wenn der Ausdruck ≤ 0 ist.

IFGT Ausdruck Der Code wird assembled, wenn der Ausdruck > 0 ist.

IFGE Ausdruck Der Code wird assembled, wenn der Ausdruck ≥ 0 ist.

ELSE der Folgende Teil wird berücksichtigt, wenn der vorhergehende nicht berücksichtigt worden ist (siehe Beispiel).

ENDC (ENDIF) kennzeichnet das Ende des Bereichs, der nur bedingt assembled werden soll. Der folgende Code wird immer assembled (wenn nicht neue IF-Befehle folgen).

IFD Label Der folgende Code wird ausgeführt, wenn das Label bereits definiert worden ist.

IFND Label Der folgende Code wird ausgeführt, wenn das Label noch nicht definiert worden ist.

FAIL [Text] bricht den Assembliervorgang ab. Wenn Sie keinen „Text“ angeben, meldet der Assembler „Benutzer-Fehler (FAIL)“, sonst wird „Text“ ausgegeben. Dieser Befehl ist im Zusammenhang mit bedingter Assemblierung sinnvoll.

Zum Beispiel:

```
IF ende-anfang<>512
  FAIL "Falsche Prglänge‘‘, ENDC
```

Beispiel für bedingtes Assemblieren: Wenn Sie eine Testversion Ihres Programms erstellen wollen, schalten Sie mit OPT eine Symboltabelle an, die Variable ^SYMTAB hat dann einen Wert von ungleich 0. Sie assemblen damit den ersten Teil der IF-Struktur. Wenn Sie OPT D- schreiben hat ^SYMTAB den Wert 0, damit erstellen Sie die Endversion ihres Programms ohne Symboltabelle.

```
OPT D+
IF ^SYMTAB ;Interne Testversion erstellen
  jmp show.coord ;Maus-Koordinaten anzeigen
prompt: DC.B 'Waddayawannado?',0
```

```
ELSE
jmp password           ;Password abfragen
prompt: DC.B 'What do you want me to do?',0
ENDC
```

Hinweis: Wenn Sie zum zweiten Mal das Label „prompt“ definieren wollen, werden Sie darauf hingewiesen, daß solch ein Symbol schon existiert. Um es trotzdem eingeben zu können, ignorieren Sie die Warnung und drücken RETURN.

Da der IF-Befehl in der Grundversion nicht geschachtelt werden darf, gibt es noch den Befehl SWITCH, der der Programmiersprache C entlehnt ist. Der Befehl ist besonders dort sinnvoll, wo ein Assemblerprogramm für verschiedene Sprachen erstellt werden soll:

Beispiel:

language: EQU 1 ;1=Deutsch, 2=Englisch, 3=Latein

```
SWITCH language ;Ein beliebiger Ausdruck ist erlaubt
CASE 1          ;Bei CASE sind nur Zahlen erlaubt!
DC.B 'Spielende'
CASE 2
DC.B 'Game Over'
CASE 3
DC.B 'Finis ludi'
CASE
      ;Default-Fall, wenn keiner
      ;der obigen CASES zutraf
FAIL 'Unerlaubte Sprache'
      ;dann einen fatalen Fehler ausgeben
ENDS    ;Ende der Konstruktion
```

5.7 Fehlermeldungen des Assemblers

Folgende Fehlermeldungen gibt der Assembler direkt bei der Eingabe aus:

5.7.1 Optimierungen

Die Optimierungs- und Warnmeldungen werden nur ausgegeben, wenn dies nicht mit OPT W- ausgeschaltet wurde (siehe Kapitel 5.6.5 auf Seite 89). Diese Optimierungsmeldungen werden vom Assembler bei der automatischen Optimierung genutzt (siehe Kapitel 5.5.1 auf Seite 34).

Kann zu B??S optimiert werden Diese Meldung bedeutet, daß der bemängelte Branch-(conditionally) Befehl zu einem Branch.Short gekürzt werden kann.

Kann zu relativem Sprung optimiert werden Man kann den JSR-Befehl durch einen BSR- bzw. ein JMP- durch einen BRA-Befehl ersetzen. Eventuell kann beim nächsten Assemblieren die Meldung „Kann zu Bcc.S optimiert werden“ erscheinen.

5.7.2 Warnungen

Adresse wurde begradiigt Wenn z. B. ein Befehl an einer ungeraden Adresse läge, wird vom Assembler automatisch ein Null-Byte davorgesetzt und diese Warnung gegeben.

Offset/Wert wird negativ Diese Meldung erklärt sich wohl selbst.

B??S in NOP gewandelt Wenn ein Branch Short auf die nächste Adresse ausgeführt werden soll, müßte der Assembler als Sprungweite 0 angeben. 0 ist für den 68000er nun aber die Kennung dafür, daß es sich nicht um einen Branch Short, sondern um einen normalen Branch handelt; nach der 0 erwartet er ein Word Sprungweite. Dieses Word ist nun aber wieder der Opcode des nächsten Befehls. Der Branch Short kann nun aber nicht ohne weiteres in einen normalen Branch umgewandelt werden, weil sich dadurch alle anderen Adreßdistanzen änderten.

Deshalb wandelt der Assembler den Branch Short in einen NOP um. Da dies z. B. bei selbstmodifizierendem Code Probleme ergeben kann, warnt Sie der Assembler.

Offset zu groß Ein Branch (oder Branch Short) war zu weit gesetzt.

5.7.3 Sofort auftretende Fehler

Doppelte Deklaration Es wurde versucht, ein bereits definiertes Label noch einmal zu definieren. Wenn das ein Versehen war, können Sie das Label ändern und erneut RETURN drücken. Sie können jedoch auch gleich RETURN drücken — die Zeile wird dann übernommen und Sie können normal weiterarbeiten. Wenn bei der Assemblierung dann jedoch zweimal das Label definiert werden soll, wird wieder mit dieser Fehlermeldung abgebrochen. In Verbindung mit bedingter Assemblierung ist es sinnvoll, Label mehrfach einzugeben; deswegen kann der Assembler durch RETURN dazu gezwungen werden, diesen Fehler zu übergehen.

Filename zu lang Ein Filename darf natürlich maximal 8 Zeichen plus (Punkt plus) 3 Zeichen Extension umfassen.

Illegaler Filename In einem Filenamen wurden Zeichen verwendet, die das GEMDOS nicht erlaubt, z. B. Zeichen mit einem ASCII-Wert kleiner 32.

Keine weiteren Fehler Dieser „Fehler“ gehört eigentlich nicht in diese Liste, er wird nämlich nicht bei der Eingabe einer Zeile, sondern beim Drücken von CONTROL-J oder ALT-J. Wenn Sie mit einer dieser Tastenkombinationen versuchen, einen Fehler anzuspringen, obwohl die Fehlerliste leer ist, erscheint in der Mitte des Bildschirms eine Box mit diesem Text (Siehe Abbildung 5.39 auf Seite 102), die ungefähr eine Sekunde angezeigt wird. Wenn Warnungen vorhanden sind, diese aber gemäß der Einstellung im Editor-Menü nicht angesprungen werden sollen, kann diese Meldung nie auftreten.

Operandensyntax falsch In der Adressierungsart, die für diesen Befehl erlaubt ist, ist ein Syntax-Fehler.



Abbildung 5.39: Keine weiteren Fehler vorhanden

Symbolname zu lang Ein Symbol darf nicht länger als 23 Zeichen sein.

Syntax-Fehler Die ersten Zeichen wurden als Befehl identifiziert, danach ist dann ein Fehler aufgetreten. „move ^d0,a0“ produziert z. B. einen Syntax-Fehler.

Unbekannter Befehl Die eingegebene Zeile kann nicht als Befehl interpretiert werden. Meistens ist der Grund ein Tippfehler; den kann man jedoch recht einfach beheben, da man ja sofort nach dem RETURN darauf hingewiesen wird.

Unerlaubter Operand In einer Formel wurde eine Registerliste verwendet.

Unzulässige Adressierung Prinzipiell gibt es diese Adressierungsart; bei diesem Befehl ist sie aber nicht zulässig. So führt z. B. „cmpl.w #5,A0“ zum Fehler „Unzulässige Adressierung“.

Wert zu groß/zu klein Sie haben versucht, einen Wert einzugeben, der zu groß für diese Befehlsbreite war; d. h. einen Wert

- kleiner als -127 oder größer als 255 bei Bytebreite
- kleiner als -32 767 oder größer als 65 535 bei Wordbreite
- kleiner als -2 147 483 647 oder größer als 4 294 967 295 bei Longwordbreite

Zweiter Operand fehlt z. B. „move #50“

5.7.4 Fehler bei der Assemblierung

Offset zu klein Dieser Fehler tritt nur bei BSR.S mit einer Sprungweite von 0 auf. Da eine Wandlung in einen NOP mit ziemlicher Sicherheit

die Wirkung des Programms verändert. Der BSR.S ist wahrscheinlich in ein BSR.W zu ändern, und der Fehler ist behoben.

Ausdruck zu komplex Die Formel kann so nicht berechnet werden. Versuchen Sie, sie in mehrere einfachere Ausdrücke zu zerlegen.

Im BSS-Segment nicht erlaubt Im BSS sind keine Opcodes o. ä. codeerzeugende Befehle zugelassen.

Division durch Null in einer Formel wurde durch 0 dividiert.

ELSE ohne IF zu einem ELSE-Befehl wurde kein vorhergehendes IF gefunden.

ENDC ohne IF zu einem ENDC-Befehl fehlt ein passendes IF.

EQU-Wert nicht änderbar es wurde versucht, einem mit EQU definierten Symbol mit SET einen anderen Wert zuzuweisen, was aber prinzipiell nicht möglich ist.

Falsche Segmentfolge Die Segmente dürfen nur in der Reihenfolge TEXT — DATA — BSS eingegeben werden.

IF nicht abgeschlossen eine bedingte Assemblierung wurde mit IF begonnen, es fehlt aber der ENDC-Befehl.

Im BSS-Bereich nur DS.X Im Block-Storage-Segment sind nur DS-, RS-, RSEVEN- u. ä. Befehle zugelassen, da es nicht mit abgespeichert wird. Es wird nur im Programmheader die Größe des BSS eingetragen.

Position außerhalb der Datei Bei IBYTES wird eine Position (2. Parameter) angegeben, d. h. es sollen mehr Byte ab dem Anfang überlesen werden, als die Datei überhaupt lang ist. Wenn die Länge (1. Parameter) zu groß ist, wird dieser Parameter einfach ignoriert.

String zu lang Ein bei DX.B angegebener String ist länger, als mit DXSET eingestellt.

Symbol nicht definiert Es wurde in einer Formel o. ä. ein Symbol verwendet, das noch nicht definiert wurde.

Unerlaubte Rechenoperation Sie wollten zum Beispiel mit einer Registerliste rechnen oder haben z. B. zwei Label (also relozierbare Symbole!) miteinander multipliziert.

5.8 Das Zusammenspiel Assembler ↔ Debugger

Der residente Debugger Um vom Assembler in den Debugger springen zu können, müssen Sie den Debugger vom Assembler nachladen oder bereits vor dem Laden des Assemblers resident installieren. Das geht ganz einfach, indem Sie vor dem Start des Assemblers den Debugger starten und mit dem RESIDENT-Befehl wieder verlassen (Es gibt noch weitere Möglichkeiten den Debugger resident zu machen (siehe Kapitel 6.12 auf Seite 174)) Dazu ist mindestens 1 MB RAM nötig; wenn Sie weniger haben, können wir Ihnen nur empfehlen, Ihren Rechner aufzurüsten — für professionelle Arbeiten sind 1 MB bei vielen Programmen erforderlich.

Vom Assembler in den Debugger... Assemblieren Sie Ihren Quellcode mit einem Druck auf F1 (bzw. mit CONTROL-A). Sinnvoll ist es, wenn Sie eine Symboltabelle anmelden: Symboltabelle „Standard“ oder „erweitert“ anklicken (zum Debugger werden dann stets alle max. 23 Zeichen übergeben, unabhängig von der Einstellung). Sie können dann alle Label Ihres Quellcodes im Debugger verwenden. Der Debugger arbeitet voll symbolisch. Ein Klick auf „DEBUGGER“ bringt Sie in denselben. Dabei wird der „DEBUGGER“-Button automatisch default, d. h., bei der nächsten Assemblierung brauchen Sie nur noch RETURN zu drücken, um in den Debugger zu kommen. Für ganz Eilige: Denken Sie an ALT-A. Sie können damit Ihren Sourcetext assemblieren und den Default-Button auslösen. Falls bei der Assemblierung ein Fehler auftritt, springt der Cursor zur fehlerhaften Stelle. Der Befehl DEFAULT 1 macht den „DEBUGGER“-Button zum Default (siehe Kapitel 5.6.7 auf Seite 97)

...und zurück, an die richtige Stelle im Quellcode Nachdem Sie mit Untrace, Breakpoints und anderen hilfreichen Nettigkeiten einen Fehler eingekreist haben, sollten Sie ihn am besten gleich im Quellcode korrigieren.

Im OMIKRON.Assembler brauchen Sie dafür die Stelle im Quellcode nicht zu suchen: Mit einem Druck auf CONTROL-He1p springt der OMIKRON.Debugger zurück in den Editor — genau an die Stelle im Quellcode, an der der PC beim Debuggen gerade stand. Mit CONTROL-SHIFT-HELP springt der Debugger zurück, ohne die Cursorposition im Editor zu ändern.

Die Marker bleiben erhalten Es werden natürlich alle Marker des Assemblers übernommen. Wenn Sie in Ihrem Sourcetext mit CONTROL-5 einen Marker auf die Zeile „mainloop: moveq #5,d0“ setzen und dann in den Debugger springen, steht in der Variable M5 die Adresse dieser Zeile. Sie können sich alle Marker mit der Funktionstaste „Marker“ (SHIFT-F6) anzeigen lassen. Wenn eine Symboltabelle vorhanden ist, wird neben der Adresse auch der Symbolname angezeigt, also in unserem Beispiel „mainloop“. Wenn der Marker nicht in einer Zeile steht, wo ein Label definiert wird, wird ein anderes Label in Light (heller) angezeigt, das in dem internen Binärbaum daneben definiert ist. Links von den Adressen sehen Sie Rechtecke, mit denen Sie an die Adresse „springen“ können: wenn Sie in das Feld des Markers 5 klicken, wird ab der Zeile „mainloop“ gelistet. Wenn der Marker nicht innerhalb des Textbereichs liegt, wird nicht gelistet, sondern ein MEMORY/DUMP angezeigt.

Es werden selbstverständlich die Marker auch an Assembler zurückübergeben. Wenn Sie im Debugger mit LET M5=Adresse den Marker auf eine andere Zeile setzen, rechnet der Assembler aus, auf welcher Zeile der Marker nun stehen muß.

Beim Wechsel zwischen Debugger und Assembler gehen eventuell gesetzte Breakpoints im Debugger nicht verloren; wenn Sie nach einer Neuassembly wieder in den Debugger springen, sind die Breakpoints noch erhalten.

Kapitel 6

Der Debugger

6.1 Vorwort

Assemblerprogramme reagieren besonders empfindlich auf Fehler, da es bis auf wenige Ausnahmen keine Möglichkeit der Plausibilitätsprüfung gibt. Eine Hochsprache (außer teilweise vielleicht C) kann z. B. erkennen, ob

1. der Wertebereich einer Variablen überschritten wird („Overflow“ oder „String too long“)
2. oder ob eine Schleifenkonstruktion fehlerhaft ist („RETURN without GOSUB“ oder „UNTIL without REPEAT“)

Der 1. Fall führt bei Assemblerprogrammen meistens nicht zum Absturz, jedoch ist mit falschen Ergebnissen zu rechnen. Im 2. Fall ist ein Absturz vorprogrammiert; da es in Assembler keine Fehlermeldungen solcher Art gibt, würde der Computer ein „RETURN“ (in Assembler also ein „RTS“) auch dann ausführen, wenn gar kein „GOSUB“ (also ein „JSR“ oder „BSR“) vorhanden ist.

Der Debugger dient nun dazu, die während der Testphase auftretenden Fehler zu lokalisieren, um sie dann beheben zu können. Der Debugger erlaubt es Ihnen, auf komfortable Weise Programme zu testen. Dabei werden praktisch alle durch Programmierfehler bedingte Abstürze abgefangen. Es können

sogar residente Programme (wie Ramdisktreiber) und Accessories debuggt werden.

Besonderheiten des Debuggers:

- Bis auf die Operationen wie Laden und Speichern werden *keine* I/O-Routinen des Atari ST benutzt. Damit können auch Programme debuggt werden, die das Betriebssystem des ST teilweise zerstören.
- Der Debugger verwaltet eine eigene Bildschirmseite. Dies ermöglicht das problemlose Testen von Programmen die Daten auf den Bildschirm schreiben (inbesondere GEM-Programme und Spiele)
- Er ermöglicht auch das Debuggen von kompilierten Programmen einer beliebigen Hochsprache. Eine ans Programm angebundene Symboltabelle wird unterstützt.
- Der Debugger kann an (fast) alle Situationen angepaßt werden.
- Das Debugging kann bequem mit Tastatur und Maus erfolgen.
- ...

6.2 Starten des Debuggers

Als einfachste Möglichkeit können Sie den Debugger starten, indem Sie einfach OM-DEBUG.PRG laden (Wer hätte es gedacht?). Um jedoch den Debugger vom Assembler aus aufrufen zu können, sollte der Debugger stets im Speicher verbleiben. Dabei werden auch alle Abstürze („Bomben“) umgelekt, so daß Sie bei einem Absturz stets in den Debugger gelangen.

Aber auch zum residenten Laden gibt es mehrere Möglichkeiten:

1. Tippen Sie, nachdem Sie den Debugger geladen haben, „RESIDENT“ ein, und beantworten Sie die folgende Sicherheitsabfrage mit „J“.
2. Kopieren Sie den Debugger in den AUTO-Ordner. Beachten Sie dabei jedoch, daß der Debugger möglichst als erstes Programm in den Ordner kopiert wird. Der Debugger wird nun bei jedem RESET automatisch geladen.

3. Weitere Möglichkeiten entnehmen Sie bitte Kapitel 6.12 auf Seite 174.

Wenn der Debugger resident im Speicher liegt, können Sie mit ihm auch vom Desktop aus aufrufen, indem Sie das Programm CALL.PRG starten.

6.3 Für den Anfänger

Es ist nicht nötig, alles zu verstehen, um sinnvoll mit dem Debugger arbeiten zu können. Einige Befehle reichen aus, um die meisten Fehler finden zu können. Für Spezialfälle ist es jedoch wichtig, über ein Höchstmaß an Flexibilität zu verfügen. Wir wollen dem Fortgeschrittenen diese Möglichkeiten nicht vorenthalten.

6.4 Allgemeine Bedienung

6.4.1 Der Bildschirmaufbau

```

        Do PC   Ifacrits Ittraps Skip PC Otr  Nextdnu Disasse List  Switch
#06820200 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
PC=1D18E98 USP=177FFC Fractc 60 Overwrt Marker Bracko Jalo Directt 0x01
+ 0x 07 6B88B888 0D88B888 0B1088B851 0B1088B851 0B0F88B888 0B0F88B888 0B0F88B888 FFFFFFFF
+ 0x 87 6B1088B87 0D81088B87 0B1088B818 0B1088B818 0B1088B818 0B1088B818 0B1088B818 FFFFFFFF

$1D18E0| plot_level5:    move.v  DL,(A1)*
$1D18E0|                 clr.r  M
$1D18E0|                 dbra   D8,plot_level5
$1D18E0| movev.l (SP)*,D8-R5
$1D18E0| rts.
$1D18E0| get_char:      lsa   $SB(R5),R0
$1D18F0| move.M  D9,D2
$1D18F0| mulu  M$15,D2
$1D18F0| add.M  D1,D2
$1D18F0| lsa   $IR(D2,W),R3
$1D18F0| moveq.M $FFER(D9),D3
$1D18F0| lddr  M$16(R0),D3
$1D18F0| lddp  D3
$1D18F0| move.M $FFFF(R0),D3
$1D18F0| move.b (R0),D2
$1D18F0| ext.M  D2
$1D18F0| rts.
$1D1910| verzGtzung:    move.M $#4E28,C9
$1D1910| verzGtzung:    dbre   D8,verzGtzung1
$1D1910| rts.

```

Abbildung 6.1: Die Bildschirmseite

Tabelle 6.1: Die Flags des Statusregisters

Bit		Das SR-Register
15	T	Trace
13	S	Supervisor
10	I2	
9	I1	Interrupt-Maske
8	I0	
		Das CCR-Register
4	X	eXtend — Erweiterungs-Bit
3	N	Negative — Vorzeichen-Bit
2	Z	Zero — Null-Bit
1	V	oVerflow — Überlauf-Bit
0	C	Carry — Übertrags-Bit

Sie sehen oben auf dem Bildschirm die Menüleiste: die Befehle in der ersten Zeile können Sie mit F1 bis F10 aufrufen, die der zweiten mit SHIFT-F1 bis SHIFT-F10. Sie können auch mit der Maus direkt auf den Befehl klicken. Näheres zu den Funktionstasten steht in Kapitel 6.5 auf Seite 117.

Darunter sehen Sie alle Register des 68000 Prozessors mit ihren aktuellen Inhalten: Zuerst der Programmzähler (PC), das ist die Adresse, ab der der nächste Opcode abgearbeitet wird.

Dann folgen der User-Stack-Pointer und Supervisor-Stack-Pointer (Durch USP und SSP abgekürzt).

Rechts davon sehen Sie das Statusregister (SR). Die einzelnen Flags des SR werden durch einen Buchstaben bzw. einer Ziffer angezeigt. Wenn das Zeichen in Light-Schrift dargestellt wird, ist das Flag gelöscht. Mit einem Mausklick auf ein Flag kann sein Zustand gewechselt werden. In Tabelle 6.1 sind die Bedeutungen der einzelnen Flags angegeben.

Den Abschluß der Zeile bildet der aktuelle Befehl (den Befehl, auf den der PC zeigt).

In der nächsten Zeile stehen alle 8 Datenregister des Prozessors (D0–D7) und in der Folgezeile die 8 Adressregister (A0–A7), wobei A7 stets entweder dem

USP oder dem SSP (s. o.), in Abhangigkeit des S(upervisor)-Flags, entspricht.

Sie können alle Register ändern, indem Sie mit der Maus auf die Registeranzeige klicken. Daraufhin wird der Cursor in die Anzeige gesetzt, und man kann einen neuen Wert eintragen. Mit den Tasten \leftarrow und \rightarrow kann man den Cursor nach links und rechts bewegen. Mit UNDO, ESC oder durch Druck auf die rechte Maustaste kann die Eingabe abgebrochen werden — dann wird das Register nicht geändert. Drücken Sie RETURN, um die Eingabe abzuschließen und dem Register einen neuen Wert zuzuweisen.

Neben den Registern werden Ihnen wohl zwei Pfeile auffallen; wenn Sie mit der Maus auf \leftarrow klicken, ändern sich die Registerinhalte, zudem erscheint ein \otimes . Mit den Pfeilen können Sie durch den Cache blättern. Näheres dazu siehe bei Kapitel 6.4.2 und 6.6.5 auf den Seiten 111 und 140.

Die restlichen 20 Zeilen des Bildschirms gehören zum Screeneditor. Dort erscheinen alle Ausgaben. Außerdem werden dort alle Befehle eingegeben. Der Bildschirmeditor erinnert ein wenig an OMIKRON.BASIC oder an den C-64, jedoch ist die Tastaturbelegung (siehe Kapitel 6.4.2 auf Seite 111) weitgehend kompatibel zum Assembler.

Den Cursor können Sie beliebig setzen, indem Sie an entsprechender Stelle mit der linken Maustaste hinklicken. Natürlich kann der Cursor auch mit den Cursortasten bewegt werden.

Wenn Sie die rechte Maustaste drücken, wird der unter der Maus stehende Text bis zu einem der folgenden Zeichen:

!&* \+ \{ } [] = ^ | ^ = . : #()? <= > >>

(oder Space, Zeilenanfang bzw. -ende) an die Cursorposition kopiert (davor wird noch ein Space eingefügt).

Der Sinn ist folgender: Sie lassen sich zum Beispiel einen Programmteil mit LIST anzeigen. Vor einen Befehl möchten Sie Breakpoint 1 setzen. Sie tippen: „B1=“ und klicken mit der rechten Maustaste auf die Adresse vor dem Befehl und drücken RETURN. Genauso kann man auch Textstücke kopieren.

Wenn Sie ein Programm geladen haben, wird nach einem Doppelklick (mit der linken Taste) auf eine beliebige Zahl ab dieser Adresse disassembliert, wenn sie innerhalb des Text-Segments liegt, sonst ein Memorydump angezeigt.

Bei Ausgaben von MEMORY, DUMP, DISASSEMBLE, LIST und SYMBOL-

TABLE wird automatisch weiter ausgegeben, wenn Sie, mit dem Cursor am oberen oder unteren Rand angekommen, scrollen. Dies geht auch mit der Maus, dazu bewegen Sie die Maus an den oberen oder unteren Rand und drücken kurz auf die rechte Maustaste. Wenn auf dem Bildschirm Daten stehen, fängt der Bildschirm an zu scrollen. Stoppen können Sie, indem Sie die Maus in Bildschirmmitte bewegen.

6.4.2 Die Tastaturbelegung

Die Tastaturbelegung entspricht soweit es möglich ist, der Tastaturbelegung der Assemblers, so daß sie wahrscheinlich keine Probleme haben werden. Der Editor verfügt über eine ganze Reihe von Tastaturkommandos (siehe Tabelle 6.2 auf Seite 115).

UNDO: eignet sich sehr gut zur Wiederholung des letzten Befehls.

CONTROL-M: Rettet die Cursorzeile in einem internen Buffer, welcher mit den Einstellungen (siehe Kapitel 6.8 auf Seite 168) gespeichert wird.

ALT-M: Holt die gerettete Zeile aus dem Buffer zurück.

CONTROL/ALT 1-9: bevor man ein Unterprogramm traced, kann man die Bildschirmseite des Debuggers (mit Cursorposition) retten, um später wieder auf der Bildschirmseite arbeiten zu können. Mit 1-9 sind nicht die Zifferntasten des Zehnerblocks gemeint.

Zusätzlich zu diesen Tasten gibt es noch einige, die spezielle Funktionen haben (siehe Tabelle 6.3 auf Seite 116).

ALT- \leftarrow , \rightarrow : Der Debugger merkt sich die letzten 256 Befehle in einem internen Buffer, dem sogenannten Cache. Er rettet zu jedem Befehl den kompletten Registersatz, und Sie haben mit diesen Tasten die Möglichkeit, rückwärts durch die Befehlsliste zu gehen.

Ein Beispiel für die Anwendung: Sie haben mit der Taste F2 ein Unterprogramm ausgeführt. Das Unterprogramm hat natürlich alle Registerwerte geändert. Sie wollen das Unterprogramm jedoch nun nochmals genau durchgehen. Gehen Sie mit ALT- \leftarrow einen Schritt zurück und drücken Sie einfach ALT-INSERT. Nun können Sie das Unterprogramm nochmals testen, vorausgesetzt, das Unterprogramm braucht keine zusätzlichen Variablen.

ALT-CLR/HOME:

Mit ALT-CLR/HOME können Sie zu dem letzten Eintrag im Cache springen. Näheres können Sie bei CACHEGET und CACHECLR (siehe Kapitel 6.4.2 auf Seite 111) nachlesen.

ALT-INSERT:

Mit ALT-INSERT können Sie den aktuellen Eintrag im Cache zum letzten Eintrag machen. Dies ist die gleiche Funktion wie CACHEGET.

CONTROL-P:

Setzt den PC auf die aktuelle Adresse¹

CONTROL-G:

Wie SHIFT-F4 (siehe Kapitel 6.5.8 auf Seite 119).

CONTROL-B:

Um einen Breakpoint zu setzen, positionieren Sie den Cursor in die entsprechende Zeile im Screeneditor. Drücken Sie dann CONTROL-B gefolgt von 0-9 oder A-F für die Nummer des Breakpoints (0-15) (siehe Kapitel 6.6.5 auf Seite 132).

SHIFT-SHIFT:

Wenn Sie ein Programm vom Debugger gestartet haben, können Sie das Programm mit einem Druck auf beide SHIFT-Tasten abbrechen. Dabei sind jedoch einige Dinge zu beachten:

1. Es wird nur abgebrochen, wenn der PC *nicht* in den Tiefen des TOS steht, d. h., wenn Sie eine Funktion wie Bconin() aufrufen, können

¹Mit „aktueller Adresse“ ist die Adresse am Zeilenanfang gemeint.

Sie nicht abbrechen, da der PC stets im TOS ist!

2. Diese Einschränkung hat jedoch den großen Vorteil, daß der Programmabbruch (normalerweise) nur im eigenen Programm stattfindet.
3. Bei ausgeschalteten Interrupts passiert (natürlich) gar nichts!
4. Für die Abfrage klinkt sich der Debugger in Keyboard-Vektor ein. Bei dem Drücken von den Tasten SHIFT-SHIFT wird ein Flag gesetzt, damit dann in der VBL-Routine der eigentliche Abbruch durchführt wird.
5. Ein Abbruch kann im VBL-Interrupt erfolgen oder aber bei TOS-Aufrufen (dies wird nebenbei von OBSERVE erledigt).

CONTROL-HELP:

Der Debugger wird *ohne* Sicherheitsabfrage verlassen! Wenn der Debugger vom Assembler aus aufgerufen wurde, lesen Sie bitte in Kapitel 5.8 näheres über die Schnittstelle von Assembler↔Debugger. Wenn man CONTROL-HELP mit SHIFT zusammen gedrückt, erfolgt *keine* PC-Umrechnung in eine Zeilennummer.

ALT-Zahl:

Wenn Sie eine Zahl über den Zehnerblock eingeben, wird das entsprechende ASCII-Zeichen ausgegeben (wie bei den PC-kompatiblen). Halten Sie ALTERNATE gedrückt und tippen Sie auf dem Zehnerblock einen ASCII-Code ein, z. B. „228“. Lassen Sie dann ALTERNATE los. Es wird „Σ“ ausgegeben.

CAPSLOCK:

Wie zu erwarten werden alle Buchstaben in Großbuchstaben gewandelt. Zudem wird der Zehnerblock mit Hexziffern belegt, damit Listings einfacher eingegeben bzw. größere Änderungen im

Speicher einfacher vorgenommen werden können.
Die Tastaturbelegung können Sie der Tabelle 6.4
auf Seite 116 entnehmen.

²siehe Funktionstastenbelegung, Befehl KEY (siehe Kapitel 6.6.8 auf Seite 164)

³Für alle Tester, die nicht vom Bildschirm abfotografieren wollen, kann die Bildschirmsseite des Debuggers im DEGAS-Format abgespeichert werden.

Tabelle 6.2: Allgemeine Tastenkommandos

$\uparrow \downarrow \leftarrow \rightarrow$	Cursortasten
SHIFT- \uparrow , - \downarrow	Scrollt den Bildschirm ohne Cursorbewegung
SHIFT-CONTROL- \uparrow , - \downarrow	Scrollt den Bildschirm mit Cursorbewegung
SHIFT- \leftarrow , - \rightarrow	Bewegt den Cursor an den linken/rechten Rand
TAB	Der Cursor springt zur nächsten 8er Position
BACKSPACE	Löscht das Zeichen vor dem Cursor
DELETE	Löscht das Zeichen unter dem Cursor und rückt den Rest auf
INSERT	Fügt ein Space an der Cursorposition ein
CONTROL-RETURN	Setzt den Cursor in die linke untere Ecke
SHIFT-RETURN	Schreibt C^2_R
CLR/HOME	Setzt den Cursor in die obere linke Ecke
SHIFT-CLR/HOME	Löscht den Bildschirm und setzt den Cursor nach oben links
ESC	Löscht den Bildschirm ab der Cursorposition
SHIFT-DELETE	Löscht die Zeile ab der Cursorposition
CONTROL-DELETE	Löscht die Zeile, in der der Cursor steht — der Rest rückt auf
CONTROL-INSERT	Fügt eine Leerzeile ein
UNDO	Restauriert die Zeile (vom letzten RETURN)
ALT-M	Schreibt eine Zeile zurück, die mit: gerettet wurde
CONTROL-M	
SHIFT-INSERT	Schaltet den INSERT-Modus an bzw. aus
CONTROL 1-9	Bildschirminhalt \rightarrow Buffer
ALT 1-9	Bildschirminhalt \leftarrow Buffer

Tabelle 6.3: Zusätzliche Tastenkommandos

ALT-←, →	Damit können Sie durch den Cache blättern (entspricht den Pfeilen links neben der Registeranzeige)
ALT-CLR/HOME	Zeigt die aktuellen Register an (entspricht dem ⌘ links von der Registeranzeige)
CONTROL-P	Setzt den PC an die aktuelle Adresse
CONTROL-G	Startet das Programm
CONTROL-B	Setzt einen Breakpoint (siehe Seite 112)
SHIFT-SHIFT	das laufende Programm wird unterbrochen und es wird in den Debugger zurückgekehrt
CONTROL-HELP	Der Debugger wird beendet
CONTROL-ALT-DELETE	Warmstart des Debuggers →RESET
CONTROL-ALT-SHIFT rechts-DELETE	Kaltstart des Rechners, der gesamte Speicher wird gelöscht!!!
ALT-SHIFT-HELP	Der Bildschirminhalt wird abgespeichert ³ .

Tabelle 6.4: Der Zehnerblock

normal und mit SHIFT				mit CAPSLOCK ohne SHIFT			
()	/	*	A	B	C	D
7	8	9	-	7	8	9	E
4	5	6	+	4	5	6	F
1	2	3	Enter	1	2	3	Enter
0		.		0		,	

6.5 Die Funktionstasten



Abbildung 6.2: Die Funktionstastenleiste

Mit den Funktionstasten (Siehe Abbildung 6.2 auf Seite 117) können Sie viele häufig benutzte Befehle des Debuggers abrufen.

6.5.1 F1 — Trace

„Trace“ führt den aktuellen Befehl im Einzelschrittmodus aus. Den aktuellen Befehl sehen Sie über der Registeranzeige. Wenn dann kein Listing Ihres Programms auf dem Bildschirm steht, wird ab dem PC gelistet. Die Zeile, auf der der PC steht, wird immer mit „>“ statt „>“ nach der Adresse (links) gekennzeichnet (Siehe Abbildung 6.1 auf Seite 108).

Beim Einzelschrittmodus wird vor der Befehlsausführung der Bildschirm physikalisch umgeschaltet. Dadurch kann es manchmal zu schwachem Bildschirmflackern kommen; meistens wird dieses jedoch durch eine gute Synchronisation vermieden. Dieses Verfahren hat gegenüber dem logischen Umschalten anderer Debugger den entscheidenden Vorteil, daß das debuggte Programm immer auf den richtigen Bildschirm schreibt.

6.5.2 SHIFT-F1 — Tracesimulator 68 020

„Trace 68 020“ simuliert den Trace-Befehl des 68 020. Es wird das Programm, ähnlich dem normalen Trace, ausgeführt; abgebrochen wird aber bei jedem TRAP oder irgendeinem Sprungbefehl (Branch, Jump, RTS o. ä.). Er ist in der Ausführung etwas schneller als UNTRACE, obwohl er nach jedem Schritt den Opcode am PC (Programm-Counter) testen muß.

6.5.3 F2 — Do PC

„Do PC“ setzt einen Breakpoint hinter den aktuellen Befehl und startet

das Programm. Im Normalfall bewirkt diese Funktion damit das gleiche wie Trace. Wenn der Befehl jedoch ein Unterprogrammaufruf ist, wird das ganze Unterprogramm ausgeführt (so, als wäre es ein Befehl). Ist der Befehl jedoch ein Sprungbefehl, kann es sein, daß das Programm nicht mehr abgebrochen wird, weil ja der Breakpoint nie erreicht wird. Außerdem kann mit „Do PC“ ein Permanent-Breakpoint übersprungen werden. Wenn man auf einem DBRA o. ä. steht, wird die ganze Schleife ausgeführt und das Programm erst wieder abgebrochen, wenn die Schleife verlassen wurde.

6.5.4 SHIFT-F2 — Trace no subroutines

„Trace no subroutines“ führt einen Befehl aus (wie Trace (siehe Kapitel 6.6.5 auf Seite 134)), überspringt aber Unterprogramme (d. h. sie werden nicht im Einzelschrittmodus ausgeführt). DBRA-Schleifen u. ä. werden aber wie bei Trace behandelt. Außerdem kann es Ihnen nicht wie bei „Do PC“ passieren, daß Sie nach einem ausgeführten Branch nicht beim Breakpoint ankommen (Wenn Sie auf einem BRA „Do PC“ drücken, wird sich der Debugger meist nicht wieder melden; bei „Trace no subroutines“ wird nur der BRA ausgeführt.).

6.5.5 F3 — Trace until RTS

„Trace to RTS“ startet das Programm und bricht erst ab, wenn das aktuelle Unterprogramm verlassen wurde. Das heißt, wenn Sie ein Unterprogramm mit Trace gestartet haben und nun den Rest bis zum RTS überspringen wollen, drücken Sie F3. Der Debugger orientiert sich dabei am PC und nicht an RTS-Befehlen; wenn das Unterprogramm also weitere Unterprogramme aufruft, die ihrerseits mit RTS zurückspringen, werden deren RTS-Befehle nicht berücksichtigt. Technisch wurde diese Funktionen relativ einfach realisiert, indem das oberste Langwort auf dem Stack gerettet wird und eine Rücksprungadresse in den Debugger dort abgelegt wird. Wenn in diese Routine durch den RTS gesprungen wird, wird die gemerkte Rücksprungadresse wieder auf dem Stack abgelegt.

6.5.6 SHIFT-F3 — Trace until RTE

„Trace to RTE“ entspricht „Trace to RTS“, es wird jedoch bis zum RTE, also ein TRAP zu Ende getraced (anderes Stack-Format!)

6.5.7 F4 — Trace Traps

„Trace Traps“ wirkt wie Trace. Es werden jedoch auch alle Traps im Einzelschrittmodus ausgeführt. Das wurde bei Trace extra verhindert, da das Durchwählen einer Stringausgabe o. ä. meist sehr langweilig und nicht sinnvoll ist. Sie können sich mit F4 die Trap-Routinen aber trotzdem ansehen, wenn Sie das Betriebssystem kennenlernen wollen oder wenn Sie eigene Trap-Routinen geschrieben haben.

6.5.8 SHIFT-F4 — Go PC

„Go“ startet das Programm ab dem PC, ohne irgendwelche Abbruchbedingungen zu setzen. Diese Taste entspricht dem gleichnamigen Befehl. Abgebrochen wird nur an Breakpoints oder manuell (durch SHIFT-SHIFT, RESET oder einen External Break). CONTROL-G bewirkt genau das gleiche. Der Befehl GO ist auch auf Seite 133 in Kapitel 6.6.5 aufgeführt.

6.5.9 F5 — Skip PC

„Skip PC“ ist der letzte der Trace-Befehle. Der PC wird hinter den aktuellen Befehl gesetzt, ohne diesen auszuführen. Der aktuelle Befehl wird also übersprungen. Das ist z. B. sinnvoll, wenn Sie ein Programm testen, welches gerade den Trace-Vektor ändern will. Das geht natürlich nicht, weil der Debugger diesen unbedingt benötigt. Also überspringen Sie diesen einen Befehl mit F5 und tracen dann normal z. B. mit F1 weiter.

6.5.10 SHIFT-F5 — Insert/Overwrite

„Overwrite/Insert“ schaltet den Autoinsertmodus ein oder aus. Das gleiche erreichen Sie auch über SHIFT-INSERT. In der Menüzeile sehen Sie an dieser

Stelle immer, welcher der beiden Modi gerade aktiviert ist. Wenn „Insert“ aktiviert ist, wird stets an der Cursorposition eingefügt, wogegen bei „Overwrite“ der bestehende Text überschrieben wird (man kann natürlich mit der INSERT-Taste zusätzlich Platz schaffen).

6.5.11 F6 — Directory

„Directory“ entspricht dem Befehl DIRECTORY ohne Parameter. Es wird das Inhaltsverzeichnis des aktuellen Laufwerks alphabetisch sortiert ausgegeben.

6.5.12 SHIFT-F6 — Marker anzeigen

„Marker“ zeigt alle Marker mit Labelnamen an (Siehe Abbildung 6.3 auf Seite 120), die vom Assembler übergeben wurden. Wenn der Debugger nicht vom Assembler gestartet wurde, oder Sie keine Marker gesetzt haben, sind alle Marker gleich Null.



Abbildung 6.3: Marker anzeigen im Debugger

6.5.13 F7 — Memorydump PC

„Hexdump“ zeigt den Speicherinhalt ab dem PC an. Das entspricht dem Befehl MEMORY bzw. DUMP mit PC als Adresse.

6.5.14 SHIFT-F7 — Breakpoints anzeigen

„Breakpoints“ listet alle Breakpoints mit den dazugehörigen Daten (Breakpointart und -zählerstand) (Siehe Abbildung 6.4 auf Seite 121). Dieser Befehl verhält sich somit genauso, wie der Befehl BREAKPOINTS ohne Parameter; allerdings werden die Werte in einer Dialogbox angezeigt, die nach dem Drücken auf den OK-Button wieder verschwindet. Dadurch bleiben die anderen Daten auf dem Bildschirm erhalten.

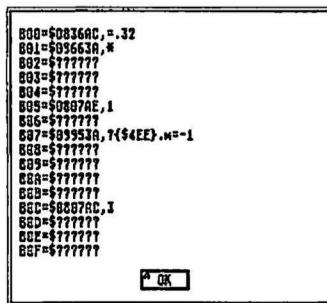


Abbildung 6.4: Die Breakpointanzeige

6.5.15 F8 — Disassemble PC

Es wird ab dem PC disassembliert, als ob Sie „DISASSEMBLE PC“ eingegeben hätten.

6.5.16 SHIFT-F8 — Memoryinfo

„Info“ zeigt die Startadressen von Text-, Data- und Block-Storage-Segment, die Endadresse des freien Speichers, die erste nicht mehr vom Programm benutzte Adresse und eventuell die Anzahl der Einträge in der Symboltabelle an (Siehe Abbildung 6.5 auf Seite 122). Außerdem wird gesagt, wo der Debugger

liegt. Diese Funktionstaste entspricht dem Befehl INFO (siehe Kapitel 6.6.8 auf Seite 160), gibt die Daten aber in einer Dialogbox und nicht auf dem Textbildschirm aus.

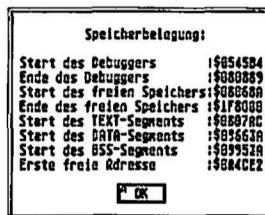


Abbildung 6.5: Die Speicherinfoanzeige

6.5.17 F9 — List PC

Es wird ab dem PC gelistet (wie der Befehl „LIST PC“). Im Unterschied zu Disassemble wird kein Hex-Teil, dafür aber werden eventuell vorhandene Symbole ausgegeben.

6.5.18 SHIFT-F9 — Direct

„Direct“⁴ löscht den Bildschirm und gibt die Einschaltmeldung aus.

6.5.19 F10 — Toggle Screen

„Switch“ schaltet zwischen dem Bildschirm des OMIKRON.Debuggers und des debuggten Programms hin und her. Dabei werden, falls nötig, auch die Auflösung und der Monitor umgeschaltet. Wenn Sie keinen automatischen Monitorumschalter verwenden, müssen Sie dann per Hand umschalten bzw. -stecken.

⁴Warum „Direct“? Ich habe keine Ahnung...

6.5.20 SHIFT-F10 — Quit

Mit „Quit“ kann man den Debugger nach einer Sicherheitsabfrage verlassen (Siehe Abbildung 6.6 auf Seite 123).

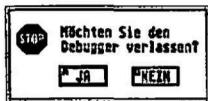


Abbildung 6.6: Programmende des Debuggers

6.6 Die Befehle

6.6.1 Allgemeines

Hinweis: In der folgenden Befehlsbeschreibung sind einige Befehle mit Δ gekennzeichnet. Diese rufen Betriebssystemroutinen auf. Sie müssen also, wenn Sie diese benutzen wollen, darauf achten, daß die entsprechenden Teile des Betriebssystems nicht durch ihr debuggtes Programm gestört wurden. Eingaben sind stets in Anführungszeichen angegeben und in Typewriter. Tastendrücke und Filenamen ohne Anführungszeichen in Typewriter. Wenn lediglich der Befehl im Allgemeinen gemeint ist, wird er in Sans Serif geschrieben.

Die folgenden Befehle können über die Tastatur eingegeben werden (am Ende RETURN drücken). Die meisten Befehle können abgekürzt eingegeben werden. Sie müssen also nicht jedesmal „DISASSEMBLE“ tippen, Sie können auch „DISASS“, „DIS“ oder „D“ eingeben. Wenn jedoch mehrere Befehle mit den gleichen Buchstaben anfangen, kann jeweils einer mit nur einem Buchstaben abgekürzt werden. Sie können zum Beispiel „DUMP“ mit „DU“ abkürzen, wenn Sie jedoch nur „D“ eingeben, interpretiert der Debugger dies als „DISASSEMBLE“. Wie weit Sie jeweils kürzen dürfen, ist bei jedem Befehl angegeben. Wenn Sie einen Befehl nicht vollständig eingegeben haben, muß dahinter vor dem ersten Term ein Space stehen. Beispiel: „DIS 1000“, „D ^A5“ aber: „DISASSEMBLE^A3“ (da der Befehl ausgeschrieben wurde, ist kein Space nötig.)

6.6.2 Schreibweise der Befehle

Bei der Befehlsbeschreibung haben wir für den Syntax eine Schreibweise benutzt, die wohl einer Erklärung bedarf. Angaben in Großbuchstaben und Kommata sind so (oder abgekürzt) einzutippen, wobei es egal ist, ob Sie sie in Groß- oder Kleinbuchstaben oder gemischt eingeben. Angaben in eckigen Klammern können weggelassen werden. Außerdem haben wir als logische Operation | (exklusiv oder) und runde Klammern verwendet.

„Term“ (manchmal auch „Von“ bzw. „Bis“ oder „Adresse“) steht für eine Zahl oder eine Formel, die der Debugger zu einer Zahl ausrechnet. Wenn

also die Syntaxbeschreibung des Befehls lautet: „?Term“ können Sie eingeben: „?5“, „?7+3“ oder „?7*2“. Sie können auch Register einsetzen: „?^A3“ (Der Inhalt vom Register A3 wird ausgegeben) oder „?^d0*2“ (Der Inhalt vom Register D0 mal 2 wird ausgegeben). Die Kennzeichnung durch „^“ ist nötig, um Register von den entsprechenden Hex-Zahlen zu unterscheiden. Ein Beispiel:

Syntax: (BREAKPOINTS|B) [Nr=Adr|K[Nr]]

Als erstes müssen Sie „B“ oder „Breakpoints“ oder eine Abkürzung davon, also z. B. „break“ oder „brE“, eingeben. Das „B“ ist einzeln aufgeführt, da es deshalb nicht mehr als Abkürzung gilt (siehe oben). Sie müssen nach „B“ also kein Space setzen; nach „break“ usw. ist jedoch eins erforderlich.

Der Rest nach „Breakpoints“ kann weggelassen werden, da er in eckigen Klammern steht. Sie können dahinter auch eine Nummer eingeben. Dann ist es noch möglich, „=“ und eine Adresse einzugeben.

Wenn Sie statt der Nummer „K“ eingeben, können Sie dahinter noch eine Nummer schreiben.

Eine vollständige Beschreibung des BREAKPOINT-Befehls können Sie ab der Seite 132 in Kapitel 6.6.5 nachlesen.

Achtung: Alle Endadressen („Bis“) müssen + 1 angegeben werden. D. h., wenn Sie den Bereich von 60 000 bis 60 005 kopieren wollen, müssen Sie „MOVE 60000,60006,...“ eingeben.

6.6.3 Variablen des Debuggers

Sie können in einem Term auch Variablen verwenden:

TEXT: Anfang des Programmspeichers

DATA: Anfang des DATA-Segments

BSS: Anfang des Block-Storage-Segments (BSS)

START: Start des zuletzt mit LOAD geladenen Programms

END: Ende des zuletzt mit LOAD geladenen Programms

BASEPAGE:

BP: Die Adresse der Basepage des mit LEXECUTE geladenen Programms

o⁶ **PC:** Liefert den Wert des Program Counters

o **USP:** Liefert den Wert des User Stack Pointers

o **SP:** Liefert den Wert des aktuellen Stack Pointers (entsprechend dem Supervisor-Bit im Status Register)

SYMBOLS: Anzahl der Einträge in der Symboltabelle

o **SSP:** Liefert den Wert des Supervisor Stack Pointers

o **SR:** Liefert den Wert des Status Registers

o **CCR:** Liefert den Wert des Condition Code Registers

*: Liefert die aktuelle Adresse (siehe Zeilenanfang)

o **DISBASE:** Zahlenbasis des Disassemblers (10 oder 16)

o **BUFFER:** Liefert die Adresse des aktuellen Sektor-Buffers

SEKBUFF: Liefert die Adresse des Default-Sektor-Buffers

TRKBUFF: Liefert die Adresse des Track Buffers (= MEMBASE)

o **TRACK:** Track für READTRACK/SEKTOR usw.

o **SEKTOR:** Sektor für READSEKTOR usw.

o **SIDE:** Seite für READTRACK/SEKTOR usw.

o **DRIVE:** Laufwerk für READTRACK/SEKTOR usw.

o **^B(Term):** Liefert den Wert eines Breakpoints

o **^M(0-9):** Zehn Zahlspeicher als Ersatz für eine Notizzettel (enthält die Marker vom Assembler (siehe Kapitel 5.8 auf Seite 104))

- ^D(0-7): Die Datenregister
- ^A(0-7): Die Adreßregister
- SHIFT: 1 = Abfrage SHIFT-SHIFT wird verhindert. Damit klinkt der Debugger sich nicht vor den Tastaturtreiber.
- CONVERT: zeigt auf eine 256 Byte-ASCII-Konvertierungstabelle. Wenn der Debugger bei MEMORY und ASCII ein Zeichen ausgeben will, wird das Zeichen gemäß dieser Tabelle konvertiert. Der Aufbau der Tabelle ist in Kapitel 6.8 auf Seite 169 beschrieben.
- ACT_P0: Basepage des aktuellen Prozesses
- KLICK: ≠ 0 Disassemble, = 0 Memory/Dump nach Doppelklick auf Adresse
- IKBD: Zeiger auf einen 20 Byte-Buffer, der vor jedem Befehl (wie TRACE, GO u.s.w.) zum Tastaturprozessor gesendet wird. Das erste Byte gibt die Länge des Buffers an. Sinnvoll ist das z. B., wenn Sie eine automatische Joystickmeldung eingestellt haben. Der Debugger muß dies natürlich immer wieder abschalten, damit die Maus und Tastatur normal funktionieren. Dann können Sie den Befehl zur automatischen Joystickmeldung in diesen Buffer schreiben.
- TDELAY: Mit dieser Variablen kann eingestellt werden, wie lange nach einem TRACE (F1-F5) gewartet werden soll. Default ist 0. Bei einigen Rechnern prellt die Tastatur (minimal), deshalb werden bei ihnen bei einem Tastendruck gleich 2 Befehle ausgeführt. Dann können Sie TDELAY auf Werte bis 65 535 hochsetzen. Auf einem unserer ST's hat sich 30 000 als praktisch erwiesen.
- SCROLLD: Scrollverzögerung bei CONTROL. Wenn Sie beim Scrollen die CONTROL-Taste drücken, wird die Geschwindigkeit verringert (da werden Erinnerungen an den 64er wach...). Je größer dieser Wert ist, desto langsamer wird gescrollt. Die Voreinstellung ist 20 000. Möglich sind Werte bis zu 65 535.

UTRACE: Zeigt auf den Speicher, in dem das Programm steht, das bei UNTRACE die Abbruchbedingungen testet. Näheres steht unter dem Befehl UNTRACE. Die Testprogramme für die Breakpoints stehen ab UTRACE-\$2000 und sind je \$200 Byte lang.

UT: Entspricht UTRACE

o **TRACE:** 1 = Disassemblieren bei Trace mit F1, 0 = Listen bei Trace mit F1

o **COL0:** Die Hintergrundfarbe des Debuggers

o **COL1:** Die Vordergrundfarbe des Debuggers

o **REZ:** Setzt die Auflösung des debuggten Programms. Der Debugger stellt die Auflösung dann je nach Bedarf automatisch ein. Sie können also beispielsweise den Debugger auf einem Monochrommonitor starten und nach ^REZ=0 ein Programm in der niedrigen Auflösung debuggen. Wenn Sie einen automatischen Monitorumschalter haben, wird auch der Monitor umgeschaltet.

o **CONTERM:** 1 = Tastaturklick an, 0 = aus

o **AESFLAG:** 1 = Schaltet den Plausibilitätstests bei VDI- und AES-Funktionen aus. (Erklärung siehe OBSERVE (siehe Kapitel 6.6.5 auf Seite 138))

CHECKSUM: Die Befehle READSEKTOR und WRITESEKTOR ziehen automatisch eine Prüfsumme über den Sektor (alle Worte werden addiert). Diese Prüfsumme steht dann in CHECKSUM. Wenn Sie einen Boot-Sektor gelesen haben, können Sie mit „?CHECKSUM“ sehen, ob er ausführbar ist (das ist er bei \$1234)

COLORS: Zeigt auf die Farbpalette des debuggten Programms

o **SMALL:** 1 = Hex-Zahlen in Kleinbuchstaben, 0 = in Großbuchstaben ausgeben

- o SIZE: gibt die Breite für MEMORY und DUMP an. Es sind Werte zwischen 1 und 16 zugelassen. Es werden entsprechend viele Bytes ausgegeben.
 - o LINES: Anzahl der Zeilen, die normalerweise bei LIST, ASCII, MEMORY usw. ausgegeben werden. Voreinstellung ist 16, es sind Werte bis 255 möglich.
- MEMBASE:** Zeigt auf den Anfang des freien Speichers (des größten zusammenhängenden freien Speicherblocks)
- SAVEAREA:** Zeigt auf einen Buffer, in den beim Start des Debuggers der Bereich 0-\$4FF kopiert wird. Bei RESET ALL (siehe Kapitel 6.6.8 auf Seite 165) und beim Verlassen wird dieser Bereich zurückkopiert. Wenn Sie beispielsweise die Variable _drvbits ändern wollen und den neuen Wert direkt nach \$4C2 schreiben, wird sie beim Verlassen des Debuggers zurückgesetzt. Um das zu verhindern, müssen Sie den Wert auch nach „SAVEAREA+\$4C2“ schreiben.

⁵mit o gekennzeichnete Variablen können mit SET bzw. LET geändert werden.

6.6.4 Die Formelauswertung

Die Zahlen können gemäß Tabelle 6.5 in verschiedenen Zahlensystemen angegeben werden.

Tabelle 6.5: Die Zahlensysteme

%	Binär
.	Dezimal
\$	Hexadezimal
"oder '	ASCII

Wenn keine Zahlenbasis angegeben wird, wird das Hexadezimalsystem angenommen. ASCII-Stringe bis zu vier Zeichen werden als Zahl gewertet.

Tabelle 6.6: Die erlaubten Vorzeichen

+	Plus (wer hätte es gedacht)
-	Minus (auch nicht schlecht) (2er Komplement)
-	NOT (1er Komplement)
!	logisches NOT (Ergebnis ist 0 oder -1)

Die Formelauswertung des OMIKRON.Debuggers funktioniert genauso wie im Assembler, d. h. es wird (natürlich) Punkt- vor Strichrechnung beachtet, zudem stehen neben den Grundrechenarten alle für Assemblerprogrammierer wichtigen Rechenarten zur Verfügung (siehe Tabelle 6.6 und Tabelle 6.7). Zudem können Sie mit geschwungenen Klammern indirekt adressieren: {^A4} hat als Wert den Inhalt der Adresse, auf den das Register A4 zeigt. Hinter der } kann optional noch ein „.B“, „.W“ oder „.L“ angegeben werden, welches die Breite der Adresse angibt (Wortbreite ist Default).

Tabelle 6.7: Die erlaubten Rechenzeichen

Rechenzeichen	Funktion:
+	Addition
-	Subtraktion
	OR
-	EOR (XOR)
<<	Linksverschiebung (LSL-Befehl des Prozessors)
>>	Rechtsverschiebung (LSR-Befehl des Prozessors)
*	Multiplikation
/	Division
&	AND
%	Modulo

/

6.6.5 Trace-Funktionen

Breakpoints

Syntax : (BREAKPOINTS|B) [Parameter]
Parameter : Nummer=Adresse,[*|=Wert|Anzahl|?Bedingung]
 K[Nummer]
Kurzform : B

Die Nummer ist ein Term, dessen Wert zwischen 0 und 15 liegt.

„BREAKPOINTS“ allein zeigt alle Breakpoints an, „BREAKPOINTS+Nummer“ nur einen. „BREAKPOINTS K“ löscht alle Breakpoints. Es kann natürlich auch nur ein einzelner Breakpoint gelöscht werden, indem man „BREAKPOINTS K+Nummer“ eingeibt. Wenn „BREAKPOINTS Nummer=Adresse“ eingegeben wird, wird der entsprechende Breakpoint auf die Adresse gesetzt.

Beispiele:

- „Break 5=pc+.100“ setzt Breakpoint Nummer 5 100 Byte hinter den Program Counter.
- „BREAKPOINTS K“ löscht alle Breakpoints
- „B“ zeigt alle Breakpoints an
- „BK 3“ löscht Breakpoint 3

Wenn ein Breakpoint ausgeführt wurde, wird er automatisch gelöscht. Es gibt neben diesen „normalen“ Breakpoints aber auch andere, die Sie setzen können, indem Sie nach der Adresse noch ein Komma, gefolgt von der Art, angeben:

1. „,Anzahl“ setzt einen Stop-Breakpoint. Der Breakpoint muß „Anzahl“ mal aufgeführt werden, bis das Programm abgebrochen wird. Dann wird der Breakpoint gelöscht. Wenn Sie sich mit B die Breakpoints anzeigen lassen, steht hinter jedem Stop-Breakpoint, wie oft er noch ausgeführt werden muß.

2. „,*“ setzt einen Permanent-Breakpoint. Dieser Breakpoint funktioniert wie der normale, er wird aber nicht gelöscht. Er kann nur durch manuelles Löschen (BK) beseitigt werden.
3. „,= Wert“ setzt einen Counter-Breakpoint. Dabei handelt es sich eigentlich um keinen echten Breakpoint, der Programmablauf wird nämlich nie abgebrochen. Es wird nur mitgezählt, wie häufig der Breakpoint durchlaufen wurde. „Wert“ stellt den Anfangswert dar; wird kein Wert angegeben, wird 0 genommen. Wenn Sie sich die Breakpoints mit B anzeigen lassen, steht hinter jedem Counter-Breakpoint, wie häufig er erreicht wurde.
4. „,?Bedingung“ setzt einen Bedingungs-Breakpoint. Hinter dem Fragezeichen steht eine Bedingung wie bei IF (siehe Kapitel 6.6.5 auf Seite 136); es wird abgebrochen, wenn der Wert der Bedingung ungleich 0 ist. Dann wird der Breakpoint gelöscht. Das erzeugte Programm kann bis zu 512 Byte lang sein und steht ab der Adresse:
„UTRACE-2000+200*Breakpointnummer“

Go

Syntax : GO [Adresse][,Breakpoint]
Kurzform : G

startet ein Programm. Wenn keine Adresse angegeben wird, wird ab PC gestartet. Es kann auch ein Breakpoint angegeben werden, der sonst nicht erreicht werden kann: Nummer 17. Er wird vor dem Start gesetzt und danach wieder gelöscht. Sie können also z. B. eingeben: „GO 10000,20000“ — dann wird das Programm von \$10 000 bis \$20 000 ausgeführt. Da die Endadresse intern als Breakpoint verwaltet wird, darf sie nicht im ROM liegen. Wenn Sie nicht eine Endadresse, sondern die Zahl der auszuführenden Befehle angeben wollen, benutzen Sie TRACE.

Beispiel: „G“
Startet ein Programm dort, wo es abgebrochen wurde, bzw. nach LEXECUTE am Anfang.

Trace

Syntax : TRACE [Anzahl]
Kurzform : T

Trace führt einen (beziehungswise eine Anzahl) Befehl(e) aus. Danach wird wieder in den Debugger zurückgesprungen. Der Debugger zählt wirklich die ausgeführten Befehle und setzt keine Breakpoints (so, als ob Sie „Anzahl“ mal F1 gedrückt hätten). Das heißt, es funktioniert auch im ROM.

Showmemory

Syntax : SHOWMEMORY [Term,[B|W|L][Anzahl]]|[,.Nummer]
Kurzform : SH

Sie können sich wichtige Speicherbereiche auch ständig anzeigen lassen. Dazu geben Sie die Adresse mit SHOWMEMORY ein; es wird dann unter der Registeranzeige eine weitere Zeile eingefügt, in der, wie bei MEMORY, der Speicherinhalt ab dem Wert des Terms angezeigt wird. Diese Anzeige wird laufend aktualisiert, also nach jedem Trace, Return usw.

Wenn der Term Register beinhaltet, wird die Adresse jedesmal neu berechnet. Das bedeutet: „SHOWMEMORY ^A0+100“ zeigt den Speicherinhalt ab dem Inhalt von Adreßregister A0+\$100 an, wobei die Adresse immer neu berechnet wird, wenn A0 sich ändert.

Mit dem B, W oder L kann angegeben werden, ob in Byte-, Word- oder langwortbreite ausgegeben werden soll. „Anzahl“ legt fest, wieviele Byte in der Zeile angezeigt werden sollen.

Vor jeder Zeile der Anzeige steht eine Nummer. Eine Zeile kann wieder gelöscht werden, indem Sie „SHOWMEMORY ,.Nummer“ eingeben.

Do, Call

Syntax : DO|CALL [Adresse][,Breakpoint]
Kurzform : DO oder C

führt ein Unterprogramm ab der Adresse aus. Das Unterprogramm sollte mit einem RTS abgeschlossen sein - es kehrt dann wieder in den Debugger zurück. Wenn weder Adresse noch Breakpoint angegeben werden, wird ein Breakpoint hinter den nächsten Befehl gesetzt und der aktuelle ausgeführt. Normalerweise entspricht DO ohne Adresse/Breakpoint also TRACE ohne Term, nur wenn der aktuelle Befehl ein JSR oder BSR ist, wird bei DO das ganze Unterprogramm ausgeführt. Bei JMP, BRA o. ä. ist DO meist nicht sinnvoll, da der PC wohl nicht so schnell hinter dem JMP (o. ä.) landen wird.

Achtung: CALL ist der einzige Fall, in dem der Debugger unter Umständen den TRAP # 3 belegt.

Untrace

Syntax : UNTRACE [Adresse] [,Anzahl]
Kurzform : U

Mit UNTRACE können Sie ein Programm wie mit TRACE durchlaufen lassen; der Debugger kontrolliert aber nach jedem Befehl, ob eine Abbruchbedingung erfüllt ist. Diese Bedingungen werden mit IF gesetzt.

Der Programmablauf wird dabei natürlich erheblich verlangsamt, weil pro Befehl des Programms eine ganze Reihe von Vergleichsbefehlen des Debuggers ausgeführt werden müssen.

Außerdem speichert der Debugger die letzten 256 Befehle in einem sogenannten Cache. Sie können die Befehle mit den zugehörigen Registerinhalten nachträglich ansehen. Dazu klicken Sie auf einen der Pfeile, die links neben der Registeranzeige stehen.

Mit dem Pfeil nach links „blättern“ Sie einen Befehl zurück, das heißt, Sie sehen den Befehl, der zuletzt ausgeführt wurde sowie die Registerinhalte. Mit dem Pfeil nach rechts „blättern“ Sie dementsprechend wieder nach vorn. Mit dem „Fuller“ oben links können Sie zum aktuellen Befehl springen. Mit der rechten Maustaste können Sie gleich neun Befehle überspringen, also in Zehnerschritten blättern.

If

Syntax : IF Term
Kurzform : IF

Wenn Sie IF ohne Parameter angeben, werden die Bedingungen für UNTRACE angezeigt, sonst werden sie gesetzt. In der Abbruchbedingung können Sie den PC, alle Register incl. SR, CCR, USP, SSP, SP und Speicherinhalte abfragen und miteinander oder mit Konstanten vergleichen. Als Vergleiche sind =, ≠, ≤ und ≥ zugelassen. Außerdem sind die logischen Operationen & (and), | (or), ^ (xor) und ~ (not) zugelassen. Auch Klammern können gesetzt werden.

Der Debugger erzeugt aus der Zeile ein Programm, daß die Bedingungen entsprechend abfragt und das Zero-Flag setzt bzw. löscht. Die Variable UT (wie UNTRACE) zeigt auf den Anfang dieses Programms; Sie können es sich also mit „L UT“ ansehen und eventuell verändern. Dadurch ist es Ihnen auch ermöglicht, schwierigere Abbruchbedingungen zu testen. Das Programm erhält in A1 einen Zeiger auf eine Registertabelle; d. h., die Registerinhalte des debuggten Programms stehen in einer Tabelle, auf deren Anfang A1 zeigt.

Die Tabelle hat das gleiche Format wie bei GETREGISTER erforderlich:

```
1 Langwort D0
1 Langwort D1
.
.
.
1 Langwort D7
1 Langwort A0
.
.
.
1 Langwort A7
1 Langwort PC
1 Langwort USP
1 Langwort SSP
1 Word SR
```

Zurückgeben müssen Sie nur, ob UNTRACE abgebrochen werden soll (dann muß das Zero-Flag Z=0 gelöscht werden) oder nicht (Z=1). Das Programm muß mit RTS beendet werden. Es darf alle Registerinhalte (natürlich außer A7) verändern. Es wird im Supervisormodus aufgerufen (der muß auch an bleiben!).

Ihr Programm sollte natürlich so kurz und schnell wie möglich sein, weil es ja nach jedem Befehl des debuggten Programms einmal ausgeführt werden muß - da kann jeder Taktzyklus Minuten ausmachen.

Der Debugger bemüht sich, die Abfrage in ein kurzes Programm zu übersetzen - da aber kein hochoptimierender Compiler eingebaut werden konnte (der Debugger darf ja auch nicht zu lang werden), können die Programme meist noch wesentlich optimiert werden. *Achtung:* Wenn Sie ein Register mit einem Wert vergleichen wollen, schreiben Sie immer „Register=Wert“ und niemals „Wert=Register“, da der Debugger dann etwas kürzen kann. Nun noch ein paar Beispiele für IF-Befehle und den daraus erzeugten Programmcode:

```
IF ^D1.w=100 ;Bricht ab, wenn das D0 (auf Wortbreite)
               den Wert $100 hat. "^" kennzeichnet,
               das es sich um ein Register und nicht
               um die Speicherstelle D0 handelt.
moveq #0,D0
move.w 6(A1),D0
cmpi.l #$100,D0
seq    D0
rts
```

```
IF 100=^D1.w ;Gleiche Abfrage wie oben; jedoch andersherum
               eingegeben
move.l #$100,-(SP)
moveq #0,D0
move.w 6(A1),D0
cmpi.l (SP)+,D0
seq    D0
rts
```

```
IF ^D1=100      ;Schneller geht's mit einem Long-Vergleich
    cmpi.l #$100,4(A1)
    seq    D0
    rts
```

```
IF 100=^D1      ;Wenn man das Register hinten angibt, ist der
                  Vorteil wieder dahin...
    move.l #$100,-(SP)
    move.l 4(A1),D0
    cmp.l  (SP)+,D0
    seq    D0
    rts
```

```
IF {${4ee}.w=-1}&(^D2.1>.10)  ;Nun 'was Komplizierteres:
    lea     $4EE.w,A0
    cmpi.w #$FFFF,(A0)
    seq    D0
    ext.w  D0
    ext.l  D0
    move.l D0,-(SP)
    cmpi.l #$A,8(A1)
    shi    D0
    ext.w  D0
    ext.l  D0
    and.l  (SP)+,D0
    rts
```

Observe

Syntax : OBSERVE [Trap][,[?|-1|Liste]]
 Kurzform : 0

OBSERVE ermöglicht es, alle Betriebssystemaufrufe eines Programms abzufangen. Es werden die entsprechenden Trap-Vektoren verbogen, so daß Sie ein Programm nicht UNTRACEn müssen, um OBSERVE zu benutzen.

- OBSERVE ohne Parameter zeigt alle Trap-Breakpoints an, die zur Zeit gesetzt sind.
- OBSERVE OFF schaltet OBSERVE ganz (!!!) aus. Die Trap-Vektoren werden nicht mehr geändert.
- OBSERVE Trapnummer setzt alle Trap-Breakpoints für diesen Trap.
- OBSERVE Trapnummer. löscht alle Trap-Breakpoints für diesen Trap.
- OBSERVE Trapnummer,? zeigt die Trap-Breakpoints für diesen Trap an.
- OBSERVE Trapnummer,liste setzt einzelne Trap-Breakpoints. Mit einem Punkt gekennzeichnete Breakpoint-Nummern werden gelöscht.
Beispiel: „OBSERVE 14,A,20.“
Es wird Flopfmt() (XBIOS 10), aber nicht mehr Dosound() (XBIOS 32) abgefangen.

Gemdos 0 (Pterm0), 76 (Pterm), 49 (Ptermres) und TRAP # 2 Funktion 0, sowie CONTROL-C bei GEMDOS-Funktionen werden immer abgefangen (wenn OBSERVE nicht ausgeschaltet ist!)

Wenn das Programm auf einen Trap-Breakpoint läuft, meldet sich der Debugger mit dem Namen der aufgerufenen Funktion (bevor die Funktion ausgeführt wird). Die übergebenen Parameter werden in einem C-ähnlichen Syntax ausgegeben: Hinter dem Funktionsnamen werden in Klammern die Parameter, durch Kommas getrennt, angezeigt. Vor jedem Parameter steht „w:“ für Wordbreite oder „l:“ für Langwortbreite. Bei AES- und VDI-Funktionen werden nur die Adressen der Arrays ausgegeben. Wenn Sie sich dann mit OBSERVE oder „OBSERVE x,?“ die beobachteten Traps anzeigen lassen, wird der Trap, bei dem abgebrochen wurde, mit einem „*“ gekennzeichnet. Das Programm kann mit GO fortgesetzt werden.

Bei den Trapnummern steht:

GEMDOS Nummer 1

AES Nummer 2A

VDI Nummer 2B

BIOS Nummer 13 oder D

XBIOS Nummer 14 oder E

Außerdem werden bei jedem Trapauftruf – unabhängig von OBSERVE – die Parameter einem Plausibilitätstest unterworfen. Bei einigen Programmen ist das störend, weil sie für ungenutzte Arrays als Adresse 0 angeben, was der Debugger bemängelt. Deshalb kann dieser Test mit der Variable AESFLAG (siehe Kapitel 6.6.3 auf Seite 128) unterbunden werden.

Cacheclr

Syntax : CACHECLR
Kurzform : CACHEC

Löscht den Cache, d. h. alle Register werden auf 0 gesetzt (bis auf die aktuellen natürlich).

Cacheget

Syntax : CACHEGET
Kurzform : CACHEG

GETCACHE holt die Werte der Registeranzeige, die momentan angezeigt werden, zurück in die aktuellen Register. Das bedeutet: Wenn Sie ein Programm mit UNTRACE durchlaufen haben, im Cache danach zurückblättern und dann das Programm ab einer Stelle nochmal durchlaufen lassen wollen, geben Sie GETCACHE ein. Dann schreibt der Debugger die Register D0-D7, A0-A7 (Supervisor und User), das Statusregister und den PC zurück.

Beim Benutzen dieses Befehls muß man natürlich aufpassen: Wenn das Programm Speicherstellen modifiziert, kann eine mehrmalige Ausführung zu anderen Ergebnissen führen. Probleme kann es selbstverständlich auch bei selbstmodifizierenden Programmen geben.

Cls Δ

Syntax : CLS

Kurzform : CL

Löscht den Programm-Bildschirm (nicht den des Debuggers). Dazu wird mit der Bios-Funktion 3 (Econout) ESC E ausgegeben (damit auch der Cursor nach oben links gesetzt wird).

Mouseon Δ

Syntax : M[OUSE]ON

Kurzform : MOU, MON

Schaltet die Maus (für das debuggte Programm; im Debugger ist sie immer an) ein. Dazu wird die entsprechende Line-A-Routine verwendet. Die Maus wird bei diesem Befehl immer eingeschaltet, unabhängig davon, wie häufig sie ausgeschaltet worden ist.

Mouseoff Δ

Syntax : M[OUSE]OFF

Kurzform : MOUSEOF, MOF

Schaltet die Maus aus. Auch hier wird Line-A benutzt. Die Maus wird immer ausgeschaltet, unabhängig davon, wie häufig sie eingeschaltet worden ist.

Getregister

Syntax : GETREGISTER [Adresse]

Kurzform : GE

Dieser Befehl holt die Register ab „Adresse“. Dabei müssen die Registerinhalte, wie bei IF auf Seite 136 beschrieben, im Speicher liegen.

Wird keine Adresse angegeben, nimmt der Debugger \$300 an.

Das ganze hat nun folgenden Sinn: Obwohl der OMIKRON.Debugger eine ganze Menge mitmacht, kann es doch vorkommen, daß man einige Unterprogramme nicht direkt debuggen kann. Wenn Sie an einer bestimmten Stelle im Programm in den Debugger zurückkehren möchten und im Debugger auch noch alle Registerinhalte haben wollen, SHIFT-SHIFT aber nicht funktioniert, können Sie eine Routine an diese Stelle ins Programm einbauen, die alle Registerinhalte rettet. Eine solche Routine finden Sie im Librarys-Ordner unter dem Namen BREAK.S. Wenn Sie dann RESET drücken, müßten Sie sich eigentlich im Debugger befinden (wenn Sie im Desktop landen, starten Sie den Debugger einfach neu). Dann können Sie sich die Registerinhalte mit GETREGISTER zurückholen.

Bssclear

Syntax : BSSCLEAR
Kurzform : BS

Löscht das Block-Storage-Segment (BSS) (füllt es mit 0). Dies ist nützlich, wenn Sie ein Programm gestartet haben und es dann vom Ausgangszustand aus noch einmal neu starten wollen.

Initregister

Syntax : INITREGISTER
Kurzform :INI

Setzt alle Daten- und Adreßregister auf 0 und PC sowie Statusregister auf die Anfangswerte.

|

Syntax : |Befehl
Kurzform : |

Der hinten angegebene Befehl wird sofort ausgeführt und die Registeranzeige entsprechend aktualisiert. Dadurch kann jedoch kein Programm gestartet werden; es wird immer nur der eine Befehl ausgeführt. „!bra \$50000“ setzt z. B. nur den PC, führt dann aber nichts weiter aus.

Sync50, Sync60

Syntax : SYNC50 bzw. SYNC60
Kurzform : SYN

Schaltet die Bildfrequenz des zu debuggenden Programms auf 50Hz bzw. 60Hz um (dies gilt natürlich nur für Farbmonitore).

6.6.6 I/O-Befehle

Lexecute ▲

Syntax : LEXECUTE ["Filename"] [, "Commandline"]
Kurzform : LE

Ein Programm wird geladen und reloziert. Letzteres ist notwendig, um es zu starten (es sei denn, es ist positionsunabhängig); man kann es danach aber nicht mehr abspeichern, da das Relozieren irreversibel ist. Wenn bereits ein anderes Programm im Speicher stand, wird es vorher gelöscht. Es werden auch alle Speicherbereiche, die das Programm angefordert haben sollte, freigegeben. Dadurch ist es möglich, beliebig viele Programme zu debuggen, ohne den Debugger jedesmal neu laden zu müssen.

Wird kein Programmname angegeben, wird das zuletzt geladene Programm noch einmal geladen. Dies ist zum Beispiel sinnvoll, wenn Sie ein Programm mit LOAD geladen, dann geändert und wieder abgespeichert haben und es jetzt testen wollen.

Wird keine Commandline angegeben, wird keine übergeben (nicht die vom letzten LEXECUTE). Als Environment-String wird der übergeben, mit dem auch der Debugger aufgerufen wurde.

Wir laden z.B. ein Programm mit „LE SCREENDUMP.PRG“. Als Antwort erscheint dann:

Start des Text-Segments	:	\$05B46E	Länge: \$00000018
Start des Data-Segments	:	\$05B586	Länge: \$0000000C
Start des BSS-Segments	:	\$05B592	Länge: \$00000022
Erste freie Adresse	:	\$05B5B4	

(Sie werden wahrscheinlich bei Ihrem Rechner andere Adressen erhalten.)

Labelbase

Syntax : LABELBASE [S|P]
 Kurzform : LA

Dieser Befehl schaltet zwischen zwei Symboltabellenformaten um. Das ist nötig geworden, als ich bemerkte, daß Turbo-C ein anderes Symboltabellenformat hat, als wir verwenden. Ich habe bis heute nicht herausbekommen können, wie das „richtige“ Format lautet.

LABELBASE P alle Symbole der Symboltabelle sind relativ zum TEXT-Segment. Nötig bei: Turbo-C, GFA-Assembler

LABELBASE S alle Symbole der Symboltabelle sind relativ zum Segment in dem sie stehen (Default-Einstellung). Bei: OMIKRONAssembler, sowie alle Linker, welche GST-Linkformat benötigen (z.B. Devpac, Metacomco Macro Assembler, GST-Assembler)

Load Δ

Syntax : LOAD [Filename] [,Adresse]
 Kurzform : LO

lädt ein File in den Speicher. Es wird dabei nicht reloziert. Wenn keine Adresse angegeben wird, wird in den Programmspeicher geladen. Wenn schon ein Programm geladen worden ist, wird es überschrieben. Start und Endadresse des File können Sie mit INFO abfragen — sie werden jedoch auch nach dem Laden ausgegeben. Die Adressen und der Name merkt sich der Debugger für SAVE automatisch.

Save ▲

Syntax : SAVE [Dateiname] [,Anfangsadresse,Endadresse]

Kurzform : S

SAVE speichert einen Speicherbereich auf Diskette (oder Festplatte, RAM-Disk) ab. Wenn Sie vorher etwas mit LOAD geladen haben (nicht mit LEXECUTE!), werden die Start- und Endadresse und der Name von LOAD übernommen, wenn nichts anderes angegeben wird. Bei LEXECUTE wurde dies absichtlich verhindert, damit man nicht versehentlich ein Programm überschreibt (Da bei LEXECUTE reloziert wird, ist ein Abspeichern nicht mehr ohne weiteres möglich). Vor dem Abspeichern kommt noch eine Sicherheitsabfrage.

Beispiel: „SAVE ,10000,20000“

Saving Test.Tos from 10000 to 20000

(Der Name stammt noch vom LOAD-Befehl)

Wird als Filenname „!“ angegeben, werden die Einstellungen des Debuggers in der Datei OM-DEBUG.INF angespeichert (siehe Kapitel 6.8 auf Seite 168).

Directory ▲

Syntax : DIRECTORY [Drive:] [Pfad] [Suchmaske]

Kurzform : DIR

DIRECTORY dient zum Anzeigen eines Inhaltsverzeichnisses von Diskette oder Festplatte. Es können das gewünschte Laufwerk, gefolgt von einem Doppelpunkt, und der Pfad (wie unter GEM üblich) angegeben werden, sonst wird der aktuelle Pfad genommen (der letzte bzw. beim ersten Mal der Pfad, von dem der Debugger geladen wurde). Außerdem wird dabei der aktuelle Pfad gesetzt. Das bedeutet: Wenn Sie „DIR C:\ASSEMBLE\“ eingegeben haben und dann „LOAD HALLO.TXT“ eingeben, versucht der Debugger, C:\ASSEMBLE\HALLO.TXT, also HALLO.TXT aus dem ASSEMBLE-Ordner des C-Laufwerks, zu laden. Das Inhaltsverzeichnis wird stets nach Namen sortiert ausgegeben.

Hinter dem Pfad kann eine Suchmaske angegeben werden. So zeigt zum Beispiel DIR D:\ASSEMBLER*.SRC alle Dateien im ASSEMBLER-Ordner, deren

Name auf „.SRC“ endet. Ordner werden immer angezeigt, auch wenn sie eine andere Endung als die im Pfad vorgegebene haben.

Vor den Namen steht dir, wenn es sich um einen Ordner handelt, bzw. lo, wenn es eine Datei ist. Wenn Sie also mit dem Cursor auf einen Eintrag gehen und RETURN drücken, wird das Unterverzeichnis angezeigt bzw. die Datei geladen. Hinter jedem Dateinamen steht, durch Semikolon abgetrennt, die Dateilänge. Wir können uns also das Tippen des Namens sparen: Wir geben einfach „DIR“ ein und sehen unter anderem lo scrndump.prg. Man kann nun das „o“ durch „e“ überschreiben und RETURN drücken.

Prn

Syntax : PRNbefehl
Kurzform : P (jedoch kein PR)

Wenn Sie PRN vor einen Befehl setzen, werden alle dazugehörigen Ausgaben nicht auf dem Bildschirm, sondern auf dem Drucker gemacht.

Beispiel: „Pd 10000,10100“
Disassembliert ein Programm von \$10000 bis \$10100 und gibt es auf dem Drucker aus.

File Δ

Syntax : FILEbefehl
Kurzform : F (jedoch ebenfalls kein FI,FIL)

Entspricht PRN, gibt jedoch nicht auf dem Drucker, sondern in eine Datei aus. Die Datei muß mit FOPEN geöffnet worden sein.

Fopen Δ

Syntax : FOPEN Filename
Kurzform : FOP

Eröffnet eine Datei, die mit FILE beschrieben werden kann.

Fclose Δ

Syntax : FCLOSE
Kurzform : FC

Schließt die mit FOPEN geöffnete Datei wieder. Dieses ist notwendig, um die im Buffer stehenden Daten zu schreiben. Die Datei wird beim Verlassen des Debuggers automatisch geschlossen.

Line

Syntax : LINE
Kurzform : LIN

LINE gibt eine Linie (79 mal das Zeichen „-“) aus. Gedacht ist dieser Befehl für eine Abgrenzung verschiedener Teile eines Ausdrucks. Wenn Sie also mit PLIST einen Programmteil auf dem Drucker gelistet haben und nun einen anderen Teil anfügen wollen, können Sie diese mit PLINE trennen.

Cr

Syntax : CR [Anzahl]
Kurzform : CR

Gibt eine (oder eine Anzahl) Leerzeile(n) aus. Dies dient der Trennung wie der Befehl LINE.

“

Syntax : "String[";]
Kurzform : "

Der String wird ausgegeben. Sinnvoll ist das wohl nur im Zusammenhang mit PRN; mit „P““ können Sie Ausdrücke beschriften. Mit dem Zeichen \,

gefolgt von zwei Hex-Ziffern, wird der entsprechende ASCII-Code ausgegeben. Das Zeichen selbst wird mit \\ ausgegeben. Mit einem „;“ am Zeilenende wird kein \ gesendet (das gute, alte BASIC...). Ansonsten wird ein „ am Zeilenende ignoriert. Damit ergibt sich auch noch eine weitere Anwendungsmöglichkeit: Es können Steuerzeichen an den Drucker gesendet werden. Haben Sie beispielsweise einen EPSON-Kompatiblen Drucker, schalten Sie ihn mit P"\OF"; auf Schmalschrift.

Erase, Kill Δ

Syntax : [ERASE|KILL] Filename
Kurzform : ER, KI

Löscht eine oder mehrere Datei auf Diskette oder Festplatte. Es wird nur die Fdelete (Gemdos 65) ausgeführt. Achtung: „KILL *.*“ löscht alle Datei (bis auf die Ordner) im aktuellen Verzeichnis!

Free Δ

Syntax : FREE [Laufwerk]
Kurzform : FR

Wenn kein Laufwerk angegeben wird, gibt der Debugger den freien Speicher im RAM, sonst auf dem entsprechenden Laufwerk, aus. Diese Funktion ruft Dfree (Gemdos 54) bzw. Malloc (Gemdos 72) auf. Bei Harddisk⁶ wird eine eigene schnelle Routine verwendet.

Mkdiritory Δ

Syntax : MKDIRECTORY Filename
Kurzform : MK

Erstellt einen neuen Ordner (Make Directory). Dazu wird Dcreate (Gemdos 57) aufgerufen.

⁶oder Ramdisk mit 16-Bit-FAT

Rmdiritory Δ

Syntax : RMDIRECTORY Filename
Kurzform : RM

Löscht einen Ordner (Remove Directory). Der Ordner muß leer sein - alle Dateien innerhalb des Ordners müssen zuvor einzeln mit KILL bzw. ERASE gelöscht werden. Diese Funktion ruft Ddelete (Gemdos 58) auf.

Name Δ

Syntax : NAME Altname,Neuname
Kurzform : N

Dieser Befehl dient zum Ändern eines Dateinamens. Es ist nicht nötig, den Pfad zweimal einzugeben: Wenn die Datei nicht im aktuellen Inhaltsverzeichnis liegt, reicht es, NAME Alter Name, Pfad\Neuer Name einzutippen. Der aktuelle Pfad wird entsprechend neu gesetzt. Dieser Befehl ruft Frename (Gemdos 86) auf.

Fattribut Δ

Syntax : FATTRIBUT [Name][,Attribut]
Kurzform : FA

Dieser Befehl setzt das Fileattribut (siehe Tabelle 6.8 auf Seite 150) für das angegebene File. Wird kein Name angegeben, wird der aktuelle Name benutzt. Wenn kein Attribut angegeben wird, gibt der Debugger das Attribut aus.

Wenn Sie ein File als schreibgeschützt und hidden kennzeichnen wollen, tippen Sie „FA NAME,1|2“ oder „FA NAME,3“. Bis zur Betriebssystemversion 1.3 ist das Ändern von Ordnern nicht möglich. *Achtung:* Wenn Sie ein File als Diskettennamen deklariert haben (Bit 3 gesetzt), können Sie es danach mit FATTRIBUT nicht mehr ändern.

Tabelle 6.8: Die Fileattribute

Bit 0	Dezimal 1	Hex 1	File schreibgeschützt (nur lesen)
Bit 1	Dezimal 2	Hex 2	File hidden (versteckt)
Bit 2	Dezimal 4	Hex 4	Systemfile
Bit 3	Dezimal 8	Hex 8	Diskettenname
Bit 4	Dezimal 16	Hex \$10	Ordner
Bit 5	Dezimal 32	Hex \$20	Archiv-Bit

Format ▲

Syntax : FORMAT [(DS|SS),Laufwerk]
 Kurzform : FO

Mit diesem Befehl können Sie eine Diskette formatieren. Wenn die Seitenzahl (DS = double sided (doppelseitig) oder SS = single sided (einseitig)) nicht eingegeben wird, nimmt der Debugger doppelseitig an. Als Laufwerksnummer können Sie 0 für Laufwerk A oder 1 für Laufwerk B angeben. Das Formatieren erfolgt mit Flopfmt (XBIOS 10) (Virgin \$E5E5, Interleave 1, 80 Tracks mit je 9 Sektoren; also ein ganz normales Format). Danach wird der Boot-Sektor mit Proboot (XBIOS 18) erstellt und mit Flopwr (XBIOS 9) geschrieben. Bevor eine Diskette formatiert wird, erfolgt (natürlich) noch eine Sicherheitsabfrage; man weiß ja nie...

Type ▲

Syntax : TYPE [Dateiname]
 Kurzform : TY

Zeigt eine Datei (in ASCII) an. Dazu wird sie in einen 8 KB großen Buffer geladen, so daß die Diskettenzugriffszeiten relativ gering sind. Die Ausgabe kann mit SPACE angehalten werden. Dann kann sie durch erneutes Drücken auf SPACE fortgesetzt oder den Druck auf eine andere Taste abgebrochen werden. Dieser Befehl kann z. B. benutzt werden, um sich eine README-Datei

anzusehen. Mit PTYPE können Sie sich natürlich diese Dateien ausdrucken lassen.

Rwabs Δ

Syntax : RWABS Rwflag, Buffer, Anzahl, Recno, Laufwerk
Kurzform : RW

Dieser Befehl dient zum direkten Schreiben oder Lesen auf/von Disketten, Festplatten und RAM-Disks. Er ruft BIOS 4 auf, welcher seinerseits eventuelle Festplatten- oder RAM-Disk-Treiber aufruft (über hdv_rw \$476). Bei diesem Befehl ist äußerste Vorsicht geboten: Man kann damit nicht nur Disketten, sondern auch Festplatten zerstören. Wenn Sie z. B. den Boot-Sektor der C-Partition löschen, haben Sie Schwierigkeiten, an irgendwelche Daten der gesamten Festplatte heranzukommen.

„Rwflag“ ist nach Bits aufgeschlüsselt:

- Bit 0: 0 = Lesen; 1 = Schreiben
- Bit 1: 1 = Diskettenwechsel ignorieren
 - Und nur für Festplatten:
- Bit 2: 0 = 3 Leseversuche, dann 6 critical error handler
 - 1 = 1 Leseversuch, kein critical error handler
- Bit 3: 0 = Logische Sektor- und Laufwerksnummern
 - 1 = Physikalische Sektor- und Laufwerksnummern (d. h. 2 für Festplatte 0, 3 für Nr. 1 usw.)

„Buffer“ gibt an, wo die Daten im Speicher zu finden sind, bzw. hingeschrieben werden sollen. „Anzahl“ ist die Anzahl der Sektoren, die übertragen werden sollen. „Recno“ gibt die Nummer des ersten Sektors und „Laufwerk“ das Laufwerk an (wer hätte das gedacht?). Bei letzterem bedeutet 0=A, 1=B usw.

Readsector Δ

Syntax : (RS|READSECTOR) [Parameter]
Parameter : [Track[, Sektor[, Seite[, Adresse[, Laufwerk]]]]]
Kurzform : RS, RE

Liest einen Sektor von der Diskette ein. Dazu wird die XBIOS 8-Funktion verwendet. Geben Sie weder Track noch Sektor an, werden die Werte der entsprechenden Variablen TRACK, SEKTOR, SIDE, SEKBUFF und DRIVE genommen - Voreinstellung ist Track 0, Sektor 1, Seite 0, Laufwerk A (0); also der Boot-Sektor von Laufwerk A. Wenn Sie andere Werte angeben, werden die genannten Variablen geändert, so daß Sie den Sektor wieder abspeichern können, ohne sich die Werte merken zu müssen. Die aktuelle Adresse wird danach auf SEKBUFF gesetzt, Sie können den Sektor also mit MEMORY ohne Adressangabe ansehen.

Writesector ▲

Syntax : (WS|WRITESECTOR) [Parameter]
Parameter : [Track[,Sektor[,Seite[,Adresse[,Laufwerk]]]]]
Kurzform : W

WRITESEKTOR ist das Gegenstück zu READSEKTOR - er schreibt einen Sektor mit XBIOS 9 auf Diskette. Vor dem Schreiben erfolgt noch eine Sicherheitsabfrage. Alles andere ist genauso wie bei READSEKTOR.

Readtrack

Syntax : (READTRACK|RTRACK) [Parameter]
Parameter : [Track[,Seite[,Adresse[,Laufwerk]]]]
Kurzform : READT, RT

READTRACK liest eine komplette Spur von einer Diskette ein. Dabei werden auch alle Daten zwischen den Sektoren eingelesen. Mit MEMORY können Sie sich die Spur danach ansehen. Ein erneutes Schreiben der Spur ist natürlich nicht möglich, da der FDC dies nicht zuläßt.

Für „Seite“ sind 0 oder 1 zulässig; „Laufwerk“ 0 = A und 1 = B. Von Festplatte ist READTRACK natürlich nicht möglich. „Adresse“ ist die Startadresse des Buffers, in den die Spurdaten gelegt werden sollen.

6.6.7 Listende Befehle mit Scrolling

Alle in diesem Abschnitt angegebenden Befehle erlauben den gleichen Syntax bei den Parametern, der deswegen an dieser Stelle erklärt wird, und im folgenden nur noch mit [Parameter] bezeichnet wird. Es gilt also:
[Parameter] = [Von][[,]#[Zeilen]|Bis|[,][Bytes[]]]

Es sind also alle Parameter wahlfrei, d. h. man braucht keine Parameter angeben. Der Debugger nimmt dann vorgegebene interne Werte.

Wenn der Ausdruck „Von“ fehlt, wird ab der aktuellen Adresse z. B. disassembliert. Die aktuelle Adresse ist die Zahl am Zeilenanfang, bzw. wenn diese fehlt, die zuletzt benutzte Adresse.

Als Endadresse gilt der Ausdruck „Bis“, der jedoch nicht angegeben werden muß. Wird statt „Bis“ ein „#“ angegeben wird genau eine Zeile ausgegeben. Ein dem „#“ folgender Term, gilt als Zeilenanzahl. Es können somit z. B. genau 8 Zeilen ausgegeben werden. Es werden jedoch maximal 99 Zeilen ausgegeben. Fehlt die Endangabe gänzlich, werden (normalerweise) 16 Zeilen ausgegeben. Die Anzahl läßt sich jedoch einstellen, indem man die Variable „Lines“ entsprechend ändert. Die letzte Möglichkeit ist die Angabe der Byteanzahl in eckigen Klammern. Sie kann genauso, wie die Zeilenanzahl angegeben werden. Die „]“ ist optional, d. h. man kann sie auch weglassen.

Beispiel: „d text #5“

Disassembliert 5 Zeilen ab Anfang des geladenen Programms.

Beispiel: „m data[30]“

Ein Memorydump des DATA-Segments (48 Bytes lang).

Disassembly

Syntax : DISASSEMBLE [Parameter]
Kurzform : D

Durch DISASSEMBLE wird ein Speicherbereich disassembliert. Rechts sehen Sie den Opcode, links die entsprechenden Hex-Werte. Der Hex-Teil kann geändert werden. Da der disassemblierte Teil durch Semikola abgegrenzt ist, werden Änderungen in ihm ignoriert. Beim Disassemblieren und Listen ist

Scrolling möglich. Eine eventuell vorhandene Symboltabelle wird ignoriert, d. h., Sprünge werden stets mit Adressen ausgegeben.

/

Syntax : /[Datum]{,Datum}
Kurzform : /

"/" gehört zu DISASSEMBLE. Es können nur die Hex-Daten geändert werden. (Übernahme mit RETURN nicht vergessen, die Zeile wird dann neu disassembliert.)

List

Syntax : LIST [Parameter]
Kurzform : L

Entspricht DISASSEMBLE, es wird jedoch kein Hex-Teil ausgegeben. Dafür werden Symbole (wenn vorhanden) angezeigt. Die Befehle können geändert werden.

!

Syntax : !Opcode
Kurzform : !

Dies ist der zu LIST gehörende Befehl. Vor jeder Zeile wird ein "!" ausgegeben; dann können Sie den Opcode ändern, indem Sie ihn überschreiben und mit RETURN den neuen Übernehmen. Es wird dann in der folgenden Zeile automatisch wieder ein "!" vorgegeben. Dies ermöglicht die einfache Eingabe von wenigen Befehlen direkt in den Speicher.

Symboltable

Syntax : SYMBOLTABLE [Parameter]

Kurzform : SY

Gibt die Symboletabellen, wenn vorhanden, aus. Dann kann mit den Cursor-tasten oder der Maus durch die Tabelle gescrollt werden.

Wenn Sie eine Start- und eine Endadresse angeben, werden nur die Symbole angezeigt, deren Wert zwischen diesen beiden Adressen liegt. Es kann auch nur die Start- oder Endadresse vorgegeben werden; es wird dann ab Anfang bzw. bis zum Ende gesucht. Sie können als Adressen auch Labels (mit „“ markiert) verwendet werden.

Hinter den Adressen werden noch weitere Informationen angegeben. Die Zeichen bedeuten folgendes:

1. + Defined

Das Label wurde definiert. Das ist bei der Grundversion immer der Fall, nur bei der Linkerunterstützung wird es möglich sein, Labels vorübergehend nicht zu definieren.

2. K Equated

(oder auf gut deutsch: Konstante): Dieses Label wurde mit einem EQU-Befehl definiert. Der OMIKRON.Assembler nimmt solche Symbole gar nicht erst in die Symboletabellen auf, da Sie nur irritierten.

3. G Global

Dieses Label ist global definiert. Das kann beim OMIKRON.Assembler intern dadurch erreicht werden, das Sie nach der Labeldefinition nicht einen, sondern zwei Doppelpunkte schreiben. Das ist aber nur für Linkerunterstützung sinnvoll.

4. R Registerlist

Registerlisten werden vom OMIKRON.Assembler auch nicht abgespeichert.

5. X External

Diese Variable wird beim Linken definiert (wird jetzt also auch noch nicht benutzt).

6. T Text-relativ

Diese Variable liegt im Text-Segment.

7. D Data-relativ

Diese Variable liegt im Data-Segment.

8. B BSS-relativ

Diese Variable liegt im Block-Storage-Segment (BSS)

9. L Long

Diese Variable ist länger als 8 Zeichen und somit im GST-Format abgelegt

Sie werden sich jetzt vielleicht fragen: Was soll der Blödsinn? Erst erklärt der mir die ganzen Buchstaben, und dann, daß sie doch nicht unterstützt werden! Der Debugger unterstützt diese Status aber alle; Sie können das nutzen, wenn Sie Programme laden, die mit anderen Assemblern oder der Makroversion erstellt worden sind. Die Symboltabelle entspricht dem DR-Format (siehe Kapitel D.4 auf Seite 197).

Dump, Memory

Syntax : [DUMP|MEMORY] [. [B|W|L]] [Parameter]

Kurzform : DU, M

Dieser Befehl zeigt den Speicherinhalt zwischen Von und Bis an. Links wird hexadezimal, rechts in ASCII angezeigt. Sie können die Inhalte ändern und mit RETURN übernehmen. Die optionale Angabe „.B“, „.W“ oder „.L“ gibt an, ob die Werte byte-, word- oder langwortweise ausgegeben werden. Bei „.W“ und „.L“ sind natürlich nur gerade Adressen möglich.

Wenn keine Endadresse angegeben wird, wird soviel angezeigt, wie auf den Bildschirm paßt. Sie können dann mit 8 und 2 durch den Speicher scrollen".

Beispiel: m ^a4

Zeigt den Speicher ab dem Inhalt von Register A4. Beispiel: Du ^a4

Zeigt den Speicher ab der Adresse, die in A4 steht, an (indirekt).

.W und .L

Syntax : .(W|L)Term{,Term}

Kurzform : .

Mit diesem Befehl können Sie einen (oder mehrere) Term(e) in den Speicher schreiben. Dieser Befehl gehört eigentlich zu DUMP und MEMORY; bei DUMP wird am Anfang jeder Zeile .(WL)— geschrieben und dann der Speicherinhalt ausgegeben. Wenn Sie diesen überschreiben und RETURN drücken, wird die Zeile als .-Befehl interpretiert. Sie können aber auch . verwenden, ohne vorher DUMP benutzt zu haben. Nachdem der Term in den Speicher geschrieben wurde, wird der Speicherinhalt wie bei DUMP ausgegeben. Die Terme dürfen keine Opcodes (!) oder ASCII-Strings ("") enthalten.

Beispiel: „.w 10000,20+d0,7a8“

,

Syntax : ,Term{,Term}

Kurzform : ,

Entspricht ., bezieht sich aber auf Bytes. Opcodes und ASCII-Strings sind hier allerdings erlaubt.

]

Syntax :]Term

Kurzform :]

Ist fast das gleiche wie oben, es wird aber kein neuer Speicherinhalt auf dem Bildschirm ausgegeben. Sinnvoll ist dieser Befehl also, wenn auf dem Bildschirm Daten stehen, die Sie noch brauchen, aber irgendeine Speicherstelle ändern müssen. Für „Term“ sind die gleichen Angaben wie bei FILL möglich, also auch ASCII-Strings (") und Opcodes (!).

Ascii

Syntax : ASCII [Parameter]
Kurzform : A

Mit ASCII wird der Speicherinhalt in ASCII ausgegeben. Er kann wie bei DUMP bzw. MEMORY geändert werden.

)

Syntax :)ASCII-Daten
Kurzform :)

Vor jeder ASCII-Dump-Zeile steht ein „)“. Dann kann der ASCII-Teil entsprechend geändert werden und mit RETURN übernommen werden.

6.6.8 Allgemeine Befehle

Find

Syntax : FIND [Von,Bis],Term
Kurzform : F

FIND sucht den angegebenen Bereich bzw. den Programmreich (TEXT und DATA, aber nicht BSS) durch und zeigt alle Adressen an, bei denen „Term“ vorkommt. „Term“ kann ein belieber Ausdruck sein, dessen Wert auch länger als ein Langwort sein kann (wichtig für ASCII-Suche). Es sind auch Opcodes (!) möglich. Ist „Term“ länger als ein Byte, wird immer die Adresse des ersten übereinstimmenden Bytes angezeigt.

Bei Zahlen kann die Länge durch Anhängen von „.B“ (Byte), „.W“ (Word) „.A“ (Adresse,3 Byte) oder „.L“ (langwort) angegeben werden. Wenn nichts angegeben wird, nimmt der Debugger die kürzestmögliche Länge an. Ist der Wert länger als Ihre Wunschlänge, werden entsprechend viele niedrigwertige Bytes genommen (näheres siehe FILL (siehe Kapitel 6.6.8 auf Seite 161)).

Die Ausgabe kann mit SPACE angehalten werden. Dann können Sie mit SPACE die Ausgabe fortsetzen. Mit einer anderen Taste kann die Ausgabe abgebrochen werden. Eine abgebrochene Suche kann mit CONTINUE wieder aufgenommen werden.

Hunt

Syntax : HUNT [Von,Bis],Term
Kurzform : H

HUNT entspricht FIND, es wird jedoch nur an geraden Adressen gesucht. Dieses ist z. B. bei Opcodes nützlich, da diese sowieso nur an geraden Adressen liegen dürfen.

Ascfind

Syntax : ASCFIND [Von,Bis],String
Kurzform : ASCF

Mit diesem Befehl können nicht nur „komplette Befehle“ wie mit FIND gesucht werden, sondern auch Teile davon. Das bedeutet: Jede durchsuchte Zeile wird disassembliert und "String" damit verglichen. Dabei sind als Allquantor „*“ und als Existenzquantor „?“ zugelassen.

So können Sie z. B. alle Befehle suchen, die etwas in D7 schreiben: „ASCFIND ,*,D7“ (Das erste Komma dient nur als Kennzeichnung des Strings, die Adressen wurden weggelassen.).

Oder Sie suchen alle Befehle, die indirekt etwas aus einem Adressregister holen: „ASCFIND ,*(A?),*“.

Das Suchen mit ASCFIND dauert natürlich wesentlich länger als mit FIND oder HUNT, weil jede Zeile erst disassembliert werden muß.

Continue

Syntax : CONTINUE
Kurzform : CON

CONTINUE setzt die Ausführung von FIND, HUNT und ASCFIND fort, wenn diese nach Space abgebrochen wurde. Dies ist nicht mehr möglich, wenn zwischendurch !Befehl (direkt ausführen) oder FILL ausgeführt wurde. Wenn die Endadresse erreicht wurde, ist CONTINUE natürlich auch nicht mehr möglich.

Info

Syntax : INFO
Kurzform : I

Dieser Befehl gibt den Start von Text-, Data- und Block-Storage-Segment (BSS) entsprechend den Einträgen der Basepage an. Dieser Befehl ist auch über eine Funktionstaste erreichbar; dann werden die Informationen aber in einer Dialogbox ausgegeben.

?

Syntax : ?Term{,Term}
Kurzform : ?

Der Befehl (angelehnt an den PRINT-Befehl in BASIC) rechnet den Term aus und zeigt das Ergebnis in hex, dezimal, binär und ASCII an.

Sie können diesen Befehl also benutzen, um in andere Zahlensysteme umzurechnen, Zahlen mit Registern zu verrechnen o. ä.

Beispiel: „?15*.10+^a5“
\$72534 .468276 %1110010010100110100 " %4"

Move, Copy

Syntax : [MOVE|COPY] Von,Bis,Nach
Kurzform : MO, COP

Kopiert einen Speicherbereich. Das Kopieren geschieht byteweise, daher sind auch ungerade Adressen möglich. Es wird dabei „intelligent“ verschoben, d. h. auch wenn der Quellbereich in den Zielbereich hereinragt, wird er trotzdem richtig verschoben. Am besten sich vergessen diesem Satz schnell wieder und merken sich nur, daß der Debugger stets richtig kopiert.

Fill

Syntax : FILL Von,Bis,Term{,Term}
Kurzform : FIL

Füllt einen Speicherbereich mit einem Wert. Auch ASCII (") und Opcode (!) sind möglich. Bei Zahlen kann die Länge durch Anhängen von „.B“ (Byte), „.W“ (Word) „.A“ (Adresse, 3 Byte) oder „.L“ (Langwort) angegeben werden. Wenn nichts angegeben wird, nimmt der Debugger die kürzestmögliche Länge an. Ist der Wert länger als Ihre Wunschlänge, werden entsprechend viele niedrigwertige Bytes genommen.

Beispiele:

Eingabe:	Wird interpretiert als
100	\$0100
50	\$50
.128.W	\$0080
123456.w	\$3456
RTS	\$4E75
100,!NOP,5.A	\$01004E71000005

Die Werte werden hintereinander in einen 80 Byte großen Buffer geschrieben, der dann in den Speicher geschrieben wird.

Clr

Syntax : CLR [Von[,Bis]]
Kurzform : CLR

Löscht einen Speicherbereich (füllt ihn mit Nullbytes). Es entspricht also einem speziellen FILL-Befehl, CLR ist aber schneller. Werden keine Adressen angegeben, nimmt der Debugger die erste und letzte freie Adresse, also den Bereich, in den normalerweise Programme geladen werden. Die Benutzung von CLR ohne Parameter dürfte wohl der Normalfall sein. Vor dem Löschkvorgang erfolgt noch eine Sicherheitsabfrage, da es ab und zu mal vorkam, daß ich mich vertippt habe und daraufhin der ganze Speicher gelöscht wurde.

Compare

Syntax : COMPARE Von,Bis,Adresse
Kurzform : CO

Dieser Befehl dient zum Vergleichen von zwei Speicherbereichen miteinander. Dabei wird „Von“ bis „Bis“ und „Adresse“ bis „Adresse+Bis-Von“ miteinander. Die ungleichen Stellen werden ausgegeben. Die Ausgabe der ungleichen Stellen, kann, wie bei FIND auch, mit SPACE gestoppt und dann mit ESC abgebrochen werden. SPACE für bei gestoppter Ausgabe auch wieder fort. Der Befehl CONTINUE in *nicht* möglich.

Checksumme

Syntax : CHECKSUMME [Adresse][,Pr"ufsumme][,Anzahl]
Kurzform : CH

Dieser Befehl berechnet eine Prüfsumme über die 512 Byte ab „Adresse“ bzw. ab BUFFER (somit über den Sektor, der im Diskettenbuffer steht). Er gibt den Wert aus, den Sie zu einem Word addieren müssen, damit die Prüfsumme \$1234 bzw. „Prüfsumme“ ergibt. Sinn des ganzen ist es, Boot-Sektoren ausführbar zu machen.

Das ganze nochmal langsam: Damit ein Boot-Sektor einer Diskette ausführbar ist, d. h. damit beim Einschalten des Rechners ein Programm im Boot-Sektor gestartet wird, das dann ein Betriebssystem o. ä. lädt, muß die Summe aller Worte im Boot-Sektor \$1234 ergeben (Überläufe werden nicht beachtet).

Um die Prüfsumme zu berechnen, dient dieser Befehl. Das letzte Word im Boot-Sektor ist gewöhnlich Null, sonst löscht man es vorher. Dann ruft man CHECKSUMME auf; der Wert, der zurückgegeben wird, wird in ein Null-Wort eingetragen. Dann kann der Sektor mit WRITESEKTOR geschrieben werden. Mit „Anzahl“ kann die Länge des Buffers eingestellt werden. Es werden „Anzahl“+1 Worte addiert.

Mit „Prüfsumme“ kann man eine andere Prüfsumme als \$1234 angeben. Sie können damit zum Beispiel die Prüfsumme über ein Reset-festes Programm im Bereich von \$600 bis \$700 - sie muß \$5678 sein, damit das Programm nach einem Reset ausgeführt wird - mit „CHECKSUM 600,5678,100“ berechnen.

Resident Δ

Syntax : RESIDENT
Kurzform : RESI

Der Debugger wird nach einer Sicherheitsabfrage verlassen, aber resident im Speicher gehalten (siehe Kapitel 6.12 auf Seite 174).

Exit, Quit, System Δ

Syntax : EXIT|QUIT|SYSTEM
Kurzform : EX, QUI, SYS

Der Debugger wird verlassen und zum Assembler beziehungsweise Desktop zurückgekehrt. Er setzt, soweit möglich, alle kritischen Speicherstellen in den Zustand zurück, in dem er sie vorgefunden hat (siehe Kapitel 6.6.8 auf Seite 165).

Vor dem Verlassen des Debuggers erfolgt noch eine Sicherheitsabfrage, die durch „j“ zu beantworten ist (oder „n“ wenn Sie ihn nicht verlassen wollen).

@

Syntax : @Befehl
Kurzform : @

Das ist sogenannte Batch-Befehl, er ermöglicht es beim Einsprung in den (residenten) Debugger automatisch einen Befehl ausführen zu lassen. Der hinter @ angegebene Befehl wird *nicht* ausgeführt, er wird nur intern gespeichert und beim nächsten Einsprung in den Debugger ausgeführt.

Beispiel: „@info“

Beim Einsprung in den Debugger wird automatisch der Befehl INFO ausgeführt. Man kann aber auch z. B. die Auflösung beim Einsprung in den Debugger umschalten (siehe Kapitel 6.9 auf Seite 170).

Wenn der Batch-Befehl nicht mehr ausgeführt werden soll, kann durch Eingabe eines einzelnen „@“ ohne Parameter der Batch-Befehl gelöscht werden. Ein Batch-Befehl kann mit SAVE ! gespeichert werden.

Set, Let

Syntax : [SET|LET|^] (ALL|Variable)=Term
 Kurzform : SE, LET, ^

Mit SET (bzw. LET) können Sie ein Register auf einen bestimmten Wert setzen oder alle Daten- und Adreßregister (SET ALL) auf einmal auf den gleichen Wert.

Als Register sind zugelassen: D0-D7, A0-A7, SP, SSP, USP, SR, PC

Beispiele:

- „SET ALL=0“, löscht D0-A6
- „SE a3=a4+pc“, setzt Register A3 auf PC+\$A4 (nicht Adreßregister A4!); das hieße SE a3=^a4+pc)
- „ sp=^a4“, setzt den aktuellen Stackpointer (abhängig vom Supervisor-Bit)

Key

Syntax : KEY [([C|A]FNumber=Text)]
 Kurzform : K

Mit KEY können die Funktionstasten belegt werden. KEY ohne Parameter zeigt die Funktionstastenbelegungen an (nicht die mit SHIFT und ohne solche Tasten erreichbaren, sondern nur die freidefinierbaren mit CONTROL und ALTERNATE).

Mit KEY C oder A FNummer=Text kann man eine Funktionstaste belegen. Die Nummer muß zwischen 1 und 20 liegen: 1 - 10 = Funktionstaste ohne; 11-20 = Funktionstaste mit SHIFT. Vor dem F können Sie angeben, ob der Text bei der Funktionstaste mit CONTROL oder ALTERNATE erscheint; wenn die Eingabe fehlt, nimmt der Debugger CONTROL an. Der Text darf nicht in Anführungszeichen stehen (bzw. dann werden diese mit zum Text gerechnet). Wenn Sie SHIFT-RETURN drücken, erscheint R , was für ein RETURN steht, wenn die Zeile ausgegeben wird.

Beispiel: „KEY AF5=FORMAT SS,B R “ (SHIFT-RETURN)

Wenn Sie dann Alternate F5 drücken, wird die Diskette in Laufwerk B einseitig formatiert.

Reset all Δ

Syntax : RESET [A[LL]]|[V[EKTOR]]
Kurzform : RES

Nach einer Sicherheitsabfrage werden die Systemvektoren (\$8 - \$516) auf die Werte des Debuggers zurückgesetzt (es werden also weiterhin Traps abgefangen usw.). Dieses ist sinnvoll, wenn ein Programm unkontrolliert im Speicher herumschreibt - manchmal kann dann mit RESET noch einiges gerettet werden. Mit RESET ALL werden alle vom Debugger beim Programmstart geretteten Werte zurückgeschrieben. RESET ALL bewirkt das gleiche, als ob Sie den Debugger verließen und gleich wieder neu starteten. Das bedeutet, wenn Ihr Programm abgestürzt ist, und nicht mehr geht, können Sie mit RESET ALL wahrscheinlich den Rechner wiederbeleben. RESET VEKTOR ist die „kleine“ Version von RESET, dabei werden nur die Exceptionvektoren auf den Debugger umgelenkt, das kann z. B. nötig sein, wenn ihr Programm den „Illegal Instruktion“-Vektor „verbogen“ hat, Sie aber Breakpoint benutzen wollen, welche ja diesen Vektor benötigen. Der „normale“ Anwender wird diesen Befehl wohl nie brauchen.

Switch

Syntax : SWITCH
Kurzform : SW

Damit wird der Debugger zwischen der mittleren und hohen Auflösung hin- und hergeschaltet. Die Bildschirmauflösung des debuggten Programms bleibt dabei unverändert. Wenn Sie einen Monitorumschalter verwenden, müssen Sie nur noch auf den anderen Monitor umschalten - bei einem automatischen Umschalter⁷ wird das Bild automatisch ungeschaltet. Bei jedem Programmstart wird automatisch auf den anderen Monitor zurückgeschaltet. Dadurch können Sie den Debugger auf der (augenfreundlicheren) hohen Auflösung laufen lassen und ein niedrig auflösendes Programm testen.

Scrolldown

Syntax : SCROLLDOWN [Fangradius]
Kurzform : SC

Wenn Sie das Mausscrolling wieder einschalten wollen, können Sie dies mit dem Befehl SCROLLDOWN. „Fangradius“ bedeutet die Breite des Streifens in Bildschirmpunkten, in den die Maus bewegt werden muß, damit gescrollt wird. Voreinstellung ist drei, d. h. der Bildschirm wird gescrollt, wenn Sie die Maus in eine der obersten bzw. untersten drei Zeilen bewegen (und kurz die rechte Maustaste drücken).

Scrolloff

Syntax : SCROLLOFF
Kurzform : SCROLLOF

Wenn Sie mit der Maus nicht scrollen wollen, können Sie dies mit SCROLLOFF abschalten. Dann ist Scrollen natürlich mit den Cursortasten weiterhin möglich.

⁷wie z. B. dem der Firma Hard & Soft A. Herberg, Bahnhofstr. 289, 4620 Castrop-Rauxel

Cursor

Syntax : CURSOR Nummer
Kurzform : CU

Mit diesem Befehl können Sie die Cursorform umschalten. Die Nummer bedeutet:

Nummer	Cursorform
0 =	█
1 =	,
2 =	—
3 =	□

Diese Nummern bzw. Cursorformen sind die gleichen wie beim Assembler.

Help

Syntax : HELP
Kurzform : HE

Gibt alle Befehle des Debuggers unformatiert aus...

6.7 Internas zum Debugger

Intern werden folgende Vektoren belegt:

- etv_term
- etv_critic (bei Gemdos-Fehlern z. B. Disk schreibgeschützt)
- swv_vec (auf RTS)
- Trap # 1 (für Gemdos-Patches bei Term und intern für LEXEC)
- 200Hz-Timer für Autorepeat (nur intern)

- Keyboard-IRQ (nur intern / Abfrage von Shift-Shift)
- VBL für die Maus
- Trap # 3 zeigt auf RTR (Supervisormode an)
- Ring Indicator (für External Break)

Abhängigkeit vom Betriebssystem Aus der Adresse \$4F2 (Sysbase) wird die Betriebssystemsversion ermittelt. Beim Blitter-TOS (oder neueren Versionen) werden Kbshift und act.pd aus dem Systemheader geholt, sonst aus den Speicherstellen \$E1B und \$602C. Die Fontadressen werden mit LINE-A ermittelt. Beim CLS-Befehl werden BIOS-Routinen genutzt (Zurücksetzen des Cursors).

Bei Disketten-/Festplattenzugriffen: XBIOS 8/9 beim Lesen/Schreiben eines Sektors, XBIOS 10/18/9 beim Formatieren.

Die Adresse der Media-Change-Variable wird durch eine eigene VBL-Routine ermittelt: Es wird Mediach aufgerufen (BIOS 9). Wenn ein Wert ungleich 2 zurückgegeben wird (Diskette wurde nicht mit Sicherheit gewechselt), wird Getpbp (BIOS 7) aufgerufen. Dann wird Mediach erneut gestartet, bis sie den Wert 2 liefert - A0 zeigt dann auf die Media-Change-Variable. Dieser Trick von Artur Södler funktioniert mit allen TOS-Versionen (auch mit dem TOS 1.4).

Bei allen Dateioperationen wird (natürlich) das GEMDOS benutzt.

6.8 Die Datei „OM-DEBUG.INF“

Mit dem Befehl SAVE ! speichert der Debugger alle Einstellungen in der Datei „OM-DEBUG.INF“ ab. Wenn Sie hinter dem Ausrufungszeichen ein „H“ eingeben, wird die Datei als „hidden“ abgespeichert, d.h., sie ist im Inhaltsverzeichnis nicht sichtbar. Wenn Sie ein „R“ angeben, ist der Debugger nach dem nächsten Laden resident (siehe Kapitel 6.12 auf Seite 174). Findet der OMIKRON.Debugger, nachdem er geladen wurde, diese Datei, wird sie wieder geladen und alle Einstellungen entsprechend gesetzt. Sie erkennen

diesen Vorgang an der Zeile „Installation wurde geladen“ unmittelbar nach dem Laden.

In der Info-Datei werden folgende Einstellungen gespeichert:

- Autoinsert (kann mit SHIFT-INSERT oder SHIFT-F5 umgeschaltet werden)
- Cursor Form (kann mit CURSOR umgeschaltet werden)
- Zahlenbasis des Disassemblers (Variable DISBASE)
- LINES (die Anzahl der Zeilen, die bei MEMORY, LIST o. ä. ausgegeben werden)
- Anzahl der Werte, die bei DUMP/MEMORY pro Zeile ausgegeben werden
- Ob beim Tracen mit den Funktionstasten disassembliert oder gelistet wird
- Ob mit der Maus gescrollt werden kann sowie der Fangradius
- Ob Groß- oder Kleinschrift bei Hexzahlen verwendet wird
- Farben des Debuggers
- Tastaturklick
- Der Inhalt des CONTROL-M-Buffers
- Die Funktionstastenbelegung
- Zeichenfilter für ASCII-Anzeigen (Wenn der Debugger das ASCII-Zeichen *x* ausgeben will, gibt er das *x-te* Zeichen des Filters aus. Wenn Sie also ein Zeichen herausfiltern wollen, ersetzen Sie es durch „CHR(FA)“ (siehe auch bei der Variablen CONVERT) auf Seite 127).

6.9 Ein paar Tips...

- Noch ein Tip für Programmierer, welche Farbprogramme (ohne GEM) entwickeln wollen (z. B. Spiele):
 1. einen elektronischen Monitorumschalter besorgen und s/w einschalten.
 2. im Debugger „~rez=0“ eingeben und Einstellungen sichern (siehe Kapitel 6.8 auf Seite 168); dann den Debugger resident im Speicher halten (siehe Kapitel 6.12 auf Seite 174).
 3. Assembler laden und im Spezialmenü „starten & zurück“ anwählen.
 4. Wenn nun ALT+A gedrückt wird, wird das Programm assembled, in den Debugger gesprungen, auf Farbe umgeschaltet, das Programm gestartet und bei fehlerfreier Beendigung wieder in den Assembler zurückgesprungen.
 5. Programmentwicklung für Farbe auf Monochrom, was für eine Freude für die Augen... Wer allerdings mit F1 im Debugger tracen will, sollte mit „~rez=2“ die Auflösung des Programms wieder auf Monochrom stellen, das ist besser für Augen und Monitor. Evtl. sollte man sich die Auflösungsumschaltung auf Funktionstasten legen (siehe Kapitel 6.6.8 auf Seite 164).
- Aus unerklärlichen Gründen ist es zu empfehlen zum Debuggen das Kontrollfeld zu entfernen (d. h. umbenennen). Sonst kann es passieren, daß der Fileselector eines zu debuggenden Programms abstürzt! Warum? Keine Ahnung, ich habe das auch nur von einem Freund, dessen Programm im Debugger mit Kontrollfeld abstürzte; jedoch ohne Kontrollfeld bzw. mit Kontrollfeld und ohne Debugger einwandfrei funktioniert.

6.10 Externe Abbruchmöglichkeiten

Die sicherste Methode ein Programm abzubrechen, ist es den Rechner für mindestens 10 Minuten auszuschalten. Wer jedoch ein Programm debug-

gen möchte, legt wahrscheinlich Wert darauf, daß er stets in den Debugger zurückspringen kann. Dazu gibt es zwei Möglichkeiten:

Der Debugger ist resetfest. Das bedeutet, der Debugger wird nicht dadurch verlassen, daß Sie RESET drücken. Wenn ein Programm sich aufhängt, so daß die Tastaturabfrage und somit SHIFT-SHIFT nicht mehr funktioniert, können Sie durch Reset in den Debugger zurückkehren. Es kann Ihnen auch passieren, daß Sie zwar das Bild des Debuggers sehen, aber nichts über die Tastatur eingeben können. Nach einem Reset funktioniert die Tastatur meist wieder fehlerfrei. Bei einem Reset werden vom Betriebssystem, bevor es die Kontrolle an den Debugger übergibt, die Register D0, A0, A5, A6, SR, SSP und PC verändert. Die Registerinhalte, die Ihnen der Debugger anzeigt, sind also nach einem Reset nicht alle richtig!

Einen etwas sanfteren Abbruch können Sie über den „Ring Indicator“ auslösen. Dazu basteln Sie sich einen Taster, der am seriellen Port die Pins 22 und 20 miteinander verbinden kann. Den Namen „Ring Indicator“ hat dieser Pin daher, daß ein Modem hierüber einen Interrupt auslöst, wenn angerufen wird. Achten Sie also darauf, daß Sie beim Debuggen kein eingeschaltetes Modem an den ST angeschlossen haben: Wenn Sie angerufen werden, wird das Programm unterbrochen. Bei diesem Abbruch werden keine Registerinhalte zerstört, er kann aber vom debuggten Programm durch Ändern des Vektors an Adresse \$138 bzw. Verbieten des Interrupts im MfP verhindert werden.

Beide Möglichkeiten sind mit Vorsicht einzusetzen. Es ist nämlich nicht sichergestellt, daß Sie nach einem externen Abbruch das Programm noch fehlerfrei starten können. Das AES hat Schwierigkeiten, wenn es bei der Arbeit abgewürgt wird und dann ohne Neuinitialisierung weitermachen soll. Daher sollten Sie Resets nur auslösen, wenn sich das Programm nicht im Betriebssystem befindet (leicht gesagt, ich weiß).

6.11 Meldungen des Debuggers

Der Debugger kann folgende (Fehler)meldungen ausgeben:

- ?Befehl unbekannt
Ein Tippfehler, den Befehl gibt es nicht.

- ?Überlauf
z. B. $10000 * 10000$ wird größer als ein Langwort
- ?Syntax-Fehler
Ein Tippfehler in den Parametern eines Befehls.
- ?Wert nicht erlaubt
z. B. Ein Spur 100 bei READSEKTOR
- ?Klammer fehlt
z. B. Die Klammerebenen in einer Formeln stimmen nicht.
- ?Welcher Drucker
Der Drucker ist off-line oder nicht angeschlossen.
- ?Datei nicht mit FOPEN geöffnet
Bevor man die Ausgabe in eine Datei umlenken kann, muß diese geöffnet worden sein.
- ?Datei wurde bereits geöffnet
Man kann nur in eine Datei zur Zeit die Ausgabe umlenken.
- ?Es ist kein Programm geladen
z. B. HUNT ohne Parameter funktioniert nur mit geladenem Programm.
- ?Label existiert nicht
Das Symbol wurde in der Symbolebene nicht gefunden.
- ?Keine Symbolebene
z. B. SYM ohne Symbolebene
- ?Interner Fehler (Bitte Eingabe notieren!)
Nur in convert_formel möglich (Formel \Rightarrow Assemblerprg).
- ?Nicht erlaubt
LEXECUTE im vom Assembler aus aufgerufenen Debugger.
- ?Illegaler Breakpoint
z. B. „B0=100“ (Adresse ist zu klein)

- ?String zu lang
Es sind max.31 Zeichen bei KEY erlaubt.
- ?Disk full
wenn bei Fwrite nicht alle Bytes geschrieben wurden.
- FAT may be defect
wenn bei Fread nicht alle Bytes gelesen wurden.
- ?Nicht möglich
Wenn FIND, HUNT bzw. ASCFIND das Ende schon erreicht haben und trotzdem CONT eingegeben wurde.
- 2, Busfehler
- 3, Adreßfehler
- 4, Illegaler Befehl
- 5, Division durch Null
- 6, Befehl CHK
- 7, Befehl TRAPV
- 8, Privilegverletzung
- 9, Trace
- 14, Abruch durch Shift-Shift
- 15, Nicht initialisierter Interrupt
- 16,104, Ende durch RTS
- 17, Alternate + HELP gedrückt
- 24, Nicht maskierbarer Interrupt (NMI)
- 30,31, Illegaler Interrupt
- 32,35,36,37,38,39,40,41,42,43,44,47, Undefinierter Trap #

- 33,34,45,46, Abbruch bei Trap #
- 33,34, Programmende bei Trap #
- 52, Illegale Parameter bei Trap #
- 78, Externer Abruch (Ring-Indicator)
- 100, Abbruch bei Breakpoint #
- 101, Stop-Breakpoint #
- 102, Permanent-Breakpoint #
- 103, User-Breakpoint #
- -1, Unbekannte Exception #
- Illegaler Speicherbereich!
wenn ein Busfehler im Debugger auftritt, also z. B. wenn Sie mit MEMORY o. ä. Speicher ansehen wollen, der nicht existiert und die Variable MEMCHECK gesetzt haben.

Bei Bus-, Adreßfehlern wird ein Funktionscode mit ausgegeben. Dieser hat folgende Bedeutung:

- Bit 2: 1 = Supervisormodus, 0 = Usermodus
- Bit 1: 1 = Programm, 0 = Daten

Dazu wird noch ein „B“ ausgegeben, wenn ein einzelner Befehl, oder ein „T“ ausgegeben, wenn ein Trap ausgeführt wurde.

6.12 Der „Resident“-Befehl

Man kann den OMIKRON.Debugger auch resident im Speicher halten, wie z. B. den Templemon. Dazu geben Sie nach dem Laden den Befehl RESIDENT ein. Nach einer Sicherheitsabfrage wird der Debugger verlassen. Sie befinden sich dann wieder auf dem Desktop, als wenn Sie ihn mit QUIT verlassen

hätten. Wenn nun aber ein Fehler auftritt, der normalerweise Bomben produzierte, wird dieser abgefangen, als ob Sie sich im Debugger befänden. Sie können dann (fast) so debuggen, als ob Sie das Programm vom Debugger aus geladen hätten. Nur der Befehl LEXECUTE kann nicht mehr benutzt werden.

Sie können den residenten Debugger auch mit dem Programm CALL.PRG aufrufen. Den Source dazu finden Sie im UTILITY.OMI-Ordner. Da das Programm nicht wissen kann, ob der Debugger wirklich im Speicher liegt, sieht er nach, ob vor dem Busfehlervektor die Texte „Σ“ und „soft“ stehen. Ist dies der Fall, holt er sich aus dem Langwort davor die Adresse, an der er den Debugger starten kann. Sonst gibt er eine Fehlermeldung aus. Wenn der Debugger mit CALL.PRG aufgerufen wird, können Sie auch LEXECUTE verwenden.

Den residenten Debugger können Sie ganz normal verlassen. Er bleibt weiterhin resident im Speicher. Der Befehl RESIDENT funktioniert im residenten Debugger wie EXIT. Um den Debugger zu löschen, muß ein RESET außerhalb des Debuggers ausgelöst werden oder ein Kaltstart ausgelöst werden.

Es gibt noch eine zweite Möglichkeit, den Debugger resident zu halten: Sie speichern die Info-Datei mit „SAVE !R“ oder „SAVE !HR“ ab. Das „R“ steht für „resident“. Der Debugger wird dann automatisch resident gehalten, wenn Sie ihn laden. Diese Möglichkeit ist in der Praxis ziemlich ungewöhnlich.

Wenn der Debugger als erstes File⁸ im AUTO-Ordner liegt, wird er ebenfalls automatisch resident. Die Info-Datei muß dann im Hauptinhaltsverzeichnis Ihres Boot-Laufwerk liegen.

Wenn Sie während des Ladevorgangs des Debugges CONTROL gedrückt halten, wird der Debugger ebenfalls automatisch resident. Das ist besonders dann praktisch, wenn Sie den Debugger nachträglich vom Desktop aus laden wollen.

6.13 Debugger-Demo

In diesem Abschnitt soll für den Einsteiger einmal demonstriert werden, wie man mit einem Debugger überhaupt umgeht. Außerdem wird Ihnen der

⁸für Fortgeschrittene: Der Linien-Vektor darf noch nicht geändert worden sein

Sinn einiger Befehle vielleicht verständlicher, wenn Sie deren Anwendung in der Praxis sehen. Auf der Programmdiskette befindet sich das Programm Screendump (SCRNDUMP.PRG und SCRNDUMP.SRC). Nachdem Sie es geladen haben, befinden Sie sich gleich wieder auf dem Desktop. Wenn Sie ab jetzt ALTERNATE-HELP drücken, wird keine Hardcopy mehr auf dem Drucker ausgegeben, sondern auf Diskette oder Festplatte gespeichert. Es gibt zwar schon eine ganze Reihe von Programmen, die das gleiche tun, es geht hierbei aber nur um die Demonstration des Debuggers. Das Programm enthält noch einen Fehler, den wir nun zu beheben versuchen. Die Datei SCRNDUMP.SRC ist der fehlerfreie Sourcetext, für dieses Demo benutzen Sie bitte SCRNRERR.SRC.

Das Programm Screendump lässt sich zwar laden, und man landet auch wieder auf dem Desktop; nach ALTERNATE-HELP kommt aber eine ganz normale Drucker-Hardcopy (Frust!). Wir starten also den Debugger (Doppelklick auf OM-DEBUG.PRG). Weil wir mit Assembler und Debugger „gleichzeitig“ arbeiten wollen⁹, geben wir erst einmal RESIDENT ein (siehe Kapitel 6.12 auf Seite 174). Wir kommen zwar wieder aufs Desktop zurück, der Debugger befindet sich aber weiterhin im RAM, fängt alle „Bomben-Vektoren“ ab und kann vom Assembler aufgerufen werden.

Nun laden wir den OMIKRONAssembler und den Source SCRNRERR.SRC. Stellen Sie im Menü „Editor 2...“ bei „Sprung in den Debugger“ auf „nichts“. Dann assemblieren wir mit F1. Dann klicken Sie auf den Button „DEBUGGER“. Dadurch wird der Debugger aufgerufen und das assemblierte Programm übergeben (siehe Kapitel 5.8 auf Seite 104). Mit ALTERNATE-A erreichen wir das gleiche wie mit F1 und dem Klick auf den Default-Button (drückt sich nur einfacher).

Auf dem Bildschirm erscheint die Meldung:

```
Start des Text-Segments : $05B46E Länge: $00000118
Start des Data-Segments : $05B586 Länge: $0000000C
Start des BSS-Segments  : $05B592 Länge: $00000022
Erste freie Adresse      : $05B5B4
```

Auf Ihrem Rechner werden Sie wohl andere Adressen erhalten, sie hängen von den Accessories u. ä. ab.

⁹ ...was aber mindestens 1 MB Speicher erfordert. Wenn Sie nur einen 520 ST haben, verzichten Sie auf den RESIDENT-Befehl und laden Sie das Programm mit LEXECUTE

Wenn Sie nur ein halbes Megabyte RAM haben, müssen Sie das Programm vom Assembler aus abspeichern, den Assembler verlassen und vom Debugger aus mit LEXECUTE das Programm laden (siehe Kapitel 6.6.6 auf Seite 143). Danach erscheint eine entsprechende Meldung.

Um ein Programm zu laden, können wir den DIRECTORY-Befehl benutzen. Vor jedem Programm steht ein „**lo**“ (Abkürzung für LOAD, das wir mit „**le**“ (Abkürzung von LEXECUTE) überschreiben können.

So, nun haben wir unser Programm auf irgendeine dieser Arten in den Speicher bekommen und reloziert¹⁰. Zuerst einmal können wir feststellen, ob unsere VBL-Routine bei ALTERNATE-HELP überhaupt angesprungen wird. Dazu setzen wir einen Breakpoint an ihren Anfang. Wir müssen dafür natürlich die Adresse der Routine ermitteln.

Wir können z. B. mit LIST durch das Programm blättern, bis wir „**tst.w \$4EE.w**“ gefunden haben (so fängt unsere VBL-Routine an). Dann setzen wir z. B. Breakpoint 1 auf die Adresse. Mit ESC können wir den Rest des Bildschirms löschen. Dann tippen wir „**b1=**“ und klicken mit der rechten Maustaste auf die Adresse, die links vor „**tst.w \$4EE.w**“ steht, wodurch diese hinter „**=**“ kopiert wird, und drücken RETURN.

Das ist natürlich sehr umständlich. Sie müssen sich für dieses Verfahren merken, wie Ihre gesuchte Routine aussieht. Das kann bei längeren Programmen schon schwieriger werden. Einfacher geht es, wenn Sie beim Assemblieren eine Symboltabelle erzeugen. Dann werden die Namen aller Routinen mit abgespeichert. Versuchen Sie es einmal:

Verlassen Sie den Debugger mit CONTROL-HELP. Nun befinden Sie sich wieder im Assembler.¹¹ Mit F1 assemblieren Sie das Programm neu und klicken eine Symboltabellenart an (in diesem Fall ist es egal welche, da Sie das Programm nicht abspeichern, sondern nur an den Debugger übergeben wollen.). Dann wählen Sie wieder den Debugger an.

Wenn Sie jetzt das Programm listen, steht links von der VBL-Routine der Symbolname: „**vbl_init:**“. Sie können die Adresse aber noch einfacher ermitteln. Sie können sich mit SYMBOLTABLE alle Symbole anzeigen lassen. Dann

¹⁰das ist vor der Ausführung eines Programmes notwendig, wenn es nicht positionsunabhängig ist (gibt der Assembler nach der Assemblierung aus). Deshalb dürfen wir statt LEXECUTE nicht LOAD verwenden! Ferner wird bei ersterem eine Basepage angelegt.

¹¹Auf einem 512KB-Rechner müssen Sie natürlich den Assembler laden...

führen Sie einen Doppelklick auf dem Symbol „vbl_init“ aus, dann wird ab dieser Adresse gelistet.

Sie können auch statt der Adresse bei LIST das Symbol angeben (mit einem Punkt davor), also in unserem Fall: „list .vbl_init“.

Es gibt noch eine weitere Möglichkeit, eine Sourcetext-Stelle im Debugger wiederzufinden. Verlassen Sie dazu wieder den Debugger (CONTROL-HELP) und suchen Sie im Sourcetext die Routine. Setzen Sie dann mit CONTROL und einer Ziffer¹² einen Marker. Nach der Neuassembly springen Sie wieder in den Debugger und drücken SHIFT-F6 (Marker). Dort sehen Sie dann die Adresse der VBL-Routine. Mit dem Button links davon können Sie ab dieser Adresse listen. Die Werte der Marker finden Sie auch in den Variablen ^M0 bis ^M9 (siehe Kapitel 5.8 auf Seite 104).

So, nachdem wir nun zum x-ten Mal an der VBL-Routine angekommen sind, setzen wir endlich unseren Breakpoint. Statt mit „B1= Adresse“ geht es auch mit „LET B1= Adresse“, da die Variablen ^B0 bis ^BF auch die Breakpointwerte enthalten. Aber auch hier gibt es eine weitere Möglichkeit (um die Verwirrung komplett zu machen): Wenn Sie mit BREAKPOINTS oder LET einen Breakpoint setzen wollen, müssen Sie immer einen Teil des Bildschirms löschen. Das ist natürlich umständlich und störend. Deshalb können Breakpoints auch mit der Tastenkombination CONTROL-B und der Breakpointnummer danach eingegeben werden. Stellen Sie also den Cursor auf die Zeile, in die der Breakpoint soll, drücken Sie CONTROL-B und anschließend 1.

Nun müssen wir unser Programm starten. Das können wir mit dem Befehl GO oder mit dem entsprechenden Eintrag in der Menüleiste. Der Bildschirm flackert kurz, weil der Debugger vor der Ausführung des Programms natürlich auf den Programm-Bildschirm umschalten muß. Dann meldet sich der Debugger mit:

```
.49 - Programmende bei TRAP #1 an Adresse $xxxxxx  
GEMDOS - Funktion ##31 = Ptermres()
```

Nur eine Kleinigkeit haben wir vergessen: Wenn wir jetzt ALTERNATE-HELP drücken, wird weder eine Hardcopy- noch unsere Abspeicherroutine angesprungen, da der Debugger ja einen eigenen Tastaturreiber hat, der so etwas nicht unterstützt.

¹²nicht auf dem Zehnerblock!

Also müssen wir an unser Programm zum Debuggen eine Endlosschleife o. ä. anhängen. Dann bleibt der Tastaturtreiber des Betriebssystems aktiv, bis wir manuell in den Debugger zurückkehren. Wir verlassen also den Debugger erneut.

Dann fügen wir im Assembler die Zeile „**DEBUG: bra.s DEBUG**“ vor dem TRAP #1 (Ptermres) ein. Wir assemblen neu und gehen wieder in den Debugger. Wir suchen die VBL-Routine (mit welcher Methode, ist egal), setzen Breakpoint 1 und starten mit GO.

Statt den Debugger zu verlassen und das Programm neu zu assemblen, können wir auch im Debugger mit LIST den TRAP #1 für Ptermres suchen und ihn durch „bra.s *“ überschreiben.

Wir drücken ALTERNATE-HELP und ... der Drucker rattert los (Nerv!). Unsere VBL-Routine wird also wirklich nicht angesprungen. Nächster Versuch: Wir setzen Breakpoint 1 auf den Anfang der VBL-Init-Routine. Wir suchen mit LIST „move.w \$454.w,D0“ und tippen wieder „b1=Adresse“. Statt selbst mit LIST das Programm nach der VBL-Routine durchzusuchen, können wir auch den Debugger mit „**hunt ,!move.w \$454.w,d0**“ dazu auffordern.

Da der PC jetzt nicht mehr wie unmittelbar nach LEXECUTE bzw. dem Start des Debuggers auf dem Anfang des Programms steht, können wir es nicht mehr mit GO allein starten, sondern schreiben „g text“. „text“ ist eine Variable und steht für den Anfang des Text-Segments. Wir können den Anfang auch mit dem Befehl INFO ermitteln, der uns den Anfang aller Segmente und noch weitere Informationen über die Speicherverteilung gibt.

Wir starten also das Programm. Nach kurzem Aufblitzen des Bildschirms meldet der Debugger, daß er bei Breakpoint 1 angekommen ist. Also muß der Fehler in dieser Routine liegen. Nun entscheiden wir uns, den Fehler im Einzelschrittmodus zu suchen (es kommen ja nur noch 14 Befehle in Frage).

Um einen einzelnen Befehl auszuführen, drücken wir F1. Die Register werden, wenn nötig, geändert und der PC in dem Listing markiert. Die Unterschiede der einzelnen Trace-Funktionen finden Sie im Kapitel 6.5.1 auf Seite 117.

Wir führen nun den ersten Befehl aus: **move.w \$454.w,D0**. In D0 wird die Länge der VBL-Queue (Anzahl der VBL-Slots) geschrieben. Klingt ganz vernünftig. **subq.w #1,D0**; eins davon abziehen. **movea.l \$456.w,A0**, die Adresse der VBL-Queue wird nach A0 geladen. In der VBL-Queue stehen

die Adressen aller VBL-Routinen. `moveq #-1,D2;` in D2 steht der Fehler, wir setzen erst D2 auf -1: „Fehler aufgetreten“ und löschen D2, wenn wir unsere VBL-Routine installieren konnten. `loop0: tst.l (a0)+ / dbeq D0,loop0,` wir testen jeden VBL-Slot auf Null, denn wenn er gleich Null ist, ist er noch frei. Wenn einer belegt ist, steht in ihm die Adresse der VBL-Routine, die ja immer ungleich Null sein muß. Die Schleife wird beendet, wenn ein freier Slot gefunden oder die ganze Queue durchsucht wurde. `bne.s return;` wenn kein Slot gefunden wurde, wird die Routine beendet — D2 steht ja schon auf -1 = Fehler. `move.l #(Adresse),A0;` die Adresse der eigenen VBL-Routine wird in — Moment mal, das darf ja wohl nicht wahr sein!!! Das soll natürlich nicht „nach A0“, sondern „indirekt A0“ heißen, denn A0 zeigt ja noch auf den freien Platz in der VBL-Queue. Durch den Postinkrement beim Schreiben ist A0 jedoch ein Longword zu hoch, das können wir durch einen Predekrement beim Schreiben der Adresse beheben. Wir ersetzen das „,A0“ durch „,-(A0)“. Endgültig müssen wir es natürlich im Sourcetext ändern.

Damit ist unser Debugger-Demo beendet. Die meisten (und die mächtigeren) Befehle haben wir hier nicht verwendet, damit das Demo für den Anfänger nicht zu unübersichtlich wird. Wir hoffen, daß Ihnen jetzt die Methode des Einkreisens von Fehlern ungefähr klar geworden ist.

Anhang A

Die Makro-Version

Wir arbeiten zur Zeit an einer stark erweiterten Version des OMIKRON Assembler. Sie wird wahrscheinlich folgende Verbesserung haben:

- Makros,
- Linker,
- Teach-In-Tasten,
- Include-Dateien,
- Sourcecode-Debugging ...

Wenn Sie noch Verbesserungsvorschläge haben, schreiben Sie uns bitte (Die Adresse steht auf der Titelseite).

Wenn Sie Ihre Registrierkarte eingesandt haben, werden wir Sie informieren, sobald die neue Version fertig ist.

Anhang B

Ein kleines internes Problem...

Dann gibt es noch eine weitere Dialogbox, die wir wohl erklären müssen, obwohl wir hoffen, daß Sie sie nicht erreichen werden:

Der OMIKRON.Assembler fängt (hoffentlich) alle Fehler ab, die entstehen können (also Busfehler, Adressfehler usw.). Dann gibt er alle Daten- und Adressregister, und was sonst noch zur Fehlersuche praktisch ist, in dieser Dialogbox aus. Schreiben Sie bitte alle diese Werte ab (für die Schreibfaulen gibt es den „Drucken“-Button, der alles auf einem beliebigen Drucker (naja gut, einen Laserdruckertreiber haben wir nicht drin) ausgibt. Mit dem „Ende“-Button wird dann der Assembler verlassen (ohne irgendetwas zu speichern), mit dem „zurück“-Button kommt man wieder in den Editor. Versuchen Sie dann, den Fehler noch einmal zu produzieren, indem Sie das gleiche wie beim letzten Mal tippen. Wenn der Fehler auch reproduzierbar ist, wenn Sie den Rechner ausgeschaltet haben und das ganze von vorne versuchen, dann schicken Sie uns bitte den Ausdruck dieser Dialogbox mit einer weiteren Beschreibung des Fehlers zu. Wenn der Fehler nur bei bestimmten Sources auftritt, packen Sie diese am besten dazu. Sie können auch Ihren Assembler dazulegen, wenn auf der Diskette noch Platz ist. Dann können wir Ihnen gleich eine neue Version zuschicken, wenn wir den Fehler behoben haben. Eine Adresse von Σ-soft finden Sie auf der Titelseite.





Abbildung B.1: Ein kleines internes Problem

Anhang C

Einige Begriffserklärungen

In dieser Anleitung haben wir einige Fachausdrücke benutzt, die jedem Profi bekannt sind, dem Anfänger aber vielleicht Probleme bereiten. Daher haben wir eine alphabetische Liste dieser Wörter angefügt.

ASCII: Abkürzung für „American Standard Code of Information Interchange“. Damit ist gemeint, daß ein Wert als Zeichen (Buchstabe, Ziffer, Sonderzeichen) angezeigt wird. Jedem Zeichen, das der Computer kennt; ist ein ASCII-Wert zugeordnet (A=65, B=66 ...). ASCII-Zeichen werden durch ein vorangestelltes „ oder ' gekennzeichnet. Sie können das Zeichen auch zwischen zwei Anführungszeichen setzen. Eine Liste aller ASCII-Zeichen finden Sie in Kapitel D.3 auf Seite 192.

Basepage: Das Betriebssystem legt beim Starten eines Programms vor diesem eine sogenannte Basepage an. Darin erfährt das Programm etwas über seine Länge und etwas über seinen Parent (über das Programm, das es aufgerufen hat). Der Aufbau der Basepage ist im Anhang D.2 beschrieben.

binär: oder auch dual heißt, das eine Zahl im System der Basis 2 angezeigt wird (nur die Ziffern 0 und 1). Binäre Zahlen werden mit einem % vor der Zahl gekennzeichnet. Näheres entnehmen Sie bitte weiterführender Literatur.

Breakpoint: Ein Breakpoint dient dazu, den Programmablauf kontrollieren zu können. Nehmen wir an, Sie wollen ein Programm testen, von dem Sie wissen, in welchem Teil es fehlerhaft ist. Nun wäre es unsinnig, das Programm Schritt für Schritt durchzuarbeiten, bis Sie zu der eigentlich interessanten Routine kommen. Daher können Sie am Anfang der Routine einen Breakpoint setzen und das Programm mit GO starten. Das Programm wird nun bis zum Breakpoint normal abgearbeitet; wenn das Programm auf den Breakpoint stößt, meldet sich der Debugger wieder.

Wie funktioniert nun ein Breakpoint? Wenn Sie ein Programm starten, schreibt der Debugger an jedem Breakpoint einen illegalen Befehl (\$4FAC) und merkt sich den alten Inhalt. Wenn der Prozessor auf \$4FAC stößt, führt das zur Exception 4. Er schreibt die alten Befehle zurück und meldet Breakpoint „x“. Deshalb ist es auch nicht möglich, Teile, die im ROM stehen, mit Breakpoints zu versehen - man kann ja nichts hineinschreiben.

Cursor: Das schwarze blinkende Kästchen zeigt die momentane Schreibposition an. Das heißt, wenn Sie ein Zeichen über die Tastatur eingeben, wird es an die Position des Cursors geschrieben. Er kann mit den Cursorsteuertasten (4, 6, 8 und 2) bewegt oder mit der Maus gesetzt werden.

Debuggen: Das Wort kommt aus dem Englischen und bedeutet: entwanzen. Als Wanzen (Bugs) werden Programmfehler bezeichnet, seit dem solch ein Tierchen in einem alten Röhrencomputer einen Fehler verursacht hatte.

Default: In Dialogboxen und Alerts sind Default-Buttons dicker umrahmt als normale. Die sie kann man auch anwählen, indem man RETURN drückt.

dezimal: Dezimale Zahlen werden mit einem „.“ vor der Zahl gekennzeichnet.

Doppelklick: zweimal schnell hintereinander eine Maustaste zu drücken.

Fx: Damit ist die Funktionstaste x gemeint (Funktionstasten sind die rautenförmigen grauen Tasten oben an der Tastatur.).

hex: Abkürzung für hexadezimal, was eigentlich sedezial heißt (es hat sich aber so eingebürgert). Hexadezimal bedeutet, daß ein Wert in dem Zahlensystem der Basis 16 angezeigt wird (eine nähere Erläuterung entnehmen Sie bitte weiterführender Literatur). Sie werden mit einem \$ gekennzeichnet (wenn nichts weiter angegeben wird, nimmt der Debugger an, daß es sich um eine hexadezimale Zahl handelt).

klicken: mit der Maus auf einen Befehl, eine Zahl o. ä. zeigen und eine Maustaste (zumeist die linke) drücken.

Longword: ein Longword (oder zu Deutsch Langwort) ist ein Wert, der 4 Byte lang ist (zuerst steht das höchstwertige Byte). Longwords müssen beim 68 000 stets auf geraden Adressen liegen.

Offset: Eine Differenz zwischen der Adresse eines Datensegments und der Adresse eines Datums innerhalb des Datenblocks nennt man Offset. Ein Beispiel: Am Anfang eines Programms wird die Adresse des BSS in ein Adreßregister geladen. Dann kann man auf jedes Element des BSS zugreifen, indem man die Adresse relativ zum Anfang des BSS angibt (eben den Offset) und dazu die Adresse des BSS addiert. Diesen Offset kann man mit RS oder BASE berechnen.

PC (oder Program-Counter): Der PC ist ein internes Register des Prozessors, das auf die Stelle im Speicher zeigt, wo das Programm weiter ausgeführt werden soll. Bei der Ausführung eines Befehls wird der PC um die Befehslänge erhöht, d. h. auf den Anfang des nächsten Befehls gesetzt. Wichtig ist der Stand des PCs z. B. beim Einzelschritt-Durchlauf eines Programms im Debugger — der PC zeigt dann an, wo Sie sich gerade im Programm befinden. Weiteres entnehmen Sie bitte Ihrem Assemblerhandbuch.

Pfad: Mit dem Pfad wird angegeben, wie eine Datei heißt und wo sie sich befindet. Näheres entnehmen Sie bitte der Anleitung zum ST.

Pseudoopcode: Unter einem Pseudoopcode versteht man einen Befehl, der wie ein Opcode (Assemblerbefehl) im Programmtext steht, aber nicht in Maschinencode umgesetzt wird, sondern die Assemblierung steuert. Pseudoopcodes sind zum Beispiel DC (Define Constant; setzt eine Zahl

an der Stelle des DCs ins Programm) und OUTPUT (legt den Filenamen fest, unter dem das assemblierte Programm abgespeichert wird).

relozieren: Wenn man ein Programm für den ST schreibt, weiß man in der Regel nicht, wo das Programm liegen wird, wenn es läuft. Da es so programmiert sein sollte, daß es überall liegen kann, muß man sein Programm entweder positionsunabhängig (position independent) schreiben, das heißt keine absoluten Adressen verwenden, oder mit einer Reloziertabelle versehen (macht der Assembler automatisch), wo alle absoluten Adressen eingetragen sind. Wenn nun solch ein Programm gestartet wird, sieht das Betriebssystem nach, ob eine Reloziertabelle vorhanden ist und paßt die Adressen bei Bedarf an.

Im Debugger können Sie selbst entscheiden, ob reloziert werden soll oder nicht: LOAD lädt ein Programm nur in den Speicher, LEXECUTE lädt und reloziert.

Der Aufbau der Reloziertabellen ist im Anhang D.4 auf Seite 192 erklärt.

Scrollen: Wenn Sie mit den Cursortasten bei LIST, DISASSEMBLE oder DUMP/MEMORY den Cursor nach unten beziehungsweise oben zu bewegen versuchen, obwohl er schon am Rand steht, wird der Inhalt des Bildschirms verschoben und unten oder oben steht eine weitere Zeile, so als ob der Bildschirm ein verschiebbbares Fenster wäre, durch das man einen Ausschnitt aus dem ganzen Speicherbereich sehen könnte.

Source: Source (englisch Quelle) nennt man ein Programm, das noch nicht assembliert wurde (das lesbare Programm) — auch Quelltext genannt.

Space: Englisch für Leertaste oder Leerzeichen (Das Wort wurde langsam eingedeutscht)

Symboltabelle: Dieses Wort hat leider zwei verschiedene Bedeutungen:

1. In der Symboltabelle stehen alle Labels, die in Ihrem Programm verwendet werden, mit den Adressen, auf die das Label zeigt (relativ zum Programmanfang). Nach der Assemblierung können Sie angeben, ob die Symboltabelle an das Programm angehängt werden soll. Der Sinn einer Symboltabelle ist, daß beim Listen eines

Programms mit dem Debugger die Labels mit angezeigt werden können (damit man sich besser im Programm zurechtfinden kann).

2. Es gibt aber auch noch eine Symboltabelle in einem anderen Zusammenhang: Dateien mit der Endung SYM beinhalten nur Konstantendefinitionen und können zu einem Sourcetext geladen werden, um dort die Definitionen zu nutzen, ohne die EQUs jedesmal im Programmtext zu haben.

Word: ein Word (oder eingedeutscht Wort) ist ein Wert, der zwei Byte lang ist (erst das höherwertige, dann das niederwertige Byte).

Anhang D

Aufbau von Betriebssystemstrukturen

Tabelle D.1: Der Programmheader

\$00	word	\$601A; kennzeichnet ein relozierbares File
\$02	long	Größe des Textsegments in Byte
\$06	long	Größe des Datensegments in Byte
\$0A	long	Größe des Block-Storage-Segments (BSS) in Byte
\$0E	long	Größe des Symbolebene in Byte
\$12	long	0 (reserviert)
\$16	long	Anfang des Textsegments (wird vom GEMDOS nicht unterstützt; enthält immer 0)
\$1A	word	-1, wenn keine Reloziereinformationen vorhanden sind. (vor dem TOS 1.4 ist dies fehlerhaft; die Programmdatei wird nicht geschlossen)!

Tabelle D.2: Die Basepage

\$00	long	Anfangsadresse der TPA
\$04	long	Ende der TPA +1
\$08	long	Anfang des Text-Segments
\$0C	long	Länge des Text-Segments
\$10	long	Anfang des Data-Segments
\$14	long	Länge des Data-Segments
\$18	long	Anfang des Block-Storage-Segments (BSS)
\$1C	long	Länge des Block-Storage-Segments (BSS)
\$20	long	Zeiger auf Default-DTA (Zeigt zunächst in die Kommandozeile)
\$24	long	Zeiger auf die Basepage des Parents des aufrufenden Programms) (bei Accessories = 0)
\$28	long	reserviert
\$2C	long	Adresse der Environment-Strings
\$30-\$35	byte	tatsächliche Handles der Standardkanäle
\$36	byte	reserviert
\$37	byte	Default-Laufwerk
\$38-\$79		wird GEMDOS-intern verwendet
\$80-\$FF		Kommandozeile (wird durch PEXEC übergeben). Im ersten Byte sollte die Länge der Zeile stehen (ohne CR und Null), zudem sollte sie durch ein CR und ein Nullbyte terminiert sein.

D.1 Boot-Sektor

Der Aufbau des Bootsektors ist in Tabelle D.3 beschrieben. Dazu noch folgende Ergänzungen:

Das letzte Word wird meist dazu verwendet, die Prüfsumme über den Bootsektor auf \$1234 zu setzen.

Wie konvertiert man nun das eklige low Byte/high Byte-Format in das normale high Byte/low Byte-Format? Einfach ist es, wenn das low Byte an einer ungeraden Adresse liegt, wie z. B. bei Nsects. Wir nehmen an, Sie haben in A0 den Anfang des Boot-Sektors:

```
move.w $14(a0),d0 ;High Byte laden 12 Taktzyklen  
move.b $13(a0),d0 ;Low Byte laden 12 Taktzyklen
```

Zusammen also 24 Taktzyklen. Schwieriger ist es bei geraden Adressen (z. B. Nsides; \$1A). Ungünstig ist folgendes:

```
move.b $1B(a0),d0 ;High Byte laden 12 Taktzyklen  
asl.w #8,d0 ;und verschieben 24 Taktzyklen  
move.b $1A(a0),d0 ;Low Byte laden 12 Taktzyklen
```

Dieses Verfahren benötigt also 48 Taktzyklen. Günstiger ist es mit MOVEP:

```
movep.w $1B(a0),d0 ;High Byte laden 16 Taktzyklen  
move.b $1A(a0),d0 ;Low Byte laden 12 Taktzyklen
```

Hierbei werden also nur 28 Taktzyklen gebraucht.

D.2 Aufbau einer Symboltabelle

Die Länge der Symboltabelle steht in der Basepage. Wenn eine Symboltabelle vorhanden ist, liegt sie hinter dem DATA-Segment. Jeder Eintrag ist 14 Byte lang:

¹L/H bedeutet, daß der Wert im MS-DOS-kompatiblen low Byte/high Byte Format abgespeichert ist (Also das niederwertige Byte zuerst). Es wird auch Intel-Format genannt.

Ein Beispiel Die haben das Label „ein_sehr_langes_Label“ im GST-Format abgespeichert.

Dann steht in der Symboltabelle:

„ein_sehr“	(die ersten 8 Zeichen)
\$82	(o. ä.; Symbolstatus)
\$48	(Erweiterung folgt)
\$00000000	(o. ä.; Symbolwert)
“_langes_Label“	(Ergänzung)
\$00	(ungenutzte Byte durch 0 gefüllt)

Es sind also im DRI-Format maximal 8 und im GST-Format maximal 8+14 = 22 Zeichen für den Symbolnamen möglich. Wenn Sie ein Programm vom Assembler direkt an den Debugger übergeben, wird die Symboltabelle in einem eigenen Format mit maximal 23 Zeichen längereichert.

D.3 Die Symboltabelle des Assemblers

Diese Symboltabellen (nicht mit den obengenannten verwechseln!) bestehen aus Konstantenzuweisungen (EQU) die zu einem Programm zugeladen werden können („Symbole laden“, siehe Seite 51). Ihr Aufbau ist in Tabelle D.5 angegeben.

D.4 Relozierinformationen

Die Relozierinformationen stehen am Ende der Programmdatei, noch hinter einer eventuellen Symboltabelle. Ihr Aufbau lässt sich nicht tabellarisch wie alle anderen Informationen beschreiben. Der Assembler berechnet alle Adressen so, als ob das Programm an die Adresse 0 geschrieben würde, und legt eine Tabelle aller Adressen an, die nach dem Programmladen angepasst werden müssen.

Die Relozierinformationen beginnen an einer geraden Adresse mit einem Longword, das auf die erste zu relozierende Adresse relativ zum Programmstart zeigt. Das Betriebssystem addiert nach dem Laden die tatsächliche

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
*	68	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	10	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5
32	20	1	1	4	8	5	8	6	1	2	3	4	5	6	7	8
40	30	0	1	2	3	4	5	6	7	8	9	:	=	>	>	>
44	40	R	R	R	S	T	U	V	W	X	Y	Z	K	L	M	H
46	50	R	R	R	S	T	U	V	W	X	Y	Z	\	/	^	~
48	56	R	R	R	S	T	U	V	W	X	Y	Z	1	2	3	4
52	60	R	R	R	S	T	U	V	W	X	Y	Z	5	6	7	8
56	62	R	R	R	S	T	U	V	W	X	Y	Z	9	10	11	12
60	72	R	R	R	S	T	U	V	W	X	Y	Z	13	14	15	16
62	78	R	R	R	S	T	U	V	W	X	Y	Z	17	18	19	20
64	80	R	R	R	S	T	U	V	W	X	Y	Z	21	22	23	24
68	94	R	R	R	S	T	U	V	W	X	Y	Z	25	26	27	28
72	88	R	R	R	S	T	U	V	W	X	Y	Z	29	30	31	32
76	96	R	R	R	S	T	U	V	W	X	Y	Z	33	34	35	36
80	80	R	R	R	S	T	U	V	W	X	Y	Z	37	38	39	40
84	92	R	R	R	S	T	U	V	W	X	Y	Z	41	42	43	44
88	98	R	R	R	S	T	U	V	W	X	Y	Z	45	46	47	48
92	88	R	R	R	S	T	U	V	W	X	Y	Z	49	50	51	52
96	84	R	R	R	S	T	U	V	W	X	Y	Z	53	54	55	56
100	80	R	R	R	S	T	U	V	W	X	Y	Z	57	58	59	60
104	76	R	R	R	S	T	U	V	W	X	Y	Z	61	62	63	64
108	72	R	R	R	S	T	U	V	W	X	Y	Z	65	66	67	68
112	68	R	R	R	S	T	U	V	W	X	Y	Z	69	70	71	72
116	64	R	R	R	S	T	U	V	W	X	Y	Z	73	74	75	76
120	56	R	R	R	S	T	U	V	W	X	Y	Z	77	78	79	80
124	52	R	R	R	S	T	U	V	W	X	Y	Z	81	82	83	84
128	48	R	R	R	S	T	U	V	W	X	Y	Z	85	86	87	88
132	44	R	R	R	S	T	U	V	W	X	Y	Z	89	90	91	92
136	40	R	R	R	S	T	U	V	W	X	Y	Z	93	94	95	96
140	36	R	R	R	S	T	U	V	W	X	Y	Z	97	98	99	100
144	32	R	R	R	S	T	U	V	W	X	Y	Z	101	102	103	104
148	28	R	R	R	S	T	U	V	W	X	Y	Z	105	106	107	108
152	24	R	R	R	S	T	U	V	W	X	Y	Z	109	110	111	112
156	20	R	R	R	S	T	U	V	W	X	Y	Z	113	114	115	116
160	16	R	R	R	S	T	U	V	W	X	Y	Z	117	118	119	120
164	12	R	R	R	S	T	U	V	W	X	Y	Z	121	122	123	124
168	8	R	R	R	S	T	U	V	W	X	Y	Z	125	126	127	128
172	4	R	R	R	S	T	U	V	W	X	Y	Z	129	130	131	132
176	0	R	R	R	S	T	U	V	W	X	Y	Z	133	134	135	136

Abbildung D.1: ASCII-Tabelle

Abbildung D.2: Scancode-Tabelle

Startadresse des Programms zu der an der zu relozierenden Adresse stehenden Adresse, die ja relativ zum Text-Segment ist (Noch umständlicher hätte man's kaum formulieren können.).

Nach diesem ersten Longword wird nur noch byteweise die Distanz zu der nächsten relozierbaren Adresse gespeichert. Wenn diese Distanz größer als 254 werden sollte, wird eine 1 für je 254 Leerbytes genommen (1 kann ja sonst nicht auftreten, da dann ungerade Adressen entstünden). Die Relozierinformationen enden mit einem Nullbyte.

D.5 Der Druckertreiber

Die Druckertreiber sind vollkommen 1st-Word-kompatibel. Zwei Druckertreiber liegen als ASCII-Source bei. Diese HEX-Dateien können mit dem Programm **MAKE_CFG.PRG** übersetzt werden. Der Aufbau kann am besten den HEX-Dateien selbst entnommen werden.

Tabelle D.3: Der Boot-Sektor

Adresse:	Breite:	Normalwert:	Beschreibung:
\$00	word	\$6038	Branch auf die Bootroutine (\$3A Byte ab dem Bootsektor), wenn der Bootsektor ausführbar ist (muß nicht sein). Filler: 6 Füllbytes
\$02-\$07		„Loader“	
\$08-\$0A			Serial: Seriennummer der Diskette
\$0B	word	\$0200	Bps: Bytes pro Sektor (L/H ¹)
\$0D	byte	\$02	Spc: Sektoren pro Cluster
\$0E	word	\$0001	Res: Reservierte Sektoren (L/H)
\$10	byte	\$02	Nfats: Anzahl der Fats
\$11	word	\$70	Ndirs: Maximale Anzahl der Einträge ins Hauptinhaltsverzeichnis (L/H)
\$13	word		Nsects: Anzahl der Sektoren auf der Diskette (beim ST \$5A0 für zweiseitige Disketten und \$2D0 für einseitige) (L/H)
\$15	byte		Media: \$F8 Einseitig/80 Tracks \$F9 Zweiseitig/80 Tracks \$FC Einseitig/40 Tracks \$FD Zweiseitig/40 Tracks Dieses Byte wird vom ST nicht abgefragt, aber aus Kompatibilitätsgründen zu MS-DOS gesetzt.
\$16	word	\$0005	Spf: Sektoren pro Fat (L/H)
\$18	word	\$0009	Spt: Sektoren pro Track (L/H)
\$1A	word		Nsides: Anzahl der Seiten (1 oder 2) (L/H)
\$1C	word	\$0000	Nhid: Anzahl der versteckten Sektoren (beim ST unbenutzt) (L/H)

Das folgende wird meist nicht benutzt (diente nur für das RAM-TOS), ist aber von Atari so dokumentiert worden:

\$1E	word	\$0000	Execflag: wenn ≠ Null, wird „COMMAND.PRG“ vor Initialisierung des Desktops geladen und gestartet.
\$20	word	\$0000	Ldmode: Wenn Ldmode = 0 ist, liegt das Betriebssystem als Datei vor, sonst liegt es ab Ssect und ist Sectcnt lang.
\$22	word	\$0000	Ssect: Startsektor des Betriebssystems
\$24	word	\$0000	Sectcnt: Länge des Betriebssystems
\$26	long	\$00040000	Ldaddr: Ladeadresse für das Betriebssystem
\$2A	long	\$00008000	Fatbuf: Zeiger auf Buffer für Fat und Inhaltverzeichnis
\$2E-\$38		„TOS.IMG“	Fname: Name der Betriebssystem-Datei
\$39	byte	\$00	Dummy: Füllbyte
\$3A-\$FF			Hier steht die Bootroutine.

Tabelle D.4: Die Symboltabelle

0-7	Symbolname (im DRI-Format), unbenutzte Zeichen mit 0 aufgefüllt
8	Symbolstatus, Bitweise aufgeschlüsselt; 1 bedeutet „Ja“: Bit 0: BSS-relativ Bit 1: Text-Segment-relativ Bit 2: Data-Segment-relativ Bit 3: External Bit 4: Registerlist Bit 5: Global Bit 6: Equated (Konstante) Bit 7: Defined reserviert; im GST-Format bedeutet \$48: Dem Symbolwert folgen 14 Byte Namensergänzung; sonst gleich 0 Bit 6: Archive (für den ALN) Bit 7: File
9	
10-13	Symbolwert

Tabelle D.5: Die Symboltabelle des Assemblers

dann 32 Byte je Symbol:	4 Byte Header: „ΣSYM“;
	Byte 0-3: Symbolwert
	Byte 4-5: \$80FD (Assembler intern)
	Byte 6-7: \$0000
	Byte 8-31: Symbolname (maximal 23 Zeichen) Rest mit 0 aufgefüllt

Anhang E

Probleme mit älteren TOS-Versionen

Damit der Debugger z. B. bei LEXECUTE den Hauptspeicher komplett wieder freigeben kann, war es notwendig, auf einige mehr oder weniger dokumentierte Systemvariablen zuzugreifen. Die beiden Variablen, welche vom Debugger benötigt werden, sind zum einen „kbshift“ (Ergebnis der BIOS(11)-Funktion) und „act_pd“ (Zeiger auf das aktuelle Programm, für das GEM-DOS nötig). Die Adressen dieser Variablen müssen im Debugger eingestellt werden.

Alle, die ein neueres TOS im Rechner haben (Blitter-TOS oder TOS 1.4), können jetzt weiterblättern, Atari hat ab dem Blitter-TOS diese Variablen dokumentiert und der Debugger kann so die Adresse selbst ermitteln. Die Mehrheit der Atari-Besitzer haben wohl das 86'er ROM-TOS. Auch diese können weiterblättern, die Adressen sind als Default eingestellt (für Interessierte: „kbshift“ liegt bei \$E1B und „act_pd“ bei \$602C).

Was, immer noch welche übrig geblieben? Ok, dann werde ich mal erklären, wie man (mit dem Debugger) diese beiden Variablenadressen ermitteln kann: Damit keine Probleme auftreten, sollten erst einmal alle Utilities entfernt werden (AUTO-Ordner und ACCs). Aber keine Sorge, der Debugger verträgt sich mit den Utilities, sie sollten nur zum Ermitteln der beiden Variablenadressen aus dem Speicher genommen werden. Nun laden wir den Debugger.

- Die Texte in Anführungszeichen sind so einzugeben und mit der Taste

RETURN abzuschließen.

Zuerst das Auffinden von „kbshift“:

1. !„MOVE.L #BFFF,-(SP)“
2. !„TRAP #D“ (Das Ausrufezeichen wird vorgegeben)
3. F1 drücken (normales Trace)
4. F4 drücken (Trap tracen)
5. mit ESC den Bildschirm löschen
6. „IF { PC}.w=4E90“ (Abbruch, wenn der PC auf „JSR (A0)“ steht)
7. „U“ für Untrace eingeben
8. F1 drücken
9. Das Listing ansehen, es muß dem Folgebeispiel (aus dem Blitter-TOS) ähnlich sehen:

```
MOVEQ #0,D0
MOVE.B $E61(A5),D0
MOVE.W 4(SP),D1
BMI.S -----
MOVE.B D1,$E61(A5) |
RTS    <-----
```

Wenn es bis auf \$E61 übereinstimmt, ist die Stelle richtig. Die Zahl, welche an der Stelle von \$E61 steht, ist unsere Adresse, welche wir notieren sollten („kbshift“-Variablenadresse).

So, hat das Auffinden von „kbshift“ geklappt? Dann ist „act_pd“ auch kein Problem mehr:

1. „RESET ALL“ eingeben und bestätigen, um den Speicher zu löschen, etc.

2. „MOVE.W #B,-(SP)“
3. „TRAP #1“
4. F1 drücken (normales Trace)
5. F4 drücken (Trap tracen)
6. mit ESC den Bildschirm löschen
7. „IF { PC}.w=4E90“ (Abbruch, wenn der PC auf „JSR (A0)“ steht)
8. „U“ für Untrace eingeben
9. F1 drücken
10. Das Listing ansehen, es muß dem Folgebeispiel (aus dem Blitter-TOS) ähnlich sehen:

```
LINK    A6,#FFFF
MOVEA.L $87CE,A0
MOVE.B $30(A0),D0
EXT.W   D0
MOVE.W  D0,(SP)
ADDQ.W  #3,(SP)
BSR.S   xxx ;Adresse unwichtig
UNLK    A6
RTS
```

Hier steht \$87CE für die Adresse der „act_pd“-Variable, sie sollte also bei Ihrer TOS-Version einen anderen Wert haben.

11. Jetzt verlassen wir den Debugger mit Shift-F10 und RETURN

So, nun müssen wir die beiden Adressen nur noch in den Debugger eintragen. Wir machen erst einmal eine Kopie vom Debugger (nicht für Freunde und Bekannte, sondern eine Sicherheitskopie; falls etwas schief geht, können wir dann immer noch auf die Kopie zurückgreifen).

Nun laden wir den Debugger erneut, um von ihm aus sich selbst zu patchen.

1. „LO OM-DEBUG.PRG“
2. „M.L START+22#“
3. Die Ausgabe müßte folgendermaßen aussehen: .I 00000E1B,0000602C,
... (der Rest der Zeile ist unwichtig)

Die erste Zahl entspricht der „kbshift“-Variablenadresse, die zweite der „act_pd“-VariablenAdresse. Wir bewegen nun den Cursor auf die erste Zahl und überschreiben diese mit unserem „kbshift“-Wert, ebenso verfahren wir mit der zweiten Zahl (dort setzen wir natürlich den „act_pd“-Wert ein).

Wenn ich die Adressen der obigen Beispiele einsetze sieht das Ergebnis so aus: „I 00000E61,000087CE, ...“

Nun übernehmen wir die Werte mit der RETURN-Taste. Wir können mit „M.L START+22#“ die Eingabe evtl. noch einmal vergleichen.

4. „SA“ zum Speichern eingeben und die Frage bejahen.
 5. Hoffen, daß alles geklappt hat.
- Nochmals möchte ich alle warnen: Wenn Ihr Rechner ein anderes TOS als die oben angegebenen hat, ist die obige Installation *unbedingt* nötig, da sonst spätestens nach dem Laden eines Programms mit „LEXECUTE“ intern alles quer läuft, ein Programmabbruch mit SHIFT-SHIFT wäre zudem auch nicht möglich.
 - Wenn Sie die Beispiele nicht nachvollziehen können, wenden Sie sich bitte an OMIKRON.Software, wir helfen Ihnen dann gern weiter.

Anhang F

Assembler und OMIKRON.BASIC

Wenn Sie von BASIC aus Assemblerprogramme aufrufen wollen, ist es häufig nötig, Variablen zu übergeben. So lange es nur wenige Werte sind, können sie direkt hinter CALL angegeben werden; bei einem größeren Array geht das aber nicht mehr. Sie haben z. B. ein Programm geschrieben, bei dem auf dem Bildschirm Zeichen ausgegeben werden müssen. Der Bildschirminhalt steht in einem zweidimensionalen Array: Indizes sind die X- und Y-Koordinaten und der Inhalt ist die Zeichenummer. Den Bildschirmaufbau wollen Sie nun in Assembler programmieren.

Sie können nun die Adresse des ersten Elements folgendermaßen ermitteln:
`LPEEK(VARPTR(Variablen(0,0)))+LPEEK(SEGPTR+20)`

Durch LPEEK(SEGPTR+20) erhält man die Adresse des Arraybuffers; durch LPEEK(VARPTR(Variablen ... den Offset für dieses Array. Es reicht auch, dem Assemblerprogramm nur LPEEK(VARPTR(Variablen(0,0))) zu übergeben — im A0-Register steht SEG PTR, man kann die Adresse also auch in Assembler berechnen. Da in unserem Beispiel Integer-Long Variablen verwendet wurden, liegen die Werte ab der Adresse als Longwords. Man kann sich also Wert für Wert mit MOVE.L (Ax)+,Dx holen. Die Werte sind folgendermaßen sortiert: (0;0), (1;0), (2;0), (0;1), (1;1), ...

Etwas unübersichtlicher wird es mit String-Variablen: Dann liegt nicht der String bei der Adresse, sondern

1. ein Zeiger auf den String relativ zu LPEEK(SEGPTR+28) (1 Langwort)
2. die Länge des Strings (1 Word)

Folgendes BASIC-Programm soll das verdeutlichen:

```

A$(0)="Test"
A$(1)="OMIKRON.Software"
A$(2)="Sigma-soft"
Adr= LPEEK( VARPTR(A$(0)))+ LPEEK( SEG PTR +20)
' Zeiger auf Array
Seg=LPEEK( SEG PTR +28)
FOR J=0 TO 2           'String 0 bis 2 ausgeben
  FOR I=0 TO WPEEK(Adr+4+6*j)-1 'Das ist die Laenge
    PRINT CHR$( PEEK( LPEEK(Adr+6*j)+ Seg+i));
  NEXT I
  PRINT
NEXT J

```

Jetzt entsprechendes in Assembler:

```

DIM A$(2)      ' Sonst dimensioniert OMIKRON.BASIC
                ' auf 10, und die Assembler-Routine
                ' versucht, 10 Texte auszugeben
A$(0)="Test"   ' Texte, die ausgegeben werden sollen.
A$(1)="OMIKRON.Software"
A$(2)="|\$\\Sigma\$\\verb|-soft"
Assprg=MEMORY(500)          'Speicher reservieren
BLOAD "PRINTARR.B",Assprg  'Programm laden
Ad= LPEEK( VARPTR(A$(0)))  'Adresse des Arrays ermitteln
CALL Assprg(L Ad)          'beim alten OMIKRON.BASIC:
'CALL Assprg(HIGH(Ad),LOW(Ad))

move.l 20(a0),a6 ;Entspricht LPEEK(SEGPTR+20),
;Also Anfang des Arraybuffers
add.l 4(sp),a6 ;a6 entspricht jetzt Adr

```

```

        move.l 28(a0),d5 ;Stringbufferadresse -> d5
        move.l -6(a6),d7 ;Anzahl der Strings-1
                      ;;-6: eindimensionale Arrays)
loop: move.l (a6)+,a5 ;Zeiger auf ersten String relativ
          ;zum Stringbuffer
        add.l  d5,a5 ;Adresse des Stringbuffers addieren
        move   (a6)+,d6 ;Laenge des Strings
        subq   #1,d6 ;Fuer DBRA eins subtrahieren

sloop:moveq #0,d0 ;D0 erstmal loeschen
        move.b (a5)+,d0 ;Ein Zeichen holen
        move   d0,-(sp) ;Der Umweg muss sein, da das Zeichen
                      ;als Word auf dem Stack stehen muss.
        move   #2,-(sp) ;Funktionsnummer
        trap   #1 ;Gemdos
        addq.l #4,sp ;Stackkorrektur

        dbra   d6,sloop ;Ganzen String ausgeben

        pea    cr(pc) ;CR und LF ausgeben
        move   #9,-(sp) ;Cconws
        trap   #1
        addq.l #6,sp

        dbra   d7,loop ;Naechster String
        rts
cr: dc.b  13,10,0 ;CR und LF

```

Über zerstörte Registerinhalte müssen wir uns innerhalb des Assemblerprogramms keine Gedanken machen: Die Register D5-D7 und A5-A6 werden durch einen Gemdos-Aufruf nicht verändert (und den Wert von D0 müssen wir uns nicht merken). Ich habe Gemdos 9 (Cconws) für den String nicht benutzt, sondern alle Zeichen einzeln ausgegeben, weil ein String nicht null-terminiert im Speicher liegen muß.

Anhang G

Laden an beliebige Adressen

Einige unsauber programmierte Software legt sich an absolute Adressen (meistens unten im Speicher, damit sie auch auf kleinen Rechnern läuft). Wenn Sie nun den Debugger normal in den Speicher laden, wird er meist dahin gelegt, wohin sich auch das Programm kopiert. Um auch solche Programme debuggen zu können, gibt es das Programm Lxxxxxx.PRG in dem Ordner UTILITY.OMI. Die „xxxxxx“ geben die Adressen an, ab welcher der OMI-KRON.Debugger liegen soll.

Das ganze funktioniert folgendermaßen: Mit F\$first wird auf der Diskette nach einer Datei mit der Maske „L.*.PRG“ gesucht und die sechs Ziffern dahinter als Adresse interpretiert. Dann wird ein Speicherbereich mit Malloc angefordert, der so groß ist, daß unmittelbar nach dem Speicherbereich der Bereich anfängt, in den der Debugger geladen werden soll. Der Debugger wird mit Pexec geladen; das Betriebssystem legt ihn in den gewünschten Bereich. Der mit Malloc angeforderte Speicher wird wieder freigegeben und der Debugger gestartet.

Der Source liegt auch im UTILITY.OMI-Ordner — Sie können damit natürlich auch andere Programme als den OMIKRON.Debugger laden. Auch der Dateinamenanfang „L.“ kann abgeändert werden, wenn er bei Ihnen zu Komplikationen führt.

Anhang H

Das Modulkonzept

So, nun noch etwas ganz feines für die absoluten Profis, die nie genug bekommen können: Das Modulkonzept des OMIKRON Assemblers. Der OMIKRON Assembler erlaubt es nach der Assemblierung (z. B. mit F1) beliebige eigene Assemblermodule aufzurufen, d. h. die 12 Buttons in der Dialogbox nach der Assemblierung können frei mit eigenen Routinen belegt werden. Neben den bekannten Möglichkeiten gibt es somit noch die Möglichkeit, ein Programm automatisch zu kodieren, eine Prüfsumme darüber zu bilden und in einem speziellen Format auf Disk abzulegen.

Zum besseren Verständnis der Dokumentation empfehle ich jedem, die Datei OM-MODUL.SRC einmal auszudrucken, um die Richtlinien am Sourcetext direkt nachzuvollziehen.

Kommen wir nun aber endlich zur Dokumentation der Schnittstelle:

H.1 Allgemeine Richtlinien:

- Das Modul darf alle Register verändern (auch den Stackpointer, wenn die Rücksprungadresse gerettet wird) *Ausnahme*: das A4-Register darf unter *keinen* Umständen geändert werden, der Assembler stürzt sonst ab!!!
- Das Modul *muß* PC-relativ, d. h. im Speicher frei verschieblich, sein. Zur Not kann eine RSC o. ä. beim init-Aufruf reloziert werden. Ferner

sollte die Adressierungsart d(PC) bzw. d(An) genutzt werden. Das Modul sollte als ABSOLUT-Programm abgespeichert werden (nur die vorgeschlagene Extension B ist richtig), da der 28-Byte-Programmheader unnötig ist (Er wird aber überlesen, falls man es doch mal vergißt)

- Dem Modul muß ein EVEN folgen, damit das folgende Modul stets auf einer geraden Adresse beginnt (Sicher ist Sicher...).
- Ein Modul wird stets om Supervisormode aufgerufen und darf diesen auch nicht verlassen.
- Punkte, welche mit Δ markiert sind, sind noch nicht getestet!

H.2 Der Aufbau des Modulheaders:

- folgender 32-Byte Header ist vorgeschrieben (Die Symbolnamen sind wahlfrei, sollten aber möglichst mit „Buchstabe“+„-“ anfangen, da der Zusammenhang dann besser deutlich wird (siehe auch im Sourcetext der mitgelieferten Module)).

```
a_start:      DC.B 'd-soft01'    ;0  Modulkennung
              bra.w  a_init      ;8  Init nach dem Laden
              bra.w  a_disable   ;12 evtl. disable
              bra.w  a_choose    ;16 Modul anwählt
              DXSET 8,' '
              DX.B 'NAME'       ;20 Name des Moduls
              DC.L a_end-a_start ;28 Laenge des Moduls
```

- Der Header beginnt mit der Modulkennung, wobei die letzten beiden Ziffern die Version des Modulkonzepts darstellen, sie werden allerdings nur bei Änderungen, welche zur Inkompatibilität (was für ein Wort) führen geändert.
- Die drei folgenden Befehle sind jeweils 2 Word lang (BRA.W !!!) Sie dürfen unter *keinen* Umständen eine andere Länge bekommen!

- Der Modulname besteht aus maximal 8 Großbuchstaben (Ziffern und Sonderzeichen sind auch möglich), welcher mit Spaces auf die Länge von 8 Zeichen aufgefüllt sein muß.
- Das letzte Langwort gibt die Gesamtlänge des Moduls an. Die Modullänge hängt nur vom Speicherplatz im Ram ab, sie ist also nicht auf 32 KB o. ä. beschränkt.

H.3 Die Routine „init“:

Diese Routine wird direkt nach dem Laden der Module, noch vor der Initialisierung des Assemblers angesprungen, man kann nun z. B. eine RSC relozieren (ACHTUNG! eigenes Format bei Resourcen)

Übergabeparameter:

A6 — Basisadresse des Moduls

Rückgaben in D0

```
 @_init_debugger: EQU 1    ; init ok, Debugger-Modul
 @_init_ok:      EQU -1   ; init ok
 @_init_error:   EQU 0    ; init fehlerhaft
                      ; Modul nicht uebernehmen
```

Neu ist der Rückgabewert 1, er besagt, daß das Modul den Debugger aufruft und der Assembler eine spezielle Symboltabelle für den Debugger erzeugen soll (wenn gewünscht), früher wurde die normale Symboltabelle an den Debugger übergeben, der dann ganz schön gestaut hat (Interne Bomben...) Wenn dieser Rückgabewert allerdings vergessen wird (und stattdessen -1 zurückgegeben wird) nimmt der Debugger einfach gar keine Symboltabelle an, da der Debugger, wenn er vom Assembler aufgerufen wird, keine normale Symboltabelle erlaubt.

Ach ja, noch was, zu diesem Zeitpunkt ist nur Mshrink() vom Assembler ausgeführt worden und der Prozessor in den Supervisormode geschaltet worden, es sind somit noch keine der Zusatzfunktionen (siehe unten) verfügbar, d. h. das GEMDOS muß mit TRAP #1 und nicht über JSR @trap_1(A4) aufgerufen werden!!!

H.4 Die Routine „disable“:

Die Routine wird vor der Ausgabe der Dialogbox des Assemblers aufgerufen, sie dient dazu den Menüpunkt zu „enable“ oder „disable“, da es evtl. nötig ist einen Aufruf bei bestimmten Voraussetzungen zu verhindern (siehe Sourcetext). Noch was, die Routine darf den Modulnamen ändern; er wird dann in der Dialogbox auch entsprechend geändert.

Übergabeparameter:

- A6 — Basisadresse des Moduls
- A5 — Startadresse des erzeugten Programms
(Format wie auf der Diskette)
- A4 — Assembler intern
- A3 — 2 KB-Diskbuffer
- A2 — 32 KB-Screenbuffer (möglichst nicht benutzen, gibt aber keine Probleme, wenn er doch benutzt wird)
- D7 — ORG-Adresse (0 = keine ORG-Adresse, wenn das oberste Byte gesetzt ist, wurde ein ORG Adr, ^ eingegeben)
- D6 — 0 - OPT D- oder OPT X-
 - 1 - OPT D+
 - 2 - OPT X+
- D5 — = 0 - relatives Programm erzeugt
≠ 0 - relozierbares Programm erzeugt
- D4 — Endadresse+1 des erzeugten Programms
- D3 — = 0 - kein Debugger vorhanden
 - <0 - Debugger resident vorhanden
 - >0 - reserviert

Rückgaben in D0

```
0_enable:      EQU -1 ;Modul darf aufgerufen werden
0_disable:     EQU 0  ;Modul wird disabled
```

H.5 Die Routine „choose“:

Diese Routine wird aufgerufen, wenn der Menüpunkt angewählt wurde.

Übergabeparameter:

wie bei „disable“

Rückgaben in D0 normales Verlassen

```
@_debug:      EQU 2 ;Dialog verlassen,  
               ;in den Debugger springen  
 @_continue:   EQU 1 ;Im Dialog bleiben  
               ;(zur Anwahl weiterer Funktionen)  
 @_exit:       EQU 0 ;Dialog verlassen
```

Fehlermeldungen ausgeben (werden bei den Funktionen (s. u.) genutzt)

```
@_no_memory:  EQU -1 ;der Speicher reicht nicht \atn  
 @_fail:       EQU -2 ;FAIL-Fehler (nicht benutzen) \atn  
 @_file_error: EQU -3 ;Fehler bei Dateien \atn  
 @_disk_full:  EQU -4 ;Die Disk ist voll \atn
```

H.6 Die vorhandenen Prozeduren:

Bei Fehlern (siehe File-Funktionen) wird automatisch abgebrochen! Außer den angegebenen Registern werden keine verändert

1. Zuerst mal was einfaches...

- Aufruf des Fileselectors:

```
MOVE.L #'SAB',D0      ;'BAS'  
LEA     titel_txt(pc),A0  
JSR     @fsel_input(A4)
```

D0—Die Extension (Buchstaben in umgedrehter Reihenfolge!)
 A0—Zeiger auf einen max.31 Zeichen langen String, welcher
 —eine Überschrift für einen TOS 1.4-Fileselector enthält.

Der Fileselector wird dargestellt, der Pfad und Filename wird intern verwaltet. Nach dem Aufruf wird ein Redraw vom Bildschirm gemacht.

- Datei existiert bereits

Testen, ob das gewählte File existiert, wenn ja, Alert ausgeben (Überschreiben ja/nein) und dann evtl. das überflüssige File löschen (interner Filename wird verwendet).

```
JSR      @overwrite_file(A4)
```

- Das mit dem Fileselector gewählte File erstellen

```
MOVEQ   #0,D0  
JSR      @fcreate(A4)
```

D0—Das Fileattribut

- Das mit dem Fileselector gewählte File öffnen Δ

```
MOVEQ   #0,D0  
JSR      @fopen(A4)
```

D0—Das R/W-Flag

- Daten aus dem geöffneten File lesen Δ

```
MOVE.L  #$20000,D0  
LEA     $1F8000,A0  
JSR      @fread(A4)
```

D0—Anzahl der Bytes

A0—Ladeadresse

- Daten ins geöffnete File schreiben

```
MOVE.L  #$20000,D0  
LEA     $1F8000,A0  
JSR      @fwrite(A4)
```

D0—Anzahl der Bytes

A0—Adresse ab der geschrieben wird

- Die geöffnete Datei schließen

```
JSR      @fclose(A4)
```

keine Parameter nötig

- Die aktuelle Datei ohne Nachfrage löschen Δ

```
JSR      @fdelete(A4)
```

keine Parameter nötig

- Eine Dezimalzahl in einen Buffer schreiben

```
LEA      buffer(PC),A0
MOVE.L  #12345678,D0
MOVEQ   #7,D1
JSR      @dezout(A4)
```

Die Zahl in D0 wird als Dezimalzahl nach A0 geschrieben, die max. Stellenzahl steht in D1 (minus 1!)

- Hexzahlen in einen Buffer schreiben

```
LEA      buffer(PC),A0
MOVE.L  #$12345678,D0
JSR      @hexlout(A4)
```

Die Zahl in D0 wird als Hexzahl nach A0 geschrieben. Als Unterfunktionen gibt es Ausgaben als Byte, Word, Adresse und Long

```
JSR      @hexbout(A4)
JSR      @hexwout(A4)
JSR      @hexaout(A4)
JSR      @hexlout(A4)
```

- Mauszeiger umdefinieren

```
MOVEQ   #1,D0
JSR      @grafmouse(A4)
```

D0=0 — Pfeil
=1 — Diskette
=2 — Sanduhr

- Pling!

```
JSR      @bell(A4)
```

2. Nun was für die Fortgeschrittenen...

- Dialog ab A0 behandeln

```
LEA      dialog_rsc(PC),A0
JSR      @form_do(A4)
```

A0 zeigt auf einen RSC-Baum, des eigenen Formates (siehe Kapitel H.7 auf Seite 214)

- Hexzahl ab A0 nach D0 holen

```
LEA      buffer(PC),A0
JSR      @hexlin(A4)
```

A0 enthält die Adresse, ab der die Hexzahl in ASCII steht, als Rückgabe steht die Zahl in D0, Das Z-Flag ist gesetzt, falls keine Hexzahl im Buffer steht (nur Spaces mit einem Nullbyte abgeschlossen), das N-Flag zeigt an, das eine fehlerhafte Hexzahl vorliegt, der Buffer wird ab der Fehlerstelle bis zum Nullbyte gelöscht.

3. Und zum Schluß was für die Profis...

- Das GEMDOS ausrufen ▲

```
MOVE.W  #$E,-(SP)          ;Dgetdrv()
JSR      @trap_1(A4)
ADDQ.L  #2,SP              ;Nicht vergessen!
```

Wird wie der Befehl TRAP #1 verwendet, ist anstelle von diesem *nötig*

- Auf die originalen Treiber (Tastatur etc.) schalten ▲

```
JSR      @org_driver(A4)
```

- und wieder auf die Treiber des Assemblers zurück ▲

```
JSR      @ass_driver(A4)
```

Ist nach „org_driver“ vor dem Rücksprung oder den eigenen Funktionen *unbedingt* auch nötig!!!

H.7 Der RSC-Editor

H.7.1 Der Aufbau des Formats

Die Dialogboxen müssen nach einem eigenen Format aufgebaut sein:

Zur Beachtung

- Die RSC muß stets an einer geraden Adresse anfangen, Icondaten ebenfalls.
- Texte können an beliebigen Adresse stehen.
- Bei form..do wird automatisch zentriert.
- objc_draw zeichnet stets den gesamten Baum neu.
- Wird UNDO während der form..do-Routine gedrückt, wird in die Haupt-schleife zurückgesprungen
- Die Rückgabe in D0 enthält die Buttonnummer (gemäß der RSC, dabei werden nur die Buttons gezählt und zwar ab 1)

Allgemeiner Aufbau

Header:	Word	0
	Word	0
	Word	Breite des Baumes (für den Rahmen und den Redraw)
	Word	Höhe des Baumes
	Word	(wird intern verwaltet)

Aufbau eines Objektes	Word Word Long Word	X-Offset (zeichenweise!) zur linken oberen Ecke des Baumes Y-Offset (wie oben) Zeiger auf Text, bzw. Icon-daten Typ (siehe unten)
Ende der RSC-Daten	Word	-1 (der Wert muß negativ sein)

Die verschiedenen Typen

- Typ 0—unbelegt
- Typ 1—ausgefülltes Rechteck mit Doppelrahmen
- Typ 2—wie Typ 1, aber nicht ausgefüllt
- Typ 3—Icon (siehe Kap 4)

Wenn im Typ des Baumes (erstes Objekt) Bit 7 gesetzt wird, wird der löschende (und flackernde) Redraw unterlassen. Das Bit wird automatisch gelöscht

- Bit 2 = 1:Button (= umrahmter selektierbarer Text)
- Bit 3 = 1:Text ((fast) ohne jegliche Extras)

Wichtig: Es darf nur Bit 2 oder (!!!) Bit 3 gesetzt sein, nie beide!

Wenn Bit 2 oder Bit 3 gesetzt sind, gelten folgende Zusätze:

- Bit 0 = 1:invers (bzw. selektiert)
- Bit 1 = 1:Defaultbutton
- Bit 4 = 1:light (bzw. disabled)
- Bit 5 = 1:Exitbutton; bei Texten : = 1 — nur Ziffern bei der Eingabe
- Bit 6 = 1:Radiobutton; bei Texten : = 1 — Fett-schrift
- Bit 7 = 1:editierbarer Text (für Bit 3 = 1)

Icons

- Icons werden von 8x8 bis max. 128x128 Pixeln unterstützt.
- - Bit 8-11 des Typs: Höhe des Icons in 8 Zeilen (0=8 Zeilen, 1=16 Zeilen, ...)
 - Bit 12-15 des Typs: Breite des Icons in Bytes (0=1 Byte, 1=2 Byte, ...)
- wenn alle Bits 8-15 gesetzt sind, gilt: $15 \cdot 8 + 8 = 128$ Zeilen $15 \cdot 8 + 8 = 128$ Pixel breit
- Wenn Bit 7 im Typ gesetzt ist, werden bei Farbe jeweils 2 Zeilen mit OR verknüpft, sonst wird jede ungerade Zeile weggelassen.
- Die Icondaten müssen linear im Ram liegen, d. h. für ein 16x16-Icon:

2 Byte 1.Zeile

2 Byte 2.Zeile

.

.

.

2 Byte 16.Zeile

H.7.2 Der Editor

Um die oben beschriebenen Dialogboxen einfach erstellen zu können, haben wir einen RSC-Editor geschrieben. Er ist zwar recht einfach, erfüllt aber seinen Zweck. Zur Bedienung folgende Hinweise:

Die Größe des Alerts kann dadurch verändert werden, daß Sie auf die untere rechte Ecke klicken und das Alert auf die gewünschte Größe draggen.

Wenn außerhalb des Alerts klicken, erscheint ein Popupmenü zum Laden, Speichern und zum Verlassen des Programms. Mit „Save as RSC“ werden die Daten in einer EST-Datei gespeichert, die wieder eingelesen werden kann. Mit „Save as DC“ werden die Daten in einer ASM-Datei (Sie können natürlich auch andere Extensions verwenden) gespeichert, die nicht wieder eingelesen

werden kann. Sie können diese Datei in Ihr Assemblerprogramm einbinden und brauchen, um dieses Alert anzeigen zu lassen, nur noch die form..do Routine aufrufen und ihr den Namen zu übergeben. Als Name wird der Dateiname verwendet. Intern werden die Labels „(Dateiname).(Nummer)“, also beim Dateinamen „RSC“ die Label „RSC_1“, „RSC_2“ usw. verwendet.

Wenn Sie innerhalb des Alerts an eine leere Stelle klicken, erscheint ein Pop-upmenü, in dem Sie auswählen können, ob Sie ein Icon, einen Text oder einen Button setzen wollen. Buttons unterscheiden sich von Texten nur dadurch, daß Sie umrahmt werden. Wenn Sie in die linke obere Ecke eines Textes/Buttons/Icons klicken, können Sie es

Ohne Tasten	verschieben
Mit CONTROL	kopieren und die Kopie verschieben
Mit SHIFT	löschen
Mit ALT	zentrieren (alle Texte/Buttons in der Zeile werden zentriert)

Wenn Sie ein Objekt mit der rechten Maustaste anklicken, können Sie es editieren.

Die Iconlibrary

Icons, die Sie häufig benutzen, können Sie in einer Iconlibrary ablegen. Das bedeutet, Sie kopieren alle (z. B. mit STAD) in ein Grafikbild und speichern es unter dem Namen LIBRARY.PIC im Screen-Format ab (ins gleiche Directory wie den Editor). Der Editor lädt es dann gleich mit 'rein. Wenn Sie dann ein Icon einsetzen wollen, brauchen Sie nicht mehr zu laden, sondern nach einem BITBLT ist es da.

Achtung beim Benutzen von Farbmonitoren (nur mittlere Auflösung):

Es können keine neuen Icons gesetzt werden. Icons, die in geladenen Alerts sind, werden jedoch angezeigt und können auch verschoben, kopiert u.ä. werden.

Anhang I

Die beiliegenden Dateien

Auf der Diskette finden Sie einen Ordner mit dem Namen LIBRARYS.OMI (der Rechtschreibfehler ließ sich nicht vermeiden; „Libraries“ ist 9 Zeichen lang und somit als Filename nicht erlaubt), einen Ordner UTILITY.OMI und einen DEMOS.OMI. Darin finden Sie Demos u. ä., die in diesem Kapitel erklärt werden:

I.1 AES_DEMO.SRC und AES_DEMO.PRG

Dieses Programm dient der Demonstration von AES- und VDI-Aufrufen. AES steht für Application Environment System und übernimmt die Benutzeroberfläche, also Windows, Drop-Down-Menüs und Maus-, Tastatur-, Zeitüberwachung usw. Das VDI (Virtual Device Interface) ist für primitivere Grafikfunktionen (Zeichnen von Linien, Rahmen, Ellipsen usw.) zuständig. Eine Liste aller VDI- und AES-Routinen wäre für diese Anleitung viel zu lang — wir müssen Sie also auf entsprechende Fachliteratur verweisen (siehe Anhang J).

Wie diese Routinen aber von Assembler aus aufgerufen werden, kann an diesem Programm sehr schön verfolgt werden. Es werden 4 Windows geöffnet und mit allen Elementen abgefragt. Entsprechend der Slider-(Schieber-)stellung werden in den Windows verschiedene Ellipsen gezeichnet. Natürlich werden auch Redraw-Messages verwaltet.

I.2 AES_VDI.S

Dieses ist ein Programmskelett für AES-Programme. Er führt das nötige `appl_init`, `open_workstation` usw. aus und bietet einen einfachen Aufruf von AES-Routinen. Am Programmende sind AES-Konstanten definiert. Es ist auch markiert, welche Funktionen beim TOS 1.4 und beim GEM 2.0 hinzugekommen sind. Aus diesen Konstantendefinitionen können Sie natürlich auch eine SYM-Datei erstellen.

I.3 AUTOBOOT.SRC

Das Autoboot-Programm schaltet, wenn es im AUTO-Ordner liegt, auf Wunsch auf 50 Hz oder 60 Hz um, installiert einen Tastatur-RESET (normal mit CONTROL-ALTERNATE-DELETE; zum Speicherlöschen gleichzeitig mit der rechten SHIFT-Taste), hat einen Bildschirmschoner, der nach 1-999 Sekunden den Bildschirm abschaltet, setzt die Auflösungen (wobei auch die Software für einen automatischen Bildschirmumschalter integriert ist) und kopiert ein entsprechendes DESKTOP.INF-File.

Die Abschaltzeit und die Bildfrequenz kann im Dateinamen angegeben werden. Wenn das Programm AUTOBOOT.PRG heißt, wird kein Bildschirmschoner installiert und die Frequenz auf 60 Hz gesetzt. Nennen Sie es AUTO300F.PRG, stehen die Ziffern für die Abschaltzeit (hier also nach 300 Sekunden = 5 Minuten) und „F“ für fünfzig; „S“ für sechzig Hz.

Im AUTO-Ordner können auch DESKTOP.IN?-Dateien liegen. Die letzte Ziffer gibt die Auflösung an — 0 für niedrige bis 2 für hohe. Das Autoboot kopiert das entsprechende in das Hauptinhaltsverzeichnis und erklärt es als hidden-Datei.

I.4 BIOS.S, XBIOS.S und GEMDOS.S

Diese Programmteile sind Libraries, die zum Aufruf von BIOS-, XBIOS- und GEMDOS-Routinen dienen. Sie sind natürlich nicht unbedingt nötig und belegen eine Menge Speicherplatz, weil viele unbekannte Routinen mit

eingebunden werden. Sie können aber auch als Vorschlag für eigene Aufrufe dienen.

I.5 BOOT_2.0.PRG

Dieses Programm installiert im Boot-Sektor einer Diskette diverse Extras:

- Virus-Schutz
- Tastaturklick-Ausschaltung
- Fastloader
- Frequenzumschaltung

und einiges mehr.

I.6 BREAK.S

Diese Routine können Sie in Ihre Programme einbinden, um alle Registerinhalte abzuspeichern. Dann können Sie mit einem RESET das Programm verlassen und sich trotzdem alle Register ansehen. Näheres steht beim Befehl GETREGISTER des Debuggers (siehe Kapitel 6.6.5 auf Seite 141).

I.7 CALL.SRC und CALL.PRG

Mit CALL können Sie den OMIKRON.Debugger starten, wenn er resident im Speicher liegt. Dabei sucht das Programm nach den Texten „soft“ und „Σ-“ vor dem Busfehlervektor. Wenn der Text dort steht, holt der Debugger sich davor die Debuggeradresse und springt sie an. Sonst kommt eine Fehlermeldung (siehe Kapitel 6.12 auf Seite 174).

Da der Source mitgeliefert wird, können Sie diese Routine auch in Ihre Programme einbinden. Wenn Sie beispielsweise den VBL, den Keyboardtreiber u. ä. ändern, können Sie nicht mehr mit SHIFT-SHIFT abbrechen, auch wenn

Sie Ihr Programm vom Debugger aus starten. Dann können Sie z. B. beim Druck einer bestimmten Taste mit dieser Routine den Debugger aufrufen.

I.8 DAY_CALC.SRC

Dieses Programm berechnet aus dem Tagesdatum (XBIOS) den Wochentag und gibt ihn in D0 als Ziffer; in A0 einen Pointer auf den Wochentag als ASCII-Text zurück.

I.9 DESKTOP.PRG und DESKTOP.SRC

Das assemblierte DESKTOP.PRG sollte als letztes Programm im AUTO-Ordner stehen, denn es tut nichts anderes als das Desktop zu starten. Man kann damit allerdings einen eigenen Environment-String vor der Assemblierung zu definieren.

Dies erlaubt es z. B., eine Environment-Variable für den OMIKRONAssembler zu definieren (siehe Sourcetext):

```
'SIGMA=D:\OM_ASSEM.ALG\',0
```

Der OMIKRONAssembler (und auch der Debugger) sucht dann seine INF, CFG und DAT-Datei zuerst in diesem Directory.

Die Environment-Variable lässt sich auf dem ST (meines Wissens nach) nicht besser ändern. Die zweite Möglichkeit wäre ein Programm, welches vom Desktop aus zu starten ist und dann alle Zeiger auf den Environment-String seiner Parents auf einen eigenen zu verbiegen. Das Programm muß danach resident im Ram bleiben. Da ein Accessory keinen Parent hat, verbietet sich auch die Möglichkeit, den Environment-String über ein Accessory zu verändern.

Die erste Environment-Variable sollte *nicht* geändert werden, denn das Desktop sucht dort die RSC-Dateien (siehe auch Shel.Find).

I.10 EPSON.HEX

(siehe MAKE_CFG.PRG)

I.11 GAME.OMI

Dieser Ordner enthält das Spiel „Think and Work“ mit Sourcetext. Think & Work ST entspricht der Version für die Atari XL-Serie, welche in Ausgabe 2/89 der Happy Computer Listing des Monats war. Ich habe zwar keinen XL, aber mein 6502-Assemblerwissen und etwas logisches Verständnis ermöglichen es mir, die XL-Level unverändert auf den ST zu übernehmen. Das Spiel ist sonst jedoch völlig unabhängig von der XL-Version in zwei Tagen programmiert worden.

Das Spiel enthält folgende Features:

- lauffähig auf Monochrom- und Farbmonitoren
- eingebauter Level-Editor
- Demo-Modus
- Joystick- oder Tastatursteuerung (Tastatur geht besser)
- maximal 99 Level (25 sind dabei)
- Highscore von jedem Level (speicherbar)
- ist etwa so lang wie die XL-Version (also ziemlich kurz)

Der Sinn des Spiels): Das Männchen muß die Blöcke auf die als Rauten markierten Zielfelder schieben. Es kann nur ein Block auf einmal verschoben werden. Wenn sich alle Blöcke auf den Zielfeldern befinden, bringt einen der Exit in den nächsten Level (Demo beseitigt wohl die Unklarheiten, zur Not hilft ein Blick in die Happy Computer weiter) Jeder Block bringt beim Start 100 Punkte, jede Verschiebung kostet 5 Punkte, jede Sekunde Zeit einen Punkt.

Einzelheiten können dem Sourcetext entnommen werden.

Wichtig: Für dieses Programm brauchen Sie eine neue OM-ASSEM.DAT-Datei. Assemblieren Sie dazu OM_MODUL.SRC — bewahren Sie sich aber die alte Datei auf; wenn Sie nicht mit dem Spiel arbeiten kostet die neue nur sinnlos Zeit und Speicher. Verlassen Sie den Assembler und starten Sie ihn neu, damit

die DM-ASSEM.DAT-Datei neu geladen wird. Nach dem Assemblieren sehen Sie nun zwei neue Buttons: „RUN GAME“ und „SAVEGAME“. Diese binden die Grafik automatisch ins Programm ein (Die beiden Bilder werden beim Start des Assemblers geladen). Durch diesen Trick werden die Turn-Around-Zeiten nochmals drastisch verkürzt, trotzdem kann man jederzeit z. B. die Bilder ändern, den Assembler neu laden und sich die Änderungen sofort ansehen, alle Konvertierungen übernimmt das Modul.

Falls jemand dem Programm einen Sound oder neue Level verpaßt hat, wäre ich dankbar, wenn ich eine solche Version bekommen könnte.

I.12 GEMDOS.S

(siehe BIOS.S)

I.13 GEM_VARS.S und GEM_VARS.SYM

Diese Symboltabelle enthält Konstanten, die zu GEM-Aufrufen benutzt werden können. Es sind alle Funktionen implementiert — also auch die vom TOS 1.4 und vom GEM 2.0. Diese Funktionskonstanten können z. B. der im AES_DEMO enthaltenen AES-Routine übergeben werden. Die Konstante setzt sich aus der Funktionsnummer und der Anzahl der Einträge im int.in, int.out und addr.in zusammen.

In der Konstantensammlung sind auch die Message-Nummern, Objekt-Status, Mausformen u. v. a. m. enthalten.

I.14 HANOI.SRC

Das „Türme-von-Hanoi-Problem“ ist Ihnen sicherlich bekannt: Es müssen Scheiben unterschiedlicher Größe von einem Stapel auf einen anderen umgepakt werden, wobei nur eine Zwischenablage vorhanden ist, und größere Scheiben nicht auf kleinere gelegt werden dürfen. Dieses Programm löst dieses Problem nun ziemlich schnell — Die Positionen werden jedoch in der jetzigen Programmversion nicht ausgegeben.

I.15 HEADER.S

Header ist ein Programmskelett, das Sie vor Ihre Programme setzen können.

Wenn ein Programm gestartet wird, übergibt das Betriebssystem ihm den ganzen zu Verfügung stehenden Speicher. Wenn dann noch ein anderes Programm Speicherplatz braucht (z.B. ein Accessory), und sich welchen mit Malloc anfordern möchte, ist natürlich keiner mehr da. Deshalb sollten Sie allen Speicher, den Sie nicht mehr (oder im Moment nicht) benötigen, mit Mshrink freigeben. Dieser Header erledigt nun genau dieses. Er berechnet, wie lang das Programm ist, reserviert noch ein wenig Platz für den Stack und gibt den Rest frei. Dann kann Ihr Programm folgen. Wenn Sie später wieder mehr Platz benötigen sollten, können Sie diesen ja wieder mit Malloc anfordern. Am Ende steht ein Pterm, um das Programm zu beenden.

I.16 L_XXXXXX.SRC und L_OC0000.PRG

Mit diesem Programm können Sie den Debugger an eine beliebige Adresse laden. Eine Erklärung finden Sie im Kapitel G auf Seite 205.

I.17 MAKE_CFG.SRC, EPSON.HEX und NEC.HEX

Die Druckeranpassungsdateien haben die Extension CFG. Sie entsprechen dem 1st-Word-Format; wenn Sie also schon 1st-Word auf Ihren Drucker angepaßt haben, brauchen Sie die CFG-Datei nur als OM-ASSEM.CFG in das gleiche Inhaltsverzeichnis wie den Assembler zu kopieren.

Da CFG-Dateien schwierig zu ändern sind, gibt es eine ASCII-Klartext-Datei, in der alle benutzten Steuerzeichen mit Kommentaren abgelegt sind. Das Programm MAKE_CFG.PRG konvertiert nun die Klartext-Datei (HEX) in eine CFG-Datei. Der Aufbau einer .HEX- und einer CFG-Datei ist im Kapitel D.5 auf Seite 194 beschrieben.

EPSON.HEX sind die Voreinstellungen für einen EPSON FX-80 und kompatible, NEC.HEX für einen NEC-P6 und kompatible Drucker.

I.18 NEC.HEX

(siehe MAKE_CFG.PRG)

I.19 OM_MODUL.SRC

Dieses ist der Source für die Datei OM-ASSEM.DAT. Wie und wozu man so etwas braucht, steht im Anhang H auf Seite 206

Achtung: Dieser Source enthält noch weitere Module als die auf ihrer Originaldiskette liegende OM-ASSEM.DAT-Datei. Diese werden für das Spiel „Think and Work“ benötigt (siehe GAME.OMI). Normalerweise kosten die Module nur unnötig Speicherplatz und Zeit — assemblieren Sie diese Datei also nur bevor Sie mit dem Spiel experimentieren wollen oder im Zusammenhang mit dem Kapitel H auf Seite 206.

I.20 SCRNDUMP.SRC und SCRNDUMP.PRG

Nachdem Sie dieses Programm gestartet haben, können Sie mit den Tasten ALTERNATE-HELP nicht mehr den Bildschirminhalt ausdrucken, sondern im DEGAS-Format abspeichern. Dabei wird auch die Farbpalette ausgelesen und abgespeichert. SCRNRERR.SRC ist das gleiche Programm, beinhaltet aber einen Programmierfehler. Dieser dient als Debugger-Demo und wird im Kapitel 6.13 behoben.

I.21 SCRNRERR.SRC

(siehe SCRNDUMP.SRC)

I.22 Σ-RSC

Dieser Ordner enthält einen RSC-Editor für das Σ-soft-RSC-Format. Die damit erstellten RSC-Strukturen können Sie in Module einbinden — näheres

finden Sie in Kapitel H.7 auf Seite 214.

I.23 SHIP.ACC, SHIP.PRG und SHIP.SRC

Dieses Accessory parkt die Festplatte mit der Nummer 0 (normalerweise hat jede Festplatte diese Nummer, sie dient nur zur Unterscheidung, wenn mehrere Festplatten u. ä. angeschlossen sind.). Das gleiche können Sie auch mit dem beigelegten HDX-Programm erreichen; da das Accessory jedoch immer griffbereit ist, kann man nun die Festplatte vor jedem Ausschalten problemlos parken.

SHIP.PRG ist das gleiche als Programm. Das Interessante, weshalb wir es auch als Demo zum Assembler gepackt haben, ist, daß beide Programme, SHIP.ACC und SHIP.PRG identisch sind. Das Programm erkennt automatisch, ob es als Programm oder als Accessory gestartet worden ist.

Das geht so: Wenn ein Programm gestartet wird, legt das Betriebssystem davor eine Basepage an. Bei einem Accessory ist nun kein Parent eingetragen (das Programm, daß dieses Programm gestartet hat). Also muß man nur in der Basepage nachgucken. Sie können diesen Programmheader natürlich für ihre eigenen Accessories verwenden.

I.24 SYS_VARS.S und SYS_VARS.SYM

Dies ist eine Zusammenstellung aller wichtigen und dokumentierten Systemvariablen. Für weitere Erläuterungen müssen wir Sie natürlich wieder auf entsprechende Fachbücher verweisen. Da diese Datei nur aus Konstantendefinitionen besteht, eignet Sie sich zum Erstellen einer Symboltabelle (SYM).

I.25 TAKT_TAB.DOC

In diesen Tabellen haben wir die Taktzyklen für die meisten Befehle des 68 000 zusammengestellt. Wir haben Sie selbst gemessen — deshalb fehlen auch die Befehle, die etwas schwerer zu messen sind.

I.26 TATÜ.ACC und TATÜ.SRC

Dieses nützliche Accessory erkennt, ob sich in Ihrer Softwaresammlung eine Raubkopie befindet und reagiert entsprechend (es geht nix kaputt, keine Angst).

I.27 Think and Work

(siehe GAME.OMI)

I.28 TOS_LOAD.SRC und TOS_LOAD.PRG

Dieses Programm lädt, in den AUTO-Ordner kopiert, ein Ram-TOS (TOS.IMG) von der Festplatte (Hauptinhaltsverzeichnis von C) — beispielsweise das TOS 1.4.

I.29 VDISK3.ACC und VDISK3.SRC

VDISK ist eine RAM-Disk. VDISK3.DOC ist die Anleitung dazu. Deshalb möchte ich hier auf längere Erklärungen verzichten und Sie auf das DOC-File verweisen. Das ACC-File ist, wie die vorigen Versionen der VDISK, Public Domain, das dürfen Sie also weitergeben. Der Source-Text unterliegt jedoch dem Copyright von Σ-soft — er dient nur als Demo zum Assembler und darf nicht kopiert werden.

I.30 WPROT.SRC

Dieses Programm simuliert einen Schreibschutz für beliebige Laufwerke, also auch für Festplatten. Am Ende des Sourcetextes können Sie die Buchstaben der Laufwerke eingeben, die geschützt werden sollen. Alle Schreibzugriffe werden nun abgefangen und die Dialogbox „Diskette in Laufwerk x: schreibgeschützt“ ausgegeben.

I.31 XBIOS.S

(siehe BIOS.S)

I.32 XBRA.SRC und XBRA.TXT

Dieses ist eine Norm für vektorverbiegende Programme. Es erlaubt Ihnen, Vektoren zu ändern zurückzusetzen und zu testen, ob ein Vektor bereits verbogen worden ist. Eine genauere Erläuterung steht im File XBRA.TXT.

Anhang J

Literaturhinweise

Wir können in diesem Handbuch natürlich nur eine kurze Übersicht über die wichtigsten Teile des Betriebssystems geben. Für nähere Informationen können wir Ihnen folgende Bücher empfehlen:

- Hans-Dieter Jankowski, Dietmar Rabich, Julian F. Reschke:
Atari ST Profibuch
SYBEX-Verlag GmbH, Düsseldorf — ISBN 3-88745-563-0
Dieses Buch enthält Informationen zum Betriebssystem und zur Hardware (auch zum Mega ST), ist übersichtlich gegliedert (mit sehr umfassenden Register) und fast fehlerfrei. Dieses Buch ist ein Standardwerk für praktisch alle Programmierer auf dem ST.
- C. Vieillefond: Programmierung des 68 000
SYBEX-Verlag GmbH, Düsseldorf — ISBN 3-88745-060-4
Es ist zwar nicht speziell für den ST geschrieben, aber enthält doch alle wichtigen Informationen für den ST-Benutzer.
- Frank Mathy: Programmierung von Grafik & Sound auf dem Atari ST
Markt & Technik Verlag, Haar bei München — ISBN 3-89090-405-X
- Motorola: MC 68000 16/32-Bit Microprocessor User's Manual
- Motorola: MC 68 020 32-Bit Microprocessor User's Manual
Motorola Inc. — ISBN 0-13-566878-6 or
Prentice Hall, Inc. — ISBN 0-13-566860-3

- Alfons Kramer, Thomas Riebls, Winfried Hübner: Das TOS-Listing
Verlag Heinz Heise GmbH & Co KG, Hannover — ISBN 3-88229-002-1
- Claus Brod, Anton Stepper: Scheibenkleister
Maxon Computer GmbH, Eschborn — ISBN 3-927065-00-5
- Doctor Dobb's Toolbook of 68 000 Programming
Prentice Hall Press — ISBN 0-13-216649-6 (case); 0-13-216557-0 (pa-
perback)

Index

Kursiv geschriebene Zahlen verweisen auf die Debuggeranleitung.

/ 154
“ 147
@ 163
! 154
= 88
, 157
] 157
) 158
== 88
? 160
* 35

Abhangigkeit vom Betriebssystem
168

Absolut 58

Accessories 48

Adressregister 109

△ 124

ALIGN.W 84

Allgemeine Befehle 158

Allgemeine Tastenkommandos 115

Allquantor 67,159

ALT-CLR/HOME 112

ALT-INSERT 112

ALT-←, → 111

ALT-M 111

ALT-Zahl 113

Anfanger 108

Anpassen (Button) 34
Anwendung anmelden 51
Ascfind 159
Ascii 158
ASCII 30,32,184
ASCII-Anzeige 81
ASCII-Sichern 53
ASCII-Sourcetext 49,50
ASCII-Sourcetext
einfugen 51
ASCII-Text suchen 69
Assembler ↔ Debugger 104
Assembler
Speicherverwaltung 64
Assemblieren 55,56
Ausdruck berechnen 160
Ausgabeurnlenkung 146,146,147
Ausgeben einer Linie 147
Ausgeben von Cr 147
Autoinsert 119
BAK-Datei 77
BASE 92
BASE DC 93
Basepage 184,190
BASIC 14,59
Batch-Befehl 163
Bedingte Assemblierung 97
Befehl abkurzen 45
Befehl berspringen 119

- Befehlseingabe 124
 Befehlssyntax 82, 124
 Begriffserklärungen 184
 Benutzeroberfläche 20
 Betriebssystemroutinen 124
 Betriebssystemstrukturen 189
 Bildschirm speichern 116
 Bildschirmabschalter 76
 Bildschirmeditor 110, 115
 Bildschirmfarben 76
 Bildschirmseite umschalten 122
 binär 184
 Blitter 22
 BLK.x 82
 Block 73
 Block
 - ausdrucken 73
 - copy 68
 Blockmarkierung 28
 Block
 - markierungen 73
 - speichern 73
 Boolesche Algebra 34
 Boot-Sektor 191
 Breakpoint 132, 184
 Breakpoints anzeigen 121
 Breakpointtaste 112, 116
 BSS 43, 87
 Bssclear 142
 Button 20
 Cache 110, 111, 116, 135, 140
 Cacheclr 140
 CACHECLR 112
 CACHEGET 112
 Cacheget 140
 Call 134
 CALL.PRG 108, 175, 220
- CAPS LOCK 30, 113, 116
 CCR-Register 109
 Centronics-Schnittstelle 66
 CFG-Datei 26
 CFG-Format 66
 Checksumme 162
 Clr 161
 Cls 141
 CNOP 95
 Commandline 51, 90
 Compare 162
 Conditions 32
 Continue 159
 CONTROL/ALT 1--9 111
 CONTROL-B 112
 CONTROL-G 112
 CONTROL-HELP 113
 CONTROL-M 111
 CONTROL-P 112
 CONTROL beim Laden 175
 Copy 160
 Cr 147
 Cursor 21, 110, 167, 185
 Cursor setzen 110
 Cursor umdefinieren 167
 Cursorform 75
 Cursorposition 29, 52, 105
 Cursorpositionierung 28, 42
 Cursorsteuerung 115
 Darstellung 80
 Dartellung 73
 DATA 86
 Datas 58
 DATA-Segment 43
 ^`DATE 35
 Datei lesen 150
 Datei löschen 54, 148

- Datei umbennen 149
Dateiattribut ändern 149
Dateiattribute 150
Dateiausgabe 146,147
Datenregister 109
DAT-Datei 26
DCB.x 83
DC.x 82
Debuggen 185
Debugger
 beenden 116,123,163,174
 Sprung in den 60
Debugger-Demo 175
Debugger
 löschen 61
 nachladen 60
 resident 104
 residenter 26
Debuggervariablen 125
DEFAULT 97
Default 185
DEGAS 44,116
DESKTOP.PRG 22,221
Devpac 144
dezimal 185
Dialogboxen 20
Digital Research 57,60
Direct 122
Directory 120,145
Disassembly 110,153
Disassembly PC 121
Disk formatieren 150
Diskettenoperationen 48
Do 134
Do PC 117
Doppelklick 28,110,185
Doppelpunkt 31,60
- Doppelte Deklaration 101
DRI-Linker-Format 60
Drop-Down-Menüs 28
Drucken 65
Drucker 78,146
Druckeranpassung 66
Druckertreiber 66
DSBSS.x 84
DS.x 82
Dump 110,156
DX.B 84
DXSET 84
Editmodus 43
Editor 30
Editor 1 75
Editor 2 77
Editor-Menü 80
Einfügemodus 30
Einfügen I/II 30
Einführung 25
Eingabe von Befehlen 124
Einstellungen 52,73
Einstellungen sichern 168
Einstellungen
 sichern 80
ELSE 98
END 88
Endadresse 125
ENDC 98
Ende 54
ENDIF 98
ENDR 96
Environment-Var. 22,26,90,221
EQU 38,62,88
Erase 148
Ersetzen 67
EVEN 84

- Existenquantor 159
Existensquantor 67
Exit 123,163
Extended-File-Selector 50
Extension 78
Externer Abbruch 170
FAIL 98
Farbe im Debuggers 128
FATTRIBUT 26
Fattribut 149
Fclose 147
Fehler bei der Assemblierung 55
Fehler
 allgemeine 55
 fatale 55
fehlerhafte Zeile 77
Fehler
 korrigierbare 55
Fehlermeldung 29
Fehler
 -meldungen 56
Fehlermeldungen 171
Fehler
 nächster 72
 Warnungen 56
 Zeile mit 77
File 146
Fileattribute 150
Fileausgabe 146,147
Filename 29
File-Selector 49,50
File-Selector
 Extended 50
Fill 161
Find 158
Flags 109
Fopen 146
Format 150
Formel 34
Formelauswertung 190
Free 148
freier Speicher 148
F-Tasten belegen 80
Funktionscode 174
Funktionstaste 185
Funktionstasten 80,117
Funktionstasten belegen 164
Funktionstasten
 abspeichern 80
F1 117
F10 122
F2 117
F3 118
F4 119
F5 119
F6 120
F7 120
F8 121
F9 122
Garbage-Collection 44
GEM 20
Gemdos 20
Gemdos-Speicher 78
Geschichte 23
Getregister 141
GFA-Assembler 144
GLOBAL 60,95
Go 133
Go PC 119
Grundrechenarten 34
Grundversion 16
GST 57
GST-Assembler 144
GST-Linkformat 144

- Hallo 53
Harddiskoperationen 48
HEADER.S 224
Help 167
hex 186
Hintergrundfarbe 76
Hunt 159
IBYTES 91
IF 99
If 136
IFD 98
IFEQ 97
IFGE 98
IFGT 98
IFLE 97
IFLT 97
IFND 98
IF[NE] 97
ILLEGAL 97
Include-Datei 181
info 144
Info 121,144,160
Informationen 64
INF-Datei 26
Initregister 142
Inline 59
Insert 119
Installation 19
Internes 167
internes Problem 182
I/O-Befehle 143
Joystickabfrage 127
Kaltstart 45,116
keine Ahnung 122
Key 164
Kill 148
klicken 186
Kommentare 39
Kompaktor 78
Kompatibilität 82
Komplement 34
Konstanten 37,38,88
Kontrollfeld 170
Konvertierung 32
.L 157
Label 37,39
Labelbase 144
Laden 49
Let 164
Lexecute 143
Line 147
Linker 16,60,95,181
List 110,154
List PC 122
Listende Befehle 153
Literaturhinweise 229
Load 144
logische Sektoren lesen 151
lokale Label 32
Longword 186
L_xxxxxx.PRG 205
Makro 16
Makro-Version 16,181
Marker 43,105
Marker anzeigen 120
Marko 181
Maus anschalten 141
Maus ausschalten 141
Mausbeschleunigung 78
Mausscrolling 166
Meldungen des Debuggers 171
Memory 110,156
Memorydump PC 120
Memoryinfo 121

Menüleiste 45,117
Merker, nächster 72
Metacomo Macro Assembler 144
MISHELL 22
Mkdiritory 148
Modul
 Absolut 58
 BASIC 59
 Datas 58
 debugger 58
 Inline 59
 Parallel 59
 SmallDRI 60
 Standard 58
Monitorumschalter 122,166
Motorola-Standard 16,32
Mouseoff 141
Mouseon 141
Move 160
Name 149
Neu anlegen 49
nicht tokenisierbare Zeilen 51,76
Observe 138
Offset 186
QM-ASSEM.CFG 26
QM-ASSEM.DAT 26
QM-ASSEM.INF 26,80
QM-DEBUG.INF 26,145,168
OMIKRONAssembler 144
OMIKRON.BASIC 58,202
Opcodes 30
Operanden 30
OPT 60,89,98
Optimierungen 34,100
OPT 37
Ordner erstellen 148
Ordner löschen 149
ORG 96
OUTPUT 90
Overwrite 119
Password 49,53
PATH 91
PC 109,186
PC setzen 112,116
PC-kompatibel 113
Permanent-Breakpoint 118
Pfad 186
Positionsmarkierung 43
Printer 146
Prn 146
Problem, ein kleines internes 182
Problem
 mit älterem TOS 198
Programm starten 112,119,133
Programmabbruch 112,116,170
Programmende 199
Programmheader 189
Pseudo-Opcode 16
Pseudoopcode 186
Pseudo-Opcodes 81
Quantoren 67,69
Quellcode 104
Quickmaus 78
Quit 123,163
README 150
Readsector 151
Readtrack 152
Rechenoperation 36
Rechenoperationen 34
Rechenzeichen 191
Rechnen mit Symbolen 38
Rechner 62,63
rechte Maustaste 110
REG 89

- Register 109
Register ändern 110
relozierbare Symbole 39
relozieren 187,192
Relozierinformationen 192
Remark 39
Remark
 durchsuchen 70
 Merker 72
 Tabulator 75
reorganisieren 60,65
REPT 96
RESET 20,116,171
Reset 45
Reset all 165
Reset Vektor 165
resident 26,61,104
Resident 107,163,174
„Ring Indicator“-Abbruch 171
Rmdiritory 149
RSBSS 86
RSC-Editor 214
^^RSCOUNT 37
RSEVEN 86
RSRESET 86
RSSET 86
RS.x 85
Rwabs 151
Rwflag 151
Save 145
Save ! 145,164,168
Schnittstelle 58,104
Schreibweise der Befehle 124
Screeneditor 110,115
Scrollen 115,153,187
scrollen 111
Scrolling 29
Scrolloff 166
Scrollon 166
Segment
 BSS 43
 DATA 43
Segmentierung 86
Segment
 TEXT 43
Seriennummernschutz 53
Set 164
SET 38,89
SFEhler, nächster 72
SHIFT-F1 117
SHIFT-F10 123
SHIFT-F2 118
SHIFT-F3 119
SHIFT-F4 119
SHIFT-F5 119
SHIFT-F6 120
SHIFT-F7 121
SHIFT-F8 121
SHIFT-F9 122
SHIFT-SHIFT 112,171
Showmemory 134
Sicherheitskopie 77
Sichern 52
Sichern unter 52
Skip PC 119
SmallDRI 60
Södler, Artur 24
Source 187
Sourcetextformat 52
Sourcode-Debugging 181
Space 187
Speicher löschen 161
Speicher
 -aufteilung 80

- Gemdos- 78
- Speicherplatz 64,65
- Sprung zu Zeile 71
- Sprünge im Sourcetext 42
- SR 109
- SRC-Format 49
- SSP 109
- Starten des DBG 107
- Statistik 64
- Statusregister 109
- Statuszeile 29
- Steuerzeichen 45
- stöhn 24
- Suchen 66
- Suchfunktionen 76
- Suchkommandos 42
- Suchzeichen 77
- Supervisor-Flag 110
- Supervisor-Stack-Pointer 109
- Switch 166
- SWITCH 99
- Symbol 37,60
 - definition 28
 - ersetzen 68
 - laden 51
 - namen 77
 - relozierbar 39
 - suchen 67
 - typen 155
- Symboltabelle
 - 37,51,52,56,65,187,197
- Symboltable 111,155
- ^^SYMTAB 37,98
- Sync 143
- System 123,163
- Systemvariablen 35,37
- | 142
- Tabulator 40,75
- Tabulator
 - Befehls- 75
 - Operanden- 75
 - Pseudoopcode- 75
 - Pseudo-Rem- 75
 - Remark- 75
- Taschenrechner 62,63
- Tastaturbelegung 111
- Tastaturklick 77
- Tastaturkommandos 40
- Teach-In-Tasten 181
- TEXT 86
- Text ersetzen 70
- Text suchen 69
- TEXT-Segment 43
- Think and Work 222
- ^^TIME 35,82
- Tips 170
- Toggle Screen 122
- tokenisieren 50
- tokenisierender Editor 15
- Trace 134
- trace 117
- Trace no subroutines 118
- Trace Traps 119
- Trace until RTS 118,119
- Trace-Funktionen 132
- Tracesimulator 68 020 117
- Trap tracen 119
- Trial-and-Error 14
- Turbo-C 144
- Type 150
- Überschreiben 30
- Uhr 80,81
- UNDO 111
- Unterprogramm verlassen 118

- Unterprogrammaufruf 118
Untrace 135
User-Stack-Pointer 109
USP 109
V:* 126
V:ACT_PD 127
V:AESFLAG 128
Variablen des Debuggers 125
V:^A(0-7) 127
V:BASEPAGE 126
V:BP 126
V:BSS 125
V:^B(Term) 126
V:BUFFER 126
V:CCR 126
V:CHECKSUM 128
V:COLORS 128
V:COL0 128
V:COL1 128
V:CONTERM 128
V:CONVERT 127
V:DATA 125
V:DISBASE 126
VDISK 227
V:DRIVE 126
V:^D(0-7) 127
V:END 125
Vergleiche 36
verketten 41
V:IKBD 127
V:KLICK 127
V:LINES 129
V:MEMBASE 129
V:^M(0-9) 126
Vordergrundfarbe 76
Vorzeichen 130
V:PC 126
V:REZ 128
V:SAVEAREA 129
V:SCROLLD 127
V:SEKBUFF 126
V:SEKTOR 126
V:SHIFT 127
V:SIDE 126
V:SIZE 129
V:SMALL 128
V:SP 126
V:SR 126
V:SSP 126
V:START 125
V:SYMBOLS 126
V:TDELAY 127
V:TEXT 125
V:TRACE 128
V:TRACK 126
V:TRKBUFF 126
V:USP 126
V:UT 128
V:UTRACE 128
.W 157
Warmstart 45,116
Warnungen 56,76,100
Word 188
Writesector 152
Zahlenbasis 35,43
Zahlensysteme 130
Zehnerblock 113,116
Zeichentabelle 63
Zielcodespeicher 57
Zuladen 51
Zusätzliche Tastenkommandos 116
68 020 15,117
- 164