Alfred Arnold

Makroassembler AS V1.42

Benutzeranleitung

Stand September 2026

CompactRISC ist ein eingetragenes Warenzeichen von National Semiconductor (heute ein Teil von Texas Instruments).

IBM, PPC403Gx, OS/2 und *PowerPC* sind eingetragene Warenzeichen der IBM Corporation.

Intel, MCS-48, MCS-51, MCS-251, MCS-96, MCS-196 und *MCS-296* sind eingetragene Warenzeichen der Intel Corp. .

Motorola und *ColdFire* sind eingetragene Warenzeichen von Motorola Inc. .

MagniV ist ein eingetragenes Warenzeichen von Freescale Semiconductor.

PicoBlaze ist ein eingetragenes Warenzeichen der Xilinx Inc.

eZ80 und *Z80* sind eingetragene Warenzeichen von Zilog Inc.

UNIX ist ein eingetragenes Warenzeichen der Open Group.

Linux ist ein eingetragenes Warenzeichen von Linus Torvalds.

Microsoft, Windows und *MS-DOS* sind eingetragene Warenzeichen der Microsoft Corporation.

Alle anderen Warenzeichen, die nicht ausdrücklich in diesem Abschnitt genannt wurden und in diesem Handbuch verwendet werden, sind Eigentum der entsprechenden Eigentümer.

Dieses Dokument wurde mit dem LaTeX-Satzsystem unter dem Betriebssystem

Linux angefertigt und formatiert.

Inhaltsverzeichnis

1 Allgemeines **13**
1.1 Lizenzbedingungen 13
1.2 allgemeine Fähigkeiten des Assemblers 15
1.3 Unterstützte Plattformen 21

2 Benutzung des Assemblers **23**
2.1 Hardware-Anforderungen 23
2.2 Lieferumfang 24
2.3 Installation 29
2.4 Aufruf, Parameter 30
2.5 Format der Eingabedateien 41
2.6 Format des Listings 44
2.7 Symbolkonventionen 48
2.8 Temporäre Symbole 51
2.8.1 Temporäre Symbole mit Namen 52
2.8.2 Zusammengesetzte temporäre Symbole 54
2.9 Formelausdrücke 55
2.9.1 Integer-Konstanten 55
2.9.2 Gleitkommakonstanten 57
2.9.3 Stringkonstanten 58
2.9.4 String- zu Integerwandlung und Zeichenkonstanten 59
2.9.5 Evaluierung 60
2.9.6 Operatoren 60
2.9.7 Funktionen 62
2.10 Vorwärtsreferenzen und andere Desaster 65
2.11 Registersymbole 69
2.12 Sharefile 70
2.13 Prozessor-Aliase 70

3 Pseudobefehle 73
3.1 Definitionen . . . 73
3.1.1 SET, EQU und CONSTANT . . . 73
3.1.2 SFR und SFRB . . . 75
3.1.3 XSFR und YSFR . . . 75
3.1.4 LABEL . . . 76
3.1.5 BIT . . . 76
3.1.6 DBIT . . . 77
3.1.7 DEFBIT und DEFBITB . . . 78
3.1.8 DEFBITFIELD . . . 79
3.1.9 PORT . . . 80
3.1.10 REG und NAMEREG . . . 81
3.1.11 LIV und RIV . . . 81
3.1.12 CHARSET . . . 82
3.1.13 CODEPAGE . . . 84
3.1.14 ENUM, NEXTENUM und ENUMCONF . . . 84
3.1.15 PUSHV und POPV . . . 85
3.2 Codebeeinflussung . . . 86
3.2.1 ORG . . . 86
3.2.2 RORG . . . 93
3.2.3 CPU . . . 93
3.2.4 SUPMODE, FPU, PMMU, CUSTOM . . . 118
3.2.5 ACCMODE bzw. EXECMODE . . . 119
3.2.6 CIS, EIS, FIS und FP11 . . . 119
3.2.7 FULLPMMU . . . 120
3.2.8 COPROC . . . 120
3.2.9 PADDING . . . 120
3.2.10 PACKING . . . 122
3.2.11 MAXMODE . . . 122
3.2.12 EXTMODE und LWORDMODE . . . 123
3.2.13 SRCMODE . . . 123
3.2.14 EXTRACOMMENTS . . . 123
3.2.15 PLAINBASE . . . 124
3.2.16 BIGENDIAN . . . 124
3.2.17 WRAPMODE . . . 125
3.2.18 PANEL . . . 125
3.2.19 WARNRELATIVE . . . 126
3.2.20 SEGMENT . . . 126
3.2.21 PHASE und DEPHASE . . . 127
3.2.22 SAVE und RESTORE . . . 128
3.2.23 ASSUME . . . 129

3.2.24 CKPT . . . 142
3.2.25 EMULATED . . . 142
3.2.26 Z80SYNTAX . . . 143
3.2.27 EXPECT und ENDEXPECT . . . 144
3.3 Datendefinitionen . . . 144
3.3.1 DC[.size] . . . 144
3.3.2 DS[.size] . . . 145
3.3.3 BLKB, BLKW, BLKL, BLKD . . . 146
3.3.4 DN,DB,DW,DD,DQ,DT,DO & BF16 . . . 146
3.3.5 D1 . . . 149
3.3.6 DC24 . . . 150
3.3.7 FLT . . . 150
3.3.8 FLT2, FLT3, FLT4 . . . 150
3.3.9 x_FLOATING . . . 150
3.3.10 DS, DS8, DS16, DS32, DS64 . . . 151
3.3.11 BLKx . . . 151
3.3.12 BYT oder FCB . . . 151
3.3.13 BYTE . . . 152
3.3.14 DC8 . . . 152
3.3.15 ADR oder FDB . . . 152
3.3.16 DDB . . . 152
3.3.17 DCM . . . 153
3.3.18 WORD . . . 153
3.3.19 DW16 . . . 153
3.3.20 ACON . . . 153
3.3.21 LONG . . . 153
3.3.22 SINGLE, DOUBLE und EXTENDED . . . 154
3.3.23 FLOAT und DOUBLE . . . 154
3.3.24 SINGLE und DOUBLE . . . 154
3.3.25 EFLOAT, BFLOAT, TFLOAT . . . 154
3.3.26 Qxx und LQxx . . . 155
3.3.27 FIELD . . . 155
3.3.28 DATA . . . 156
3.3.29 ZERO . . . 156
3.3.30 FB und FW . . . 156
3.3.31 ASCII, ASCIC und ASCIZ . . . 156
3.3.32 STRING und RSTRING . . . 157
3.3.33 PACKED . . . 157
3.3.34 RADIX50 . . . 157
3.3.35 FCC . . . 158
3.3.36 TEXT . . . 158

3.3.37 DFS oder RMB . . . 158
3.3.38 BLOCK . . . 158
3.3.39 SPACE . . . 158
3.3.40 RES . . . 159
3.3.41 BSS . . . 159
3.3.42 DSB und DSW . . . 159
3.3.43 DS16 . . . 159
3.3.44 ALIGN . . . 160
3.3.45 EVEN . . . 160
3.3.46 LTORG . . . 160
3.4 Makrobefehle . . . 161
3.4.1 MACRO . . . 161
3.4.2 IRP . . . 169
3.4.3 IRPC . . . 170
3.4.4 REPT . . . 170
3.4.5 WHILE . . . 171
3.4.6 EXITM . . . 171
3.4.7 SHIFT . . . 172
3.4.8 MAXNEST . . . 173
3.4.9 FUNCTION . . . 173
3.5 Strukturen . . . 174
3.5.1 Definition . . . 175
3.5.2 Nutzung . . . 176
3.5.3 geschachtelte Strukturen . . . 177
3.5.4 Unions . . . 177
3.5.5 Namenlose Strukturen . . . 177
3.5.6 Strukturen und Sektionen . . . 178
3.5.7 Strukturen und Makros . . . 178
3.6 bedingte Assemblierung . . . 178
3.6.1 IF / ELSEIF / ENDIF . . . 179
3.6.2 SWITCH / CASE / ELSECASE / ENDCASE . . . 180
3.7 Listing-Steuerung . . . 182
3.7.1 PAGE, PAGESIZE . . . 182
3.7.2 NEWPAGE . . . 183
3.7.3 MACEXP_DFT und MACEXP_OVR . . . 183
3.7.4 LISTING . . . 185
3.7.5 PRTINIT und PRTEXIT . . . 185
3.7.6 TITLE . . . 186
3.7.7 RADIX . . . 186
3.7.8 OUTRADIX . . . 187
3.8 lokale Symbole . . . 187

3.8.1 Grunddefinition (SECTION/ENDSECTION) 188
3.8.2 Verschachtelung und Sichtbarkeitsregeln 189
3.8.3 PUBLIC und GLOBAL 191
3.8.4 FORWARD 193
3.8.5 Geschwindigkeitsaspekte 194
3.9 Diverses 194
3.9.1 SHARED 194
3.9.2 INCLUDE 195
3.9.3 BINCLUDE 196
3.9.4 MESSAGE, WARNING, ERROR und FATAL 196
3.9.5 READ 197
3.9.6 INTSYNTAX 198
3.9.7 RELAXED 199
3.9.8 COMPMODE 199
3.9.9 END 200

4 Prozessorspezifische Hinweise 201
4.1 6811 201
4.2 PowerPC 203
4.3 PALM 203
4.4 DSP56xxx 205
4.5 H8/300 205
4.6 H8/500 206
4.7 SH 206
4.8 HMCS400 209
4.9 H16 210
4.10 OLMS-40 212
4.11 OLMS-50 213
4.12 MELPS-4500 213
4.13 6502UNDOC 214
4.14 MELPS-740 218
4.15 MELPS-7700/65816 219
4.16 M16 222
4.17 CP-3F 223
4.18 4004/4040 225
4.19 MCS-48 225
4.20 MCS-51 226
4.21 MCS-251 227
4.22 8080/8085 230
4.23 8085UNDOC 231
4.24 8086..V35 233

4.25 8X30x . . . 235
4.26 XA . . . 236
4.27 AVR . . . 237
4.28 Z80, Z18, Z280, Z380 . . . 238
4.29 Z80UNDOC . . . 239
4.30 GB_Z80 bzw. LR35902 . . . 240
4.31 Z380 . . . 240
4.32 Z8, Super8 und eZ8 . . . 242
4.33 Z8000 . . . 243
4.33.1 Bedingungen . . . 244
4.33.2 Flags . . . 244
4.33.3 Indirekte Adressierung . . . 244
4.33.4 Direkte versus unmittelbare Adressierung . . . 244
4.34 TLCS-900(L) . . . 245
4.35 TLCS-90 . . . 248
4.36 TLCS-870 . . . 249
4.37 TLCS-47 . . . 249
4.38 TLCS-9000 . . . 250
4.39 TC9331 . . . 251
4.40 29xxx . . . 251
4.41 80C16x . . . 252
4.42 PIC16C5x/16C8x . . . 254
4.43 PIC17C4x . . . 255
4.44 SX20/28 . . . 255
4.45 ST6 . . . 255
4.46 ST7 . . . 256
4.47 ST9 . . . 257
4.48 6804 . . . 257
4.49 TMS3201x . . . 258
4.50 TMS320C2x . . . 258
4.51 TMS320C3x/C4x . . . 259
4.52 TMS9900 . . . 260
4.53 TMS70Cxx . . . 260
4.54 TMS370xxx . . . 261
4.55 MSP430(X) . . . 262
4.56 TMS1000 . . . 262
4.57 COP8 . . . 263
4.58 SC/MP . . . 263
4.59 SC144xxx . . . 264
4.60 NS32xxx . . . 265
4.61 CR16 . . . 266

4.62 uPD78(C)1x . . . 267
4.63 75K0 . . . 274
4.64 78K0 . . . 275
4.65 78K2/78K3/78K4 . . . 276
4.66 uPD772x . . . 276
4.67 uCOM-43 . . . 276
4.68 F2MC16L . . . 277
4.69 MN161x . . . 278
4.70 CDP180x . . . 278
4.71 KENBAK . . . 279
4.72 HP Nanoprocessor . . . 281
4.73 IM61x0 . . . 281

5 Dateiformate 283
5.1 Code-Dateien . . . 283
5.2 Debug-Dateien . . . 287

6 Hilfsprogramme 291
6.1 PLIST . . . 292
6.2 PBIND . . . 293
6.3 P2HEX . . . 293
6.4 P2BIN . . . 298
6.5 AS2MSG . . . 300

A Fehlermeldungen von AS 301

B E/A-Fehlermeldungen 375

C Programmierbeispiele 379
C.1 16-Bit-Befehle per Makro . . . 379

D Häufig gestellte Fragen 383

E Pseudobefehle und Integer-Syntax, gesammelt 387

F Vordefinierte Symbole 415

G Mitgelieferte Includes 419
G.1 BITFUNCS.INC . . . 419
G.2 CTYPE.INC . . . 420

H Danksagungen 423

I Änderungen seit Version 1.3 **427**

J Hinweise zum Quellcode von AS **443**
J.1 Verwendete Sprache 443
J.2 Abfangen von Systemabhängigkeiten 444
J.3 Systemunabhängige Dateien 445
J.3.1 Von AS genutzte Module 445
J.3.2 Zusätzliche Module für die Hilfsprogramme 450
J.4 Während der Erzeugung von AS gebrauchte Module 451
J.5 Generierung der Nachrichtendateien 453
J.5.1 Format der Quelldateien 453
J.6 Dokumentationserzeugung 455
J.7 Testsuite 456
J.8 Einhängen eines neuen Zielprozessors 456
J.9 Lokalisierung auf eine neue Sprache 463

Kapitel 1

Allgemeines

Diese Anleitung wendet sich an Programmierer, die bereits mit der Programmierung in Assembler vertraut sind und sich darüber informieren möchten, wie man mit AS umgeht. Sie hat eher die Form eines Referenz- und nicht Benutzerhandbuches. Als solches macht sie weder den Versuch, die Sprache Assembler an sich zu erklären, noch erläutert sie die Architektur bestimmter Prozessoren. Im Literaturverzeichnis habe ich weiterführende Literatur aufgelistet, die bei der Implementation der einzelnen Codegeneratoren maßgebend war. Um Assembler von Grund auf zu lernen, kenne ich kein Buch; ich habe es im wesentlichen im ,,Trial and error"-Verfahren gelernt.

1.1 Lizenzbedingungen

Bevor es in medias res geht, erst einmal der unvermeidliche Prolog:

AS in der vorliegenden Version untersteht der GNU General Public License (GPL); die Details dieser Lizenz können Sie in der beiliegenden Datei COPYING nachlesen. Falls Sie diese nicht mit AS erhalten haben, beschweren Sie sich bei demjenigen, von dem Sie AS erhalten haben!

Kurz gesagt, beinhaltet die GPL folgende Punkte:

- Auf AS aufbauende Programme müssen ebenfalls der GPL unterstehen;
- Weiterverbreitung ausdrücklich erlaubt;
- expliziter Haftungsausschluß für durch die Anwendung dieses Programms entstehende Schäden.

...aber für die Details bitte ich wirklich, in den Originaltext der GPL zu schauen!

Um eine möglichst schnelle Fehlerdiagnose und -korrektur zu ermöglichen, bitte ich, Fehlerberichten folgende Angaben beizufügen:

- Betriebssystem (DOS, Windows, Linux...) mit Version
- benutzte Version von AS bzw. Datum des EXE-Files
- bei selbst kompilierten Versionen: verwendeter Compiler und Version
- möglichst die Quelldatei, bei der der Fehler auftritt

Zu erreichen bin ich folgendermaßen:

- per Post:

  Alfred Arnold
  Hirschgraben 29
  52062 Aachen

- per E-Mail: `alfred@ccac.rwth-aachen.de`

Wer mir persönlich Fragen stellen will (und in der Nähe von Aachen wohnt), kann dies mit hoher Wahrscheinlichkeit donnerstags von 20.00 bis 21.00 Uhr im Computerclub an der RWTH Aachen (Elisabethstraße 16, erster Stock, rechter Flur).

Von Telefonanrufen bitte ich abzusehen. Erstens, weil sich die komplizierten Zusammenhänge am Telefon nur äußerst schwer erörtern lassen, und zweitens ist die Telekom schon reich genug...

Die neueste Version von AS (DPMI, Win32, C) findet sich auf folgendem Server:

`http://john.ccac.rwth-aachen.de:8000/as`

oder auch kurz

`http://www.alfsembler.de`

Wer über keinen FTP-Zugang verfügt, kann den Assembler auch von mir anfordern. Ich werde aber nur Anfragen beantworten, die einen CD-Rohling und einen passenden, frankierten Rückumschlag enthalten. **KEIN** Geld schicken!!!

So. Nach diesem unvermeidlichen Vorwort können wir wohl beruhigt zur eigentlichen Anleitung schreiten:

1.2 allgemeine Fähigkeiten des Assemblers

AS bietet im Gegensatz zu normalen Assemblern die Möglichkeit, Code für völlig verschiedene Prozessoren zu erzeugen. Momentan sind folgende Prozessorfamilien implementiert:

- Motorola 68000..68040, 683xx, Coldfire inkl. Koprozessor und MMU
- Motorola ColdFire
- Motorola DSP5600x,DSP56300
- Motorola/IBM MPC601/MPC505/PPC403/MPC821
- IBM PALM
- Motorola M-Core
- Motorola 6800, 6801, 68(HC)11(K4) sowie Hitachi 6301
- Motorola/Freescale 6805, 68HC(S)08
- Motorola 6809 / Hitachi 6309
- Motorola/Freescale 68HC12(X) inklusive XGATE
- Freescale/NXP S12Z ("MagniV")
- Freescale 68RS08
- Motorola 68HC16
- Konami 052001
- Hitachi H8/300(H)
- Hitachi H8/500
- Hitachi SH
- Hitachi HMCS400
- Hitachi H16
- Rockwell 6502, 65(S)C02, Commodore 65CE02, WDC W65C02S, Rockwell 65C19 und Hudson HuC6280

- Rockwell PPS-4
- CMD 65816
- Mitsubishi MELPS-740
- Mitsubishi MELPS-7700
- Mitsubishi MELPS-4500
- Mitsubishi M16
- Mitsubishi M16C
- DEC PDP-11
- DEC VAX
- Western Digital WD16
- AT&T WE32100/32200 (”Bellmac32”)
- Intel 4004/4040
- Intel MCS-48/41, einschließlich Siemens SAB80C382 und der OKI-Varianten
- Intel MCS-51/251, Dallas DS80C390
- Intel MCS-96/196(Nx)/296
- Intel 8080/8085
- Intel i960
- Signetics 8X30x
- Signetics 2650
- Philips XA
- Atmel (Mega-)AVR
- AMD 29K
- Siemens 80C166/167
- Zilog Z80 (einschließlich undokumentierter Befehle), Z180, Z280, Z380, eZ80
- Sharp LR35902 („Gameboy-Z80”)

- Sharp SC61860
- Sharp SC62015
- Zilog Z8, Super8, Z8 Encore
- Zilog Z8000
- Xilinx KCPSM/KCPSM3 ('PicoBlaze')
- LatticeMico8
- Toshiba TLCS-900(L)
- Toshiba TLCS-90
- Toshiba TLCS-870(/C)
- Toshiba TLCS-47
- Toshiba TLCS-42
- Toshiba TLCS-9000
- Toshiba TC9331
- Microchip PIC16C54..16C57
- Microchip PIC16C84/PIC16C64
- Microchip PIC17C42
- Parallax SX20/SX28
- SGS M380/GI LP8000
- SGS-Thomson ST6
- SGS-Thomson ST7/STM8
- SGS-Thomson ST9
- SGS-Thomson 6804
- Texas Instruments TMS32010/32015
- Texas Instruments TMS3202x
- Texas Instruments TMS320C3x/TMS320C4x

- Texas Instruments TMS320C20x/TMS320C5x
- Texas Instruments TMS320C54x
- Texas Instruments TMS320C6x
- Texas Instruments TMS340xx
- Texas Instruments TMS99xx/TMS99xxx
- Texas Instruments TMS7000
- Texas Instruments TMS1000
- Texas Instruments TMS370xxx
- Texas Instruments MSP430(X)
- National Semiconductor IMP-16
- National Semiconductor IPC-16 ('PACE'), INS8900
- National Semiconductor SC/MP
- National Semiconductor INS807x
- National Semiconductor COP4
- National Semiconductor COP8
- National Semiconductor SC144xx
- National Semiconductor NS32xxx
- National Semiconductor CR16A/B/C
- Olympia CP-3F (bzw. SGS M380, GI LP8000)
- Fairchild ACE
- Fairchild F8
- NEC μPD78(C)0x/μPD78(C)1x
- NEC μCOM-43/44/45
- NEC μPD75xx
- NEC μPD75xxx (alias 75K0)

- NEC 78K0
- NEC 78K2
- NEC 78K3
- NEC 78K4
- NEC μPD7720/7725
- NEC μPD77230
- NEC V60
- Fujitsu F^2MC8L
- Fujitsu F^2MC16L
- OKI OLMS-40
- OKI OLMS-50
- Panafacom MN1610/MN1613
- Renesas RX
- Padauk PMS/PMC/PFSxxx
- Symbios Logic SYM53C8xx (ja, die kann man programmieren!)
- Intersil CDP1802/1804/1805(A)
- Intersil IM6100/6120
- XMOS XS1
- MIL STD 1750
- KENBAK-1
- GI CP-1600
- HP Nano Processor

in Arbeit / Planung / Überlegung :

- ARM

- Analog Devices ADSP21xx
- DEC VAX
- SGS-Thomson ST20
- Texas Instruments TMS320C8x
- Zilog eZ80

ungeliebt, aber *doch* vorhanden :

- Intel 8086, 80186, NEC V30..V55 inkl. Koprozessor 8087

Die Umschaltung des Codegenerators darf dabei auch mitten in der Datei erfolgen, und das beliebig oft!

Der Grund für diese Flexibilität ist, daß AS eine Vorgeschichte hat, die auch in der Versionsnummer deutlich wird: AS ist als Erweiterung eines Makroassemblers für die 68000er-Familie entstanden. Auf besonderen Wunsch habe ich den ursprünglichen Assembler um die Fähigkeit zur Übersetzung von 8051-Mnemonics erweitert, und auf dem Weg (Abstieg?!) vom 68000 zum 8051 sind eine Reihe anderer fast nebenbei abgefallen...die restlichen Prozessoren wurden allesamt auf Benutzeranfrage hin integriert. Zumindest beim prozessorunabhängigen Kern kann man also getrost davon ausgehen, daß er gut ausgetestet und von offensichtlichen Bugs frei ist. Leider habe ich aber häufig mangels passender Hardware nicht die Möglichkeit, einen neuen Codegenerator praktisch zu testen, so daß bei Neuerungen Überraschungen nie ganz auszuschließen sind. Das in Abschnitt 1.1 gesagte hat also schon seinen Grund...

Diese Flexibilität bedingt ein etwas exotisches Code-Format, für dessen Bearbeitung ich einige Tools beigelegt habe. Deren Beschreibung findet sich in Abschnitt 6.

AS ist ein Makroassembler, d.h. dem Programmierer ist die Möglichkeit gegeben, sich mittels Makros neue ,,Befehle" zu definieren. Zusätzlich beherrscht er die bedingte Assemblierung. Labels in Makrorümpfen werden automatisch als lokal betrachtet.

Symbole können für den Assembler sowohl Integer-, String- als auch Gleitkommawerte haben. Diese werden — wie Zwischenergebnisse bei Formeln — mit einer Breite von 32, 64, oder 128 Bit für Integerwerte, 80/64 Bit für Gleitkommawerte und 255 Zeichen für Strings gespeichert. Für eine Reihe von Mikrokontrollern besteht die Möglichkeit, durch Segmentbildung die Symbole bestimmten Klassen zuzuordnen.

Dem Assembler kann man auf diese Weise die — begrenzte — Möglichkeit geben, Zugriffe in falsche Adreßräume zu erkennen.

Der Assembler kennt keine expliziten Beschränkungen bzgl. Verschachtelungstiefe von Include-Dateien oder Makros, eine Grenze bildet lediglich die durch den Hauptspeicher beschränkte Rekursionstiefe. Ebenso gibt es keine Grenze für die Symbollänge, diese wird nur durch die maximale Zeilenlänge begrenzt.

Ab Version 1.38 ist AS ein Mehrpass-Assembler. Dieser hochtrabende Begriff bedeutet nicht mehr, als das die Anzahl der Durchgänge durch die Quelltexte nicht mehr zwei sein muß. Sind keine Vorwärtsreferenzen im Quellcode enthalten, so kommt AS mit einem Durchgang aus. Stellt sich dagegen im zweiten Durchgang heraus, daß ein Befehl mit einer kürzeren oder längeren Kodierung benutzt werden muß, so wird ein dritter (vierter, fünfter...) Durchgang eingelegt, um alle Symbolreferenzen richtig zu stellen. Mehr steckt hinter dem Begriff ,,Multipass" nicht...er wird im weiteren Verlauf dieser Anleitung deswegen auch nicht mehr auftauchen.

Nach soviel Lobhudelei ein dicker Wermutstropfen: AS erzeugt keinen linkfähigen Code. Eine Erweiterung um einen Linker wäre mit erheblichem Aufwand verbunden und ist momentan nicht in Planung.

Wer einen Blick in die Quellen von AS werfen will, besorge sich einfach die Unix-Version von AS, die als Quelltext zum Selber übersetzen kommt. Die Quellen sind mit Sicherheit nicht in einem Format, daß das Verständnis möglichst leicht macht - an vielen Stellen schaut noch der originale Pascal-Quellcode heraus, und ich teile einige häufig vertretene Ansichten über 'guten' C-Stil nicht...

1.3 Unterstützte Plattformen

DOS

Obwohl AS als ein reines DOS-Programm angefangen hat, stehen auch eine Reihe von Versionen zur Verfügung, die etwas mehr als den Real-Mode eines Intel-Prozessors ausnutzen können. Diese sind in ihrer Benutzung soweit als möglich kompatibel gehalten zur DOS-Version, es ergeben sich natürlich bisweilen Unterschiede in der Installation und der Einbindung in die jeweilige Betriebssystemumgebung. Abschnitte in dieser Anleitung, die nur für eine bestimmte Version von AS gelten, sind mit einer entsprechenden Randbemerkung (an diesem Absatz für die DOS-Version) gekennzeichnet. Im einzelnen existieren die folgenden, weiteren Versionen (die als getrennte Pakete distributiert werden):

DPMI

Für den Fall, daß man bei der Übersetzung großer, komplexer Programme unter DOS Speicherplatzprobleme bekommt, existiert eine DOS-Version, die mittels eines

DOS-Extenders im Protected Mode abläuft und so das komplette Extended Memory eines ATs nutzen kann. Die Übersetzung wird durch den Extender merklich langsamer, aber immerhin läuft es dann noch...

OS/2

Für Freunde von IBM's Betriebssystem OS/2 gibt es eine native OS/2-Version von AS. Seit 1.41r8 ist diese nur eine volle 32-bittige OS/2-Anwendung, was natürlich zur Folge hat, daß OS/2 2.x und ein 80386-Prozessor jetzt zwingend erforderlich sind.

UNIX

Den reinen PC-Bereich verläßt man mit der C-Version von AS, die so gehalten wurde, daß sie auf einer möglichst großen Zahl von UNIX-artigen Systemen (dazu zählt aber auch OS/2 mit dem emx-Compiler) ohne großartige Verrenkungen übersetzbar ist. Im Gegensatz zu den vorherigen Versionen (die auf den auf Anfrage erhältlichen Pascal-Sourcen basieren) wird die C-Version im Quellcode ausgeliefert, d.h. man muß sich mittels eines Compilers selbst die Binaries erzeugen. Dies ist aber (für mich) der eindeutig einfachere Weg, als ein Dutzend Binaries für Maschinen vorzukompilieren, auf die ich auch nicht immer Zugriff habe...

Kapitel 2

Benutzung des Assemblers

Scotty: Captain, we din' can reference it!
Kirk: Analysis, Mr. Spock?
Spock: Captain, it doesn't appear in the symbol table.
Kirk: Then it's of external origin?
Spock: Affirmative.
Kirk: Mr. Sulu, go to pass two.
Sulu: Aye aye, sir, going to pass two.

2.1 Hardware-Anforderungen

Je nach Version von AS variieren die Hardware-Anforderungen deutlich:

DOS

Die DOS-Version läuft prinzipiell auf allen IBM-kompatiblen PCs, angefangen vom PC/XT mit vierkommawenig Megaherz bis hin zum Pentium. Wie bei vielen anderen Programmen aber auch, steigt der Lustgewinn mit der Hardware-Ausstattung. So dürfte ein XT-Benutzer ohne Festplatte erhebliche Probleme haben, die über 500 Kbyte große Overlay-Datei von AS auf einer Diskette unterzubringen...eine Festplatte sollte der PC also schon haben, allein um vernünftige Ladezeiten zu erreichen. Im Hauptspeicherbedarf ist AS recht genügsam: Das Programm selber belegt knapp 300 Kbyte Hauptspeicher, AS sollte also ab einer Hauptspeichergröße von 512 Kbyte ausführbar sein.

DPMI

Die Version von AS für das DOS-Protected-Mode-Interface (DPMI) benötigt zum Ablaufen mindestens einen 80286-Prozessor und 1 Mbyte freies Extended Memory. Daher stellen 2 Mbyte Hauptspeicher das absolute Minimum dar, wenn man im XMS sonst keine anderen Spielereien (Platten-Cache, RAM-Disk, hochgeladenes

DOS) installiert hat, sonst entsprechend mehr. Falls man die DPMI-Version in einer DOS-Box von OS/2 laufen läßt, so sollte DPMI auch in den DOS-Einstellungen der Box erlaubt sein (Einstellung `An` oder `Auto`) und der Box eine entsprechende Menge von XMS-Speicher zugeordnet sein. Die virtuelle Speicherverwaltung von OS/2 sorgt hier übrigens dafür, daß man sich keine Gedanken machen muß, ob der eingestellte Speicher auch real verfügbar ist.

NIX

Die C-Version von AS wird im Quellcode ausgeliefert und erfordert damit ein Unix- oder OS/2-System mit einem C-Compiler. Der Compiler muß dem ANSI-Standard genügen (GNU-C erfüllt diese Bedingung zum Beispiel). Ob Ihr UNIX-System bereits getestet und die nötigen Definitionen vorgenommen wurden, können Sie der `README`-Datei entnehmen. Als zur Kompilation benötigten Plattenplatz sollten Sie ca. 15 Mbyte veranschlagen; dieser Wert (und der nach der Übersetzung noch benötigte Platz für die übersetzten Programme) variiert allerdings stark von System zu System, so daß man diesen Wert nur als Richtschnur betrachten sollte.

2.2 Lieferumfang

Prinzipiell erhält man AS in einer von zwei Formen: Als *Binärdistribution* oder *Quellcodedistribution*. Im Falle einer Binärdistribution bekommt man AS mit den zugehörigen Dienstprogrammen und Hilfsdateien fertig übersetzt, so daß man nach dem Auspacken des Archivs an die gewünschte Stelle direkt loslegen kann. Binärdistributionen werden für verbreitete Plattformen gemacht, bei denen die Mehrzahl der Benutzer keinen Compiler hat oder die Übersetzung trickreich ist (im Moment sind dies DOS und OS/2). Eine Quellcodedistribution enthält im Gegensatz den kompletten Satz an C-Quellen, um AS zu generieren; es ist letzten Endes ein Schnappschuß des Quellenbaumes, an dem ich AS weiterentwickele. Die Generierung von AS aus dem Quellcode und dessen Struktur ist näher in Anhang J beschrieben, weshalb an dieser Stelle nur auf den Umfang und die Installation einer Binärdistribution beschrieben wird:

Das Archiv des Lieferumfangs gliedert sich in einige Unterverzeichnisse, so daß man nach dem Auspacken sofort einen Verzeichnisbaum erhält. Die Verzeichnisse enthalten im einzelnen:

- `BIN`: ausführbare Programme, Text-Resourcen;
- `INCLUDE`: Include-Dateien für Assemblerprogramme, z.B. Registerdefinitionen oder Standardmakros;
- `MAN`: Kurzreferenzen für die Programme im Unix-Man-Format;

- `DOC`: diese Dokumentation in verschiedenen Formaten;
- `LIB`: vorgesehen für Initialisierungsdateien.

Eine Auflistung der Dateien, die in jeder Binärdistribution enthalten sind, findet sich in Tabelle 2.1. Falls eine der in diesen (oder den folgenden) Tabellen aufgeführten Dateien fehlt, hat jemand (im Zweifelsfalle ich) beim Kopieren geschlafen...

Datei	Funktion
Verzeichnis BIN	
AS.EXE	Programmdatei Assembler
PLIST.EXE	listet Inhalt von Codedateien auf
PBIND.EXE	kopiert Codedateien zusammen
P2HEX.EXE	wandelt Code- in Hexdateien um
P2BIN.EXE	wandelt Code- in Binärdateien um
AS.MSG	Textresourcen zu AS *)
PLIST.MSG	Textresourcen zu PLIST *)
PBIND.MSG	Textresourcen zu PBIND *)
P2HEX.MSG	Textresourcen zu P2HEX *)
P2BIN.MSG	Textresourcen zu P2BIN *)
TOOLS.MSG	gemeinsame Textresourcen zu den Tools *)
CMDARG.MSG	gemeinsame Textresourcen zu allen Programmen *)
IOERRS.MSG	
*) nur DOS	
Verzeichnis DOC	
AS_DE.DOC	deutsche Dokumentation, ASCII-Format
AS_DE.HTML	deutsche Dokumentation, HTML-Format
AS_DE.TEX	deutsche Dokumentation, LaTeX-Format
AS_EN.DOC	englische Dokumentation, ASCII-Format
AS_EN.HTML	englische Dokumentation, HTML-Format
AS_EN.TEX	englische Dokumentation, LaTeX-Format
Verzeichnis INCLUDE	
BCDIC.INC	Definition BCDIC/Codepage 359
BITFUNCS.INC	Funktionen zur Bitmanipulation
CTYPE.INC	Funktionen zur Klassifizierung von Zeichen
EBCDIC.INC	Inkludiert alle EBCDIC-Varianten
CP037.INC	Definition EBCDIC (Codepage 037)
CP5100.INC	Definition Zeichensatz IBM 5100
CP5110.INC	Definition EBCDIC (IBM 5110)

Datei	Funktion
80C50X.INC	Registeradressen SAB C50x
80C552.INC	Registeradressen 80C552
H8_3048.INC	Registeradressen H8/3048
KENBAK.INC	Registeradressen Kenbak-1
RADIX50.INC	Definition RADIX 50 Zeichensatz
REG166.INC	Adressen & Befehlsmakros 80C166/167
REG251.INC	Adressen & Bits 80C251
REG29K.INC	Peripherieadressen AMD 2924x
REG53X.INC	Registeradressen H8/53x
REG6303.INC	Registeradressen 6303
REG683XX.INC	Registeradressen 68332/68340/68360
REG7000.INC	Registeradressen TMS70Cxx
REG78310.INC	Registeradressen & Vektoren 78K3
REG78K0.INC	Registeradressen 78K0
REG96.INC	Registeradressen MCS-96
REGACE.INC	Registeradressen ACE
REGAVROLD.INC	Register- & Bitadressen AVR-Familie (veraltet)
REGAVR.INC	Register- & Bitadressen AVR-Familie (aktuell)
REGCOLD.INC	Registeradressen ColdFire
REGCOP8.INC	Registeradressen COP8
REGEZ80.INC	Registeradressen eZ80
REGF8.INC	Register- & Speicheradressen F8
REGGP32.INC	Registeradressen 68HC908GP32
REGH16.INC	Registeradressen H16
REGHC12.INC	Registeradressen 68HC12
REGM16C.INC	Registeradressen Mitsubishi M16C
REGMSP.INC	Registeradressen TI MSP430
REGPDK.INC	Register- & Bitadressen PMC/PMS/PFSxxx
REGS12Z.INC	Register- & Bitadressen S12Z-Familie
REGST6.INC	Register- & Makrodefinitionen ST6 (aktuell)
REGST7.INC	Register- & Makrodefinitionen ST7
REGSTM8.INC	Register- & Makrodefinitionen STM8
REGST9.INC	Register- & Makrodefinitionen ST9
REGZ280.INC	Registeradressen Z280
REGZ380.INC	Registeradressen Z380
STDDEF04.INC	Registeradressen 6804
STDDEF16.INC	Befehlsmakros und Registeradressen PIC16C5x
STDDEF17.INC	Registeradressen PIC17C4x

Datei	Funktion
STDDEF18.INC	Registeradressen PIC16C8x
STDDEF2X.INC	Registeradressen TMS3202x
STDDEF37.INC	Register- & Bitadressen TMS370xxx
STDDEF3X.INC	Peripherieadressen TMS320C3x
STDDEF4X.INC	Peripherieadressen TMS320C4x
STDDEF47.INC	Befehlsmakros TLCS-47
STDDEF51.INC	Definition von SFRs und Bits für 8051/8052/80515
STDDEF56K.INC	Registeradressen DSP56000
STDDEF5X.INC	Peripherieadressen TMS320C5x
STDDEF60.INC	Befehlsmakros & Registeradressen PowerPC
STDDEF62.INC	Registeradressen & Makros ST6 (veraltet)
STDDEF75.INC	Registeradressen 75K0
STDDEF87.INC	Register- & Speicheradressen TLCS-870
STDDEF90.INC	Register- & Speicheradressen TLCS-90
STDDEF96.INC	Register- & Speicheradressen TLCS-900
STDDEFXA.INC	SFR-& Bitadressen Philips XA
STDDEFZ8.INC	Registeradressen Z8-Familie (alt)
REGV60.INC	Registeradressen NEC V60
REGZ8.INC	Registeradressen Z8-Familie (neu)
REGSX20.INC	Register- & Bitadressen Parallax SX20/28
AVR/.INC	Register- & Bitadressen AVR-Familie (nicht direkt benutzen, REGAVR.INC inkludieren)
COLDFIRE*.INC	Register- & Bitadressen ColdFire-Familie (nicht direkt benutzen, REGCOLD.INC inkludieren)
EZ80*.INC	Register- & Bitadressen eZ80-Familie (nicht direkt benutzen, REGEZ80.INC inkludieren)
PDK*.INC	Register- & Bitadressen PMC/PMS/PFSxxx (nicht direkt benutzen, REGPDK.INC inkludieren)
S12Z*.INC	Register- & Bitadressen S12Z-Familie (nicht direkt benutzen, REGS12Z.INC inkludieren)
ST6*.INC	Register- & Bitadressen ST6-Familie (nicht direkt benutzen, REGST6.INC

Datei	Funktion
	inkludieren)
ST7*.INC	Register- & Bitadressen ST7-Familie (nicht direkt benutzen, REGST7.INC inkludieren)
STM8*.INC	Register- & Bitadressen STM8-Familie (nicht direkt benutzen, REGSTM8.INC inkludieren)
Z8*.INC	Register- & Bitadressen Z8-Familie (nicht direkt benutzen, REGZ8.INC inkludieren)
Verzeichnis LIB	
Verzeichnis MAN	
ASL.1	Kurzanleitung zu AS
PLIST.1	Kurzanleitung zu PLIST
PBIND.1	Kurzanleitung zu PBIND
P2HEX.1	Kurzanleitung zu P2HEX
P2BIN.1	Kurzanleitung zu P2BIN

Tabelle 2.1: Standardumfang einer Binärdistribution

Je nach Plattform kann eine Binärdistribution aber noch weitere Dateien enthalten, um einen Betrieb zu ermöglichen, wie es z.B. bei DOS-Extendern der Fall ist.

PMI

Für die DOS-DPMI-Version ergeben sich die in Tabelle 2.2 gelisteten Ergänzungen. Es spricht übrigens nichts dagegen, als Hilfsprogramme die Versionen aus einer DOS-Distribution zu verwenden, da diese einerseits ohne den Extender-Overhead deutlich schneller ablaufen und andererseits den vom Extender bereitgestellten erweiterten Speicher nicht benötigen.

Datei	Funktion
Verzeichnis BIN	
DPMI16BI.OVL	DPMI-Server für den Assembler
RTM.EXE	Laufzeit-Modul des Extenders

Tabelle 2.2: Zusätzliche Dateien in einer DPMI-Binärdistribution

S/2

Eine OS/2-Binärdistribution enthält neben den Basisdateien eine Reihe von DLLs, die zur Laufzeitumgebung des verwendeten emx-Compilers gehören (Tabelle 2.3). Falls man diese DLLs (oder neuere Versionen davon) bereits besitzt, kann man diese auch wieder löschen und seine eigenen benutzen.

Datei	Funktion
Verzeichnis BIN	
EMX.DLL EMXIO.DLL EMXLIBC.DLL EMXWRAP.DLL	Laufzeitbibliotheken für AS und die Dienstprogramme

Tabelle 2.3: Zusätzliche Dateien in einer OS/2-Binärdistribution

2.3 Installation

Eine besondere Installation ist für die Nutzung einer Binärdistribution nicht notwendig, es genügt, das Archiv an passender Stelle auszupacken und dann noch einige Kleinigkeiten zu ergänzen. Als Beispiel hier eine Installation, wie sie vielleicht ein UNIX-Anhänger vornehmen würde: *DOS*

Legen Sie ein Verzeichnis `c:\as` an (im folgenden nehme ich an, daß Sie AS auf Laufwerk C installieren wollen), wechseln Sie in dieses und entpacken Sie das Archiv unter Erhalt der Verzeichnisnamen (bei Verwendung von PKUNZIP ist dazu die Kommandozeilenoption `-d` erforderlich). Sie sollten jetzt folgenden Verzeichnisbaum haben:

```
c:\as
c:\as\bin
c:\as\include
c:\as\lib
c:\as\man
c:\as\doc
```

Ergänzen Sie jetzt die `PATH`-Anweisung in Ihrer `AUTOEXEC.BAT` um das Verzeichnis `c:\as\bin`, so daß AS und seine Hilfsprogramme vom System gefunden werden. In dem `lib`-Verzeichnis erzeugen Sie mit einem beliebigen Texteditor eine Datei `AS.RC` mit folgendem Inhalt:

```
-i c:\as\include
```

Diese sogenannte *Key-Datei* zeigt AS, in welchem Verzeichnis er seine Include-Dateien suchen soll. Damit AS diese Key-Datei bei Start auch beachtet, muß noch folgende Anweisung in die `AUTOEXEC.BAT`:

```
set ASCMD=@c:\as\lib\as.rc
```

Was Sie alles noch in der Key-Datei voreinstellen können, steht im folgenden Abschnitt.

PMI

Die Installation der DPMI-Version sollte im Prinzip genauso verlaufen wie der reinen DOS-Version; wenn der Pfad das `bin`-Verzeichnis enthält, werden die Dateien des DOS-Extenders automatisch gefunden und man sollte von dieser Mimik (mit Ausnahme der längeren Anlaufzeit...) nichts mitbekommen. Theoretisch ist es möglich, daß Sie auf 80286-Rechnern beim ersten Start mit einer Meldung der folgenden Form konfrontiert werden:

```
machine not in database (run DPMIINST)
```

Da das Tool DPMIINST bei neueren Versionen des DOS-Extenders von Borland aber nicht mehr dabei ist, nehme ich einmal an, daß diese Sache sich erledigt hat...falls doch nicht, bitte ich um Rückmeldung!

S/2

Die Installation der OS/2-Version kann in weiten Zügen genauso ablaufen wie für die DOS-Version, nur daß dem System noch die DLLs bekannt gemacht werden müssen. Wenn Sie den `LIBPATH`-Eintrag in Ihrer `CONFIG.SYS` nicht erweitern wollen, ist es natürlich auch möglich, die DLLs in ein Verzeichnis zu verschieben, das bereits dort aufgeführt ist.

NIX

Wie bereits erwähnt, beschränkt sich die Installationsbeschreibung hier nur auf Binärdistributionen. Da eine Installation unter Unix im Augenblick immer eine Quellcodedistribution ist, geht der Verweis hier unisono in Anhang J.

2.4 Aufruf, Parameter

AS ist ein Kommandozeilen-gesteuertes Programm, d.h. alle Parameter und Dateiangaben sind in der Kommandozeile anzugeben.

Zu AS gehört eine Reihe Reihe von Nachrichtendateien (erkennbar an der Endung `MSG`, aus denen AS zur Laufzeit die für die jeweilige Landessprache dynamisch nachlädt. AS sucht nach diesen Dateien in den folgenden Verzeichnissen:

- im aktuellen Verzeichnis;
- im Verzeichnis der EXE-Datei;
- in dem in der Environment-Variablen `AS_MSGPATH` angegebenen Verzeichnis, oder alternativ in den in der `PATH`-Variablen gelisteten Verzeichnissen;

- In dem Verzeichnis, das AS zur Kompilationszeit durch das Makro `LIBDIR` mitgegeben wurde.

Diese Dateien werden von AS *zwingend* zum Betrieb benötigt, d.h. findet AS diese Dateien nicht, bricht er an dieser Stelle sofort ab.

Die Auswahl der Sprache (momentan Deutsch oder Englisch) orientiert sich unter DOS und OS/2 an der `COUNTRY`-Einstellung in der `CONFIG.SYS`, unter Unix an der `LANG`-Environment-Variablen.

Um den Speicherbedarf von AS unter DOS überhaupt befriedigen zu können, wurden die verschiedenen Codegeneratormodule in der DOS-Version in einen Overlay verlegt, der Teil des EXE-Files ist. Eine getrennte OVR-Datei wie bei früheren Versionen von AS existiert also nicht mehr, AS versucht aber wie bisher auch weiterhin, die durch das Overlaying entstehenden Verzögerungen durch Nutzung von eventuellem EMS- oder XMS-Speicher zu reduzieren. Sollte diese Verwendung zu Problemen führen, so können Sie die Verwendung von EMS bzw. XMS unterbinden, indem Sie einer Environment-Variablen `USEXMS` bzw. `USEEMS` den Wert `n` zuweisen. So kann man z.B. mit dem Befehl *DOS*

```
SET USEXMS=n
```

die Verwendung von Extended Memory verhindern.

Da AS alle Ein-und Ausgaben über das Betriebssystem abwickelt (und daher unter DOS auch auf nicht ganz so kompatiblen PCs laufen sollte) und eine rudimentäre Bildschirmsteuerung benötigt, gibt er während der Assemblierung ANSI-Steuersequenzen aus. Falls Sie in den Ausgaben von AS also seltsame Zeichen sehen sollten, fehlt offensichtlich in Ihrer CONFIG.SYS die Einbindung des ANSI-Treibers (`device=ansi.sys`), die weitere Funktion von AS wird dadurch aber nicht beeinflußt. Alternativ können Sie aber auch die Ausgabe von ANSI-Sequenzen durch das Setzen der Environment-Variablen `USEANSI` auf `n` ganz unterdrücken. *DOS/ DPMI*

Der DOS-Extender der DPMI-Version läßt sich in seiner Speicherbelegung durch diverse Kommandozeilenoptionen beeinflussen. Diese können Sie bei Bedarf der Datei DPMIUSER.DOC entnehmen. Zusätzlich ist sie in der Lage, bei Bedarf den vorhandenen Speicher durch eine Swap-Datei zu „erweitern". Dazu belegt man eine Environment-Variable `ASXSWAP` folgendermaßen: *DPMI*

```
SET ASXSWAP=<Größe>[,Dateiname]
```

Die Größenangabe erfolgt in Megabytes und **muß** gemacht werden. Der Name der Datei ist dagegen optional; fehlt er, so wird die Swap-Datei im aktuellen Verzeichnis unter dem Namen `ASX.TMP` angelegt. In jedem Falle wird die Swap-Datei nach Programmende wieder gelöscht.

Die Kommandozeilenparameter können grob in drei Klassen eingeteilt werden: Schalter, Key-File-Referenzen (s.u.) und Dateispezifikationen. Parameter dieser beiden Klassen können beliebig gemischt in der Kommandozeile auftreten, AS wertet zuerst alle Parameter aus und assembliert dann die angegebenen Dateien. Daraus folgen zwei Dinge:

- Die angegebenen Schalter wirken auf alle angegebenen Quelldateien. Sollen mehrere Quelldateien mit unterschiedlich gesetzten Schaltern assembliert werden, so muß dies in getrennten Läufen erfolgen.
- Es können in einem Durchgang mehrere Dateien assembliert werden. Um der Sache die Krone aufzusetzen, dürfen die Dateiangaben Jokerzeichen enthalten.

Schalterparameter erkennt AS daran, daß sie durch einen Schrägstrich (/) oder Bindestrich (-) eingeleitet werden. Es gibt dabei sowohl Schalter, die nur aus einem Buchstaben bestehen, als auch Schalter, die aus einem ganzen Wort bestehen. Immer wenn AS einen Schalter nicht als ,,Wort-Schalter" verstehen kann, so versucht er, die Buchstaben des Wortes als einzelne Schalter zu interpretieren. Wenn man also z.B.

```
-queit
```

anstelle von

```
-quiet
```

geschrieben hätte, würde AS die Buchstaben `q, u, e, i` und `t` als einzelne Schalter auffassen. Mehrbuchstabige Schalter unterscheiden sich weiterhin von einbuchstabigen dadurch, daß AS bei ihnen beliebige Groß-und Kleinschreibungen akzeptiert, während einbuchstabige Schalter je nach Groß- oder Kleinschreibung unterschiedliche Bedeutung haben.

Momentan sind folgende Schalter definiert:

- `l`: Assembler-Listing auf Konsole ausgeben. Falls mehrere Passes ausgeführt werden müssen, landen im Gegensatz zur nächsten Option die Listings aller Durchgänge auf der Ausgabe!

- `L`: Assembler-Listing auf Datei schreiben. Die Listdatei erhält dabei den gleichen Namen wie die Quelldatei, lediglich die Endung wird durch `LST` ersetzt, es sei denn...

- `OLIST`: mit einem zusätzlichen Argument legt einen anderen Pfad bzw. Namen für die Listdatei fest. Falls mehrere Dateien assembliert werden, kann diese Option auch mehrfach gegeben werden.

- `listline-prefix`: Mit dieser Option können die Daten bestimmt werden, die im Listing jeder Zeile vorangestellt werden. Der Aufbau des dafür verwendeten Formatstrings ist in 2.6 beschrieben.

- `list-unknown-values` bzw. `no-list-unknown-values`: Schaltet die Anzeige von im ersten Durchlauf unbekannten Werten als Fragezeichen an bzw. aus.

- `RADIX`: Stellt das Default-Zahlensystem auf einen Wert zwischen 2 und 36, anstelle von 10. Dieser Wert kann durch die gleichnamige Anweisung im Programm übersteuert werden.

- `fred`: Erlaubt es, für Ganzzahl-Symbole eine reduzierte Bitlänge vorzugeben. Diese ergibt sich bei der Definition aus der verwendeten Stellenzahl.

- `listing`: Gibt vor, ob im Listing des assemblierten Codes alles ausgegeben werden soll, oder durch bedingte Assemblierung ausgelassenene Teile nicht. Die erlaubten Werte sind die gleichen wie in 3.7.4 beschrieben.

- `LISTRADIX`: Defaultmäßig erfolgen alle Zahlenausgaben im Listing (Adressen, erzeugter Code, Symboltabelle) im Hexadezimalsystem. Mit diesem Schalter kann ein beliebiges anderes Zahlensystem im Bereich 2 bis 36 angegeben werden, z.B. '-listradix 8' für oktale Ausgaben. Wird der Zahlenwert mit führender Null geschrieben (also z.B. 08 anstelle 8), wird der Programmzähler im Listing mit führenden Nullen geschrieben.

- `SPLITBYTE [Zeichen]`: Zahlen im Listing werden Byte-weise dargestellt, mit dem angegebenen Zeichen als Trenner. Ein Punkt wird als Trenner verwendet, wenn kein Zeichen angegeben wurde. Diese Option wird üblicherweise zusammen mit der `LISTRADIX`-Option verwendet; Listradix 8 zusammen mit einem Punkt als Trenner ergibt die sogenannte "Split-Octal"-Darstellung.

- `o`:Bestimmt einen neuen Namen für die von AS zu erzeugende Code-Datei. Wird diese Option mehrfach verwendet, so werden die angegebenen Namen nacheinander den zu assemblierenden Quelldateien zugeordnet; Negation (s.u.) dieser Option in Verbindung mit einem Namen löscht den Namen aus der Liste; Negation ohne Namensangabe löscht die komplette Liste.

- `SHAREOUT`:dito, nur für eine eventuell zu erzeugende SHARE-Datei
- `c`: SHARED-Variablen werden in einem Format abgelegt, das die Einbindung in eine C-Quelldatei erlaubt. Die Endung der Datei ist `H`.
- `p`: SHARED-Variablen werden in einem Format abgelegt, das die Einbindung in den `CONST`-Block eines Pascal- oder Modula-Programms erlaubt. Die Endung der Datei ist `INC`.
- `a`: SHARED-Variablen werden in einem Format abgelegt, das die Einbindung in eine Assembler-Quelldatei erlaubt. Die Endung der Datei ist `INC`.

Zu Sinn und Funktion der SHARED-Symbole siehe Kapitel 2.12 bzw. 3.9.1.

- `g [Format]`: Mit diesem Schalter erzeugt AS zusätzlich eine Datei, die Debug-Informationen für dieses Programm enthält. Als Format ist dabei entweder ein AS-eigenes Format (`Format=MAP`), eine NoICE-kompatible Kommandodatei (`Format=NOICE`) oder das Format der AVR-Tools (`Format=ATMEL`) erlaubt. Zu den im MAP-Format gespeicherten Informationen gehört zum einen die Symboltabelle, zum anderen eine Zuordnung von Quellzeilen zu Maschinenadressen. Eine genauere Beschreibung des benutzten MAP-Dateiformates findet sich in Abschnitt 5.2. Die Endung der Datei ist `MAP`, `NOI` bzw. `OBJ`, je nach gewähltem Format. Wird keine explizite Formatangabe gemacht, wird das MAP-Format gewählt.
- `noicemask [Wert]`: Normalerweise listet AS in NoICE-Debuginfos nur Symbole aus dem CODE-Segment. Mit dieser Option und einem als Bitmaske zu verstehenden Wert lassen sich andere Symbole aus anderen Segmenten zuschalten. Die Zuordnung von Bits zu Segmenten kann Tabelle 5.2 entnommen werden.
- `w`: Ausgabe von Warnungen unterdrücken;
- `E [Datei]`: Die von AS erzeugten Fehlermeldungen und Warnungen in eine Datei umleiten. Anstatt einer Datei können auch die 5 Standardhandles (STDIN..STDPRN) als !0 bis !4 angegeben werden. Default ist !2, also STDERR. Wird die Dateiangabe weggelassen, so ist der Name der Fehlerdatei gleich dem der Quelldatei, nur mit der Endung `LOG`.
- `q`: Dieser Schalter unterdrückt alle Meldungen von AS mit Ausnahme von Fehlermeldungen und vom Programm selber erzeugten Ausgaben. Die Assemblierzeit kann dadurch geringfügig reduziert werden, und beim Aufruf aus einer Shell heraus kann man sich eine Umleitung ersparen. Der Nachteil ist, daß man

u.U. einige Minuten „im Dunklen" steht... Anstelle von 'q' darf auch 'quiet' geschrieben werden.

- `v`: Dies fordert auführliche Angaben an, also das Gegenteil zum Quiet-Modus. Die einzige Angabe, die momentat zusätzlich ausgegeben wird, ist die Versionsinformation.

- `version`: Gibt Versionsinformationen aus und beendet sich.

- `h`: Hexadezimalzahlen mit Klein- anstelle von Großbuchstaben ausgeben. Diese Option ist in erster Linie eine Frage des persönlichen Geschmacks.

- `i <Pfadliste>`: gibt eine Liste von Verzeichnissen an, in denen der Assembler automatisch nach Include-Dateien suchen soll, falls er diese nicht im aktuellen Verzeichnis findet. Die einzelnen Verzeichnisse müssen durch Semikolons getrennt werden.

- `u`: eine Liste der in den Segmenten belegten Bereiche berechnen. Sie ist nur sinnvoll, falls ein Listing erzeugt wird. Diese Option benötigt erhebliche zusätzliche Speicher- und Rechenleistung, im Normalbetrieb sollte sie daher abgeschaltet sein. Da AS aber unabhängig vom eingeschalteten Listing mit dieser Option auf überlappende Speicherbelegung prüft, hat sie auch unabhängig vom Listing einen gewissen Sinn...

- `C`: erzeugt eine Liste mit Querverweisen. Aufgelistet wird, welche (globalen) Symbole in welchen Dateien in welchen Zeilen benutzt werden. Auch diese Liste wird nur generiert, falls ein Listing erzeugt wird und belegt während der Assemblierung zusätzlichen Speicherplatz.

- `s`: eine Liste aller Sektionen (s. Abschnitt 3.8) ausgeben. Die Verschachtelung wird dabei durch Einrückungen angedeutet.

- `I`: Analog zur Sektionsliste eine Liste aller bearbeiteten Include-Dateien ausgeben.

- `t <Maske>`: Mit diesem Schalter lassen sich einzelne Komponenten des standardmäßig ausgegebenen Assembler-Listings ein-und ausblenden. Welcher Teil dabei welchem Bit zugeordnet ist, ist im übernächsten Abschnitt, der genauer auf das Format des Assembler Listings eingeht, nachgelesen werden.

- `D <Symbolliste>`: Symbole definieren. Die hinter dieser Option angegebenen, durch Kommas getrennten Symbole werden in der globalen Symboltabelle vor Beginn der Assemblierung abgelegt. Defaultmäßig werden diese Symbole als ganze Zahlen mit dem Wert TRUE abgelegt, mit einem nachgestellten

Gleichheitszeichen kann aber auch eine andere Belegung gewählt werden. Der dem Gleichheitszeichen folgende Ausdruck darf dabei auch Operatoren oder interne Funktionen beinhalten, jedoch **KEINE** anderen Symbole, selbst wenn diese schon davor in der Liste definiert sein sollten! Zusammen mit den Befehlen zur bedingten Assemblierung (siehe dort) können so per Kommandozeile aus einer Quelldatei unterschiedliche Programmversionen erzeugt werden. **ACHTUNG!** Wenn case-sensitiv gearbeitet werden soll, muß dies in der Kommandozeile *vor* Symboldefinitionen angegeben werden, sonst werden Symbolnamen schon an dieser Stelle in Großbuchstaben umgewandelt!

- `A`: Die Liste globaler Symbole in einer anderen, kompakteren Form ablegen. Verwenden Sie diese Option, wenn der Assembler bei langen Symboltabellen mit einem Stapelüberlauf abstürzt. Eventuell kann diese Option die Arbeitsgeschwindigkeit des Assemblers erhöhen, dies hängt jedoch von den Quellen ab.

- `x`: Legt die Ausführlichkeitsstufe von Fehlermeldungen fest. Jedes Mal, wenn diese Option angegeben wird, wird die Stufe um eins erhöht oder gesenkt. Während auf Stufe 0 (Vorgabe) nur der Fehler selber ausgegeben wird, wird ab Stufe 1 noch eine erweiterte Meldung ausgegeben, anhand der die Identifizierung des Fehlers erleichtert werden soll. Welche Fehlermeldungen welche Zusatzinformationen tragen können, steht in Anhang A mit der Liste aller Fehlermeldungen. Auf Stufe 2 (Maximum) wird zusätzlich noch die betroffene Quellzeile mit ausgegeben.

- `n`: Wird diese Option angegeben, so werden Fehlermeldungen nicht nur mit ihrem Klartext, sondern auch mit ihren im Anhang A genannten internen Nummern ausgegeben. Diese Option ist primär für Shells und Entwicklungsumgebungen gedacht, denen mit diesen Nummern die Identifizierung von Fehlern erleichtert werden soll.

- `U`: Mit dieser Option schaltet man AS in den case-sensitiven Modus um, d.h. in Namen von Symbolen, Sektionen, Makros, Zeichentabellen und selbst definierte Funktionen werden Klein- und Großbuchstaben unterschieden, was normalerweise nicht der Fall ist.

- `P`: weist AS an, den von Makroprozessor und bedingter Assemblierung bearbeiteten Quelltext in einer Datei abzulegen. Dieser Datei fehlen zusätzlich Leer- und reine Kommentarzeilen. Die Endung der Datei ist `I`.

- `M`: mit diesem Schalter erzeugt AS eine Datei, in der die Definitionen der Makros aus der Quelldatei abgespeichert werden, die die `EXPORT`-Option ver-

wenden. Diese neue Datei hat den gleichen Namen wie die Quelldatei, lediglich die Endung wird in `MAC` geändert.

- `G`: Dieser Schalter bestimmt, ob AS Code erzeugen soll oder nicht. Ist er ausgeschaltet, wird die Datei zwar assembliert, aber keine Code-Datei geschrieben. Dieser Schalter ist defaultmäßig aktiviert (logisch, sonst bekäme man ja auch gar kein Code-File).

- `r [n]`: Warnungen ausgeben, falls Situationen eintreten, die einen weiteren Pass erfordern. Diese Information kann genutzt werden, um die Anzahl der Durchläufe zu verringern. Optional kann man die Nummer des Passes angeben, ab dem diese Warnungen erzeugt werden; ohne Angabe kommen die Warnungen ab dem ersten Pass. Machen Sie sich aber so oder so auf einen ziemlichen Haufen an Meldungen gefaßt!!

- `bigendian`: Mit diesem Schalter schaltet man vom Beginn des Programms die Ablage von Werten im Speicher im Big Endian Modus ein, sofern die Zielarchitektur den gleichnamigen Pseudobefehl unterstützt (siehe Abschnitt 3.2.16).

- `plainbase`: Mit diesem Schalter erlaubt man vom Beginn des Programms an das Fortlassen eines leeren Indexarguments (siehe Abschnitt 3.2.15).

- `underscore-macroargs`: Erlaubt die Verwendung von Unterstrichen in Namen von Makro-Argumenten (siehe 3.4.1).

- `relaxed`: Mit diesem Schalter aktiviert man den RELAXED-Modus vom Beginn des Programms an, der ansonsten erst durch die gleichnamige Pseudoanweisung (siehe Abschnitt 3.9.7) eingeschaltet werden muß.

- `intsyntax`: Mit diesem Schalter kann man vom Beginn des Programms an die Liste der erlaubten Schreibweisen für Ganzzahl-Konstanten erweitern oder reduzieren, analog zur gleichnamigen Pseudoanweisung (siehe Abschnitt 3.9.6 für eine Liste erlaubter Argumente).

- `supmode`: Mit diesem Schalter erlaubt man vom Beginn des Programms an die Verwendung von Maschinenbefehlen, die nur im Supervisor-Modus des Prozessors verwendet werden dürfen (siehe Abschnitt 3.2.4).

- `extracomments`: Mit diesem Schalter erlaubt man vom Beginn an die Verwendung alternativer Methoden, um Kommentare im Programm unterzubringen (siehe Abschnitt 3.2.14).

- `Y`: Mit diesem Schalter weist man AS an, alle Fehlermeldungen wegen zu langer Sprungdistanzen zu verwerfen, sobald die Notwendigkeit eines neuen Durchlaufs feststeht. In welchen (seltenen) Situationen dieser Schalter notwendig ist, kann man in Abschnitt 2.10 nachlesen.

- `cpu <Name>`: Hiermit kann man man den Zielprozessor vorgeben, für den AS Code erzeugen soll, wenn die Quelldatei keinen `CPU`-Befehl enthält. Falls das gewählte Ziel CPU-Argumente unterstützt (siehe Abschnitt 3.2.3), können diese auch hier angegeben werden. Ein Aufruf mit `?` oder `list` als Argument listet alle implementierten Zielprozessoren auf.

- `alias <neu>=<alt>`:
  definiert den Prozessortyp `<neu>` als einen Alias für den Typen `<alt>`. Zu den Sinn und Zweck von Aliasen siehe Abschnitt 2.13

- `gnuerrors`: Meldungen über Fehler bzw. Warnungen und deren Position nicht im Standardformat von AS, sondern in einem dem GNU C-Compiler entlehnten Format anzeigen. Dies erleichtert die Integration von AS in für dieses Format ausgelegte Umgebungen, unterdrückt aber gleichzeitig die Anzeige der präzisen Fehlerposition innerhalb Makrorümpfen!

- `maxerrors [n]`: Weist den Assembler an, nach der gegebenen Anzahl von Fehlern die Assemblierung abzubrechen.

- `maxinclevel [n]`: Weist den Assembler an, nach der gegebenen Include-Verschachtelungstiefe abzubrechen (Default ist 200).

- `maxsympass [n]`: Definiert, bis zu welchen Pass unaufgelöste Vorwärtsreferenzen erlaubt sind. Eine Umstellung auf einen Wert größer als den Vorgabewert (1) kann in Sonderfällen, wie bei verketteten Vorwärtsreferenzen erforderlich sein.

- `Werror`: Weist den Assembler an, Warnungen als Fehler zu behandeln.

- `wrelative` bzw. `wno-relative`: Weist den Assembler an, Warnungen auszugeben, falls ein relativer anstelle eines absoluten Sprungs möglich ist (nur für Z80-Target).

- `wimplicit-sign-extension` bzw. `wno-implicit-sign-extension`: Weist den Assembler an, Warnungen über implizite Vorzeichen-Erweiterungen auszugeben (nur 68K, MOVEQ).

- `compmode`: Dieser Schalter weist den Assembler an, im Default im Kompatibilitätsmodus zu arbeiten. Genauere Informationen zu diesem Modus finden sich im Abschnitt 3.9.8.

- `packing`: Dieser Schalter übersteuert den Architektur- abhängigen Default für die `PACKING`-Option (siehe Abschnitt 3.2.10).

Sofern Schalter keine Argumente benötigen und ihre Zusammenziehung keinen mehrbuchstabigen Schalter ergibt, können mehrere Schalter auch auf einen Rutsch angegeben werden, wie z.B im folgenden Beispiel:

```
asl test*.asm firstprog -cl /i c:\as\8051\include
```

Es werden alle Dateien TEST*.ASM sowie die Datei FIRSTPROG.ASM assembliert, wobei für alle Dateien Listings auf der Konsole ausgegeben und Sharefiles im C-Format erzeugt werden. Nach Includes soll der Assembler zusätzlich im Verzeichnis `C:\AS\8051\INCLUDE` suchen.

Dieses Beispiel zeigt nebenbei, daß AS als Defaultendung für Quelldateien `ASM` annimmt.

Etwas Vorsicht ist bei Schaltern angebracht, die ein optionales Argument haben: Folgt auf einen solchen Schalter ohne Argument ein Dateiname, so versucht AS, diesen als Argument zu verwerten, was naturgemäß schief geht:

```
as -g test.asm
```

Die Lösung wäre in diesem Fall, die `-g`-Option ans Ende der Kommandozeile zu setzen oder ein explizites `MAP`-Argument zu spezifizieren.

Neben der Angabe in der Kommandozeile können dauernd benötigte Optionen in der Environment-Variablen ASCMD abgelegt werden. Wer z.B. immer Listdateien haben möchte und ein festes Include-Verzeichnis hat, kann sich mit dem Befehl

```
set ASCMD=-L -i c:\as\8051\include
```

eine Menge Tipparbeit ersparen. Da die Environment-Optionen vor der Kommandozeile abgearbeitet werden, können Optionen in der Kommandozeile widersprechende im Environment übersteuern.

Bei sehr langen Pfaden kann es jedoch auch in der ASCMD-Variablen eng werden. Für solche Fälle kann auf eine sogenannte *Key*-Datei ausgewichen werden, in der die Optionen genauso wie in der Kommandozeile oder ASCMD-Variablen abgelegt werden können, nur daß diese Datei mehrere Zeilen mit jeweils maximal 255 Zeichen enthalten darf. Wichtig ist dabei, daß bei Optionen, die ein Argument benötigen, sowohl Schalter als auch Argument in **einer** Zeile stehen müssen. Der Name der Datei wird AS dadurch mitgeteilt, daß er mit einem vorangestellten Klammeraffen in der ASCMD-Variablen abgelegt wird, z.B.

```
set ASCMD=@c:\as\as.key
```

Um Optionen in der ASCMD-Variablen (oder der Key-Datei) wieder aufzuheben, kann die Option mit einem vorangestellten Pluszeichen wieder aufgehoben werden. Soll in einem Einzelfall z.B. doch kein Listing erzeugt werden, so kann es mit

```
as +L <Datei>
```

wieder aufgehoben werden. Natürlich ist es nicht ganz logisch, eine Option mit einem Pluszeichen zu negieren...UNIX soit qui mal y pense.

Referenzen auf eine Key-Datei können nicht nur von der `ASCMD`-Variablen aus erfolgen, sondern auch direkt von der Kommandozeile aus, indem man analog zur `ASCMD`-Variablen dem Dateinamen einen Klammeraffen voranstellt:

```
as @<Datei> ....
```

Die in einem solchen Fall aus dem Key-File gelesenen Optionen werden so eingearbeitet, als hätten sie anstelle dieser Referenz in der Kommandozeile gestanden - es ist also *nicht* wie bei der `ASCMD`-Variablen so, daß sie vor allen anderen Kommandozeilenoptionen abgearbeitet werden würden.

Das Referenzieren eines Key-Files von einem Key-File selber ist nicht erlaubt und wird von AS mit einer Fehlermeldung quittiert.

Für den Fall, daß Sie AS von einem anderen Programm oder einer Shell aufrufen wollen und diese Shell nur Klein- oder Großbuchstaben in der Kommandozeile übergeben will, existiert folgendes Workaround: Wird vor den Buchstaben der Option eine Tilde gesetzt, so werden die folgenden Buchstaben immer als Kleinbuchstaben interpretiert. Analog erzwingt ein Lattenzaun die Interpretation als Großbuchstaben. Es ergeben sich z.B. folgende Transformationen:

```
/~I ⟶ /i
-#u ⟶ -U
```

Abhängig vom Ablauf der Assemblierung endet der Assembler mit folgenden Returncodes:

0 fehlerfreier Ablauf, höchstens Warnungen aufgetreten

1 Der Assembler hat nur die Aufrufparameter ausgegeben und endete danach sofort.

2 Es sind Fehler bei der Assemblierung aufgetreten, es wurde keine Code-Datei erzeugt.

3 Es trat ein fataler Fehler während des Ablaufes auf, der zum sofortigen Abbruch geführt hat.

4 Bereits während des Starts des Assemblers ist ein Fehler aufgetreten. Dies kann ein Parameterfehler oder eine fehlerhafte Overlay-Datei sein.

255 Bei der Initialisierung ist irgendein interner Fehler aufgetreten, der auf keinen Fall auftreten sollte...neu booten, noch einmal probieren, und bei Reproduzierbarkeit mit mir Verbindung aufnehmen!

Zusätzlich endet jede Assemblierung einer Datei mit einer kleinen Statistik, die Fehlerzahlen, Laufzeit, Anzahl der Durchläufe und freien Speicher ausgibt. Bei eingeschaltetem Assembler-Listing wird diese Statistik zusätzlich auch in das Listing geschrieben.

OS/2 erweitert wie Unix das Datensegment einer Anwendung erst dann, wenn sie wirklich mehr Speicher anfordert. Eine Angabe wie *OS/2*

```
511 KByte verfügbarer Restspeicher
```

bedeutet also nicht einen nahenden Systemabsturz wegen Speichermangel, sondern stellt nur den Abstand zu der Grenze dar, bei der OS/2 einfach ein paar mehr Kohlen in den Ofen schaufelt...

Da es unter C auf verschiedenen Betriebssystemen keine kompatible Möglichkeit gibt, den noch verfügbaren Speicher bzw. Stack zu ermitteln, fehlen bei der C-Version diese beiden Angaben ganz. *UNIX*

2.5 Format der Eingabedateien

Wie die meisten Assembler auch erwartet AS genau einen Befehl pro Zeile (Leerzeilen sind natürlich auch zugelassen). Die Zeilen dürfen nicht länger als 255 Zeichen werden, darüber hinaus gehende Zeichen werden abgeschnitten.

Eine einzelne Zeile hat folgendes Format:

```
[Label[:]]<Befehl>[.Attribut] [Parameter[,Parameter..]] [;Kommentar]
```

Eine Zeile darf dabei auch über mehrere Zeilen in der Quelldatei verteilt sein, Folgezeichen (\) verketten diese Teile dann zu einer einzigen Zeile. Zu beachten ist allerdings, daß aufgrund der internen Pufferstruktur die Gesamtzeile nicht 256 Zeichen überschreiten darf. Zeilenangaben in Fehlermeldungen beziehen sich immer auf die letzte Zeile einer solchen zusammengesetzten Zeile.

Der Doppelpunkt nach dem Label ist optional, falls das Label in der ersten Spalte beginnt (woraus folgt, daß ein Befehl, sei es ein Maschinen- oder Pseudobefehl niemals in Spalte 1 beginnen darf). Man muß ihn aber setzen, falls das Label nicht in der ersten Spalte beginnt, damit AS es von einem Befehl unterscheiden kann. In letzterem Fall muß übrigens zwischen Doppelpunkt und dem Befehl mindestens ein Leerzeichen stehen, falls der eingestellte Zielprozessor zu denjenigen gehört, bei denen das Attribut auch eine mit einem Doppelpunkt abgetrennte Formatangabe sein darf. Diese Einschränkung ist aus Eindeutigkeits-Gründen nötig, da sonst keine Unterscheidung zwischen Befehl mit Format und Label mit Befehl möglich wäre.

Einige Signalprozessorreihen von Texas Instruments verwenden den für das Label vorgesehenen Platz wahlweise auch für einen Doppelstrich (||), der die parallele Ausführung mit der vorangehenden Instruktion anzeigt. Wenn diese beiden Instruktionen auf Maschinenebene in einem einzigen Wort vereinigt werden (C3x/C4x), macht ein zusätzliches Label vor der zweiten Anweisung natürlich keinen Sinn und ist auch nicht vorgesehen. Anders sieht es beim C6x mit seinen Instruktionspaketen variabler Länge aus: Wer dort (unschönerweise...) mitten in ein Paket hinein springen will, muß das Label dafür in eine Extrazeile davor setzen (das gleiche gilt übrigens auch für Bedingungen, die aber zusammen mit dem Doppelstrich in einer Zeile stehen dürfen).

Das Attribut wird von einer Reihe von Prozessoren benutzt, um Spezialisierungen oder Kodierungsvarianten eines bestimmten Befehls zu spezifizieren. Die bekannteste Nutzung des Attributs ist die Angabe der Operandengröße, wie z. B. bei der 680x0-Familie (Tabelle 2.4).

Da sich diese Anleitung nicht gleichzeitig als Handbuch für die von AS unterstützten Prozessorfamilien versteht, ist dies leider auch nicht der richtige Platz, um hier alle möglichen Attribute für alle unterstützten Familien aufzuzählen. Es sei aber angemerkt, daß i.a. nicht alle Befehle alle Attribute zulassen, andererseits das Fortlassen eines Attributs meist zur Verwendung der für diese Familie „natürlichen" Operandengröße führt. Zum genaueren Studium greife man auf ein Programmierhandbuch für die jeweilige Familie zurück, z.B. in [1] für die 68000er.

Bei TLCS-9000, H8/500 und M16(C) dient das Attribut sowohl der Angabe der Operandengröße, falls diese nicht durch die Operanden klar sein sollte, als auch der des zu verwendenden Befehlsformates. Dieses muß durch einen Doppelpunkt von der Operandengröße getrennt werden, z.B. so:

Attribut	arithmetisch-logischer Befehl	Sprungbefehl
B	Byte (8 Bit)	8-bit-displacement
W	Wort (16 Bit)	16-Bit-Displacement
L	Langwort (32 Bit)	16-Bit-Displacement
Q	Vierfachwort (64 Bit)	———
C	Half Precision (16 Bit)	———
S	Single Precision (32 Bit)	8-Bit-Displacement
D	Double Precision (64 Bit)	———
X	Extended Precision (80/96 Bit)	32-Bit-Displacement
P	Dezimalgleitkomma (80/96 Bit)	———

Tabelle 2.4: Erlaubte Attribute (Beispiel 680x0)

```
add.w:g   rw10,rw8
```

Was dieses Beispiel nicht zeigt, ist, daß die Formatangabe auch ohne Operandengröße geschrieben werden darf. Steht demgegenüber eine Operandengröße ohne Formatangabe, verwendet AS automatisch das kürzeste Format. Die erlaubten Befehlsformate und Operandengrößen sind vom Maschinenbefehl abhängig und können z.B. [181], [36], [67] bzw. [68] entnommen werden.

Die Zahl der Befehlsparameter ist abhängig vom Befehl und kann prinzipiell zwischen 0 und 20 liegen. Die Trennung der Parameter voneinander erfolgt ausschließlich durch Kommas (Ausnahme: DSP56xxx, dessen parallele Datentransfers durch Leerzeichen getrennt werden), wobei in Klammern oder Hochkommas eingeschlossene Kommas natürlich nicht beachtet werden.

Alles was hinter einem Semikolon steht, wird als Kommentar betrachtet und bei der Assemblierung nicht weiter ausgewertet. Je nachdem wo es in der Zeile steht, kann diese dann auch nur ein Label und/oder einen Befehl enthalten, oder gänzlich nur aus einem Kommentar bestehen. Für einzelne Targets existieren weitere Möglichkeiten, Kommentare als solche zu kennzeichnen:

- Bei Padauk-Controllern können Kommentare anstelle eines Semikolons auch mit einem doppelten Schrägstrich gekennzeichnet werden, wie man es von C++ kennt.

- Schaltet man die `EXTRACOMMENTS`-Option (siehe 3.2.14) bei 68xx-Targets ein, so wird der Rest der Zeile als Kommentar betrachtet, falls Label oder Mnemonic mit einem Stern beginnen.

- Ebenfalls bei eingeschalteter `EXTRACOMMENTS`-Option wird die Argumentliste durch ein Leerzeichen abgeschlossen und alles danach als Kommentar betrachtet.

Bei den Leerzeichen zur Trennung einzelnen Komponenten darf es sich genauso gut um Tabulatoren handeln.

2.6 Format des Listings

Das von AS bei Angabe der Kommandozeilenoptionen `l` oder `L` erzeugte Listing läßt sich grob in folgende Teile gliedern:

1. Ergänzte Wiedergabe des assemblierten Quellcodes;
2. Symbolliste;
3. Makroliste;
4. Funktionsliste;
5. Belegungsliste;
6. Querverweisliste.

Letztere beide werden nur erzeugt, wenn sie durch zusätzliche Kommandozeilenoptionen angefordert wurden.

Im ersten Teil listet AS den kompletten Inhalt aller Quelldateien inklusive des erzeugten Codes auf. Eine Zeile in diesem Listing hat dabei folgende Form:

```
[<n>] <Zeile>/<Adresse> <Code> <Quelle>
```

Im Feld n zeigt AS die Include-Verschachtelungstiefe an. Die Hauptdatei (die Datei, mit der die Assemblierung begann), hat dabei die Tiefe 0, von dort aus eingebundene Dateien haben Tiefe 1 usw. Die Tiefe 0 wird dabei nicht angezeigt: Für Zeilen der Hauptdatei wird dieses Feld durch eine passende Zahl von Leerzeichen ersetzt, oder gänzlich fortgelassen, falls es bisher gar keine Include-Anweisungen gegeben hat. Das 'Gedächtnis', ob es Include-Anweisungen gegeben hat, und bis zu welcher Tiefe, reicht über den einzelnen Pass hinaus. Auf diese Weise 'lernt' der Assembler im ersten Pass die maximale Tiefe und kann in folgenden Durchläufen dieses Feld mit durchgehend gleicher Breite ausgeben.

Im Feld `Zeile` wird die Zeilennummer bezogen auf die jeweilige Datei ausgegeben. Die erste Zeile einer Datei hat dabei Nummer 1. Die Adresse, an der der für diese Zeile erzeugte Code abgelegt wurde, folgt hinter dem Schrägstrich im Feld `Adresse`. Das dafür verwendete Zahlensystem wird durch den `listradix`-Schalter festgelegt (2.4), ebenso ob der Wert mit führenden Nullen angezeigt werden soll oder nicht. Das gerade aktuell verwendete Target mit der Größe seines Adreßraumes bestimmt die Breite des Feldes: Für einen Prozessor mit 64 KByte Adreßraum sind nur 4 Hex-Stellen erforderlich, bei 4 GByte dagegen acht.

Der erzeugte Code selber steht dahinter im Feld `Code`, ebenfalls im durch den List-Radix gegebenen Zahlensystem. Je nach Prozessortyp und aktuellem Segment bzw. Adreßraum können die Werte dabei mit unterschiedlicher Länge formatiert sein. Das können Bytes mit 8 Bits sein, aber auch Worte mit 16 oder 32 Bits, wenn der gerade genutzte Adreßraum nicht Byte-adressierbar ist, oder die Instruktionsworte des Prozessors länger als 8 Bit sind. Dies ist z.B. beim 68000 oder der PDP-11 der Fall: Deren Adreßraum ist Byte-weise organisiert, Instruktionsworte sind aber 16 Bit lang. Bei einigen wenigen Targets sind auch Längen möglich, die nicht ein mehrfaches von 8 Bit betragen, zum Beispiel 12 Bit bei der PDP-8.

Bei Vorwärtsreferenzen gibt es im ersten Durchlauf die Situation, dass der endgültige Code gar nicht angegeben werden kann. AS macht dann eine Annahme für den Wert, der die weitere Assemblierung nicht stört. Bei einer (zunehmenden) Zahl von Targets werden diese Felder im Code auch explizit mit Fragezeichen angezeigt. Die folgenden Targets unterstützen dies aktuell:

- 68xxx/Coldfire
- DSP56000/56300
- PowerPC
- 65xx/65xxx
- 6800/6805/68HC08/6809/68HC11/68HC12
- SH
- 8080/8085
- (e)Z80/Z180/Z280/Z380
- 8048
- 8051/80251

- 8096/80196/80296
- 8086/V20...V55
- AVR
- TMS370xxx
- uCOM-43
- CP-1600

Falls die Fragezeichen beim Weiterverarbeiten des Listings stören, känn diese Funktion über den Kommandozeilen-Schalter `-no-list-unknown-values` abgeschaltet werden.

Sollte mehr Code erzeugt worden sein, als in das Feld hineinpaßt, so werden im Anschluß an die Zeile weitere Zeilen erzeugt, in denen nur dieses Feld belegt ist.

Im Feld `Quelle` schlußendlich wird die Zeile aus der Quelldatei in ihrer Originalform ausgegeben.

Die Struktur der Daten vor der Quellcode-Zeile wird intern durch einen Formatstring gesteuert, der über die Kommandozeilen-Option `-listline-prefix` geändert werden kann. Dieser darf folgende Platzhalter enthalten:

- `%[c]i`: Die aktuelle Include-Verschachtelungstiefe, mit optionaler Feldbreite. Ohne Feldbreite wird dies auf gar nichts oder eine passende Menge Leerzeichen expandiert, falls die Tiefe Null ist.
- `%[c]n`: Die aktuelle Zeilennummer, mit einer optionalen Feldbreite, in der die Zeilennummer rechtsbündig ausgegeben wird. Der Default für die Feldbreite ist fünf Zeichen.
- `%[c]a`: Analog die aktuelle Speicheradresse, wobei der Default der Feldbreite der oben erwähnte, Target-abhängige Wert ist.

In allen drei Fällen bedeutet eine führende Null bei einer Feldbreite, daß links mit Nullen anstelle von Leerzeichen aufgefüllt werden soll. Der Default für diesen Formatstring ist `%i%n/%a`. Um eine Ausgabe zu erhalten, wie sie Versionen von AS vor 1.42 Build 249 erzeugten, kann als Formatstring `%1i%5n/%8a` verwendet werden.

Die Symboltabelle ist so ausgelegt, daß sie nach Möglichkeit immer in 80 Spalten dargestellt werden kann. Für Symbole ,,normaler Länge" wird eine zweispaltige Ausgabe gewählt. Sollten einzelne Symbole mit ihrem Wert die Grenze von 40 Spalten

überschreiten, werden sie in einer einzelnen Zeile ausgegeben. Die Ausgabe erfolgt in alphabetischer Reihenfolge. Symbole, die zwar definiert, aber nie benutzt wurden, werden mit einem vorangestellten Stern (*) gekennzeichnet.

Die bisher genannten Teile sowie die Auflistung aller definierten Makros / Funktionen lassen sich selektiv aus dem Gesamtlisting ein-und ausblenden, und zwar mit dem bereits erwähnten `t`-Kommandozeilenschalter. Intern existiert in AS ein Byte, dessen Bits repräsentieren, welche Teile ausgegeben werden sollen. Die Zuordnung von Bits zu den Teilen ist in Tabelle 2.5 aufgelistet.

Defaultmäßig sind alle Bits auf 1 gesetzt, bei Verwendung des Schalters

```
-t <Maske>
```

werden die in `<Maske>` gesetzten Bits gelöscht, so daß die entsprechenden Listing-Teile unterdrückt werden. Analog lassen sich mit einem Pluszeichen einzelne Teile wieder einschalten, falls man es in der `ASCMD`-Variablen übertrieben hat...will man z.B. nur die Symboltabelle haben, so reicht

```
-t 2  .
```

In der Belegungsliste werden für jedes Segment einzeln die belegten Bereiche hexadezimal ausgegeben. Handelt es sich bei einem Bereich um eine einzige Adresse, wird nur diese ausgegeben, ansonsten erste und letzte Adresse.

In der Querverweisliste wird für jedes definierte Symbol in alphabetischer Reihenfolge eine Ausgabe folgender Form erzeugt:

```
Symbol <Symbolname> (=<Wert>,<Datei>/<Zeile>):
 Datei <Datei 1>:
 <n1>[(m1)]  ..... <nk>[(mk)]
 .
 .
 Datei <Datei l>:
 <n1>[(m1)]  ..... <nk>[(mk)]
```

Für jedes Symbol wird aufgelistet, in welchen Dateien es in welchen Zeilen angesprochen wurde. Sollte ein Symbol mehrmals in der gleichen Zeile benutzt worden sein, so wird dies durch eine in Klammern gesetzte Anzahl hinter der Zeilennummer angedeutet. Sollte ein Symbol niemals benutzt worden sein, erscheint es auch nicht in der Liste; entsprechend erscheint eine Datei auch überhaupt nicht in der Liste eines Symbols, falls es in der entsprechenden Datei nicht referenziert wurde.

ACHTUNG! AS kann dieses Listing nur dann korrekt aufs Papier bringen, wenn man ihm vorher die Länge und Breite des Ausgabemediums mit Hilfe des `PAGE`-Befehls (siehe 3.7.1) mitgeteilt hat! Der voreingestellte Default sind 60 Zeilen und eine unbegrenzte Zeilenbreite.

2.7 Symbolkonventionen

Symbole dürfen zwar (wie in der Einleitung bereits angedeutet) bis zu 255 Zeichen lang werden und werden auch auf der ganzen Länge unterschieden, die Symbolnamen müssen aber einigen Konventionen genügen:

Symbolnamen dürfen aus einer beliebigen Kombination von Buchstaben, Ziffern, Unterstrichen und Punkten bestehen, wobei das erste Zeichen keine Ziffer sein darf. Der Punkt wurde nur zugelassen, um der MCS-51-Notation von Registerbits zu genügen, und sollte möglichst nicht in eigenen Symbolnamen verwendet werden. Zur Segmentierung von Symbolnamen sollte auf jeden Fall der Unterstrich und nicht der Punkt verwendet werden.

Defaultmäßig ist AS nicht case-sensitiv, es ist also egal, ob man Groß-oder Kleinbuchstaben verwendet. Mittels des Kommandozeilenschalters `U` läßt sich AS jedoch in einen Modus umschalten, in dem Groß- und Kleinschreibung unterschieden wird. Ob AS umgeschaltet wurde, kann mit dem vordefinierten Symbol `CASESENSITIVE` ermittelt werden: TRUE bedeutet Unterscheidung, FALSE keine.

Tabelle 2.6 zeigt die wichtigsten, von AS vordefinierten Symbole. **VORSICHT!** Während es im case-insensitiven Modus egal ist, mit welcher Kombination von Groß- und Kleinbuchstaben man vordefinierte Symbole anspricht, muß man sich im case-sensitiven Modus exakt an die oben angegebene Schreibweise (nur Großbuchstaben) halten!

Zusätzlich definieren einige Pseudobefehle noch Symbole, die eine Abfrage des damit momentan eingestellten Wertes ermöglichen. Deren Beschreibung findet sich bei den zugehörigen Befehlen.

Auf den meisten Plattformen ist der Name `INF` als Sonderwert für Unendlich im Gleitkomma-Format reserviert und kann nicht für eigene Symbole verwendet werden.

Ein etwas verstecktes (und mit Vorsicht zu nutzendes) Feature ist, Symbolnamen aus String-Variablen zusammenzubauen, indem man den Namen des Strings mit geschweiften Klammern in den Symbolnamen einbaut. So kann man z.B. den Namen eines Symbols anhand des Wertes eines anderen Symbols festlegen:

```
cnt  set cnt+1
temp  equ "\{CNT}"
  jnz skip{temp}
  .
  .
skip{temp}: nop
```

Bit	Teil
0	Quelldatei(en)+erzeugter Code
1	Symboltabelle
2	Makroliste
3	Funktionsliste
4	Zeilennumerierung
5	Registersymboltabelle
7	Zeichentabellenliste

Tabelle 2.5: Zuordnung der Bits zu den Listingkomponenten

Name	Bedeutung
`TRUE`	logisch „wahr”
`FALSE`	logisch „falsch”
`CONSTPI`	Kreiszahl Pi (3.1415.....)
`INTWIDTH`	Bitbreite interne Integer-Arithmetik
`HAS64`	interne Integer-Arithmetik verwendet mindestens 64 Bit
`FLOATMAX`	größte darstellbare Gleitkommazahl
`VERSION`	Version von AS in BCD-Kodierung, z.B. 1331 hex für Version 1.33p1
`ARCHITECTURE`	Zielplattform, für die AS übersetzt wurde, in der Form Prozessor-Hersteller-Betriebssystem
`DATE`	Datum und
`TIME`	Zeitpunkt der Assemblierung (Beginn)
`MOMCPU`	momentan gesetzte Ziel-CPU
`MOMCPUNAME`	dito, nur als voll ausgeschriebener String
`MOMFILE`	augenblickliche Quelldatei
`MOMLINE`	Zeilennummer in Quelldatei
`MOMPASS`	Nummer das laufenden Durchgangs
`MOMSECTION`	Name der aktuellen Sektion oder Leerstring
`MOMSEGMENT`	Name des mit `SEGMENT` gewählten Adreßraumes
`*`, $, . bzw. `PC`	mom. Programmzähler

Tabelle 2.6: Vordefinierte Symbole

ACHTUNG! Der Programmierer ist selber dafür verantwortlich, daß sich dabei gültige Symbolnamen ergeben!

Eine vollständige Auflistung aller von AS verwendeten Symbolnamen findet sich in Anhang F.

Neben seinem Wert besitzt auch jedes Symbol eine Markierung, zu welchen *Segment* es gehört. In erster Linie wird eine solche Unterscheidung bei Prozessoren benötigt, die mehrere Adreßräume besitzen. AS kann mit dieser Zusatzinformation bei Zugriffen über ein Symbol warnen, wenn ein für diesen Adreßraum ungeeigneter Befehl verwendet wird. Ein Segmentattribut wird einem Symbol automatisch angehängt, wenn es als Label oder mit einem Spezialbefehl (z.B. `BIT`) definiert wird; ein mit dem ,,Universalbefehl" `SET` oder `EQU` definiertes Symbol ist jedoch ,,typenlos", d.h. seine Verwendung wird niemals Warnungen auslösen. Das Segmentattribut eines Symbols kann mit der eingebauten Funktion `SYMTYPE` abgefragt werden, etwa so:

```
Label:
        .
        .
Attr    equ     symtype(Label)  ; ergibt 1
```

Den einzelnen Segmenttypen sind die in Tabelle 2.7 aufgelisteten Nummern zugeordnet. Die aus der Ordnung normaler Symbole etwas herausfallenden Registersymbole sind näher in Abschnitt 2.11 erläutert. Mit einem undefinierten Symbol als Argument liefert die `SYMTYPE`-Funktion -1 als Ergebnis.

Ob ein Symbol definiert ist, kann mit den Funktionen `DEFINED` bzw. `SYMEXIST` geprüft werden. Der Unterschied: `SYMEXIST` liefert bei einer Vorwärtsdeklaration im ersten Durchlauf ein 'false' und bei weiteren Durchläufen ein 'true'. Im Gegensatz dazu liefert `DEFINED` nur dann ein 'true', wenn das Symbol im bis zu diesem Punkt assemblierten Code definiert wurde. Bei Vorwärtsdeklarationen bleibt es also bei einem 'false', unabhängig vom Durchlauf.

Mittels der Funktion `SYMUSED` kann abgefragt werden, ob ein Symbol bisher schon einmal verwendet wurde. Dies schließt auch Bezüge auf ein Symbol ein, die *vor* seiner Definition erfolgen, falls eine der folgenden Bedingungen erfüllt ist:

- Sowohl die Vorwärtsreferenz als auch die Definition erfolgen im globalen Symbol-Namensraum, d.h. außerhalb von Sektionen oder expandierten Makro-Rümpfen.

- Die Vorwärtsreferenz erfolgt in einer Sektion, aber außerhalb eines expandierten Makro-Rumpfes, und die Definition erfolgt ebenfalls außerhalb eines expandierten Makro-Rumpfes, in der selben Sektion, oder einer Eltern-Sektion dieser Sektion, oder global.
- Die Vorwärtsreferenz erfolgt innerhalb eines expandierten Makro-Rumpfes, und die Definition im selben Makro-Rumpf.

Falls ein Symbol auf Benutzung geprüft werden soll und es Vorwärtsreferenzen gibt, die keine dieser drei Situationen erfüllt, bleibt die Möglichkeit, ein Symbol vor der ersten Vorwärtsreferenz mittels des `FORWARD`-Befehls anzukündigen. Es wird dann quasi ein 'Leereintrag' in der Symboltabelle angelegt, an dem Vorwärtsreferenzen explizit vermerkt werden können.

`DEFINED` akzeptiert nicht nur einen einfachen Symbolnamen, sondern auch einen kompletten Formelausdruck. Der Rückgabewert ist in so einem Fall nur dann 'true', falls der Ausdruck keine Symbole enthält, die nach der im vorigen Absatz gegebenen Erklärung undefiniert sind.

Segment	Rückgabewert
<keines>	0
CODE	1
DATA	2
IDATA	3
XDATA	4
YDATA	5
BITDATA	6
IO	7
REG	8
ROMDATA	9
EEDATA	10
<Registersymbol>	128

Tabelle 2.7: Rückgabewerte der `SYMTYPE`-Funktion

2.8 Temporäre Symbole

Besonders bei Programmen mit vielen aufeinander folgenden Schleifen oder IF-artigen Strukturen steht man immer wieder vor dem Problem, sich ständig neue

Namen für Labels ausdenken zu müssen. Dabei weiß man an sich genau, daß man dieses Label nie wieder brauchen wird und am liebsten in irgendeiner Weise 'verwerfen' möchte. Eine einfache Lösung, wenn man nicht gleich den großen Hammer des Sektionskonzeptes (siehe Kapitel 3.8) schwingen möchte, sind *temporäre* Symbole, die solange ihre Gültigkeit behalten, bis ein neues, nicht-temporäres Symbol definiert wird. Andere Assembler bieten einen ähnlichen Mechanismus an, der dort unter dem Stichwort 'lokale Symbole' läuft, zur besseren Abgrenzung gegen das Sektionskonzept möchte ich aber beim Begriff 'temporäre Symbole' bleiben. AS kennt drei unterschiedliche Typen von temporären Symbolen, um jedem 'Umsteiger' ein Konzept anzubieten, das den Umstieg so einfach wie möglich macht. Leider kocht quasi jeder Assembler bei diesem Thema sein eigenes Süppchen, so daß es nur in Ausnahmefällen eine 1:1-Lösung für existierenden Code geben wird:

2.8.1 Temporäre Symbole mit Namen

Ein Symbol, dessen Name mit zwei Dollarzeichen beginnt (dies ist weder für normale Symbole noch Konstanten zulässig), ist ein temporäres Symbol mit Namen. AS führt intern einen Zähler mit, der zu Beginn der Assemblierung auf Null gesetzt wird und bei jeder Definition eines nicht-temporären Symbols inkrementiert wird. Wird ein temporäres Symbol definiert oder referenziert, so werden die beiden führenden Dollarzeichen gestrichen und der momentane Stand des Zählers wird angehängt. Auf diese Weise erhält man mit jedem nicht-temporären Symbol sozusagen die Symbolnamen zurück - man kommt an die Symbole vor dieser Definition aber auch nicht mehr heran! Temporäre Symbole bieten sich daher besonders für den Einsatz in kleinen Anweisungsblöcken an, typischerweise etwa ein Dutzend Befehle, auf keinen Fall mehr als eine Bildschirmseite, sonst kommt man leicht durcheinander...

Hier ein kleines Beispiel:

```
$$loop: nop
        dbra    d0,$$loop

split:

$$loop: nop
        dbra    d0,$$loop
```

Wäre das zwischen den Schleifen liegende nicht-temporäre Label nicht vorhanden, gäbe es natürlich eine Fehlermeldung wegen eines doppelt definierten Symbols.

Namenlose Temporäre Symbole

Für all jene, denen temporäre Symbole mit Namen noch immer zu kompliziert sind, gibt es eine noch einfachere Variante: Setzt man als Label ein einfaches Plus- oder Minuszeichen, so werden diese in die Namen `__forwnn` bzw. `__backmm` umgesetzt, wobei `nn` bzw. `mm` von Null an laufende Zähler sind. Referenziert werden diese Symbole über die Sonderwerte `- -- ---` bzw. `+ ++ +++`, womit sich die drei letzten 'Minussymbole' bzw die drei nächsten 'Plussymbole' referenzieren lassen. Welche Variante man benutzt, hängt also davon ab, ob man ein Symbol vorwärts oder rückwärts referenzieren will.

Bei der *Definition* namenloser temporärer Symbole gibt es neben dem Plus- und Minuszeichen noch eine dritte Variante, nämlich einen Schrägstrich (/). Ein so definiertes temporäres Symbol kann gleichermaßen vorwärts wie rückwärts referenziert werden; d. h. je nach Referenzierung wird es wie ein Minus oder Plus behandelt.

Namenlose temporäre Symbole finden ihre Anwendung üblicherweise in Konstruktionen, die auf eine Bildschirmseite passen, wie das bedingte Überspringen von ein paar Maschinenbefehlen oder kleinen Schleifen - ansonsten würde die Sache zu unübersichtlich werden (das ist aber nur ein gut gemeinter Rat...). Ein Beispiel dafür ist das folgende Stück Code, zur Abwechslung mal als 65xx-Code:

```
        cpu     6502

-       ldx     #00
-       dex
        bne     -               ; springe zu 'dex'
        lda     RealSymbol
        beq     +               ; springe zu 'bne --'
        jsr     SomeRtn
        iny
+       bne     --              ; springe zu 'ldx #00'

SomeRtn:
        rts

RealSymbol:
        dfs     1

   inc ptr
    bne  +              ; springe zu 'tax'
    inc  ptr+1
+  tax
```

```
  bpl  ++          ; springe zu 'dex'
    beq  +           ; springe vorwaerts zu 'rts'
    lda  #0
/   rts                ; Schraegstrich = Wildcard
+  dex
    beq  -           ; springe rueckwaerts zu 'rts'

ptr: dfs 2
```

2.8.2 Zusammengesetzte temporäre Symbole

Dies ist vielleicht der Typ von temporären Symbolen, der dem Konzept von lokalen Symbolen und Sektionen am nächsten kommt. Wann immer der Name eines Symboles mit einem Punkt (.) anfängt, wird das Symbol nicht mit diesem Namen in der Symboltabelle abgelegt. Stattdessen wird der Name des zuletzt definierten Symbols ohne vorangestellten Punkt davor gehängt. Auf diese Weise nehmen Symbole, deren Name nicht mit einem Punkt anfängt, quasi die Rolle von 'Bereichsgrenzen' ein und Symbole, deren Name mit einem Punkt anfängt, können in jedem Bereich neu verwendet werden. Sehen wir uns das folgende kurze Beispiel an:

```
proc1:                      ; nicht-temporaeres Symbol 'proc1'

.loop   moveq   #20,d0      ; definiert in Wirklichkeit 'proc1.loop'
        dbra    d0,.loop
        rts

proc2:                      ; nicht-temporaeres Symbol 'proc2'

.loop   moveq   #10,d1      ; definiert in Wirklichkeit 'proc2.loop'
        jsr     proc1
        dbra    d1,.loop
        rts
```

Man beachte, daß es weiterhin möglich ist, auf alle temporären Symbole zuzugreifen, auch wenn man sich nicht im gleichen 'Bereich' befindet, indem man einfach den zusammengesetzten Namen benutzt (wie z.B. 'proc2.loop' im voranstehenden Beispiel).

Zusammengesetzte Symbole lassen sich prinzipiell mit Sektionen kombinieren und können so auch zu lokalen Symbolen werden. Man beachte allerdings, daß das

zuletzt definierte, nicht temporäre Symbol nicht pro Sektion gespeichert wird, sondern lediglich global. Das kann sich aber auch irgendwann einmal ändern, man sollte sich also nicht auf das augenblickliche Verhalten verlassen.

2.9 Formelausdrücke

An den meisten Stellen, an denen der Assembler Zahlenangaben erwartet, können nicht nur einfache Symbole oder Konstanten angegeben werden, sondern ganze Formelausdrücke. Bei den Komponenten der Formelausdrücke kann es sich sowohl um ein einzelnes Symbol als auch um eine Konstante handeln. Konstanten dürfen entweder Integer-, Gleitkomma-, oder Stringkonstanten sein.

2.9.1 Integer-Konstanten

Integer-Konstanten bezeichnen ganze Zahlen. Sie werden als eine Folge von Ziffern geschrieben. Dies kann in verschiedenen Zahlensystemen erfolgen, deren Notation von verwendeten Zielprozessor abhängt (Tabelle 2.8).

	Intel-Modus	Motorola-Modus	C-Modus	IBM-Modus
dezimal	direkt	direkt	direkt	direkt
hexadezimal	Suffix H	Präfix $	Präfix 0x	X'..' oder H'..'
Kennung	`hexh`	`$hex`	`0xhex`	`x'hex'` `h'hex'`
binär	Suffix B	Präfix %	Präfix 0b	B'..'
Kennung	`binb`	`%bin`	`0bbin`	`b'bin'`
oktal	Suffix O oder Q	Präfix @	Präfix 0	O'..'
Kennung	`octo` `octq`	`@oct`	`0oct`	`o'oct'`
ASCII				A'..'
Kennung				`a'asc'`

Tabelle 2.8: definierte Zahlensysteme und Schreibweisen

Falls das Zahlensystem nicht explizit durch vor-oder nachgestellte Zeichen vorgegeben wird, wird die Basis folgendermaßen bestimmt:

- Falls ein `RADIX`-Befehl gegeben wurde, benutze die dadurch definierte Basis, ansonsten:

- Falls ein `-radix` Kommandozeilenschalter gegeben wurde, benutze die dadurch definierte Basis, ansonsten:
- Verwende das Dezimalsystem (Basis 10).

Sowohl mit dem`RADIX`-Befehl als auch mit dem `-radix`-Kommandozeilenschalter lassen sich auch ,,ungewöhnliche" Zahlensysteme, d.h. andere als 2, 8, 10 oder 16 einstellen.

Gültige Ziffern sind die Zahlen 0 bis 9 sowie die Buchstaben A bis Z (Wert 10 bis 35) bis zur Basis des Zahlensystems minus eins. Die ASCII- Darstellung fällt etwas aus diesem System heraus: hier beschreibt der ASCII-Wert (bzw. der Code im aktuell eingestellten Zeichensatz, siehe Abschnitt 3.1.12) ein ganzes Byte. Solcherart geschriebene Integer-Konstanten sind also letzten Endes identisch zu Multi-Character-Konstanten. Die beiden Schreibweisen:

```
'ABCD'
A'ABCD'
```

sind also identisch, das vorangestellte 'A' ist eigentlich redundant. Man kann diese Schreibweise aber für existierenden Code erlauben, weil einige ältere Original-Assembler (z.B. Signetics 2650) diese Schreibweise unterstützen.

Multi-Character-Konstanten sind bei AS grundsätzlich immer Big-Endian, d.h. 'ABCD' ergibt den Zahlenwert 0x41424344. Warum das so ist? Naja, das erste AS-Target war der Motorola 68000, und es hat sich nie jemand darüber beklagt...die einzige Ausnahme von dieser Regel ist die PDP-11 (und der einen LSI-11 nutzende WD16): zwecks besserer Kompatibilität zum MACRO-11 von DEC sind solche Konstanten ausnahmsweise Little-Endian, 'AB' ergibt also den Zahlenwert 0x4241.

Die Verwendung von Buchstaben in Integer-Konstanten bringt allerdings auch einige Mehrdeutigkeiten mit sich, da Symbolnamen ja auch Ketten aus Zahlen und Buchstaben sind: Ein Symbolname darf nicht mit einem Zeichen von 0 bis 9 beginnen, was bedeutet, daß eine Integer-Konstante, die nicht durch ein anderes Sonderzeichen eindeutig als solche erkennbar ist, niemals mit einem Buchstaben beginnen darf; notfalls muß man eine eigentlich überflüssige Null voranstellen. Der bekannteste Fall ist das Schreiben von Hexadezimalkonstanten im Intel-Modus: Ist die vorderste Stelle zwischen A und F, so hilft das hintan gestellte H überhaupt nichts, es muß noch eine Null davor (statt F0H also 0F0H). Die Motorola-oder C-Syntax, die beide das Zahlensystem am Anfang einer Integer-Konstante kennzeichnen, kennen dieses Problem nicht.

Reichlich heimtückisch ist auch, daß bei immer höheren, mit `RADIX` eingestellten Zahlensystemen, die bei Intel- und C-Syntax benutzten Buchstaben zur Zahlensystemkennung immer weiter ,,aufgefressen" werden; so kann man z.B. nach `RADIX 16` keine binären Konstanten mehr schreiben, und ab `RADIX 18` in Intel-Syntax auch keine hexadezimalen Konstanten mehr. Also **VORSICHT!**

Welche Syntax für welchen Zielprozessor im Default verwendet wird, ist im Anhang E gesammelt. Unabhängig von diesem Default ist es möglich, einzelne Schreibweisen mit dem `INTSYNTAX`-Befehl (siehe Abschnitt 3.9.6) weg- oder dazuzuschalten. Die unter *Ident* aufgeführten Namen, mit einem vorangestellten Plus- oder Minuszeichen, dienen bei diesem Befehl als Argumente.

Eine Art "Generalschalter' für Integer-Schreibweisen ist der `RELAXED`-Befehl (siehe Abschnitt 3.9.7): Unabhängig vom Zielprozessor können beliebige Schreibweisen verwendet werden kann (auf Kosten der Kompatibilität zu Standard-Assemblern).

Mit `INTSYNTAX` bzw. `RELAXED` eröffnet sich insbesondere die 'IBM-Schreibweise', wie man sie bei manchen Fremdassemblern antrifft, für alle Ziele:

Bei dieser Schreibweise wird der eigentliche Wert in Hochkommas geschrieben und das Zahlensystem ('x' oder 'h' für hexadezimal, 'o' für oktal und 'b' für binär) direkt davor. Die Integer-Konstante 305419896 kann damit also folgendermaßen geschrieben werden:

```
x'12345678'
h'12345678'
o'2215053170'
b'00010010001101000101011001111000'
```

Als weitere Variante dieser Schreibweise erlauben einige wenige Targets das Fortlassen des schließenden Hochkommas, zwecks Kompatibiltät zu Code, der für andere Assembler geschrieben wurde. Für eigenen, neu erstellten Code wird von der Verwendung abgeraten.

2.9.2 Gleitkommakonstanten

Gleitkommazahlen werden in der üblichen halblogarithmischen Schreibweise geschrieben, die in der allgemeinsten Form

```
[-]<Vorkommastellen>[.Nachkommastellen][E[-]Exponent]
```

lautet. **ACHTUNG!** Der Assembler versucht eine Konstante zuerst als Integer-Konstante zu verstehen und macht erst dann einen Versuch mit Gleitkomma, falls dies gescheitert ist. Will man aus irgendwelchen Gründen die Auswertung als Gleitkommazahl erzwingen, so kann man dies durch Dummy-Nachkommastellen erreichen, z.B. `2.0` anstelle `2`.

2.9.3 Stringkonstanten

Stringkonstanten müssen in einfache oder doppelte Hochkommas eingeschlossen werden. Um diese selber und andere Sonderzeichen ohne Verrenkungen in String-Konstanten einbauen zu können, wurde ein ,,Escape-Mechanismus" eingebaut, der C-Programmierer*innen bekannt vorkommen dürfte:

Schreibt man einen Backslash mit einer maximal dreiziffrigen Zahl im String, so versteht der Assembler dies als Zeichen mit dem entsprechenden dezimalen ASCII-Wert. Alternativ kann der Zahlenwert auch hexadezimal oder oktal mit einem vorangestellten x oder einer vorangestellten 0 geschrieben werden. Für die hexadezimale Schreibweise reduziert sich die Maximalanzahl von Stellen auf 2. So kann man z.B. mit `\3` ein ETX-Zeichen definieren. Vorsicht allerdings mit der Definition von NUL-Zeichen! Da die C-Version von AS momentan intern zur Speicherung von String-Symbolen C-Strings benutzt (die durch NUL-Zeichen terminiert werden), sind NUL-Zeichen in Strings momentan nicht portabel!

NIX

Einige besonders häufig gebrauchte Steuerzeichen kann man auch mit folgenden Abkürzungen erreichen:

`\b` : Backspace	`\a` : Klingel	`\e` : Escape
`\t` : Tabulator	`\n` : Zeilenvorschub	`\r` : Wagenrücklauf
`\\` : Backslash	`\'` oder `\h` : Hochkomma	
`\"` oder `\i` : Gänsefüßchen		

Die Kennbuchstaben dürfen sowohl groß als auch klein geschrieben werden.

Über dieses Escape-Zeichen können sogar Formelausdrücke in den String eingebaut werden, wenn sie in geschweifte Klammern eingefaßt werden: z.B. ergibt

```
message "Wurzel aus 81 : \{sqrt(81)}"
```

die Ausgabe

```
Wurzel aus 81 : 9
```

Der Assembler wählt anhand des Formelergebnistyps die richtige Ausgabeform, zu vermeiden sind lediglich weitere Stringkonstanten im Ausdruck, da der Assembler bei der Groß-zu-Kleinbuchstabenumwandlung sonst durcheinander kommt. Integer-Ausdrücke werden defaultmäßig hexadezimal ausgegeben, dies läßt sich jedoch mit dem `OUTRADIX`-Befehl ändern.

Bis auf den Einbau von Formelausdrücken ist dieser Escape-Mechanismus auch in als ASCII definierten Integer-Konstanten zulässig, z.B. so:

```
move.b   #'\n',d0
```

Jedoch hat alles seine Grenzen, weil der darüber liegende Splitter, der die Zeile in Opcode und Parameter zerlegt, nicht weiß, womit er da eigentlich arbeitet, z.B. hier:

```
move.l   #'\'abc',d0
```

Nach dem dritten Hochkomma findet er das Komma nicht mehr, weil er vermutet, daß eine weitere Zeichenkonstante beginnt, und eine Fehlermeldung über eine falsche Parameterzahl ist die Folge. Abhilfe wäre z.B., `\h` anstelle `\'` zu schreiben.

2.9.4 String- zu Integerwandlung und Zeichenkonstanten

Frühere Versionen von AS verfolgten eine strikte Trennung von Strings und sogenannten "Zeichenkonstanten": Eine Zeichenkonstante sieht auf den ersten Blick aus wie ein String, nur sind die Zeichen in einfache Hochkommas statt doppelte eingeschlossen. Ein solches Objekt hatte den Datentyp 'Integer', war also eine Zahl, deren Wert durch den (ASCII-)Wert des jeweiligen Zeichens definiert war, und wurde strikt von einer String-Konstante unterschieden:

```
move.b   #65,d0
move.b   #'A',d0       ; gleichwertig
move.b   #"A",d0       ; nicht erlaubt in aelteren Versionen!
```

Diese Unterscheidung existiert *nicht mehr*, es ist also egal, ob man einfache oder doppelte Hochkommas verwendet. Wird an einer Stelle eine Zahl als Argument erwartet, und ein String verwendet, so erfolgt die Umwandlung anhand der (ASCII-)Werte "on-the-fly" an dieser Stelle. Im obigen Beispiel würden alle drei Anweisungen den gleichen Maschinencode erzeugen.

Eine solche implizite Wandlung findet auch für aus mehreren Zeichen bestehende Strings statt, die dann bisweilen als "Mehrzeichenkonstanten" bezeichnet werden:

```
'A'    == $41
"AB"   == $4142
'ABCD' == $41424344
```

Mehrzeichenkonstanten sind der einzige Fall, in denen die Verwendung von einfachen oder doppelten Hochkommas noch einen Unterschied macht. Für viele Zielprozessoren sind Pseudobefehle zur Ablage von Konstanten definiert, die als Argument verschiedene Datentypen akzeptieren. Will man wirklich eine Zeichenkette haben, so muß man in diesem Fall weiterhin doppelte Hochkommas verwenden:

```
dc.w    "ab"  ; legt zwei Worte (0x0041,0x0042) ab
dc.w    'ab'  ; legt ein Wort (0x4142) ab
```

Wichtig: dies ist nicht erforderlich, wenn die Zeichenkette länger als die verwendete Operandegröße ist, in diesem Beispiel also länger als zwei Zeichen bzw. 16 Bit.

2.9.5 Evaluierung

Die Berechnung von im Formelausdruck entstehenden Zwischenergebnissen erfolgt immer mit der höchsten auf dem Host-System verfügbaren Wortbreite. Bei Ganzzahlen sind dies 32, 64 oder 128 Bit, und bei Gleitkommazahlen ca. $+/-1.8 * 10^{308}$ (IEEE double precision) oder ca. $+/-1.1 * 10^{4932}$ (x86/M68K extended precision). Eine eventuelle Prüfung auf Wertebereichsüberschreitungen findet erst am Endergebnis statt.

Die interne Bitlänge von Ganzzahlen kann aus dem vordefinierten Symbol `INTWIDTH` ausgelesen werden. Die größtmögliche Gleitkommazahl ist in dem ebenso vordefinierten Symbol `FLOATMAX` hinterlegt.

Optional ist es möglich, Ganzzahlen als Attribut eine reduzierte Bitbreite zu geben. Diese ergibt sich bei der Definition als Binär-, Oktal- oder Hexadezimal-Konstante aus der verwendeten Zahl von Stellen. Aktuell wird dieses Attribut nur bei der Anwendung des Einerkomplements beachtet. Der entsprechende Operator wird dann nur auf die reduzierte Anzahl von Bits angewendet. Diese Betriebsart kann mit dem Kommandzeilen-Schalter `-fred` eingeschaltet werden.

2.9.6 Operatoren

Der Assembler stellt zur Verknüpfung die in Tabelle 2.9 genannten Operanden zur Verfügung. Unter „Rang" ist dabei die Priorität zu verstehen, die dieser Operator bei der Teilung eines Ausdruckes in Unterausdrücke hat, der ranghöchste Operator wird also *zuletzt* ausgewertet. Die Reihenfolge der Evaluierung läßt sich durch Klammerung neu festlegen.

Die Vergleichsoperatoren liefern TRUE, falls die Bedingung zutrifft, und FALSE falls nicht. Vergleiche betrachten Integerzahlen dabei als 32 oder 64 Bit breit (je nach Host-System) und vorzeichenbehaftet. Für die logischen Operatoren ist ein Ausdruck TRUE, falls er ungleich 0 ist, ansonsten FALSE.

Grundsätzlich ist bei Operatoren mit zwei Argumenten nicht definiert, in welcher Reihenfolge die Operanden bestimmt werden. Die einzige Ausnahme von dieser

Op.	Funktion	#Ops.	Int	Float	String	Register	Rang
<>	Ungleichheit	2	ja	ja	ja	ja	14
! =	Alias für <>						
>=	größer o. gleich	2	ja	ja	ja	ja	14
<=	kleiner o. gleich	2	ja	ja	ja	ja	14
<	echt kleiner	2	ja	ja	ja	ja	14
>	echt größer	2	ja	ja	ja	ja	14
=	Gleichheit	2	ja	ja	ja	ja	14
==	Alias für =						
!!	log. XOR	2	ja	nein	nein	nein	13
||	log. OR	2	ja	nein	nein	nein	12
&&	log. AND	2	ja	nein	nein	nein	11
~~	log. NOT	1	ja	nein	nein	nein	2
-	Differenz	2	ja	ja	nein	nein	10
+	Summe	2	ja	ja	ja	nein	10
#	Modulodivision	2	ja	nein	nein	nein	9
/	Quotient	2	ja*)	ja	nein	nein	9
*	Produkt	2	ja	ja	nein	nein	9
^	Potenz	2	ja	ja	nein	nein	8
!	binäres XOR	2	ja	nein	nein	nein	7
|	binäres OR	2	ja	nein	nein	nein	6
&	binäres AND	2	ja	nein	nein	nein	5
><	Bitspiegelung	2	ja	nein	nein	nein	4
>>	log. Rechtsschieben	2	ja	nein	nein	nein	3
<<	log. Linksschieben	2	ja	nein	nein	nein	3
~	binäres NOT	1	ja	nein	nein	nein	1
*) Rest wird verworfen							

Tabelle 2.9: in AS definierte Operatoren

Regel sind das logische AND und OR: Falls das Ergebnis bereits durch den linken Operanden eindeutig bestimmt ist, wird der rechte Operand nicht mehr ausgewertet,

Bei Vergleichen von Register-Symbolen sind zwei Details zu beachten. Zum einen sind zwei Register-Symbole dann gleich, wenn sie auf das gleiche Register verweisen. Da Register bei einigen Prozessoren Alias-Namen haben, werden diese Aliase als gleich betrachtet. Zum Beispiel ist das Register `A7` auf einem 68000 auch als `SP` ansprechbar, und diese beiden Registersymbole sind gleich. Zum anderen haben manche Prozessoren mehrere Sätze von Registern, auf einem 68040 zum Beispiel die normalen (Integer-)Register und die FPU-Register. Zwischen zwei Registern aus unterschiedlichen Gruppen läßt sich keine kleiner/größer-Beziehung angeben, die entsprechenden Operatoren liefern immer ein FALSE zurück. Lediglich ein Test auf Gleichheit oder Ungleichheit ergibt einen Sinn.

Die Bitspiegelung ist wohl etwas erklärungsbedürftig: Der Operator spiegelt die untersten Bits im ersten Operanden, läßt die darüber liegenden Bits aber unverändert. Die Zahl der zu spiegelnden Bits ist der rechte Operand und darf zwischen 1 und 32 liegen.

Eine keine Fußangel beim binären Komplement: Da die Berechnung grundsätzlich auf 32- oder 64-Bit-Ebene erfolgt, ergibt seine Anwendung auf z.B. 8-Bit-Masken üblicherweise Werte, die durch voranstehende Einsen nicht mehr im entferntesten in 8-Bit-Zahlen hineinpassen. Eine binäre UND-Verknüpfung mit einer passenden Maske ist daher unvermeidlich!

2.9.7 Funktionen

Zusätzlich zu den Operatoren definiert AS noch eine Reihe in erster Linie transzendenter Funktionen mit Gleitkomma-Argument, die Tabellen 2.10 und 2.11 auflisten.

Name	Funktion	Argument	Ergebnis
SQRT	Quadratwurzel	$arg \geq 0$	Gleitkomma
SIN	Sinus	$arg \in \mathbb{R}$	Gleitkomma
COS	Kosinus	$arg \in \mathbb{R}$	Gleitkomma
TAN	Tangens	$arg \neq (2 * n + 1) * \frac{\pi}{2}$	Gleitkomma
COT	Kotangens	$arg \neq n * \pi$	Gleitkomma
ASIN	inverser Sinus	$\mid arg \mid \leq 1$	Gleitkomma
ACOS	inverser Kosinus	$\mid arg \mid \leq 1$	Gleitkomma
ATAN	inverser Tangens	$arg \in \mathbb{R}$	Gleitkomma

ACOT	inverser Kotangens	$arg \in \mathbb{R}$	Gleitkomma
EXP	Exponentialfunktion	$arg \in \mathbb{R}$	Gleitkomma
ALOG	10 hoch Argument	$arg \in \mathbb{R}$	Gleitkomma
ALD	2 hoch Argument	$arg \in \mathbb{R}$	Gleitkomma
SINH	hyp. Sinus	$arg \in \mathbb{R}$	Gleitkomma
COSH	hyp. Kosinus	$arg \in \mathbb{R}$	Gleitkomma
TANH	hyp. Tangens	$arg \in \mathbb{R}$	Gleitkomma
COTH	hyp. Kotangens	$arg \neq 0$	Gleitkomma
LN	nat. Logarithmus	$arg > 0$	Gleitkomma
LOG	dek. Logarithmus	$arg > 0$	Gleitkomma
LD	2er Logarithmus	$arg > 0$	Gleitkomma
ASINH	inv. hyp. Sinus	$arg \in \mathbb{R}$	Gleitkomma
ACOSH	inv. hyp. Kosinus	$arg \geq 1$	Gleitkomma
ATANH	inv. hyp. Tangens	$\mid arg \mid < 1$	Gleitkomma
ACOTH	inv. hyp. Kotangens	$\mid arg \mid > 1$	Gleitkomma
INT	ganzzahliger Anteil	$arg \in \mathbb{R}$	Integer
BITCNT	binäre Quersumme	Integer	Integer
FIRSTBIT	niedrigstes 1-Bit	Integer	Integer
LASTBIT	höchstes 1-Bit	Integer	Integer
BITPOS	einziges 1-Bit	Integer	Integer

Tabelle 2.10: vordefinierte Funktionen in AS - Teil 1 (Integer- und Gleitkomma-Funktionen)

Name	Funktion	Argument	Ergebnis
SGN	Vorzeichen (0/1/-1)	Integer oder Gleitkomma	Integer
ABS	Betrag	Integer oder Gleitkomma	Integer oder Gleitkomma
TOUPPER	pass. Großbuchstabe	Integer	Integer
TOLOWER	pass. Kleinbuchstabe	Integer	Integer
UPSTRING	wandelt alle Zeichen in Großbuchstaben	String	String
LOWSTRING	wandelt alle Zeichen	String	String

	in Kleinbuchstaben		
STRLEN	liefert Länge eines Strings	String	Integer
SUBSTR	extrahiert Teil eines Strings	String, Integer, Integer	String
CHARFROMSTR	extrahiert ein Zeichen aus einem String	String, Integer	Integer
STRSTR	sucht Teilstring in einem String	String, String	Integer
VAL	evaluiert Stringinhalt als Ausdruck	String	abh. von Argument
EXPRTYPE	liefert Typ des Arguments	Integer, Gleitkomma, String	0 1 2
FSIZE	liefert Dateigröße	String	Integer

Tabelle 2.11: vordefinierte Funktionen in AS - Teil 2 (Integer- und String-Funktionen)

Die Funktionen `FIRSTBIT`, `LASTBIT` und `BITPOS` liefern als Ergebnis -1, falls überhaupt kein bzw. nicht genau ein Bit gesetzt ist. Zusätzlich gibt `BITPOS` in einem solchen Fall eine Fehlermeldung aus.

Die String-Funktion `SUBSTR` erwartet als ersten Parameter den Quellstring, als zweiten die Startposition und als dritten die Anzahl zu extrahierender Zeichen (eine 0 bedeutet, alle Zeichen bis zum Ende zu extrahieren). Analog erwartet `CHARFROMSTR` den Quellstring als erstes Argument und die Zeichenposition als zweites Argument. Falls die angegebene Position größer oder gleich der Länge des Quellstrings ist, liefert `SUBSTR` einen Leerstring, während `CHARFROMSTR` eine -1 ergibt. Eine Position kleiner Null wird von `SUBSTR` als Null behandelt, während `CHARFROMSTR` in diesem Fall ebenfalls eine -1 liefert.

Hier ein Beispiel, wie man die beiden Funktionen einsetzt, um einen String im Speicher abzulegen, wobei das String-Ende durch ein gesetztes MSB gekennzeichnet ist:

```
dbstr   macro   arg
```

```
if      strlen(arg) > 1
 db     substr(arg, 0, strlen(arg) - 1)
endif
if      strlen(arg) > 0
 db     charfromstr(arg, strlen(arg) - 1) | 80h
endif
endm
```

STRSTR liefert das erste Auftreten des zweiten Strings im ersten bzw. -1, falls das Suchmuster nicht gefunden wurde. Analog zu SUBSTR und CHARFROMSTR hat das erste Zeichen den Positionswert 0.

FSIZE liefert die Größe einer Datei in Bytes. Die Datei wird sowohl im aktuellen Verzeichnis gesucht, als auch relativ zum Verzeichnis der gerade assemblierten Quelldatei. Im Gegensatz zu INCLUDE und BINCLUDE wird jedoch nicht im Include-Pfad gesucht!

Wenn eine Funktion auch Gleitkommaargumente erwartet, so soll dies nicht bedeuten, daß man nicht z.B.

```
wur2   equ      sqrt(2)
```

schreiben dürfte — in solchen Fällen findet automatisch eine Typkonvertierung statt. Umgekehrt muß allerdings die INT-Funktion angewandt werden, um eine Gleitkommazahl ganz zu bekommen. Bei der Benutzung dieser Funktion ist zu beachten, daß sie als Ergebnis immer einen vorzeichenbehafteten Integer liefert, sie hat also einen Wertebereich von ca. +/-2.0E9.

Schaltet man AS in den case-sensitiven Modus, so können im Gegensatz zu vordefinierten Symbolen die vordefinierten Funktionen weiterhin in beliebiger Schreibweise angesprochen werden. Bei selbst definierten Funktionen (siehe Abschnitt 3.4.9 wird allerdings unterschieden. Dies hat zur Folge, daß z.B. bei der Definition einer Funktion Sin man mit Sin diese Funktion auch erreicht, mit allen anderen Schreibweisen jedoch die eingebaute Funktion.

Für die korrekte Umwandlung von Klein-zu Großbuchstaben ist eine DOS-Version $\geq$ 3.30 erforderlich. *DOS/DPMI*

2.10 Vorwärtsreferenzen und andere Desaster

Dieser Abschnitt ist das Produkt eines gewissen Grolls auf die (durchaus legale) Art und Weise, wie einige Leute programmieren, die in Zusammenhang mit AS bisweilen

das eine oder andere Problem verursachen kann. Die Rede ist hier von sogenannten ,,Vorwärtsreferenzen". Was unterscheidet eine Vorwärtsreferenz von einer normalen Referenz? Dazu sehe man sich folgendes Programmbeispiel an (man sehe mir bitte meine – auch im Rest dieser Anleitung anzutreffende – 68000-Lastigkeit nach):

```
        move.l  #10,d0
loop:   move.l  (a1),d1
        beq     skip
        neg.l   d1
skip:   move.l  d1,(a1+)
        dbra    d0,loop
```

Denkt man sich den Scheifenrumpf mit dem Sprung weg, so bleibt ein äußerst angenehm zu assemblierendes Programm übrig: die einzige Referenz ist der Rücksprung zum Anfang des Rumpfes, und da ein Assembler ein Programm von vorne nach hinten durcharbeitet, hat er den Symbolwert bereits ermittelt, bevor er ihn zum erstem Mal benötigt. Sofern man ein Programm hat, das nur solche Rückwärtsreferenzen besitzt, ist man in der angenehmen Lage, nur einmal durch den Quellcode gehen zu müssen, um den korrekten und optimalen Maschinencode zu finden. Einige Hochsprachen wie Pascal mit ihrer strikten Regel, daß alles vor der ersten Benutzung definiert sein muß, nutzen genau diese Eigenschaft aus, um den Übersetzungsvorgang zu beschleunigen.

Leider ist die Sache im Falle von Assembler nicht so einfach, denn man will ja bisweilen auch vorwärts im Code springen oder muß aus bestimmten Gründen Variablen Definitionen hinter den Code verlegen. Dies ist im Beispiel der Fall für den bedingten Sprung, mit dem ein anderer Befehl übersprungen wird. Wenn der Assembler im ersten Durchlauf auf den Sprungbefehl trifft, so sieht er sich mit der Situation konfrontiert, entweder die Teilfelder der Instruktion, die die Sprungadresse beinhalten, leer zulassen, oder seitens des Formelparsers (der das Adreßargument ja auswerten muß) anstelle des korrekten, aber unbekannten Wertes einen Wert anzubieten, der ,,niemandem wehtut". Bei einem einfachen Assembler, der nur eine Zielarchitektur kennt und bei dem sich die betroffenen Befehle an einer Hand abzählen lassen, wird man sicher die erste Variante wählen, bei AS mit seinen vielen Dutzend Zielen wäre die Zahl der Sonderabfragen aber extrem hoch geworden, so daß nur der zweite Weg in Frage kam: Falls im ersten Pass ein unbekanntes Symbol auftaucht, so liefert der Formelparser den momentanen Stand des Programmzählers als Ergebnis zurück! Nur dieser Wert ist geeignet, relativen Sprüngen mit Sprungdistanzen unbekannter Länge eine Adresse anzubieten, die nicht zu Fehlern führt. Dies beantwortet auch die bisweilen gestellte Frage, warum in einem Listing des ersten Passes (dies bleibt z.B. stehen, wenn AS aufgrund anderer Fehler den zweiten Pass

erst gar nicht beginnt), z.T. falsche Adressen im erzeugten Binärcode gezeigt werden - dies sind noch nicht aufgelöste Vorwärtsreferenzen.

Das obige Beispiel offenbart allerdings noch eine weitere Schwierigkeit von Vorwärtsreferenzen: Je nach Abstand von Quelle und Ziel im Code kann der Sprungbefehl entweder lang oder kurz sein. Diese Entscheidung über die Code-Länge - und damit auch die Adressen folgender Labels - kann jedoch mangels genauer Kenntnis der Zieladresse im ersten Pass nicht erfolgen. Sofern der Programmierer nicht explizit kenntlich gemacht hat, ob der Sprung lang oder kurz sein soll, behelfen sich reine 2-Pass-Assembler wie ältere MASM-Versionen von Microsoft damit, im ersten Pass (nach diesem müssen alle Adressen festliegen) Platz für die längste Version zu reservieren und im zweiten Pass den überschüssigen Platz mit `NOP`s aufzufüllen. AS-Versionen bis 1.37 taten dieses ebenfalls, danach bin ich auf das Multipass-Verfahren übergegangen, das die strenge Einteilung in zwei Passes aufhebt und beliebig viele Durchgänge erlaubt. Dazu wird im ersten Pass der optimale Code mit den angenommenen Symbolwerten erzeugt. Stellt AS fest, daß im zweiten Pass durch Codelängenveränderungen sich Werte von Symbolen geändert haben, so wird einfach noch ein dritter Pass eingelegt, und da durch die neuen Symbolwerte des zweiten Passes auch im dritten Pass sich der Code wieder verkürzen oder verlängern kann, ist ein weiterer Pass nicht unmöglich. Ich habe schon 8086-Programme erlebt, bei denen erst nach 12 Durchgängen alles stimmte. Leider erlaubt dieser Mechanismus nicht die Vorgabe einer Maximalzahl von Durchläufen, ich kann als Regel nur sagen, daß die Anzahl von Durchläufen sinkt, je mehr man davon Gebrauch macht, Sprung- oder Adreßlängen explizit vorzugeben.

Speziell bei großen Programmen kann es zu einer interessanten Situation kommen: Die Lage eines vorwärts gerichteten Sprunges hat sich im zweiten Pass so weit gegenüber dem ersten verschoben, daß der jetzt noch benutzte Label-Wert aus dem ersten Pass außerhalb der erlaubten Sprungdistanz liegt. AS berücksichtigt solche Situationen, indem er jegliche Fehlermeldungen über zu weite Sprungdistanzen unterdrückt, sobald er erkannt hat, daß er wegen sich ändernder Symbolwerte ohnehin einen weiteren Durchlauf machen muß. Dies funktioniert zwar in 99% aller Fälle, es gibt jedoch auch Konstrukte, in denen der erste, derartig kritische Befehl bereits auftaucht, bevor AS eine Chance hat, zu erkennen, daß ein neuer Pass erforderlich ist. Das folgende Beispiel konstruiert eine solche Situation mit Hilfe einer Vorwärtsreferenz (und war der Anlaß für die Überschrift dieses Abschnitts...):

```
cpu   6811

org     $8000
beq     skip
rept    60
```

```
         ldd     Var
        endm
skip:   nop

Var     equ     $10
```

Aufgrund der Adreßlage nimmt AS im ersten Pass lange Adressen für die `LDD`-Befehle an, was eine Code-Länge von 180 Bytes ergibt und im zweiten Pass (zum Zeitpunkt des `BEQ`-Befehls ist noch der ,,falsche" Wert von `skip` aktuell, d.h. AS weiß zu diesem Zeitpunkt noch nicht, daß der Code in Wirklichkeit nur 120 Bytes lang ist) gibt es eine Fehlermeldung wegen einer überschrittenen Sprungdistanz. Dieser Fehler läßt sich auf drei Arten vermeiden:

1. Weisen Sie AS explizit darauf hin, daß er für die `LDD`-Befehle kurze Adressen verwenden darf (`ldd <Var`)
2. Entfernen Sie diese verfluchte Vorwärtsreferenz und setzen Sie die `EQU`-Anweisung nach vorne, wo sie hingehört (OK, ich beruhige mich ja schon wieder...)
3. Für ganz Unentwegte: Benutzten Sie die `-Y`-Option, so daß AS die Fehlermeldung beim Erkennen der Adreßverschiebung nachträglich verwirft. Nicht schön, aber...

Noch ein Hinweis zum `EQU`-Befehl: Da AS nicht wissen kann, in welchem Zusammenhang ein mit `EQU` definiertes Symbol später verwendet wird, wird ein `EQU` mit Vorwärtsreferenzen im ersten Pass überhaupt nicht durchgeführt. Wird das mit `EQU` definierte Symbol also im zweiten Pass vorwärts referenziert:

```
        move.l  #sym2,d0
sym2    equ     sym1+5
sym1    equ     0
```

so handelt man sich im zweiten Pass eine Fehlermeldung wegen eines undefinierten Symbols ein...aber warum machen Leute eigentlich solche Dinge ???

Zugegeben, das war ein ziemlich länglicher Ausflug, aber es mußte einfach einmal sein. Was sollte man als Erkenntnis aus diesem Abschnitt mitnehmen?

1. AS versucht immer, den kürzest möglichen Code zu erzeugen. Dazu benötigt er eine endliche Zahl von Durchläufen. Wenn man ihn nicht gerade knebelt, kennt AS keine Rücksichten...

2. Wenn sinnvoll und möglich, Sprung- und Adreßlängen explizit vorgeben. Man kann damit u.U. die Anzahl der Durchläufe deutlich reduzieren.

3. Vorwärtsreferenzen auf das allernötigste beschränken. Man erleichtert sich und AS das Leben damit erheblich!

2.11 Registersymbole

Gültigkeit: PowerPC, M-Core, XGate, 4004/4040, MCS-48/(2)51, 8086, 80C16x, AVR, XS1, Z8, KCPSM, Mico8, TMS340xx, MSP430(X), ST9, M16, M16C, H8/300, H8/500, SH, H16, 8080/8085, Zx80, i960, XA, 29K, TLCS-9000, KENBAK, SC/MP, PDP-11, VAX

Manchmal ist es erwünscht, nicht nur einer Speicheradresse oder einer Konstanten, sondern auch einem Register einen symbolischen Namen zuzuweisen, um seine Funktion in einem bestimmten Programmabschnitt zu verdeutlichen. Dies ist bei Prozessoren, die die Register schlicht als einen weiteren Adreßraum behandeln, recht problemlos, da als Register damit auch Zahlenausdrücke erlaubt sind und man solche Symbole mit schlichten `EQU`s definieren kann (z.B. bei MCS-96 oder TMS7000). Bei den allermeisten Prozessoren jedoch sind Registernamen festgelegte Literale, und AS behandelt sie beim Parsing aus Geschwindigkeitsgründen gesondert, so daß auch ein getrennter Typ von Symbolen für solche Registersymbole oder -aliase existiert. Registersymbole können wie gewöhnliche Symbole mit `EQU` oder `SET` definiert und umdefiniert werden, zudem existiert eine spezielle `REG`-Anweisung, die explizit nur Symbole bzw. Ausdrücke dieses Typs akzeptiert.

Registersymbole unterliegen einer Reihe von Einschränkungen: zum einen ist die Menge der Literale beschränkt und durch den jeweiligen Zielprozessor vorgegeben, zum anderen kann man mit Registersymbolen nicht rechnen. Etwas in dieser Form:

```
myreg   reg     r17         ; Definition Registersymbol
        addi    myreg+1,3   ; geht nicht!
```

ist also *nicht* zulässig. Einfache Zuweisungen sind dagegen auch über mehrere Stufen hinweg erlaubt:

```
myreg   reg     r17         ; Definition Registersymbol
myreg2  reg     myreg       ; myreg2 -> r17
```

Des weiteren sind Vorwärtsreferenzen bei Registersymbolen noch kritischer als bei anderen Typen von Symbolen. Ist ein Symbol nicht definiert, so kann AS nur mutmaßen, was für ein Typ von Symbol es sein wird, und entscheidet sich in Zweifelsfall

für eine einfache Zahl, was bei den meisten Prozessoren einem Zugriff auf eine absolute Adresse im Speicher gleichkommt. Nun sind bei den meisten Prozessoren die Nutzungsmöglichkeiten für Speicheradressen als Operand deutlich eingeschränkter als für Register. Je nach Situation erhält man so eine Fehlermeldung über einen nicht erlaubten Adressierungsmodus, und es kommt zu keinem zweiten Pass...

Registersymbole sind analog zu normalen Symbolen lokal zu Sektionen, und es ist auch durch Anhängen eines in eckige Klammern gesetzten Sektionsnamens möglich, auf ein Registersymbol aus einer bestimmten Sektion zuzugreifen.

2.12 Sharefile

Diese Funktion ist ein Abfallprodukt aus den reinen 68000er-Vorgängern von AS, da sie vielleicht doch der (die?!) eine oder andere gebrauchen könnte, habe ich sie drin gelassen. Grundproblem ist es, an bestimmte beim Assemblieren entstehende Symbole heranzukommen, weil man evtl. mit diesen Adreßinformationen auf den Speicher des Zielsystems zugreifen möchte. Der Assembler erlaubt es, mit Hilfe des `SHARED`-Pseudobefehles (siehe dort) Symbolwerte extern zur Verfügung zu stellen. Zu diesem Zweck erstellt der Assembler im zweiten Pass eine Textdatei mit den gewünschten Symbolen und ihren Werten, die mittels Include in ein Hochsprachen- oder weiteres Assemblerprogramm eingebunden werden können. Das Format der Textdatei (C, Pascal oder Assembler) wird durch die Kommandozeilenschalter `p`, `c` oder `a` festgelegt.

ACHTUNG! Ist keiner dieser Schalter angegeben, so wird auch keine Datei erzeugt, egal ob sich `SHARED`-Befehle im Quelltext finden oder nicht!

AS prüft beim Anlegen der Share-Datei nicht, ob bereits eine Datei gleichen Namens existiert, eine solche wird ggfs. einfach überschrieben. Eine Abfrage halte ich nicht für sinnvoll, da AS dann bei jedem Lauf fragen würde, ob er die alte Version der Share-Datei überschreiben darf, und das wäre doch sehr lästig...

2.13 Prozessor-Aliase

Mit Varianten gängiger Mikrocontroller-Familien ist es wie mit Kaninchen: Sie vermehren sich schneller, als man mit der Versorgung hinterher kommen kann. Im Zuge der Entwicklung von Prozessorkernen als Bausteine für ASICs und von Controller-Familien mit vom Kunden wählbarer Peripherie wird die Zahl von Controller-Varianten, die sich von einem bekannten Typ nur in einigen Peripherie-Details unterscheiden, immer größer. Die Unterscheidung der einzelnen Typen ist aber trotz

meist identischer Prozessorkerns wichtig, um z.B. in den Include-Dateien den korrekten Satz von Peripherieregistern einzublenden. Bisher habe ich mich zwar immer bemüht, die wichtigsten Vertreter einer Familie in AS einzubauen (und werde das auch weiter tun), aber manchmal läuft mir die Entwicklung einfach auf und davon...es mußte also ein Mechanismus her, mit dem man die Liste der unterscheidbaren Prozessortypen selbst erweitern kann.

Das Ergebnis davon sind Prozessor-Aliasse: Mit der Kommandozeilenoption `alias` kann man einen neuen Prozessortyp definieren, der im Befehlssatz einem anderen, in AS fest eingebauten Typ entspricht. Bei Benutzung dieses Typs im `CPU`-Befehl wird sich AS also wie beim ,,Original" verhalten, mit einem Unterschied: Die Variablen `MOMCPU` bzw. `MOMCPUNAME` werden auf den Namen des Alias gesetzt, wodurch der neue Name zur Unterscheidung z.B. in Include-Dateien dienen kann.

Die Definition dieser Aliasse wurde aus zwei Gründen mit Kommandozeilenoptionen anstatt Pseudobefehlen vorgenommen: zum einen wäre es ohnehin nicht möglich gewesen, die Definition der Aliasse zusammen mit den Registerdefinitionen in eine Include-Datei zu legen, denn in einem Programm, das so eine Datei benutzen wollte, müßte sie ja sowohl vor als auch nach dem `CPU`-Befehl in der Hauptdatei eingebunden werden - eine Vorstellung, die irgendwo zwischen unelegant und unmöglich liegt. Zum zweiten ermöglicht diese Implementierung, die Definition der neuen Typen in eine Datei zu legen, die über die `ASCMD`-Variable beim Start automatisch ausgeführt wird, ohne das sich das Programm darum kümmern müßte.

Kapitel 3

Pseudobefehle

Nicht für alle Prozessoren sind alle Pseudobefehle definiert. Vor der Beschreibung eines Befehls ist deshalb jeweils vermerkt, für welche Prozessortypen dieser Befehl erlaubt ist.

3.1 Definitionen

3.1.1 SET, EQU und CONSTANT

Gültigkeit: alle Prozessoren, `CONSTANT` *nur KCPSM(3)*

`SET` und `EQU` erlauben die Definition typenloser Konstanten, d.h. sie werden keinem Segment zugeordnet und ihre Verwendung erzeugt in keinem Fall eine Warnung wegen Segment-Vermischung. Während `EQU` Konstanten definiert, die nicht wieder (mit `EQU`) geändert werden können, erlaubt `SET` die Definition von Variablen, die sich während des Assemblerlaufes verändern lassen. Dies ist nützlich z.B. bei der Allokation von Resourcen à la Interrupt-Vektoren, wie im folgenden Beispiel:

```
VecCnt  SET     0          ; irgendwo am Anfang
        ...
DefVec  MACRO   Name       ; einen neuen Vektor belegen
Name    EQU     VecCnt
VecCnt  SET     VecCnt+4
        ENDM
        ...
        DefVec  Vec1       ; ergibt Vec1=0
        DefVec  Vec2       ; ergibt Vec2=4
```

Intern werden Konstanten und Variablen identisch gespeichert, der einzige Unterschied ist, daß sie als unveränderbar markiert werden, wenn sie mit `EQU` definiert werden. Der Versuch, eine Konstante mit `SET` zu verändern, gibt eine Fehlermeldung.

Mit `EQU/SET` lassen sich Konstanten aller Typen definieren, z.B.

```
IntZwei   EQU   2
FloatZwei EQU   2.0
```

Einige Prozessoren besitzen leider bereits selber einen `SET`-Befehl. Bei diesen muß `EVAL` anstelle von `SET` verwendet werden, falls sich der Maschinenbefehl nicht durch die andere Anzahl der Argumente erkennen läßt. Alternativ ist es auch immer möglich, durch einen vorangestellten Punkt (`.SET` anstelle `SET`) explizit den Pseudobefehl aufzurufen.

Anstelle von `EQU` darf auch `.EQU` oder einfach ein Gleichheitszeichen geschrieben werden, analog kann man anstelle von `SET` bzw. `EVAL` einfach `:=` schreiben. Des weiteren existiert eine 'alternative' Syntax, bei der der Synbolname nicht aus dem Feld für das Label genommen wird, sondern das erste Argument ist. Wahlweise darf man also auch schreiben:

```
          EQU   IntZwei,2
          EQU   FloatZwei,2.0
```

Das Feld für das Label muß in diesem Fall leer bleiben.

Aus Kompatibilitätsgründen zum Originalassembler gibt es für das KCPSM-Target auch den `CONSTANT`-Befehl, der im Gegensatz zu `EQU` Namen und Wert immer als Argumente erwartet, also z.B. so:

```
      CONSTANT  const1, 2
```

`CONSTANT` ist allerdings auf Integer-Konstanten beschränkt.

Defaultmäßig sind mit `SET` oder `EQU` definierte Symbole typenlos, optional kann jedoch als zweites bzw. drittes Argument ein Segmentname (`CODE, DATA, IDATA, XDATA, YDATA, BITDATA, IO oder REG`) oder `MOMSEGMENT` für das aktuell gesetzte Segment angegeben werden, um das Symbol einem bestimmten Adreßraum zuordnen. AS berücksichtigt dabei nicht, ob der benutzte Adreßraum bei dem aktuell gesetzten Zielprozessor auch vorhanden ist!

Als kleines verstecktes Extra ist es möglich, über `SET` oder `EQU` den Programmzähler zu setzen, also das zu machen, wozu man ansonsten `ORG` verwenden würde. Dazu gibt man als Symbolnamen den Sonderwert an, mit dem sich auch der

aktuelle Programmzähler abfragen läßt, also je nach gewählter Zielarchitektur ein Stern, ein Dollarzeichen, ein Punkt oder `PC`.

Falls die gewählte Zielarchitektur ein Attribut an den Befehlen zur Angabe der Operandengröße unterstützt (z.B. 680x0), so ist dieses ebenfalls bei `SET` und `EQU` erlaubt. Das definierte Symbol wird dann mit dieser Operandengröße in der Symboltabelle abgelegt. Deren Verwendung bei Benutzung des Symbols ist architekturabängig.

3.1.2 SFR und SFRB

Gültigkeit: diverse, SFRB nur MCS-51

Diese Befehle funktionieren wie `EQU`, nur sind die damit definierten Symbole dem direkt adressierbaren Datensegment zugeordnet, d.h. sie dienen bevorzugt zur Definition von (wie der Name ahnen läßt) im Daten- bzw. I/O-Bereich eingeblendeten Hardwareregistern. Der dabei zugelassene Wertebereich ist identisch mit dem bei `ORG` für das `DATA` bzw. `IO`-Segment zugelassenen (s. Abschnitt 3.2.1). `SFR` und `SFRB` unterscheiden sich darin, daß `SFRB` das Register als bitadressierbar kennzeichnet, weshalb AS zusätzlich 8 Symbole erzeugt, die dem Bitsegment zugeordnet werden und die Namen `xx.0` bis `xx.7` tragen, z.B.

```
PSW     SFR     0d0h    ; ergibt PSW = D0H (Datensegment)

PSW     SFRB    0d0h    ; zusaetzlich PSW.0 = D0H (Bit)
                        ; bis PSW.7 = D7H (Bit)
```

Da beim 80C251 grundsätzlich alle SFRs ohne zusätzliche Bit-Symbole bitadressierbar sind, ist der `SFRB`-Befehl für ihn auch nicht mehr definiert; die Bits `PSW.0` bis `PSW.7` sind automatisch vorhanden.

AS überprüft bei der Definition eines bitadressierbaren Registers mit `SFRB`, ob die Speicherstelle überhaupt bitadressierbar ist (Bereich 20h..3fh bzw. 80h, 88h, 90h, 98h...0f8h). Ist sie es nicht, so wird eine Warnung ausgegeben; die dann erzeugten Bit-Symbole sind undefiniert.

3.1.3 XSFR und YSFR

Gültigkeit: DSP56xxx

Auch der DSP56000 hat einige Peripherieregister memory-mapped im Speicher liegen, die Sache wird jedoch dadurch komplizierter, daß es zwei Datenbereiche gibt,

den X-und Y-Bereich. Diese Architektur erlaubt einerseits zwar einen höheren Parallelitätsgrad, zwingt jedoch andererseits dazu, den normalen `SFR`-Befehl in die beiden oben genannten Varianten aufzuspalten. Sie verhalten sich identisch zu `SFR`, nur daß `XSFR` ein Symbol im X-Adreßraum definiert und `YSFR` entsprechend eines im Y-Adreßraum. Der erlaubte Wertebereich ist 0..$ffff.

3.1.4 LABEL

Gültigkeit: alle Prozessoren

Die Funktion des `LABEL`-Befehls ist identisch zu `EQU`, nur wird das Symbol nicht typenlos, sondern erhält das Attribut ,,Code". `LABEL` wird genau für einen Zweck benötigt: Labels in Makros sind normalerweise lokal, also nicht außerhalb des Makros zugreifbar. Mit einem `EQU`-Befehl kann man sich zwar aus der Affäre ziehen, die Formulierung

```
<Name>   label    $
```

erzeugt aber ein Symbol mit korrekten Attributen.

3.1.5 BIT

Gültigkeit: MCS-(2)51, XA, 80C166, 75K0, STM8, ST9, AVR, S12Z, SX20/28, H16, H8/300, H8/500, KENBAK, Padauk

`BIT` dient dazu, ein einzelnes Bit einer Speicherstelle mit einem symbolischen Namen gleichzusetzen. Da die Art und Weise, wie verschiedene Prozessoren Bitverarbeitung und -adressierung betreiben, stark variiert, verhält sich auch dieser Befehl je nach Zielplattform anders:

Für die MCS/51-Familie, die einen eigenen Adreßraum für Bitoperanden besitzt, ist die Funktion von `BIT` ganz analog zu `SFR`, d.h. es wird einfach ein Integer-Symbol mit dem angegebenen Wert und dem Segment BDATA erzeugt. Für alle anderen Prozessoren wird die Bitadressierung dagegen zweidimensional mit Adresse und Bitstelle vorgenommen. In diesem Fall verpackt AS beide Teile in einer vom jeweiligen Prozessor abhängigen Weise in ein Integer-Symbol und dröselt dieses bei der Benutzung wieder in die beiden Teile auseinander. Letzterer Fall trifft auch schon für den 80C251 zu: Während zum Beispiel der Befehl

```
Mein_Carry bit PSW.7
```

auf einem 8051 noch dem Symbol `Mein_Carry` den Wert 0d7h zuweisen würde, würde auf einem 80C251 dagegen ein Wert von 070000d0h generiert werden, d.h. die Adresse steht in Bit 0..7 sowie die Bitstelle in Bit 24..26. Dieses Verfahren entspricht dem, das auch beim DBIT- Befehl des TMS370 angewendet wird und funktioniert sinngemäß so auch beim 80C166, nur daß dort Bitstellen von 0 bis 15 reichen dürfen:

```
MSB     BIT     r5.15
```

Beim Philips XA findet sich in Bit 0..9 die Bitadresse, wie sie auch in die Maschinenbefehle eingesetzt wird, für Bits aus den RAM-Speicher wird in Bit 16..23 die 64K-Bank eingesetzt.

Noch etwas weiter geht der `BIT`-Befehl bei der 75K0-Familie: Da dort Bitadressierungen nicht nur absolute Basisadressen verwenden dürfen, sind sogar Ausdrücke wie

```
bit1    BIT     @h+5.2
```

erlaubt.

Beim ST9 ist es hingegen möglich, Bits auch invertiert anzusprechen, was beim `BIT`-Befehl auch berücksichtigt wird:

```
invbit  BIT     r6.!3
```

Näheres zum `BIT`-Befehl beim ST9 findet sich bei den prozessorspezifischen Hinweisen.

Im Falle des H16 sind die Argumente für Speicheradresse und Bitposition vertauscht. Dies wurde getan, um die Syntax zur Definition von Bit identisch zu den Maschinenbefehlen zu machen, die einzelne Bits manipulieren.

3.1.6 DBIT

Gültigkeit: TMS 370xxx

Die TMS370-Reihe hat zwar kein explizites Bit-Segment, jedoch können einzelne Bits als Symbol durch diesen Befehl simuliert werden. `DBIT` benötigt zwei Operanden, nämlich einmal die Adresse der Speicherstelle, in der das Bit liegt, sowie die genaue Position des Bits im Byte. So definiert man z.B. mit

```
INT3            EQU     P019
INT3_ENABLE     DBIT    0,INT3
```

das Bit, welches Interrupts von Anschluß INT3 freigibt. So definierte Bits können dann von den Befehlen `SBIT0, SBIT1, CMPBIT, JBIT0` und `JBIT` genutzt werden.

3.1.7 DEFBIT und DEFBITB

S12Z

Der Prozessorkern der S12Z-Familie verfügt über Befehle, mit denen sich einzelne Bits in Register oder Speicherzellen manipulieren lassen. Um Bits im I/O-Bereich des Prozessors (erste 4 KByte des Adreßraumes) bequem ansprechen zu können, kann man einem einzelnen Bit, definiert durch Speicheradresse und Bitposition, einen symbolischen Namen geben:

```
<Name>          defbit[.Size]   <Adresse>,<Position>
```

Die `Adresse` muß in den ersten 4 KByte liegen, als Operandengröße sind 8, 16 oder 32 Bit (`Size`=b/w/l) zugelassen. Dementsprechend darf `Position` maximal 7, 15 oder 31 sein. Falls keine Opoerandengröße angegeben wird, werden 8 Bit (.b) angenommen. Ein solchermaßen definiertes Bit kann als Argument für die Befehle `BCLR, BSET, BTGL, BRSET` und `BRCLR` verwendet werden:

```
mybit   defbit.b  $200,4
        bclr.b    $200,#4
        bclr      mybit
```

Die beiden Aufrufe von `bclr` in diesem Beispiel erzeugen identischen Code. Da ein solchermaßen definiertes Bit seine Operandengröße "kennt", kann diese bei der Benutzung fortgelassen werden.

Bit-Definitionen innerhalb einer Struktur, die sich auf ein Element einer Struktur beziehen, sind ebenfalls möglich:

```
mystruct struct    dots
reg      ds.w      1
flag     defbit    reg,4
         ends

         org       $100
data     mystruct

         bset      data.flag  ; entspricht bset.w $100,#4
```

Super8

Im Gegensatz zum "klassischen" Z8 verfügt der Super8-Kern über Befehle, mit denen sich Bits in allgemeinen oder Arbeitsregistern bearbeiten lassen. Dabei ist jedoch zu beachten, daß einige davon nur auf Bits arbeiten, die Teil eines der 16 Arbeitsregister sind. Mit der `DEFBIT`-Anweisung lassen sich Bits beider Sorten definieren:

```
workbit defbit r3,#4
slow defbit emt,#6
```

Derart definierte Bits lassen sich dann bei den Befehlen wie ein Pärchen aus Register und Bitposition einsetzen:

```
 ldb r3,emt,#6
 ldb r3,slo  ; gleich bedeutend

 bitc r3,#4
 bitc workbit  ; gleich bedeutend
```

Z8000

Der Z8000 verfügt zwar über Befehle zum Setzen und Rücksetzen von Bits, diese können jedoch nur auf Adressen im Speicher- und nicht im I/O-Adreßraum wirken. Aus diesem Grund lassen sich mit `DEFBIT` bzw. `DEFBITB` auch nur Bit-Objekte im Speicher-Adreßraum definieren. Die Unterscheidung der Operandengröße ist wichtig, weil der Z8000 ein Big-Endian-Prozessor ist: Bit n eines 16-Bit-Worts bei Adresse m entspräche Bit n eines 8-Bit-Byte bei Adresse $m+1$.

μPD7807...μPD7809

Die ersten 16 Byte der Working Area und Sonderregister mit einer Adresse kleiner 16 sind bitadressierbar.

3.1.8 DEFBITFIELD

Gültigkeit: S12Z

Der Prozessorkern der S12Z-Familie kann nicht nur mit einzelnen Bits umgehen, sondern auch zusammenhängende Felder von Bits in einem 8/16/24/32-Bit-Wert extrahieren oder schreiben. Analog zu `DEFBIT` läßt sich auch ein Bitfeld symbolisch definieren:

```
<Name>          defbitfield[.Size]   <Adresse>,<Breite>:<Position>
```

Im Gegensatz zu einzelnen Bits sind hier auch 24 Bits (.p) als Operandengröße zugelassen, der Wertebereich von `Position` und `Breite` ist dementsprechend von 0 bis 23 bzw. 1 bis 24. Auch hier ist es wieder zulässig, Bitfelder als Teil von Strukturen zu definieren:

```
mystruct struct      dots
reg      ds.w        1
clksel   defbitfield reg,4:8
         ends

         org       $100
data     mystruct

         bfext     d2,data.clksel ; fetch $100.w bits 4..11
                                  ; to D2 bits 0..7
         bfins     data.clksel,d2 ; insert D2 bits 0..7 into
                                  ; $100.w bits 4..11
```

Die interne Darstellung von Bits, die mit `DEFBIT` definiert wurden, ist gleich der von Bitfeldern der Breite eins. Ein symbolisch definiertes einzelnes Bit kann also auch als Argument für `BFINS` und `BFEXT` verwendet werden.

3.1.9 PORT

Gültigkeit: PALM, 8008/8080/8085/8086, XA, Z80, Z8000, 320C2x/5x, TLCS-42/47, AVR, F8, IMP-16

`PORT` arbeitet analog zu `SFR`, nur wird das Symbol dem I/O-Adreßbereich zugeordnet. Erlaubte Werte sind 0..7 beim 3201x und 8008, 0..15 beim 320C2x und PALM, 0..65535 beim 8086, Z8000 und 320C5x, 0..63 beim AVR und 0..255 beim Rest.

Beispiel: eine PIO 8255 liege auf Adresse 20H:

```
PIO_Port_A PORT 20h
PIO_Port_B PORT PIO_Port_A+1
PIO_Port_C PORT PIO_Port_A+2
PIO_Ctrl   PORT PIO_Port_A+3
```

3.1.10 REG und NAMEREG

*Gültigkeit: 680x0, PowerPC, PALM, M*Core, XGate, H8, SH, H16, M16(C), PDP-11, WD16, VAX, 4004, MCS-48/51, 8086, i960, XA, AVR, 29xxx, 80C16x, Z8, Z80, Z8000, KCPSM(3), LatticeMico8, TLCS-9000, ST9, MSP430(X), V60, SC/MP, NS32xxx, WE32xxx, XCore, KENBAK, CP-1600 (*`NAMEREG` *nur für KCPSM(3)), LatticeMico8, MSP430(X)*

Obwohl immer mit gleicher Syntax, hat diese Anweisung von Prozessor zu Prozessor eine leicht abweichende Bedeutung: Falls der Zielprozessor für Register einen eigenen Adreßraum verwendet, so hat `REG` die Wirkung eines simplen `EQU`s für eben diesen Adreßraum (z.B. beim ST9). Für alle anderen Prozessoren definiert `REG` Registersymbole, deren Funktion in Abschnitt 2.11 beschrieben sind.

`NAMEREG` existiert aus Kompatibilitätsgründen zum Originalassembler für den KCPSM. Es hat die gleiche Funktion, lediglich werden sowohl Register- als auch symbolischer Name als Argumente angegeben, z.B. so:

```
NAMEREG  s08, treg
```

Auf PDP-11 und TMS340xx darf `REG` zusätzlich ohne Namen im Label-Feld benutzt werden. Es wird dann als einziges Argument entweder `ON` oder `OFF` erwartet, und damit werden die eingebauten Register-Aliase:

- PDP-11: `Rn` = `%n`, `SP` = `R6`, `PC` = `R7`
- TMS340xx: `SADDR` = `B0`, `SPTCH` = `B1`, `DADDR` = `B2`, `DPTCH` = `B3`, `OFFSET` = `B4`, `WSTART` = `B5`, `WEND` = `B6`, `DYDX` = `B7`, `COLOR0` = `B8`, `COLOR1` = `B9`, `COUNT` = `B10`, `INC1` = `B11`, `INC2` = `B12`, `PATTRN1` = `B13`

ein- und ausgeschaltet. Sie sind im Default verfügbar, und sollten nur abgeschaltet werden, wenn sie mit eigenen Symbolnamen in einem Programm kollidieren. Die augenblickliche Einstellung läßt sich aus dem Symbol `DEFAULT_REGSYMS` lesen.

3.1.11 LIV und RIV

Gültigkeit: 8X30x

`LIV` und `RIV` dienen dazu, sogenannte IV-Bus-Objekte zu definieren. Bei diesen handelt es sich um Bitgruppen in peripheren Speicherzellen mit einer Länge von 1..8 Bit, die fortan symbolisch angesprochen werden können, so daß man bei den

entsprechenden Befehlen nicht mehr Adresse, Länge und Position separat angeben muß. Da die 8X30x-Prozessoren zwei periphere Adreßräume besitzen (einen ,,linken" und einen ,,rechten", sind auch zwei separate Befehle definiert. Die Parameter dieser Befehle sind allerdings identisch: es müssen drei Parameter sein, die Adresse, Startposition und Länge angeben. Weitere Hinweise zur Benutzung von Busobjekten finden sich in Abschnitt 4.25.

3.1.12 CHARSET

Gültigkeit: alle Prozessoren

Einplatinen-Systeme, zumal wenn sie LCDs ansteuern, benutzen häufig einen anderen Zeichensatz als ASCII, und daß die Umlautkodierung mit der im PC übereinstimmt, dürfte wohl reiner Zufall sein. Und dann gibt es auch (historische) Systeme, die z.B. eine EBCDIC-Variante verwenden...um nun aber keine fehlerträchtigen Umkodierungen im Code per Hand vornehmen zu müssen, enthält der Assembler eine Umsetzungstabelle für Zeichen, die jedem (ASCII-)zeichen im Quellcode ein Zielzeichen zuordnet. Zur Modifikation dieser Tabelle (die initial 1:1 übersetzt), dient der Befehl `CHARSET`. `CHARSET` kann mit verschiedenen Parameterzahlen und -typen aufgerufen werden:

Ein

```
CHARSET
```

ohne jegliche Argumente setzt die Übersetzungstabelle auf den 1:1-Default wieder zurück.

Wird nur ein Argument gegeben, muß es sich dabei um einen String-Ausdruck handeln, der von AS als Dateiname interpretiert wird:

```
CHARSET  "mapping.bin"
```

Aus dieser Datei liest AS dann die ersten 256 Bytes aus und kopiert sie in die Übersetzungstabelle. Hiermit lassen sich also komplexere, extern erzeugte Tabellen mit einer Anweisung laden.

In allen anderen Varianten wird ein einzelner Eintrag oder ein ganzer Bereich von Einträgen in der Tabelle modifiziert. Mit zwei (Integer-)Argumenten kann ein einzelner Eintrag neu gesetzt werden. Ein

```
CHARSET 'ä',128
```

bedeutet zum Beispiel, daß das Zielsystem das ä mit der Zahl 128 kodiert. Es ist auch möglich zu definieren, daß ein bestimmtes Zeichen überhaupt nicht dargestellt werden kann. Dafür läßt man das zweite Argument leer:

```
CHARSET '[',
```

Soll fortan das ,,gelöschte" Zeichen irgendwo in einem String im Speicher abgelegt werden, führt dies zu einer Fehlermeldung.

Anstelle eines einzelnen Zeichens kann auch ein ganzer Bereich umgemappt werden. Das erste und zweite Argument geben den Bereich der Zeichen an, das dritte das Mapping des ersten Zeichens. Falls z.B. das Zielsystem keine Kleinbuchstaben unterstützt, können mit

```
CHARSET 'a','z','A'
```

alle Kleinbuchstaben auf die passenden Großbuchstaben automatisch umgesetzt werden. Und auch ein Bereich von Zeichen kann als ,,nicht verfügbar" markiert werden:

```
CHARSET 'a','z',
```

verbietet die Verwendung von Kleinbuchstabem.

In der letzten Variante (wieder mit nur zwei Argumenten) folgt nach dem Startindex ein String, der das Mapping ab dem Start-Zeichen auflistet. Das Umlegen von Klein- auf Großbuchstaben könnte man also auch so formulieren:

```
CHARSET 'a',"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

ACHTUNG! Das durch `CHARSET` definierte Mapping greift nicht nur beim Ablegen von Strings im Speicher, es greift auch bei Multi-Character-Konstanten, also als ,,ASCII" formulierten Integer-Konstanten. Dies bedeutet, daß eine bereits modifizierte Umsetzungstabelle in den obigen Beispielen zu anderen Ergebnissen führen kann!

Mit der eingebauten Funktion `CODEPAGE_VAL` kann die Übersetzung eines einzelnen Zeichens abgefragt werden. Für nicht gemappte Zeichen liefert die Funktion eine -1 zurück.

3.1.13 CODEPAGE

Gültigkeit: alle Prozessoren

Mit der `CHARSET`-Anweisung hat man zwar beliebige Freiheiten in der Zeichenzuordnung zwischen Entwicklungs- und Zielplattform, wenn auf der Zielplattform jedoch verschiedene Zeichensätze existieren, kann das Umschalten zwischen diesen jedoch zu einer umständlichen Orgie von `CHARSET`-Kommandos werden. Mit der `CODEPAGE`-Anweisung kann man jedoch mehrere Zeichentabellen vorhalten und zwischen diesen mit einem Befehl umschalten. Als Parameter erwartet `CODEPAGE` ein oder zwei Namen: zum einen den Namen der fortan zu benutzenden Tabelle, zum anderen optional den Namen der Tabelle, die die initiale Belegung der Tabelle vorgibt (dieser Parameter hat somit auch nur eine Bedeutung beim ersten Umschalten auf eine Tabelle, bei der AS sie automatisch anlegt). Fehlt der zweite Parameter, so ist die initiale Belegung der neuen Tabelle gleich der vorher aktiven Tabelle. Alle folgenden `CHARSET`-Anweisungen verändern *nur* die momentan aktive Tabelle.

Zu Beginn eines Durchlaufes wird von AS automatisch eine einzelne Zeichentabelle mit dem Namen `STANDARD` erzeugt und 1:1 vorbelegt. Verwendet man keine `CODEPAGE`-Anweisungen, so beziehen sich alle mit `CHARSET` gemachten Einstellungen auf diese Tabelle.

3.1.14 ENUM, NEXTENUM und ENUMCONF

Gültigkeit: alle Prozessoren

`ENUM` dient analog zu dem entsprechenden Befehl in C dazu, Aufzählungstypen zu definieren, d.h. eine Reihe von Integer-Konstanten, denen fortlaufende Werte (von 0 an beginnend) zugewiesen werden. Als Parameter werden dabei die Namen der zu definierenden Symbole angegeben, wie in dem folgenden Beispiel:

```
ENUM SymA,SymB,SymC
```

Dieser Befehl weist den Symbolen `SymA`, `SymB` und `SymC` die Werte 0, 1 und 2 zu.

Möchte man eine Aufzählung über mehrere Zeilen verteilen, so verwendet man ab der zweiten Zeile den Befehle `NEXTENUM` anstelle von `ENUM`. Der interne Zähler, der den Symbolen der Aufzählung fortlaufende Werte zuweist, wird dann nicht wieder auf Null zurück gesetzt, wie in dem folgenden Fall:

```
ENUM     Januar=1,Februar,Maerz,April,Mai,Juni
NEXTENUM Juli,August,September,Oktober
NEXTENUM November,Dezember
```

An diesem Beispiel sieht man auch, daß man einzelnen Symbolen explizit Werte anstelle des aktuellen Zählerstandes zuweisen kann. Der interne Zähler wird anhand dieses Wertes auch aktualisiert.

Die Definition von Symbolen mit `ENUM` gleicht einer Definition mit `EQU`, d.h. es ist nicht möglich, einem Symbol einen neuen Wert zuzuweisen.

Die `ENUMCONF`-Anweisung erlaubt das Verhalten von `ENUM` zu beeinflussen. `ENUMCONF` akzeptiert ein oder zwei Argumente, wobei das erste Argument immer der Wert ist, um den der interne Zähler pro Symbol in einer Aufzählung hochgezählt wird. Mit einem

```
ENUMCONF 2
```

werden den Symbolen also zum Beispiel die Werte 0,2,4,6... anstelle 0,1,2,3... zugewiesen.

Das zweite (optionale) Argument von `ENUMCONF` bestimmt, welchen Adreßraum die Symbole zugeordnet werden. Per Default sind mit `ENUM` definierte Symbole typenlos, man kann aber zum Beispiel mit einem

```
ENUMCONF 1,CODE
```

bestimmen, daß sie im Instruktions-Adreßraum liegen sollen. Die Namen der Adreßräume sind die gleichen wie beim `SEGMENT`-Befehl (3.2.20), zusätzlich ist als Argument ein `NOTHING` erlaubt, um wieder typenlose Symbole zu erzeugen.

3.1.15 PUSHV und POPV

Gültigkeit: alle Prozessoren

Mit `PUSHV` und `POPV` ist es möglich, den Wert von (nicht Makro-lokalen) Symbolen temporär zu speichern und zu einem späteren Zeitpunkt wiederherzustellen. Die Speicherung erfolgt auf *Stacks*, d.h. Last-In-First-Out-Speichern. Ein Stack hat einen Namen, der den allgemeinen Symbolkonventionen genügen muß, und existiert so lange, wie er mindestens ein Element enthält: Ein bisher nicht existierender Stack wird bei `PUSHV` automatisch angelegt, ein durch `POPV` leer werdender Stack wird automatisch wieder aufgelöst. Der Name des Stacks, auf den Symbole abgelegt und von dem sie wieder abgeholt werden sollen, ist der erste Parameter von `PUSHV` bzw. `POPV`, danach folgt eine beliebige Menge von Symbolen als weitere Parameter. Alle in der Liste aufgeführten Symbole müssen bereits existieren, es ist also *nicht* möglich, mit einem `POPV`-Befehl implizit neue Symbole zu definieren.

Stacks stellen eine globale Ressource dar, d.h. ihre Namen sind nicht lokal zu Sektionen.

Wichtig ist, daß die Variablenliste *immer* von links nach rechts abgearbeitet wird. Wer also mehrere Variablen mit `POPV` von einem Stack herunter holen will, muß diese in genau umgekehrter Reihenfolge zum entsprechenden `PUSHV` angeben!

Der Name des Stacks kann auch weggelassen werden, etwa so:

```
pushv   ,var1,var2,var3
.
.
popv    ,var3,var2,var1
```

AS verwendet dann einen internen, vordefinierten Default-Stack.

Nach Ende eines Durchlaufes überprüft AS, ob noch Stacks existieren, die nicht leer sind, und gibt deren Namen sowie ,,Füllstand" aus. Mit diesen Warnungen kann man herausfinden, ob an irgendeiner Stelle die `PUSHV`'s und `POPV`'s nicht paarig sind. Es ist jedoch in keinem Fall möglich, Symbolwerte in einem Stack über mehrere Durchläufe hinwegzuretten: Zu Beginn eines Durchlaufes werden alle Stacks geleert!

3.2 Codebeeinflussung

3.2.1 ORG

Gültigkeit: alle Prozessoren

`ORG` erlaubt es, den Assembler-internen Adreßzähler mit einem neuen Wert zu besetzen. Der Wertebereich ist vom momentan gewählten Segment und vom Prozessortyp abhängig (Tabelle 3.1). Die untere Grenze ist dabei immer 0; die obere Grenze der angegebene Wert minus eins.

Falls in einer Familie verschiedene Varianten unterschiedlich große Adreßräume haben, ist jeweils der maximale Raum aufgeführt.

ORG wird in erster Linie benötigt, um dem Code eine neue Startadresse zu geben und damit verschiedene, nicht zusammenhängende Codestücke in einer Quelldatei unterzubringen. Sofern nicht in einem Feld explizit anders angegeben, ist die vorgegebene Startadresse in einem Segment (d.h. die ohne `ORG` angenommene) immer 0.

WICHTIG: Falls auch mit dem `PHASE`-Befehl gearbeitet wird, muß berücksichtigt werden, daß das Argument von `ORG` immer die *Ladeadresse* des Codes ist, nicht die *Ausführungsadresse*. Ausdrücke, die sich mit dem $- oder -Symbol auf den aktuellen Programmzähler beziehen, liefern aber die *Ausführungsadresse* des Codes und führen als Argument von `ORG` nicht zum gewünschten Ergebnis. In solchen Fällen ist die `RORG`-Anweisung (3.2.2) das Mittel der Wahl.

Ziel	CODE	DATA	I-DATA	X-DATA	Y-DATA	BIT-DATA	IO	REG	ROM-DATA	EE-DATA
68xxx/ MCF	4G	—	—	—	—	—	—	—	—	—
DSP56000 DSP56300	64K/ 16M	—	—	64K/ 16M	64K/ 16M	—	—	—	—	—
PowerPC	4G	—	—	—	—	—	—	—	—	—
PALM	64K	—	—	—	—	—	16	—	—	—
M*Core	4G	—	—	—	—	—	—	—	—	—
6800,6301, 6811	64K	—	—	—	—	—	—	—	—	—
6805/ HC08	8K/ 64K	— —	— —	— —	— —	— —	— —	— —	— —	— —
6809, 6309, 052001	64K	—	—	—	—	—	—	—	—	—
68HC12, 68HC12X, XGATE	64K	—	—	—	—	—	—	—	—	—
S12Z	16M	—	—	—	—	—	—	—	—	—
68HC16	1M	—	—	—	—	—	—	—	—	—
68RS08	16K	—	—	—	—	—	—	—	—	—
H8/300 H8/300H	64K 16M	—	—	—	—	—	—	—	—	—
H8/500 (Min)	64K	—	—	—	—	—	—	—	—	—
H8/500 (Max)	16M	—	—	—	—	—	—	—	—	—
SH7000/ 7600/7700	4G	—	—	—	—	—	—	—	—	—
HD614023	2K	160	—	—	—	—	16	—	—	—
HD614043	4K	256	—	—	—	—	16	—	—	—
HD614081	8K	512	—	—	—	—	16	—	—	—

Ziel	CODE	DATA	I-DATA	X-DATA	Y-DATA	BIT-DATA	IO	REG	ROM-DATA	EE-DATA
HD641016	16M	—	—	—	—	—	—	—	—	—
6502, MELPS-740	64K	—	—	—	—	—	—	—	—	—
HUC6280	2M	—	—	—	—	—	—	—	—	—
65816, MELPS-7700	16M	—	—	—	—	—	—	—	—	—
PPS-4	4K	4K	—	—	—	—	16	—	—	—
MELPS-4500	8K	416	—	—	—	—	—	—	—	—
M16	4G	—	—	—	—	—	—	—	—	—
M16C	1M	—	—	—	—	—	—	—	—	—
PDP-11	64K 256K 4M[10]	— — —	— — —	— — —	— — —	— — —	— — —	— — —	— — —	— — —
VAX	4G	—	—	—	—	—	—	—	—	—
WD16	64K	—	—	—	—	—	—	—	—	—
WE32xxx	4G	—	—	—	—	—	—	—	—	—
4004	4K	256	—	—	—	—	—	—	—	—
8008	16K	8	—	—	—	—	—	—	—	—
MCS-48, MCS-41	1/2/4/6/8K[6]	—	256	256[8]	—	—	—	—	—	—
MCS-51	64K	256	256[1]	64K	—	256	—	—	—	—
80C390	16M	256	256[1]	16M	—	256	—	—	—	—
MCS-251	16M	—	—	—	—	—	512	—	—	—
MCS-96 196(N)/296	64K 16M	—	—	—	—	—	—	—	—	—
8080, 8085	64K	—	—	—	—	—	256	—	—	—
80x86, V20..55	64K	64K	—	64K	—	—	64K	—	—	—
68xx0	4G	—	—	—	—	—	—	—	—	—
8X30x	8K	—	—	—	—	—	—	—	—	—
2650	32K	—	—	—	—	—	—	—	—	—
XA	16M	16M	—	—	—	—	2K[3]	—	—	—

Ziel	CODE	DATA	I-DATA	X-DATA	Y-DATA	BIT-DATA	IO	REG	ROM-DATA	EE-DATA
AVR	128K[6]	32K[6]	—	—	—	—	64	—	—	8K[7]
29XXX	4G	—	—	—	—	—	—	—	—	—
80C166, 80C167	256K 16M	—	—	—	—	—	—	—	—	—
GBZ80 Z80, Z180, Z280, eZ80, Z380	64K 64K 512K[2] 16M[2] 16M 4G	— — —	— — —	— — —	— — —	— — —	—- 256 256 256 64K 4G	— — —	— — —	— — — — —
Z8	64K	256	—	—	—	—	—	—	—	—
eZ8	64K	256	—	64K	—	—	—	—	—	—
Z8001, Z8003	8M	—	—	—	—	—	64K	—	—	—
Z8002, Z8004	64K	—	—	—	—	—	64K	—	—	—
KCPSM	256	256	—	—	—	—	—	—	—	—
KCPSM3	256	64	—	—	—	—	256	—	—	—
Mico8	4096	256	—	—	—	—	256	—	—	—
TLCS- 900(L)	16M	—	—	—	—	—	—	—	—	—
TLCS-90	64K	—	—	—	—	—	—	—	—	—
TLCS- 870(/C)	64K	—	—	—	—	—	—	—	—	—
TLCS-42 TLCS-42	512, 1024	32	—	—	—	—	3, 6	— —	— —	— —
TLCS-47	64K	1K	—	—	—	—	16	—	—	—
TLCS- 9000	16M	—	—	—	—	—	—	—	—	—
TC9331	320	—	—	—	—	—	—	—	—	—
PIC 16C5x	2K	32	—	—	—	—	—	—	—	—
PIC 16C5x	2K	32	—	—	—	—	—	—	—	—
PIC 16C64, 16C86	8K	512	—	—	—	—	—	—	—	256[6]
PIC	64K	256	—	—	—	—	—	—	—	—

Ziel	CODE	DATA	I-DATA	X-DATA	Y-DATA	BIT-DATA	IO	REG	ROM-DATA	EE-DATA
17C42										
SX20	2K	256	—	—	—	—	—	—	—	—
ST6	4K	256	—	—	—	—	—	—	—	—
ST7	64K	—	—	—	—	—	—	—	—	—
STM8	16M	—	—	—	—	—	—	—	—	—
ST9	64K	64K	—	—	—	—	—	256	—	—
6804	4K	256	—	—	—	—	—	—	—	—
32010 32015	4K 4K	144 256	—	—	—	—	8 8	—	—	—
320C2x	64K	64K	—	—	—	—	16	—	—	—
320C3x	16M	—	—	—	—	—	—	—	—	—
320C40	4G	—	—	—	—	—	—	—	—	—
320C44	32M	—	—	—	—	—	—	—	—	—
320C5x/ 320C20x/ 320C54x	64K	64K	—	—	—	—	64K	—	—	—
32060	4G	—	—	—	—	—	—	—	—	—
34010 34020	1G 4G	— —	— —	— —	— —	— —	— —	— —	— —	— —
TMS 9900	64K	—	—	—	—	—	—	—	—	—
TMS 70Cxx	64K	—	—	—	—	—	—	—	—	—
370xxx	64K	—	—	—	—	—	—	—	—	—
MSP430	64K	—	—	—	—	—	—	—	—	—
TMS1000 TMS1200	1K	64	—	—	—	—	—	—	—	—
TMS1100 TMS1300	2K	128	—	—	—	—	—	—	—	—
IMP-16	64K	—	—	—	—	—	128	—	—	—
IPC-16	16K	—	—	—	—	—	—	—	—	—
SC/MP	64K	—	—	—	—	—	—	—	—	—
807x	64K	—	—	—	—	—	—	—	—	—
COP4	512	—	—	—	—	—	—	—	—	—
COP8	8K	256	—	—	—	—	—	—	—	—

Ziel	CODE	DATA	I-DATA	X-DATA	Y-DATA	BIT-DATA	IO	REG	ROM-DATA	EE-DATA
SC144xx	256	—	—	—	—	—	—	—	—	—
NS16008/ NS32008/ NS08032/ NS16032/ NS32016/ NS32032/ NS32CG16	16M	—	—	—	—	—	—	—	—	—
NS32332/ NS32532	4G	—	—	—	—	—	—	—	—	—
CR16A	256K[11]	—	—	—	—	—	—	—	—	—
CR16B	2M	—	—	—	—	—	—	—	—	—
CR16C	2M	—	—	—	—	—	—	—	—	—
ACE	4K[4]	—	—	—	—	—	—	—	—	—
CP-3F/ M380/ LP8000	16K	48	—	—	—	—	8			
F3850	64K	64	—	—	—	—	256	—	—	—
F8	4K	64	—	—	—	—	256	—	—	—
μPD 78(C)xx	64K	—	—	—	—	—	—	—	—	—
μPD 550	640	32	—	—	—	—	—	—	—	—
μPD 554, 652	1000	32	—	—	—	—	—	—	—	—
μPD 547, 552 651	1000	64	—	—	—	—	—	—	—	—
μPD 546, 553 556, 557 650	2000	96	—	—	—	—	—	—	—	—
7566	1K	64	—	—	—	—	—	—	—	—
7508	4K	256	—	—	—	—	16	—	—	—
75K0	16K	4K	—	—	—	—	—	—	—	—
78K0	64K	—	—	—	—	—	—	—	—	—
78K2	1M	—	—	—	—	—	—	—	—	—
78K3	64K	—	—	—	—	—	—	—	—	—

Ziel	CODE	DATA	I-DATA	X-DATA	Y-DATA	BIT-DATA	IO	REG	ROM-DATA	EE-DATA
78K4	16M[5]	—	—	—	—	—	—	—	—	—
7720	512	128	—	—	—	—	—	—	512	—
7725	2K	256	—	—	—	—	—	—	1024	—
77230	8K	—	—	512	512	—	—	—	1K	—
70616	4G	—	—	—	—	—	16M	—	—	—
53C8XX	4G	—	—	—	—	—	—	—	—	—
F²MC8L	64K	—	—	—	—	—	—	—	—	—
F²MC16L	16M	—	—	—	—	—	—	—	—	—
MSM5840	2K	128	—	—	—	—	—	—	—	—
MSM5842	768	32	—	—	—	—	—	—	—	—
MSM58421 MSM58422	1.5K	40	—	—	—	—	—	—	—	—
MSM5847	1.5K	96	—	—	—	—	—	—	—	—
MSM5054	1K	62	—	—	—	—	—	—	—	—
MSM5055	1.75K	96	—	—	—	—	—	—	—	—
MSM5056	1.75K	90	—	—	—	—	—	—	—	—
MSM6051	2.5K	119	—	—	—	—	—	—	—	—
MN1610	64K	—	—	—	—	—	64K	—	—	—
MN1613	256K	—	—	—	—	—	64K	—	—	—
PMCxxx/ PMSxxx/ PFSxxx	1.. 4K[9]	64.. 256[9]	—	—	—	—	32.. 128[9]	—	—	—
180x	64K	—	—	—	—	—	8	—	—	—
XS1	4G	—	—	—	—	—	—	—	—	—
1750	64K	—	—	—	—	—	—	—	—	—
KENBAK	256	—	—	—	—	—	—	—	—	—
CP1600	64K	—	—	—	—	—	—	—	—	—
NANO	2K	—	—	—	—	—	—	—	—	—
IM6100	4K	—	—	—	—	—	—	—	—	—
IM6120	32K	—	—	—	—	—	—	—	—	—
RX...	4G	—	—	—	—	—	—	—	—	—
SC61860	64K	—	—	—	—	—	—	—	—	—
SC62015	1M	—	—	—	—	—	—	—	—	—
TBIL	2K	—	—	—	—	—	—	—	—	—

[1] Initialwert 80h.

Ziel	CODE	DATA	I-DATA	X-DATA	Y-DATA	BIT-DATA	IO	REG	ROM-DATA	EE-DATA
Da der 8051 kein RAM jenseits 80h hat, muß der Initialwert für den 8051 als Zielprozessor auf jeden Fall mit `ORG` angepaßt werden.										
[2] Da Z180 und Z280 weiterhin logisch nur 64K ansprechen können, ist der ganze Adreßraum nur mittels `PHASE`-Anweisungen oder passenden `ASSUME`-Werten für die MMU-Register erreichbar.										
[3] Initialwert 400h.										
[4] Initialwert 800h bzw. 0C00h										
[5] Bereich für Programmcode auf 1 MByte begrenzt										
[6] Größe ist vom Zielprozessor abhängig										
[7] Größe und Verfügbarkeit sind vom Zielprozessor abhängig										
[8] Nur auf Varianten mit `MOVX`-Anweisung										
[9] typabhängig										
[10] modellabhängig										
[11] Nur die ersten 128 KByte sind für ausführbaren Code nutzbar.										

Tabelle 3.1: Adreßbereiche für `ORG`

3.2.2 RORG

Gültigkeit: alle Prozessoren

`RORG` setzt wie `ORG` den Programmzähler neu, erwartet als Argument allerdings keine absolute Adresse, sondern einen relativen Wert (positiv oder negativ), der zum Programmzähler addiert wird. Eine Anwendungsmöglichkeit ist das Freilassen einer bestimmten Menge von Adreßraum, oder die Anwendung in Code-Teilen, die an mehreren Stellen (z.B. via Makros oder Includes) eingebunden werden und lageunabhängig arbeiten sollen. Eine weitere Anwendungsmöglichkeit ergibt sich in Code, der eine Ausführungsadresse unterschiedlich zur Ladeadresse hat (d.h. es wird mit der `PHASE`-Anweisung gearbeitet). Es gibt kein Symbol, über das man in so einer Situation auf die aktuelle *Ladeadresse* zugreifen kann, aber mittels `RORG` kann man sich indirekt darauf beziehen.

3.2.3 CPU

Gültigkeit: alle Prozessoren

Mit diesem Befehl wird festgelegt, für welchen Prozessor im weiteren Code erzeugt werden soll. Die Befehle der anderen Prozessorfamilien sind dann nicht greifbar und erzeugen eine Fehlermeldung!

Die Prozessoren können grob in Familien unterschieden werden, in den Familien dienen unterschiedliche Typen noch einmal zur Feinunterscheidung:

a) 68008 → 68000 → 68010 → 68012 →
MCF5202 → MCF5204 → MCF5206 → MCF5208→
MCF52274 → MCF52277 → MCF5307 → MCF5329 → MCF5373 →
MCF5407 → MCF5470 → MCF5471 → MCF5472 → MCF5473 →
MCF5474 → MCF5475 → MCF51QM →
68332 → 68340 → 68360 →
68020 → 68030 → 68040

In dieser Familie liegen die Unterschiede in hinzukommenden Befehlen und Adressierungsarten (ab 68020). Eine kleine Ausnahme stellt der Schritt zum 68030 dar, dem 2 Befehle fehlen: `CALLM` und `RTM`. Die drei Vertreter der 683xx-Familie haben den gleichen Prozessorkern (eine leicht abgemagerte 68020-CPU), jedoch völlig unterschiedliche Peripherie. MCF5xxx repräsentiert verschiedene ColdFire-Varianten von Motorola/Freescale/NXP, zum 680x0 binär abwärtskompatible RISC-Prozesoren. Beim 68040 kommen die zusätzlichen Steuerregister (via `MOVEC` erreichbar) für On-Chip-MMU und Caches sowie einige Systembefehle für selbige hinzu.

b) 56000 ⟶ 56002 ⟶ 56300

Während der 56002 nur Befehle zum Inkrementieren und Dekrementieren der Akkus ergänzt, ist der 56300-Kern schon fast ein neuer Prozessor: Er vergrößert alle Adreßräume von 64K-Wörtern auf 16M und verdoppelt fast die Anzahl der Befehle.

c) PPC403 → PPC403GC → MPC505 → MPC601 → MPC821 → RS6000

Der PCC403 ist eine abgespeckte Version der PowerPC-Linie ohne Gleitkommaeinheit, demzufolge sind sämtliche Gleitkommabefehle bei ihm gesperrt; dafür sind einige Mikrocontroller-spezifische Befehle enthalten, die er als einziges Mitglied in dieser Familie kennt. Die GC-Variante des PPC403 hat zusätzlich eine MMU und deshalb einige Befehle zu deren Steuerung mehr. Der MPC505 (eine Mikrokontroller-Variante mit FPU) unterscheidet sich solange vom 601er nur in den Peripherieregistern, wie ich es nicht besser weiß - [83] hält sich da noch etwas bedeckt... Die RS6000-Reihe kennt noch einige Befehle mehr (die auf vielen 601er-Systemen emuliert werden, um vollständige Kompatibilität herzustellen), außerdem verwendet IBM z.T. andere Mnemonics für diese reinen Workstation-Prozessoren, als Remineszenz an die 370er-Großrechner...

d) IBM5100, IBM5110, IBM5120

Diese drei Typen referenzieren aktuell alle auf den gleichen (PALM-) Prozessorkern.

e) MCORE

f) XGATE

g) 6800 → 6801 → 6301 → 6811

Während der 6301 nur einige neue Befehle definiert (und der 6301 noch ein paar mehr), bietet der 6811 neben weiteren Befehlen ein zweites Indexregister Y zur Adressierung.

h) 6809/6809UNDOC/6309 und 6805/68HC08/68HCS08

Diese Prozessoren sind zwar teilweise Quellcode-kompatibel zu den anderen 68xx-ern, haben aber ein anderes Binärcode-Format und einen deutlich eingeschränkteren (6805) bzw. erweiterten (6809) Befehlssatz. Der 6309 ist eine CMOS-Version des 6809, die zwar offiziell nur kompatibel zum 6809 ist, inoffiziell aber mehr Register und deutlich mehr Befehle besitzt (siehe [57]). 6809UNDOC schaltet die undokumentierte `HCF`-Instruktion frei.

i) 68HC12 ⟶ 68HC12X

Der 12X-Kern bietet eine Reihe neuer Befehle, bzw. bestehende Befehle wurden um neue Adressierungsarten ergänzt.

j) S912ZVC19F0MKH, S912ZVC19F0MLF,
S912ZVCA19F0MKH, S912ZVCA19F0MLF,
S912ZVCA19F0WKH, S912ZVH128F2CLQ,
S912ZVH128F2CLL, S912ZVH64F2CLQ,
S912ZVHY64F1CLQ, S912ZVHY32F1CLQ,
S912ZVHY64F1CLL, S912ZVHY32F1CLL,
S912ZVHL64F1CLQ, S912ZVHL32F1CLQ,
S912ZVHL64F1CLL, S912ZVHL32F1CLL,
S912ZVFP64F1CLQ, S912ZVFP64F1CLL,
S912ZVH128F2VLQ, S912ZVH128F2VLL,
S912ZVH64F2VLQ, S912ZVHY64F1VLQ,
S912ZVHY32F1VLQ, S912ZVHY64F1VL,
S912ZVHY32F1VLL, S912ZVHL64F1VLQ

Alle Derivate beinhalten den gleichen Prozessorkern und den gleichen Befehlssatz, lediglich die on-Chip-Peripherie und die Menge eingebauten Speichers (RAM, Flash-ROM, EEPROM) variieren.

k) 68HC16

l) 052001

Dieser Baustein ist eine Eigenentwicklung von Konami und in Architektur und Befehlssatz an den Motorola 6809 angelehnt. Er ist jedoch nicht binärkompatibel und stellt auch nicht alle Befehle und Adressierungsarten des Vorbilds zur Verfügung.

m) HD6413308 $\longrightarrow$ HD6413309

Diese beiden Namen repräsentieren die 300er und 300H-Varianten der H8-Familie; die H-Version besitzt dabei einen größeren Adreßraum (16 Mbyte statt 64Kbyte), doppelt so breite Register (32 Bit) und kennt einige zusätzliche Befehle und Adressierungsarten. Trotzdem ist sie binär aufwärtskompatibel.

n) HD6475328 $\longrightarrow$ HD6475348 $\longrightarrow$ HD6475368 $\longrightarrow$ HD6475388

Diese Prozessoren besitzen alle den gleichen CPU-Kern; Die unterschiedlichen Typen dienen lediglich der Einbindung des korrekten Registersatzes in der Datei `REG53X.INC`.

o)
SH7000 (SH1)
SH7201 (SH2A)
SH7205 (SH2A+FPU64)
SH7600 (SH2)
SH7615 (SH2+DSP)
SH7700 (SH3)
SH7720 (SH3+DSP)
SH7750 (SH4)
SH7780 (SH4A)

Es gab sehr viele Varianten von CPUs und MCUs mit SH-Kern. Diese Targets sind Stellvertreter für die verschiedenen Generationen von Kernen und die optionalen Funktionseinheiten. Der SH2-Kern bietet eine Handvoll Befehle mehr, die Lücken im Befehlssatz des SH1 schließen (verzögerte, bedingte sowie relative und indirekte Sprünge, Multiplikationen mit 32-Bit-Operanden sowie Multiplizier/Addier-Befehle). Der SH3-Kern ergänzt eine zweite Registerbank, bessere Schiebebefehle sowie Befehle zur Cache-Steuerung. SH2A ist eine 'Seitenentwicklung' mit Befehlserweiterungen, die nur dort vorhanden sind (Register-indirekte Adressierung mit 12 Bit Displacement, Bitoperationen).

p) HD614023 $\longrightarrow$ HD614043 $\longrightarrow$ HD614081

Diese drei Varianten der HMCS400-Serie unterscheiden sich in der Größe des internen ROM- und RAM-Speichers.

q) HD641016

Dies ist aktuell das einzige Target mit H16-Kern.

r) 6502 $\rightarrow$ 65(S)C02 $\rightarrow$ 65CE02 / W65C02S
65C19 / MELPS740 / HUC6280 / 6502UNDOC

Die CMOS-Version definiert einige zusätzliche Befehle, außerdem sind bei einigen Befehlen Adressierungsarten hinzugekommen, die beim 6502 nicht möglich waren. Der W65SC02 ergänzt den 65C02-Befehlssatz um zwei Befehle, mit denen die Low-Power-Modi der CPU feiner eingestellt werden können. Dem 65SC02 fehlen die Bitmanipulationsbefehle des 65C02. Der 65CE02 ergänzt Sprungbefehle mit 16-Bit-Displacement, ein Z-Register, einen 16-bittigen Stack-Pointer, eine Reihe neuer Befehle und eine programmierbare Base-Page.

Der 65C19 ist *nicht* binär aufwärtskompatibel zum originalen 6502! Einige Adressierungsarten wurden durch andere ersetzt. Des weiteren enthält dieser Prozessor Befehlssatz-Erweiterungen, die die Implementierung digitaler Signalverarbeitung erleichtern.

Die Mitsubishi-Mikrokontroller dagegen erweitern den 6502-Befehlssatz in erster Linie um Bitoperationen und Multiplikations-/Divisionsbefehle. Bis auf den unbedingten Sprung und Befehle zur Inkrementierung/Dekremetierung des Akkumulators sind die Erweiterungen disjunkt.

Das herausstechendste Merkmal des HuC 6280 ist der größere Adreßraum von 2 MByte anstelle 64 KByte, der durch eingebaute Bankregister erreicht wird. Des weiteren existieren einige Sonderbefehle zur Kommunikation mit dem Videoprozessor (dieser Chip wurde in Videospielen eingesetzt) und zum Kopieren von Speicherbereichen.

Mit dem Prozessortyp 6502UNDOC sind die ,,undokumentierten" 6502-Befehle erreichbar, d.h. die Operationen, die sich bei der Verwendung nicht als Befehle definierter Bitkombinationen im Opcode ergeben. Die von AS unterstützten Varianten sind im Kapitel mit den prozessorspezifischen Hinweisen beschrieben.

s) MELPS7700, 65816

Neben einer ,,16-Bit-Version" des 6502-Befehlssatzes bieten diese Prozessoren einige Befehlserweiterungen. Diese sind aber größerenteils disjunkt, da sie sich an ihren jeweiligen 8-bittigen Vorbildern (65C02 bzw. MELPS-740) orientieren. Z.T. werden auch andere Mnemonics für gleiche Befehle verwendet.

t) PPS-4

u) MELPS4500

v) M16

w) M16C

x) PDP-11/03, PDP-11/04, PDP-11/05, PDP-11/10,
PDP-11/15, PDP-11/20, PDP-11/23, PDP-11/24,
PDP-11/34, PDP-11/35, PDP-11/40, PDP-11/44,
PDP-11/45, PDP-11/50, MicroPDP-11/53,
PDP-11/55, PDP-11/60, PDP-11/70,
MicroPDP-11/73, MicroPDP-11/83, PDP-11/84,
MicroPDP-11/93, PDP-11/94, T-11

Die Modelle der PDP-11-Serien unterscheiden sich im Befehlssatz (sowohl dem eingebauten als auch den verfügbaren Erweiterungen) als auch im verfügbaren Adreßraum (64, 256 oder 4096 KByte).

y) WD16

Der WD16 benutzt den gleichen Prozessor wie der LSI-11, lediglich mit anderem Mikrocode. Demensprechend sind Registersatz und Adressierungsarten zur PDP-11 identisch, der Befehlsumfang ist aber leicht unterschiedlich und auf der PDP-11 in gleicher Form vorhandene Maschinenbefehle haben durchgängig andere Kodierungen.

z) MICROVAX-I, MICROVAX-II,
VAX-11/725, VAX-11/730, VAX-11/750,
VAX-11/780, VAX-11/782, VAX-11/785,
VAX-8200, VAX-8300, VAX-8500, VAX-8600
VAX-8650, VAX-8800

Alle Implementierungen der VAX-Architektur unterstützen den gleichen Kern-Befehlssatz. Einige Erweiterungen, wie z.B. Befehle zur Verarbeitung von Strings, gepackten Dezimalzahlen oder bestimmten Gleitkommaformaten, sind nicht immer in Hardware implementiert.

aa) WE32100 → WE32200

Die dritte Generation des WE32xxx-Prozessors bietet sechzehn zusätzliche CPU-Register, und mit diesen zusammen neue Adressierungsmodi wie Indiziert oder Auto-Inkrement und -Dekrement. Des weiteren wurden neue Maschinenbefehle hinzu gefügt.

ab) CP-3F, LP8000, M380

Das Prozessorelement des Chipsatzes ist von AEG/Olympia, GI und SGS-Ates unter den jeweiligen Namen vertrieben worden. Im Befehlssatz und den Adreßräumen bestehen keine Unterschiede.

ac) 4004 → 4040

Der 4040 besitzt gegenüber seinem Vorgänger ein gutes Dutzend zusätzlicher Maschineninstruktionen.

ad) 8008 → 8008NEW

Intel hat 1975 die Mnemonics des 8008 umdefiniert, die zweite Variante spiegelt diesen neuen Befehlssatz wieder. Eine gleichzeitige Unterstützung beider Varianten war nicht möglich, da teilweise Überschneidungen vorliegen.

ae) 8021, 8022,
8401, 8411, 8421, 8461,
8039, (MSM)80C39, 8048, (MSM)80C48, 8041, 8042,
80C382

Bei den ROM-losen Versionen 8039 und 80C39 sind die Befehle verboten, die den BUS (Port 0) ansprechen. Der 8021 und 8022 sind Sonderversionen mit stark abgemagertem Befehlssatz, wofür der 8022 zwei A/D-Wandler und die dazugehörigen Steuerbefehle enthält. MAB8401 bis 8461 sind von Philips entwickelte Derivate, die in ihrem Befehssatz irgendwo zwischen dem 8021/8022 und einem 'vollständigen" 8048 stehen. Dafür verfügen sie über serielle Ports und je nach Variante bis zu 8 KByte Programmspeicher.

Die CMOS-Versionen lassen sich mit dem `IDL`- bzw. `HALT`-Befehl in einen Ruhezustand niedriger Stromaufnahme überführen. Der 8041 und 8042 haben einige Zusatzbefehle zur Steuerung der Busschnittstelle, dafür fehlen aber einige andere Befehle. Beim 8041, 8042, 84x1, 8021 und 8022 ist der Programmadreßraum nicht extern erweiterbar, weshalb AS das Codesegment bei diesen Prozessoren auf die Größe des internen ROM beschränkt. Der (SAB)80C382 ist eine von Siemens speziell für Telefone entwickelte Variante, die ebenfalls einen `HALT`-Befehl kennt sowie `DJNZ` und `DEC` auch mit indirekter Adressierung erlaubt. Im Gegenzug wurden einige Befehle des 'normalen' 8048 entfernt. Die OKI-Varienaten (MSM...) unterstützen ebenfalls `DJNZ` und `DEC` mit indirekter Adressierung, sowie eine erweiterte Steuerung der Power-Down-Modi, ohne den Basis-MCS-48-Befehlssatz zu beschneiden.

af) 87C750 → 8051, 8052, 80C320, 80C501, 80C502,
80C504, 80515, und 80517
→ 80C390
→ 80C251

Der 87C750 kann nur max. 2 Kbyte Programmspeicher adressieren, weshalb die `LCALL`- und `LJMP`-Befehle bei ihm fehlen. Zwischen den acht mittleren Prozessoren nimmt AS selber überhaupt keine Unterscheidung vor, sondern verwaltet den Unterschied lediglich in der Variablen `MOMCPU` (s.u.), die man mit `IF`-Befehlen abfragen kann. Eine Ausnahme stellt lediglich der 80C504, der in seiner momentanen Form noch einen Maskenfehler zeigt, wenn eine `AJMP`- oder `ACALL`-Anweisung auf der vorletzten Adresse einer 2K-Seite steht. AS benutzt in einem solchen Fall automatisch lange Sprungbefehle bzw. gibt eine Fehlermeldung aus. Der 80C251 hingegen stellt einen drastischen Fortschritt in Richtung 16/32 Bit, größerer Adreßräume und orthogonalerem Befehlssatz dar. Den 80C390 könnte man vielleicht als die 'kleine Lösung' bezeichnen: Dallas Semiconductor hat den Befehlssatz und die Architektur nur so weit verändert, wie es für die 16 MByte großen Adreßräume notwendig war.

ag) 8096 → 80196 → 80196N → 80296

Neben einem anderen Satz von SFRs (die übrigens von Unterversion zu Unterversion stark differieren) kennt der 80196 eine Reihe von zusätzlichen Befehlen und kennt einen „Windowing"-Mechanismus, um das größere interne RAM anzusprechen. Die 80196N-Familie wiederum erweitert den Adreßraum auf 16 Mbyte und führt eine Reihe von Befehlen ein, mit denen man auf Adressen jenseits 64 Kbyte zugreifen kann. Der 80296 erweitert den CPU-Kern um Befehle zur Signalverarbeitung und ein zweites Windowing-Register, verzichtet jedoch auf den *Peripheral Transaction Server* (PTS) und verliert damit wieder zwei Maschinenbefehle.

ah) 8080 → V30EMU → 8085 → 8085UNDOC

Der 8085 kennt zusätzlich die Befehle `RIM` und `SIM` zum Steuern der Interruptmaske und der zwei I/O-Pins. Der Typ `8085UNDOC` schaltet zusätzliche, nicht von Intel dokumentierte Befehle ein. Diese Befehle sind in Abschnitt 4.23 dokumentiert.

`V30EMU` als Ziel schaltet gegenüber einem 8080 die Befehle `RETEM` und *CALLN* frei, mit denen die 8080-Emulation auf einem V20/V30/V40/V50 verlassen bzw. unterbrochen werden kann.

ai) 8088,8086
→ 80188,80186
→ V20,V30,V40,V50
→ V33,V53
→ V25,V35
→ V55
→ V55SC
→ V55PI

Prozessoren in der gleichen Zeile verfügen über den gleichen CPU-Kern und damit den gleichen Befehlssatz. Von Zeile zu Zeile kommen neue Befehle hinzu, wobei die NEC-CPUs ausgehend vom 'V20-Basisbefehlssatz' jeweils über unterschiedliche Erweiterungen verfügen.

aj) 80960

ak) 8X300 → 8X305

Der 8X305 besitzt eine Reihe zusätzlicher Arbeitsregister, die dem 8X300 fehlen und kann mit diesen auch zusätzliche Operationen ausführen, wie das direkte Schreiben von 8-Bit-Werten auf Peripherieadressen.

al) XAG1, XAG2, XAG3

Diese Prozessoren unterscheiden sich nur in der Größe des eingebauten ROMs, die in `STDDEFXA.INC` definiert ist.

am) AT90S1200, AT90S2313, AT90S2323, AT90S233, AT90S2343, AT90S4414, AT90S4433, AT90S4434, AT90S8515, AT90C8534, AT90S8535, ATTINY4, ATTINY5, ATTINY9, ATTINY10, ATTINY11, ATTINY12, ATTINY13, ATTINY13A, ATTINY15, ATTINY20, ATTINY24(A), ATTINY25, ATTINY26, ATTINY28, ATTINY40, ATTINY44(A), ATTINY45, ATTINY48, ATTINY84(A), ATTINY85, ATTINY87, ATTINY88, ATTINY102, ATTINY104, ATTINY167, ATTINY261, ATTINY261A, ATTINY43U, ATTINY441, ATTINY461, ATTINY461A, ATTINY828, ATTINY841, ATTINY861, ATTINY861A, ATTINY1634, ATTINY2313, ATTINY2313A, ATTINY4313, ATMEGA48, ATMEGA8, ATMEGA8515, ATMEGA8535, ATMEGA88, ATMEGA8U2, ATMEGA16U2, ATMEGA32U2, ATMEGA16U4, ATMEGA32U4, ATMEGA32U6, AT90USB646, AT90USB647, AT90USB1286, AT90USB1287, AT43USB355, ATMEGA16, ATMEGA161, ATMEGA162, ATMEGA163, ATMEGA164, ATMEGA165, ATMEGA168, ATMEGA169, ATMEGA32, ATMEGA323, ATMEGA324, ATMEGA325, ATMEGA3250, ATMEGA328, ATMEGA329, ATMEGA3290, ATMEGA406, ATMEGA64, ATMEGA640, ATMEGA644, ATMEGA644RFR2, ATMEGA645, ATMEGA6450, ATMEGA649, ATMEGA6490, ATMEGA103, ATMEGA128, ATMEGA1280, ATMEGA1281, ATMEGA1284, ATMEGA1284RFR2, ATMEGA2560, ATMEGA2561

Die verschiedenen AVR-Varianten unterscheiden sich in erster Linie in der Größe des On-Chip-Speichers (Flash, SRAM, EEPROM) und der integrierten Peripherie (GPIO, Timer, UART, A/D-Wandler,...). Die ATmegas bringen im Vergleich zu den AT90...-Vorgängern auch neue Maschinenbefehle mit, den ATtinys fehlen wiederum die Multiplikationsbefehle.

an) AM29245 → AM29243 → AM29240 → AM29000

Je weiter man sich in der Liste nach rechts bewegt, desto weniger Befehle müssen in Software emuliert werden. Während z.B. der 29245 noch nicht einmal einen Hardware-Multiplizierer besitzt, fehlen den beiden Vertretern in der Mitte nur die Gleitkommabefehle. Der 29000 dient dabei als ,,generischer" Typ, der alle Befehle in Hardware versteht.

ao) 80C166 ⟶ 80C167,80C165,80C163

80C167 und 80C165/163 haben anstelle 256 Kbyte max. 16 Mbyte Adreßraum, außerdem kennen sie einige zusätzliche Befehle für erweiterte Adressierungsmodi sowie atomare Befehlssequenzen. Untereinander unterscheiden sich diese Prozessoren der ,,zweiten Generation" nur in der eingebauten Peripherie.

ap) LR35902/GBZ80 → Z80 → Z80UNDOC
→ Z180
→ eZ80190, eZ80L92, eZ80F91,
eZ80F92, eZ80F93,
→ Z280
→ Z380

Während für den Z180 nur die zusätzlichen Befehle definiert sind (d.h. die Z180-MMU findet noch keine Berücksichtigung), besitzt der Z380 32-Bit-Register, einen linearen 4Gbyte-Adreßraum sowie neben einer Reihe von Befehlserweiterungen, die den Befehlssatz deutlich orthogonaler machen, neue Adressierungsmodi (Ansprechen der Indexregisterhälften, Stack-relativ). Zu einem kleinen Teil existieren diese Erweiterungen aber auch schon beim Z80 als undokumentierte Befehle, die mit der Variante `Z80UNDOC` zugeschaltet werden können. Eine Liste mit den zusätzlichen Befehlen findet sich im Kapitel mit den prozessorspezifischen Hinweisen.

Der im Gameboy verbaute Prozessor (offizielle Bezeichnung LR35902, umgangssprachlich auch als ,,Gameboy-Z80" bezeichnet) ist eine Mischung aus Z80 und 8080. Ihm fehlen die IX/IY-Register, der I/O-Adreßraum, die zweite Registerbank sowie eine Reihe von 16-Bit-Befehlen.

Die Zilog eZ80 Varianten erweitern die Z80 Architektur um einem 16 MByte großen Adreßraum, 24 Bit breite Register sowie eine begrenzte Zahl neuer Befehle. Einige Varianten enthalten ein I-Register, das nur 8 Bit breit ist. Dem eZ80190 fehlen zusätzlich einige String-I/O Befehle. Ansonsten liegen die Unterschiede nur in der Menge des eingebauten Speichers sowie der integrierten Peripherie.

aq) Z8601, Z8603, Z86C03, Z86E03, Z86C06, Z86E06,
Z86C08, Z86C21, Z86E21, Z86C30, Z86C31, Z86C32 Z86C40
→ Z88C00, Z88C01
→ eZ8, Z8F0113, Z8F011A, Z8F0123, Z8F012A,
Z8F0130, Z8F0131, Z8F0213, Z8F021A, Z8F0223, Z8F022A,
Z8F0230, Z8F0231, Z8F0411, Z8F0412, Z8F0413, Z8F041A,
Z8F0421, Z8F0422, Z8F0423, Z8F042A, Z8F0430, Z8F0431,
Z8F0811, Z8F0812, Z8F0813, Z8F081A, Z8F0821, Z8F0822,
Z8F0823, Z8F082A, Z8F0830, Z8F0831, Z8F0880, Z8F1232,
Z8F1233, Z8F1621, Z8F1622, Z8F1680, Z8F1681, Z8F1682,
Z8F2421, Z8F2422, Z8F2480, Z8F3221, Z8F3222, Z8F3281,
Z8F3282, Z8F4821, Z8F4822, Z8F4823, Z8F6081, Z8F6082,
Z8F6421, Z8F6422, Z8F6423, Z8F6481, Z8F6482

Die Varianten mit Z8-Kern unterscheiden sich nur in Speicherausbau und Peripherie, d.h. die Wahl hat auf den unterstützten Befehlssatz keinen Effekt. Deutlich anders sind jedoch die Super8- und eZ8-Varianten, jeweils mit (in unterschiedliche Richtungen) stark erweiterten Befehlssätzen, die auch auf Quellcode-Ebene nur gößtenteils aufwärts-kompatibel sind.

ar) Z8001, Z8002, Z8003, Z8004

Über die Wahl des Prozessors wird die Betriebsart (segmentiert für Z8001 und Z8003, nicht segmentiert für Z8002 und Z8004) bestimmt. Eine Unterscheidung zwischen dem Z8001/8002 einerseits und Z8003/8004 andererseits findet aktuell nicht statt.

as) KCPSM, KCPSM3

Bei beiden Prozessorkernen handelt es sich um keine eigenständigen Bausteine, sondern Logik-Kerne für Gate-Arrays der Firma Xilinx. Die 3er-Variante bietet einen größeren Adreßraum sowie einige zusätzliche Instruktionen. Es ist zu beachten, daß sie nicht binär aufwärtskompatibel ist!

at) MICO8_05, MICO8_V3, MICO8_V31

Leider hat Lattice die Maschinencodes des Mico8 mehrfach geändert, so daß verschiedene Targets notwendig wurden, um auch alte Designs weiter zu unterstützen. Die erste Variante entspricht der Variante, wie sie im 2005er-Manual beschrieben wurde, die beiden anderen die Versionen 3.0 bzw. 3.1.

au) 96C141, 93C141

Diese beiden Prozessoren repräsentieren die beiden Varianten der Prozessorfamilie: TLCS-900 und TLCS-900L. Die Unterschiede dieser beiden Varianten werden in Abschnitt 4.34 genauer beleuchtet.

av) 90C141

aw) 87C00, 87C20, 87C40, 87C70

Die Prozessoren der TLCS-870-Reihe haben zwar den identischen CPU-Kern, je nach Variante aber eine unterschiedliche Peripherieausstattung. Zum Teil liegen Register gleichen Namens auf unterschiedlichen Adressen. Die Datei STDDEF87.INC benutzt analog zur MCS-51-Familie die hier mögliche Unterscheidung, um automatisch den korrekten Symbolsatz bereitzustellen.

ax) TLCS-870/C

Momentan ist nur der Prozessorkern der TLCS-870/C-Familie implementiert.

ay) 47C00 → 470C00 → 470AC00

Diese drei Varianten der TLCS-47-Familie haben unterschiedlich große RAM-und ROM-Adreßbereiche, wodurch jeweils einige Befehle zur Bankumschaltung hinzukommen oder wegfallen.

az) 4240P, 4250N, 4260P, 4270N
42C00Y, 42C40P, 42C50N, 42C60P, 42C70N

Die CMOS-Varianten verfügen über zwei zusätzliche Befehle. Ansonsten unterscheiden die Varianten sich in der Anzahl der I/O-Ports und der Größe des Proramm-speichers.

ba) 97C241

bb) TC9331

bc) 16C54 → 16C55 → 16C56 → 16C57

Diese Prozessoren unterscheiden sich durch den verfügbaren Adreßraum im Programmspeicher, d.h. durch die Adresse, ab der der AS Überläufe anmeckert.

bd) 16C64, 16C84

Analog zur MCS-51-Familie findet hier keine Unterscheidung im Codegenerator statt, die unterschiedlichen Nummern dienen lediglich der Einblendung der korrekten SFRs in STDDEF18.INC.

be) 17C42

bf) SX20, SX28

Der SX20 steckt in einem kleineren Gehäuse, weshalb der Port C fehlt.

bg) ST6200, ST6201, ST6203, ST6208, ST6209,
ST6210, ST6215, ST6218, ST6220, ST6225,
ST6228, ST6230, ST6232, ST6235, ST6240,
ST6242, ST6245, ST6246, ST6252, ST6253,
ST6255, ST6260, ST6262, ST6263, ST6265,
ST6280, ST6285

Die einzelnen ST6-Varianten differieren in der Menge der On-Chip-Peripherie und dem eingebauten Speicher.

bh) ST7
ST72251G1, ST72251G2, ST72311J2, ST72311J4,
ST72321BR6, ST72321BR7, ST72321BR9, ST72325S4,
ST72325S6, ST72325J7, ST72325R9, ST72324J6,
ST72324K6, ST72324J4, ST72324K4, ST72324J2,
ST72324JK21, ST72325S4, ST72325J7, ST72325R9,
ST72521BR6, ST72521BM9, ST7232AK1, ST7232AK2,
ST7232AJ1, ST7232AJ2, ST72361AR4, ST72361AR6,
ST72361AR7, ST72361AR9, ST7FOXK1, ST7FOXK2,
ST7LITES2Y0, ST7LITES5Y0, ST7LITE02Y0,
ST7LITE05Y0, ST7LITE09Y0
ST7LITE10F1, ST7LITE15F1, ST7LITE19F1,

ST7LITE10BF0, ST7LITE15BF0, ST7LITE15BF1,
ST7LITE19BF0, ST7LITE19BF1,
ST7LITE20F2, ST7LITE25F2, ST7LITE29F2,
ST7LITE30F2, ST7LITE35F2, ST7LITE39F2,
ST7LITE49K2,
ST7MC1K2, ST7MC1K4, ST7MC2N6, ST7MC2S4,
ST7MC2S6, ST7MC2S7, ST7MC2S9, ST7MC2R6,
ST7MC2R7, ST7MC2R9, ST7MC2M9,
STM8
STM8S001J3, STM8S003F3, STM8S003K3, STM8S005C6,
STM8S005K6, STM8S007C8, STM8S103F2, STM8S103F3,
STM8S103K3, STM8S105C4, STM8S105C6, STM8S105K4,
STM8S105K6, STM8S105S4, STM8S105S6, STM8S207MB,
STM8S207M8, STM8S207RB, STM8S207R8, STM8S207R6,
STM8S207CB, STM8S207C8, STM8S207C6, STM8S207SB,
STM8S207S8, STM8S207S6, STM8S207K8, STM8S207K6,
STM8S208MB, STM8S208RB, STM8S208R8, STM8S208R6,
STM8S208CB, STM8S208C8, STM8S208C6, STM8S208SB,
STM8S208S8, STM8S208S6, STM8S903K3, STM8S903F3,
STM8L050J3, STM8L051F3, STM8L052C6, STM8L052R8,
STM8L001J3, STM8L101F1, STM8L101F2, STM8L101G2,
STM8L101F3, STM8L101G3, STM8L101K3, STM8L151C2,
STM8L151K2, STM8L151G2, STM8L151F2, STM8L151C3,
STM8L151K3, STM8L151G3, STM8L151F3, STM8L151C4,
STM8L151C6, STM8L151K4, STM8L151K6, STM8L151G4,
STM8L151G6, STM8L152C4, STM8L152C6, STM8L152K4,
STM8L152K6, STM8L151R6, STM8L151C8, STM8L151M8,
STM8L151R8, STM8L152R6, STM8L152C8, STM8L152K8,
STM8L152M8, STM8L152R8, STM8L162M8, STM8L162R8,
STM8AF6366, STM8AF6388, STM8AF6213, STM8AF6223,
STM8AF6226, STM8AF6246, STM8AF6248, STM8AF6266,
STM8AF6268, STM8AF6269, STM8AF6286, STM8AF6288,
STM8AF6289, STM8AF628A, STM8AF62A6, STM8AF62A8,
STM8AF62A9, STM8AF62AA, STM8AF5268, STM8AF5269,
STM8AF5286, STM8AF5288, STM8AF5289, STM8AF528A,
STM8AF52A6, STM8AF52A8, STM8AF52A9, STM8AF52AA,
STM8AL3136, STM8AL3138, STM8AL3146, STM8AL3148,
STM8AL3166, STM8AL3168, STM8AL3L46, STM8AL3L48,
STM8AL3L66, STM8AL3L68, STM8AL3188, STM8AL3189,
STM8AL318A, STM8AL3L88, STM8AL3L89, STM8AL3L8A,
STM8TL52F4, STM8TL52G4, STM8TL53C4, STM8TL53F4,

STM8TL53G4

Der STM8-Kern erweitert den Adreßraum auf bis zu 16 MByte und führt eine ganze Reihe neuer Befehle ein. Obwohl viele Befehle den gleichen Maschinencode wie beim ST7 haben, ist er nicht binär aufwärtskompatibel.

bi) ST9020, ST9030, ST9040, ST9050

Diese 4 Namen vetreten die vier ,,Unterfamilien" der ST9-Familie, die sich durch eine unterschiedliche Ausstattung mit On-Chip-Peripherie auszeichnen. Im Prozessorkern sind sie identisch, so daß diese Unterscheidung wieder nur in der Include-Datei mit den Peripherieadressen zum Zuge kommt.

bj) 6804

bk) 32010 → 32015

Der TMS32010 besitzt nur 144 Byte internes RAM, weshalb AS Adressen im Datensegment auf eben diesen Bereich begrenzt. Für den 32015 gilt diese Beschränkung nicht, es kann der volle Bereich von 0–255 angesprochen werden.

bl) 320C25 → 320C26 → 320C28

Diese Prozessoren unterscheiden sich nur leicht in der On-Chip-Peripherie sowie den Konfigurationsbefehlen.

bm) 320C30, 320C31 → 320C40, 320C44

Der 320C31 ist eine etwas ,,abgespeckte" Version des 320C30 mit dem gleichen Befehlssatz, jedoch weniger Peripherie. In STDDEF3X.INC wird diese Unterscheidung ausgenutzt. Die C4x-Varianten sind Quellcode-aufwärtskompatibel, unterscheiden sich im Maschinencode einiger Befehle jedoch subtil. Auch hier ist ist der C44 eine abgespeckte Version des C40, mit weniger Peripherie und kleinerem Adreßraum.

bn) 320C203 → 320C50, 320C51, 320C53

Ersterer ist der generelle Repräsentant für die C20x-Signalprozessorfamilie, die eine Untermenge des C5x-Befehlssatzes implementieren. Die Unterscheidung zwischen den verschiedenen C5x-Prozessoren wird von AS momentan nicht ausgenutzt.

bo) 320C541

Dies ist momentan der Repräsentant für die TMS320C54x-Familie...

bp) 32060

bq) 34010 → 34020

Der TMS34020 unterstützt eine vollen 4 GBit-Adresraum, zusätzliche Maschinenbefehle sowie eine Koprozessor-Schnittstelle.

br) TI990/4, TI990/10, TI990/12
TMS9900, TMS9940, TMS9995, TMS99105, TMS99110

Die TMS99xx/99xxx-Prozessoren sind im Prinzip Single-Chip-Implementierungen der TI990-Minicomputer, einige TI990-Modelle basieren auch auf einem solchen Prozessor anstatt einer diskret aufgebauten CPU. Die einzelnen Modelle unterscheiden sich im Befehlssatz (der TI990/12 hat den größten), und dem Vorhandensein eines privilegierten Modus.

bs) TMS70C00, TMS70C20, TMS70C40,
TMS70CT20, TMS70CT40,
TMS70C02, TMS70C42, TMS70C82,
TMS70C08, TMS70C48

Alle Mitglieder dieser Familie haben den gleichen CPU-Kern, unterscheiden sich im Befehlssatz also nicht. Die Unterschiede finden sich nur in der Datei REG7000.INC, in der Speicherbereiche und Peripherieadressen definiert werden. Die in einer Zeile stehenden Typen besitzen jeweils gleiche Peripherie und gleiche interne RAM-Menge, unterscheiden sich also nur in der Menge eingebauten ROMs.

bt) 370C010, 370C020, 370C030, 370C040 und 370C050

Analog zur MCS-51-Familie werden die unterschiedlichen Typen nur zur Unterscheidung der Peripherie in STDDEF37.INC genutzt, der Befehlssatz ist identisch.

bu) MSP430 → MSP430X

Die X-Variante des CPU-Kerns erweitert den Adreßraum von 64 KiByte auf 1 MiByte und erweitert den Befehlssatz, um Instruktionen mehrfach ausführen zu können.

bv) TMS1000, TMS1100, TMS1200, TMS1300

Für TMS1000 und TMS1200 sind jeweils 1 KByte ROM und 64 Nibbles RAM vorgesehen, für TMS1100 und TMS1300 jeweils das doppelte. Des weiteren hat TI für TMS1100 und TMS1300 einen deutlich anderen Default-Befehlssatz vorgesehen (AS kennt nur die Default- Befehlssätze!).

bw) IMP-16C/200, IMP-16C/300, IMP-16P/200, IMP-16P/300, IMP-16L

Beim IMP-16L sind einige zusätzliche Bits im Statusregister sowie weitere Sprungbedingungen definiert. Den erweiterten Befehlssatz unterstützt er ebenso wie die 300er-Varianten.

bx) IPC-16, INS8900

Der INS8900 ist lediglich eine Re-Implementierung des PACE in einem modernerem NMOS-Fertigungsprozess; es bestehen keine Unterschiede im Befehlssatz.

by) SC/MP

bz) 8070

Dieser Prozessor repräsentiert die gesamte 807x-Familie (die mindestens aus den 8070, 8072 und 8073 besteht), der jedoch ein einheitlicher CPU-Kern gemeinsam ist.

ca) COP87L84

Dies ist das momentan einzige unterstützte Mitglied der COP8-Familie von National Semiconductor. Mir ist bekannt, daß die Familie wesentlich größer ist und auch Vertreter mit unterschiedlich großem Befehlssatz existieren, die nach Bedarf hinzukommen werden. Es ist eben ein Anfang, und die Dokumentation von National ist ziemlich umfangreich...

cb) COP410 → COP420 → COP440 → COP444

Die COP42x-Derivate bieten einige weitere Befehle, des weiteren wurden Befehlen in ihrem Wertebereich erweitert.

cc) SC14400, SC14401, SC14402, SC14404, SC14405,
SC14420, SC14421, SC14422, SC14424

Diese Gruppe von DECT-Controller unterscheidet sich in ihrem Befehlsumfang, da jeweils unterschiedliche B-Feld Datenformate unterstützt werden und deren Architektur im Laufe der Zeit optimiert wurde.

cd) NS16008, NS32008, NS08032, NS16032, NS32016, NS32032,
NS32332, NS32CG16, NS32532

National hat in den ersten Jahren die CPUs der ersten Generation mehrfach umbenannt, hinter NS16008/NS32008/NS08032 bzw. NS16032/NS32016 versteckt sich der gleiche Baustein. NS32332 und NS32532 unterstützen eine Adreßraum von 4 GByte statt 16 MByte, und der NS32CG16 ist eine Embedded-Variante mit Zusatzbefehlen für Bitblock-Transfers.

ce) CR16A → CR16B → CR16C

CR16B erweitert den Adreßraum von 256 KByte auf 2 MByte und stellt eine Reihe neuer Befehle zur Verfügung. CR16C erweitert ihn nochmals auf 16 MByte, ist aber nur noch Quellcode- und nicht mehr binär aufwärtskompatibel.

cf) ACE1101, ACE1202

cg) F3850, MK3850,
MK3870, MK3870/10, MK3870/12, MK3870/20, MK3870/22,
MK3870/30, MK3870/32, MK3870/40, MK3870/42,
MK3872, MK3873, MK3873/10, MK3873/12, MK3873/20, MK3873/22,
MK3874, MK3875, MK3875/22, MK3875/42, MK3876, MK38P70/02,
MK38C70, MK38C70/10,
MK38C70/20, MK97400, MK97410, MK97500, MK97501, MK97503

Die große Menge an Varianten ergibt sich zum Teil daraus, daß Mostek Anfang der 80er-Jahre diversen Varianten neue Namen gegeben hat. Am neuen Benamungsschema kann man am Suffix die Menge internen ROMs (0 bis 4 für 0..4 KByte) bzw. die Menge des eingebauten Executable RAM (0 oder 2 für 0 oder 64 Byte) ablesen. 3850 und MK975xx unterstützen einen 64 KByte großen Adreßraum, beim Rest ist er 4 KByte groß. P-Varianten haben einen Piggyback-Sockel für ein EPROM, C-Varianten sind in CMOS ausgeführt und kennen zwei neue Maschinenbefehle (HET und HAL). Der MK3873 enthält als ,,Spezialität" eine eingebaute serielle Schnittstelle, der MK3875 bietet einen zweiten Betriebsspannungsanschluß, um den internen RAM-Inhalt im Standby halten zu können.

ch) 7800, 7801, 7802
78C05, 78C06
7807, 7808, 7809
7810 → 78C10, 78C11, 78C12, 78C14, 78C17, 78C18

μPD7800 bis μPD7802 repräsentieren die "erste Generation" der uCOM87-Familie von NEC. μPD78C05 und μPD78C06 sind davon abgespeckte Versionen, die nur eine Untermenge des Befehlssatzes unterstützen. 7807 bis 7809 stellen die uCOM87-Serie dar, die um weitere Befehle und ergänzt wurde. Alle μPD781x-Varianten gehören zur uCOM87AD-Serie, die darüber hinaus um einen A/D-Wandler ergänzt wurde, dafür wurden die Befehle zur Verarbeitung von Bits wieder entfernt. **ACHTUNG!** Der Befehlssatz ist generell nur teilweise binär aufwärtskompatibel! Die NMOS-Version μPD7810 besitzt keinen STOP-Modus; der entsprechende Befehl sowie das ZCM-Register fehlen demzufolge. **VORSICHT!** NMOS- und CMOS-Version differieren zum Teil in den Reset-Werten einiger Register!

ci) uPD550, uPD554, uPD652,
uPD547, uPD552, uPD651,
uPD546, uPD553, uPD556, uPD557, uPD650, iso650

Die drei Gruppen vom Controllers gehören jeweils zur μCOM-45-, μCOM-44- und μCOM-43-Familie. Die ersten beiden Familien implementieren eine Untermenge des μCOM-43-Befehlssatzes. Ansonsten unterscheiden die Varianten sich in der Menge des integrierten ROM- und RAM-Speichers. iso650 ist eine FPGA-Implementierung des uPD650 und unterscheidet sich vom 'Original' durch eine NAND-Anweisung sowie einen von 2000 auf 4096 Byte vergrößerten Adreßraum.

cj) 7566 ↔ 7508

Es existieren in der μPD75xx-Familie zwei verschiedene CPU-Kerne: Der 7566 repräsentiert den 'instruction set B', der deutlich weniger Befehle, einige Register weniger und kleinere Adreßräume erlaubt. Der 7508 repräsentiert den 'vollen' Befehlssatz A. **VORSICHT!** Beide Maschinen-Befehlssätze sind nicht 100%-ig binärkompatibel!

ck) 75402,
75004, 75006, 75008,
75268,
75304, 75306, 75308, 75312, 75316,
75328,
75104, 75106, 75108, 75112, 75116,
75206, 75208, 75212, 75216,
75512, 75516

Dieses ,,Füllhorn" an Prozessoren unterscheidet sich innerhalb einer Gruppe nur durch die RAM- und ROM-Größe; die Gruppen untereinander unterscheiden sich einmal durch ihre on-chip-Peripherie und zum anderen durch die Mächtigkeit des Befehlssatzes.

cl) 78070

Dies ist das einzige, mir momentan vertraute Mitglied der 78K0-Familie von NEC. Es gelten ähnliche Aussagen wie zur COP8-Familie!

cm) 78214

Dies ist momentan der Repräsentant der 78K2-Familie von NEC.

cn) 78310

Dies ist momentan der Repräsentant der 78K3-Familie von NEC.

co) 784026

Dies ist momentan der Repräsentant der 78K4-Familie von NEC.

cp) 7720 $\rightarrow$ 7725

Der μPD7725 bietet im Vergleich zu seinem Vorgänger größere Adreßräume und einige zusätzliche Befehle. **VORSICHT!** Die Prozessoren sind nicht zueinander binärkompatibel!

cq) 77230

cr) 70616

Dies ist momentan der Repräsentant der V60-Familie von NEC.

cs) SYM53C810, SYM53C860, SYM53C815, SYM53C825, SYM53C875, SYM53C895

Die einfacheren Mitglieder dieser Familie von SCSI-Prozessoren besitzen einige Befehlsvarianten nicht, außerdem unterscheiden sie sich in ihrem Satz interner Register.

ct) MB89190

Dieser Prozessortyp repräsentiert die F^2MC8L-Serie von Fujitsu...

cu) MB9500

...so wie dieser es momentan für die 16-Bit-Varianten von Fujitsu tut!

cv) MSM5840, MSM5842, MSM58421, MSM58422, MSM5847

Diese Varianten der OLMS-40-Familie unterscheiden sich im Befehlssatz sowie im internen Programm- und Datenspeicher.

cw) MSM5054, MSM5055, MSM5056, MSM6051, MSM6052

Gleiches wie bei der OLMS-40-Familie: Unterschiede im Befehlssatz sowie im internen Programm- und Datenspeicher.

cx) MN1610[ALT] $\rightarrow$ MN1613[ALT]

Zusätzlich zu den Funktionen seines Vorgängers bietet der MN1613 einen größeren Adreßraum, eine Floating-Point-Einheit sowie eine ganze Reihe neuer Befehle.

cy) RXV1, RX110, RX111, RX113, RX130, RX210,
RX21A, RX220, RX610, RX621, RX62N, RX630,
RX631 ⟶
RXV2, RX140, RX230, RX231, RX64M,
RX651 ⟶
RXV3, RX660, RX671, RX72M, RX72N

Die Controller der RX-Serie können grob in drei Gruppen bzw. Generationen eingeteilt werden, wobei von Generation zu Generation (RXv1, RXv2, RXv3) jeweils neue Instruktionen hinzu gekommen sind.

cz) PMC150, PMS150, PFS154, PMC131, PMS130, PMS131
PMS132, PMS132B, PMS152, PMS154B, PMS154C, PFS173
PMS133, PMS134, DF69, MCS11, PMC232, PMC234, PMC251
PMC271,PMC884, PMS232, PMS234, PMS271

Die Padauk-Controller unterscheiden sich allesamt in der Größe des internen (ROM/RAM)-Speichers, der Art des internen ROMs (löschbar oder OTP), der eingebauten Peripherie sowie in Umfang und Kodierung des Befehlssatzes.

da) 1802 → 1804, 1805, 1806 → 1804A, 1805A 1806A

1804, 1805 und 1806 haben gegenüber dem 'Original' 1802 einen leicht erweiterten Befehlssatz sowie on-chip-RAM und einen integrierten Timer. Die A-Versionen erweitern den Befehlssatz um `DSAV`, `DBNZ`, sowie um Befehle für Addition und Subtraktion im BCD-Format.

db) XS1

Dieser Typ repräsentiert die XCore-"Familie".

dc) 1750

MIL STD 1750 ist ein Standard, also gibt es auch nur eine (Standard-)Variante...

dd) KENBAK

Es hat nie einen KENBAK-2 gegeben...

de) CP-1600

df) HPNANO

dg) 6100 → 6120

Der IM6120 unterstützt einen größeren Adreßraum (32K anstelle 4K) sowie zusätzliche Maschinenbefehle.

dh) SC61860

Dies ist der in den meisten Sharp Pocket Computern (PC-12xx...PC-15xx) verbaute Prozessor.

di) SC62015

Dies ist der im Sharp PC-E500 verbaute Prozessor.

Ein spezielles, bisher hier nicht aufgelistetes Target ist NONE. Es ist als Default eingestellt, wenn kein Target per `-cpu` Argument auf der Kommandozeile vorgegeben wurde, und auch bisher keine `CPU`-Anweisung assembliert wurde. Target-unabhängige Pseudo-Befehle sind in diesem Zustand weiter möglich, aber es kann kein Code erzeugt werden, weder durch Maschinenbefehle noch durch Datenablage. Prinzipiell ist es auch möglich, dieses Target per `-cpu` oder `CPU` auszuwählen, aber der praktische Nutzen davon ist natürlich beschränkt.

Beim CPU-Befehl muß der Prozessortyp als einfaches Literal angegeben werden, eine Berechnung à la

```
CPU     68010+10
```

ist also nicht zulässig. Gültige Aufrufe sind z.B.

```
CPU     8051
```

oder

```
CPU     6800
```

Egal, welcher Prozessortyp gerade eingestellt ist, in der Integervariablen MOMCPU wird der momentane Status als Hexadezimalzahl abgelegt. Für den 68010 ist z.B. `MOMCPU=$68010`, für den 80C48 `MOMCPU=80C48H`. Da man Buchstaben außer A..F nicht als Hexziffer interpretieren kann, muß man sich diese bei der Hex-Darstellung des Prozessors wegdenken. Für den Z80 ist z.B. `MOMCPU=80H`.

Dieses Feature kann man vorteilhaft einsetzen, um je nach Prozessortyp unterschiedlichen Code zu erzeugen. Der 68000 z.B. kennt noch keinen Befehl für den Unterprogrammrücksprung mit Stapelkorrektur. Mit der Variablen `MOMCPU` kann man ein Makro definieren, das je nach Prozessortyp den richtigen Befehl benutzt oder ihn emuliert:

```
myrtd   MACRO   disp
        IF      MOMCPU$<$68010   ; auf 68008 und
         MOVE.L (sp),disp(sp)    ; 68000 emulieren
         LEA    disp(sp),sp
         RTS
        ELSEIF
         RTD    #disp            ; ab 68010 direkt
        ENDIF                    ; benutzen
        ENDM

        CPU     68010
        MYRTD   12               ; ergibt RTD #12

        CPU     68000
        MYRTD   12               ; ergibt MOVE.. /
                                 ; LEA.. / RTS
```

Da nicht alle Prozessornamen nur aus Ziffern und Buchstaben zwischen A und F bestehen, wird zusätzlich der volle Name in der String-Variablen `MOMCPUNAME` abgelegt.

Implizit schaltet der Assembler mit dem `CPU`-Befehl das aktuelle Segment wieder auf Code zurück, da dies das einzige Segment ist, das alle Prozessoren definieren.

Eine wichtige Änderung: Das Default-Target ist nicht mehr der 68008. Falls kein `-cpu` Kommandozeilen-Argument gegeben wurde, dann ist bis zur ersten `CPU`-Anweisung das reservierte Target `NONE` gesetzt. Vom Target unabhängig verfügbare Pseudo-Anweisungen sind in dieser Situation weiterhin erlaubt, zum Beispiel um Konstanten oder Makros zu definieren. Es ist aber nicht möglich, irgendwelchen Code zu definieren, weder über Maschinenbefehle, noch durch Ablegen von Daten im Speicher.

Für einige Ziele sind Optionen bzw. Varianten definiert, die so grundlegend sind, daß sie direkt zusammen mit dem `CPU`-Befehl gewählt werden müssen. Solche Optionen hängt man direkt an das Argument mit Doppelpunkten an, und sie haben die Form von Variablenzuweisungen:

```
CPU <CPU-Name>:<var1>=<wert1>:<var2>=<wert2>:...
```

Ob das jeweilige Ziel solche Optionen unterstützt, und wenn ja welche, wird im jeweils zugehörigen Unterkapitel mit prozessorspezifischen Hinweisen beschrieben.

3.2.4 SUPMODE, FPU, PMMU, CUSTOM

- *Gültigkeit SUPMODE: 680x0, NS32xxx, CR16C, PDP-11, i960, TLCS-900, SH, i960, 29K, Z280, XA, PowerPC, M*CORE, V60 und TMS9900*
- *Gültigkeit FPU: 680x0, NS32xxx, 80x86, WE32xxx, SH*
- *Gültigkeit PMMU: 680x0, NS32xxx*
- *Gültigkeit CUSTOM: NS32xxx*

Mit diesen Schaltern kann bestimmt werden, auf welche Teile des Befehlssatzes verzichtet werden soll, weil die dafür nötigen Vorbedingungen im folgenden Codestück nicht gegeben sind. Als Parameter für diese Befehle darf entweder `ON` oder `OFF` gegeben werden, der momentan gesetzte Zustand kann aus einer Variablen ausgelesen werden, die entweder TRUE oder FALSE ist. Die Befehle bedeuten im einzelnen folgendes:

- `SUPMODE`: erlaubt bzw. sperrt Befehle, für deren Ausführung der Prozessor im Supervisorstatus sein muß. Die Statusvariable heißt `INSUPMODE`.
- `FPU`: erlaubt bzw. sperrt die Befehle des numerischen Koprozessors 8087, NS32081/32381 bzw. 68881/68882. Die Statusvariable heißt `FPUAVAIL`. Für NS32xxx als Ziel darf neben einem einfachen `ON` oder `OFF` auch der FPU-Typ (`NS32081`, `NS32181`, `NS32381` oder `NS32580`) angegeben werden, um die erweiterten Register und Befehle freizugeben oder zu verbieten.
- `PMMU`: erlaubt bzw. sperrt die Befehle der Speicherverwaltungseinheit 68851 bzw. der im 68030 eingebauten MMU. **ACHTUNG!** Die 68030-MMU erlaubt nur eine relativ kleine Untermenge der 68851-Befehle, dies wird über den `FULLPMMU`-Befehl gesteuert. Die Statusvariable heißt `PMMUAVAIL`. Für NS32xxx als Ziel darf neben einem einfachen `ON` oder `OFF` auch der MMU-Typ (`NS32082`, `NS32382` oder `NS32532`) angegeben werden, um den jeweiligen MUU-Registersatz freizugeben.

- `CUSTOM`: erlaubt bzw. sperrt die für benutzerdefinierte Slave-Prozessoren vorgesehenen Befehle.

Benutzung von auf diese Weise gesperrten Befehlen erzeugt bei `SUPMODE` eine Warnung, bei `PMMU` und `FPU` eine echte Fehlermeldung.

3.2.5 ACCMODE bzw. EXECMODE

- *Gültigkeit: VAX (ACCMODE),*
  WE32xxx (EXECMODE)

VAX und WE32xxx kennen nicht nur einen Supervisor- un User-Mode, sondern vier derartige Privilegstufen. Mit absteigenden Rechten heissen diese Zugriffs-Modi Kernel, Executive, Supervisor und User. Mit dem `ACCMODE` bzw. `EXECMODE`-Befehl teilt man dem Assembler mit, in welchen Modus der folgende Code ausgeführt wird. Je nach Modus sind nicht alle Maschinenbefehle erlaubt. Als Argument sind entweder die genannten Namen der vier Modi zulässig, oder eine Zahl von Null (Kernel-Modus) bis Drei (User-Modus). Im Default wird der User-Modus angenommen, und die aktuelle Einstellung (als Zahlenwert) lässt sich aus dem Symbol gleichen Namens lesen.

3.2.6 CIS, EIS, FIS und FP11

- *Gültigkeit: PDP-11*

Mit diesen Anweisungen schaltet man die Verfügbarkeit bestimmter PDP-11-Befehlssatzerweiterungen ein oder aus. Voraussetzung für die Verfügbarkeit einer dieser Anweisungen ist, daß die fraglichen Befehle nicht bereits im Basis-Befehlssatz enthalten sind, und daß eine entsprechende Aufrüstmöglichkeit bestanden hat. Im einzelnen:

- `CIS`: „Commercial Instruction Set", d.h. Befehle zur Verarbeitung gepackter und nicht gepackter BCD-Zahlen mit variabler Länge. Diese waren auf dem LSI-11 sowie der PDP-11/44 als Option verfügbar.
- `EIS`: Die Befehle `MUL, DIV, ASH` und `ASHC`, die auf frühen bzw. einfachen PDP-11-Systemen nicht Teil des Basisbefehlssatzes waren. Auf dem LSI-11 bzw. der PDP-11/35 und PDP-11/40 waren sie per Option nachrüstbar.

- `FIS`: Stack-orientierte Befehle, die die Grundrechenarten für Gleitkommazahlen im F-Format (32 Bit) implementieren. Auf dem LSI-11 bzw. der PDP-11/35 und PDP-11/40 waren sie per Option nachrüstbar.
- `FP11`: Volle Gleitkomma-Unterstützung mit separaten FPU-Registern im F- und D-Format (32/64 Bit).

3.2.7 FULLPMMU

Gültigkeit: 680x0

Motorola hat zwar ab dem 68030 die PMMU in den Prozessor integriert, diese aber nur mit einer Funktionsuntermenge der externen PMMU 68851 ausgestattet. AS sperrt bei aktiviertem PMMU-Befehlssatz (s.o.) deshalb alle fehlenden Befehle, wenn als Zielprozessor 68030 oder höher eingestellt wurde. Nun kann es aber sein, daß in einem System mit 68030-Prozessor die interne MMU abgeschaltet wurde und der Prozessor mit einer externen 68851 betrieben wird. Mit `FULLPMMU ON` kann man AS dann mitteilen, daß der vollständige MMU-Befehlssatz zugelassen ist. Umgekehrt kann man, wenn man portablen Code erzeugen will, alle zusätzlichen Befehle trotz 68020-Zielplattform mit `FULLPMMU OFF` abschalten. Die Umschaltung darf beliebig oft erfolgen, die momentane Einstellung kann aus einem gleichnamigen Symbol ausgelesen werden. **ACHTUNG!** Der `CPU`-Befehl besetzt für 680x0-Argumente implizit diese Einstellung vor! `FULLPMMU` muß also auf jeden Fall nach dem `CPU`-Befehl kommen!

3.2.8 COPROC

Gültigkeit: TMS34020

Mit diesem Befehl wird festgelegt, welche Id (Adresse) in Koprozessor- Befehle eingesetzt werden soll, wenn keine explizite Id angegeben wurde. Der Default für diesen Wert wiederum ist Null. Erlaubte Werte sind von 0 bis 7.

3.2.9 PADDING

*Gültigkeit: 680x0, 68xx, M*Core, XA, H8, SH, TMS9900, MSP430(X), ST7/STM8, AVR (nur wenn die Granularität des Code-Segments 8 Bit ist)*

Diverse Prozessorfamilien verlangen, daß Objekte von mehr als einem Byte Länge auf einer geraden Adresse liegen müssen. Neben Datenobjekten schließt dies auch Instruktionsworte selber ein - auf einem 68000 lösen Wortzugriffe auf eine ungerade Adresse zum Beispiel eine Exception aus, andere Prozessoren wie die H8-Familie setzen das unterste Adreßbit bei einem Wortzugriff einfach hart auf Null.

Mit dem `PADDING`-Befehl kann man einen Mechanismus aktivieren, mit dem der Assembler versucht, solches 'Misalignment' nach Möglichkeit zu verhindern. Steht die Situation an, daß ein Instruktionswort, oder auch z.B. mit `DC` angelegte Daten von 16 Bit oder mehr auf einer ungeraden Adresse landen würden, dann wird automatisch ein Füllbyte davor eingefügt. Im Listing wird dieses Füllbyte in einer separaten Zeile mit dem Hinweis

```
<padding>
```

ausgewiesen.

Steht in der Quellzeile ein Label, so verweist dieses Label weiterhin auf den von dieser Zeile erzeugten Code, also auf die Adresse unmittelbar nach dem Füllbyte. Das gleiche gilt auch für ein Label in einer separaten Zeile unmittelbar davor, sofern diese Zeile *alleine* das Label und selber keine Anweisung enthält. Im folgenden Beispiel:

```
        padding  on
        org      $1000

        dc.b     1
adr1:   nop

        dc.b     1
adr2:
        nop

        dc.b     1
adr3:   equ      *
        nop
```

würden die Labels `adr1` und `adr2` die (durch ein Füllbyte auf einen geraden Wert aufgerundete) Adresse der jeweiligen `NOP`- Instruktion beinhalten, `adr3` würde jedoch auf das Füllbyte *vor* der dritten `NOP`-Instruktion zeigen.

Als Argument zu `PADDING` ist analog zu den vorherigen Befehlen `ON` oder `OFF` erlaubt, und die augenblickliche Einstellung kann aus dem gleichnamigen Symbol ausgelesen werden. Defaultmäßig ist `PADDING` nur für die 680x0-Familie eingeschaltet, für alle anderen werden erst nach Umschaltung Padding-Bytes eingefügt.

3.2.10 PACKING

Gültigkeit: 56000, AVR, TMS3203x/4x, TMS3206x, MN1610, CP1600, µPD7720/7725, µPD77230

`PACKING` ist in gewisser Weise ähnlich zu `PADDING`, es arbeitet nur gewissermaßen anders herum: während `PADDING` die abgelegten Daten ergänzt, um komplette Worte und damit ein Alignment zu erhalten, quetscht `PACKING` mehrere Werte in ein einzelnes Wort. Dies macht im Code-Segment des AVR Sinn, weil dort mit einem Spezialbefehl (`LPM`) auf einzelne Bytes in den 16-Bit-Worten zugegriffen werden kann. Ist diese Option eingeschaltet (Argument `ON`), so werden immer zwei Byte-Werte bei `DATA` in ein Wort gepackt, analog zu den einzelnen Zeichen von String-Argumenten. Der Wertebereich der Integer-Argumente reduziert sich dann natürlich auf -128...+255. Ist diese Option dagegen ausgeschaltet, (Argument `OFF`), so bekommt jedes Integer-Argument sein eigenes Wort und darf auch Werte von -32768...+65535 annehmen.

Diese Unterscheidung betrifft nur Integer-Argumente von `DATA`, Strings werden immer gepackt. Zu beachten ist weiterhin, daß dieses Packen nur innerhalb der Argumente eines `DATA`-Befehls funktionieren kann, wer also mehrere `DATA`-Befehle hintereinander hat, fängt sich bei ungeraden Argumentzahlen trotzdem halbvolle Wörter ein!

3.2.11 MAXMODE

Gültigkeit: TLCS-900, H8

Die Prozessoren der TLCS-900-Reihe können in 2 Betriebsarten arbeiten, dem Minimum-und Maximum-Modus. Je nach momentaner Betriebsart gelten für den Betrieb und den Assembler etwas andere Eckwerte. Mit diesem Befehl und den Parametern `ON` oder `OFF` teilt man AS mit, daß der folgende Code im Maximum- oder Minimum-Modus abläuft. Die momentane Einstellung kann aus der Variablen `INMAXMODE` ausgelesen werden. Voreinstellung ist `OFF`, d.h. Minimum-Modus.

Analog dazu teilt man im H8-Modus AS mit diesem Befehl mit, ob mit einem 64K- oder 16Mbyte-Adreßraum gearbeitet wird. Für den einfachen 300er ist diese Einstellung immer `OFF` und kann nicht verändert werden.

3.2.12 EXTMODE und LWORDMODE

Gültigkeit: Z380

Der Z380 kann in insgesamt 4 Betriebsarten arbeiten, die sich durch die Einstellung von 2 Flags ergeben: Das XM-Flag bestimmt, ob der Prozessor mit einem 64 Kbyte oder 4 Gbyte großen Adreßraum arbeiten soll und kann nur gesetzt werden (nach einem Reset steht es Z80-kompatibel auf 0). Demgegenüber legt das LW-Flag fest, ob Wort-Befehle mit einer Wortlänge von 16 oder 32 Bit arbeiten sollen. Die Stellung dieser beiden Flags beeinflußt Wertebereichseinschränkungen von Konstanten oder Adressen, weshalb man AS über diese beiden Befehle deren Stellung mitteilen muß. Als Default nimmt AS an, daß beide Flags auf 0 stehen, die momentane Einstellung (`ON` oder `OFF`) kann aus den vordefinierten Variablen `INEXTMODE` bzw. `INLWORDMODE` ausgelesen werden.

3.2.13 SRCMODE

Gültigkeit: MCS-251

Intel hat den Befehlssatz der 8051er beim 80C251 deutlich erweitert, hatte aber leider nur noch einen einzigen freien Opcode für diese Befehle frei. Damit der Prozessor nicht auf alle Ewigkeit durch einen Präfix behindert bleibt, hat Intel zwei Betriebsarten vorgesehen: Den Binär- und den Quellmodus. Im Binärmodus ist der Prozessor voll 8051-kompatibel, alle erweiterten Befehle benötigen den noch freien Opcode als Präfix. Im Quellmodus tauschen diese neuen Befehle ihre Position in der Code-Tabelle mit den entsprechenden 8051-Instruktionen, welche dann wiederum mit einem Präfix versehen werden müssen. Damit AS weiß, wann er Präfixe setzen muß und wann nicht, muß man ihm mit diesem Befehl mitteilen, ob der Prozessor im Quellmodus (`ON`) oder Binärmodus (`OFF`) betrieben wird. Die momentane Einstellung kann man aus der Variablen `INSRCMODE` auslesen. Der Default ist `OFF`.

3.2.14 EXTRACOMMENTS

Gültigkeit: 68xx

Motorolas Original-Assembler haben das Feature, dass wenn Label oder Mnemonic mit einem Stern beginnen, der Rest der Zeile als Kommentar betrachtet wird. AS unterstützt dies normalerweise nicht. Man kann dies aber mit einem

```
extracomments on
```

einschalten, um die Portierung von Code zu vereinfachen, der für andere Assembler geschrieben wurde.

Des weiteren werden sogenannte 'End-of-Line' Kommentare zugelassen: Die Argumentliste endet bei ersten (nicht-gequoteten) Leerzeichen, alles danach wird als Kommentar interpretiert.

Es ist möglich, diesen Modus über eine gleichnamige Kommandozeilen-Option von Beginn einzuschalten. Der aktuelle Wert kann aus einer Variable gleichen Namens ausgelesen werden.

3.2.15 PLAINBASE

Gültigkeit: 6809

Historisch erlaubt AS, bei indizierten Adressierungsausdrücken ein leeres, erstes Argument wegzulassen. Ein

```
lda  x
```

zum Beispiel war also äquivalent zu

```
lda  ,x
```

Obwohl als Feature gedacht, wurde dies jedoch gelegentlich eher als Bug angesehen. Deshalb darf ein leeres Index-Argument nicht mehr weggelassen werden bzw. wird mit einer Fehlermeldung über eine falsche Anzahl Argumente quittiert. Falls die Funktion gewünscht ist bzw. für existierenden Code gebraucht wird, kann sie mit einem

```
plainbase on
```

wieder eingeschaltet werden. Die aktuelle Einstellung kann aus dem gleichnamigen Symbol ausgelesen werden.

3.2.16 BIGENDIAN

Gültigkeit: MCS-51/251, PowerPC, SC/MP, 2650, NS32000

Bei den Prozessoren der 8051-Serie ist Intel seinen eigenen Prinzipien untreu geworden: Der Prozessor verwendet entgegen jeglicher Tradition eine Big-Endian-Orientierung von Mehrbytewerten! Während dies bei den MCS-51-Prozessoren noch

nicht großartig auffiel, da der Prozessor ohnehin nur 8-bittig auf Speicherzellen zugreifen konnte, man sich die Byte-Anordnung bei eigenen Datenstrukturen also aussuchen konnte, ist dies beim MCS-251 nicht mehr so, er kann auch ganze (Lang-)Worte aus dem Speicher lesen und erwartet dabei das MSB zuerst. Da dies nicht der bisherigen Arbeitsweise von AS bei der Konstantenablage entspricht, kann man nun mit diesem Befehl umschalten, ob die Befehle `DB, DW, DD, DQ, DT` und `DO` mit Big- oder Little-Endian-Orientierung arbeiten sollen. Mit `BIGENDIAN OFF` (Voreinstellung) wird wie bei älteren AS-Versionen zuerst das niederwertigste Byte abgelegt, mit `BIGENDIAN ON` wird die MCS-251-kompatible Variante benutzt. Natürlich kann man diese Einstellung beliebig oft im Code ändern; die momentane Einstellung kann aus dem gleichnamigen Symbol ausgelesen werden.

Für Renesas RX ist die Endianess ebenfalls umschaltbar, aus Kompatibilität zum Original-Assembler heißt der Befehl hier jedoch `ENDIAN` und akzeptiert ein `LITTLE` oder `BIG` als Argument.

3.2.17 WRAPMODE

Gültigkeit: Atmel AVR

Ist dieser Schalter auf `ON` gesetzt, so veranlaßt man AS dazu, anzunehmen, der Programmzähler des Prozessors habe nicht die volle, durch die Architektur gegebene Länge von 16 Bits, sondern nur eine Länge, die es gerade eben erlaubt, das interne ROM zu adressieren. Im Falle des AT90S8515 sind dies z.B. 12 Bit, entsprechend 4 KWorten oder 8 KBytes. Damit werden relative Sprünge vom Anfang des ROMs zum Ende und umgekehrt möglich, die bei strenger Arithmetik einen out-of-branch ergeben würden, hier jedoch funktionieren, weil die Übertragsbits bei der Zieladressenberechnung 'unter den Tisch' fallen. Vergewissern Sie sich genau, ob die von Ihnen eingesetzte Prozessorvariante so arbeitet, bevor Sie diese Option einschalten! Im Falle des oben erwähnten AT90S8515 ist diese Option sogar zwingend nötig, um überhaupt quer durch den ganzen Adreßraum springen zu können...

Defaultmäßig steht dieser Schalter auf `OFF`, der momentane Stand läßt sich aus einem gleichnamigen Symbol auslesen.

3.2.18 PANEL

Gültigkeit: IM61x0

Mit diesem Schalter teilt man dem Assembler mit, ob der folgende Code mit gesetztem oder gelöschtem *Control Panel Flip-Flop* ausgeführt wird. Eine Reihe von

`IOT`-Instruktionen sind nur bei einer bestimmten Stellung des Flip-Flops zulässig. Ihre Verwendung im anderen Modus wird vom Assembler mit einer Fehlermeldung quittiert.

Die aktuelle Einstellung kann aus dem Symbol `INPANEL` gelesen werden.

3.2.19 WARNRELATIVE

Gültigkeit: Zx80

Mit diesem Schalter teilt man dem Assembler mit, ob eine Warnung erfolgen soll, falls ein relativer anstelle eines absoluten Sprungs möglich gewesen wäre. Der Default ist OFF bzw. das, was auf der Kommandozeile durch die Argumente `-wrelative` bzw. `-wno-relative` vorgegeben wurde.

Die aktuelle Einstellung kann aus dem gleichnamigen Symbol gelesen werden.

3.2.20 SEGMENT

Gültigkeit: alle Prozessoren

Bestimmte Mikrokontroller und Signalprozessoren kennen mehrere Adreßbereiche, die nicht miteinander mischbar sind und jeweils auch verschiedene Befehle zur Ansprache benötigen. Um auch diese verwalten zu können, stellt der Assembler mehrere Programmzähler zur Verfügung, zwischen denen mit dem `SEGMENT`-Befehl hin-und hergeschaltet werden kann. Dies erlaubt es, sowohl in mit `INCLUDE` eingebundenen Unterprogrammen als auch im Hauptprogramm benötigte Daten an der Stelle zu definieren, an denen sie benutzt werden. Im einzelnen werden folgende Segmente mit folgenden Namen verwaltet:

- `CODE`: Programcode;
- `DATA`: direkt adressierbare Daten (dazu rechnen auch SFRs);
- `XDATA`: im extern angeschlossenen RAM liegende Daten oder X-Adreßraum beim DSP56xxx oder ROM-Daten beim μPD772x;
- `YDATA`: Y-Adreßraum beim DSP56xxx;
- `IDATA`: indirekt adressierbare (interne) Daten;
- `BITDATA`: der Teil des 8051-internen RAMs, der bitweise adressierbar ist;

- `IO`: I/O-Adreßbereich;
- `REG`: Registerbank des ST9;
- `ROMDATA`: Konstanten-ROM der NEC-Signalprozessoren;
- `EEDATA`: eingebautes EEPROM.

Zu Adreßbereich und Initialwerten der Segmente siehe Abschnitt 3.2.1. (`ORG`). Je nach Prozessorfamilie sind auch nicht alle Segmenttypen erlaubt.

Das Bitsegment wird so verwaltet, als ob es ein Bytesegment wäre, d.h. die Adressen inkrementieren um 1 pro Bit.

Labels, die in einem Segment eines bestimmten Typs definiert werden, erhalten diesen Typ als Attribut. Damit hat der Assembler eine begrenzte Prüfmöglichkeit, ob mit den falschen Befehlen auf Symbole in einem Segment zugegriffen wird. In solchen Fällen wird der Assembler eine Warnung ausgeben.

Beispiel:

```
        CPU     8051    ; MCS-51-Code

        SEGMENT code    ; Testcodeblock

        SETB    flag    ; keine Warnung
        SETB    var     ; Warnung : falsches Segment

        SEGMENT data

var     DB      ?

        SEGMENT bitdata

flag    DB      ?
```

3.2.21 PHASE und DEPHASE

Gültigkeit: alle Prozessoren

In manchen Anwendungen (speziell Z80-Systeme) muß Code vor der Benutzung in einen anderen Adreßbereich verschoben werden. Da der Assembler davon aber

nichts weiß, würde er alle Labels in dem zu verschiebenden Teil auf die Ladeadressen ausrichten. Der Programmierer müßte Sprünge innerhalb dieses Bereiches entweder lageunabhängig kodieren oder die Verschiebung bei jedem Symbol ,,zu Fuß" addieren. Ersteres ist bei manchen Prozessoren gar nicht möglich, letzteres sehr fehleranfällig.

Mit dem Befehlen `PHASE` und `DEPHASE` ist es möglich, dem Assembler mitzuteilen, auf welcher Adresse der Code im Zielsystem effektiv ablaufen wird:

```
PHASE   <Adresse>
```

informiert den Assembler davon, daß der folgende Code auf der spezifizierten Adresse ablaufen soll. Der Assembler berechnet daraufhin die Differenz zum echten Programmzähler und addiert diese Differenz bei folgenden Operationen dazu:

- Adreßangabe im Listing
- Ablage von Labelwerten
- Programmzählerreferenzen in relativen Sprüngen und Adreßausdrücken
- Abfrage des Programmzählers mit den Symbolen `*` bzw. `$`

Diese ,,Verschiebung" wird mit dem Befehl

```
DEPHASE
```

wieder auf den vor der zugehörigen `PHASE`-Anweisung zurück geändert. `PHASE` und `DEPHASE` können also auf diese Weise geschachtelt verwendet werden.

Obwohl dieses Befehlspaar vornehmlich in Codesegmenten Sinn macht, verwaltet der Assembler für alle definierten Segmente Phasenwerte.

3.2.22 SAVE und RESTORE

Gültigkeit: alle Prozessoren

Mit dem Befehl `SAVE` legt der Assembler den Inhalt folgender Variablen auf einen internen Stapel:

- momentan gewählter Prozessortyp (mit `CPU` gesetzt);
- momentan aktiver Speicherbereich (mit `SEGMENT` gesetzt);

- Flag, ob Listing ein- oder ausgeschaltet ist (mit `LISTING` gesetzt);
- Flags, zu welchem Teil Expansionen folgender Makos im Listing ausgegeben werden sollen (mit `MACEXP_DFT/MACEXP_OVR` gesetzt).
- momentan aktive Zeichenübersetzungstabelle (mit `CODEPAGE` gesetzt).

Mit dem Gegenstück `RESTORE` wird entsprechend der zuletzt gesicherte Zustand von diesem Stapel wieder heruntergeladen. Diese beiden Befehle sind in erster Linie für Include-Dateien definiert worden, um in diesen Dateien die obigen Variablen beliebig verändern zu können, ohne ihren originalen Inhalt zu verlieren. So kann es z.B. sinnvoll sein, in Include-Dateien mit eigenen, ausgetesteten Unterprogrammen die Listingerzeugung auszuschalten:

```
SAVE            ; alten Zustand retten
LISTING OFF     ; Papier sparen
..              ; der eigentliche Code
RESTORE         ; wiederherstellen
```

Gegenüber einem einfachen `LISTING OFF..ON`-Pärchen wird hier auch dann der korrekte Zustand wieder hergestellt, wenn die Listingerzeugung bereits vorher ausgeschaltet war.

Der Assembler überprüft, ob die Zahl von `SAVE`-und `RESTORE`-Befehlen übereinstimmt und liefert in folgenden Fällen Fehlermeldungen:

- `RESTORE` und der interne Stapel ist leer;
- nach Ende eines Passes ist der Stapel nicht leer.

Falls das aktuell benutzte Target bereits einen Maschinenbefehl namens `SAVE` bzw. `RESTORE` hat, kann die Funktionalität über `SAVEENV` bzw. `RESTOREENV` erreicht werden. Wahlweise ist es auch immer möglich, explizit die Pseudobefehle durch voran gestellte Punkte (`.SAVE` bzw. `.RESTORE`) aufzurufen.

3.2.23 ASSUME

Gültigkeit: diverse

Mit diesem Befehl kann man AS den aktuellen Stand bestimmter Register mitteilen, deren Inhalt sich nicht mit einem einfachen `ON` oder `OFF` beschreiben läßt. Typischerweise sind dies Register, die die Adressierungseinheiten beeinflussen und

deren Werte AS wissen muß, um korrekte Adressierungen zu erzeugen. Wichtig ist, daß man AS mit ASSUME diese Werte nur mitteilt, es wird *kein* Maschinencode erzeugt, der diese Werte in die entsprechenden Register lädt!

Ein mit `ASSUME` definierter Wert läßt sich mit der eingebauten Funktion `ASSUMEDVAL` wieder abfragen oder in Ausdrücke einbauen. Dies gilt für alle im folgenden gelisteten Architekturen mit Ausnahme des 8086.

65CE02

Der 65CE02 besitzt ein Register 'B', mit dem die 'Base-Page' festgelegt wird. Im Gegensatz zum 'einfachen' 6502 läßt sich damit die Speicherseite, die mit kurzen (8-bittigen) Adressen ansprechbar ist, frei im 64K-Adre'sraum hin- und herschieben. Nach einem Reset steht dieses Register auf Null, der 65CE02 verhält sich also wie sein Vorbild. Dies ist auch die Default-Annahme des Assemblers. Mittels eines `ASSUME B:xx` kann man ihm den aktuellen Wert mitteilen, und für Adressen in dieser Seite werden dann automatisch kurze Adressierungsarten benutzt.

6809

Im Gegensatz zu seinen ,,Vorgängern" wie 6800 und 6502 kann beim 6809 die Lage der direct page, d.h. des Adressbereiches, der mit ein Byte langen Adressen erreichbar ist, frei bestimmt werden. Dazu dient das sog. ,,Direct Page Register" (`DPR`), das die Seitennummer festlegt. Ihm muß man mittels `ASSUME` einen passenden Wert zuweisen, wenn man einen anderen Wert als die Vorgabe von 0 in DPR schreibt, sonst werden Adressen falscher Länge erzeugt...

68HC11K4

Auch beim HC11 kamen die Entwickler letzen Endes nicht um eine Form von Banking herum, um mit 16 Adreßleitungen mehr als 64 Kbyte anzusprechen. Die Register `MMSIZ`, `MMWBR`, `MM1CR` und `MM2CR` legen fest, ob und wie die beiden zusätzlichen 512K-Bereiche in den Adreßraum der CPU eingeblendet werden sollen. Initial nimmt AS den Reset-Zustand dieser Register an, d.h. alle mit $00 belegt und das Windowing ist abgeschaltet.

Weiterhin kann man die Werte der Register `CONFIG`, `INIT` und `INIT2` festlegen. Aus diesen kann der Assembler die Lage der CPU-internen Register sowie des CPU-internen RAM/EEPROM bestimmen. Deren Mapping hat eine höhere Priorität als die per Windowing eingeblendeten Speicherbereiche.

68HC12X

Wie die Variante ohne anhängendes 'X' kennt auch der HC12X eine kurze direkte Adressierungsart, die hier jedoch auch andere Adreßbereiche als die ersten 256 Byte erreichen kann. Über das `DIRECT`-Register kann die 256-Byte-Seite vorgegeben werden, die mit dieser kurzen Adressierungsart angesprochen wird. Mittels `ASSUME` wird AS der momentane Stand dieses Registers mitgeteilt, so daß bei absoluten Adressen automatisch die effizienteste Adressierungsart gewählt werden kann. Default ist 0, was auch dem Reset-Zustand entspricht.

68HC16

Um mit seinen nur 16 Bit breiten Adreßoperanden einen 1 Mbyte großen Adreßraum ansprechen zu können, bedient sich der 68HC16 einer Reihe von Bank-Registern, die die fehlenden oberen vier Adreßbits nachliefern. Davon ist das `EK`-Register für absolute Datenzugriffe (nicht Sprünge!) zuständig. AS überprüft bei jeder absoluten Adressierung, ob die oberen vier Bits der Adresse mit dem über `ASSUME` spezifizierten Wert übereinstimmen. Differieren die Werte, gibt AS eine Warnung aus. Der Vorgabewert für `EK` ist 0.

H8/500

Im Maximum-Modus wird der erweiterte Adreßraum dieser Prozessorreihe durch eine Reihe von Bank-Registern adressiert. Diese tragen die Namen DP (Register 0..3, absolute Adressen), EP (Register 4/5) und TP (Stack). Den momentanen Wert von DP benötigt AS, um zu überprüfen, ob absolute Adressen in der momentan adressierbaren Bank liegen; die beiden anderen Register werden nur für indirekte Adressierungen benutzt und entziehen sich daher der Kontrolle; ob man ihre Werte angibt oder nicht, ist daher Geschmackssache. Wichtig ist dagegen wieder das BR-Register, das angibt, auf welchen 256-Byte-Bereich mit kurzen Adressen zugegriffen werden kann. Allen Registern ist gemeinsam, daß AS *keine* Initialwerte für sie annimmt, da sie nach einem Prozessor-Reset undefiniert sind; wer absolut adressieren will, muß daher auf jeden Fall DR und DP belegen!

MELPS740

Die Mikrokontroller dieser Reihe kennen für den `JSR`-Befehl eine besondere Adressierungsart „special page", mit deren Hilfe man Sprünge in die oberste Seite des internen ROMs kürzer kodieren kann. Diese ist natürlich vom jeweiligen Chip abhängig,

und es gibt mehr Chips, als es mit dem `CPU`-Befehl sinnvoll wäre, zu kodieren...also muß `ASSUME` herhalten, um die Lage dieser Seite vorzugeben, z.B.

```
ASSUME SP:$1f ,
```

falls das interne ROM 8K groß ist.

MELPS7700/65816

Diese Prozessoren beinhalten eine Reihe von Registern, deren Inhalt AS kennen muß, um den korrekten Code zu erzeugen. Es handelt sich um folgende Register:

Name	Bedeutung	Wertebereich	Default
DT/DBR	Datenbank	0-$ff	0
PG/PBR	Code-Bank	0-$ff	0
DPR	direkt adr. Seite	0-$ffff	0
X	Indexregisterbreite	0 oder 1	0
M	Akkumulatorbreite	0 oder 1	0

Um mich nicht in endlose Wiederholungen zu ergehen, verweise ich für die Benutzung dieser Werte auf Kapitel 4.15. Die Handhabung erfolgt ansonsten genauso wie beim 8086, d.h. es können auch hier mehrere Werte auf einmal gesetzt werden und es wird **kein** Code erzeugt, der die Register mit den Werten besetzt. Dies bleibt wieder einzig und allein dem Programmierer überlassen!

MCS-196/296

Alle Prozessoren der MCS-96-Familie besitzen ab dem 80196 ein Register `WSR`, mit dessen Hilfe Speicherbereiche aus dem erweiterten internen RAM oder dem SFR-Bereich in Bereiche des Registerfiles eingeblendet werden und so mit kurzen Adressen angesprochen werden können. Teilt man AS mit Hilfe des `ASSUME`-Befehls mit, welchen Wert das WSR-Register hat, so stellt er bei absoluten Adressen automatisch fest, ob sie durch das Windowing mit 1-Byte-Adressen erreicht werden können; umgekehrt werden auch für durch das Windowing überdeckte Register automatisch lange Adressen erzeugt. Der 80296 besitzt ein zusätzliches, zweites Register `WSR1`, um zwei unterschiedliche Speicherbereiche gleichzeitig in das Registerfile einblenden zu können. Sollte es möglich sein, eine Speicherzelle über beide Bereiche zu adressieren, so wählt AS immer den Weg über `WSR`!

Bei indirekter Adressierung können Displacements wahlweise kurz (8 Bit, -128 bis +127) oder lang (16 Bit) sein. Der Assembler wählt automatisch anhand des Displacements die kürzestmögliche Kodierung. Es ist aber möglich, durch ein vorangestelltes Größer-Zeichen (>) eine 16-Bit-Kodierung des Displacements zu erzwingen. Gleiches gilt für absolute Adressen im Bereich 0ff80h...0ffffh, die mit einem kurzen Offset relativ zum Nullregisterädressiert werden können.

8086

Der 8086 kann Daten aus allen Segmenten in einem Befehl adressieren, benötigt jedoch sog. „Segment-Präfixe", wenn ein anderes Segmentregister als DS verwendet werden soll. Zusätzlich kann es sein, daß das DS-Register auf ein anderes Segment verstellt ist, um z.B. über längere Strecken nur Daten im Codesegment zu adressieren. Da AS aber keine Sinnanalyse des Codes vornimmt, muß ihm über diesen Befehl mitgeteilt werden, auf welche Segmente die Segmentregister momentan zeigen, z.B.

```
ASSUME  CS:CODE, DS:DATA    .
```

Allen vier Segmenten des 8086 (SS,DS,CS,ES) können auf diese Weise Annahmen zugewiesen werden. Dieser Befehl erzeugt jedoch **keinen** Code, um die Werte auch wirklich in die Segmentregister zu laden, dies muß vom Programm getan werden.

Die Benutzung diese Befehls hat zum einen die Folge, daß AS bei sporadischen Zugriffen ins Codesegment automatisch Präfixe voranstellen kann, andererseits daß man AS mitteilen kann, daß das DS-Register verstellt wurde und man sich im folgenden explizite `CS:`-Anweisungen sparen kann.

Gültige Argumente hinter dem Doppelpunkt sind `CODE, DATA` und `NOTHING`. Letzterer Wert dient dazu, AS mitzuteilen, daß das Segmentregister keinen für AS verwendbaren Wert enthält. Vorinitialisiert sind folgende `ASSUME`s :

```
CS:CODE, DS:DATA, ES:NOTHING, SS:NOTHING
```

Z180

Der Z180 besitzt eine eingebaute MMU, die den „logischen Adreßraum" des CPU-Kerns von 64 KByte auf einen physischen Adreßraum von 512 KByte übersetzt. Das genaue Mapping wird durch die drei Register `CBAR`, `CBR` und `BBR` gesteuert. Ähnlich wie beim 68HC11K4 nimmt AS automatische Umrechnungen von physischen auf logische Adressen vor, und zwar sowohl für absolute Adressen als auch Quelle und Ziel von relativen Sprüngen. Es ist ebenfalls möglich, auf die Mapping-Tabellen über die Funktionen `phys2cpu()` und `cpu2phys()` zuzugreifen.

Z280

Beim Z280 kann man über das I-Bit im Trap Control Register festlegen, ob I/O-Zugriffe im User-Modus erlaubt sein sollen oder nicht. Teilt man AS über ein

```
assume i:1
```

mit, dass sie verboten sind, dann wird eine entsprechende Warnung ausgegeben, falls I/O-Befehle im User-Modus verwendet werden.

Des weiteren enthält der Z280 eine MMU, um den logischen 64K-Adreßraum der CPU auf die physikalisch adressierbaren 16 MByte abzubilden. Ihre Funktion wird zum einen über das MMU Master Control Register (`MMUMCR`) gesteuert, zum anderen über 32 Page Descriptor Register ((`UPD0...UPD15`, `SPD0...SPD15`). Die aktuellen Werte aller Register können AS via `ASSUME`-Anweisungen mitgeteilt werden.

Für die Berechnung des aktuellen Mappings werden die folgenden Bits des `MMUMCR` beachtet:

- Bit 15 (UTE): MMU im User-Modus einschalten
- Bit 14 (UPD): Programm/Data-Separation im User-Modus einschalten
- Bit 11 (STE): MMU im System-Modus einschalten
- Bit 10 (SPD): Programm/Data-Separation im System-Modus einschalten

Die folgenden Bits von `UPDn/SPDn` werden beachtet:

- Bit 3 (V): Seite gültig/aktiv
- Bit 15...4 bzw. 15...5: Seitenadresse, also Bits 23...12 bzw. 23...13 der Startadresse im physischen Adressraum.

Abhängig von den Registerwerten kann die Adressübersetzung im System- und/oder User-Modus aktiv sein (STE und/oder UTE gesetzt), oder überhaupt nicht (sowohl STE als auch UTE nicht gesetzt). Die aktuelle Einstellung von `SUPMODE` hat deshalb auch einen Einfluß auf das aktuelle Mapping. Falls es eingeschaltet ist, kann es für Code- und Datenzugriffe gleich oder unterschiedlich sein (UPD/SPD nicht gesetzt oder gesetzt).

eZ80

Der eZ80 kann in zwei Modi operieren:

- Im Z80 Modus sind die Register BC, DE, HL, IX, IY und SP 16 bit breit, und nur die 64K-Speicherseite, die durch das MBASE-Register definiert ist, kann adressiert werden. Bei Befehlen, die eine absolute Adresse oder einen nicht-8-Bit Wert als Argument beinhalten, werden zwei Bytes geladen.

- Im ADL Modus sind die Register BC, DE, HL, IX, IY und SP 24 Bit breit, und der komplette 16 MByte Adressraum kann angesprochen werden. Bei Befehlen, die eine absolute Adresse oder einen nicht-8-Bit Wert als Argument beinhalten, werden drei Bytes geladen.

Da die Binärkodierung und Adreßbereichsprüfung von diesem Modus abhängt, muß der Assembler wissen, welcher Modus gerade verwendet wird. Indem man `ADL` mittels `ASSUME` entweder auf 0 oder 1 setzt, teilt man dem Assembler den Default-Modus mit, d.h. den Modus ohne Verwendung expliziter Suffixe an einem Befehl. Des weiteren kann dem Assembler der aktuelle Wert von MBASE (0 bis 0ff hex) mitgeteilt werden. Die Default-Annahme für beide Werte ist 0, ähnlich den Werten nach einem Hardware-Reset.

XA

Die XA-Familie besitzt einen Datenadreßraum von 16 Mbyte, ein Prozeß kann jedoch nur immer innerhalb einer 64K-Seite adressieren, die durch das DS-Register vorgegeben wird. AS muß man den momentanen Wert dieses Registers vorgeben, damit er Zugriffe auf absolute Adressen überprüfen kann.

29K

Die Prozessoren der 29K-Familie besitzen ein Register `RBP`, mit dessen Hilfe Bänke von 16 Registern vor der Benutzung im User-Modus geschützt werden können. Dazu kann man ein entsprechendes Bit in diesem Register setzen. Mit `ASSUME` kann man AS nun mitteilen, welchen Wert RBP gerade hat. Auf diese Weise kann AS warnen, falls versucht wird, im User-Modus auf geschützte Register zuzugreifen.

80C166/167

Obwohl keines der Register im 80C166/167 breiter als 16 Bit ist, besitzt dieser Prozessor 18/24 Adreßleitungen, kann also bis zu 256 Kbyte/16 Mbyte adressieren. Um diesen Widerspruch unter einen Hut zu bekommen, verwendet er nicht die von Intel her bekannte (...und berüchtigte) Segmentierung oder hat unflexible Bankregister...nein, er macht Paging! Dazu wird der ,,logische" Adreßraum von 64 Kbyte in 4 Seiten zu 16 Kbyte eingeteilt, und für jede Seite existiert ein Seitenregister (bezeichnet als `DPP0...DPP3`), das bestimmt, welche der physikalischen 16/1024 Seiten dort eingeblendet wird. AS versucht nun, den Adreßraum grundsätzlich mit 256 Kbyte/16 Mbyte aus der Sicht des Programmierers zu verwalten, d.h. bei absoluten Zugriffen ermittelt AS die physikalische Seite und schaut in der mit `ASSUME` eingestellten Seitenverteilung nach, wie die Bits 14 und 15 der logischen Adresse gesetzt werden müssen. Paßt kein Seitenregister, so wird eine Warnung ausgegeben. Defaultmäßig nimmt AS an, daß die vier Register linear die ersten 64 Kbyte abbilden, etwa in der folgenden Form:

```
ASSUME  DPP0:0,DPP1:1,DPP2:2,DPP3:3
```

Der 80C167 kennt noch einige Befehle, die die Seitenregister in ihrer Funktion übersteuern können. Wie diese Befehle die Adreßgenerierung beeinflussen, ist im Kapitel mit den prozessorspezifischen Hinweisen beschrieben.

Einige Maschineninstruktionen kennen ein verkürzte Kodierung, wenn das Argument in einem bestimmten Wertebereich liegt:

- `MOV Rn,#<0..15>`
- `ADD/ADDC/SUB/SUBC/CMP/XOR/AND/OR Rn, #<0..7>`
- `LOOP Rn,#<0..15>`

Der Assembler verwendet im Default automatisch die kürzere Kodierung wenn möglich. Falls man die längere erzwingen möchte, schreibt man analog zum 65xx/68xx ein Größerzeichen vor den Operanden (hinter das Doppelkreuz!). Umgekehrt kann man auch ein Kleinerzeichen schreiben, wenn man die kurze Kodierung erzwingen will. Falls der Operand nicht im erlaubten Wertebereich liegt, gibt es eine Fehlermeldung. Singemäß das gleiche gilt für Sprünge, die mit kurzem Displacement oder langem, absolutem Argument kodiert werden können.

TLCS-47

Der von der Architektur her vorgegebene Datenadreßraum dieser Prozessoren (egal ob man direkt oder über das HL-Register adressiert) beträgt lediglich 256 Nibbles. Da die ,,besseren" Familienmitglieder aber bis zu 1024 Nibbles RAM on chip haben, war Toshiba gezwungen, einen Bankingmechanismus über das DMB-Register einzuführen. AS verwaltet das Datensegment als einen durchgehenden Adreßraum und prüft bei jeder direkten Adressierung, ob die Adresse in der momentan aktiven Bank liegt. Die von AS momentan angenommene Bank kann mittels

```
ASSUME  DMB:<0..3>
```

festgelegt werden. Der Default ist 0.

IPC-16/INS8900

Der Prozessor verfügt über einen Eingangs-Pin namens `BPS`, mit dem man wählen kann, welcher Adreßbereich direkt adressierbar sein soll: entweder die untersten 256 Worte des Speichers, oder die jeweils obersten und untersten 128 Worte. Der Default ist ersteres, mit einem `ASSUME BPS:1` schaltet man auf die zweite Variante um.

CR16C

Der CR16C kann in einer alternativen Betriebsart laufen, in der die Register `R12...SP` auf 16 Bit beschränkt sind, um eine bessere architektonische Aufwärtskompatibilität zu seinen Vorgängern zu erreichen. Die Betriebsart wird durch das `SR`-Bit im `CFG`-Register aktiviert und hat die folgenden weiteren Effekte auf das Programmiermodell:

- Weitere 16-Bit-Registerpaare: `(R13_L,R12_L)`, `(RA_L,R13_L)`, `(SP_L,RA_L)`, `(SP_H,SP_L)`.
- Indexregister-Adressierung (`[Rn]abs20`, `[Rn]disp(rp)`) ist nicht erlaubt.
- Einige Befehle erlauben kürzere Kodierung für den Adressierungsmodus `disp(reg)`, falls das Displacement Null oder kleiner als 0x4000 ist.

Der Wert dieses Bits zur Laufzeit kann über die Anweisung

```
assume sr:n
```

mitgeteilt werden, wobei `n` entweder Null oder Eins ist. Null ist der Default, d.h. alle Register haben ihre volle Breite und Indexadressierung ist erlaubt.

ST6

Die Mikrokontroller der ST6-Reihe sind in der Lage, einen Teil (64 Byte) des Codebereiches in den Datenbereich einzublenden, z.B. um Konstanten aus dem ROM zu laden. Dies bedeutet aber auch, daß zu einem Zeitpunkt immer nur ein Teil des ROMs adressiert werden kann. Welcher Teil dies ist, wird durch ein bestimmtes Register bestimmt. Dem Inhalt dieses Registers kann AS zwar nicht direkt kontrollieren, man kann ihm aber mit diesem Befehl mitteilen, wenn man dem Register einen neuen Wert zugewiesen hat. AS kann dann prüfen und ggfs. warnen, falls auf Adressen im Codesegment zugegriffen wird, die nicht im ,,angekündigten" Fenster liegt. Hat die Variable `VARI` z.B. den Wert 456h, so setzt

```
ASSUME  ROMBASE:VARI>>6
```

die AS-interne Variable auf 11h, und ein Zugriff auf `VARI` erzeugt einen Zugriff auf die Adresse 56h im Datensegment.

Anstelle eines Symbols kann auch schlicht `NOTHING` angegeben werden, z.B. wenn das Bank-Register temporär als Speicherzelle benutzt wird. Dieser Wert ist auch die Voreinstellung.

Der Programmzähler dieser Mikrokontroller ist lediglich 12 Bit breit. Das bedeutet für Varianten mit mehr als 4 KByte Programmspeicher, daß man sich eine Art von Banking einfallen lassen mußte. Dazu werden Adreßraum und Programmspeicher in 2 KByte-Seite eingeteilt. Seite 1 des Adreßraumes greift immer auf Seite 1 des Programmspeichers zu. Über das bei diesen Varianten vorhandene `PRPR`-Register kann der Programmierer bestimmen, welche Seite des Programmspeichers über die Adressen 000h bis 7ffh zugegriffen wird. AS betrachtet den Adreßraum in erster Näherung als linear und von der Größe des Programmspeichers. Erfolgt ein Sprung von Seite 1 aus auf eine Adresse außerhalb dieser Seite, wird geprüft, ob deren Adresse mit dem aktuell angenommenen Wert des `PRPR`-Registers identisch ist. Erfolgt ein Sprung von einer anderen Seite aus auf eine Adresse außerhalb Seite 1, wird geprüft, ob die Zieladresse innerhalb der gleichen Seite liegt. **WICHTIG**: Da der Programmzähler nur 12 Bit breit ist, ist es nicht möglich, ohne einen Umweg über Seite 1 von einer Seite in eine andere zu springen - Mit einem Umsetzen des `PRPR`-Registers außerhalb von Seite 1 würde man sich seinen eigenen Code unter den Füßen wegziehen.

ST9

Die ST9-Familie verwendet zur Adressierung von Code- und Datenbereich exakt die gleichen Befehle. Welcher Adreßraum dabei jeweils angesprochen wird, hängt vom

Stand des DP-Flags im Flag-Register ab. Damit AS bei absoluten Zugriffen überprüfen kann, ob man mit Symbolen aus dem korrekten Adreßraum arbeitet (das funktioniert natürlich *nur* bei absoluten Zugriffen!), muß man ihm per `ASSUME` mitteilen, ob das DP-Flag momentan auf 0 (Code) oder 1 (Daten) steht. Der Initialwert dieser Annahme ist 0.

uPD78(C)1x

Diese Prozessoren besitzen ein Register (V), mit dessen Hilfe die „Zeropage", d.h. die Lage der mit nur einem Byte adressierbaren Speicherzellen sich in Seitengrenzen im Speicher frei verschieben läßt. Da man aber aus Bequemlichkeitsgründen nicht mit Ausdrücken wie

```
        inrw    Lo(Zaehler)
```

arbeiten will, übernimmt AS diese Arbeit, allerdings nur unter der Voraussetzung, daß man ihm über einen `ASSUME`-Befehl den Inhalt des V-Registers mitteilt. Wird ein Befehl mit Kurzadressierung benutzt, so wird überprüft, ob die obere Hälfte des Adreßausdrucks mit dem angenommenen Inhalt übereinstimmt. Stimmt sie nicht, so erfolgt eine Warnung.

78K2

78K2 ist eine 8/16-Bit-Architektur, die nachträglich durch Banking auf einen (Daten-)Adreßraum von einem MByte erweitert wurde. Das Banking wird mit den Registern PM6 (Normalfall) bzw. P6 (alternativer Fall mit vorangestelltem `&`) realisiert, die die fehlenden oberen vier Bits nachliefern. Zumindest bei absoluten Adressen kann AS überprüfen, ob die gerade angesprochene, lineare 20-bittige Adresse innerhalb des gegebenen 64K-Fensters liegt.

78K3

Prozessoren mit 78K3-Kern besitzen Registerbänke mit insgesamt 16 Registern, die man über ihre Nummern ansprechen kann (`R0` bis `R15`) oder ihre symbolischen Namen (`X=R0, A=R1, C=R2, B=R3, VPL=R8, VPH=R9, UPL=R10, UPH=R11, E=R12, D=R13, L=R14, H=R15`). Der Prozessorkern besitzt ein Register-Auswahlbit (`RSS`), mit dem man das Mapping von A/X und B/C von R0..R3 auf R4..R7 umschaltet. Dies ist in erste Linie für Befehle wichtig, die implizit eines dieser Register benutzen (d.h. bei denen die Registernummer nicht im Maschinenbefehl kodiert ist). Man kann dem Assembler aber auch über ein

```
assume rss:1
```

mitteilen, daß die folgenden Befehle mit diesem geänderten Mapping arbeiten. Der Assembler wird für Befehle, in denen die Registernummer explizit kodiert ist, dann auch die alternativen Registernummern einsetzen. Umgekehrt wird dann z.B. auch `R5` statt `R1` im Quellcode wie `A` behandelt.

78K4

78K4 war als 'Upgrade-Pfad' vom 78K3 konzipiert, deshalb besitzt dessen Prozessorkern auch ein RSS-Bit, mit dem man das Mapping der Register AX und BC umschalten kann (auch wenn NEC von dessen Verwendung in neuem Code abrät).

Neben vielen neuen Befehlen und Adressierungsarten ist die wesentliche Erweiterung der größere Adreraum von 16 MByte, von dem allerdings nur das erste MByte für Programmcode genutzt werden kann. Das CPU-interne RAM sowie die Special Function Register können wahlweise am oberen Ende des ersten MByte oder der ersten 64 KByte Seite liegen. Dies teilt man dem Prozessor durch den `LOCATION`-Befehl mit, der als Argument wahlweise eine 0 oder 15 akzeptiert. Parallel damit schaltet der Prozessor auch die Adreßbereiche um, die mit kurzen (8-Bit) Adressen erreicht werden können. Parallel dazu muß man dem Assembler mittels `ASSUME LOCATION:..` ebenfalls dieser Wert mitgeteilt werden, damit er kurze Adressen in den dazu passenden Bereichen erzeugt. Der Assembler nimmt für LOCATION einen Default von Null an.

320C3x/C4x

Da alle Instruktionsworte dieser Prozessorfamilie nur 32 Bit lang sind, und von diesen 32 Bit nur 16 Bit für absolute Adressen vorgesehen wurden, müssen die fehlenden oberen 8/16 Bit aus dem DP-Register ergänzt werden. Bei Adressierungen kann man aber trotzdem die volle 24/32-Bit-Adresse angeben, AS prüft dann, ob die oberen 8/16 Bit mit dem angenommenen Inhalt von DP übereinstimmen. Gegenüber dem `LDP`-Befehl weicht `ASSUME` darin ab, daß man hier nicht eine beliebige Adresse aus der Speicherbank angeben kann, das Herausziehen der oberen Bits muß man also „zu Fuß" machen, z.B. so:

```
ldp     @adr
assume  dp:adr>>16
.
.
.
ldi     @adr,r2
```

75K0

Da selbst mit Hilfe von Doppelregistern (8 Bit) nicht der komplette Adreßraum von 12 Bit zu erreichen ist, mußte NEC (wie andere auch...) auf Banking zurück greifen: Die oberen 4 Adreßbits werden aus dem `MBS`-Register geholt (welchem demzufolge mit `ASSUME` Werte zwischen 0 und 15 zugeordnet werden können), das aber nur beachtet wird, falls das `MBE`-Flag auf 1 gesetzt wurde. Steht es (wie die Vorgabe ist) auf 0, so kann man die obersten und untersten 128 Nibbles des Adreßraumes ohne Bankumschaltung erreichen. Da der 75402 weder `MBE`-Flag noch `MBS`-Register kennt, ist für ihn der `ASSUME`-Befehl nicht definiert; Die Initialwerte von `MBE` und `MBS` lassen sich daher nicht ändern.

F^{2}MC16L

Wie viele andere Mikrokontroller auch, leidet diese Familie etwas unter der Knauserei seiner Entwickler: einem 24 Bit breiten Adreßraum stehen 16 Bit breite Adreßregister etwas unterbemittelt gegenüber. Also mußten wieder mal Bank-Register her. Im einzelnen sind dies PCB für den Programmcode, DTB für alle Datenzugriffe, ADB für indirekte Zugriffe über RW2/RW6 und SSB/USB für die Stacks. Sie können alle Werte zwischen 0 und 255 annehmen. Defaultmäßig stehen alle Annahmen von AS auf 0, mit Ausnahme von 0ffh für PCB.

Des weiteren existiert das DPR-Register, das angibt, welche Seite innerhalb der durch DTB gegebenen 64K-Bank mit 8-Bit-Adressen erreicht werden kann. Der Default für DPR ist 1, zusammen mit dem Default für DTB ergibt dies also eine Default-Seite bei 0001xxh.

MN1613

Beim MN1613 wurde eine Architektur mit 16-Bit-Adressen nachträglich erweitert, Dies wird durch einen Satz vier Bit breiter ,,Segment-Register" (CSBR, SSBR, TSR0 und TSR1) erreicht, deren Wert (um 14 Bit nach links geschoben) zu den 16-Bit-Adressen hinzu addiert wird. Ein Prozeß kann auf diese Weise immer ein 64 KWorte großes Fenster im 256 KWorte großen Adreßraum adressieren. Der Assembler benutzt die per `ASSUME` mitgeteilten Werte, um zu warnen, wenn eine absolute Adresse innerhalb des 256K-Adreßraums mit den aktuellen Werten nicht adressierbar ist, und rechnet ansonsten den korrekten 16-bittigen Offset aus. Bei indirekter Adressierung ist so eine Prüfung (naturgemäß) nicht möglich.

IM61x0

Diese Mikroprozessoren implementieren den Befehlssatz einer PDP/8 und unterstützen grundsätzlich einen Adreßraum von 4 Kiloworten. Durch Banking kann dieser auf acht ,,Felder" von 4 Kiloworten erweitert werden. Adressierung von Daten und Sprünge sind prinzipiell nur im gleichen Feld möglich, mit einer Ausnahme: über das IB-Register sind Sprünge in ein anderes 4K-Feld möglich. Dieses gibt die oberen Bits der insgesamt 15 Bit langen Zieladresse vor, falls IB per `ASSUME` auf einen Wert ungleich `NOTHING` gesetzt ist.

3.2.24 CKPT

Gültigkeit: TI990/12

Typ 12-Instruktionen erfordern für ihre Ausführung ein sogenanntes *Checkpoint Register*. Dieses Register kann entweder explizit als viertes Argument angegeben werden, oder es wird mit dieser Anweisung ein Default für allen folgenden Code festgelegt. Wenn weder eine `CKPT`-Anweisung noch ein explizites Register angegeben wurde, wird eine Fehlermeldung ausgegeben. Der Default von keinem Default-Register kann wiederhergestellt werden, indem man die `CKPT`-Anweisung mit `NOTHING` als Argument aufruft.

3.2.25 EMULATED

Gültigkeit: 29K

AMD hat die Ausnahmebehandlung für undefinierte Befehle bei der 29000-Serie so definiert, daß für jeden einzelnen Befehl ein Exception-Vektor zur Verfügung steht. Dies legt es nahe, durch gezielte Software-Emulationen den Befehlssatz eines kleineren Mitgliedes dieser Familie zu erweitern. Damit nun aber AS diese zusätzlichen Befehle nicht als Fehler anmeckert, erlaubt es der `EMULATED`-Befehl, AS mitzuteilen, daß bestimmte Befehle doch erlaubt sind. Die Prüfung, ob der momentan gesetzte Prozessor diesen Befehl beherrscht, wird dann übergangen. Hat man z.B. für einen Prozessor ohne Gleitkommaeinheit ein Modul geschrieben, das aber nur mit 32-Bit-IEEE-Zahlen umgehen kann, so schreibt man

```
EMULATED FADD,FSUB,FMUL,FDIV
EMULATED FEQ,FGE,FGT,SQRT,CLASS
```

`BRANCHEXT` mit `ON` oder `OFF` als Argument legt fest, ob AS kurze, nur mit einem 8-Bit-Displacement verfügbare Sprünge automatisch ,,verlängern" soll, indem z.B. aus einem einfachen

```
        bne     target
```

automatisch eine längere Sequenz mit gleicher Funktion wird, falls das Sprungziel zu weit von momentanen Programmzähler entfernt ist. Für `bne` wäre dies z.B. die Sequenz

```
        beq     skip
        jmp     target
skip:
```

Falls für eine Anweisung aber kein passendes ,,Gegenteil" existiert, kann die Sequenz auch länger werden, z.B. für `jbc`:

```
        jbc     dobr
        bra     skip
dobr:   jmp     target
skip:
```

Durch dieses Feature gibt es bei Sprüngen keine eineindeutige Zuordnung von Maschinen- und Assemblercode mehr, und bei Vorwärtsreferenzen handelt man sich möglicherweise zusätzliche Passes ein. Man sollte dieses Feature daher mit Vorsicht einsetzen!

3.2.26 Z80SYNTAX

Gültigkeit: 8008, 8080/8085, µPD78xx

Mit `ON` als Argument kann man Assmebler-Befehle wahlweise auch in der Form schreiben, wie sie Zilog für den Z80 definiert hat. Zum Beispiel benutzt man einfach nur noch `LD` mit sich selbst erklärenden Operanden, wo man in der originalen Syntax (bzw. neueren 8008-Syntax) je nach Operanden `MVI, LXI, MOV, STA, LDA, SHLD, LHLD, LDAX, STAX` oder `SPHL` schreiben muß.

Weil einige Mnemonics in der originalen und Z80-Syntax unterschiedliche Bedeutung haben, kann man nicht zu 100% im 'Z80-Stil' programmieren. Alternativ schaltet man für 8080/8085 mit einem `EXCLUSIVE` als Argument die originale Syntax ganz ab. Details zu dieser Betriebsart kann man im Abschnitt 4.22 nachlesen.

Ein eingebautes Symbol mit gleichem Namen gestattet es, die aktuelle Betriebsart auszulesen. Es gilt `0=OFF`, `1=ON` und `2=EXCLUSIVE`.

3.2.27 EXPECT und ENDEXPECT

Mit diesen beiden Befehlen rahmt man ein Stück Quellcode ein, in dem ein oder mehrere Fehler *erwartet* werden. Treten die über ihre Nummern (siehe Kapitel A) identifizierten Fehler oder Warnungen auf, werden sie unterdrückt, und die Assemblierung läuft ohne Fehler durch - natürlich ohne an dieser Stelle Code zu erzeugen. Erwartete, aber nicht aufgetretene Fehler oder Warnungen lösen ihrerseits jedoch eine Fehlermeldung von `ENDEXPECT` aus. Der Haupt-Anwendungszweck dieser Befehle findet sich in den Selbst-Tests im tests/-Unterverzeichnis. Z.B. kann man so testen, ob Wertebereiche korrekt geprüft werden:

```
cpu      68000
expect   1320     ; immediate-Shift nur 1..8
lsl.l    #10,d0
endexpect
```

3.3 Datendefinitionen

Die hier beschriebenen Befehle überschneiden sich teilweise in ihrer Funktionalität, jedoch definiert jede Prozessorfamilie andere Namen für die gleiche Funktion. Um mit den Standardassemblern konform zu bleiben, wurde diese Form der Implementierung gewählt.

Sofern nicht ausdrücklich anders erwähnt, kann bei allen Befehlen zur Datenablage (nicht bei denen zur Speicherreservierung!) eine beliebige Zahl von Parametern angegeben werden, die der Reihe nach abgearbeitet werden.

3.3.1 DC[.size]

*Gültigkeit: 680x0, M*Core, 68xx, H8, SH, DSP56xxx,*
XA, ST7/STM8, MN161x, IM61x0, CP-3F, SC61860

Dieser Befehl legt eine oder mehrere Konstanten des beim durch das Attribut bestimmten Typs im Speicher ab. Die Attribute entsprechen den in Abschnitt 2.5 definierten, zusätzlich ist für Byte-Konstanten die Möglichkeit vorhanden, Stringausdrücke im Speicher abzulegen, wie z.B.

```
String  dc.b    "Hello world!\0"
```

Die Parameterzahl darf zwischen 1 und 20 liegen, zusätzlich darf jedem Parameter ein in eckigen Klammern eingeschlossener Wiederholungsfaktor vorausgehen, z.B. kann man mit

```
dc.b    [(*+255)&$ffffff00-*]0
```

den Bereich bis zur nächsten Seitengrenze mit Nullen füllen. **Vorsicht!** Mit dieser Funktion kann man sehr leicht die Grenze von 1 Kbyte erzeugten Codes pro Zeile Quellcode überschreiten!

Sollte die Byte-Summe ungerade sein, so kann vom Assembler automatisch ein weiteres Byte angefügt werden, um die Wortausrichtung von Daten zu erhalten. Dieses Verhalten kann mit dem `PADDING`-Befehl ein- und ausgeschaltet werden.

Mit diesem Befehl abgelegte Dezimalgleitkommazahlen (`DC.P` ...) können zwar den ganzen Bereich der extended precision überstreichen, zu beachten ist dabei allerdings, daß die von Motorola verfügbaren Koprozessoren 68881/68882 beim Einlesen solcher Konstanten die Tausenderstelle des Exponenten ignorieren!

Default-Attribut ist `W`, also 16-Bit-Integerzahlen.

Beim DSP56xxx ist der Datentyp auf Integerzahlen festgelegt (ein Attribut ist deshalb weder nötig noch erlaubt), die im Bereich -8M..16M-1 liegen dürfen. Stringkonstanten sind ebenfalls erlaubt, wobei jeweils drei Zeichen in ein Wort gepackt werden.

Es ist im Gegensatz zum Original Motorola-Assembler auch erlaubt, mit diesem Kommando Speicher zu reservieren, indem man als Argument ein Fragezeichen angibt. Diese Erweiterung haben wohl einige Drittanbieter von 68K-Assemblern eingebaut, in Anlehnung an das, was Intel-Assembler machen. Wer dies benutzt, sollte sich aber im klaren sein, daß dies zu Problemen beim Portieren von Code auf andere Assembler führen kann. Des weiteren dürfen Fragezeichen als Operanden nicht mit 'normalen' Konstanten in einer Anweisung gemischt werden.

3.3.2 DS[.size]

*Gültigkeit: 680x0, M*Core, 68xx, H8, SH, DSP56xxx,*
XA, ST7/STM8, MN161x, IM61x0, CP-3F, PPS-4, SC61860

Mit diesem Befehl läßt sich zum einen Speicherplatz für die angegebene Zahl im Attribut beschriebener Zahlen reservieren. So reserviert

```
DS.B    20
```

z.B. 20 Bytes Speicher,

```
DS.X    20
```

aber 240 Byte !

Die andere Bedeutung ist die Ausrichtung des Programmzählers, die mit der Wertangabe 0 erreicht wird. So wird mit

```
DS.W    0
```

der Programmzähler auf die nächste gerade Adresse aufgerundet, mit

```
DS.D    0
```

dagegen auf die nächste Langwortgrenze. Eventuell dabei freibleibende Speicherzellen sind nicht etwa mit Nullen oder NOPs gefüllt, sondern undefiniert.

Vorgabe für die Operandengröße ist — wie üblich — W, also 16 Bit.

Beim 56xxx ist die Operandengröße auf Worte (a 24 Bit) festgelegt, Attribute gibt es deswegen wie bei `DC` auch hier nicht.

3.3.3 BLKB, BLKW, BLKL, BLKD

Gültigkeit: Renesas RX

Mit diesen Befehlen wird auf dem Renesas RX Speicher reserviert. Die Gesamtgröße des reservierten Speicher resultiert aus dem Produkt des Arguments und der durch den Befehl gegebenen Operandengröße (1 Byte für `BLKB`, 2 Byte für `BLKW`, 4 Byte für `BLKL` und 8 Byte für `BLKD`).

3.3.4 DN,DB,DW,DD,DQ,DT,DO & BF16

Gültigkeit: Intel (außer 4004/4040), Zilog, Toshiba, NEC, TMS370, Siemens, AMD, M16(C), MELPS7700/65816, National, ST9, Atmel, TMS7000, TMS1000, µPD77230, Signetics, Fairchild, Intersil, XS1, SC62015, WE32xxx

Diese Befehle stellen sozusagen das Intel-Gegenstück zu `DS` und `DC` dar, und wie nicht anders zu erwarten, ist die Logik etwas anders:

Zum einen wird die Kennung der Operandengröße in das Mnemonic verlegt:

- `DN`: 4-Bit-Integer
- `DB`: Byte oder ASCII-String wie bei `DC.B`
- `DW`: 16-Bit-Integer oder Gleitkomma mit halber Genauigkeit
- `DD`: 32-Bit-Integer oder Gleitkomma mit einfacher Genauigkeit
- `DQ`: 64-Bit-Integer oder Gleitkomma mit doppelter Genauigkeit
- `DT`: Gepackter BCD Integer oder Gleitkomma mit erweiterter Genauigkeit (80 Bit)
- `DO`: Gleitkomma mit vierfacher Genauigkeit (128 Bit)
- `BF16`: Gleitkomma im BFloat16-Format

Zum anderen erfolgt die Unterscheidung, ob Konstantendefinition oder Speicherreservierung, im Operanden. Eine Reservierung von Speicher wird durch ein `?` gekennzeichnet:

```
db  ?   ; reserviert ein Byte
dw  ?,? ; reserviert Speicher fuer 2 Worte (=4 Byte)
dd  -1  ; legt die Konstante -1 (FFFFFFFFH) ab !
```

Speicherreservierung und Konstantendefinition dürfen **nicht in einer Anweisung** gemischt werden:

```
db  "Hallo",?  ; -->Fehlermeldung
```

Zusätzlich ist noch der `DUP`-Operator erlaubt, der die mehrfache Ablage von Konstantenfolgen oder die Reservierung ganzer Speicherblöcke erlaubt:

```
db  3 dup (1,2) ; --> 1 2 1 2 1 2
dw  20 dup (?)  ; reserviert 40 Byte Speicher.
```

Wie man sehen kann, muß das `DUP`-Argument geklammert werden, darf dafür aber auch wieder aus mehreren Teilen bestehen, die selber auch wieder `DUP`s sein können...das ganze funktioniert also rekursiv.

`DUP` ist aber auch eine Stelle, an der man mit einer anderen Grenze des Assemblers in Berührung kommen kann: maximal können 1024 Byte Code oder Daten in einer Zeile erzeugt werden. Dies bezieht sich **nicht** auf die Reservierung von Speicher, nur auf die Definition von Konstantenfeldern!

Der DUP-Operator wird nur erkannt, wenn er selber geklammert ist und links von ihm ein nicht-leeres Argument steht; auf diese Weise ist es möglich ein eventuell definiertes Symbol gleichen Namens auch als Argument zu verwenden.

DB und DW auf 65xx- und 68xx-Plattformen verstehen neben dem DUP-Operator zusätzlich die 'Motorola-Variante' davon, nämlich eine in eckigen Klammern davor geschriebene Zahl von Wiederholungen.

Auf verschiedenen Plattformen existieren Befehle mit identischer Funktionalität, jedoch anderem Namen:

- Um mit dem M80 verträglich zu sein, darf im Z80-Modus anstelle von DB/DW auch DEFB/DEFW geschrieben werden.
- BYTE/ADDR bzw. WORD/ADDRW beim COP4/8 stellen einen Alias für DB bzw. DW dar, wobei die beiden Paare sich jedoch in der Byte-Order unterscheiden: Die Befehle, die von National zur Adreßablage vorgesehen waren, benutzen Big-Endian, BYTE bzw. WORD jedoch Little-Endian.
- BYTE bzw. WORD funktionieren auf der PDP-11, VAX und WD16 wie DB bzw. DW, akzeptieren aber nur Ganzzahl-Argumente.
- LWORD bzw. QUAD funktionieren auf der VAX wie DD bzw. DQ, akzeptieren aber nur Ganzzahl-Argumente.
- BYTE funktioniert auf WE32xxx wie DB, HALF wie DW, und WORD wie DD, akzeptiert aber nur Ganzzahl-Argumente.
- DOUBLE funktionieren auf WE32xxx wie DQ, akzeptiert aber nur Gleitkomma-Argumente)
- Auf dem TI TMS340xx existieren:
  - BYTE (wie DB, nur Ganzzahl-Argumente)
  - STRING (wie DB, nur Zeichenketten-Argumente)
  - WORD (wie DW, nur Ganzzahl-Argumente)
  - LONG oder INT (wie DD, nur Ganzzahl-Argumente)
  - FLOAT (wie DD, nur Gleitkomma-Argumente)
  - DOUBLE (wie DQ, nur Gleitkomma-Argumente)
- Auf dem Renesas RX existieren:
  - BYTE (wie DB)

  - `WORD` (wie `DW`, nur Ganzzahl-Argumente)
  - `LWORD` (wie `DD`, nur Ganzzahl-Argumente)
  - `FLOAT` (wie `DD`, nur Gleitkomma-Argumente)
  - `DOUBLE` (wie `DQ`, nur Gleitkomma-Argumente)

- Auf CR16 existieren:
  - `BYTE` oder `DC8` (wie `DB`, Strings oder Ganzzahl-Argumente)
  - `WORD` oder `DC16` (wie `DW`, nur Ganzzahl-Argumente)
  - `LONG` oder `DC32` (wie `DD`, nur Ganzzahl-Argumente)
  - `DC64` (wie `DQ`, nur Ganzzahl-Argumente)
  - `FLOAT` oder `DF32` (wie `DD`, nur Gleitkomma-Argumente)
  - `DOUBLE` oder `DF64` (wie `DQ`, nur Gleitkomma-Argumente)

Wird `DB` in einem Adreßraum angewendet, der nicht byte-adressierbar ist (z.B. das `CODE`-Segment des Atmel AVR), so werden immer zwei Bytes in ein 16-Bit-Wort gepackt, entsprechend der durch die Architektur gegebenen Endinaness - das untere Byte wird bei Little-Endian also zuerst gefüllt. Ist die Gesamtmenge aller Bytes ungerade, so bleibt die andere Worthälfte ungenutzt und ist quasi "Padding". Sie wird auch nicht genutzt, falls im Quellcode eine weitere `DB`-Anweisung unmittelbar folge sollte. Sinngemäß gilt das gleiche für `DN`, nur werden hier zwei oder vier Nibbles in ein Byte oder 16-Bit-Wort gepackt.

Der NEC 77230 nimmt mit seiner `DW`-Anweisung eine Sonderstellung ein: Sie funktioniert eher wie `DATA` bei seinen kleineren Brüdern, akzeptiert aber neben String- und Integer-Argumenten auch Gleitkommawerte (und legt sie prozessorspezifischen 32-Bit-Format ab). `DUP` gibt es *nicht*!

Bei der Ablage von Gleitkommazahlen ist zu beachten, daß diese keine höhere Genauigkeit und Wertebereich haben können als das auf dem Host-System verwendete Format. Handelt es sich dabei z.B. um das 'übliche' IEEE 754 64-Bit-Format, so ist der Wertebereich auf ca. $+/-1.8 * 10^{308}$ beschränkt (siehe Variable `FLOATMAX`), und die untersten zwei bzw. acht Bytes bei `DT` bzw. `DO` sind Null.

3.3.5 D1

Gültigkeit: MCS-51

`D1` funktioniert analog zu und , der Unterschied ist aber, daß hiermit nur ein einzelnes Bit abgelegt bzw. reserviert wird. Diese Anweisung ergibt daher auch nur in Adreßräumen eine Sinn, die bitweise adressiert werden. Das ist aktuell nur im `BITDATA`-Segment des 8051 der Fall, falls die `BITSEGSIZE:1`-Option benutzt wurde.

3.3.6 DC24

Gültigkeit: CR16

`DC24` funktioniert analog zu und , legt aber (Ganzzahl-)Werte mit 24 Bit Länge ab.

3.3.7 FLT

Gültigkeit: WE32xxx

`FLT` funktioniert analog zu `DD`, legt jedoch ausschließlich Gleitkomma-Konstanten im IEEE-Format mit einfacher Genauigkeit ab.

3.3.8 FLT2, FLT3, FLT4

Gültigkeit: PDP-11 (`FLT2, FLT4`),
WD16 (`FLT3`)

`FLT2` und `FLT4` funktionieren analog zu `DD` bzw. `DQ`, legen jedoch ausschließlich Gleitkomma-Konstanten im DEC-eigenen F- bzw. D-Format ab. Der WD16 verwendet hingegen ein eigenes, 48 Bit (drei Maschinenworte) langes Format. Gleitkomma-Konstanten in diesem Format können mit der `FLT3`-Anweisung im Speicher abgelegt werden.

3.3.9 x_FLOATING

Gültigkeit: VAX

Mit diesen Befehlen werden auf der VAX Gleitkomma-Konstanten in den DEC-eigenen Formaten im Speicher abgelegt, wobei `x` für die vier unterstützten Formate (F, D, G und H) steht. `FLOAT` ist ein Alias für `F_FLOATING`, `DOUBLE` einer für `D_FLOATING`. Die Befehle funktionieren ansonsten wie `DD` und `DQ`.

3.3.10 DS, DS8, DS16, DS32, DS64

Gültigkeit: Intel, Zilog, Toshiba, NEC, TMS370, Siemens, AMD, M16(C), National, ST9, TMS7000, TMS1000, Intersil, 6502, 68xx, WE32xxx, CR16
CR16, SC14xxx (DS8)
CR16 (DS16, DS32, DS64)

Dieser Befehl stellt eine Kurzschreibweise dar, um Speicherbereiche zu reservieren:

`DS` < *Anzahl* >

ist eine Kurzschreibweise für

`DB` < *Anzahl* > `DUP (?)`

dar, ließe sich also prinzipiell auch einfach über ein Makro realisieren, nur scheint dieser Befehl in den Köpfen einiger mit Motorola-CPUs groß gewordener Leute so fest verdrahtet zu sein, daß sie ihn als eingebauten Befehl erwarten.

`DS8` ist beim National SC14xxx als Alias für `DS` definiert. Achten Sie aber darauf, daß der Speicher dieser Prozessoren in Worten zu 16 Bit organisiert ist, d.h. es ist unmöglich, einzelne Bytes zu reservieren. Falls das Argument von `DS` ungerade ist, wird es auf die nächstgrößere gerade Zahl aufgerundet.

3.3.11 BLKx

Gültigkeit: VAX, CR16

Diese Befehle reservieren Speicher für die angegebene Menge von Datenelementen, deren Typ im letzten Zeichen des Befehls kodiert ist. Dementsprechend ist der insgeamt reservierte Bereich in Bytes bei `BLKB` gleich der Anzahl der Elemente, bei `BLKW` das doppelte, und bei `BLKO` und `BLKH` das sechzehnfache.

3.3.12 BYT oder FCB

Gültigkeit: 6502, 68xx, SC61860

Mit diesem Befehl werden im 65xx/68xx-Modus Byte-Konstanten oder ASCII-Strings abgelegt, er entspricht also `DC.B` beim 68000 oder bei Intel dem `DB` (was ebenfalls erlaubt ist). Ein Wiederholungsfaktor darf analog zu `DC` jedem einzelnen Parameter in eckigen Klammern vorangestellt werden.

3.3.13 BYTE

Gültigkeit: ST6, 320C2(0)x, 320C5x, MSP, TMS9900, CP-1600

Dito. Ein im 320C2(0)x/5x-Modus vor dem Befehl stehendes Label wird als untypisiert gespeichert, d.h. keinem Adreßraum zugeordnet. Der Sinn dieses Verhaltens wird bei den prozessorspezifischen Hinweisen erläutert.

Ob beim MSP bzw. TMS9900 ungerade Mengen von Bytes automatisch um ein Null-Byte ergänzt werden sollen, kann mit dem PADDING-Befehl eingestellt werden.

Auf dem CP-1600 weicht die Verhaltensweise von `BYTE` etwas ab: die angegebenen 16-Bit-Werte werden in jeweils zwei aufeinander folgenden Wörtern aufgeteilt abgelegt (LSB zuerst). Wenn einzelne Bytes (gepackt) abgelegt werden sollen, muß stattdessen `TEXT` verwendet werden!

3.3.14 DC8

Gültigkeit: SC144xx

Dieser Befehl ist ein Alias für `DB`, d.h. mit ihm können Byte-Konstanten oder Strings im Speicher abgelegt werden.

3.3.15 ADR oder FDB

Gültigkeit: 6502, 68xx, SC61860

Mit diesem Befehl werden im 65xx/68xx-Modus Wortkonstanten abgelegt, er entspricht also `DC.W` beim 68000 oder bei Intel dem `DW` (was auch zulässig ist). Ein Wiederholungsfaktor darf analog zu `DC` jedem einzelnen Parameter in eckigen Klammern vorangestellt werden.

3.3.16 DDB

Gültigkeit: 6502, MELPS-7700

Dieser Befehl funktioniert analog zu `ADR`, nur mit dem Unterschied, daß die 16-BitWerte im Big-Endian-Format im Speicher abgelegt werden.

3.3.17 DCM

Gültigkeit: 6502

Mit diesem Befehl werden Gleitkomma-Konstanten im Speicher abgelegt, und zwar in dem in [3] beschriebenen Format: Ein Exponent von 8 Bit und eine 24-bittige Mantisse im Zweierkomplement, im Big-Endian-Format abgelegt.

3.3.18 WORD

Gültigkeit: ST6, i960, 320C2(0)x, 320C3x/C4x/C5x, MSP, CP-1600, IMP-16, IPC-16

Für den 320C3x/C4x und i960 werden hiermit 32-Bit-Worte abgelegt, für die alle anderen Familien 16-Bit-Worte. Ein im 320C2(0)x/5x-Modus vor dem Befehl stehendes Label wird als untypisiert gespeichert, d.h. keinem Adreßraum zugeordnet. Der Sinn dieses Verhaltens wird bei den prozessorspezifischen Hinweisen erläutert.

3.3.19 DW16

Gültigkeit: SC144xx

Dieser Befehl ist beim SC144xx der Weg, Konstanten mit Wortlänge (16 Bit) im Speicher abzulegen und damit ein ALIAS für DW.

3.3.20 ACON

Gültigkeit: 2650

`ACON` arbeitet analog zu `DW`, jedoch werden die 16-Bit-Zahlenwerte im Big-Endian-Format abgelegt.

3.3.21 LONG

Gültigkeit: 320C2(0)x, 320C5x

Hiermit werden 32-Bit-Integer im Speicher abgelegt, und zwar in der Reihenfolge LoWord-HiWord. Ein eventuell vor dem Befehl stehendes Label wird dabei wieder als untypisiert abgelegt (der Sinn dieser Maßnahme ist in den prozessorspezifischen Hinweisen erläutert).

3.3.22 SINGLE, DOUBLE und EXTENDED

Gültigkeit: 320C3x/C4x (nicht `DOUBLE`*), 320C6x (nicht* `EXTENDED`*)*

Mit diesen Befehlen werden Gleitkomma-Konstanten im Speicher abgelegt, jedoch beim 320C3x/C4x nicht im IEEE-Format, sondern in den vom Prozessor verwendeten 32- und 40-Bit-Formaten. Da 40 Bit nicht mehr in eine Speicherzelle hineinpassen, werden im Falle von `EXTENDED` immer derer 2 pro Wert belegt. Im ersten Wort finden sich die oberen 8 Bit (der Exponent), der Rest (Vorzeichen und Mantisse) in zweiten Wort.

3.3.23 FLOAT und DOUBLE

Gültigkeit: 320C2(0)x, 320C5x

Mit diesen Befehlen können 32- bzw. 64-Bit-Gleitkommazahlen im IEEE-Format im Speicher abgelegt werden. Dabei wird das niederwertigste Byte jeweils auf der ersten Speicherstelle abgelegt. Ein eventuell vor dem Befehl stehendes Label wird wieder als untypisiert gespeichert (der Sinn dieser Maßnahme ist in den prozessorspezifischen Hinweisen erläutert).

3.3.24 SINGLE und DOUBLE

Gültigkeit: TMS99xxx

Mit diesen Befehlen können 32- bzw. 64-Bit-Gleitkommazahlen im prozessoreigenen Format im Speicher abgelegt werden. Das Format entspricht dem IBM/360-Gleitkommaformat.

3.3.25 EFLOAT, BFLOAT, TFLOAT

Gültigkeit: 320C2(0)x, 320C5x

Auch diese Befehle legen Gleitkommazahlen im Speicher ab, jedoch in einem nicht-IEEE-Format, das evtl. leichter von Signalprozessoren zu verarbeiten ist:

- `EFLOAT`: Mantisse mit 16 Bit, Exponent mit 16 Bit
- `BFLOAT`: Mantisse mit 32 Bit, Exponent mit 16 Bit

- `TFLOAT`: Mantisse mit 64 Bit, Exponent mit 32 Bit

Gemeinsam ist den Befehlen, daß die Mantisse vor dem Exponenten abgelegt wird (Lo-Word jeweils zuerst) und beide im Zweierkomplement dargestellt werden. Ein eventuell vor dem Befehl stehendes Label wird wieder als untypisiert gespeichert (der Sinn dieser Maßnahme ist in den prozessorspezifischen Hinweisen erläutert).

3.3.26 Qxx und LQxx

Gültigkeit: 320C2(0)x, 320C5x

Mit diesen Befehlen können Gleitkommazahlen in einem Festkommaformat abgelegt werden. `xx` ist dabei eine zweistellige Zahl, mit deren Zweierpotenz der Gleitkommawert vor der Umwandlung in eine ganze Zahl multipliziert werden soll. Er bestimmt also praktisch, wieviele Bits für die Nachkommastellen reserviert werden sollen. Während aber `Qxx` nur ein Wort (16 Bit) ablegt, wird das Ergebnis bei `LQxx` in 2 Worten (LoWord zuerst) abgelegt. Das sieht dann z.B. so aus:

```
q05     2.5      ; --> 0050h
lq20    ConstPI  ; --> 43F7h 0032h
```

Mich möge niemand steinigen, wenn ich mich auf meinem HP28 verrechnet haben sollte...

3.3.27 FIELD

Gültigkeit: TMS340xx

`FIELD` legt im Speicher ein Bitfeld mit einer bestimmten Länge und Wert ab. Das zweite Argument ist die Länge des Feldes (1 bis 32 Bit), das erste ist der darin abzulegende (Integer-)Wert. Der erlaubte Wertebereich ist von der Länge abhängig und kann wahlweise vorzeichenlos oder vorzeichenbehaftet im Zweierkomplement sein. Für ein Bitfeld mit 16 Bit Länge geht der erlaubte Wertebereich z.B. von -32768 bis +65535.

3.3.28 DATA

Gültigkeit: PIC, 320xx, AVR, MELPS-4500, H8/500, HMCS400, 4004/4040, µPD772x, OLMS-40/50, Padauk

Mit diesem Befehl werden Daten im aktuellen Segment abgelegt, wobei sowohl Integer- als auch Stringwerte zulässig sind. Bei Strings belegt beim 16C5x/16C8x, 17C4x im Datensegment, beim 4500er, 4004 und HMCS400 im Code-Segement ein Zeichen ein Wort, bei AVR, 17C4x im Codesegment, µPD772x in den Datensegmenten und 3201x/3202x passen zwei Zeichen in ein Wort (LSB zuerst), beim µPD7725 drei und beim 320C3x/C4x sogar derer 4 (MSB zuerst). Im Gegensatz dazu muß im Datensegment des 4500 bzw. ein Zeichen auf zwei Speicherstellen verteilt werden, ebenso wie beim 4004 und HMCS400. Der Wertebereich für Integers entspricht der Wortbreite des jeweiligen Prozessors im jeweiligen Segment. Das bedeutet, daß `DATA` beim 320C3x/C4x die Funktion von `WORD` mit einschließt (die von `SINGLE` übrigens auch, wenn AS das Argument als Gleitkommazahl erkennt).

3.3.29 ZERO

Gültigkeit: PIC, CP-1600

Dieser Befehl legt einen durch den Parameter spezifizierte Zahl von Nullworten (=NOP bei PIC) im Speicher ab.

3.3.30 FB und FW

Gültigkeit: COP4/8

Mit diesen Befehlen kann ein größerer Block von Speicher (dessen Länge in Bytes bzw. Worten der erste Parameter angibt) mit einer Byte- bzw. Wortkonstanten gefüllt werden, die durch den zweiten Parameter angegeben wird.

3.3.31 ASCII, ASCIC und ASCIZ

Gültigkeit: ST6, PDP-11, VAX, IMP-16, IPC-16 (`ASCII`)
ST6, PDP-11, VAX (`ASCIZ`)
PDP-11, VAX (`ASCIC`)

Mit diesen Befehlen können Stringkonstanten im Speicher abgelegt werden. Während ASCII nur die reinen Daten im Speicher ablegt, versieht `ASCIZ` automatisch *jeden* angegebenen String mit einem NUL-Zeichen am Ende, und `ASCIC` stellt ihm ein Längen- Byte voran.

3.3.32 STRING und RSTRING

Gültigkeit: 320C2(0)x, 320C5x

Diese Anweisungen funktionieren analog zu `DATA`, jedoch werden hier Integer-Ausdrücke grundsätzlich als *Bytes* mit einem entsprechend eingeschränkten Wertebereich betrachtet, wodurch es möglich wird, die Zahlen zusammen mit anderen Zahlen oder Zeichen paarweise in Worte zu verpacken. Die beiden Befehle unterscheiden sich lediglich in der Reihenfolge der Bytes in einem Wort: Bei `STRING` wird zuerst das obere und danach das untere gefüllt, bei `RSTRING` ist es genau umgekehrt.

Ein eventuell vor dem Befehl stehendes Label wird wieder als untypisiert gespeichert. Der Sinn dieser Maßnahme ist im entsprechenden Kapitel mit den prozessorspezifischen Befehlen erläutert.

3.3.33 PACKED

Gültigkeit: PDP-11, VAX

Mit diesem Befehl werden Zahlen im gepackten dezimalen Format (zwei Stellen pro Byte) im Speicher abgelegt. Als Argument sind entweder eine (vorzeichenbehaftete) Ganzzahl oder ein String erlaubt, wobei letzterer natürlich nur dezimale Ziffern und optional ein Vorzeichen am Anfang enthalten darf. Die Maximalzahl Stellen ist 31, und das Vorzeichen wird als letzte Stelle angefügt, wobei 12 und 13 für Plus bzw. Minus stehen. Ist die Gesamtzahl aus Stellen und Vorzeichen ungerade, wird vorne eine Null eingefügt.

Optional darf als zweites Argument des Name eines Symbols angegeben werden, in dem die Anzahl Stellen hinterlegt wird. In dieser Stellenzahl sind weder Vorzeichen noch eine automatisch eingefügte Null zu Beginn enthalten.

3.3.34 RADIX50

Gültigkeit: diverse

Mit dieser Anweisung kann ein String im gepackten RADIX50-Format im Speicher abgelegt werden. Diese Darstellung war besonders im DEC-Umfeld für Dateinamen gängig und packt drei Zeichen in ein 16-Bit-Wort. `RADIX50` ist keine eingebaute Anweisung, sondern in der Include- Datei `radix50.inc` als Makro definiert.

3.3.35 FCC

Gültigkeit: 6502, 68xx

Mit diesem Befehl werden im 65xx/68xx-Modus String-Konstanten abgelegt. Beachten Sie jedoch, daß im Gegensatz zum Originalassembler AS11 von Motorola (dessentwegen dieser Befehl existiert, bei AS ist diese Funktion im `BYT`-Befehl enthalten), String-Argumente nur in Gänsefüßchen und nicht in Hochkommas oder Schrägstrichen eingeschlossen werden dürfen! Ein Wiederholungsfaktor darf analog zu `DC` jedem einzelnen Parameter in eckigen Klammern vorangestellt werden.

3.3.36 TEXT

Mit diesem Befehl werden im CP-1600-Modus String-Konstanten gepackt, d.h. zwei Zeichen pro Wort, abgelegt.

3.3.37 DFS oder RMB

Gültigkeit: 6502, 68xx

Dieser Befehl dient im 65xx/68xx-Modus zur Reservierung von Speicher, er entspricht `DS.B` beim 68000 oder `DB ?` bei Intel.

3.3.38 BLOCK

Gültigkeit: ST6

Dito.

3.3.39 SPACE

Gültigkeit: i960

Dito.

3.3.40 RES

Gültigkeit: PIC, MELPS-4500, HMCS400, 3201x, 320C2(0)x, 320C5x, AVR, µPD772x, OLMS-40/50, Padauk, CP-1600, PPS-4, 2650

Dieser Befehl dient zur Reservierung von Speicher. Er reserviert im Codesegment immer Wörter (10/12/14/16 Bit), im Datensegment bei den PICs Bytes, beim 4500er, PPS-4 und OLMS-40/50 Nibbles sowie bei TI Wörter.

3.3.41 BSS

Gültigkeit: 320C2(0)x, 320C3x/C4x/C5x/C6x, MSP

`BSS` arbeitet analog zu `RES`, lediglich ein eventuell vor dem Befehl stehendes Symbol wird beim 320C2(0)x/5x als untypisiert gespeichert. Der Sinn dieser Maßnahme kann im Kapitel mit den prozessorspezifischen Hinweisen nachgelesen werden.

3.3.42 DSB und DSW

Gültigkeit: COP4/8

Diese beiden Befehle stellen im COP4/8-Modus die zum ASMCOP von National kompatible Methode dar, Speicher zu reservieren. Während `DSB` nur einzelne Bytes freihält, reserviert `DSW` Wörter und damit effektiv doppelt soviel Bytes wie `DSB`.

3.3.43 DS16

Gültigkeit: SC144xx

Dieser Befehl reserviert Speicher in Schritten von vollständigen Worten, d.h. 16 Bit. Er stellt einen Alias zu `DW` dar.

3.3.44 ALIGN

Gültigkeit: alle Prozessoren

`ALIGN` mit einem Integer-Ausdruck als Argument erlaubt es, den Programmzähler auf eine bestimmte Adresse auszurichten. Die Ausrichtung erfolgt dergestalt, daß der Programmzähler so weit erhöht wird, daß er ein ganzzahliges mehrfaches des Argumentes wird. In seiner Funktion entspricht `ALIGN` also `DS.x 0` beim den 680x0ern, nur ist die Ausrichtung noch flexibler.

Beispiel:

```
align 2
```

macht den Programmzähler gerade. Wird `ALIGN` in dieser Form mit nur einem Argument verwendet, ist der Inhalt des dadurch frei bleibenden Speicherbereichs nicht definiert. Alternativ kann als zweites Argument ein (Byte-)Wert angegeben werden, mit dem dieser Bereich gefüllt wird.

3.3.45 EVEN

Gültigkeit: TMS340xx

`EVEN` ist eine Spezialisierung von `ALIGN`. Der Befehl inkrementiert den Programmzähler so weit, daß er ein mehrfaches von 16 Bit ist, also passend zum Beispiel zur Ablage von Prozessorbefehlen.

3.3.46 LTORG

Gültigkeit: SH, IM61x0, IMP-16, IPC-16

Da der SH-Prozessor seine Register immediate nur mit 8-Bit-Werten laden kann, AS dem Programmierer jedoch vorgaukelt, daß es eine solche Einschränkung nicht gäbe, muß er die dabei entstehenden Konstanten irgendwo im Speicher ablegen. Da es nicht sinnvoll wäre, dies einzeln zu tun (wobei jedes Mal Sprungbefehle anfallen würden...), werden die Literale gesammelt und können vom Programmierer mit diesem Befehl gezielt blockweise (z.B. am Ende eines Unterprogrammes) abgelegt werden. Zu den zu beachtenden Details und Fallen sei auf das Kapitel mit den SH-spezifischen Dingen hingewiesen.

3.4 Makrobefehle

Gültigkeit: alle Prozessoren

Kommen wir nun zu dem, was einen Makroassembler vom normalen Assembler unterscheidet: der Möglichkeit, Makros zu definieren (ach was ?!).

Unter Makros verstehe ich hier erst einmal eine Menge von Anweisungen (normal oder Pseudo), die mit bestimmten Befehlen zu einem Block zusammengefaßt werden und dann auf bestimmte Weise bearbeitet werden können. Zur Bearbeitung solcher Blöcke kennt der Assembler folgende Befehle:

3.4.1 MACRO

ist der wohl wichtigste Befehl zur Makroprogrammierung. Mit der Befehlsfolge

```
<Name>  MACRO   [Parameterliste]
        <Befehle>
        ENDM
```

wird das Makro <`Name:` > als die eingeschlossene Befehlsfolge definiert. Diese Definition alleine erzeugt noch keinen Code! Dafür kann fortan die Befehlsfolge einfach durch den Namen abgerufen werden, das Ganze stellt also eine Schreiberleichterung dar. Um die ganze Sache etwas nützlicher zu machen, kann man der Makrodefinition eine Parameterliste mitgeben.

Während der Makroname selber nur den üblichen Regeln für Symbolnamen (2.7) genügen muß, gilt für die Namen der (kommaseparierten) Parameternamen die weitere Einschränkung, daß sie keine Unterstriche enthalten dürfen. Dies kann durch den Kommandozeilen-Parameter `-underscore-macroargs` aufgehoben werden, hat aber Nebenwirkungen bei der Expansion von Makros und sollte nur eingesetzt werden, wenn unbedingt nötig.

Sowohl Makronamen als auch -parameter sind von einer Umschaltung von AS in den case-sensitiven Modus betroffen.

Makros sind ähnlich wie Symbole lokal, d.h. bei Definition in einer Sektion sind sie nur in dieser Sektion und ihren Untersektionen bekannt. Dieses Verhalten läßt sich aber durch die weiter unten beschriebenen Optionen `PUBLIC` und `GLOBAL` in weiten Grenzen steuern.

Für jeden Makroparameter kann ein Defaultwert mit angehängtem Gleichheitszeichen angegeben werden. Dieser Wert wird für den Parameter eingesetzt, wenn

beim Makroaufruf kein Argument für diesen Parameter angegeben wird, bzw. wenn ein Positionsargument (s.u.) für diesen Parameter leer ist.

Neben den eigentlichen Makroparametern können in der Parameterliste auch Steuerparameter enthalten sein, die die Abarbeitung des betroffenen Makros beeinflussen; diese Parameter werden von normalen Parametern dadurch unterschieden, daß sie in geschweifte Klammern eingeschlossen sind. Es sind folgende Steuerparameter definiert:

- `EXPAND/NOEXPAND` : legen fest, ob bei der späteren Verwendung diese Makros der expandierte Code mit angezeigt werden soll. Default ist der durch den Pseudobefehl `MACEXP_DFT` festgelegte Wert.
- `EXPIF/NOEXPIF` : legen fest, ob bei der späteren Verwendung diese Makros Befehle zur bedingten Assemblierung und dadurch ausgeschlossener Code angezeigt werden soll. Default ist der durch den Pseudobefehl `MACEXP_DFT` festgelegte Wert.
- `EXPMACRO/NOEXPMACRO` : legen fest, ob bei der späteren Verwendung diese Makros darin definierte Makros angezeigt werden sollen. Default ist der durch den Pseudobefehl `MACEXP_DFT` festgelegte Wert.
- `EXPREST/NOEXPREST` : legen fest, ob bei der späteren Verwendung Code-Zeilen angezeigt werden sollen, die weder Makro-Definitionen, bedingte Assemblierung noch durch bedingte Assemblierung ausgeschlossene Zeilen sind. Default ist der durch den Pseudobefehl `MACEXP_DFT` festgelegte Wert.
- `PUBLIC[:Sektionsname]` : ordnet das Makro nicht der aktuellen, sondern einer ihr übergeordneten Sektion zu. Auf diese Weise kann eine Sektion Makros für die „Außenwelt" zur Verfügung stellen. Fehlt eine Sektionsangabe, so wird das Makro völlig global, d.h. ist überall benutzbar.
- `GLOBAL[:Sektionsname]` : legt fest, daß neben diesem Makro noch ein weiteres Makro abgelegt werden soll, das zwar den gleichen Inhalt hat, dessen Name aber zusätzlich mit dem Namen der Sektion versehen ist, in der es definiert wurde und das der spezifizierten Sektion zugeordnet werden soll. Bei dieser muß es sich um eine Obersektion zu der aktuellen Sektion handeln; fehlt die Angabe, so wird das zusätzliche Makro global sichtbar. Wird z.B. ein Makro `A` in der Sektion `B` definiert, die wiederum eine Untersektion der Sektion `C` ist, so würde neben z.B. dem Makro A ein weiteres globales mit dem Namen `C_B_A` erzeugt. Würde dagegen `C` als Zielsektion angegeben, so würde das Makro `B_A` heißen und der Sektion `C` zugeordnet. Diese Option ist defaultmäßig ausgeschaltet und hat auch nur einen Effekt, falls sie innerhalb einer Sektion benutzt wird. Das lokal bekannte Originalmakro wird von ihr nicht beeinflußt.

- `EXPORT/NOEXPORT` : legen fest, ob die Definition dieses Makros in einer getrennten Datei abgelegt werden soll, falls die Kommandozeilenoption `-M` gegeben wurde. Auf diese Weise können einzelne Definitionen ,,privater" Makros selektiv ausgeblendet werden. Der Default ist FALSE, d.h. die Definition wird nicht in der Datei abgelegt. Ist zusätzlich die `GLOBAL`-Option gegeben worden, so wird das Makro mit dem modifizierten Namen abgelegt.

- `INTLABEL/NOINTLABEL` : legen fest, ob ein in der Zeile mit dem Makroaufruf definiertes Label innerhalb des Rumpfes als zusätzlicher Parameter verwendet werden soll, als einfach nur die Adresse dieser Zeile zu 'labeln'.

- `GLOBALSYMBOLS/NOGLOBALSYMBOLS` : legt fest, ob im Makro definierte Labels lokal zu diesem Makro sein sollen oder auch außerhalb des Makros verfügbar sein sollen. Der Default ist, daß Labels lokal sind, weil mehrfache Benutzung eines Makros ansonsten schwierig wäre.

Diese eben beschriebenen Steuerparameter werden von AS aus der Parameterliste ausgefiltert, haben also keine weitere Wirkung in der folgenden Verarbeitung und Benutzung.

Beim Aufruf eines Makros werden die beim Aufruf angegebenen Parameternamen überall textuell im Befehlsblock eingesetzt und der sich so ergebene Assemblercode wird normal assembliert. Sollten beim Aufruf zu wenige Parameter angegeben werden, werden Nullstrings eingefügt. Wichtig ist zu wissen, daß bei der Makroexpansion keine Rücksicht auf eventuell in der Zeile enthaltene Stringkonstanten genommen wird. Zu diesem Detail gilt die alte IBM-Regel:

It's not a bug, it's a feature!

Diese Lücke kann man bewußt ausnutzen, um Parameter mittels Stringvergleichen abzuprüfen. So kann man auf folgende Weise z.B. prüfen, wie ein Makroparameter aussieht:

```
mul     MACRO   para,parb
        IF      UpString("PARA")<>"A"
         MOV    a,para
        ENDIF
        IF      UpString("PARB")<>"B"
         MOV    b,parb
        ENDIF
        !mul     ab
        ENDM
```

Wichtig ist bei obigem Beispiel, daß der Assembler alle Parameternamen im casesensitiven Modus in Großbuchstaben umsetzt, in Strings aber nie eine Umwandlung in Großbuchstaben erfolgt. Die Makroparameternamen müssen in den Stringkonstanten daher groß geschrieben werden.

Des weiteren wird die Parameter-Ersetzung davon beeinflusst, ob Unterstriche im Parameternamen erlaubt sind oder nicht. Falls nicht — was der Default ist — wirken Unterstriche auch als 'Begrenzungszeichen', um Parameternamen im Makrorumpf zu erkennen. Im Default würde also im folgenden Beispiel

```
setled  macro led,value
        out   led,value
        ld    led_shadow,value
        endm
```

der Parameter 'led' in beiden Zeilen ersetzt, mit dem Kommandozeilen-Schalter `-underscore-macroargs` jedoch nur in der ersten. Diverse der mitgelieferten Include-Dateien setzen das Default-Verhalten voraus, weshalb dieser Schalter nur wenn unbedingt nötig genutzt werden sollte.

Argumente an ein Makro können in zwei Formen angegeben werden: als *Positionsargumente* oder als *Schlüsselwortargumente*.

Bei Positionsargumenten ergibt sich die Zuordnung von Argumenten zu Makro-Parametern einfach durch ihre Position in der Aufrufliste, d.h. das erste Argument wird dem ersten Parameter zugeordnet, das zweite Argument dem zweiten Parameter usw.. Werden weniger Argumente angegeben als das Makro Parameter hat, werden eventuell definierte Defaultwerte oder ein Leerstring eingesetzt. Gleiches gilt auch für leere Argumente.

Schlüsselwortargumente geben jedoch explizit an, für welchen Makro-Parameter sie gelten, indem der Parametername dem Wert vorangestellt wird, z.B. so:

```
      mul  para=r0,parb=r1
```

Wiederum wird für nicht definierte Parameter ein eventuell vorhandener Default oder ein Leerstring eingesetzt.

Im Unterschied zu Positionsargumenten ist es mit Schlüsselwortargumenten auch möglich, einem Parameter einen Leerstring zuzuweisen, der einen nicht-leeren Default-Wert hat.

Positions- und Schlüsselwortargumente dürfen auch in einem Aufruf gemischt werden, jedoch dürfen ab dem ersten Schlüsselwortargument keine Positionsargumente mehr verwendet werden.

Für die Makroparameter gelten die gleichen Konventionen wie bei normalen Symbolen, mit der Ausnahme, daß hier nur Buchstaben und Ziffern zugelassen sind, also weder Punkte noch Unterstriche. Diese Einschränkung hat ihren Grund in einem verstecktem Feature: Der Unterstrich erlaubt es, einzelne Makroparameternamen zu einem Symbol zusammenzuketten, z.B. in folgendem Beispiel:

```
concat  MACRO   part1,part2
        CALL    part1_part2
        ENDM
```

Der Aufruf

```
        concat  Modul,Funktion
```

ergibt also

```
        CALL    Modul_Funktion
```

Neben den am Makro selber angegebenen Parametern existieren vier weitere 'implizite' Parameter, die immer vorhanden sind und daher nicht als eigene Makroparameter verwendet werden sollten:

- `ATTRIBUTE` bezeichnet bei Architekturen, die Attribute für Prozessorbefehle zulassen, das bei einem Makroaufruf angehängte Argument. Für ein Beispiel siehe z.B. unten!
- `ALLARGS` bezeichnet eine kommaseparierte Liste aller Makroargumente, z.B., um sie an eine `IRP`-Anweisung weiterzureichen.
- `ARGCOUNT` bezeichnet die tatächlich übergebene Anzahl der an das Makro übergebenen Argumente. Zu beachten ist allerdings, daß diese Zahl niemals geringer als die Zahl der formalen Parameter ist, da AS fehlende Argumente mit Leerstrings auffüllt!
- `__LABEL__` bezeichnet das Label, das in der das Makro aufrufenden Zeile stand. Diese Ersetzung findet nur statt, wenn für dieses Makro die `INTLABEL`-Option gesetzt wurde!

WICHTIG: Die Namen dieser impliziten Parameter sind auch case-insensitiv, wenn AS insgesamt angewiesen wurde, case-sensitiv zu arbeiten!

Der Zweck, ein Label 'intern' im Makro verwenden zu können, ist sicher nicht unmittelbar einleuchtend. Den einen oder anderen Fall mag es ja geben, in dem

es sinnvoll ist, den Einsprungpunkt in ein Makro irgendwo in seinen Rumpf zu verschieben. Der wichtigste Anwendungsfall sind aber TI-Signalprozessoren, die eine Parallelisierung von Befehlen durch einen doppelten senkrechten Strich in der Label-Spalte kennzeichnen, etwa so:

```
    instr1
||  instr2
```

(da die beiden Instruktionen im Maschinencode in ein Wort verschmelzen, kann man die zweite Instruktion übrigens gar nicht separat anspringen - man verliert also durch das Belegen der Label-Position nichts). Das Problem ist aber, daß einige 'Bequemlichkeits-Befehle' durch Makros realisiert werden. Ein vor das Makro geschriebenes Parallelisierungssymbol würde normalerweise dem Makro selber zugeordnet, *nicht dem ersten Befehl im Makro selber*. Aber mit diesem Trick funktioniert's:

```
myinstr    macro {INTLABEL}
__LABEL__  instr2
           endm

           instr1
||         myinstr
```

Das Ergebnis nach der Expansion von `myinstr` ist identisch zu dem vorherigen Beispiel ohne Makro.

Rekursion von Makros, also das wiederholte Aufrufen eines Makros innerhalb seines Rumpfes oder indirekt über andere von ihm aufgerufene Makros ist vollkommen legal. Wie bei jeder Rekursion muß man dabei natürlich sicherstellen, daß sie irgendwann ein Ende findet. Für den Fall, daß man dies vergessen hat, führt AS in jedem definierten Makro einen Zähler mit, der bei Beginn einer Makroexpansion inkrementiert und an deren Ende wieder dekrementiert wird. Bei rekursiven Aufrufen eines Makros erreicht dieser Zähler also immer höhere Werte, und bei einem per `NESTMAX` einstellbaren Wert bricht AS ab. Vorsicht, wenn man diese Bremse abschaltet: der Speicherbedarf auf dem Heap kann so beliebig steigen und selbst ein Unix-System in die Knie zwingen...

Um alle Klarheiten auszuräumen, ein einfaches Beispiel: Ein an ,,Intel-Syntax" gewöhnter Programmierer möchte die Befehle `PUSH/POP` unbedingt auch auf dem 68000 haben. Er löst das folgendermaßen:

```
push    MACRO   op
        MOVE.ATTRIBUTE op,-(sp)
```

```
        ENDM

pop     MACRO   op
        MOVE.ATTRIBUTE (sp)+,op
        ENDM
```

Schreibt man nun im Code

```
        push    d0
        pop.l   a2        ,
```

so wird daraus

```
        MOVE.   d0,-(sp)
        MOVE.L  (sp)+,a2
```

Eine Makrodefinition darf nicht über Include-Datei-Grenzen hinausgehen.

In Makrorümpfen definierte Labels werden immer als lokal betrachtet, außer bei der Definition des Makros wurde die `GLOBALSYMBOLS`-Option verwendet. Ist es aus irgendwelchen Gründen erforderlich, ein einzelnes Label in einem Makro global zu machen, das ansonsten lokale Labels benutzt, so kann man es mit `LABEL` definieren, dessen Anwendung (wie bei `BIT,SFR`...) immer globale Symbole ergibt :

```
<Name>  LABEL   *
```

Da der Assembler beim Parsing einer Zeile zuerst die Makroliste und danach die Prozessorbefehle abklappert, lassen sich auch Prozessorbefehle neu definieren. Die Definition sollte dann aber vor der ersten Benutzung des Befehles durchgeführt werden, um Phasenfehler wie im folgenden Beispiel zu vermeiden:

```
        BSR     ziel

bsr     MACRO   target
        JSR     ziel
        ENDM

        BSR     ziel
```

Im ersten Pass ist bei der Assemblierung des `BSR`-Befehles das Makro noch nicht bekannt, es wird ein 4 Byte langer Befehl erzeugt. Im zweiten Pass jedoch steht die Makrodefinition sofort (aus dem ersten Pass) zur Verfügung, es wird also ein 6 Byte langer `JSR` kodiert. Infolgedessen sind alle darauf folgenden Labels um zwei zu

niedrig, bei allen weiteren Labels sind Phasenfehler die Folge, und ein weiterer Pass ist erforderlich.

Da durch die Definition eines Makros ein gleichnamiger Maschinen- oder Pseudobefehl nicht mehr zugreifbar ist, gibt es eine Hintertür, die Originalbedeutung zu erreichen: Stellt man dem Mnemonic ein `!` voran, so wird das Durchsuchen der Makroliste unterdrückt. Das kann beispielsweise nützlich sein, um Befehle in ihrer Mächtigkeit zu erweitern, z.B. die Schiebebefehle beim TLCS-90:

```
srl     macro   op,n      ; Schieben um n Stellen
        rept    n         ; n einfache Befehle
         !srl   op
        endm
        endm
```

Fortan hat der `SRL`-Befehl einen weiteren Parameter...

Expansion im Listing

Wird ein Makro im Quellcode aufgerufen, wird der durch dieses Makro definierte Quellcode, inklusiver eingesetzter Parameter, an dieser Stelle im Listing expandiert. Das kann das Listing stark aufblähen und schwerer lesbar machen. Es ist daher möglich, diese Expansion ganz oder teilweise zu unterdrücken. Generell teilt AS die in einem Makrorumpf enthaltenen Quelltext-Zeilen in drei Klassen ein:

- Darin enthaltene Makrodefinitionen, d.h. das Makro wird benutzt, um seinerseits weitere Makros zu definieren, oder es enthält `REPT/ IRP/IRPC/WHILE`-Blöcke.
- Befehle zur bedingten Assemblierung plus Zeilen, die aufgrund solcher Anweisungen *nicht* assembliert werden. Da bedingte Assemblierung von Makro-Parametern abhängig sein darf, kann diese Untermenge ebenfalls davon abhängen.
- Alle restlichen Zeilen, die nicht unter die beiden ersten Klassen fallen.

Für jedes Makro kann einzeln festgelegt werden, welche Teile im Listing auftauchen oder nicht auftauchen sollen. Vorgabewert bei der Definition eines Makros ist dabei die zuletzt mit dem Befehl `MACEXP_DFT` (3.7.3) vorgegebene Menge. Wird bei der Definition eines Makros eine der Direktiven `EXPAND/NOEXPAND`, `EXPIF/NOEXPIF`, `EXPMACRO/NOEXPMACRO` oder `EXPREST/NOEXPREST` gegeben, so wirken diese *zusätzlich* und mit höherer Priorität. Ist z.B. die Expansion global komplett ausgeschaltet

(MACEXP_DFT OFF), so bewirkt das Hinzufügen von EXPREST, daß bei der Benutzung dieses Makro nur die Zeilen im Listing angezeigt werden, die nach Auswertung bedingter Assemblierung verblieben sind und auch keine Makrodefinition selber sind.

Daraus ergibt sich, daß eine Änderung der Untermenge per MACEXP_DFT keine Auswirkungen mehr auf Makros hat, die *vor* dieser Anweisung *definiert* wurden. Im Listing führt der Abschnitt mit definierten Makros für jedes Makro auf, welche Direktiven in der Summe für dieses Makro gelten. Die in geschweiften Klammern aufgeführte Liste ist dabei soweit gekürzt, daß für jede Klasse nur die letztgültige Direktive aufgeführt wird. Ein per MACEXP_DFT gegebenes NOIF taucht dort also nicht mehr auf, falls speziell für dieses Makro die Direktive EXPIF gegeben wurde.

In Einzelfällen kann es sinnvoll sein, die für ein Makro definierten Expansionsregeln zu übersteuern, egal ob diese per MACEXP_DFT oder Direktiven gesetzt wurden. Dazu dient der Befehl MACEXP_OVR (3.7.3), der auf in der Folge *expandierte* Makros wirkt. Auch bei diesem Befehl gilt, daß damit gegebene Direktiven zusätzlich zu denen in einem Makro hinterlegten und mit höherer Priorität wirken. Ein MACEXP_OVR ohne jegliche Argumente schaltet so einen "Override" wieder ab.

3.4.2 IRP

ist die eine vereinfachte Form von Makrodefinitionen für den Fall, daß eine Befehlsfolge einmal auf mehrere Operanden angewendet werden soll und danach nicht mehr gebraucht wird. IRP benötigt als ersten Parameter ein Symbol für den Operanden, und danach eine (fast) beliebige Menge von Parametern, die nacheinander in den Befehlsblock eingesetzt werden. Um eine Menge von Registern auf den Stack zu schieben, kann man z.B. schreiben

```
IRP     op, acc,b,dpl,dph
PUSH    op
ENDM
```

was in folgendem resultiert:

```
PUSH    acc
PUSH    b
PUSH    dpl
PUSH    dph
```

Die Argumentliste darf analog zu einer Makro-Definition die folgenden Steueranweisungen (durch geschweifte Klammern als solche gekennzeichnet) enthalten:

- GLOBALSYMBOLS bzw. NOGLOBALSYMBOLS, um zu steuern, ob benutzte Labels für jeden Durchgang automatisch lokal sind oder nicht.
- EXPAND bzw. NOEXPAND
- EXPIF bzw. NOEXPIF
- EXPMACRO bzw. NOEXPMACRO
- EXPREST bzw. NOEXPREST

3.4.3 IRPC

IRPC ist eine Variante von IRP, bei der das erste Argument in den bis ENDM folgenden Zeilen nicht sukzessiv durch die weiteren Parameter, sondern durch die Zeichen eines Strings ersetzt wird. Einen String kann man z.B. also auch ganz umständlich so im Speicher ablegen:

```
irpc    char,"Hello World"
db      'CHAR'
endm
```

ACHTUNG! Wie das Beispiel schon zeigt, setzt IRPC nur das Zeichen selber ein, daß daraus ein gültiger Ausdruck entsteht (also hier durch die Hochkommas, inklusive des Details, daß hier keine automatische Umwandlung in Großbuchstaben vorgenommen wird), muß man selber sicherstellen.

3.4.4 REPT

ist die einfachste Form der Makrobenutzung. Der im Rumpf angegebene Code wird einfach sooft assembliert, wie der Integerparameter von REPT angibt. Dieser Befehl wird häufig in kleinen Schleifen anstelle einer programmierten Schleife verwendet, um den Schleifenoverhead zu sparen.

Der Vollständigkeit halber ein Beispiel:

```
REPT    3
RR      a
ENDM
```

rotiert den Akku um 3 Stellen nach rechts.

An Steueranweisungen sind die gleichen wie für IRP erlaubt.

Ist das Argument von REPT kleiner oder gleich Null, so wird überhaupt keine Expansion durchgeführt. Dies ist ein Unterschied zu früheren Versionen von AS, die hier etwas „schlampig" waren und immer mindestens eine Expansion ausführten.

3.4.5 WHILE

WHILE arbeitet analog zu REPT, allerdings tritt an die Stelle einer festen Anzahl als Argument ein boolescher Ausdruck, und der zwischen WHILE und ENDM eingeschlossene Code wird sooft assembliert, bis der Ausdruck logisch falsch wird. Im Extremfall kann dies bedeuten, daß der Code überhaupt nicht assembliert wird, falls die Bedingung bereits beim Eintritt in das Konstrukt falsch ist. Andererseits kann es natürlich auch passieren, daß die Bedingung immer wahr bleibt, und AS läuft bis an das Ende aller Tage...hier sollte man also etwas Umsicht walten lassen, d.h. im Rumpf muß eine Anweisung stehen, die die Bedingung auch beeinflußt, z.B. so:

```
cnt     set     1
sq      set     cnt*cnt
        while   sq<=1000
         dc.l    sq
cnt      set     cnt+1
sq       set     cnt*cnt
        endm
```

An Steueranweisungen sind die gleichen wie für IRP und REPT erlaubt.

Dieses Beispiel legt alle Quadratzahlen bis 1000 im Speicher ab.

Ein unschönes Detail bei WHILE ist im Augenblick leider noch, daß am Ende der Expansion eine zusätzliche Leerzeile, die im Quellrumpf nicht vorhanden war, eingefügt wird. Dies ist ein „Dreckeffekt", der auf einer Schwäche des Makroprozessors beruht und leider nicht so einfach zu beheben ist. Hoffentlich stört es nicht allzusehr....

3.4.6 EXITM

EXITM stellt einen Weg dar, um eine Makroexpansion oder einen der Befehle REPT, IRP oder WHILE vorzeitig abzubrechen. Eine solche Möglichkeit hilft zum Beispiel, umfangreichere Klammerungen mit IF-ENDIF-Sequenzen in Makros übersichtlicher zu gestalten. Sinnvollerweise ist ein EXITM aber selber auch immer bedingt, was zu einem wichtigen Detail führt: Der Stack, der über momentan offene IF- oder SWITCH-Konstrukte Buch führt, wird auf den Stand vor Beginn der Makroexpansion zurück gesetzt. Dies ist für bedingte EXITM's zwingend notwendig, da das den EXITM-Befehl in irgendeiner Form einschließende ENDIF oder ENDCASE nicht mehr erreicht wird und AS ohne einen solchen Trick eine Fehlermeldung erzeugen würde. Weiterhin

ist es für verschachtelte Makrokonstruktionen wichtig, zu beachten, daß `EXITM` immer nur das momentan innerste Konstrukt abbricht! Wer aus seiner geschachtelten Konstruktion vollständig ,,ausbrechen" will, muß auf den höheren Ebenen ebenfalls `EXITM`'s vorsehen!

3.4.7 SHIFT

`SHIFT` ist ein Mittel, um Makros mit variablen Argumentlisten abzuarbeiten: Es verwirft den ersten Parameter, so daß der zweite Parameter seinen Platz einnimmt usw. Auf diese Weise könnte man sich durch eine variable Argumentliste durcharbeiten...wenn man es richtig macht. Folgendes funktioniert zum Beispiel *nicht*...

```
pushlist  macro reg
          rept  ARGCOUNT
          push  reg
          shift
          endm
          endm
```

...weil das Makro *einmal* expandiert wird, seine Ausgabe von `REPT` aufgenommen und dann n-fach ausgeführt wird. Das erste Argument wird also n-fach gesichert...besser geht es schon so:

```
pushlist  macro reg
          if      "REG"<>""
           push    reg
           shift
           pushlist ALLARGS
          endif
          endm
```

Also eine Rekursion, in der pro Schritt die Argumentliste (`ALLARGS`) um eins verkürzt wird. Der wichtige Trick ist, daß jedes Mal eine neue Expansion gestartet wird...

Auf Plattformen, bei denen `SHIFT` bereits eine Maschineninstruktion ist, kann stattdessen `SHFT` geschrieben werden, oder man spricht explizit die Pseudoinstruktion durch einen vorangestellten Punkt (`.SHIFT` anstelle `SHIFT`) an.

3.4.8 MAXNEST

Mit `MAXNEST` kann man einstellen, wie oft ein Makro maximal rekursiv aufgerufen werden kann, bevor AS mit einer Fehlermeldung abbricht. Dies darf ein beliebiger ganzer, positiver Wert sein, wobei der Sonderwert 0 diese Sicherheitsbremse komplett abschaltet (vorsicht damit...). Der Vorgabewert für die maximale Verschachtelungstiefe ist 256; die momentane Einstellung kann aus einer gleichnamigen Variablen gelesen werden.

3.4.9 FUNCTION

`FUNCTION` ist zwar kein Makrobefehl im engeren Sinne, da hierbei aber ähnliche Mechanismen wie bei Makroersetzungen angewendet werden, soll er hier beschrieben werden.

Dieser Befehl dient dazu, neue Funktionen zu definieren, die in Formelausdrücken wie die vordefinierten Funktionen verwendet werden können. Die Definition muß in folgender Form erfolgen:

```
<Name>  FUNCTION <Arg>,..,<Arg>,<Ausdruck>
```

Die Argumente sind die Werte, die sozusagen in die Funktion ,,hineingesteckt" werden. In der Definition werden für die Argumente symbolische Namen gebraucht, damit der Assembler bei der Benutzung der Funktion weiß, an welchen Stellen die aktuellen Werte einzusetzen sind. Dies kann man an folgendem Beispiel sehen:

```
isgit   FUNCTION ch,(ch>='0')&&(ch<='9')
```

Diese Funktion überprüft, ob es sich bei dem Argument (wenn man es als Zeichen interpretiert) um eine Ziffer im momentan gültigen Zeichencode handelt (der momentane Zeichencode ist mittels `CHARSET` veränderbar, daher die vorsichtige Formulierung).

Die Argumentnamen (in diesem Falle `CH`) müssen den gleichen härteren Symbolkonventionen genügen wie Parameter bei einer Makrodefinition, d.h. die Sonderzeichen . und _ sind nicht erlaubt.

Selbst definierte Funktionen werden genauso benutzt wie eingebaute, d.h. mit einer durch Kommas getrennten, geklammerten Argumentliste:

```
IF isdigit(Zeichen)
 message "\{Zeichen} ist eine Ziffer"
ELSEIF
 message "\{Zeichen} ist keine Ziffer"
ENDIF
```

Bei dem Aufruf der Funktion werden die Argumente nur einmal berechnet und danach an allen Stellen der Formel eingesetzt, um den Rechenaufwand zu reduzieren und Seiteneffekte zu vermeiden. Bei Funktionen mit mehreren Argumenten müssen die einzelnen Argumente bei der Benutzung durch Kommata getrennt werden.

ACHTUNG! Analog wie bei Makros kann man mit der Definition von Funktionen bestehende Funktionen umdefinieren. Damit lassen sich auch wieder Phasenfehler provozieren. Solche Definitionen sollten daher auf jeden Fall vor der ersten Benutzung erfolgen!

Der Ergebnistyp kann bei Funktionen vom Typ der Argumente abhängen. So kann bei folgender Funktion

```
double  function x,x+x
```

das Ergebnis ein Integer, eine Gleitkommazahl oder sogar ein String sein, je nach Typ des Arguments!

Bei der Definition und Ansprache von Funktionen wird im case-sensitiven Modus zwischen Groß- und Kleinschreibung unterschieden, im Gegensatz zu eingebauten Funktionen!

3.5 Strukturen

Gültigkeit: alle Prozessoren

Auch in Assemblerprogrammen ergibt sich dann und wann die Notwendigkeit, analog zu Hochsprachen zusammengesetzte Datenstrukturen zu definieren. AS unterstützt sowohl die Definition als auch die Nutzung von Strukturen mit einer Reihe von Konstrukten und Anweisungen, die im folgenden erläutert werden sollen:

3.5.1 Definition

Die Definition einer Struktur wird duch den Befehl `STRUCT` eingeleitet und durch `ENDSTRUCT` abgeschlossen (schreibfaule Zeitgenossen dürfen aber auch stattdessen `STRUC` bzw. `ENDSTRUC` oder `ENDS` schreiben). Ein eventuell diesen Befehlen voranstehendes Label wird als Name der zu definierenden Struktur genommen; am Ende der Definition ist der Name optional und kann von zur Festlegung des Längennamens (s.u.) genutzt werden. Das restliche Verfahren ist simpel: Mit einem `STRUCT` wird der momentane Programmzähler gesichert und auf Null zurück gesetzt. Alle zwischen `STRUCT` und `ENDSTRUCT` definierten Labels ergeben mithin die Offsets der einzelnen Datenfelder in der Struktur. Die Reservierung des Platzes für die einzelnen Felder erfolgt mit den für den jeweils aktiven Zielprozessor zulässigen Befehlen zur Speicherplatzreservierung, also z.B. `DS.x` für die Motorolas oder `DB` & Co. für Intels. Es gelten hier auch gleichfalls die Regeln für das Aufrunden von Längen, um Alignments zu erhalten - wer also 'gepackte' Strukturen definieren will, muß eventuell ein `PADDING OFF` voranstellen. Umgekehrt lassen sich Ausrichtungen natürlich mit Befehlen wie `ALIGN` erzwingen.

Da eine solche Definition nur eine Art 'Prototypen' darstellt, können nur Befehle benutzt werden, die Speicherplatz reservieren, aber keine solchen, die Konstanten im Speicher ablegen oder Code erzeugen.

Innerhalb von Strukturen definierte Labels (also die Namen der Elemente) werden nicht direkt abgespeichert, sondern es wird ihnen der Name der Struktur vorangestellt, durch ein Trennzeichen verbunden, bei dem es sich defaultmäßig um den Unterstrich (_) handelt. Dieses Verhalten läßt sich aber durch dem `STRUCT`-Befehl mitgegebene Argumente steuern:

- `NOEXTNAMES` unterdrückt das Voranstellen des Strukturnamens. Der Programmierer ist in diesem Falle selber dafür verantworlich, daß Feldnamen nicht mehrfach verwendet werden.

- `DOTS` weist AS an, als verbindendes Zeichen einen Punkt anstelle des Unterstriches zu verwenden. Es sei jedoch ausdrücklich darauf hingewiesen, daß der Punkt bei vielen Zielprozessoren ein Sonderfunktion zur Bitadressierung hat und diese zu Problemen führen kann!

Des weiteren ist es möglich, die Verwendung des Punktes durch den Befehl

```
dottedstructs <on|off>
```

dauerhaft ein- bzw. auszuschalten.

Neben den Namen der Elemente definiert AS beim Abschluß der Definition ein weiteres Symbol mit dem Namen `LEN`, das nach dem gleichen Regeln um den Namen der Struktur erweitert wird - oder um den Label-Namen, der optional bei `ENDSTRUCT` angegeben werden kann.

Das ganze sieht dan in der Praxis z.B. so aus:

```
Rec     STRUCT
Ident   db      ?
Pad     db      ?
Pointer dd      ?
Rec     ENDSTRUCT
```

Hier würde z.B. dem Symbol `REC_LEN` der Wert 6 zugewiesen.

3.5.2 Nutzung

Ist eine Struktur einmal definiert, ist die Nutzung denkbar einfach und ähnlich wie ein Makro: ein einfaches

```
thisrec Rec
```

reserviert Speicher in der Menge, wie er von der Struktur belegt wird, und definiert gleichzeitig für jedes Element der Struktur ein passendes Symbol mit dessen Adresse, in diesem Falle also `THISREC_IDENT, THISREC_PAD` und `THISREC_POINTER`. Das Label darf bei dem Aufruf einer Struktur naturgemäß nicht fehlen; wenn doch, gibt's eine Fehlermeldung.

Über zusätzliche Argumente ist es möglich, nicht nur Speicher für eine einzelne Struktur, sondern ein ganzes Feld davon zu reservieren. Die (bis zu drei) Dimensionen werden über in eckige Klammern gesetzte Argumente definiert:

```
thisarray Rec [10],[2]
```

In diesem Beispiel wird Platz für $2 * 10 = 20$ Strukturen reserviert, und für jede Einzelstruktur werden Symbole erzeugt, die die Indizes im Namen enthalten.

3.5.3 geschachtelte Strukturen

Es ist ohne weiteres erlaubt, eine bereits definierte Struktur in einer anderen Struktur aufzurufen. Das dabei ablaufende Verfahren ist eine Kombination aus den beiden vorigen Punkten: Elemente der Substruktur werden definiert, mit dem Namen dieser Instanz vorangestellt, und vor diese zusammengesetzten Namen wird wieder der Name der Struktur bzw. später bei einer Benutzung gesetzt. Das sieht dann z.B. so aus:

```
TreeRec struct
left    dd         ?
right   dd         ?
data    Rec
TreeRec endstruct
```

Ebenso ist es erlaubt, eine Struktur direkt in einer anderen Struktur zu definieren:

```
TreeRec struct
left    dd         ?
right   dd         ?
TreeData struct
name      db         32 dup(?)
id        dw         ?
TreeData endstruct
TreeRec endstruct
```

3.5.4 Unions

Eine Union ist eine Sonderform einer Struktur, bei der die einzelnen Elemente nicht hintereinander, sondern *übereinander* liegen, d.h. alle Elemente liegen an Startadresse 0 innerhalb der Struktur und belegen den gleichen Speicherplatz. Naturgemäß tut so eine Definition nicht mehr, als einer Reihe von Symbolen den Wert Null zuzuweisen, sie kann aber sinnvoll sein, um programmtechnisch die Überlappung der Elemente zu verdeutlichen und den Code so etwas 'lesbarer' zu gestalten. Die Größe einer Struktur ist das Maximum der Größen aller Elemente.

3.5.5 Namenlose Strukturen

Der Name einer Struktur oder Union ist optional, allerdings nur, wenn diese Teil einer anderen, nicht namenlosen Struktur ist. Elemente dieser Struktur werden dann Teil der 'nächsthöheren' benamten Struktur:

```
TreeRec struct
left    dd         ?
right   dd         ?
        struct
name      db         32 dup(?)
id        dw         ?
        endstruct
TreeRec endstruct
```

erzeugt also die Symbole `TREEREC_NAME` und `TREEREC_ID`.

Des weiteren wird für namenlose Strukturen oder Unions kein Symbol mit deren Länge angelegt.

3.5.6 Strukturen und Sektionen

Im Verlaufe der Definition oder der Nutzung von Strukturen definierte Symbole werden genauso behandelt wie normale Symbole, d.h. bei der Nutzung innerhalb einer Sektion werden diese Symbole als lokal zu dieser Sektion definiert. Analoges gilt aber auch für die Strukturen selber, d.h. eine innerhalb einer Sektion definierte Struktur kann nicht auërhalb der Sektion benutzt werden.

3.5.7 Strukturen und Makros

Will man Strukturen über Makros instanziieren, so muß man die `GLOBALSYMBOLS`-Option bei der Definition des Makros benutzen, damit die darüber erzeugten Symbole auch außerhalb des Makros verwendbar sind. Eine Reihe von Strukturen kann man z.B. so anlegen:

```
        irp     name,{GLOBALSYMBOLS},rec1,rec2,rec3
name    Rec
        endm
```

3.6 bedingte Assemblierung

Gültigkeit: alle Prozessoren

Der Assembler unterstützt die bedingte Assemblierung mit Hilfe der Konstrukte `IF`... sowie `SWITCH`... . Diese Befehle wirken zur Assemblierzeit, indem entsprechend

der Bedingung Teile übersetzt oder übersprungen werden. Diese Befehle sind also *nicht* mit den IF-Statements höherer Programmiersprachen zu vergleichen (obwohl es sehr verlockend wäre, den Assembler um die Strukturierungsbefehle höherer Sprachen zu erweitern...).

Die folgenden Konstrukte dürfen beliebig (bis zum Speicherüberlauf) geschachtelt werden.

3.6.1 IF / ELSEIF / ENDIF

`IF` ist das gebräuchlichere und allgemeiner verwendbare Konstrukt. Die allgemeine Form eines `IF`-Befehles lautet folgendermaßen:

```
IF       <Ausdruck 1>
 <Block 1>
ELSEIF   <Ausdruck 2>
 <Block 2>
(evtl. weitere ELSEIFs)
ELSEIF
 <Block n>
ENDIF
```

`IF` dient als Einleitung und wertet den ersten Ausdruck aus und assembliert Block 1, falls der Ausdruck wahr (d.h. ungleich 0) ist. Alle weiteren `ELSEIF`-Teile werden dann ignoriert. Falls der Ausdruck aber nicht wahr ist, wird Block 1 übersprungen und Ausdruck 2 ausgewertet. Sollte dieser nun wahr sein, wird Block 2 assembliert. Die Zahl der `ELSEIF`-Teile ist variabel und ergibt eine `IF-THEN-ELSE`-Leiter beliebiger Länge. Der dem letzten `ELSEIF` (ohne Parameter) zugeordnete Block wird nur assembliert, falls alle vorigen Ausdrücke falsch ergaben und bildet sozusagen einen „Default-Zweig". Wichtig ist, daß von den Blöcken immer nur *einer* assembliert wird, und zwar der erste, dessen zugeordnetes `IF/ELSEIF` einen wahren Ausdruck hatte.

Die `ELSEIF`-Teile sind optional, d.h. auf `IF` darf auch direkt `ENDIF` folgen, ein parameterloses `ELSEIF` bildet aber immer den letzten Zweig. Ein `ELSEIF` bezieht sich immer auf das letzte, noch nicht abgeschlossene `IF`.

Neben `IF` sind noch folgende weitere bedingte Befehle definiert:

- `IFDEF` <Ausdruck> : wahr, falls der Ausdruck keine Symbole enthält, die im bis zu diesem Punkt assemblierten Quellcode (noch) nicht definiert wurden. ACHTUNG! Bei Vorwärtsdeklarationen liefert diese Abfrage in allen Durchläufen ein negatives Ergebnis!

- `IFNDEF` <Ausdruck> : Umkehrung zu `IFDEF`
- `IFSYMEXIST` <Symbol> : wahr, falls dieses Symbol in der Symboltabelle existiert. ACHTUNG! Bei Vorwärtsdeklarationen liefert diese Abfrage im ersten Durchlauf ein negatives Ergebnis, aber ein positives in allen weiteren!
- `IFNSYMEXIST` <Symbol> : Umkehrung zu `IFSYMEXIST`
- `IFUSED` <Symbol> : wahr, falls das Symbol bisher mindestens einmal benutzt wurde. Siehe Abschnitt 2.7 für Ausnahmen davon.
- `IFNUSED` <Symbol> : Umkehrung zu `IFUSED`
- `IFEXIST` <Name: > : wahr, falls die angegebene Datei existiert. Für Schreibweise und Suchpfade gelten gleiche Regeln wie beim `INCLUDE`-Befehl (siehe Abschnitt 3.9.2).
- `IFNEXIST` <Name: > : Umkehrung zu `IFEXIST`
- `IFB` <Arg-Liste> : wahr, falls alle Argumente der Parameterliste leer sind.
- `IFNB` <Arg-Liste> : Umkehrung zu IFB.

Anstelle von `ELSEIF` darf auch `ELSE` geschrieben werden, weil das wohl alle so gewohnt sind....

Zu jeder `IF...`-Anweisung gehört ein entsprechendes `ENDIF`, 'offene' Konstrukte führen zu einer Fehlermeldung am Ende des Assemblierungslaufes. Die Zuordnung, welches `ENDIF` AS mit welchem `IF...` 'gepaart' hat, läßt sich im Listing erkennen: dort wird die Zeilennummer des entsprechenden `IF`s angezeigt.

3.6.2 SWITCH / CASE / ELSECASE / ENDCASE

`SWITCH` ist ein Spezialfall von `IF` und für den Fall gedacht, daß ein Ausdruck mit einer Reihe von Werten verglichen werden soll. Dies ist natürlich auch mit `IF` und einer Reihe von `ELSEIF`s machbar, die folgende Form

```
SWITCH   <Ausdruck>
...
CASE     <Wert 1>
...
<Block 1>
...
```

```
CASE <Wert 2>
...
<Block 2>
...
(weitere CASE-Konstrukte)
...
CASE <Wert n-1>
...
<Block n-1>
...
ELSECASE
...
<Block n>
...
ENDCASE
```

bietet aber den Vorteil, daß der zu prüfende Ausdruck nur einmal hingeschrieben und berechnet werden muß, er ist also weniger fehleranfällig und etwas schneller als eine `IF`-Kette, dafür natürlich auch nicht so flexibel.

Es ist möglich, bei den `CASE`-Anweisungen mehrere, durch Kommata getrennte Werte anzugeben, um den entsprechenden Block in mehreren Fällen assemblieren zu lassen. Der `ELSECASE`-Zweig dient wiederum als ,,Auffangstelle" für den Fall, daß keine der `CASE`-Bedingungen greift. Fehlt er und fallen alle Prüfungen negativ aus, so gibt AS eine Warnung aus.

Auch wenn die Wertelisten der `CASE`-Teile sich überlappen, so wird immer nur *ein* Zweig ausgeführt, und zwar bei Mehrdeutigkeiten der erste.

`SWITCH` dient nur der Einleitung des ganzen Konstruktes; zwischen ihm und dem ersten `CASE` darf beliebiger Code stehen (andere `IF`s dürfen aber nicht offen bleiben!), im Sinne eines durchschaubaren Codes sollte davon aber kein Gebrauch gemacht werden.

Ist `SWITCH` auf dem gewählten Target ein Maschinenbefehl, so leitet man das Konstrukt stattdessen mit `SELECT` ein, oder spricht den Pseudobefehl explizit durch einen vorangestellten Punk an (`.SWITCH` anstelle von `SWITCH`).

Ähnlich wie bei `IF...`-Konstrukten, muß es für jedes `SWITCH` genau ein `ENDCASE` geben. Analog zu `ENDIF` wird bei `ENDCASE` im Listing die Zeilennummer des korrespondierenden `SWITCH` angezeigt.

3.7 Listing-Steuerung

Gültigkeit: alle Prozessoren

3.7.1 PAGE, PAGESIZE

Mit `PAGE` kann man AS die Dimensionen des Papiers, auf dem das Listing ausgedruckt werden soll, mitteilen. Als erster Parameter wird dabei die Anzahl von Zeilen angegeben, nach der AS automatisch einen Zeilenvorschub ausgeben soll. Zu berücksichtigen ist allerdings, daß bei dieser Angabe die Kopfzeilen inklusive einer evtl. mit `TITLE` spezifizierten Zeile nicht mitgerechnet werden. Der Minimalwert für die Zeilenzahl ist 5, der Maximalwert 255. Eine Angabe von 0 führt dazu, daß AS überhaupt keine automatischen Seitenvorschübe ausführt, sondern nur noch solche, die explizit durch `NEWPAGE`-Befehle oder implizit am Ende des Listings (z.B. vor der Symboltabelle) von AS ausgelöst wurden.

Die Angabe der Breite des Listings in Zeichen kann als optionaler zweiter Parameter erfolgen und erfüllt zwei Zwecke: Zum einen läuft der Zeilenzähler von AS korrekt weiter, wenn eine Quell-Zeile über mehrere Listing-Zeilen geht, zum anderen gibt es Drucker (wie z.B. Laserdrucker), die beim Überschreiten des rechten Randes nicht automatisch in eine neue Zeile umbrechen, sondern den Rest einfach „verschlucken". Aus diesem Grund führt AS auch den Zeilenumbruch selbstständig durch, d.h. zu lange Zeilen werden in Bruchstücke zerlegt, die eine Länge kleiner oder gleich der eingestellten Länge haben. In Zusammenhang mit Druckern, die einen automatischen Zeilenumbruch besitzen, kann das aber zu doppelten Zeilenvorschüben führen, wenn man als Breite exakt die Zeilenbreite des Druckers angibt. Die Lösung in einem solchen Fall ist, als Zeilenbreite ein Zeichen weniger anzugeben. Die eingestellte Zeilenbreite darf zwischen 5 und 255 Zeichen liegen; analog zur Seitenlänge bedeutet ein Wert von 0, daß AS keine Splittung der Listing-Zeilen vornehmen soll; eine Berücksichtigung von zu langen Zeilen im Listing beim Seitenumbruch kann dann natürlich auch nicht mehr erfolgen.

Die Defaulteinstellung für die Seitenlänge ist 60 Zeilen, für die Zeilenbreite 0; letztere Wert wird auch angenommen, wenn `PAGE` nur mit einem Argument aufgerufen wird.

Falls `PAGE` auf dem gewählten Target bereits ein Maschinenbefehl ist, benutzt man stattdessen `PAGESIZE`. Alternativ ist es immer möglich, durch Voranstellen eines Punktes (`.PAGE` anstelle `PAGE`) explizit den Pseudobefehl anzusprechen.

ACHTUNG! AS hat keine Möglichkeit, zu überprüfen, ob die eingestellte Listing-Länge und Breite mit der Wirklichkeit übereinstimmen!

3.7.2 NEWPAGE

`NEWPAGE` kann dazu benutzt werden, einen Seitenvorschub zu erzwingen, obwohl die Seite noch gar nicht voll ist. Dies kann z.B. sinnvoll sein, um logisch voneinander getrennte Teile im Assemblerprogramm auch seitenmäßig zu trennen. Der programminterne Zeilenzähler wird zurück gesetzt, der Seitenzähler um Eins heraufgezählt. Der optionale Parameter steht in Zusammenhang mit einer hierarchischen Seitennumerierung, die AS bis zu einer Kapiteltiefe von 4 unterstützt. 0 bedeutet dabei immer die tiefste Kapitelebene, der Maximalwert kann sich während des Laufes verändern, wenn das auch verwirrend wirken kann, wie folgendes Beispiel zeigt:

Seite 1,	Angabe `NEWPAGE 0`	→ Seite 2
Seite 2,	Angabe `NEWPAGE 1`	→ Seite 2.1
Seite 2.1,	Angabe `NEWPAGE 1`	→ Seite 3.1
Seite 3.1,	Angabe `NEWPAGE 0`	→ Seite 3.2
Seite 3.2,	Angabe `NEWPAGE 2`	→ Seite 4.1.1

Je nach momentan vorhandener Kapiteltiefe kann `NEWPAGE <Nummer>` also an verschiedenen Stellen eine Erhöhung bedeuten. Ein automatischer Seitenvorschub wegen Zeilenüberlauf oder ein fehlender Parameter ist gleichbedeutend mit `NEWPAGE 0`. Am Ende des Listings wird vor Ausgabe der Symboltabelle ein implizites `NEWPAGE <bish. Maximum>` durchgeführt, um sozusagen ein neues Hauptkapitel zu beginnen.

3.7.3 MACEXP_DFT und MACEXP_OVR

Ist ein Makro einmal ausgestestet und 'fertig', möchte man es bei Benutzung vielleicht gar nicht mehr im Listing sehen. Wie im Abschnitt über Makros (3.4.1) erläutert, kann man bei der Definition eines Makros über Zusatzargumente steuern, ob und wenn ja welche Teile des Makro-Rumpfes im Listing expandiert werden. Für den Fall, daß eine ganze Reihe von Makros in Folge definiert werden, muß man dies jedoch nicht für jedes Makro einzeln festlegen. Der Befehl `MACEXP_DFT` setzt für alle im folgenden definerten Makros, welche Teile ihres Rumpfes expandiert werden sollen:

- `ON` bzw. `OFF` schalten die Expansion komplett ein bzw. aus.
- Mit den Argumenten `IF` bzw. `NOIF` werden Befehle im Rumpf zur bedingten Assemblierung und derentwegen nicht assemblierte Code-Teile aus- bzw. eingeblendet.

- Makro-Definitionen (dazu zählen auch `REPT`, `WHILE` und `IRP(C)`) können über die Argumente `MACRO` bzw. `NOMACRO` ein- und ausgeblendet werden.
- Mit den Argumenten `REST` bzw. `NOREST` können die Zeilen ein- und ausgeblendet werden, die nicht in die ersten beiden Klassen fallen.

Default ist `ON`, d.h. im folgenden definierte Makros werden komplett expandiert, außer natürlich bei den einzelnen Makros wurde dies durch Direktiven übersteuert. Weiterhin wirken angegebene Schalte relativ zur aktuellen Einstellung: ist z.B. initial alles eingeschaltet, sorgt ein

```
MACEXP_DFT noif,nomacro
```

dafür, daß nur noch das gelistet wird, was weder eine Makrodefinition ist noch per bedingter Assemblierung ausgeschlossen wurde.

Mit diesem Befehl und den pro Makro setzbaren Direktiven läßt sich für jedes einzelne Makro genau festlegen, welche Teile bei einer Expansion im Listing erscheinen sollen und welche nicht. Es kann jedoch in der Praxis auch vorkommen, daß man ein bestimmtes Makro an einzelnen Stellen im Quellcode expandiert haben möchte, an anderen jedoch nicht. Dies ist mit dem Befehl `MACEXP_OVR` möglich: er akzeptiert die gleichen Argumente, diese wirken jedoch als Overrides für alle im folgenden *expandierten* Makros, im Gegensatz zu `MACEXP_DFT`, das auf alle im folgenden *definierten* Makros wirkt. Ist zum Beispiel für ein Makro festgelegt worden, daß weder Makrodefinitionen noch per bedingte Assemblierung ausgeschlossene Teile gelistet werden sollen, so schaltet ein

```
MACEXP_OVR MACRO
```

für folgende Expansionen das Listen von Makrodefinitionen wieder ein, während ein

```
MACEXP_OVR ON
```

wieder alles ins Listing expandiert. `MACEXP_OVR` ohne Argumente schaltet wiederum sämtliche Overrides aus, Makros verhalten sich bei der Expansion wieder so, wie zum Zeitpunkt ihrer Definition festgelegt.

Beide Befehle wirken ebenfalls auf andere Makro-artige Konstrukte (`REPT, IRP, IRPC WHILE`), da diese aber einmalig ,,in-place" expandiert werden, verschwimmt der funktionale Unterschied zwischen den beiden Befehlen - im Zweifelsfalle hat aber der per `MACEXP_OVR` gesetzte Override eine höhere Priorität.

Die momentane mit `MACEXP_DFT` gesetzte Einstellung läßt sich aus dem Symbol `MACEXP` auslesen. Anstelle von `MACEXP_DFT` darf auch einfach `MACEXP` geschrieben werden, davon sollte aber in neuen Programmen kein Gebrauch mehr gemacht werden.

3.7.4 LISTING

Mit diesem Befehl kann das Listing komplett ein- und ausgeschaltet werden. Nach einem

```
LISTING off
```

wird *überhaupt* nichts mehr im Listing ausgegeben. Diese Anweisung macht Sinn für erprobte Codeteile oder Include-Dateien, um den Papierverbrauch nicht ins Unermeßliche zu steigern. **ACHTUNG!** Wer später das Gegenstück vergißt, bekommt auch keine Symboltabelle mehr zu sehen! Zusätzlich zu `ON` und `OFF` akzeptiert `LISTING` auch `NOSKIPPED` und `PURECODE` als Argument. Mit der `NOSKIPPED`-Einstellung werden aufgrund bedingter Assemblierung nicht assemblierte Teile nicht im Listing aufgeführt, während `PURECODE` - wie der Name schon erahnen läßt - auch die `IF`-Konstrukte selber nicht mehr im Listing aufführt. Diese Einstellungen sind nützlich, wenn man Makros, die anhand von Parametern verschiedene Aktionen ausführen, benutzt, und im Listing nur noch die jeweils benutzten Teile sehen möchte.

Die momentane Einstellung läßt sich aus dem Symbol `LISTING` (0=`OFF`, 1=`ON`, 2=`NOSKIPPED`, 3=`PURECODE`) auslesen. Der Startwert kann über die gleichnamige Kommandozeilen-Option gesetzt werden.

3.7.5 PRTINIT und PRTEXIT

Bei der Listingausgabe auf Druckern ist es oftmals sinnvoll, den Drucker in eine andere Betriebsart (z.B. Schmalschrift) umzuschalten und am Ende des Listings diese Betriebsart wieder zu deaktivieren. Mit diesen Befehlen kann die Ausgabe dieser Steuerfolgen automatisiert werden, indem man mit

```
PRTINIT <String>
```

die Zeichenfolge angibt, die vor Beginn des Listings an das Ausgabegerät geschickt werden soll und mit

```
PRTEXIT <String>
```

analog den Deinitialisierungsstring. In beiden Fällen muß `<String>` ein Stringausdruck sein. Die Syntaxregeln für Stringkonstanten ermöglichen es, ohne Verrenkungen Steuerzeichen in den String einzubauen.

Bei der Ausgabe dieser Strings unterscheidet der Assembler **nicht**, wohin das Listing geschickt wird, d.h. Druckersteuerzeichen werden rücksichtslos auch auf den Bildschirm geschickt!

Beispiel :

Bei Epson-Druckern ist es sinnvoll, für die breiten Listings in den Kompreßdruck zu schalten. Die beiden Zeilen

```
PRTINIT "\15"
PRTEXIT "\18"
```

sorgen dafür, daß der Kompreßdruck ein- und nach dem Druck wieder ausgeschaltet wird.

3.7.6 TITLE

Normalerweise versieht der Assembler bereits jede Listingseite mit einer Titelzeile, die Quelldatei, Datum und Uhrzeit enthält. Mit diesem Befehl kann man den Seitenkopf um eine beliebige zusätzliche Zeile erweitern. Der anzugebende String ist dabei ein beliebiger Stringausdruck.

Beispiel:

Bei dem bereits oben angesprochenen Epson-Drucker soll eine Titelzeile im Breitdruck ausgegeben werden, wozu vorher der Kompreßmodus abgeschaltet werden muß:

```
TITLE   "\18\14Breiter Titel\15"
```

(Epson-Drucker schalten den Breitdruck automatisch am Zeilenende aus.)

3.7.7 RADIX

`RADIX` mit einem numerischen Argument zwischen 2 und 36 legt das Default-Zahlensystem für Integer-Konstanten fest, d.h. das Zahlensystem, das angenommen wird, wenn man nichts ausdrücklich anderes angegeben hat. Defaultmäßig ist dies 10, und bei der Veränderung dieses Wertes sind einige Fallstricke zu beachten, die in Abschnitt 2.9.1 beschrieben sind.

Unabhängig von der momentanen Einstellung ist das Argument von `RADIX` *immer dezimal*; weiterhin dürfen keine symbolischen oder Formelausdrücke verwendet werden, sondern nur einfache Zahlenkonstanten!

Eine `RADIX`-Anweisung übersteuert einen eventuell gesetzten `-radix`-Schalter auf der Kommandozeile.

Auf dem IM61x0 stehen die Anweisungen `DECIMAL` und `OCTAL` zur Verfügung, dies sind Kurzschreibweisen für `RADIX 10` beziehungsweise `RADIX 8`.

3.7.8 OUTRADIX

`OUTRADIX` is gewissermaßen das Gegenstück zu `RADIX`: Mit ihm kann man festlegen, in welchem Zahlensystem berechnete Integer-Ausdrücke in Strings eingesetzt werden sollen, wenn man `\{...}`-Konstrukte in Stringkonstanten verwendet (siehe Abschnitt 2.9.3). Als Argument sind wieder Werte zwischen 2 und 36 erlaubt; der Default ist 16.

3.8 lokale Symbole

Gültigkeit: alle Prozessoren

Bei den lokalen Labels und den dazu eingeführten Sektionen handelt es sich um eine grundlegend neue Funktion, die mit Version 1.39 eingeführt wird. Da dieser Teil sozusagen „1.0" ist, ist er sicherlich noch nicht der Weisheit letzter Schluß. Anregungen und (konstruktive) Kritik sind daher besonders erwünscht. Insbesondere habe ich die Verwendung von Sektionen hier so dargestellt, wie ich sie mir vorstelle. Es kann dadurch passiert sein, daß die Realität nicht ganz meinem Modell im Kopf entspricht. Für den Fall von Diskrepanzen verspreche ich, daß die Realität der Dokumentation angepaßt wird, und nicht umgekehrt, wie es bei größeren Firmen schon einmal vorgekommen sein soll...

AS erzeugt keinen linkfähigen Code (und wird es wohl auch nicht in näherer Zukunft tun `:-(`). Diese Tatsache zwingt dazu, ein Programm immer im ganzen zu übersetzen. Dieser Technik gegenüber hätte eine Aufteilung in Linker-Module einige Vorteile:

- kürzere Übersetzungszeiten, da lediglich die geänderten Module neu übersetzt werden müssen;
- die Möglichkeit, durch Definition öffentlicher und privater Symbole definierte Schnittstellen zwischen den Modulen festzulegen;

- Durch die geringere Länge der einzelnen Module reduziert sich die Anzahl der Symbole im einzelnen Modul, so daß kürzere und trotzdem eindeutige Symbolnamen benutzt werden können.

Insbesondere der letzte Punkt hat mich persönlich immer etwas gestört: War ein Label-Name einmal am Anfang eines 2000 Zeilen langen Programms benutzt, so durfte er nirgendwo wieder verwendet werden — auch nicht am anderen Ende des Quelltextes, wo Routinen mit ganz anderem Kontext standen. Ich war dadurch gezwungen, zusammengesetzte Namen der Form

```
<Unterprogrammname>_<Symbolname>
```

zu verwenden, die dann Längen zwischen 15 und 25 Zeichen hatten und das Programm unübersichtlich machten. Das im folgenden eingehender beschriebene Sektionen-Konzept sollte zumindest den beiden letzten genannten Punkten abhelfen. Es ist vollständig optional: Wollen Sie keine Sektionen verwenden, so lassen Sie es einfach bleiben und arbeiten weiter wie unter den älteren AS-Versionen.

3.8.1 Grunddefinition (SECTION/ENDSECTION)

Eine *Sektion* stellt einen durch spezielle Befehle eingerahmten Teil des Assembler-Programms dar und hat einen vom Programmierer festlegbaren, eindeutigen Namen:

```
...
<anderer Code>
...
SECTION <Sektionsname>
...
<Code in der Sektion>
...
ENDSECTION [Sektionsname]
...
<anderer Code>
...
```

Der Name für eine Sektion muß den Konventionen für einen Symbolnamen entsprechen; da AS Sektions-und Symbolnamen in getrennten Tabellen speichert, darf ein Name sowohl für ein Symbol als auch eine Sektion verwendet werden. Sektionsnamen müssen in dem Sinne eindeutig sein, daß auf einer Ebene nicht zwei Sektionen den gleichen Namen haben dürfen (was es mit den „Ebenen" auf sich hat, erläutere ich im nächsten Abschnitt). Das Argument zu `ENDSECTION` ist optional, es darf auch

weggelassen werden; Falls es weggelassen wird, zeigt AS den Namen der Sektion an, der er das `ENDSECTION` zugeordnet hat. Code in einer Sektion wird von AS genauso behandelt wie außerhalb, lediglich mit drei entscheidenden Unterschieden:

- Innerhalb der Sektion definierte Symbole (z.B. Labels, `EQU`s...) werden mit einer von AS intern vergebenen, der Sektion zugeordneten Nummer versehen. Diese Symbole sind von Code außerhalb der Sektion nicht ansprechbar (das läßt sich natürlich durch Pseudobefehle variieren, aber dazu später mehr).
- Durch das zusätzliche Attribut kann ein Symbolname sowohl außerhalb der Sektion als auch innerhalb definiert werden, das Attribut erlaubt also, Symbolnamen mehrfach zu benutzen, ohne daß AS Protest anmeldet.
- Falls ein Symbol sowohl außerhalb als auch innerhalb definiert ist, wird innerhalb der Sektion das ,,lokale" verwendet, d.h. AS sucht in der Symboltabelle zuerst nach einem Symbol des gewünschten Namens, das auch gleichzeitig der Sektion zugeordnet wurde. Erst danach wird nach einem globalen Symbol dieses Namens gefahndet.

Mit diesem Mechanismus kann man z.B. den Code in Module aufteilen, wie man es mit einem Linker getan hätte. Eine feinere Aufteilung wäre dagegen, alle Routinen in getrennte Sektionen zu verpacken. Je nach Länge der Routinen können die nur intern benötigten Symbole dann sehr kurze Namen haben.

Defaultmäßig unterscheidet AS Groß-und Kleinschreibung in Sektions- namen nicht; schaltet man jedoch in den case-sensitiven Modus um, so wird die Schreibweise genauso wie bei Symbolnamen berücksichtigt.

Die bisher beschriebene Aufteilung würde in etwa der Sprache C entsprechen, in der alle Funktionen auf gleicher Ebene nebeneinander stehen. Da mein ,,hochsprachliches" Vorbild aber Pascal ist, bin ich noch einen Schritt weiter gegangen:

3.8.2 Verschachtelung und Sichtbarkeitsregeln

Es ist erlaubt, in einer Sektion weitere Sektionen zu definieren, analog zu der Möglichkeit in Pascal, in einer Prozedur/Funktion weitere Prozeduren zu definieren. Dies zeigt folgendes Beispiel:

```
sym     EQU         0

        SECTION     ModulA
```

```
         SECTION    ProcA1
sym       EQU        5
         ENDSECTION ProcA1
         SECTION    ProcA2
sym       EQU        10
         ENDSECTION ProcA2
        ENDSECTION ModulA

        SECTION    ModulB
sym      EQU        15
         SECTION    ProcB
         ENDSECTION ProcB
        ENDSECTION ModulB
```

Bei der Suche nach einem Symbol sucht AS zuerst ein Symbol, das der aktuellen Sektion zugeordnet ist, und geht danach die ganze ,,Liste" der Vatersektionen durch, bis er bei den globalen Symbolen angekommen ist. Im Beispiel sehen die Sektionen die in Tabelle 3.2 angegebenen Werte für das Symbol `sym`. Diese Regel kann man

Sektion	Wert	aus Sektion...
Global	0	Global
`ModulA`	0	Global
`ProcA1`	5	`ProcA1`
`ProcA2`	10	`ProcA2`
`ModulB`	15	`ModulB`
`ProcB`	15	`ModulB`

Tabelle 3.2: Für die einzelnen Sektionen gültigen Werte

durchbrechen, indem man explizit an den Symbolnamen die Sektion anhängt, aus der man das Symbol holen will, und zwar in eckigen Klammern am Ende des Symbolnamens:

```
        move.l  #sym[ModulB],d0
```

Es dürfen dabei nur Sektionsnamen verwendet werden, die eine Obersektion zur aktuellen Sektion darstellen. Als Sonderwert sind die Namen `PARENT0..PARENT9` erlaubt, mit denen man die n-ten ,,Vatersektionen" relativ zur momentanen Sektion ansprechen kann; `PARENT0` entspricht also der momentanen Sektion selber, `PARENT1` der direkt übergeordneten usw. Anstelle `PARENT1` kann man auch kurz nur `PARENT` schreiben. Läßt man dagegen den Platz zwischen den Klammern komplett frei, also etwa so

```
        move.l  #sym[],d0 ,
```

so erreicht man das globale Symbol. **ACHTUNG!** Wenn man explizit ein Symbol aus einer Sektion anspricht, so wird auch nur noch bei den Symbolen dieser Sektion gesucht, der Sektionsbaum wird nicht mehr bis nach oben durchgegangen!

Analog zu Pascal ist es erlaubt, daß verschiedene Sektionen Untersektionen gleichen Namens haben dürfen, das Prinzip der Lokalität verhindert hier Irritationen. M.E. sollte man davon aber trotzdem sparsamen Gebrauch machen, da in Symbol- und Querverweisliste Symbole zwar mit der Sektion, in der sie definiert wurden, gekennzeichnet werden, aber nicht mit der über dieser Sektion evtl. liegenden ,,Sektionshierarchie" (das hätte einfach den Platz in der Zeile gesprengt); Unterscheidungen sind dadurch nicht erkennbar.

Da ein `SECTION`-Befehl von selber kein Label definiert, besteht hier ein wichtiger Unterschied zu Pascal: Eine Pascal-Prozedur kann ihre Unterprozeduren/funktionen automatisch ,,sehen", unter AS muß man noch einen Einsprungpunkt extra definieren. Das kann man z.B. mit folgendem Makro-Pärchen tun:

```
proc    MACRO   name
        SECTION name
name    LABEL   $
        ENDM

endp    MACRO   name
        ENDSECTION name
        ENDM
```

Diese Beispiel zeigt gleichzeitig, daß die Lokalität von Labels in Makros nicht von den Sektionen beeinflußt wird, deshalb der Trick mit dem `LABEL`-Befehl.

Natürlich ist mit dieser Definition das Problen noch nicht ganz gelöst, bisher ist das Einsprung-Label ja noch lokal und von außen nicht zu erreichen. Wer nun meint, man hätte das Label einfach nur vor der SECTION-Anweisung plazieren müssen, sei jetzt bitte ruhig, denn er verdirbt mir den Übergang auf das nächste Thema:

3.8.3 PUBLIC und GLOBAL

Die `PUBLIC`-Anweisung erlaubt es, die Zugehörigkeit eines Symbols zu einer bestimmten Sektion zu verändern. Es ist möglich, mit einem `PUBLIC`-Befehl mehrere Symbole zu bearbeiten, ohne Beschränkung der Allgemeinheit will ich aber ein Beispiel mit nur einer Variable verwenden: Im einfachsten Falle erklärt man ein Symbol als vollständig global, d.h. es ist von allen Stellen des Programms ansprechbar:

```
        PUBLIC  <Name>
```

Da ein Symbol bei seiner Definition endgültig in der Symboltabelle einsortiert wird, muß diese Anweisung **vor** der Definition des Symbols erfolgen. Alle `PUBLIC`s werden von AS in einer Liste vermerkt und bei ihrer Definition aus dieser Liste wieder entfernt. Bei Beendigung einer Sektion gibt AS Fehlermeldungen für alle nicht aufgelösten ,,Vorwärtsreferenzen" aus.

Angesichts des hierarchischen Sektionenkonzepts erscheint die Methode, ein Symbol als vollständig global zu definieren, reichlich brachial. Es geht aber auch etwas differenzierter, indem man zusätzlich einen Sektionsnamen angibt:

```
        PUBLIC  <Name>:<Sektion>
```

Damit wird das Symbol der genannten Sektion zugeordnet und damit auch allen ihren Untersektionen zugänglich (es sei denn, diese definieren wiederum ein Symbol gleichen Namens, das dann das ,,globalere" übersteuert). Naturgemäß protestiert AS, falls mehrere Untersektionen ein Symbol gleichen Namens auf die gleiche Ebene exportieren wollen. Als Spezialwert für <`Sektion`> sind die im vorigen Abschnitt genannten `PARENTx`-Namen zugelassen, um das Symbol genau n Ebenen hinaufzuexportieren. Es sind als Sektionen nur der momentanen Sektion übergeordnete Sektionen zugelassen, also keine, die im Baum aller Sektionen in einem anderen Zweig stehen. Sollten dabei mehrere Sektionen den gleichen Namen haben (dies ist legal), so wird die tiefste gewählt.

Mit diesem Werkzeug kann das obige Prozedurmakro nun Sinn ergeben:

```
proc    MACRO   name
        SECTION name
        PUBLIC  name:PARENT
name    LABEL   $
        ENDM
```

Diese Einstellung entspricht dem Modell von Pascal, in der eine Unterprozedur auch nur von ihrem ,,Vater" gesehen werden kann, jedoch nicht vom ,,Großvater".

Falls mehrere Untersektionen versuchen, ein Symbol gleichen Namens in die gleiche Obersektion zu exportieren, meckert AS über doppelt definierte Symbole, was an sich ja korrekt ist. War das gewollt, so muß man die Symbole in irgendeiner Weise ,,qualifizieren", damit sie voneinander unterschieden werden können. Dies ist mit der `GLOBAL`-Anweisung möglich. Die Syntax von `GLOBAL` ist der von `PUBLIC` identisch, das Symbol bleibt aber lokal, anstatt einer höheren Sektion zugeordnet zu werden. Stattdessen wird ein weiteres Symbol gleichen Werts erzeugt, dem jedoch der Untersektionsname mit einem Unterstrich vorangestellt wird, und nur dieses Symbol wird

der Sektionsangabe entsprechend öffentlich gemacht. Definieren z.B. zwei Sektionen `A` und `B` ein Symbol `SYM` und exportieren es mit `GLOBAL` zu ihrer Vatersektion, so werden dort die Symbole unter den Namen `A_SYM` und `B_SYM` eingeordnet.

Falls zwischen Quell- und Zielsektion mehrere Stufen stehen sollten, so wird entsprechend der komplette Namenszweig von der Ziel- bis zur Quellsektion dem Symbolnamen vorangestellt.

3.8.4 FORWARD

So schön das bisher besprochene Modell ist, ein bei Pascal nicht auftauchendes Detail macht Ärger: die bei Assembler möglichen Vorwärtsreferenzen. Bei Vorwärtsreferenzen kann es sein, daß AS im ersten Pass auf ein Symbol einer höheren Sektion zugreift. Dies ist an sich nicht weiter tragisch, solange im zweiten Pass das richtige Symbol genommen wird, es können aber Unfälle der folgenden Art passieren:

```
loop:   .
        <Code>
        ..
        SECTION sub
        ..              ; ***
        bra.s   loop
        ..
loop:   ..
        ENDSECTION
        ..
        jmp     loop    ; Hauptschleife
```

AS wird im ersten Pass das globale Label `loop` verwenden, sofern das Programmstück bei `<Code>` hinreichend lang ist, wird er sich über eine zu große Sprungdistanz beklagen und den zweiten Pass erst gar nicht versuchen. Um die Uneindeutigkeit zu vermeiden, kann man den Symbolnamen mit einem expliziten Bezug versehen:

```
        bra.s   loop[sub]
```

Falls ein lokales Symbol häufig referenziert wird, können die vielen Klammern mit dem `FORWARD`-Befehl eingespart werden. Das Symbol wird damit explizit als lokal angekündigt. AS wird bei Zugriffen auf dieses Symbol automatisch nur im lokalen Symbolbereich suchen. In diesem Falle müßte an der mit `***` gekennzeichneten Stelle dafür der Befehl

```
        FORWARD loop
```

stehen. Damit FORWARD Sinn macht, muß es nicht nur vor der Definition des Symbols, sondern vor seiner ersten Benutzung in der Sektion gegeben werden. Ein Symbol gleichzeitig privat und öffentlich zu definieren, ergibt keinen Sinn und wird von AS auch angemahnt.

Im Gegensatz zu PUBLIC und GLOBAL darf FORWARD auch außerhalb von Sektionen verwendet werden. In diesem Fall bleibt die Funktion erhalten, daß in der Symboltabelle ein 'Platzhalter-Eintrag' angelegt wird. Dieser enthält zwar keinen Wert, und die Bezugnahme auf so einen Eintrag wird wie der Zugriff auf ein (bisher) nicht definiertes Symbol behandelt. Es wird in so einem Fall aber das Used-Flag an diesem Platzhalter-Eintrag gesetzt. Eine FORWARD-Deklaration eines Symbols kann daher nützlich sein, wenn ein Symbol per SYMUSED auf Benutzung geprüft werden soll, und es Referenzen vor seiner eigentlichen Definition im Quellcode gibt.

3.8.5 Geschwindigkeitsaspekte

Die mehrstufige Suche in der Symboltabelle und die Entscheidung, mit welchem Attribut ein Symbol eingetragen werden soll, kosten naturgemäß etwas Rechenzeit. Ein 1800 Zeilen langes 8086-Programm z.B. wurde nach der Umstellung auf Sektionen statt in 33 in 34,5 Sekunden assembliert (80386 SX, 16MHz, 3 Durchgänge). Der Overhead hält sich also in Grenzen: Ob man ihn in Kauf nehmen will, ist (wie am Anfang erwähnt) eine Frage des Geschmacks; man kann AS genauso gut ohne Sektionen verwenden.

3.9 Diverses

3.9.1 SHARED

Gültigkeit: alle Prozessoren

Mit diesem Befehl weist man den AS an, die in der Parameterliste angegebenen Symbole (egal ob Integer, Gleitkomma oder String) im Sharefile mit ihren Werten abzulegen. Ob eine solche Datei überhaupt und in welchem Format erzeugt wird, hängt von den in 2.4 beschriebenen Kommandozeilenschaltern ab. Findet AS diesen Befehl und es wird keine Datei erzeugt, führt das zu einer Warnung.

VORSICHT! Ein eventuell der Befehlszeile anhängender Kommentar wird in die erste, ausgegebene Zeile mit übertragen (sofern die Argumentliste von SHARED

leer ist, wird nur der Kommentar ausgegeben). Falls die Share-Datei für C oder Pascal erzeugt wird, sind einen C/Pascal-Kommentar schließende Zeichenfolgen (`*/` bzw. `*)`) im Kommentar zu vermeiden. AS prüft dies *nicht*!

3.9.2 INCLUDE

Gültigkeit: alle Prozessoren

Dieser Befehl fügt die im Parameter angegebene Datei (die optional in Gänsefüßchen eingeschlossen sein darf) so im Text ein, als ob sie dort stehen würde. Dieser Befehl ist sinnvoll, um Quelldateien aufzuspalten, die alleine nicht in den Speicher passen würden oder um sich "Toolboxen" zu erzeugen.

Falls der angegebene Dateiname keine Endung hat, wird in einem ersten Schritt eine Endung `.INC` angenommen. Erst wenn keine solche Datei existiert, oder der angegebene Name bereits einen Punkt und damit eine Endung enthält, wird nach einer Datei genau mit dem anggegebenen Namen gesucht.

Der Assembler versucht als erstes, die Datei in dem Verzeichnis zu finden, in dem sich auch die Quelldatei befindet, die das `INCLUDE`-Statement enthält - ein eventuell in der Dateiangabe enthaltener Pfad ist also relativ zu deren Pfad, und nicht zu dem Verzeichnis, von dem aus man den Assembler aufgerufen hat. Mit der Kommandozeilenoption

```
-i <Pfadliste>
```

läßt sich eine Liste von Verzeichnissen angeben, in denen automatisch zusätzlich nach der Include-Datei gesucht werden soll. Wird die Datei nicht gefunden, so ist dies ein *fataler* Fehler, d.h. der Assembler bricht sofort ab.

Aus Kompatibilitätsgründen ist es erlaubt, den Namen in Gänsefüßchen zu schreiben,

```
        INCLUDE stddef51
```

und

```
        INCLUDE "stddef51.inc"
```

sind also äquivalent. **ACHTUNG!** Wegen dieser Wahlfreiheit ist hier nur eine Stringkonstante, aber kein Stringausdruck zulässig!

3.9.3 BINCLUDE

Gültigkeit: alle Prozessoren

BINCLUDE dient dazu, in den von AS erzeugten Code Binärdaten einzubetten, die von einem anderen Programm erzeugt wurden (das kann natürlich theoretisch auch von AS selber erzeugter Code sein...). BINCLUDE hat drei Formen:

```
BINCLUDE <Datei>
```

In dieser Form wird die Datei komplett eingebunden.

```
BINCLUDE <Datei>,<Offset>
```

In dieser Form wird der Inhalt der Datei ab <Offset> bis zum Ende der Datei eingebunden.

```
BINCLUDE <Datei>,<Offset>,<Len>
```

In dieser Form werden <Len> Bytes ab Offset <Offset> eingebunden.

Es gelten die gleichen Regeln bezüglich Suchpfaden und impliziten Suffixen wie bei INCLUDE.

3.9.4 MESSAGE, WARNING, ERROR und FATAL

Gültigkeit: alle Prozessoren

Der Assembler prüft zwar die Quelltexte so streng wie möglich und liefert differenzierte Fehlermeldungen, je nach Anwendung kann es aber sinnvoll sein, unter bestimmten Bedingungen zusätzliche Fehlermeldungen auszulösen, mit denen sich logische Fehler automatisch prüfen lassen. Der Assembler unterscheidet drei Typen von Fehlermeldungen, die über die drei Befehle auch dem Programmierer zugänglich sind:

- WARNING: Fehler, die auf möglicherweise falschen oder ineffizienten Code hinweisen. Die Assemblierung läuft weiter, eine Codedatei wird erzeugt.
- ERROR: echte Fehler im Programm. Die Assemblierung läuft weiter, um mögliche weitere Fehler in einem Durchgang entdecken und korrigieren zu können. Eine Codedatei wird nicht erzeugt.

- FATAL: schwerwiegende Fehler, die einen sofortigen Abbruch des Assemblers bedingen. Eine Codedatei kann möglicherweise entstehen, ist aber unvollständig.

Allen drei Befehlen ist das Format gemeinsam, in dem die Fehlermeldung angegeben werden muß: Ein beliebiger String-Ausdruck, der sowohl eine einfache Konstante als auch eine komplizierte Formel sein darf, die zu einem String evaluiert. Dies schließt auch die in 2.7 beschriebene Funktion ein, um Zahlenwerte mit auszugeben:

```
message "Startadresse ist \{start_address}"
```

Diejenigen Anweisungen, die Warnungen oder Fehlermeldungen erzeugen, ergeben nur in Zusammenhang mit bedingter Assemblierung Sinn. Ist für ein Programm z.B. nur ein begrenzter Adreßraum vorhanden, so kann man den Überlauf folgendermaßen testen:

```
ROMSize equ     8000h   ; 27256-EPROM

ProgStart: ..
        <das eigentliche Programm>
           ..
ProgEnd:
        if      ProgEnd-ProgStart>ROMSize
         error   "\aDas Programm ist zu lang!"
        endif
```

Neben diesen fehlererzeugenden Befehlen gibt es noch den Befehl MESSAGE, der einfach nur eine Meldung im Listing und auf der Konsole erzeugt (letzteres nur, wenn nicht im quiet-Modus gearbeitet wird). Seine Benutzung ist den anderen drei Befehlen gleich.

3.9.5 READ

Gültigkeit: alle Prozessoren

READ ist sozusagen das Gegenstück zu der vorigen Befehlsgruppe: mit ihm ist es möglich, *während* der Assemblierung Werte von der Tastatur einzulesen. Wozu das gut sein soll? Um das darzulegen, soll hier ausnahmsweise einmal das Beispiel vor die genauere Erläuterung gezogen werden:

Ein Programm benötigt zum Datentransfer einen Puffer mit einer zur Übersetzungszeit festzulegenden Größe. Um die Größe des Puffers festzulegen, könnte man sie einmal mit EQU in einem Symbol ablegen, es geht aber auch interaktiv mit READ :

```
IF      MomPass=1
 READ    "Puffer (Bytes)",BufferSize
ENDIF
```

Auf diese Weise können Programme sich während der Übersetzung interaktiv konfigurieren, man kann sein Programm z.B. jemandem geben, der es mit seinen Parametern übersetzen kann, ohne im Quellcode „herum stochern" zu müssen. Die im Beispiel gezeigte `IF-` Abfrage sollte übrigens immer verwendet werden, damit der Anwender nur einmal mit der Abfrage belästigt wird.

`READ` ähnelt sehr stark dem `SET-` Befehl, nur daß der dem Symbol zuzuweisende Wert nicht rechts vom Schlüsselwort steht, sondern von der Tastatur eingelesen wird. Dies bedeutet z.B. auch, daß AS anhand der Eingabe automatisch festlegt, ob es sich um eine Integer- oder Gleitkommazahl oder einen String handelt und anstelle einzelner Konstanten auch ganze Formelausdrücke eingegeben werden können.

`READ` darf entweder nur einen Parameter oder zwei Parameter haben, denn die Meldung zur Eingabeaufforderung ist optional. Fehlt sie, so gibt AS eine aus dem Symbolnamen konstruierte Meldung aus.

3.9.6 INTSYNTAX

Gültigkeit: alle Prozessoren

Mit diesem Befehl kann die Menge der Schreibweisen, die für Integer-Konstanten in verschiedenen Zahlensystemen zulässig sind, verändert werden. Nach Auswahl eines bestimmten Zieles gilt ein bestimmter Default-Satz (siehe Abschnitt E), der um einzelne Schreibweisen ergänzt oder bereinigt werden kann. `INTSYNTAX` akzeptiert eine beliebige Liste von Argumenten, von denen jedes mit einem Plus- oder Minuszeichen beginnt, direkt gefolgt von der Kennung der jeweiligen Schreibweise. So stellt man z.B. mit folgendem Befehl

```
intsyntax    -0oct,+0hex
```

ein, daß eine führende Null keine Oktal-, sondern eine Hexadezimalzahl kennzeichnet, was bei manchen Assemblern für den SC/MP gebräuchlich ist. Die Kennungen für die einzelnen Schreibweisen finden sich in Tabelle 2.8. Den Kombinationsmöglichkeiten sind keine Grenzen gesetzt, außer wenn sich zwei Schreibweisen direkt widersprechen. Im obigen Beispiel wäre es z.B. nicht erlaubt, gleichzeitig `0oct` und `0hex` einzuschalten.

3.9.7 RELAXED

Gültigkeit: alle Prozessoren

Defaultmäßig ist einer Prozessorfamilie eine bestimmte Schreibweise von Integer-Konstanten zugeordnet (die i.a. der Herstellervorgabe entspricht, solange der nicht eine allzu abgefahrene Syntax benutzt...). Nun hat aber jeder seine persönlichen Vorlieben für die eine oder andere Schreibweise und kann gut damit leben, daß sich seine Programme nicht mehr mit dem Standard-Assembler übersetzen lassen. Setzt man ein

```
RELAXED ON
```

an den Programmanfang, so kann man fortan alle Schreibweisen beliebig gemischt und durcheinander verwenden; bei jedem Ausdruck versucht AS automatisch zu ermitteln, welche Schreibweise verwendet wurde. Daß diese Automatik nicht immer das Ergebnis liefert, das man sich vorgestellt hat, ist auch der Grund, weshalb diese Option explizit eingeschaltet werden muß (und man sich davor hüten sollte, sie einfach in einem existierenden Programm dazuzusetzen): Ist nicht durch vor- oder nachgestellte Zeichen zu erkennen, daß es sich um Intel- oder Motorola-Konstanten handelt, wird im C-Modus gearbeitet. Eventuell vorangestellte, eigentlich überflüssige Nullen haben in diesem Modus durchaus eine Bedeutung:

```
move.b  #08,d0
```

Diese Konstante würde als Oktalkonstante verstanden werden, und weil Oktalzahlen nur Ziffern von 0..7 enthalten können, führt das zu einem Fehler. Dabei hätte man in diesem Fall noch Glück gehabt, bei der Zahl `077` z.B. hätte man ohne Meldung Probleme bekommen. Ohne `RELAXED`-Modus wäre in beiden Fällen klar gewesen, daß es sich um dezimale Konstanten handelt.

Die momentane Einstellung kann aus dem gleichnamigen Symbol ausgelesen werden.

3.9.8 COMPMODE

Gültigkeit: verschiedene

Auch wenn sich AS bemüht, sich möglichst genauso zu verhalten wie die jeweiligen "Original-Assembler", so gibt es in der Praxis immer wieder Details, wo ein hundertprozentiges Nachbilden des jeweiligen Original- Verhaltens Optimierungen verhindern würde, die aus meiner Sicht valide und nützlich sind. Mit einem

```
compmode on
```

kann man in eine Betriebsart umschalten, die dem "Original-Verhalten" Vorrang vor optimalem Code gibt. Ob für das jeweilige Target solche Fälle vorliegen, ist im jeweiligen Unterkapitel mit dem prozessorspezifischen Hinweisen ausgeführt.

Im Default ist dieser Kompatibilitäts-Modus ausgeschaltet, außer er wurde durch die gleichnamige Kommandozeilen-Option eingeschaltet. Die momentane Einstellung kann aus dem gleichnamigen Symbol ausgelesen werden.

3.9.9 END

Gültigkeit: alle Prozessoren

`END` kennzeichnet das Ende des Assemblerprogrammes. Danach noch in der Quelldatei stehende Zeilen werden ignoriert. **WICHTIG:** `END` darf zwar aus einem Makro heraus aufgerufen werden, der Stapel der bedingten Assemblierung wird aber nicht automatisch abgeräumt. Das folgende Konstrukt führt daher zu einer Fehlermeldung:

```
IF      KeineLustMehr
 END
ENDIF
```

Optional darf `END` auch einen Integer-Ausdruck als Argument haben, der den Startpunkt des Programms vermerkt. Dieser wird von AS in einem speziellen Record der Datei vermerkt und kann z.B. von P2HEX weiterverarbeitet werden.

`END` war eigentlich schon immer in AS definiert, nur war es bei früheren Versionen von AS aus Kompatibilität zu anderen Assemblern vorhanden und hatte keine Wirkung.

Kapitel 4

Prozessorspezifische Hinweise

Ich habe mich bemüht, die einzelnen Codegeneratoren möglichst kompatibel zu den Originalassemblern zu halten, jedoch nur soweit, wie es keinen unvertretbaren Mehraufwand bedeutete. Wichtige Unterschiede, Details und Fallstricke habe ich im folgenden aufgelistet.

4.1 6811

,,Wo gibt es denn das zu kaufen, den HC11 in NMOS?", fragt jetzt vielleicht der eine oder andere. Gibt es natürlich nicht, aber ein H läßt sich nun einmal nicht in einer Hexzahl darstellen (ältere Versionen von AS hätten solche Namen deswegen nicht akzeptiert), und dann habe ich die Buchstaben gleich ganz weggelassen...

> *,,Jemand, der sagt, etwas sei unmöglich,sollte wenigstens so kooperativ sein, denjenigen, der es gerade tut, nicht am Arbeiten zu hindern."*

Ab und zu ist man gezwungen, seine Meinung zu revidieren. Vor einigen Versionen hatte ich an dieser Stelle noch behauptet, ich könne es im Parser von AS nicht realisieren, daß man die Argumente von `BSET/BCLR` bzw. `BRSET/BRCLR` auch mit Leerzeichen trennen kann. Offensichtlich kann selbiger aber mehr, als ich vermutet habe...nach der soundsovielten Anfrage habe ich mich noch einmal drangesetzt, und jetzt scheint es zu laufen. Man darf sowohl Leerzeichen als auch Kommas verwenden, aber nicht in allen Varianten, um es nicht uneindeutig zu machen: Es gibt zu jeder Befehlsvariante zwei Möglichkeiten; eine, die nur Kommas verwendet, sowie eine, wie sie von Motorola wohl definiert wurde (leider sind die Datenbücher nicht immer so gut wie die zugehörige Hardware...):

```
Bxxx  abs8 #mask         entspricht    Bxxx  abs8,#mask
Bxxx  disp8,X #mask      entspricht    Bxxx  disp8,X,#mask
BRxxx abs8 #mask adr     entspricht    BRxxx abs8,#mask,adr
BRxxx disp8,X #mask adr  entspricht    BRxxx disp8,X,#mask,adr
```

Dabei steht `xxx` entweder für `SET` oder `CLR` und `#mask` für die zu verwendende Bitmaske; der Lattenzaun ist dabei optional. Anstelle des X-Registers darf natürlich auch Y verwendet werden.

Mit der K4-Version des HC11 hat Motorola ein Banking-Schema eingeführt, mit dem man zwar einerseits eine zu klein gewordene Architektur noch einmal aufbohren kann, den Software- und Tool-Entwicklern aber nicht unbedingt das Leben einfacher macht...wie stellt man so etwas vernünftig dar?

Die K4-Architektur *erweitert* den Adreßraum des HC11 um 2x512 Kbyte, so daß jetzt insgesamt 64+1024=1088 Kbyte zur Verfügung stehen. AS tut so, als ob es sich dabei um einen Adreßraum handeln würde, der folgendermaßen organisiert ist:

- $000000...$00ffff: der alte HC11-Adreßraum
- $010000...$08ffff: Fenster 1
- $090000...$10ffff: Fenster 2

Über den `ASSUME`-Befehl teilt man AS mit, wie die Banking-Register eingestellt sind und damit, wie und wo die erweiterten Bereiche eingeblendet werden. Bei absoluten Adressierungen mit Adressen jenseits $10000 berechnet AS dann automatisch, welche Adresse innerhalb der ersten 64K anzusprechen ist. Das kann natürlich wieder nur für direkte Adressierungsarten funktionieren, bei indizierten/indirekten Adreßausdrücken ist der Programmierer dafür verantwortlich, über die momentan aktiven Banks den Überblick zu behalten!

Wer sich nicht ganz sicher ist, ob die momentane Einstellung korrekt ist, kann den Pseudobefehl `PRWINS` benutzen, der dann z.B.

```
MMSIZ e1 MMWBR 84 MM1CR 00 MM2CR 80
Window 1: 10000...12000 --> 4000...6000
Window 1: 90000...94000 --> 8000...c000
```

ausgibt. Ein z.B. an Stelle $10000 liegender Befehl

```
        jmp     *+3
```

würde effektiv einen Sprung auf Adresse $4003 auslösen.

4.2 PowerPC

Sicher hat es ein bißchen den Anflug einer Schnapsidee, einen Prozessor, der eher für den Einsatz in Workstations konzipiert wurde, in AS einzubauen, der sich ja eher an Programmierer von Einplatinencomputern wendet. Aber was heute noch das Heißeste vom Heißen ist, ist es morgen schon nicht mehr, und sowohl der Z80 als auch der 8088 haben ja inzwischen die Mutation von der Personal Computer-CPU zum sog. ,,Mikrocontroller" vollzogen. Mit dem Erscheinen von MPC505 und PPC403 hat sich die Vermutung dann auch bestätigt, daß IBM und Motorola diese Prozessorserie auf allen Ebenen durchdrücken wollen.

Die Unterstützung ist momentan noch nicht vollständig: Als Pseudobefehle zur Datenablage werden momentan provisorisch die Intel-Mnemonics unterstützt und es fehlen die etwas ungewöhnlicheren, in [82] genannten RS6000-Befehle (die aber hoffentlich keiner vermißt...). Das wird aber nachgeholt, sobald Informationen verfügbar sind!

4.3 PALM

Der PALM-Prozessor von IBM war lange Zeit ,,Terra Incognita", weil er außerhalb von IBM nie eingesetzt wurde. Des weiteren waren die damit bestückten IBM-Systeme 5100 bis 5120 recht exotisch und teuer, und gerieten über den Erfolg des IBM PC schnell in Vergessenheit. Es ist Christian Corti und seinem akribischen Reverse Engineering zu verdanken, daß dieses Target überhaupt implementiert werden konnte [44].

Als Christian mit dem Reverse Engineering des PALM-Prozessors begann, waren ihm die von IBM definierten Assembler-Mnemonics nicht bekannt, er mußte sich seine eigenen ausdenken. Das geschah natürlich mit dem Wissen im Hinterkopf über all die Architekturen, die nach 1973 bis heute entwickelt wurden.

Vergleicht man seine Mnemonics mit den originalen von IBM (für die dann doch noch eine Dokumentation in [41] aufgetaucht ist...), dann drängt sich ein Vergleich mit den Mnemonics für den Intel 8080/8085 auf der einen Seite, und den Zilog Z80 auf der anderen Seite auf: Die einen packen zum Teil die Adressierungsart mit in den Befehlsnamen, z.B. `MVI` für ,,Move Immediate" oder `LDHD` für ,,LoaD Halfword Direct". Dies ist für einen Assembler deutlich einfacher zu parsen und in Maschinencode umzusetzen.

Die anderen fassen unter einem Mnemonic alle Maschinenbefehle zusammen, die grob das gleiche tun, zum Beispiel `LD` bzw. `MOVE` um Daten zu transferieren. Das

macht die Nutzung für den Programmierer einfacher, sie sind mit den als Operanden geschriebenen Adressierungsarten für einen Assembler aber etwas komplexer zu parsen.

Auf diese Weise haben beide Mnemonic-Sätze ihre Berechtigung: Die einen, weil sie einfach das ,,Original" sind und in allen Herstellerdokumenten verwendet werden, und die anderen, weil sie einfach moderner, zugänglicher und praktischer sind. Ich habe mich deshalb entschieden, *beide* in meinem Assembler zu unterstützen, und glücklicherweise war dies auch ohne Konflikte möglich. Das schließt auch die ,,Makrobefehle" `CALL, RCALL, JMP, BRA, LWI` und `RET` ein. Und was ich selber noch dazu gedichtet habe:

- `AND` und `OR` akzeptieren auch einen Immediate-Wert als zweites Argument. Dies wird auf `SET` bzw. `CLR` abgebildet, wobei bei `AND` natürlich das Einerkomplement des Wertes eingesetzt wird.
- `MOVE` akzeptiert auch einen immediate-Wert als Quelle und erzeugt dann den gleichen Code wie `LWI`. Gleiches gilt für `MOVB` und `LBI`.

Makrobefehle, die aus mehr als einem (Halb-)Wort bestehen, bringen allerdings ein neues Problem mit sich: Der PALM-Prozessor kennt als bedingten Sprung nur das Überspringen des folgenden Instruktionsworts. Folgt auf so einen Sprung ein Makrobefehl, so würde dieser nur teilweise übersprungen. Ich habe daher eine kleine State-Machine vorgesehen, die solche Befehlsfolgen zu erkennen versucht und dann warnt.

Die IBM-Systeme 5110 und 5120 verwenden nicht den heute üblichen ASCII-Zeichensatz, sondern den von IBM-Großrechnern bekannten EBCDIC. Im Include-Verzeichnis befindet sich eine Datei, die man zur Umsetzung von ASCII nach EDCDIC einbinden kann. WICHTIG: EBCDIC ist in dieser Datei als eigene Codepage definiert, man muß die Umsetzung also noch mit einem

```
codepage        cp037
```

aktivieren.

Noch ein Wort zur Integer-Syntax: Christian Corti hatte sich bei seiner Implementierung für die Motorola-Syntax entschieden, also Hex-Konstanten mit vorangestelltem Dollar-Zeichen. Ich habe mich für die 'IBM-Syntax' entschieden, also Zahlenkonstanten in Hochkommas, und z.B. ein vorangestelltes `X` für hexadezimal. Um die Programmbeispiele auf Christians Seiten ohne Änderungen zu übersetzen, füge man folgenden Befehl zu Beginn ein:

```
intsyntax       +$hex,-x'hex'
```

4.4 DSP56xxx

Motorola, was ist nur in Dich gefahren! Wer bei Dir ist nur auf das schmale Brett gekommen, die einzelnen parallelen Datentransfers ausgerechnet durch Leerzeichen zu trennen! Wer immer nun seine Codes etwas übersichtlicher formatieren will, z.B. so:

```
move    x:var9 ,r0
move    y:var10,r3   ,
```

der ist gekniffen, weil das Leerzeichen als Trennung paralleler Datentransfers erkannt wird!

Sei's drum; Motorola hat es so definiert, und ich kann es nicht ändern. Als Trennung der Operationen sind statt Leerzeichen auch Tabulatoren zugelassen, und die einzelnen Teile sind ja wieder ganz normal mit Kommas getrennt.

In [77] steht, daß bei den Befehlen `MOVEC, MOVEM, ANDI` und `ORI` auch die allgemeineren Mnemonics `MOVE, AND` und `OR` verwendet werden können. Bei AS geht das (noch) nicht.

4.5 H8/300

Bei der Assemblersyntax dieser Prozessoren hat Hitachi reichlich bei Motorola abgekupfert (was so verkehrt ja nun auch nicht war...), nur leider wollte die Firma unbedingt ihr eigenes Format für Hexadezimalzahlen einführen, und dazu noch eines, das ein einzelne Hochkommas verwendet, etwa in dieser Form:

```
mov.w #h'ff,r0
```

Dieses Format wird von AS im Default nicht unterstützt, es wird stattdessen die "Motorola-Syntax" mit vorangestelltem Dollarzeichen angeboten. Falls man doch unbedingt die 'Hitachi-Syntax' benutzen will, z.B. um existierenden Code zu übersetzen, so muß der RELAXED-Modus eingeschaltet werden. Bitte beachten, daß diese Syntax bisher wenig getestet wurde und ich keine Garantie geben kann, daß sie in allen Fällen funktioniert!

4.6 H8/500

Der `MOV`-Befehl des H8/500 bietet eine interessante und ungewöhnliche Optimierung: Hat der Zieloperand eine Länge von 16 Bit, so ist es trotzdem möglich, einen nur 8-bittigen (Immediate-)Quelloperanden zu verwenden. Bei einer Anweisung der Form

```
mov.w #$ffff,@$1234
```

ist es also möglich, den Immediate-Quellwert lediglich in einem Byte mit dem Wert $ff zu kodieren und ein Byte einzusparen. Der Prozessor führt eine Vorzeichenerweiterung durch, so daß aus $ff das gewünschte $ffff wird. AS kennt diese Optimierung und benutzt sie auch, wenn dies nicht durch einen expliziten `:16`-Suffix am Immediate-Operanden verboten wird.

Es hat sich in diesem Zusammenhang nur herausgestellt, daß der Original-Assembler von Hitachi diese Optimierung anders umsetzt: er scheint von einer Null- statt einer Vorzeichen-Erweiterung auszugehen. Das bedeutet, nicht Argumente von -128 bis +127 ($ff80 bis $007f) werden als ein Byte kodiert, sondern von 0 bis 255 ($0000 bis $00ff). Tests an echter Hardware haben ergeben, daß das Programmers Manual in dieser Hinsicht Recht hat: Es wird eine Vorzeichen- Erweiterung durchgeführt. Deshalb verwendet AS im Default die kürzere Kodierung nur, wenn im Quellcode ein Wert von -128 bis +127 bzw. ($ff80 bis $007f) benutzt wird. Für existierenden Code, der sich darauf verläßt, daß Werte von $80 bis $ff mit nur einem Byte kodiert werden, kann in einen Kompatibilitätsmodus geschaltet werden, entweder durch ein

```
compmode on
```

im Quellcode oder den gleichnamigen Schalter auf der Kommandozeile.

Ansonsten gelten die gleichen Hinweise bezüglich der Syntax hexadezimaler Zahlen wie für H8/300.

4.7 SH

Leider hat Hitachi auch hier wieder das Extrawurst-Format für Hexadezimalzahlen verwendet, und wieder habe ich in AS das nicht nachvollzogen...bitte Motorola-Syntax benutzen!

Bei der Verwendung von Literalen und dem `LTORG`-Befehl sind einige Details zu beachten, wenn man nicht auf einmal mit eigenartigen Fehlermeldungen konfrontiert werden will:

Literale existieren, weil der Prozessor nicht in der Lage ist, Konstanten außerhalb des Bereiches von -128 bis 127 mit immediate-Adressierung zu laden. AS (und der Hitachi-Assembler) verstecken diese Unzulänglichkeit, indem sie automatisch entsprechende Konstanten im Speicher ablegen, die dann mittels PC-relativer Adressierung angesprochen werden. Die Frage, die sich nun erhebt, ist die, wo diese Konstanten im Speicher abgelegt werden sollen. AS legt sie nicht sofort ab, sondern sammelt sie so lange auf, bis im Programm eine `LTORG`-Anweisung auftritt. Dort werden alle Konstanten abgelegt, wobei deren Adressen mit ganz normalen Labels versehen werden, die man auch in der Symboltabelle sehen kann. Ein Label hat die Form

```
LITERAL_s_xxxx_n    .
```

Dabei repräsentiert `s` den Typ des Literals. Unterschieden werden Literale, die 16-Bit-Konstanten (`s=W`), 32-Bit-Konstanten (`s=L`) oder Vorwärtsreferenzen, bei denen AS die Operandengröße nicht im voraus erkennen kann (`s=F`), enthalten. Für `W` oder `L` bedeutet `xxxx` den hexadezimal geschriebenen Wert der Konstante, bei Vorwärtsreferenzen, bei denen man den Literalwert ja noch nicht kennt, bezeichnet `xxxx` eine einfache Durchnumerierung. `n` kennzeichnet das wievielte Auftreten dieses Literals in dieser Sektion. Literale machen ganz normal die Lokalisierung durch Sektionen mit, es ist daher zwingend erforderlich, in einer Sektion entstandene Literale mit `LTORG` auch dort abzulegen!

Die Durchnumerierung mit `n` ist erforderlich, weil ein Literal in einer Sektion mehrfach auftreten kann. Dies ist einmal bedingt dadurch, daß die PC-relative Adressierung nur positive Displacements erlaubt, einmal mit `LTORG` abgelegte Literale also im folgenden Code nicht mitbenutzt werden können, andererseits auch, weil die Reichweite der Displacements beschränkt ist (512 bzw. 1024 Byte). Ein automatisches `LTORG` am Ende des Programms oder beim Umschalten zu einer anderen CPU erfolgt nicht; findet AS in einer solchen Situation noch abzulegende Literale, so wird eine Fehlermeldung ausgegeben.

Da bei der PC-relativen Adressierung der zur Adressierung herangezogene PC-Wert der Instruktionsadresse+4 entspricht, ist es nicht möglich, ein Literal zu benutzen, welches direkt hinter dem betroffenen Befehl abgelegt wird, also z.B. so:

```
        mov     #$1234,r6
        ltorg
```

Da der Prozessor dann aber sowieso versuchen würde, Daten als Code auszuführen, sollte diese Situation in realen Programmen nicht auftreten. Wesentlich realer ist aber ein anderer Fallstrick: Wird hinter einem verzögerten Sprung PC-relativ zugegriffen, so ist der Programmzähler bereits auf die Sprungzieladresse gesetzt, und das

Displacement wird relativ zum Sprungziel+2 berechnet. Im folgenden Beispiel kann daher das Literal nicht erreicht werden:

```
        bra     Target
        mov #$12345678,r4        ; wird noch ausgefuehrt
        .
        .
        ltorg                        ; hier liegt das Literal
        .
        .
Target: mov     r4,r7                ; hier geht es weiter
```

Da Target+2 hinter dem Literal liegt, würde sich ein negatives Displacement ergeben. Besonders haarig wird es, wenn mit den Befehlen `JMP, JSR, BRAF` oder `BSRF` verzweigt wird: Da AS die Zieladresse hier nicht ermitteln kann (sie ergibt sich erst zur Laufzeit aus dem Registerinhalt), nimmt AS hier eine Adresse an, die nach Möglichkeit nie paßt, so daß PC-relative Adressierung gänzlich unmöglich wird.

Es ist nicht direkt möglich, aus der Zahl und Größe der Literale auf den belegten Speicher zu schließen. U.u. muß AS ein Füllwort einbauen, um einen Langwort-Wert auf eine durch 4 teilbare Adresse auszurichten, andererseits kann er möglicherweise Teile eines 32-bittigen Literals für 16-Bit-Literale mitbenutzten. Mehrfach auftretende Literale erzeugen natürlich nur einen Eintrag. Solche Optimierungen werden für Vorwärtsreferenzen allerdings ganz unterdrückt, da AS den Wert dieser Literale noch nicht kennt.

Da Literale die PC-relative Adressierung ausnutzen, die nur beim `MOV`-Befehl erlaubt sind, beschränken sich Literale ebenfalls auf die Verwendung in `MOV`. Etwas trickreich ist hier die Art und Weise, in der AS die Operandengröße auswertet. Eine Angabe von Byte oder Wort bedeutet, daß AS einen möglichst kurzen `MOV`-Befehl erzeugt, der den angegebenen Wert in den unteren 8 oder 16 Bit erzeugt, d.h. die oberen 24 oder 16 Bit werden als don't care behandelt. Gibt man dagegen Langwort oder gar nichts an, so sagt dies aus, daß das komplette 32-Bit-Register den angegebenen Wert enthalten soll. Das hat z.B. den Effekt, daß in folgendem Beispiel

```
        mov.b   #$c0,r0
        mov.w   #$c0,r0
        mov.l   #$c0,r0
```

der erste Befehl echte immediate-Adressierung erzeugt, der zweite und dritte jedoch ein Wort-Literal benutzen: Da das Bit 7 in der Zahl gesetzt ist, erzeugt der Byte-Befehl effektiv $FFFFFFC0 im Register, was nach der Konvention nicht das wäre,

was man im zweiten und dritten Fall haben möchte. Im dritten Fall reicht auch ein Wort-Literal, weil das gelöschte Bit 15 des Operanden vom Prozessor in Bit 16..31 fortgesetzt wird.

Wie man sieht, ist dieses ganze Literal-Konzept reichlich kompliziert; einfacher ging's aber wirklich nicht. Es liegt leider in der Natur der Sache, daß man manchmal Fehlermeldungen über nicht gefundene Literale bekommt, die eigentlich logisch nicht auftreten könnten, weil AS die Literale ja komplett in eigener Regie verwaltet. Treten aber bei der Assemblierung Fehler erst im zweiten Pass auf, so verschieben sich z.B. hinter der Fehlerstelle liegende Labels gegenüber dem ersten Pass, weil AS für die jetzt als fehlerhaft erkannten Befehle keinen Code mehr erzeugt. Da aber Literalnamen u.a. aus den Werten von Symbolen erzeugt werden, werden als Folgefehler davon eventuell andere Literalnamen nachgefragt, als im ersten Pass abgelegt wurden und AS beschwert sich über nicht gefundene Symbole...sollten also neben anderen Fehlern solche Literal-Fehler auftreten, beseitigen Sie erst die anderen Fehler, bevor Sie mich und alle Literale verfluchen...

Wer aus der Motorola-Ecke kommt und PC-relative Adressierung explizit benutzen will (z.B. um Variablen lageunabhängig zu erreichen), sollte wissen, daß beim Ausschreiben der Adressierung nach Programmierhandbuch, also z.B. so:

```
mov.l   @(Var,PC),r8
```

keine implizite Umrechnung der Adresse auf ein Displacement erfolgt, d.h. der Operand wird so eingesetzt, wie er ist (und würde in diesen Beispiel wohl mit hoher Wahrscheinlichkeit eine Fehlermeldung hervorrufen...). Will man beim SH PC-relativ adressieren, so tut man das einfach mit ,,absoluter" Adressierung, die auf Maschinenebene ja gar nicht existiert:

```
mov.l   Var,r8
```

Hier wird das Displacement korrekt berechnet (es gelten natürlich die gleichen Einschränkungen für das Displacement wie bei Literalen).

4.8 HMCS400

Beim Befehlssatz dieser 4-Bit-Prozessoren fühlte ich mich spontan an den 8080/8085 erinnert - sehr viele Menemonics, die Adressierungsart (z.B. indirekt oder direkt) ist in den Befehl einkodiert, die Befehle sind zum Teil nur schwer zu merken. Natürlich unterstützt AS diese Syntax, wie Hitachi sie seinerzeit definiert hat, ich habe aber

Meta-Instruktion	Ersetzt
LD *src, dest*	LAI, LBI, LMID, LMIIY, LAB, LBA, LAY, LASPX, LASPY, LAMR, LWI, LXI, LYI, LXA, LYA, LAM, LAMD LBM, LMA, LMAD, LMAIY, LMADY
XCH *src, dest*	XMRA, XSPX, XSPY, XMA, XMAD, XMB
ADD *src, dest*	AYY, AI, AM, AMD
ADC *src, dest*	AMC, AMCD
SUB *src, dest*	SYY
SBC *src, dest*	SMC, SMCD
OR *src, dest*	OR, ORM, ORMD
AND *src, dest*	ANM, ANMD
EOR *src, dest*	EORM, EORMD
CP *cond, src, dest*	INEM, INEMD, ANEM, ANEMD, BNEM, YNEI, ILEM, ILEMD, ALEM, ALEMD, BLEM, ALEI
BSET *bit*	SEC, SEM, SEMD
BCLR *bit*	REC, REM, REMD
BTST *bit*	TC, TM, TMD

Tabelle 4.1: Meta-Befehle HMCS400

zusätzlich für die meisten Befehle eine - finde ich - schönere und besser lesbare Variante implementiert, so wie Zilog es seinerzeit mit den Z80 gemacht hat. Zum Beispiel können alle Maschineninstruktionen, die in irgendeiner Form Daten transferieren, egal ob die Operanden Register, Konstanten oder Speicherstellen sind, über den AS-spezifischen LD-Befehl angesprochen werden. Ähnliche 'Meta-Befehle' gibt es für arithmetische und logische Befehle. Eine vollständige Liste aller Meta-Befehle und ihrer Operanden findet sich in den Tabellen 4.1 und 4.2, ihre praktische Verwendung kann man sich in der Datei t_hmcs4x.asm ansehen.

4.9 H16

Der Befehlssatz des H16-Kerns verdient mit Recht den Namen ,,CISC": komplexe Adressierungsarten, sehr variable Instruktionslängen, und für viele Befehle mit gängigen Operanden gibt es Kurzschreibweisen. So gibt es für diverse Befehle mehrere ,,Formate", je nachdem welchen Typ Quell- und Zieloperand haben. Die generelle Regel ist, daß AS immer das kürzestmögliche Format benutzt, es sei denn, es wurde explizit angegeben:

Operand	Typen
src, dest	`A, B, X, Y, W, SPX, SPY` (Register)
	`M` (Speicher adressiert durch X/Y/W)
	`M+` (dito, mit Autoinkrement)
	`M-` (dito, mit Autodekrement)
	`#val` (2/4 bit immediate)
	`addr10` (Speicherzelle direkt)
	`MRn` (Memory-Register 0..15)
cond	`NE` (ungleich)
	`LE` (kleiner oder gleich)
bit	`CA` (Carry)
	bitpos,`M`
	bitpos,`addr10`
bitpos	`0..3`

Tabelle 4.2: Operandentypen für Meta-Befehle HMCS400

```
mov.l     r4,r7     ; benutzt R-Format
mov.l     #4,r7     ; benutzt RQ-Format
mov.l     #4,@r7    ; benutzt Q-Format
mov.l     @r4,@r7   ; benutzt G-Format
mov:q.l   #4,r7     ; Q- statt RQ-Format erzwungen
mov:g.l   #4,r7     ; G- statt RQ-Format erzwungen
```

Für Immediate-Argumente wird die ,,natürliche" Operandenlänge benutzt, also z.B. 2 Bytes für 16 Bits. Kürzere oder längere Argumente lassen sich durch eine angehängte Operandengröße (.b, .w, .l oder :8, :16, :32) erzwingen. Bei Displacements oder absoluten Adressen gilt jedoch, daß ohne explizite Längenangabe immer die kürzestmögliche Schreibweise benutzt wird. Das schließt ein, daß bei absoluten Adressen die oberen acht Adreßbits vom Prozessor nicht herausgegeben werden: eine Adresse $ffff80 kann also mit einem Byte ($80) kodiert werden.

Des weiteren kennt AS das "Akkumulator-Bit", d.h. bei Instruktionen mit zwei beliebigen Operanden kann der zweite Operand weggelassen werden, falls das Ziel Register Null ist. Dieses Verhalten kann nicht übersteuert werden.

Des weiteren werden folgende Optimierungen durchgeführt:

- `MOV R0,<ea>` wird zu `MOVF <ea>` optimiert, sofern `<ea>` kein PC-relativer Ausdruck ist und sich die Länge des Displacements ändern würde. Diese Optimierung kann durch eine explizite Formatangabe unterdrückt werden.

Meta-Instruktion	Ersetzt
`LD` *dest, src*	`LAI, LLI, LHI, L,`
	`LAL, LLA, LAW, LAX, LAY, LAZ,`
	`LWA, LXA, LYA, LPA, LTI, RTH, RTL`
`DEC` *dest*	`DCA, DCL, DCM, DCW, DCX, DCY, DCZ, DCH`
`INC` *dest*	`INA, INL, INM, INW, INX, INY, INZ`
`BSET` *bit*	`SPB, SMB, SC`
`BCLR` *bit*	`RPB, RMB, RC`
`BTST` *bit*	`TAB, TMB, Tc`

Tabelle 4.3: Meta-Befehle OLMS-40

Operand	Typen
src, dest	`A, W, X, Y, Z, DPL, DPH` (Register)
	`T, TL, TH` (Timer, obere/untere Hälfte)
	`(DP), M` (Speicher adressiert durch DPH/DPL)
	`#val` (4/8 bit immediate)
	`PP` (Port-Pointer)
bit	`C` (Carry)
	`(PP)`,*bitpos*
	`(DP)`,*bitpos*
	`(A)`,*bitpos*
bitpos	`0..3`

Tabelle 4.4: Operandentypen für Meta-Befehle OLMS-40

- `SUB` existiert nicht im Q-Format, kann aber durch ein `ADD:Q` mit negiertem immediate-Argument ersetzt werden, falls das Argument zu `SUB` im Bereich -127...+128 liegt. Auch diese Optimierung kann durch eine explizite Formatangabe unterdrückt werden.

4.10 OLMS-40

Ähnlich wie beim HMCS400 sind die Adressierungsarten zu einem großen Teil in die Mnemonics hineinkodiert, und ich habe mich auch hier dafür entschieden, für häufig genutzte Befehle eine alternative, modernere und besser lesbare Notation bereitzustellen. Eine vollständige Liste aller Meta-Befehle und ihrer Operanden findet sich in den Tabellen 4.3 und 4.4, ihre praktische Verwendung kann man sich in der Datei `t_olms4.asm` ansehen.

4.11 OLMS-50

Der Datenspeicher dieser 4-Bit-Controller besteht aus bis zu 128 Nibbles. Für die dafür benötigten sieben Adreßbits war jedoch nur in den wenigsten Instruktionen Platz, so daß einmal wieder Banking zur Adressierung herhalten muß. Die meisten Befehle, die Speicher adressieren, enthalten nur die untersten vier Bits der RAM-Adresse, und sofern nicht die untersten 16 Nibbles angesprochen werden sollen, liefert das P-Register die notwendigen obere Adreßbits. Dessen aktuellen Wert teilt man dem Assembler über ein

```
assume  p:<Wert>
```

mit, z.B. direkt nach einem `PAGE`-Befehl.

Mit `PAGE` ist auch ein anderes Thema angeschnitten: sowohl `PAGE` als auch `SWITCH` sind auf diesen Controllern Maschinenbefehle, d.h. haben nicht ihre von anderen Targets übliche Funktion. Der Pseudobefehl, um ein `SWITCH/CASE`- Konstrukt einzuleiten, lautet im OLMS-50-Modus `SELECT`, und die Seitengröße des Listings legt man mit `PAGESIZE` fest.

4.12 MELPS-4500

Der Programmspeicher dieser Mikrokontroller ist in Seiten zu 128 Worten eingeteilt. Diese Einteilung existiert eigentlich nur deswegen, weil es Sprungbefehle gibt, deren Ziel innerhalb der gleichen Seite liegen darf, und andererseits ,,lange" Exemplare, die den ganzen Adreßbereich erreichen können. Die Standard-Syntax von Mitsubishi verlangt eigentlich, daß Seite und Offset als getrennte Argument geschrieben werden müssen. Da das aber reichlich unpraktisch ist (ansonsten hat man als Programmierer keine Veranlassung, sich um Seiten zu kümmern, mit der Ausnahme von indirekten Sprüngen), erlaubt es AS auch wahlweise, die Zieladresse linear zu schreiben, also z.B.

```
bl        $1234
```

anstelle

```
bl        $24,$34  .
```

4.13 6502UNDOC

Da die undokumentierten Befehle des 6502 sich naturgemäß in keinem Datenbuch finden, sollen sie an dieser Stelle kurz aufgelistet werden. Die Verwendung erfolgt naturgemäß auf eigene Gefahr, da es keine Gewähr gibt, daß alle Maskenversionen alle Varianten unterstützen! Bei den CMOS-Nachfolgern des 6502 funktionieren sie sowieso nicht mehr, da diese die entsprechenden Bitkombinationen mit offiziellen Befehlen belegen...

Es bedeuten:

&	binäres UND
—	binäres ODER
^	binäres EXOR
<<	logischer Linksshift
>>	logischer Rechtsshift
<<<	Linksrotation
>>>	Rechtsrotation
←	Zuweisung
(..)	Inhalt von ..
..	Bits ..
A	Akkumulator
X,Y	Indexregister X,Y
S	Stapelzeiger
An	Akkumulatorbit n
M	Operand
C	Carry
PCH	obere Hälfte Programmzähler

Anweisung : `JAM, KIL` oder `CRS`
Funktion : keine, Prozessor wird angehalten
Adressierungsmodi : implizit

Anweisung : `SLO`
Funktion : $M \leftarrow ((M) << 1)|(A)$
Adressierungsmodi : absolut lang/kurz, X-indiziert lang/kurz,
Y-indiziert lang, X/Y-indirekt

Anweisung : `ANC`
Funktion : $A \leftarrow (A)\&(M), C \leftarrow A7$
Adressierungsmodi : immediate

Anweisung : `RLA`
Funktion : $M \leftarrow ((M) << 1)\&(A)$
Adressierungsmodi : absolut lang/kurz, X-indiziert lang/kurz,
Y-indiziert lang, X/Y-indirekt

Anweisung : `SRE`
Funktion : $M \leftarrow ((M) >> 1)\hat{}(A)$
Adressierungsmodi : absolut lang/kurz, X-indiziert lang/kurz,
Y-indiziert lang, X/Y-indirekt

Anweisung : `ASR`
Funktion : $A \leftarrow ((A)\&(M)) >> 1$
Adressierungsmodi : immediate

Anweisung : `RRA`
Funktion : $M \leftarrow ((M) >>> 1) + (A) + (C)$
Adressierungsmodi : absolut lang/kurz, X-indiziert lang/kurz,
Y-indiziert lang, X/Y-indirekt

Anweisung : `ARR`
Funktion : $A \leftarrow ((A)\&(M)) >>> 1$
Adressierungsmodi : immediate

Anweisung : `SAX`
Funktion : $M \leftarrow (A)\&(X)$
Adressierungsmodi : absolut lang/kurz, Y-indiziert kurz,
Y-indirekt

Anweisung : `ANE`
Funktion : $M \leftarrow ((A)\&\$ee)|((X)\&(M))$
Adressierungsmodi : immediate

Anweisung : `SHA`
Funktion : $M \leftarrow (A)\&(X)\&(PCH+1)$
Adressierungsmodi : X/Y-indiziert lang

Anweisung : `SHS`
Funktion : $X \leftarrow (A)\&(X), S \leftarrow (X), M \leftarrow (X)\&(PCH+1)$
Adressierungsmodi : Y-indiziert lang

Anweisung : `SHY`
Funktion : $M \leftarrow (Y)\&(PCH+1)$
Adressierungsmodi : Y-indiziert lang

Anweisung : `SHX`
Funktion : $M \leftarrow (X)\&(PCH+1)$
Adressierungsmodi : X-indiziert lang

Anweisung : `LAX`
Funktion : $A, X \leftarrow (M)$
Adressierungsmodi : absolut lang/kurz, Y-indiziert lang/kurz,
X/Y-indirekt

Anweisung : `LXA`
Funktion : $X04 \leftarrow (X)04\&(M)04$,
$A04 \leftarrow (A)04\&(M)04$
Adressierungsmodi : immediate

Anweisung : `LAE`
Funktion : $X, S, A \leftarrow ((S)\&(M))$
Adressierungsmodi : Y-indiziert lang

Anweisung : `DCP`
Funktion : $M \leftarrow (M) - 1, Flags \leftarrow ((A) - (M))$
Adressierungsmodi : absolut lang/kurz, X-indiziert lang/kurz,
Y-indiziert lang, X/Y-indirekt

Anweisung : `SBX`
Funktion : $X \leftarrow ((X)\&(A)) - (M)$
Adressierungsmodi : immediate

Anweisung : `ISB`
Funktion : $M \leftarrow (M) + 1, A \leftarrow (A) - (M) - (C)$
Adressierungsmodi : absolut lang/kurz, X-indiziert lang/kurz,
Y-indiziert lang, X/Y-indirekt

4.14 MELPS-740

Die Mikrokontroller dieser Reihe haben ein sehr nettes, verstecktes Feature: Setzt man mit dem Befehl `SET` das Bit 5 des Statusregisters, so wird bei allen arithmetischen Operationen (und Ladebefehlen) der Akkumulator durch die durch das X-Register adressierte Speicherzelle ersetzt. Dieses Feature syntaxmäßig sauber zu integrieren, ist bisher nicht geschehen, d.h. es kann bisher nur im ,,Handbetrieb" (`SET`...Befehle mit Akkuadressierung...`CLT`) genutzt werden.

Nicht alle MELPS-740-Prozessoren implementieren alle Befehle. An dieser Stelle muß der Programmierer aufpassen, daß er nur die Befehle benutzt, die auch wirklich vorhanden sind, da AS die Prozessoren dieser Familie nicht näher unterscheidet. Die Besonderheiten der Special-Page-Adressierung werden bei der Erklärung von `ASSUME` näher erläutert.

4.15 MELPS-7700/65816

Offensichtlich haben diese beiden Prozessorfamilien ausgehend vom 6502 (über ihre 8-bittigen Vorgänger) etwas disjunkte Entwicklungswege hinter sich. Kurz aufgelistet, ergeben sich folgende Unterschiede:

- der 65816 hat keinen B-Akkumulator.
- beim 65816 fehlen Multiplikations- sowie Divisionsbefehle.
- Die Befehle `SEB`, `CLB`, `BBC`, `BBS`, `CLM`, `SEM`, `PSH`, `PUL` und `LDM` fehlen beim 65816. An deren Stelle in der Code-Tabelle finden sich `TSB`, `TRB`, `BIT`, `CLD`, `SED`, `XBA`, `XCE` und `STZ`.

Identische Funktion, jedoch andere Namen haben folgende Befehle:

65816	MELPS-7700	65816	MELPS-7700
REP	CLP	PHK	PHG
TCS	TAS	TSC	TSA
TCD	TAD	TDC	TDA
PHB	PHT	PLB	PLT
WAI	WIT		

Besonders tückisch sind die Befehle `PHB`, `PLB` und `TSB`: diese Befehle haben jeweils eine völlig andere Funktion und Kodierung!

Leider tun diese Prozessoren mit ihrem Speicher etwas, was für mich auf der nach oben offenen Perversitätsskala noch vor der Intel-mäßigen Segmentierung rangiert: sie banken ihn! Nunja, dies ist wohl der Preis für die 6502-Aufwärtskompatibilität; wie dem auch sei, damit AS den gewünschten Code erzeugen kann, muß man ihn über den `ASSUME`-Befehl über den Inhalt einiger Register in Kenntnis setzen:

Das M-Flag bestimmt, ob die Akkumulatoren A und B 8 Bit (1) oder 16 Bit (0) breit sein sollen. Analog entscheidet das Flag X über die Breite der Indexregister X und Y. AS benötigt die Information über die Registerbreite bei unmittelbarer Adressierung (`#<Konstante>`), ob das Argument 8 oder 16 Bit breit sein soll.

Der Speicher ist in 256 Bänke zu 64 Kbyte geteilt. Da alle Register im Prozessor nur maximal 16 Bit breit sind, kommen die obersten 8 Adreßbits aus 2 speziellen Bank-Registern: DT liefert die oberen 8 Bits bei Datenzugriffen, PG erweitert den 16-bittigen Programmzähler auf 24 Bit. Die vom 6502 her bekannte ,,Zero-Page" ist mittels des 16 Bit breiten Registers DPR frei innerhalb der ersten Bank verschiebbar. Trifft AS nun im Code auf eine Adresse (egal ob in einem absoluten, indizierten oder indirekten Ausdruck), so versucht er der Reihe nach folgende Adressierungsvarianten:

1. Liegt die Adresse im Bereich von DPR...DPR+$ff? Falls ja, Verwendung von direkter Adressierung mit 8-Bit-Adresse.

2. Liegt die Adresse innerhalb der durch DT (bzw. PG für Sprungbefehle) festgelegten Seite? Falls ja, Verwendung von absoluter Adressierung mit 16-Bit-Adresse.

3. Falls nichts anderes hilft, Verwendung von langer Adressierung mit 24-Bit-Adresse.

Aus dieser Aufzählung folgt, daß das Wissen über die momentanen Werte von DT,PG und DPR für die Funktion von AS essentiell ist; sind die Angaben fehlerhaft, adressiert das Programm „in die Wüste". Diese Aufzählung geht übrigens davon aus, daß alle drei Adreßlängen verfügbar sind; sollte dies einmal nicht der Fall sein, so wird die Entscheidungskette entsprechen kürzer.

Die oben geschilderte, automatische Festlegung der Adreßlänge läßt sich auch durch die Verwendung von Präfixen übersteuern. Stellt man der Adresse ein <, > oder >> ohne trennendes Leerzeichen voran, so wird eine Adresse mit 1, 2 oder 3 Bytes benutzt, unabhängig davon, ob dies die optimale Länge ist. Benutzt man eine für diesen Befehl nicht erlaubte oder für die Adresse zu kurze Länge, gibt es eine Fehlermeldung.

Um die Portierung von 6502-Programmen zu erleichtern, verwendet AS für Hexadezimalkonstanten die Motorola-Syntax und nicht die von Mitsubishi übrigens für die 740er favorisierte Intel/IEEE-Schreibweise. Ich halte erstere auch für die bessere Schreibweise, und die Entwickler des 65816 werden dies vermutlich ähnlich gesehen haben (da man mittels der `RELAXED`-Anweisung auch Intel-Notation benutzen kann, wird durch diese Entscheidung auch niemand festgelegt). Ein für die Portierung ähnlich wichtiges Detail ist, daß der Akkumulator A als Ziel von Operationen auch weggelassen werden darf, anstelle von `LDA A,#0` darf also z.B. auch einfach `LDA #0` geschrieben werden.

Ein echtes Bonbon in dem Befehlssatz sind dagegen die Blocktransferbefehle `MVN` und `MVP`. Etwas eigenartig ist nur die Adreßangabe: Bit 0–15 im Indexregister, Bit 16–23 im Befehl. Bei AS gibt man als Argument für beide Speicherblöcke einfach die vollen Adressen an, AS fischt sich dann die passenden Bits automatisch heraus. Dies ist ein feiner, aber wichtiger Unterschied zum Mitsubishi-Assembler, bei dem man die oberen 8 Bit selber herausziehen muß. Richtig bequem wird es aber erst mit einem Makro im folgendem Stil:

```
mvpos   macro   src,dest,len
        if      MomCPU=$7700
```

```
 lda    #len
elseif
 lda    #(len-1)
endif
ldx     #(src&$ffff)
ldy     #(dest&$ffff)
mvp     dest,src
endm
```

Vorsicht, Falle: Steht im Akkumulator die Zahl n, so transferiert der Mitsubishi n Bytes, der 65816 jedoch $n + 1$ Bytes!

Sehr nett sind auch die Befehle `PSH` und `PUL`, mit deren Hilfe es möglich ist, mit einem Befehl einen frei wählbaren Satz von Registern auf dem Stack zu sichern oder von ihm zu laden. Nach dem Mitsubishi-Datenbuch[63] muß die Angabe der Bitmasken immediate erfolgen, der Programmierer soll also entweder alle Register↔Bitstellen-Zuordnungen im Kopf behalten oder sich passende Symbole definieren. Hier habe ich die Syntax eigenmächtig erweitert, um die Sache etwas angenehmer zu machen: Es darf eine Liste angegeben werden, die sowohl immediate-Ausdrücke als auch Registernamen enthalten darf. Damit sind z.B. die Anweisungen

```
psh     #$0f
```

und

```
psh     a,b,#$0c
```

und

```
psh     a,b,x,y
```

äquivalent. Da die immediate-Version weiterhin erlaubt ist, bleibt AS hier ,,aufwärtskompatibel" zu den Mitsubishi-Assemblern.

Nicht ganz habe ich beim Mitsubishi-Assembler die Behandlung des `PER`-Befehles verstanden: Mit diesem Befehl kann man eine 16-Bit-Variable auf den Stack legen, deren Adresse relativ zum Programmzähler angegeben wird. Es ist aus der Sicht des Programmierers also eine absolute Adressierung einer Speicherzelle. Nichtsdestotrotz verlangt Mitsubishi eine immediate-Adressierung, und das Argument wird so in den Code eingesetzt, wie es im Quelltext steht. Die Differenz muß man selber ausrechnen, was mit der Einführung von symbolischen Assemblern ja abgeschafft werden sollte...da ich aber auch ein bißchen ,,kompatibel" denken muß, enthält AS eine Kompromißlösung: Wählt man immediate-Adressierung (also mit Gartenzaun),

so verhält sich AS wie das Original von Mitsubishi. Läßt man ihn jedoch weg, so berechnet AS die Differenz vom Argument zum momentanen Programmzähler und setzt diese ein.

Ähnlich sieht es beim `PEI`-Befehl aus, der den Inhalt einer 16-Bit-Variablen auf der Zeropage auf den Stack legt: Obwohl der Operand eine Adresse ist, wird wieder immediate-Adressierung verlangt. Hier läßt AS schlicht beide Versionen zu (d.h. mit oder ohne Gartenzaun).

4.16 M16

Die M16-Familie ist eine Familie äußerst komplexer CISC-Prozessoren mit einem entsprechend komplizierten Befehlssatz. Zu den Eigenschaften dieses Befehlssatzes gehört es unter anderem, daß bei Operationen mit zwei Operanden beide Operanden verschiedene Längen haben dürfen. Die bei Motorola übliche und von Mitsubishi übernommene Methode, die Operandengröße als Attribut an den Befehl anzuhängen, mußte daher erweitert werden: Es ist erlaubt, auch an die Operanden selber Attribute anzuhängen. So wird im folgenden Beispiel

```
mov     r0.b,r6.w
```

Register 0 8-bittig gelesen, auf 32 Bit vorzeichenerweitert und das Ergebnis in Register 6 kopiert. Da man in 9 von 10 Fällen aber von diesen Möglichkeiten doch keinen Gebrauch macht, kann man weiterhin die Operandengröße an den Befehl selber schreiben, z.B. so:

```
mov.w   r0,r6
```

Beide Varianten dürfen auch gemischt verwendet werden, eine Größenangabe am Operanden übersteuert dann den ,,Default" am Befehl. Eine Ausnahme stellen Befehle mit zwei Operanden dar. Bei diesen ist der Default für den Quelloperanden die Größe des Zieloperanden. In folgendem Beispiel

```
mov.h   r0,r6.w
```

wird also auf Register 0 32-bittig zugegriffen, die Größenangabe am Befehl wird überhaupt nicht mehr benutzt. Finden sich überhaupt keine Angaben zur Operandengröße, so wird Wort(w) verwendet. Merke: im Gegensatz zu den 68000ern bedeutet dies 32 und nicht 16 Bit!

Reichlich kompliziert sind auch die verketteten Adressierungsmodi; dadurch, daß AS die Verteilung auf Kettenelemente automatisch vornimmt, bleibt die Sache aber

einigermaßen übersichtlich. Die einzige Eingriffsmöglichkeit, die bei AS gegeben ist (der Originalassembler von Mitsubishi/Green Hills kann da noch etwas mehr), ist die explizite Festlegung von Displacement-Längen mittels der Anhängsel `:4`, `:16` und `:32`.

4.17 CP-3F

Der CP-3F wurde Anfang der 70er-Jahre entwickelt, wo auch die Entwicklungssysteme deutlich weniger leistungsfähig waren und bei der Assembler-Syntax auch auf die leichte Umsetzbarkeit im Maschinensprache geachtet wurde. So gruppieren sich die Original-Mnemonics in wenige Gruppen: Befehle mit 3/4- oder 8-Bit-immediate-Argument, solche mit einem Registeroperanden, Sprünge und Befehle ganz ohne Argumente. Das läst sich mit relativ wenig Aufwand in Maschinencode übersetzen, der Lesbarkeit des Quellcodes ist das aber eher abträglich - ähnlich wie beim Intel 8080. Ich habe mich deshalb entschlossen, für die meisten Befehle eine alternative Schreibweise anzubieten, die verständlicher ausdrückt, was der jeweilige Befehl tut:

Original	Alternativ	Funktion
`las imm4`	`ld a,#<imm4`	A $\longleftarrow$ 0:imm4
`lal imm8`	`ld a,#>imm8`	A $\longleftarrow$ imm8
`lss imm3`	`ld s,#imm3`	S $\longleftarrow$ imm3
`lts imm3`	`ld t,#imm3`	T $\longleftarrow$ imm3
`anl imm8`	`and [a,]#imm8`	A $\longleftarrow$ A $\wedge$ imm8
`eol imm8`	`xor [a,]#imm8`	A $\longleftarrow$ A $\oplus$ imm8
`orl imm8`	`or [a,]#imm8`	A $\longleftarrow$ A $\vee$ imm8
`adl imm8`	`add [a,]#imm8`	A $\longleftarrow$ A + imm8
`cml imm8`	`cp [a,]#imm8`	Flags $\longleftarrow$ A - imm8
`lav`	`ld a,v`	A $\longleftarrow$ V
`law`	`ld a,w`	A $\longleftarrow$ W
`lax`	`ld a,x`	A $\longleftarrow$ X
`lay`	`ld a,y`	A $\longleftarrow$ Y
`sav`	`ld v,a`	V $\longleftarrow$ A
`saw`	`ld w,a`	W $\longleftarrow$ A
`sax`	`ld x,a`	X $\longleftarrow$ A
`say`	`ld y,a`	Y $\longleftarrow$ A
`sat`	`ld t,a`	T $\longleftarrow$ A
`sst`	`ld st,a`	S|T $\longleftarrow$ A
`als`	`sla [a]`	A(7..1) $\longleftarrow$ A(6..0),

Original	Alternativ	Funktion
	sla [a],1	A(0) $\longleftarrow$ 0
ars	srl [a] srl [a],1	A(6..0) $\longleftarrow$ A(7..1), A(7) $\longleftarrow$ 0
alf	sla [a],4	A(7..4) $\longleftarrow$ A(3..0), A(3..0) $\longleftarrow$ 0
arf	srl [a],4	A(3..0) $\longleftarrow$ A(7..4), A(7..4) $\longleftarrow$ 0
lar n	ld a,n	A $\longleftarrow$ R(n), n=0..11
lar 12	ld a,(st)	A $\longleftarrow$ R(S,T)
lar 13	ld a,(st)-	A $\longleftarrow$ R(S,T), S $\longleftarrow$ S-1
lar 14	ld a,(st)+	A $\longleftarrow$ R(S,T), S $\longleftarrow$ S+1
sar n	ld n,a	R(n) $\longleftarrow$ A, n=0..11
sar 12	ld (st),a	R(S,T) $\longleftarrow$ A
sar 13	ld (st)-,a	R(S,T) $\longleftarrow$ A, S $\longleftarrow$ S-1
sar 14	ld (st)+,a	R(S,T) $\longleftarrow$ A, S $\longleftarrow$ S+1
adr n	add a,n	A $\longleftarrow$ A $+$ R(n), n=0..11
adr 12	add a,(st)	A $\longleftarrow$ A $+$ R(S,T)
adr 13	add a,(st)-	A $\longleftarrow$ A $+$ R(S,T), S $\longleftarrow$ S-1
adr 14	add a,(st)+	A $\longleftarrow$ A $+$ R(S,T), S $\longleftarrow$ S+1
anr n	and a,n	A $\longleftarrow$ A $\wedge$ R(n), n=0..11
anr 12	and a,(st)	A $\longleftarrow$ A $\wedge$ R(S,T)
anr 13	and a,(st)-	A $\longleftarrow$ A $\wedge$ R(S,T), S $\longleftarrow$ S-1
anr 14	and a,(st)+	A $\longleftarrow$ A $\wedge$ R(S,T), S $\longleftarrow$ S+1
eor n	xor a,n	A $\longleftarrow$ A $\oplus$ R(n), n=0..11
eor 12	xor a,(st)	A $\longleftarrow$ A $\oplus$ R(S,T)
eor 13	xor a,(st)-	A $\longleftarrow$ A $\oplus$ R(S,T), S $\longleftarrow$ S-1
eor 14	xor a,(st)+	A $\longleftarrow$ A $\oplus$ R(S,T), S $\longleftarrow$ S+1
dec n	dec a,n	R(n) $\longleftarrow$ R(n) - 1, n=0..11

Original	Alternativ	Funktion
dec 12	dec a,(st)	R(S,T) ⟵ R(S,T) - 1
dec 13	dec a,(st)-	R(S,T) ⟵ R(S,T) - 1, S ⟵ S-1
dec 14	dec a,(st)+	R(S,T) ⟵ R(S,T) - 1, S ⟵ S+1
six	ld (z(x)),a	(Z(module X)) ⟵ A
lix	ld a,(z(x))	A ⟵ (Z(module X))
liy	ld a,(z(y))	A ⟵ (Z(module Y))
sqx	ld q(x),a	Q(module X) ⟵ X|A
sqy	ld q(y),a	Q(module Y) ⟵ Y|A
szx	ld z(x),a	Z(module X) ⟵ X|A
szy	ld z(y),a	Z(module Y) ⟵ Y|A

Tabelle 4.5: Alternative Befehlsschreibweisen für CP-3F

4.18 4004/4040

John Weinrich sei dank, habe ich nun auch die offiziellen Datenblätter von Intel über diese 'Urväter' aller Mikroprozessoren, und die Unklarheiten über die Syntax von Registerpaaren (für 8-Bit-Operationen) sind fürs erste ausgeräumt. Die Syntax lautet `RnRm`, wobei `n` bzw. `m` gerade Integers im Bereich 0 bis E bzw. 1 bis F sind. Dabei gilt immer `m = n + 1`.

4.19 MCS-48

Der maximale Adreßraum dieser Prozessoren beträgt 4 KByte, bzw. 8 KByte bei einigen Philips-Varianten. Dieser Raum ist jedoch nicht linear organisiert (wie könnte das bei Intel auch anders sein...), sondern in 2 Bänke zu 2 Kbyte geteilt. Ein Wechsel zwischen diesen beiden Bänken ist nur durch die Befehle `CALL` und `JMP` erlaubt, indem vor dem Sprung das höchste Adreßbit mit den Befehlen `SEL MB0` bis `SEL MB3` vorgegeben wird.

Man kann dem Assembler mit einem

```
ASSUME MB:<0..3>
```

mitteilen, welche Speicherbank gerade für Sprungziele gewählt ist; wird auf eine Adresse gesprungen, die außerhalb dieser Bank liegt, wird eine Warnung ausgegeben.

Wenn der Sonderwert `NOTHING` angegeben wird (dies ist auch der Default), so greift eine in den Befehlen `JMP` und `CALL` eingebaute Automatik , die den Wechsel zwischen den Bänken vereinfacht. Sie fügt automatisch einen `SEL MBx` Befehl ein, falls die Adresse des Sprungbefehles und das Sprungziel in unterschiedlichen Bänken liegen. Die explizite Benutzung der `SEL MBx`-Befehle ist dann nicht mehr notwendig (obwohl sie möglich bleibt) und kann die Automatik auch durcheinanderbringen, wie in dem folgenden Beispiel:

```
000:    SEL     MB1
 JMP     200h
```

AS nimmt an, daß das MB-Flag auf 0 steht und fügt keinen `SEL MB0`-Befehl vor dem Sprung ein, mit der Folge, daß der Prozessor zur Adresse A00h springt. Weiterhin ist zu beachten, daß ein Sprungbefehl durch diesen Mechanismus unter Umständen ein Byte länger wird.

4.20 MCS-51

Dem Assembler liegen die Dateien STDDEF51.INC bzw. 80C50X.INC bei, in denen alle Bits und SFRs der Prozessoren 8051, 8052 und 80515 bzw. 80C501, 502 und 504 verzeichnet sind. Je nach Einstellung des Prozessortyps mit dem `CPU`-Befehl wird dabei die korrekte Untermenge eingebunden, die richtige Reihenfolge für den Anfang eines Programms ist daher

```
 CPU     <Prozessortyp>
 INCLUDE stddef51.inc   ,
```

sonst führen die MCS-51-Pseudobefehle in der Include-Datei zu Fehlermeldungen.

Da der 8051 keinen Befehl kennt, um die Register 0..7 auf den Stack zu legen, muß mit deren absoluten Adressen gearbeitet werden. Diese hängen aber von der momentan aktiven Registerbank ab. Um diesem Mißstand etwas abzuhelfen, ist in den Include-Dateien das Makro `USING` definiert, dem als Parameter die Symbole `Bank0..Bank3` gegeben werden können. Das Makro belegt daraufhin die Symbole `AR0..AR7` mit den passenden absoluten Adressen der Register. Dieses Makro sollte nach jeder Bankumschaltung benutzt werden. Es erzeugt selber *keinen* Code zur Umschaltung!

Das Makro führt in der Variablen `RegUsage` gleichzeitig Buch über alle jemals benutzten Registerbänke; Bit 0 entspricht Bank 0, Bit 1 der Bank 1 usw. . Der Inhalt kann am Ende der Quelldatei z.B. mit folgendem Codestück ausgegeben werden:

```
irp     BANK,Bank0,Bank1,Bank2,Bank3
 if      (RegUsage&(2^BANK))<>0
  message "Bank \{BANK} benutzt"
 endif
endm
```

Mit der Mehrpass-Fähigkeit ab Version 1.38 wurde es möglich, zusätzlich die Befehle `JMP` und `CALL` einzuführen. Bei der Kodierung von Sprüngen mit diesen Befehlen wählt AS je nach Adreßlage automatisch die optimale Variante, d.h. `SJMP/AJMP/LJMP` für `JMP` und `ACALL/LCALL` für `CALL`. Es ist natürlich weiterhin möglich, die Varianten direkt zu verwenden, um eine bestimmte Kodierung zu erzwingen.

Aus historischen Gründe ist das `BITDATA`-Segment im Default *nicht* bitweise organisiert, d. h. eine Anweisung wie

```
flag db ?
```

setzt den Programmzähler nur um eins weiter, und nicht um acht. Es ist quasi implizit vereinbart, dass Bytes in Wirklichkeit nur Bits sind - und wer nicht nur Speicher reserviert, sollte nur Null und eins als Werte ablegen. Dieses etwas unintuitive Verhalten ist der Tatsache geschuldet, daß frühere Versionen von AS schlicht keine Adreßräume mit einer Wortbreite unterstützten, die nicht 8, 16 oder 32 Bit war. Und als es dann möglich war, hätte eine harte Umstellung dazu geführt, daß existierende Programme nicht mehr wie vorher funktionieren.

Wer ein in Bits organisiertes `BITDATA`-Segment haben möchte, muß dieses daher explizit anfordern, indem bei der Auswahl des Ziels eine Option angegeben wird:

```
cpu     8051:bitsegsize=1
```

Des weiteren ist zu beachten, daß bei der Ausgabe von in diesem Segment abgelegten Daten mittels P2BIN oder P2HEX ein einzelnes Bit nach wie vor ein komplettes Byte belegt.

4.21 MCS-251

Intel hat sich beim 80C251 ja bemüht, den Übergang für den Programmierer auf die neue Familie so weich wie möglich zu gestalten, was darin gipfelt, daß alte Anwendungen ohne Neuübersetzung auf dem neuen Prozessor ablaufen können. Sobald man jedoch den erweiterten Befehlssatz der 80C251 nutzen will, gilt es, einige Details zu beachten, die sich als versteckte Fußangeln auftun.

An vorderster Stelle steht dabei die Tatsache, daß der 80C251 keinen getrennten Bitadreßraum mehr hat. Es sind nunmehr alle SFRs unabhängig von ihrer Adreßlage sowie die ersten 128 Speicherstellen des internen RAMs bitadressierbar. Möglich wird dies dadurch, daß die Bitadressierung nicht mehr über einen zusätzlichen virtuellen Adreßraum, der andere Adreßräume überdeckt, erfolgt, sondern so wie bei anderen Prozessoren auch durch eine zweidimensionale Adressierung, die aus der Speicherstelle, die das Bit beinhaltet sowie der Bitstelle im Byte besteht. Dies bedeutet zum einen, daß bei einer Bitangabe wie z.B. PSW.7 AS die Zerlegung der Teile links und rechts vom Punkt selber vornimmt. Es ist also nicht mehr nötig, mittels eines `SFRB`-Befehls wie noch beim 8051 explizit 8 Bitsymbole zu erzeugen. Dies bedeutet zum anderen, daß es den `SFRB`-Befehl überhaupt nicht mehr gibt. Wird er in zu portierenden 8051-Programmen benutzt, kann er durch einen einfachen `SFR`-Befehl ersetzt werden.

Weiterhin hat Intel in den unterschiedlichen Adreßräumen des 8051 gehörig aufgeräumt: Der Bereich des internen RAMs (`DATA` bzw. `IDATA`), der `XDATA`-Bereich und er bisherige `CODE`-Bereich wurden in einem einzigen, 16 Mbyte großen `CODE`-Bereich vereinigt. Das interne RAM beginnt bei Adresse 0, das interne ROM beginnt bei Adresse ff0000h, dorthin muß also auch der Code mittels `ORG` hinverlagert werden. Ausgelagert wurden dagegen die `SFRs` in einen eigenen Adreßraum (der bei AS als `IO`-Segment definiert ist). In diesem neuen Adreßraum haben sie aber die gleichen Adressen wie beim 8051. Der `SFR`-Befehl kennt diesen Unterschied und legt mit ihm erzeugte Symbole je nach Zielprozessor automatisch ins `DATA`- bzw. `IO`-Segment. Da es keinen Bit-Adreßraum mehr gibt, funktioniert der `BIT`-Befehl völlig anders: anstelle einer linearen Adresse von 0 bis 255 beinhalten Bit-Symbole jetzt in Bit 0..7 die Adresse, in Bit 24..26 die Bitstelle. Damit ist es jetzt leider nicht mehr so einfach möglich, Felder von Flags mit symbolischen Namen anzulegen: Wo man beim 8051 noch z.B.

```
        segment bitdata

bit1    db      ?
bit2    db      ?
```

oder

```
defbit macro   name
name    bit     cnt
cnt     set     cnt+1
        endm
```

schreiben konnte, hilft jetzt nur noch die zweite Variante weiter, z.B. so:

```
adr     set     20h     ; Startadresse Flags im internen RAM
bpos    set     0

defbit  macro   name
name    bit     adr.bpos
bpos    set     bpos+1
        if      bpos=8
bpos     set     0
adr      set     adr+1
        endif
        endm
```

Ein weiteres, kleines Detail: Da Intel als Kennzeichnung für den Carry nun CY statt C bevorzugt, sollte man ein eventuell benutztes Symbol umbenennen. AS versteht aber auch weiterhin die alte Variante in den Befehlen `CLR, CPL, SETB, MOV, ANL,` und `ORL`. Gleiches gilt sinngemäß für die dazugekommenen Register `R8..R15, WR0..WR30, DR0..DR28, DR56, DR60, DPX` und `SPX`.

Intel möchte es gerne, daß man absolute Adressen in der Form `XX:YYYY` schreibt, wobei `XX` eine 64K-Bank im Adreßraum angibt bzw. mit einem `S` Adressen im IO-Raum kennzeichnet. Wie man sich schon denken kann, halte ich davon nicht allzu viel, weshalb man an allen Stellen Adressen genauso gut linear angeben kann; lediglich um das S für die Kennzeichnung von I/O-Adressen kommt man nicht herum, z.B. hier:

```
Carry   bit   s:0d0h.7
```

Ohne den Präfix würde AS die absolute Adresse in das Code-Segment legen, und dort sind ja nur die ersten 128 Byte bitadressierbar...

Wie auch schon beim 8051 gibt es die generischen Befehle `JMP` und `CALL`, die je nach Adreßlage automatisch die kürzeste Variante einsetzen. Während `JMP` aber die Variante mit 24 Bit mitberücksichtigt, tut `CALL` dies aus gutem Grund nicht: Der `ECALL`-Befehl legt nämlich im Gegensatz zu `ACALL` und `LCALL` 3 Bytes auf den Stack, und man hätte sonst einen `CALL`-Befehl, bei dem man nicht mehr genau weiß, was er tut. Bei `JMP` tritt diese Problem nicht auf.

Aus einer Sache bin ich nicht ganz schlau geworden: Der 80251 kann auch immediate-Operanden auf den Stack legen, und zwar sowohl einzelne Bytes als auch ganze Wörter. Für beide Varianten ist aber der gleiche Befehl `PUSH` vorgesehen – und woher soll bitte ein Assembler bei einer Anweisung wie

```
        push #10
```

wissen, ob ein Byte oder ein Wort mit dem Wert 10 auf den Stack gelegt werden soll? Daher gilt im Augenblick die Regelung, daß `PUSH` grundsätzlich ein Byte ablegt; wer ein Wort ablegen will, schreibt einfach `PUSHW` anstelle `PUSH`.

Noch ein gut gemeinter Ratschlag: Wer den erweiterten Befehlssatz des 80C251 nutzt, sollte den Prozessor auch tunlichst im Source-Modus betreiben, sonst werden alle neuen Anweisungen ein Byte länger! Um die originären 8051-Anweisungen, die dafür im Source-Modus länger werden, ist es nicht besonders schade: Sie werden entweder von AS automatisch durch neue, leistungsfähigere ersetzt oder sind betreffen veraltete Adressierungsarten (indirekte Adressierung mit 8-Bit-Registern).

4.22 8080/8085

Wie schon weiter vorne erwähnt, ist es möglich, durch ein

```
Z80SYNTAX <OFF|ON|EXCLUSIVE>
```

für die allermeisten 8080/8085-Befehle möglich, sie auch wahlweise oder ausschließlich im 'Z80-Stil' zu schreiben, d.h. mit weniger Mnemonics, dafür aber mit deutlich aussagekräftigeren Operanden. Für die folgenden Befehle ist die Z80-Syntax im nicht-exklusiven Modus nicht möglich, da sie mit existierenden 8080-Mnemonics kollidieren:

- `CP` ist in der 'Intel-Syntax' der Befehl für 'Call on Positive', in der Zilog-Syntax jedoch der Befehl für 'Compare'. Verwendet man `CP` mit einem absoluten Zahlenwert als Argument, so ist für den Assembler nicht zu erkennen, ob das ein Sprung zu einer absoluten Adresse sein soll oder ein Vergleich mit einem immediate-Wert. Der Assembler wird in so einem Fall einen Sprung kodieren, da die Intel-Syntax bei Mehrdeutigkeiten Vorrang hat. Möchte man den Vergleich haben, so kann man den Akkumulator als Zieloperanden explizit hinschreiben, also z.B. `CP A,12h` anstatt `CP 12h`.

- `JP` ist in der Intel-Syntax der Befehl für 'Jump on Positive', in der Zilog-Syntax jedoch der allgemeine Sprung-Befehl. Für bedingte Sprünge in Zilog-Syntax (`JP cond,addr`) ist die Sache wegen der zwei Argumente eindeutig, bei nur einem Argument wird der Assembler aber den bedingten Sprung kodieren. Wer einen unbedingten Sprung zu einer absoluten Adresse haben möchte, muß weiterhin die Intel-Variante (`JMP addr`) verwenden.

Der 8085 unterstützt mit `RIM` und `SIM` zwei Befehle, die im Z80-Befehlssatz nicht existieren. Diese können "Z80-artig" als `LD A,IM` bzw. `LD IM,A` geschrieben werden.

Der beim Z80-Target definierte 'generische Sprungbefehl' `J` ist bei aktivierter Z80-Syntax ebenfalls verfügbar, da der 8080/8085 jedoch keine relativen Sprünge unterstützt, wird `J` immer in `JP` übersetzt.

4.23 8085UNDOC

Ähnlich wie beim Z80 oder 6502, sind auch beim 8085 die undokumentierten Befehle nicht näher von Intel spezifiziert worden, weshalb es nicht undenkbar ist, daß andere Assembler andere Mnemonics dafür verwenden. Deshalb sollen auch diese Befehle und ihre Funktion hier kurz aufgelistet werden. Und auch hier wieder ist die Verwendung dieser Befehle auf eigenes Risiko - schon der an sich zum 8085 aufwärtskompatible Z80 benutzt diese Opcodes für völlig andere Funktionen...

Anweisung : `DSUB [reg]`
Z80-Syntax : `SUB HL,reg`
Funktion : HL ← HL - reg
Flags : CY, S, X5, AC, Z, V, P
Argumente : `reg` = B für BC (optional für nicht-Z80-Syntax)

Anweisung : `ARHL`
Z80-Syntax : `SRA HL`
Funktion : HL,CY ← HL >> 1 (arithmetisch)
Flags : CY
Argumente : keine bzw. fix für Z80-Syntax

Anweisung : `RDEL`
Z80-Syntax : `RLC DE`
Funktion : CY,DE ← DE << 1
Flags : CY, V
Argumente : keine bzw. fix für Z80-Syntax

Anweisung : `LDHI d8`
Z80-Syntax : `ADD DE,HL,d8`
Funktion : DE ← HL + `d8`
Flags : keine
Argumente : `d8` = 8-Bit-Konstante, Register fix für Z80-Syntax

Anweisung : `LDSI d8`
Z80-Syntax : `ADD DE,SP,d8`
Funktion : DE ← SP + `d8`
Flags : keine
Argumente : `d8` = 8-Bit-Konstante, Register fix für Z80-Syntax

Anweisung : `RSTflag`
Z80-Syntax : `RST flag`
Funktion : Restart zu 40h wenn `flag`=1
Flags : keine
Argumente : `flag` = V für Overflow-Bit

Anweisung : `SHLX [reg]`
Z80-Syntax : `LD (reg),HL`
Funktion : [reg] ← HL
Flags : keine
Argumente : `reg` = D/DE für DE (optional für nicht-Z80-Syntax)

Anweisung : `LHLX [reg]`
Z80-Syntax : `LD HL,(reg)`
Funktion : HL ←[reg]
Flags : keine
Argumente : `reg` = D/DE für DE (optional für nicht-Z80-Syntax)

Anweisung : `JNX5 adr`
Z80-Syntax : `JP NX5, adr`
Funktion : springe zu `adr` wenn X5=0
Flags : keine
Argumente : `adr` = absolute 16-Bit-Adresse

Anweisung : `JX5 adr`
Funktion : springe zu `adr` wenn X5=1
Flags : keine
Argumente : `adr` = absolute 16-Bit-Adresse

Mit X5 ist dabei das ansonsten unbenutzte Bit 5 im PSW-Register gemeint.

4.24 8086..V35

Eigentlich hatte ich mir geschworen, die Segmentseuche der 8086er aus diesem Assembler herauszuhalten. Da aber nun eine Nachfrage kam und Studenten flexiblere Menschen als die Entwickler dieses Prozessors sind, findet sich ab sofort auch eine rudimentäre Unterstützung dieser Prozessoren in AS. Unter ,,rudimentär" verstehe ich dabei nicht, daß der Befehlssatz nicht vollständig abgedeckt wird, sondern daß ich nicht den ganzen Wust an Pseudoanweisungen integriert habe, die sich bei MASM, TASM & Co. finden. AS ist auch nicht in erster Linie geschrieben worden, um PC-Programme zu entwickeln (Gott bewahre, das hieße wirklich, das Rad neu zu erfinden), sondern zur Programmentwicklung für Einplatinenrechner, die eben unter anderem auch mit 8086ern bestückt sein können.

Für Unentwegte, die mit AS doch DOS-Programme schreiben wollen, eine kleine Liste dessen, was zu beachten ist:

- Es können nur COM-Programme erzeugt werden.
- Verwenden Sie nur das `CODE`-Segment, und legen Sie auch alle Variablen darin ab.
- Alle Segmentregister werden von DOS auf das Codesegment vorinitialisiert. Ein `ASSUME DS:CODE, SS:CODE` am Programmanfang ist daher notwendig.

Assembler	Adresse	Inhalt

MASM	`mov ax,offset vari`
	`lea ax,vari`
	`lea ax,[vari]`
		`mov ax,vari`
		`mov ax,[vari]`

AS	`mov ax,vari` `lea ax,[vari]`	`mov ax,[vari]`

Tabelle 4.6: Unterschiede in der Adressierungssyntax AS↔MASM

- DOS lädt den Code ab Adresse 100h. Ein `ORG` auf diese Adresse ist daher zwingend.
- Die Umwandlung in eine Binärdatei erfolgt mit P2BIN (s.u.), wobei als als Adreßbereich `$-$` anzugeben ist.

Allgemein unterstützt AS für diese Prozessoren nur ein Small-Programmiermodell, d.h. *ein* Codesegment mit maximal 64 KByte und ein ebenfalls höchstens 64 KByte großes Datensegment mit (für COM-Dateien uninitialisierten) Daten. Zwischen diesen beiden Segmenten kann mit dem `SEGMENT`-Befehl hin-und hergeschaltet werden. Aus dieser Tatsache folgert, daß Sprünge immer intrasegmentär sind, sofern sie sich auf Adressen im Codesegment beziehen. Falls weite Sprünge doch einmal erforderlich sein sollten, können sie mit `CALLF` und `JMPF` und einer Speicheradresse oder einen Segment:Offset-Wert als Argument erreicht werden.

Ein weiteres großes Problem dieser Prozessoren ist deren Assemblersyntax, deren genaue Bedeutung nur aus dem Zusammenhang erkennbar ist. So kann im folgenden Beispiel je nach Symboltyp sowohl unmittelbare als auch absolute Adressierung gemeint sein:

```
mov     ax,wert
```

Bei AS ist immer unmittelbare Adressierung gemeint, wenn um den Operanden keine eckigen Klammern stehen. Soll z.B. die Adresse oder der Inhalt einer Variablen geladen werden, so ergeben sich die in Tabelle 4.6 aufgelisteten Unterschiede.

Der Assembler prüft bei Symbolen, ob sie im Datensegment liegen und versucht, automatisch einen passenden Segmentpräfix einzufügen, z.B. falls ohne CS-Präfix auf Symbole im Code zugegriffen wird. Dieser Mechanismus kann jedoch nur funktionieren, falls der `ASSUME`-Befehl (siehe dort) korrekt angewendet wurde.

Die Intel-Syntax verlangt eine Abspeicherung, ob an einem Symbol Bytes oder Wörter abgelegt wurden. AS nimmt diese Typisierung nur vor, falls in der gleichen Zeile wie das Label ein `DB` oder `DW` steht. Für alle anderen Fälle muß mit den Operatoren `WORD PTR, BYTE PTR` usw. explizit angegeben werden, um was für eine Operandengröße es sich handelt. Solange ein Register an der Operation beteiligt ist, kann auf diese Kennzeichnung verzichtet werden, da durch den Registernamen die Operandengröße eindeutig bestimmt ist.

Der Koprozessor in 8086-Systemen wird üblicherweise durch den TEST-Eingang des Prozessors synchronisiert, indem selbiger mit dem BUSY-Ausgang des Koprozessors verbunden wird. AS unterstützt dieses Handshaking, indem vor jedem 8087-Befehl automatisch ein `WAIT`-Befehl eingefügt wird. Ist dies aus irgendwelchen Gründen unerwünscht (z.B. während der Initialisierung), so muß im Opcode hinter dem `F` ein `N` eingefügt werden; aus

```
FINIT
FSTSW   [vari]
```

wird so z.B.

```
FNINIT
FNSTSW  [vari]
```

Diese Variante ist bei *allen* Koprozessorbefehlen erlaubt.

4.25 8X30x

Die Prozessoren dieser Reihe sind auf eine einfache Manipulation von Bitgruppen auf Peripherieadressen optimiert worden. Um mit solchen Bitgruppen auch symbolisch umgehen zu können, existieren die Befehle `LIV` und `RIV`, mit denen einer solchen Bitgruppe ein symbolischer Name zugewiesen wird. Diese Befehle arbeiten ähnlich wie `EQU`, benötigen aber drei Parameter:

1. die Adresse der peripheren Speicherzelle, in der sich die Bitgruppe befindet (0..255);
2. die Bitnummer des ersten Bits in der Gruppe (0..7);
3. die Länge der Gruppe in Bits (1..8).

ACHTUNG! Der 8X30x unterstützt keine Bitgruppen, die über mehrere Speicherstellen hinausreichen, so daß je nach Startposition der Wertebereich für die Länge eingeschränkt sein kann. AS nimmt hier **keine** Prüfung vor, man bekommt lediglich zur Laufzeit merkwürdige Ergebnisse!

Im Maschinencode drücken sich Länge und Position durch ein 3-Bit-Feld im Instruktionswort sowie ein passende Registernummer (`LIVx` bzw. `RIVx`) aus. Bei der Verwendung eines symbolischen Objektes wird AS diese Felder automatisch richtig besetzen, es ist aber auch erlaubt, die Länge als dritten Operanden explizit anzugeben, wenn man nicht mit symbolischen Busobjekten arbeitet. Trifft AS auf eine Längenangabe trotz eines symbolischen Operanden, so vergleicht er beide Längen und gibt eine Fehlermeldung bei Ungleichheit aus (das gleiche passiert übrigens auch, wenn man bei einem `MOVE`-Befehl zwei symbolische Operanden mit unterschiedlicher Länge benutzt - die Instruktion hat einfach nur ein Längenfeld...).

Neben den eigentlichen Maschinenbefehlen des 8X30x implementiert AS noch ähnlich wie das ,,Vorbild" MCCAP einige Pseudoinstruktionen, die als eingebaute Makros ausgeführt sind:

- `NOP` ist eine Kurzschreibweise für `MOVE AUX,AUX`
- `HALT` ist eine Kurzschreibweise für `JMP *`
- `XML ii` ist eine Kurzschreibweise für `XMIT ii,R12` (nur 8X305)
- `XMR ii` ist eine Kurzschreibweise für `XMIT ii,R13` (nur 8X305)
- `SEL <busobj>` ist eine Kurzschreibweise für `XMIT <adr>,IVL/IVR`, führt also die notwendige Vorselektion durch, um `<busobj>` ansprechen zu können.

Die bei MCCAP ebenfalls noch vorhandenen `CALL-` und `RTN-`Instruktionen sind mangels ausreichender Dokumentation momentan nicht implementiert. Das gleiche gilt für einen Satz an Pseudoinstruktionen zur Datenablage. Kommt Zeit, kommt Rat...

4.26 XA

Ähnlich wie sein Vorgänger MCS/51, jedoch im Unterschied zu seinem ,,Konkurrenten" MCS/251 besitzt der Philips XA einen getrennten Bitadreßraum, d.h. alle mit Bitbefehlen manipulierbaren Bits haben eine bestimmte, eindimensionale Adresse, die in den Maschinenbefehlen auch so abgelegt wird. Die naheliegende Möglichkeit,

diesen dritten Adreßraum (neben Code und Daten) auch so in AS anzubieten, habe ich nicht nutzen können, und zwar aus dem Grund, daß ein Teil der Bitadressen im Gegensatz zum MCS/51 nicht mehr eindeutig ist: Bits mit den Adressen 256 bis 511 bezeichnen Bits der Speicherzellen 20h..3fh aus dem aktuellen Datensegment. Dies bedeutet aber, daß diese Adressen je nach Situation unterschiedliche Bits ansprechen können - ein definieren von Bits mit Hilfe von `DC`-Befehlen, was durch ein extra Segment möglich geworden wäre, würde also nicht übermäßig viel Sinn ergeben. Zur Definition einzelner, symbolisch ansprechbarer Bits steht aber nach wie vor der `BIT`-Befehl zur Verfügung, mit dem beliebige Bitadressen (Register, RAM, SFR) definiert werden können. Für Bitadressen im internen RAM wird auch die 64K-Bank-Adresse gespeichert, so daß AS Zugriffe überprüfen kann, sofern das DS-Register korrekt mit `ASSUME` vorbesetzt wurde.

Nichts drehen kann man dagegen an den Bemühungen von AS, potentielle Sprungziele (also Zeilen im Code mit Label) auf gerade Adressen auszurichten. Dies macht AS genauso wie andere XA-Assembler auch durch Einfügen von `NOP`s vor dem fraglichen Befehl.

4.27 AVR

Im Gegensatz zum AVR-Assembler verwendet AS defaultmäßig das Intel-Format zur Darstellung von Hexadezimalkonstanten und nicht die C-Syntax. OK, nicht vorher in den (freien) AVR-Assembler hineingeschaut, aber als ich mit dem AVR-Teil anfing, gab es zum AVR noch nicht wesentlich mehr als ein vorläufiges Datenbuch mit Prozessortypen, die dann doch nie kamen...mit einem `RELAXED ON` schafft man dieses Problem aus der Welt.

Optional kann AS für die AVRs (es geht auch für andere CPU's, nur macht es dort keinen Sinn...) sogenannte ,,Objekt-Dateien" erzeugen. Das sind Dateien, die sowohl Code als auch Quellzeileninformationen enthalten und z.B. eine schrittweise Abarbeitung auf Quellcodeebene mit dem von Atmel gelieferten Simulator WAVR-SIM erlauben. Leider scheint dieser mit Quelldateispezifikationen, die länger als ca. 20 Zeichen sind, seine liebe Not zu haben: Namen werden abgeschnitten oder um wirre Sonderzeichen ergänzt, wenn die Maximallänge überschritten wird. AS speichert deshalb in den Objekt-Dateien Dateinamen ohne Pfadangabe, so daß es eventuell Probleme geben könnte, wenn Dateien (z.B. Includes) nicht im Arbeitsverzeichnis liegen.

Eine kleine Besonderheit sind Befehle, die Atmel bereits in der Architektur vorgesehen hat, aber noch in keinem Mitglied der Familie implementiert wurden. Dabei

handelt es sich um die Befehle `MUL, JMP` und `CALL`. Besonders bei letzteren fragt man sich vielleicht, wie man denn nun den 4 KWorte großen Adreßraum des AT90S8515 erreichen kann, wenn die 'nächstbesten' Befehle `RJMP` und `RCALL` doch nur 2 KWorte weit springen kann. Der Kunstgriff lautet 'Abschneiden der oberen Adreßbits' und ist näher bei der `WRAPMODE`-Anweisung beschrieben.

Für alle AVR-CPUs ist das CPU-Argument `CODESEGSIZE` definiert. Mit einem

```
cpu atmega8:codesegsize=0
```

weist man den Assembler an, das Code-Segment (also das interne Flash-ROM) nicht als in 16-Bit-Worten, sondern in 8-Bit-Bytes organisiert zu betrachten. Dies ist die Sichtweise, wie man sie beim `LPM`-Befehl hat und wie sie einige andere (nicht-Atmel) Assembler grundsätzlich verfolgen. Sie hat den Vorteil, daß man Adressen im `CODE`-Segment für Datenzugriffe nicht selber mit zwei multiplizieren muß, andererseits muß aber darauf geachtet werden, daß Instruktionen niemals auf einer ungeraden Adresse liegen dürfen - sie würden dann ja quasi halb auf einem und halb auf dem nächsten 16-Bit-Wort im Flash-ROM liegen. `PADDING` ist deshalb im Default aktiviert, es bleibt aber möglich, mit `DB` oder `DATA` Byte-Felder zu definieren, ohne daß zwischen den Anweisungen Padding-Bytes eingestreut werden. Bei relativen oder absoluten Sprüngen werden die Adressen im "Byte-Modus" automatisch durch zwei geteilt. Default ist die vom Atmel-Assembler vorgegebene Organisation in unteilbare 16-Bit-Worte. Diese kann auch explizit mit dem Argument `codesegsize=1` gewählt werden.

4.28 Z80, Z18, Z280, Z380

Der Z80 kennt zwei Arten von Sprüngen: relativ (`JR`) erlaubt Sprungdistanzen von -128 bis +127 Byte, während man mit absoluten Sprüngen (`JP`) den kompletten Adreßraum erreicht. AS unterstützt einen Pseudobefehl `J`, der anhand der Zieladresse und der Sprungbedingung automatisch die kürzestmögliche Variante wählt (relative Sprünge sind nicht für alle Bedingungen definiert). Dies gilt auch für alle vom Z80 abgeleiteten Targets, wie Z80UNDOC, Z180, Z280, RABBIT2000 und LR35902.

Für den Z380 wird `J` ebenfalls angeboten, da der Z380 jedoch größere Sprungdistanzen unterstützt, wird erst auf absolute Sprünge ausgewichen, wenn die gößtmögliche Sprungdistanz (+/- 8 MByte) nicht mehr ausreicht.

Generell sollte dieser Befehl mit Bedacht eingesetzt werden, da `JR` und `JP` nicht gänzlich funktionsgleich sind: Code, der ausschließlich relative Sprünge benutzt, kann positionsunabhängig ausgeführt werden, während absolute Sprünge einer Relozierung bedürfen.

4.29 Z80UNDOC

Da es von Zilog naturgemäß keine Syntaxvorgaben für die undokumentierten Befehle gibt und wohl auch nicht jeder den kompletten Satz kennt, ist es vielleicht sinnvoll, diese Befehle hier kurz aufzuzählen:

Wie auch beim Z280/Z380 und eZ80 ist es möglich, die Byte-Hälften von IX und IY einzeln anzusprechen. Im einzelnen sind dies folgende Varianten:

```
INC Rx                LD R,Rx              LD  Rx,n
DEC Rx                LD Rx,R              LD  Rx,Ry
ADD/ADC/SUB/SBC/AND/XOR/OR/CP A,Rx
```

Dabei stehen `Rx` bzw. `Ry` für `IXL, IXU, IYL` oder `IYU`. Zu beachten ist jedoch, daß in der `LD Rx,Ry`-Variante beide Register aus dem gleichen Indexregister stammen müssen.

Die Kodierung von Schiebebefehlen besitzt noch eine undefinierte Bitkombination, die als `SL1`-, `SLI`-, `SLIA`- oder `SLS`-Befehl zugänglich ist. Er funktioniert wie `SLA`, es wird jedoch eine Eins und nicht eine Null in Bit 0 eingeschoben. **ACHTUNG**: In Internet findet man für diesen Befehl auch den Namen `SLL`, ich habe mich aber entschieden, dies nicht zu implementieren. `SLL` steht für „Shift Logically Left" und die von diesem Befehl ausgeführte Operation ist kein logischer Linksshift - wenn `SLL` überhaupt definiert werden sollte, dann als Alias zu `SLA`. Wer für existierenden Code `SLL` mit der Funktion von `SL1/SLI` braucht, definiere sich dafür ein Makro.

Dieser undokumentierte Schiebebefehl kann, wie alle anderen Schiebebefehle auch, noch in zwei weiteren Varianten benutzt werden:

```
SLIA    R,(XY+d)
SLIA    (XY+d),R
```

Dabei steht `R` für ein beliebiges 8-Bit-Register (aber nicht eine Indexregisterhälfte...), und `(XY+d)` für eine normale indexregister-relative Adressierung. Das Ergebnis dieser Operation ist, daß das Schiebeergebnis zusätzlich ins Register geladen wird. Dies funktioniert auch bei den `RES-` und `SET-`Befehlen:

```
SET/RES R,n,(XY+d)
SET/RES n,(XY+d),R
```

Des weiteren gibt es noch zwei versteckte I/O-Befehle:

```
IN      (C) bzw. TSTI
OUT     (C),0
```

Deren Funktionsweise sollte klar sein. **ACHTUNG!** Es gibt keine Garantie dafür, daß alle Z80-Masken alle diese Befehle beherrschen, und die Z80-Nachfolger lösen zuverlässig Traps aus. Anwendung daher auf eigene Gefahr...

4.30 GB_Z80 bzw. LR35902

Das im originalen Gameboy verbaute SoC namens LR35092 wurde von Sharp entwickelt, und der darin verbaute Prozessorkern ist (vermutlich) der gleiche wie in den SM83-Mikrokontrollern. Dieser ist vom Befehlssatz auf dem ,,halben Weg" zwischen 8080 und Z80, allerdings auch mit seinen eigenen Auslassungen und Erweiterungen. Für die neuen Befehle hat Sharp natürlich eine Assembler-Syntax definiert, in der ,,Gameboy-Szene" haben sich aber auch einige alternative Schreibweisen eingebürgert. Ich habe mich bemüht, diese (so weit bekannt) zu berücksichtigen:

Sharp	alternativ	Funktion
LD A,(HLD)	LD A,(HL-) LDD A,(HL)	A ⟵ (HL), HL ⟵ HL-1
LD A,(HLI)	LD A,(HL+) LDI A,(HL)	A ⟵ (HL), HL ⟵ HL+1
LD (HLD),A	LD (HL-),A LDD (HL),A	(HL) ⟵ A, HL ⟵ HL-1
LD (HLI),A	LD (HL+),A LDI (HL),A	(HL) ⟵ A, HL ⟵ HL+1
LD A,(C)	LD A,(FF00+C) LDH A,(C)	A ⟵ (0ff00h+C)
LD (C),A	LD (FF00+C),A LDH (C),A	(0ff00h+C) ⟵ A
LD (FF00+n),A	LDH (n),A	(0ff00h+n) ⟵ A
LD A,(FF00+n)	LDH A,(n)	A ⟵ (0ff00h+n)
LDHL SP,d	LD HL,SP+d	HL ⟵ SP + d
LDX A,(nn)	LD A,(nn)	A ⟵ (nn) [1]
LDX (nn),A	LD (nn),A	(nn) ⟵ A [1]
[1] erzwingt 16-Bit-Adressierung.		

4.31 Z380

Da dieser Prozessor als Enkel des wohl immer noch beliebtesten 8-Bit-Prozessors konzipiert wurde, war es bei der Entwicklung unabdingbar, daß dieser bestehende Z80-Programme ohne Änderung ausführen kann (natürlich geringfügig schneller, etwa um den Faktor 10...). Die erweiterten Fähigkeiten können daher nach einem Reset mit zwei Flags zugeschaltet werden, die XM (eXtended Mode, d.h. 32- statt 16-Bit-Adreßraum) und LW (long word mode, d.h. 32- statt 16- Bit-Operanden) heißen. Deren Stand muß man AS über die Befehle `EXTMODE` und `LWORDMODE` mitteilen, damit

Adressen und Konstantenwerte gegen die korrekten Obergrenzen geprüft werden. Die Umschaltung zwischen 32- und 16-Bit-Befehlen bewirkt natürlich nur bei solchen Befehlen etwas, die auch in einer 32-Bit-Version existieren; beim Z380 sind das momentan leider nur Lade- und Speicherbefehle, die ganze Arithmetik kann nur 16-bittig ausgeführt werden. Hier sollte Zilog wohl noch einmal etwas nachbessern, sonst kann man den Z380 selbst beim besten Willen nur als ,,16-Bit-Prozessor mit 32-Bit-Erweiterungen" bezeichnen...

Kompliziert wird die Sache dadurch, daß sowohl die mit LW eingestellte Operandengröße, als auch die Länge von immediate- und absoluten Operanden, für einzelne Befehle mit den Präfixen `DDIR W/LW/IB/IW` AS merkt sich die Verwendung solcher Befehle und berücksichtigt sie für den nächsten Befehl automatisch mit. Dabei sind die folgenden Fälle zu unterscheiden:

Ging gar kein `DDIR`-Befehl voraus bzw. einer, der keine explizite Operandenlänge enthielt (also weder `IN`, `IB`, noch `IW`), so wird die Operandenlänge automatisch festgelegt. Ein eventuell erforderlicher `DDIR`-Präfix wird automatisch eingefügt. Ein einzelnes

```
        LD      BC,12345678h
```

wird also automatisch um ein führendes `DDIR IW` ergänzt. In diesem Fall jedoch:

```
        DDIR    LW
        LD      BC,12345678h
```

wird der `DDIR`-Präfix automatisch um ein `IW` ergänzt, effektiv wird also der Code für

```
        DDIR    LW,IW
        LD      BC,12345678h
```

erzeugt. Der im ersten Schritt erzeugte Code für `DDIR LW` wird verworfen, was an einem `R` im Listing zu erkennen ist.

Explizite `DDIR`-Präfixe mit einem `IN`, `IB`, oder `IW` können genutzt werden, um eine bestimmte Operandenlänge zu erzwingen. Zu lange Operanden führen zu einer Fehlermeldung, kürzere werden mit Nullen auf die geforderte Länge gebracht. Ein

```
        DDIR    IB
        LD      BC,12345678h
```

führt also zu einer Fehlermeldung, weil die Konstante nicht in 24 Bit hineinpasst. Umgekehrt wird hier

```
        DDIR    LW,IW
        LD      BC,345678h
```

die Konstante mit einem Byte mehr kodiert, als eigentlich erforderlich wäre.

4.32 Z8, Super8 und eZ8

Der Prozessorkern der Z8-Mikrokontroller beinhaltet keine eigenen Register. Stattdessen kann ein 16er-Block des internen Adreßraums aus RAM und I/O-Registern als 'Arbeitsregister' benutzt werden, die mit 4-Bit-Adressen angesprochen werden können. Welcher 16er-Block als Arbeitsregister benutzt werden soll, wird mit den RP-Registern festgelegt: Bits 4 bis 7 von RP definieren beim klassischen Z8 den 'Offset', der auf die 4-Bit-Arbeitsregisteradresse addiert wird, um eine 8-Bit-Adresse zu erhalten. Beim Super8-Kern existieren zwei RP-Register (RP0 und RP1), die es erlauben, obere und untere Hälfte der Arbeitsregister an getrennte Stellen zu legen.

Üblicherweise werden die Arbeitsregister in der Assemblersyntax als Register R0...R15 angesprochen, man kann diese Arbeitsregister aber auch als eine Methode zur effizienteren (kürzeren) Adressierung eines 16er-Bocks im internen RAM betrachten.

Mit dem `ASSUME`-Befehl teilt man AS den aktuellen Wert von RP mit. AS ist dann in der Lage, bei einer Adresse aus dem internen RAM automatisch zu entscheiden, ob dieser Operand mit einer 4-Bit Adresse angesprochen werden kann oder eine 8-Bit-Adresse verwendet werden muß. Man kann diese Funktion auch dazu benutzen, Arbeitsregistern symbolische Namen zu verpassen:

```
op1     equ     040h
op2     equ     041h

        srp     #040h
        assume  rp:040h

        ld      op1,op2  ; entspricht ld r0,r1
```

Es ist auch auf dem Super8 möglich, RP als Argument von `ASSUME` anzugeben, obwohl dieser kein RP-Register hat (nur RP0 und RP1). In diesem Fall werden die angenommen Werte von RP0 und RP1 auf *wert* bzw. $wert + 8$ gesetzt, analog zum `SRP` Maschinenbefehl auf dem Super8- Kern.

Im Gegensatz zum Original Zilog-Assembler ist es nicht erforderlich, eine 'Arbeitsregisteradressierung' explizit durch ein vorangestelltes Ausrufezeichen anzufordern, wobei AS diese Syntax nichtsdestotrotz versteht - ein vorangestelltes Ausrufezeichen erzwingt quasi 4-Bit-Adressierung, auch wenn die Adresse eigentlich nicht im durch RP festgelegten 16-Bit-Fenster liegt (dann wird eine Warnung ausgegeben). Umgekehrt ist es durch ein vorangestelltes >-Zeichen möglich, eine Adressierung mit 8 Bit zu erzwingen, auch wenn die Adresse eigentlich im aktuellen 16er-Fenster liegt.

Beim eZ8 wird das Spielchen quasi eine Stufe weiter getrieben: der interne Daten-Adreßbereich ist jetzt 12 statt 8 Bit groß. Um kompatibel zum alten Z8-Kern zu sein, hat Zilog die zusätzlichen Banking-Bits in den *unteren* vier Bits von RP untergebracht - ein RP-Wert von 12h definiert also das 16er-Adreßfenster von 210h bis 21fh.

Die unteren vier Bits von RP definieren beim eZ8 gleichzeitig das 256er-Fenster, das man mit 8-Bit-Adressen erreichen kann - hier gilt ein analoger Mechanismus, der dafür sorgt, daß AS automatisch 12- oder 8-Bit-Adressen verwendet. 'Lange' 12-Bit-Adressen kann man mit zwei vorangestellten >-Zeichen erzwingen.

4.33 Z8000

Der Z8001/8003 kann in zwei verschiedenen Modi betrieben werden:

- *Nicht segmentiert*: Der Adreßraum ist auf 64 KByte beschränkt, alle Adressen sind 'einfache' lineare 16-Bit- Adressen. Adreßregister sind einfache 16-Bit-Register (Rn), und absolute Adressen in Befehlen sind ein 16-Bit-Wort lang.
- *Segmentiert*: Der Speicher ist in bis zu 128 Segmente von jeweils maximal 64 KByte aufgeteilt. Adressen bestehen aus einem 7-bittigen Segment und einem 16-bittigen Offset. Adreßregister sind immer Registerpaare (RRn). Absolute Adressen in Befehlen sind zwei 16-Bit-Worte lang, außer der Offset is kleiner 256.

Die Betriebsart (segmentiert oder nicht segmentiert) hat also einen Einfluß auf den erzeugten Code und wird implizit über den verwendeten Prozessortyp umgeschaltet. Ist das Ziel also z.B. ein Z8001 im nicht segmentierten Modus, so wählt man einfach Z8002 als Ziel.

Eine 'echte' Unterstützung eines segmentierten Speichermodells bietet AS indes für den Z8000 genauso wenig wie für den 8086. Im segmentierten Modus wird die Segmentnummer einfach als die oberen sieben Adreßbits eines gedacht linearen 8MB-Adreßraums behandelt. Dies ist eigentlich nicht im Sinne des Erfinders, aber es entspricht der Art und Weise, wie der Z8001 in Systemen ohne MMU effektiv betrieben wurde.

Generell implementiert AS die Assembler-Syntax der Z8000-Maschinenbefehle so, wie es von Zilog in der Dokumentation vorgesehen ist. Es existieren jedoch Assembler, die Erweiterungen bzw. Variationen unterstützen. AS implementiert davon folgendes:

4.33.1 Bedingungen

Zusätzlich zu den von Zilog definierten Bedingungen sind folgende alternative Namen definiert:

Alternativ	Zilog	Bedeutung
ZR	Z	Z = 1
CY	C	C = 1
LLE	ULE	(C OR Z) = 1
LGE	UGE	C = 0
LGT	UGT	((C = 0) AND (Z = 0)) = 1
LLT	ULT	C = 1

4.33.2 Flags

Als Argument für die Befehle `SETFLG`, `COMFLG` und `RESFLG` werden auch die folgenden alternativen Namen akzeptiert:

Alternativ	Zilog	Bedeutung
ZR	Z	Zero-Flag
CY	C	Carry-Flag

4.33.3 Indirekte Adressierung

Anstelle `@Rn` darf auch `Rn^` geschrieben werden, falls beim `CPU`- Statement die Option `AMDSyntax=1` gesetzt wurde. Wird eine I/O-Adresse indirekt adressiert, so reicht es mit dieser Option auch aus, *nur* `Rn` zu schreiben.

4.33.4 Direkte versus unmittelbare Adressierung

Bei der von Zilog vorgegebenen Assembler-Syntax muß unmittelbare Adressierung durch ein vorangestelltes Doppelkreuz kenntlich gemacht werden. Wurde dem `CPU`-Statement jedoch die Option `AMDSyntax=1` mitgegeben, wird anhand des Arguments (Label oder Konstante) entschieden, ob direkte oder unmittelbare Adressierung verwendet werden soll. Unmittelbare Adressierung kann erzwungen werden, indem dem Argument ein Circumflex vorangestellt wird, z.B. um die Adresse eines Labels in ein Register zu laden.

4.34 TLCS-900(L)

Diese Prozessoren können in zwei Betriebsarten laufen, einmal im *Minimum*-Modus, der weitgehende Z80- und TLCS-90-Quellcodekompatibilität bietet, und zum anderen im *Maximum*-Modus, in dem der Prozessor erst seine wahren Qualitäten entfaltet. Die Hauptunterschiede zwischen den beiden Betriebsarten sind:

- Breite der Register WA,BC,DE und HL: 16 oder 32 Bit;
- Anzahl der Registerbanks: 8 oder 4;
- Programmadreßraum: 64 Kbyte oder 16 Mbyte;
- Breite von Rücksprungadressen: 16 oder 32 Bit.

Damit AS gegen die richtigen Grenzen prüfen kann, muß man ihm zu Anfang mit dem Befehl `MAXMODE` (siehe dort) mitteilen, in welcher Betriebsart der Code ausgeführt werden wird; Voreinstellung ist der Minimum-Modus.

Je nach Betriebsart müssen demzufolge auch die 16- oder 32-Bit-Versionen der Bankregister zur Adressierung verwendet werden, d.h. WA, BC, DE und HL im Minimum-Modus sowie XWA, XBC, XDE und XHL im Maximum-Modus. Die Register XIX..XIZ und XSP sind **immer** 32 Bit breit und müssen zur Adressierung auch immer in dieser Form verwendet werden; hier muß bestehender Z80-Code also auf jeden Fall angepaßt werden (neben der Tatsache, daß es gar keinen I/O-Adreßraum mehr gibt und alle I/O-Register memory-mapped sind...).

Absolute Adressen sowie Displacements können in unterschiedlichen Längen kodiert werden. AS wird ohne explizite Angaben immer versuchen, die kürzestmögliche Schreibweise zu verwenden; dies schließt ein, daß ein Displacement von Null überhaupt nicht im Code erscheint und aus einen `(XIX+0)` einfach ein `(XIX)` wird. Ist eine bestimmte Länge erwünscht, so kann sie durch Anhängen eines passenden Suffixes (:8, :16, :24) an das Displacement bzw. die Adresse erreicht werden.

Die von Toshiba gewählte Syntax für Registernamen ist in der Hinsicht etwas unglücklich, als daß zur Anwahl der vorherigen Registerbank ein Hochkomma (') benutzt wird. Dieses Zeichen wird von den prozessorunabhängigen Teilen von AS bereits zur Kennzeichnung von Zeichenkonstanten benutzt. Im Befehl

```
        ld      wa',wa
```

erkennt AS z.B. nicht das Komma zur Parametertrennung. Dieses Problem kann man aber umgehen, indem man ein umgekehrtes Hochkomma (‘) verwendet, z.B.

```
        ld      wa‘,wa
```

Toshiba liefert für die TLCS-900-Reihe selber einen Assembler (TAS900), der sich in einigen Punkten von AS unterscheidet:

Symbolkonventionen

- TAS900 unterscheidet Symbolnamen nur auf den ersten 32 Zeichen. AS dagegen speichert Symbolnamen immer in der vollen Länge (bis 255 Zeichen) und unterscheidet auch auf dieser Länge.

- Unter TAS900 können Integer-Konstanten sowohl in C-Notation (mit vorangestellter 0 für oktal bzw. 0x für hexadezimal) als auch in normaler Intel-Notation geschrieben werden. AS unterstützt in der Default-Einstellung **nur** die Intel-Notation. Mit dem `RELAXED`-Befehl bekommt man (unter anderem) auch die C-Notation.

- AS macht keinen Unterschied zwischen Groß- und Kleinschreibung, TAS900 hingegen unterscheidet Groß-und Kleinbuchstaben in Symbolnamen. Dieses Verhalten erhält man bei AS erst, wenn man die `-u`-Kommandozeilenoption benutzt.

Syntax

AS ist bei vielen Befehlen in der Syntaxprüfung weniger streng als TAS900, bei einigen weicht er (sehr) geringfügig ab. Diese Erweiterungen bzw. Änderungen dienen teilweise der leichteren Portierung von bestehendem Z80-Code, teilweise einer Schreiberleichterung und teilweise einer besseren Orthogonalität der Assemblersyntax:

- Bei den Befehlen `LDA, JP` und `CALL` verlangt TAS, daß Adreßausdrücke wie `XIX+5` nicht geklammert sein dürfen, wie es sonst üblich ist. AS verlangt im Sinne der Orthogonalität für `LDA` dagegen immer eine Klammerung, bei `JP` und `CALL` ist sie dagegen für einfache, absolute Adressen optional.

- Bei den bedingten Befehlen `JP, CALL, JR` und `SCC` stellt AS es dem Programmierer frei, die Default-Bedingung `T` (= true) als ersten Parameter anzugeben oder nicht. TAS900 hingegen erlaubt es nur, die Default-Bedingung implizit zu benutzen (also z.B. `jp (xix+5)` anstelle von `jp t,(xix+5)`).

- AS erlaubt beim `EX`-Befehl auch Operandenkombinationen, die zwar nicht direkt im User's Manual[178] genannt werden, aber durch Vertauschung auf eine genannte zurück geführt werden können. Kombinationen wie `EX f',f` oder `EX wa,(xhl)` werden damit möglich. TAS900 hingegen läßt nur die „reine Lehre" zu.

- AS erlaubt, bei den Befehlen `INC` und `DEC` die Angabe des Inkrements oder Dekrements wegzulassen, wenn dies 1 ist. Unter TAS900 dagegen muß auch eine 1 hingeschrieben werden.
- Ähnlich verhält es sich bei allen Schiebebefehlen: Ist der zu verschiebende Operand ein Register, so verlangt TAS900, daß auch eine Schiebeamplitude von 1 ausgeschrieben werden muß; ist dagegen eine Speicherstelle der Operand, so ist die Schiebezahl (hardwarebedingt) immer 1 und darf auch nicht hingeschrieben werden. Unter AS dagegen ist die Schiebezahl 1 immer optional und auch für alle Operandentypen zulässig.

Makroprozessor

Der Makroprozessor wird TAS900 als externes Programm vorgeschaltet und besteht aus zwei Komponenten: einem C-artigen Präprozessor und einer speziellen Makrosprache (MPL), die an höhere Programmiersprachen erinnert. Der Makroprozessor von AS dagegen orientiert sich an ,,klassischen" Makroassemblern wie dem M80 oder MASM (beides Programme von Microsoft). Er ist fester Bestandteil des Programms.

Ausgabeformat

TAS900 erzeugt relokatiblen Code, so daß sich mehrere, getrennt assemblierte Teile zu einem Programm zusammenbinden lassen. AS hingegen erzeugt direkt absoluten Maschinencode, der nicht linkbar ist. An eine Erweiterung ist (vorläufig) nicht gedacht.

Pseudoanweisungen

Bedingt durch den fehlenden Linker fehlen in AS eine ganze Reihe von für relokatiblen Code erforderlichen Pseudoanweisungen, die TAS900 implementiert. In gleicher Weise wie bei TAS900 sind folgende Anweisungen vorhanden:

```
EQU, DB, DW, ORG, ALIGN, END, TITLE, SAVE, RESTORE,
```

wobei die beiden letzteren einen erweiterten Funktionsumfang haben. Einige weitere TAS900-Pseudobefehle lassen sich durch äquivalente AS-Befehle ersetzen (siehe Tabelle 4.7).

Von Toshiba existieren zwei Versionen des Prozessorkerns, wobei die L-Variante eine ,,Sparversion" darstellt. Zwischen TLCS-900 und TLCS-900L macht AS folgende Unterschiede:

- Die Befehle `MAX` und `NORMAL` sind für die L-Version nicht erlaubt, der `MIN`-Befehl ist für die Vollversion gesperrt.
- Die L-Version kennt den Normal-Stapelzeiger `XNSP/NSP` nicht, dafür das Steuerregister `INTNEST`.

Die Befehle `SUPMODE` und `MAXMODE` werden nicht beeinflußt, ebenso nicht deren initiale Einstellung OFF. Die Tatsache, daß die L-Version im Maximum-Modus startet und keinen Normal-Modus kennt, muß also vom Programmierer berücksichtigt werden. AS zeigt sich jedoch insofern kulant gegenüber der L-Variante, als daß Warnungen wegen privilegierter Anweisungen im L-Modus unterdrückt werden.

TAS900	AS	Bedeutung/Funktion
`DL` < *Daten* >	`DD` < *Daten* >	Speicher in Langworten belegen
`DSB` < *Zahl* >	`DB` < *Zahl* > `DUP` (?)	Speicher byteweise reservieren
`DSW` < *Zahl* >	`DW` < *Zahl* > `DUP` (?)	Speicher wortweise reservieren
`DSD` < *Zahl* >	`DD` < *Zahl* > `DUP` (?)	Speicher langwortweise reservieren
`$MIN[IMUM]`	`MAXMODE OFF`	folgender Code im Minimum-Modus
`$MAX[IMUM]`	`MAXMODE ON`	folgender Code im Maximum-Modus
`$SYS[TEM]`	`SUPMODE ON`	folgender Code im System-Modus
`$NOR[MAL]`	`SUPMODE OFF`	folgender Code im User-Modus
`$NOLIST`	`LISTING OFF`	Assembler-Listing ausschalten
`$LIST`	`LISTING ON`	Assembler-Listing einschalten
`$EJECT`	`NEWPAGE`	neue Seite im Listing beginnen

Tabelle 4.7: äquivalente Befehle TAS900↔AS

4.35 TLCS-90

Vielleicht fragt sich der eine oder andere, ob bei mir die Reihenfolge durcheinander gekommen ist, es gab ja von Toshiba zuerst den 90er als ,,aufgebohrten Z80" und danach den 900er als 16-Bit-Version. Nun, ich bin einfach über den 900er zum 90er gekommen (Danke, Oliver!). Die beiden Familien sind sich sehr artverwandt, nicht nur was ihre Syntax angeht, sondern auch ihre Architektur. Die Hinweise für den 90er sind daher eine Untermenge derer für den 900er: Da Schieben, Inkrementieren und Dekrementieren hier nur um eins möglich sind, braucht und darf diese Eins auch nicht als erstes Argument hingeschrieben werden. Bei den Befehlen `LDA, JP` und `CALL` möchte Toshiba wieder die Klammern um Speicheroperanden weglassen,

bei AS müssen sie aber aus Gründen der Orthogonalität gesetzt werden (der tiefere Grund ist natürlich, daß ich mir damit eine Sonderabfrage im Parser gespart habe, aber das sagt man nicht so laut).

Die TLCS-90er besitzen bereits prinzipiell einen Adreßraum von 1 Mbyte, dieser Raum erschließt sich aber nur bei Datenzugriffen über die Indexregister. AS verzichtet daher auf eine Berücksichtigung der Bankregister und begrenzt den Adreßraum für Code auf 64 Kbyte. Da der Bereich jenseits aber sowieso nur über indirekte Adressierung erreichbar ist, sollte dies keine allzu große Einschränkung darstellen.

4.36 TLCS-870

Schon wieder Toshiba...diese Firma ist im Augenblick wirklich sehr produktiv! Speziell dieser Sproß der Familie (Toshibas Mikrokontroller sind sich ja alle in Binärkodierung und Programmiermodell recht ähnlich) scheint auf den 8051-Markt abzuzielen: Die Methode, Bitstellen durch einen Punkt getrennt an den Adreßausdruck anzuhängen, hatte ja beim 8051 ihren Ursprung, führt jetzt aber auch genau zu den Problemen, die ich beim 8051 geahnt hatte: Der Punkt ist jetzt einerseits legales Zeichen in Symbolnamen, andererseits aber auch Teil der Adreßsyntax, d.h. AS muß Adresse und Bitstelle trennen und einzeln weiterverarbeiten. Diesen Interessenkonflikt habe ich vorerst so gelöst, daß der Ausdruck von **hinten** an nach Punkten durchsucht wird und so der letzte Punkt als Trenner gilt, eventuelle weitere Punkte werden dem Symbolnamen zugerechnet. Es gilt weiterhin die flehentliche Bitte, im eigenen Interesse auf Punkte in Symbolnamen zu verzichten, sie führen nur zu Verwirrungen:

```
LD      CF,A.7   ; Akku Bit 7 nach Carry
LD      C,A.7    ; Konstante A.7 nach Register C
```

4.37 TLCS-47

Mit dieser 4-Bit-Prozessorfamilie dürfte wohl das unter Ende dessen erreicht sein, was AS unterstützen kann. Neben dem `ASSUME`-Befehl für das Datenbankregister (siehe dort) ist eigentlich nur ein Detail erwähnenswert: im Daten- und I/O-Segment werden keine Bytes, sondern Nibbles reserviert (eben 4-Bitter...). Die Sache funktioniert ähnlich wie das Bitdatensegment beim 8051, wo ein `DB` ja nur einzelne Bit reserviert, nur daß es hier eben Nibbles sind.

Toshiba hat für diese Prozessorfamilie einen ,,erweiterten Befehlssatz" in Makroform definiert, um das Arbeiten mit diesem doch recht beschränkten Befehlssatz zu erleichtern. Im Fall von AS ist er in der Datei STDDEF47.INC definiert. Einige Befehle, deren makromäßige Realisierung nicht möglich war, sind allerdings ,,eingebaut" und stehen daher auch ohne die Include-Datei zur Verfügung:

- der `B`-Befehl, der die jeweils optimale Version des Sprungbefehls (`BSS, BS oder BSL`) automatisch wählt;
- `LD` in der Variante HL mit immediate;
- `ROLC` und `RORC` mit einer Schiebeamplitude >1.

4.38 TLCS-9000

Hier ist es zum ersten Mal passiert, daß ich einen Prozessor in AS implementiert habe, der zu diesem Zeitpunkt noch gar nicht auf dem Markt war. Toshiba hatte sich nach meinen Informationen leider zwischenzeitlich auch dazu entschieden, diesen Prozessor ,,auf Eis" zu legen, also auch kein Silizium geben. Das hatte natürlich zur Folge, daß dieser Teil

1. ein ,,Paper-Design" ist, d.h. noch nicht praktisch getestet wurde und

2. Die Unterlagen, die ich zum 9000er hatte [181], noch vorläufig waren, also noch nicht bis ins letzte Klarheit lieferten.

und dieses Target foratn in einen Dornröschenschlaf fiel...

...Schnitt, 20 Jahre später: auf einmal melden sich Leute bei mir, daß Toshiba wohl doch TLCS-9000-Chips an Kunden verkauft hat, und fragen nach den Unterlagen, weil sie Reverse-Engineering betreiben. Vielleicht bekommen wir ja auf diesem Wege noch das eine oder andere unklare Detail bestätigt oder geklärt. Fehler in diesem Teil sind also weiterhin noch möglich und werden natürlich bereinigt. Zumindest die Handvoll Beispiele in [181] werden aber richtig übersetzt.

Displacements im Maschinenbefehl selber können nur eine bestimmte maximale Länge (z.B. 13 oder 9 Bit) haben. Ist das Displacement länger, muß dem Befehl ein Präfix mit den "oberen Bits" vorangestellt werden. AS wird solche Präfixe automatisch nach Bedarf einsetzen, man kann jedoch auch mit einem dem Displacement vorangestellten `'>'` das Setzen eines Präfix erzwingen, z.B. so:

```
ld:g.b  (0h),0        ; kein Praefix
ld:g.b  (400000h),0   ; Praefix automatisch erzeugt
ld:g.b  (>0h),0       ; Praefix erzwungen
```

4.39 TC9331

Toshiba hat seinerzeit für diesen Prozessor einen (DOS-basierten) Assembler namens ASM31T geliefert. Dieser Assembler unterstützt eine Reihe von Syntax-Elementen, die sich auf AS nicht ohne Änderungen abbilden ließen, die die Kompatibilität zu existierenden Quelldateien für andere Targets gefährdert hätten. An folgenden Stellen werden möglicherweise Änderungen erforderlich sein, um für den ASM31T geschriebene Programme mit AS übersetzen zu können:

- ASM31T unterstützt C-artige Kommentare (`/* ... */`), die auch über mehrere Zeilen gehen dürfen. Diese werden von AS nicht unterstützt und müssen in mit einem Semikolon eingeleitete Kommentare umgesetzt werden.
- Wie ASM31T unterstützt AS für den TC9331 Kommentare in runden Klammern (`( ... )`), aber nur innerhalb eines Befehlsarguments. Enthält ein solcher Kommentar ein Komma, wird dieses Komma als Argument-Trenner behandelt und der Kommentar nicht beim Parsing der Argumente übersprungen.
- ASM31T erlaubt Symbol- und Label-Namen, die einen Bindestrich enthalten. Das ist bei AS nicht zugelassen, der Bindestrich ist hier der Subtraktionsoperator und in einem Ausdruck wie `end-start` wäre sonst nicht klar, ob ein einzelnes Symbol oder die Differenz von zwei Symbolen gemeint ist.
- ASM31T verlangt zwingend ein `END`-Statement am Ende des Programms; bei AS ist dies optional.

Des weiteren fehlen AS im Moment die Fähigkeiten, auf miteinander kollidierende Nutzungen von Funktionseinheiten in einem Befehl hinzuweisen. Die Dokumentation von Toshiba ist an diesem Punkt leider etwas schwer verständlich.

4.40 29xxx

Wie schon beim `ASSUME`-Befehl beschrieben, kann AS mit der Kenntnis über den Inhalt des RBP-Registers feststellen, ob im User-Modus auf gesperrte Register zugegriffen wird. Diese Fähigkeit beschränkt sich natürlich auf direkte Zugriffe (also nicht, wenn die Register IPA...IPC benutzt werden), und sie hat noch einen weiteren Haken: da lokale Register (also solche mit Nummern>127) relativ zum Stackpointer adressiert werden, die Bits in RBP sich aber immer auf absolute Nummern beziehen, wird die Prüfung für lokale Register NICHT durchgeführt. Eine Erweiterung auf lokale Register würde bedingen, daß AS zu jedem Zeitpunkt den absoluten Wert von SP kennt, und das würde spätestens bei rekursiven Unterprogrammen scheitern...

4.41 80C16x

Wie in der Erklärung des `ASSUME`-Befehls schon erläutert, versucht AS, dem Programmierer die Tatsache, daß der Prozessor mehr physikalischen als logischen Speicher hat, soweit als möglich zu verbergen. Beachten Sie aber, daß die DPP-Register **nur** Datenzugriffe betreffen und auch dort nur absolute Adressierung, also weder indirekte noch indizierte Zugriffe, da AS ja nicht wissen kann, wie die berechnete Adresse zur Laufzeit aussehen wird...Bei Codezugriffen arbeitet die Paging-Einheit leider nicht, man muß also explizit mit langen oder kurzen `CALL`s, `JMP`s oder `RET`s arbeiten. Zumindest bei den ,,universellen" Befehlen `CALL` und `JMP` wählt AS automatisch die kürzeste Form, aber spätestens beim `RET` sollte man wissen, woher der Aufruf kam. Prinzipiell verlangen `JMPS` und `CALLS` dabei, daß man Segment und Adresse getrennt angibt, AS ist jedoch so geschrieben, daß er eine Adresse selber zerlegen kann, z.B.

```
        jmps    12345h
```

anstelle von

```
        jmps    1,2345h
```

Leider sind nicht alle Effekte der chipinternen Instruktions-Pipeline versteckt: Werden CP (Registerbankadresse), SP (Stack) oder eines der Paging-Register verändert, so steht der neue Wert noch nicht für den nächsten Befehl zur Verfügung. AS versucht, solche Situationen zu erkennen und gibt im Falle eines Falles eine Warnung aus. Aber auch diese Mimik greift nur bei direkten Zugriffen.

Mit `BIT` definierte Bits werden intern in einem 13-Bit-Wort abgelegt, wobei die Bitadresse in Bit 4..11 liegt und die Bitnummer in den unteren vier Bits. Diese Anordnung erlaubt es, das nächsthöhere bzw. nächstniedrigere Bit durch Inkrementieren bzw. Dekrementieren anzusprechen. Bei expliziten Bitangaben mit Punkt funktioniert das aber nicht über Wortgrenzen hinaus. So erzeugt folgender Ausdruck eine Wertebereichsüberschreitung:

```
        bclr    r5.15+1
```

Hier muß ein `BIT` her:

```
msb     bit     r5.15
        .
        .
        .
        bclr    msb+1
```

Für den 80C167/165/163 ist der SFR-Bereich verdoppelt worden; daß ein Bit im zweiten Teil liegt, wird durch ein gesetztes Bit 12 vermerkt. Leider hatte Siemens bei der Definition des 80C166 nicht vorausgesehen, daß 256 SFRs (davon 128 bit-adressierbar) für Nachfolgechips nicht reichen würden. So wäre es unmöglich, den zweiten SFR-Bereich von F000H..F1DFH mit kurzen Adressen oder Bitbefehlen zu erreichen, hätten die Entwickler nicht einen Umschaltbefehl eingebaut:

```
EXTR    #n
```

Dieser Befehl bewirkt, daß für die nächsten `n` Befehle (0<`n`<5) anstelle des normalen der erweiterte SFR-Bereich angesprochen werden kann. AS erzeugt bei diesm Befehl nicht nur den passenden Code, sondern setzt intern ein Flag, daß für die nächsten `n` Befehle nur Zugriffe auf den erweiterten SFR-Bereich zuläßt. Da dürfen natürlich keine Sprünge dabei sein... Bits aus beiden Bereichen lassen sich natürlich jederzeit definieren, ebenso sind komplette Register aus beiden SFR-Bereichen jederzeit mit absoluter Adressierung erreichbar. Nur die kurze bzw. Bitadressierung geht immer nur abwechselnd, Zuwiderhandlungen werden mit einer Fehlermeldung geahndet.

Ähnlich sieht es mit den Präfixen für absolute bzw. indirekte Adressierung aus: Da aber sowohl Argument des Präfixes als auch der Adreßausdruck nicht immer zur Übersetzungszeit bestimmbar sind, sind die Prüfungsmöglichkeiten durch AS sehr eingeschränkt, weshalb er es auch bei Warnungen beläßt...im einzelnen sieht das folgendermaßen aus:

- feste Vorgabe einer 64K-Bank mittels `EXTS` oder `EXTSR`: Im Adreßausdruck werden direkt die unteren 16 Bit der Zieladresse eingesetzt. Haben sowohl Präfix als auch Befehl einen konstanten Operanden, so wird überprüft, ob Präfixargument und Bit 16..23 der Zieladresse identisch sind.

- feste Vorgabe einer 16K-Seite mittels `EXTP` oder `EXTPR`: Im Adreßausdruck werden direkt die unteren 14 Bit der Zieladresse eingesetzt. Bit 14 und 15 bleiben konstant 0, da sie in diesem Modus nicht vom Prozessor ausgewertet werden. Haben sowohl Präfix als auch Befehl einen konstanten Operanden, so wird überprüft, ob Präfixargument und Bit 14..23 der Zieladresse identisch sind.

Damit das etwas klarer wird, ein Beispiel (die DPP-Register haben die Reset-Vorbelegung) :

```
extp    #7,#1           ; Bereich von 112K..128K
mov     r0,1cdefh       ; ergibt Adresse 0defh im Code
mov     r0,1cdefh       ; -->Warnung
exts    #1,#1           ; Bereich von 64K..128K
mov     r0,1cdefh       ; ergibt Adresse 0cdefh im Code
mov     r0,1cdefh       ; -->Warnung
```

4.42 PIC16C5x/16C8x

Ähnlich wie die MCS-48-Familie teilen auch die PICs ihren Programmspeicher in mehrere Bänke auf, da im Opcode nicht genügend Platz für die vollständige Adresse war. AS verwendet für die Befehle `CALL` und `GOTO` die gleiche Automatik, d.h. setzt die PA-Bits im Statuswort entsprechend Start- und Zieladresse. Im Gegensatz zu den 48ern ist dieses Verfahren hier aber noch deutlich problematischer:

1. Die Befehle sind nicht mehr nur ein Wort, sondern bis zu drei Worten lang, können also nicht mehr in jedem Fall mit einem bedingten Sprung übergangen werden.

2. Es ist möglich, daß der Programmzähler beim normalen Programmfortgang eine Seitengrenze überschreitet. Die vom Assembler angenommene Belegung der PA-Bits stimmt dann nicht mehr mit der Realität überein.

Bei den Befehlen, die das Register W mit einem anderen Register verknüpfen, muß normalerweise als zweiter Parameter angegeben werden, ob das Ergebnis in W oder im Register abgelegt werden soll. Bei diesem Assembler ist es erlaubt, den zweiten Parameter wegzulassen. Welches Ziel dann angenommen werden soll, hängt vom Typ des Befehls ab: bei unären Operationen wird defaultmäßig das Ergebnis zurück ins Register gelegt. Diese Befehle sind:

`COMF, DECF, DECFSZ, INCF, INCFSZ, RLF, RRF` und `SWAPF`

Die anderen Befehle betrachten W defaultmäßig als Akkumulator, zu dem ein Register verknüpft wird:

`ADDWF, ANDWF, IORWF, MOVF, SUBWF` und `XORWF`

Die von Microchip vorgegebene Schreibweise für Literale ist ziemlich abstrus und erinnert an die auf IBM 360/370-Systemen übliche Schreibweise (Grüße aus Neandertal...). Um nicht noch einen Zweig in den Parser einfügen zu müssen, sind bei AS Konstanten in Motorola-Syntax zu schreiben (wahlweise auch Intel oder C im `RELAXED`-Modus).

Dem Assembler liegt die Include-Datei STDDEF16.INC bei, in der die Adressen der Hardware-Register und Statusbits verewigt sind. Daneben enthält sie eine Liste von ,,Befehlen", die der Microchip-Assembler als Makro implementiert. Bei der Benutzung dieser Befehlsmakros ist große Vorsicht angebracht, da sie mehrere Worte lang sind und sich somit nicht überspringen lassen!!

4.43 PIC17C4x

Für diese Prozessoren gelten im wesentlichen die gleichen Hinweise wie für ihre kleinen Brüder, mit zwei Ausnahmen: Die zugehörige Include-Datei enthält nur Registerdefinitionen, und die Probleme bei Sprungbefehlen sind deutlich kleiner. Aus der Reihe fällt nur `LCALL`, der einen 16-Bit-Sprung erlaubt. Dieser wird mit folgendem ,,Makro" übersetzt:

```
MOVLW   <Adr15..8>
MOWF    3
LCALL   <Adr0..7>
```

4.44 SX20/28

Durch die beschränkte Länge des Instruktionswortes ist es nicht möglich, darin eine vollständige Programmspeicher-Adresse (11 Bit) oder Datenspeicher-Adresse (8 Bit) unterzubringen. Der Prozessor ergänzt die gekürzten Adressen um die PA-Bits aus dem STATUS-Register bzw. oberen Bits aus dem FSR-Register. Über `ASSUME`-Befehle teilt man dem Assembler deren aktuellen Inhalt mit. Falls auf Adressen zugegriffen wird, die mit den vermerkten Werten nicht zugreifbar sind, erfolgt eine Warnung.

4.45 ST6

Diese Prozessoren können das Code-ROM seitenweise in den Datenbereich einblenden. Weil ich nicht die ganze Mimik des `ASSUME`-Befehles hier wiederkäuen möchte, verweise ich auf das entsprechende Kapitel (3.2.23), in dem steht, wie man mit diesem Befehl einigermaßen unfallfrei Konstanten aus dem ROM lesen kann.

Bei näherer Betrachtung des Befehlssatzes fallen einige eingebaute ,,Makros" auf. Die Befehle, die mir aufgefallen sind (es gibt aber vielleicht noch mehr...), sind in Tabelle 4.8 aufgelistet.

Insbesondere der letztere Fall verblüfft doch etwas... Leider fehlen aber einige Anweisungen wirklich. So gibt es z.B. zwar einen `AND`-Befehl, aber kein `OR`...von `XOR` gar nicht zu reden. In der Datei STDDEF62.INC finden sich deshalb neben den Adressen der SFRs noch einige Makros zur Abhilfe.

Der Original-Assembler AST6 von SGS-Thomson verwendet teilweise andere Pseudobefehle als AS. Außer der Tatsache, daß AS Pseudobefehle nicht mit einem vorangestellten Punkt kennzeichnet, sind folgende Befehle identisch:

Befehl	in Wirklichkeit
CLR A	SUB A,A
SLA A	ADD A,A
CLR adr	LDI adr,0
NOP	JRZ PC+1

Tabelle 4.8: versteckte Makros im ST6225-Befehlssatz

```
ASCII, ASCIZ, BLOCK, BYTE, END, ENDM, EQU, ERROR, MACRO,
ORG, TITLE, WARNING
```

Tabelle 4.9 zeigt die AST6-Befehle, zu denen analoge in AS existieren.

AST6	AS	Bedeutung/Funktion
.DISPLAY	MESSAGE	Meldung ausgeben
.EJECT	NEWPAGE	neue Seite im Listing
.ELSE	ELSEIF	bed. Assemblierung
.ENDC	ENDIF	bed. Assemblierung
.IFC	IF...	bed. Assemblierung
.INPUT	INCLUDE	Include-Datei einbinden
.LIST	LISTING, MACEXP_DFT	Listing-Einstellung
.PL	PAGE	Seitenlänge Listing
.ROMSIZE	CPU	Zielprozessor einstellen
.VERS	VERSION (Symbol)	Version abfragen
.SET	EVAL	Variablen neu setzen

Tabelle 4.9: äquivalente Befehle AST6↔AS

4.46 ST7

In [141] ist der '.w'-Postfix für 16-Bit-Adressen nur für speicherindirekte Operanden definiert, um zu vermerken, daß auf einer Zeropageadresse eine 16-bittige Adresse liegt; AS unterstützt ihn jedoch zusätzlich auch für absolute Adressen oder Displacements in indizierter Adressierung, um trotz eines nur 8 Bit langen Wertes (0..255) ein 16-bittiges Displacement zu erzeugen.

4.47 ST9

Die Bitadressierungsmöglichkeiten des ST9 sind relativ eingeschränkt: Mit Ausnahme des BTSET-Befehls ist es nur möglich, auf Bits innerhalb des aktuellen Arbeitsregistersatzes zuzugreifen. Eine Bit-Adresse sieht also folgendermaßen aus:

```
rn.[!]b
```

wobei ! eine optionale Invertierung eines Quelloperanden bedeutet. Wird ein Bit symbolisch mittels des BIT-Befehles definiert, so wird die Registernummer im Symbolwert in Bit 7..4, die Bitnummer in Bit 3..1 und eine optionale Invertierung in Bit 0 vermerkt. AS unterscheidet direkte und symbolische Bitangaben am Fehlen eines Punktes, der Name eines Bitsymboles darf also keinen Punkt enthalten, obwohl sie an sich zulässig wären. Es ist auch zulässig, bei der Referenzierung von Bitsymbolen diese zu nachträglich zu invertieren:

```
bit2    bit     r5.3
        .
        .
        bld r0.0,!bit2
```

Auf diese Weise ist es auch möglich, eine inverse Definition nachträglich wieder aufzuheben.

Bitdefinitionen finden sich in großer Zahl in der Include-Datei REGST9.INC, in der die Register- und Bitnamen aller On-Chip-Peripherie beschrieben sind. Beachten Sie jedoch, daß deren Nutzung nur möglich ist, wenn die Arbeitsregisterbank vorher auch auf diese Register ausgerichtet wurde!

Im Gegensatz zu der zum AST9 von SGS-Thomson gehörenden Definitionsdatei sind für AS die Namen der Peripherieregister nur als allgemeine Registernamen definiert (R...), nicht auch noch als Arbeitsregister (r...). Dies ist so, weil AS Geschwindigkeitsgründen keine Aliasnamen für Register definieren kann.

4.48 6804

Eigentlich habe ich diesen Prozessor ja nur eingebaut, um mich über das seltsame Gebaren von SGS-Thomson zu beklagen: Als ich das 6804-Datenbuch zum ersten Mal in die Hand bekam, fühlte ich mich ob des etwas ,,unvollständigen" Befehlssatzes und der eingebauten Makros spontan an die ST62-Serie vom gleichen Hersteller erinnert. Ein genauerer Vergleich der Opcodes förderte erstaunliches zu

Tage: Ein 6804-Opcode ergibt sich durch Spiegelung aller Bits im entsprechenden ST62-OpCode! Thomson hat hier also offensichtlich etwas Prozessorkern-Recycling betrieben...wogegen ja auch nichts einzuwenden wäre, wenn nicht so eine Verschleierungstaktik betrieben werden würde: andere Peripherie, Motorola- anstelle Zilog-Syntax sowie das häßliche Detail, in Opcodes enthaltene Argumente (z.B. Bitfelder mit Displacements) **nicht** zu drehen. Letzterer Punkt hat mich auch nach längerem Überlegen dazu bewogen, den 6804 doch in AS aufzunehmen. Ich wage übrigens keine Spekulationen, welche Abteilung bei Thomson von welcher abgekupfert hat...

Im Gegensatz zur ST62-Version enthält die Include-Datei für den 6804 keine Makros, die die Lücken im Befehlssatz etwas „auspolstern" sollen. Dies überlasse ich dem geneigten Leser als Fingerübung!

4.49 TMS3201x

Offensichtlich ist es Ehrgeiz jedes Prozessorherstellers, seine eigene Notation für Hexadezimalkonstanten zu erfinden. Texas Instruments war bei diesen Prozessoren besonders originell: ein vorangestelltes >-Zeichen! Die Übernahme dieses Formates in AS hätte zu schweren Konflikten mit den Vergleichs-und Schiebeoperatoren von AS im Formelparser geführt. Ich habe mich deshalb für die Intel-Notation entschieden, zu der sich TI bei der 340x0-Serie und den 3201x-Nachfolgern ja dann auch durchgerungen hat...

Leider hat das Instruktionswort dieser Prozessoren nicht genügend Bits, um bei direkter Adressierung alle 8 Bits zu enthalten, weshalb der Datenadreßraum logisch in 2 Bänke zu 128 Wörtern gespalten ist. AS verwaltet diesen als ein durchgehendes Segment von 256 Wörtern und löscht bei direkten Zugriffen automatisch das Bit 7 (Ausnahme: Befehl `SST`, der nur in die obere Bank schreiben kann). Der Programmierer ist dafür erforderlich, daß das Bank-Bit stets den richtigen Wert hat!

Ein weiterer, nur sehr versteckt im Datenbuch stehender Hinweis: Die `SUBC`-Anweisung benötigt zur Ausführung intern mehr als einen Takt, das Steuerwerk arbeitet jedoch schon an dem nächsten Befehl weiter. Im auf ein `SUBC` folgenden Befehl darf deshalb nicht auf den Akkumulator zugegriffen werden. AS nimmt hier **keine** Prüfung vor!

4.50 TMS320C2x

Da ich nicht selber diesen Codegenerator geschrieben habe (was nichts an seiner Qualität mindert), kann ich nur kurz hier umreißen, wieso es Befehle gibt, bei denen ein vorangestelltes Label als untypisiert, d.h. keinem Adreßraum zugeordnet,

gespeichert wird: Der 20er der TMS-Reihe kennt sowohl ein 64 Kbyte großes Code- als auch Datensegment. Je nach externer Beschaltung kann man dabei Code- und Datenbereiche überlappen, um z.B. Konstanten im Codebereich zu abzulegen und auf diese als Daten zuzugreifen (Ablage im Code ist notwendig, weil ältere AS-Versionen davon ausgehen, daß ein Datensegment aus RAM besteht, das in einem Standalone-System nach dem Einschalten keinen definierten Inhalt hat und verweigern in Segmenten außer Code deshalb die Ablage von Daten). Ohne dieses Feature würde AS nun jeden Zugriff auf die abgelegten Daten mit einer Warnung („Symbol aus falschem Segment") quittieren. Im einzelnen erzeugen folgende Pseudobefehle untypisierte Labels:

`BSS, STRING, RSTRING, BYTE, WORD , LONG, FLOAT`
`DOUBLE, EFLOAT, BFLOAT` und `TFLOAT`

Sollten doch einmal typisierte Labels gewünscht sein, so kann man sich behelfen, indem man das Label in eine getrennte Zeile vor dem Pseudobefehl schreibt. Umgekehrt kann man einen der anderen Pseudobefehle mit einem typenlosen Label versehen, indem man vor dem Befehl das Label mit

```
<Name>  EQU     $
```

definiert.

4.51 TMS320C3x/C4x

Die größten Magenschmerzen bei diesem Prozessor hat mir die Syntax paralleler Befehle bereitet, die auf zwei Zeilen verteilt werden, wobei beide Befehle an sich auch sequentiell ausgeführt werden können. Deshalb erzeugt AS zuerst den Code für die einzelne erste Operation, wenn er dann in der zweiten Zeile erkennt, daß eine parallele Anweisung vorliegt, wird der zuerst erzeugte Code durch den neuen ersetzt. Im Listing kann man dies daran erkennen, daß der Programmzähler nicht weiterläuft und in der zweiten Zeile anstelle eines Doppelpunktes ein `R` vor dem erzeugten Code steht.

Bezüglich der doppelten senkrechten Striche und ihrer Position in der Zeile ist man nicht ganz so flexibel wie beim TI-Assembler: Entweder man schreibt sie anstelle eines Labels (d.h. in der ersten Spalte oder mit einem angehängten Doppelpunkt, das ist aber nicht mehr TI-kompatibel...) oder direkt vor den zweiten Befehl ohne Leerzeichen, sonst bekommt der Zeilenparser von AS Probleme und hält die Striche für das Mnemonic.

4.52 TMS9900

Wie bei den meisten älteren Prozessorfamilien auch, hatte TI seinerzeit ein eigenes Format zur Schreibweise von Hexadezimal- und Binärkonstanten verwendet, anstelle deren AS die normale, heute auch bei TI gebräuchliche Intel-Notation verwendet.

Die TI-Syntax für Register erlaubt es, daß anstelle eines echten Namens (entweder `Rx` oder `WRx`) auch eine einfache Integer-Zahl zwischen 0 und 15 benutzt werden kann. Dies hat zwei Folgen:

- `R0...R15` bzw. `WR0..WR15` sind einfache, vordefinierte Integersymbole mit den Werten 0..15, und die Definition von Registeraliasen funktioniert über schlichte `EQUs`.
- Im Gegensatz zu einigen anderen Prozessoren kann ich nicht das zusätzliche AS-Feature anbieten, daß das Kennzeichen für absolute Adressierung (hier ein Klammeraffe) weggelassen werden darf. Da ein fehlendes Kennzeichen hier aber Registernummern (im Bereich 0 bis 15) bedeuten würde, war das hier nicht möglich.

Weiterhin wechselt TI mit der Registerbezeichnung zwischen `Rx` und `WRx`...vorerst ist beides zugelassen.

4.53 TMS70Cxx

Diese Prozessorreihe gehört noch zu den älteren, von TI entwickelten Reihen, und deswegen benutzt TI in ihren eigenen Assemblern noch die herstellereigene Syntax für hexadezimale und binäre Konstanten (vorangestelltes < bzw. ?). Da das in AS aber so nicht machbar ist, wird defaultmäßig die Intel-Syntax verwendet. Auf diese ist Texas bei den Nachfolgern dieser Familie, nämlich den 370ern auch umgestiegen. Beim genaueren Betrachten des Maschinenbefehlssatzes stellt man fest, daß ca. 80% der 7000er-Befehle binär aufwärtskompatibel sind, und auch die Assemblersyntax ist fast gleich - aber eben nur fast. Bei der Erweiterung des 7000er-Befehlssatzes hat TI nämlich auch gleich die Chance genutzt, die Syntax etwas zu vereinheitlichen und zu vereinfachen. Ich habe mich bemüht, einen Teil dieser Änderungen auch in die 7000er Syntax einfließen zu lassen:

- Anstelle eines Prozentzeichens zur Kennzeichnung von unmittelbarer Adressierung darf auch das allgemein bekanntere Doppelkreuz verwendet werden.

- Wenn bei den Befehlen `AND, BTJO, BTJZ, MOV, OR` und `XOR` eine Port-Adresse (`P...`) als Quelle oder Ziel benutzt wird, ist es nicht notwendig, die Mnemonic-Form mit explizit angehängtem `P` zu benutzen - die allgemeine Form reicht genauso aus.
- Der vorangestelle Klammeraffe für absolute oder B-indizierte Adressierung darf weggelassen werden.
- Anstelle des `CMPA`-Befehls darf auch einfach `CMP` mit `A` als Ziel benutzt werden.
- Anstelle `LDA` oder `STA` darf auch einfach der `MOV`-Befehl mit `A` als Ziel bzw. Quelle benutzt werden.
- Anstelle des `MOVD`-Befehls darf auch `MOVW` benutzt werden.
- Anstelle von `RETS` oder `RETI` darf auch verkürzt `RTS` bzw. `RTI` geschrieben werden.
- `TSTA` bzw. `TSTB` dürfen auch als `TST A` bzw. `TST B geschrieben werden.`
- `XCHB B` ist als Alias für `TSTB` zugelassen.

Wichtig - diese Varianten sind nur beim TMS70Cxx zugelassen - entsprechende 7000er-Varianten sind bei den 370ern *nicht* erlaubt!

4.54 TMS370xxx

Obwohl diese Prozessoren keine speziellen Befehle zur Bitmanipulation besitzen, wird mit Hilfe des Assemblers und des `DBIT`-Befehles (siehe dort) die Illusion erzeugt, als ob man einzelne Bits manipulieren würde. Dazu wird beim `DBIT`-Befehl eine Adresse mit einer Bitposition zusammengefaßt und in einem Symbol abgelegt, das man dann als Argument für die Pseudobefehle `SBIT0, SBIT1, CMPBIT, JBIT0` und `JBIT1` verwenden kann. Diese werden in die Befehle `OR, AND, XOR, BTJZ` und `BTJO` mit einer passenden Bitmaske übersetzt.

An diesen Bit-Symbolen ist überhaupt nichts geheimnisvolles, es handelt sich um schlichte Integerwerte, in deren unterer Hälfte die Speicheradresse und in deren oberer Hälfte die Bitstelle gespeichert wird. Man könnte sich seine Symbole also auch ohne weiteres selber basteln:

```
defbit  macro   name,bit,adr
name    equ     adr+(bit<<16)
        endm
```

aber mit dieser Schreibweise erreicht man nicht den `EQU`-artigen Stil, den Texas vorgegeben hat (d.h. das zu definierende Symbol steht anstelle eines Labels). ACHTUNG! Obwohl `DBIT` eine beliebige Adresse zuläßt, können für die Pseudobefehle nur die Adressen 0..255 und 1000h..10ffh verwendet werden, eine absolute Adressierungsart kennt der Prozessor an dieser Stelle nicht...

4.55 MSP430(X)

Der MSP430 wurde als RISC-Prozessor mit minimalem Stromverbrauch konzipiert. Aus diesem Grund ist der Satz von Befehlen, die der Prozessor in Hardware versteht, auf das absolut notwendige reduziert worden (da RISC-Prozessoren keinen Mikrocode besitzen, muß jeder Befehl mit zusätzlichem Silizium implementiert werden und erhöht so den Stromverbrauch). Eine Reihe von Befehlen, die bei anderen Prozessoren in Hardware gegossen wurden, werden beim MSP durch eine Emulation mit anderen Befehlen realisiert. Frühere Versionen von AS implementierten diese Befehle über Makros in der Datei `REGMSP.INC`. Wer diese Datei nicht einband, erhielt bei über der Hälfte der insgesamt von TI definierten Befehle Fehlermeldungen. Dies ist aktuell nicht mehr so, zusammen mit der Erweiterung auf den CPU430X-Befehlssatz werden die Instruktionen vom Assembler direkt implementiert. `REGMSP.INC` enthält nur noch die Adressen von I/O-Registern. Wer aus irgendwelchen Gründen die alten Makros braucht, findet sie jetzt in der Datei `EMULMSP.INC`.

Die emulierten Instruktionen decken auch einige Sonderfälle ab, die der TI-Assembler nicht beherrscht. So wird zum Beispiel

```
rlc  @r6+
```

automatisch in

```
addc @r6+,-2(r6)
```

umgesetzt.

4.56 TMS1000

Der erste Mikrocontroller der Welt nun endlich auch in AS - lange hat es gedauert, nun ist die Lücke geschlossen. Dieses Target hat aber einige Tücken, die in diesem Abschnitt kurz angesprochen werden sollen.

Zum einen ist der Befehlssatz dieser Controller teilweise über die ROM-Maske veränderbar, d.h. man kann die Funktion einiger Opcodes in Grenzen frei definieren. AS kennt nur die Befehle und deren Kodierungen, die in [171] als Default-Kodierungen beschrieben sind. Wer für eine spezielle Anwendung andere Befehle bzw. gleiche Befehle mit anderem Opcode hat, kann diese über Makros mit passende `DB`-Befehlen ändern.

Des weiteren ist zu beachten, daß Sprünge und Unterprogrammaufrufe nur die unteren 6 Bit der Zieladresse im Befehl selber beinhalten. Die oberen 4 bzw. 5 Bits kommen aus Page- bzw. Chapter-Registern, die vorher passend zu setzen sind. AS selber kann hier nicht überprüfen, ob die Register vom Programmierer korrekt gesetzt werden!. Zumindest für den Fall, daß man im gleichen Chapter bleibt, gibt es die Assembler-Pseudobefehle `CALLL` bzw. `BL`, die einen `LDP`- und einen `CALL/BR`-Befehl zusammenfassen (was angesichts des kanppen Programmspeichers eine bequeme, aber nicht immer effiziente Variante ist).

4.57 COP8

Leider Gottes hat sich auch National dazu entschieden, als Schreibweise für nichtdezimale Integer-Konstanten die von IBM-Großrechnern bekannte (und von mir nicht geliebte) Variante `X'...` zu benutzen. Das geht natürlich (wie immer) nicht. Zum Glück scheint der ASMCOP aber auch die C-Variante zuzulassen, und diese wurde deshalb der Default für die COPs...

4.58 SC/MP

Benutzt man beim SC/MP indirekte Adressierung mit Displacement, und ist das benutzte Basis- bzw. Pointer-Register nicht P0 bzw. PC, dann hat ein Displacement von -128 (80 hex) eine Sonderbedeutung: Anstelle eben dieses Werts wird der Inhalt des E-Registers als Displacement verwendet. Im 'klassischen' NS-Assembler muß der Programmierer um diesen Umstand wissen bzw. ihn explizit benutzen, z.B. so:

```
ereg   equ -128
       ld  ereg(p1)
```

Dies birgt aber das Risiko, daß sich der Wert -128 auch einmal 'versehentlich' als Teil einer Berechnung ergeben kann, und man sucht erst einmal, warum das Programm nicht tut, was erwartet wurde. Ich habe mich deshalb entschlossen, diesen Sonderwert etwas expliziter zu machen:

Wird ein Displacement von -128 verwendet, so erfolgt eine Warnung. Diese Warnung kann man ignorieren; will man sie los werden, so verwende man das eingebaute Literal `E`, das eben dieses Register referenziert:

```
        ld e(p1)
```

Da das SC/MP-Target Registersymbole unterstützt, ist es auch möglich, das 'eigene Symbol' passend zu definieren:

```
ereg    reg e
        ld  ereg(p1)
```

Dies sollte die Anzahl notwendiger Anpassungen in existierendem Code auf ein Minimum reduzieren.

4.59 SC144xxx

Original gab es für diese Reihe von DECT-Controllern mit relativ einfachem Befehlssatz nur einen sehr schlichten Assembler von National selber. Ein Assembler von IAR Systems ist angekündigt, aber noch nicht erhältlich. Da die Entwicklungstools von IAR allerdings auch nach Möglichkeit CPU-unabhängig angelegt sind, kann man anhand erhältlicher Zielplattformen in ungefähr abschätzen, wie dessen Pseudobefehle aussehen werden, und damit im Blick sind die (wenigen) SC144xx-spezifisch realisierten Befehle `DC, DC8, DW16, DS, DS8, DS16, DW` angelegt. Bei Befehlen, die bereits im AS-Kern angelegt sind, wollte ich natürlich nicht das Rad neu erfinden, deshalb hier eine Tabelle mit Äquivalenzen:

Die Befehle `ALIGN, END, ENDM, EXITM, MACRO, ORG, RADIX, SET` und `REPT` exisieren sowohl bei IAR als auch AS und haben gleiche Bedeutung. Bei folgenden Befehlen muß man umstellen:

Keine direkte Entsprechung gibt es für die Befehle `CASEON, CASEOFF, LOCAL, LSTPAG, #undef` und `REPTI`.

Ein direktes Äquivalent der Präprozessorbefehle ist natürlich nicht möglich, solange AS keinen C-artigen Präprozessor besitzt. C-artige Kommentare sind im Moment leider auch nicht möglich. Achtung: Wer IAR-Codes für AS umsetzt, muß die Präprozessorstatements nicht nur umwandeln, sondern auch aus Spalte 1 herausbewegen, da bei AS in Spalte 1 nur Labels stehen dürfen!

IAR	AS	Funktion
#include	include	Include-Datei einbinden
#define	SET, EQU	Symbole definieren
#elif, ELIF, ELSEIF	ELSEIF	Weiterer Zweig einer IF-Kette
#else, ELSE	ELSE	Letzter Zweig einer IF-Kette
#endif, ENDIF	ENDIF	Beendet eine IF-Kette
#error	ERROR, FATAL	Fehlermeldung erzeugen
#if, IF	IF	Beginn einer IF-Kette
#ifdef	IFDEF	Symbol definiert ?
#ifndef	IFNDEF	Symbol nicht definiert ?
#message	MESSAGE	Nachricht ausgeben
=, DEFINE, EQU	=, EQU	Feste Wertzuweisung
EVEN	ALIGN 2	Programmzähler gerade machen
COL, PAGSIZ	PAGE	Seitengröße für Listing setzen
ENDR	ENDM	Ende einer REPT-Struktur
LSTCND, LSTOUT	LISTING	Umfang des Listings steuern
LSTEXP, LSTREP	MACEXP	Expandierte Makros anzeigen?
LSTXRF	<Kommandozeile>	Querverweisliste erzeugen
PAGE	NEWPAGE	Neue Seite im Listing
REPTC	IRPC	Repetition mit Zeichenersetzung

4.60 NS32xxx

Wie es sich für CISC-Prozessoren gehört, verfügt diese Serie über ausgefeilte und komplexe Adressierungsarten. National definiert in ihren Manuals für jede davon die Assembler-Syntax, und AS setzt auch genau diese um. Wie bei fast jeder Architektur, für die Werkzeuge von Drittherstellern erstellt wurden, gibt es jedoch auch Abweichungen und Erweiterungen, und ein paar davon habe ich auch „mitgenommen”:

Die von National vorgesehene Methode, PC-relativ zu adressieren, ist folgende:

```
movb r0,*+disp
```

Das drückt natürlich sehr klar aus, was in diesem Fall passiert, will man aber einfach eine bestimmte Speicherstelle PC-relativ adressieren, dann muß man die Distanz selber berechnen:

```
movb r0,*+(addr-*)
```

Die erste Vereinfachung ist, daß es unter bestimmten Umständen auch ausreicht, einfach nur

```
movb r0,addr
```

zu schreiben - absolute Adressierung wird ja durch ein vorangestelltes -Zeichen kenntlich gemacht. Diese vereinfachte Schreibweise ist unter folgenden Bedingungen erlaubt:

- Immediate-Adressierung ist nicht erlaubt, z.B. weil es sich um einen Zieloperanden handelt, so daß keine Verwechslung damit möglich ist.
- Es wird mit einer Index-Erweiterung gearbeitet (in eckigen Klammern angehängt), die nicht mit Immediate-Adressierung kombinierbar ist.

Alternativ unterstützt AS für PC-relative Addressierung immer die Schreibweise

```
movb r0,addr(pc)
```

Hier wird - analog zum 68000 - die Distanz ebenfalls automatisch berechnet.

Für den External-Mode, der in der National-Syntax folgendermaßen geschrieben wird:

```
movb r0,ext(disp1)+disp2
```

wird auch die Variante

```
movb r0,disp2(disp1(ext))
```

unterstützt, die wohl im UNIX-Umfeld gängig war.

4.61 CR16

Als National die CR16-Architektur mit dem CR16B auf 2 MByte Adreßraum erweiterte, war die Kompatibilität zum Vorgänger CR16A ein Thema. CR16B hat deshalb ein 'Kompatibilitätsbit', in dem er vollständig binär aufwärtskompatibel ist. Der Preis dafür ist, daß diverse Sprungbefehle weiterhin nur die ersten 128 Kbyte des Adreßraumes erreichen können. Im „Large Model" ändert sich die binäre Kodierung dieser Befehle, so daß die vollen 2 MByte erreicht werden können. Ob man beim CR16B Code für das kleine oder große Speichermodell erzeugen will, wählt man zusammen mit der `CPU`-Anweisung:

```
cpu cr16b:model=0
```

wählt das kleine Modell,

```
cpu cr16b:model=1
```

wählt das große Modell. Letzteres ist auch der Default, wenn diese Option weggelassen wird.

4.62 uPD78(C)1x

Für relative, unbedingte Sprünge gibt es den `JR`-Befehl (Sprungdistanz -32...+31, 1 Byte) sowie den `JRE`-Befehl (Sprungdistanz -256...+255, 2 Bytes). AS kennt weiterhin den Pseudobefehl `J`, der automatisch den kürzestmöglichen Befehl benutzt.

Architektur und Befehlssatz dieser Prozessoren sind grob an den Intel 8080/8085 angelehnt - das gilt auch für die Mnemonics. Die Adressierungsart (direkt, indirekt, immediate) ist mit in das Mnemonic verpackt, und 16-Bit-Register (BC, DE, HL) werden wie beim 8080 mit einem Buchstaben abgekürzt. Da NEC in der Erklärung der einzelnen Adressierungsarten aber immer mal wieder die ausgeschriebenen Registernamen benutzt, und auch mal und mal nicht Klammern benutzt, um indirekte Adressierung anzudeuten, habe ich mich entschlossen, neben den 'offiziellen' Notationen aus dem NEC-Manual auch einige alternative Notationen zuzulassen. Einige nicht-NEC-Tools wie z.B. Disassembler scheinen solche Notationen ebenfalls zu benutzen:

- Anstelle `B` darf auch `BC`, `(B)` oder `(BC)` geschrieben werden.
- Anstelle `D` darf auch `DE`, `(D)` oder `(DE)` geschrieben werden.
- Anstelle `H` darf auch `HL`, `(H)` oder `(HL)` geschrieben werden.
- Anstelle `D+` darf auch `DE+`, `(D+)`, `(DE+)` oder `(DE)+` geschrieben werden.
- Anstelle `H+` darf auch `HL+`, `(H+)`, `(HL+)` oder `(HL)+` geschrieben werden.
- Anstelle `D-` darf auch `DE-`, `(D-)`, `(DE-)` oder `(DE)-` geschrieben werden.
- Anstelle `H-` darf auch `HL-`, `(H-)`, `(HL-)` oder `(HL)-` geschrieben werden.
- Anstelle `D++` darf auch `DE++`, `(D++)`, `(DE++)` oder `(DE)++` geschrieben werden.
- Anstelle `H++` darf auch `HL++`, `(H++)`, `(HL++)` oder `(HL)++` geschrieben werden.
- Anstelle `D--` darf auch `DE--`, `(D--)`, `(DE--)` oder `(DE)--` geschrieben werden.
- Anstelle `H--` darf auch `HL--`, `(H--)`, `(HL--)` oder `(HL)--` geschrieben werden.
- Anstelle `H+A` darf auch `HL+A`, `A+H`, `A+HL`, `(H+A)`, `(HL+A)`, `(A+H)` oder `(A+HL)` geschrieben werden.
- Anstelle `H+B` darf auch `HL+B`, `B+H`, `B+HL`, `(H+B)`, `(HL+B)`, `(B+H)` oder `(B+HL)` geschrieben werden.

- Anstelle `H+EA` darf auch `HL+EA`, `EA+H`, `EA+HL`, `(H+EA)`, `(HL+EA)`, `(EA+H)` oder `(EA+HL)` geschrieben werden.

Da Architektur und Befehlssatz so ,,8080-artig" sind, lag es auch nahe, den Schalter `Z80SYNTAX` zu unterstützen, so daß sich viele Befehle in einer eingängigeren und vertrauteren Art und Weise schreiben lassen. Da die Architektur und der Original-Befehlssatz der uCOM87-Familie sich jedoch in einigen Punkten von 8080 unterscheiden, ist hier keine eins-zu-eins Abbildung möglich: Nicht alle Originalbefehle haben ein ,,Z80-Äquivalent", und einge vom 8080/Z80 bekannten Befehle haben kein Pendant beim uCOM87. Deshalb ergibt ein `Z80SYNTAX EXCLUSIVE` hier auch keinen Sinn.

Die folgende Tabelle listet alle im `Z80SYNTAX`-Modus definierten Befehle und ihre Äquivalente in der Original-Syntax:

Z80SYNTAX	Original	Operation	CPUs
`LD r1,A`	`MOV r1,A`	r1←A	1,2,3,4 (r1≠EAx) 3,4 (r1=EAx)
`LD A,r1`	`MOV A,r1`	A←r1	1,2,3,4 (r1≠EAx) 3,4 (r1=EAx)
`LD sr,A`	`MOV sr,A`	sr←A	1,2,3,4
`LD A,sr1`	`MOV A,sr1`	A←sr1	1,2,3,4
`LD r,(word)`	`MOV r,word`	r←(word)	1,2,3,4 (r1≠V) 2,3,4 (r1=V)
`LD (word),r`	`MOV word,r`	(word)←r	1,2,3,4 (r1≠V) 2,3,4 (r1=V)
`LD r,byte`	`MVI r,byte`	r←byte	1,2,3,4 (r1≠V) 2,3,4 (r1=V)
`LD sr2,byte`	`MVI sr2,byte`	sr2←byte	2,3,4
`LD (wa),byte`	`MVIW wa,byte`	(wa)←byte	2,3,4
`LD (rpa1),byte`	`MVIX rpa1,byte`	(rpa1)←byte	2,3,4

Z80SYNTAX	Original	Operation	CPUs
LD (wa),a	STAW wa	(wa)←A	1,2,3,4
LD A,(wa)	LDAW wa	A←(wa)	1,2,3,4
LD (rpa),a	STAX rpa	(rpa)←(wa)	1,2
LD (rpa2),a	STAX rpa2	(rpa2)←(wa)	3,4
LD A,(rpa)	LDAX rpa	(wa)←(rpa)	1,2
LD A,(rpa2)	LDAX rpa2	(wa)←(rpa2)	3,4
LD rp3,EA	DMOV rp3,EA	rp3←EA	3,4
LD sr3,EA	DMOV sr3,EA	sr3←EA	3,4
LD EA,sr4	DMOV EA,sr4	EA←sr4	3,4
LD EA,rp3	DMOV EA,rp3	EA←rp3	3,4
LD (word),BC	SBCD word	(word)←BC	1,2,3,4
LD (word),DE	SDED word	(word)←DE	1,2,3,4
LD (word),HL	SHLD word	(word)←HL	1,2,3,4
LD (word),SP	SSPD word	(word)←SP	1,2,3,4
LD (rpa3),EA	STEAX rpa3	(rpa3)←EA	3,4
LD BC,(word)	LBCD word	BC←(word)	1,2,3,4
LD DE,(word)	LDED word	DE←(word)	1,2,3,4
LD HL,(word)	LHLD word	HL←(word)	1,2,3,4
LD SP,(word)	LSPD word	SP←(word)	1,2,3,4
LD EA,(rpa3)	LDEAX rpa3	EA←(word)	3,4
LD rp,word	LXI rp,word	rp←word	1,2,3,4
LD EA,word	LXI EA,word	EA←word	3,4
ADD A,(rpa)	ADDX rpa	A←A+(rpa)	1,2,3,4
ADD A,byte	ADI A,byte	A←A+byte	1,2,3,4
ADD r,byte	ADI r,byte	r←r+byte	2,3,4
ADD sr2,byte	ADI sr2,byte	sr2←A+sr2	2,3,4
ADD A,(wa)	ADDW wa	A←A+(wa)	2,3,4
ADD EA,r2	EADD EA,r2	EA←EA+r2	3,4
ADD EA,rp3	DADD EA,rp3	EA←EA+rp3	3,4
ADC A,(rpa)	ADCX rpa	A←A+(rpa)+CY	1,2,3,4
ADC A,byte	ACI A,byte	A←A+byte+CY	1,2,3,4
ADC r,byte	ACI r,byte	r←r+A+CY	2,3,4
ADC sr2,byte	ACI sr2,byte	sr2←A+sr2+CY	2,3,4
ADC A,(wa)	ADCW wa	A←A+(wa)+CY	2,3,4
ADC EA,rp3	DADC EA,rp3	EA←EA+rp3+CY	3,4
ADDNC A,(rpa)	ADDNCX rpa	A←A+(rpa) skip if !CY	1,2,3,4
ADDNC A,byte	ADINC A,byte	A←A+byte skip if !CY	1,2,3,4

Z80SYNTAX	Original	Operation	CPUs
ADDNC r,byte	ADINC r,byte	r←r+A skip if !CY	2,3,4
ADDNC sr2,byte	ADINC sr2,byte	sr2←A+sr2 skip if !CY	2,3,4
ADDNC A,(wa)	ADDNCW wa	A←A+(wa) skip if !CY	2,3,4
ADDNC EA,rp3	DADDNC EA,rp3	EA←EA+rp3 skip if !CY	3, 4
SUB A,(rpa)	SUBX rpa	A←A-(rpa)	1,2,3,4
SUB A,byte	SUI A,byte	A←A-byte	1,2,3,4
SUB r,byte	SUI r,byte	r←r-A	2,3,4
SUB sr2,byte	SUI sr2,byte	sr2←A-sr2	2,3,4
SUB A,(wa)	SUBW wa	A←A-(wa)	2,3,4
SUB EA,r2	ESUB EA,r2	EA←EA-r2	3,4
SUB EA,rp3	DSUB EA,rp3	EA←EA-rp3	3,4
SBB A,(rpa)	SBBX rpa	A←A-(rpa)-CY	1,2,3,4
SBB A,byte	SBI A,byte	A←A-byte-CY	1,2,3,4
SBB r,byte	SBI r,byte	r←r-A-CY	2,3,4
SBB sr2,byte	SBI sr2,byte	sr2←A-sr2-CY	2,3,4
SBB A,(wa)	SBBW wa	A←A-(wa)-CY	2,3,4
SBB EA,rp3	DSBB EA,rp3	EA←EA-rp3-CY	3,4
SUBNB A,(rpa)	SUBNBX rpa	A←A-(rpa) skip if !CY	1,2,3,4
SUBNB A,byte	SUINB A,byte	A←A-byte skip if !CY	1,2,3,4
SUBNB r,byte	SUINB r,byte	r←r-A skip if !CY	2,3,4
SUBNB sr2,byte	SUINB sr2,byte	sr2←A-sr2 skip if !CY	2,3,4
SUBNB A,(wa)	SUBNBW wa	A←A-(wa) skip if !CY	2,3,4
SUBNB EA,rp3	DSUBNB EA,rp3	EA←EA-rp3 skip if !CY	3,4
AND A,r	ANA A,r	A←A∧r	1,2,3,4 (r≠V) 2,3,4 (r=V)
AND r,A	ANA r,A	r←A∧r	1,2,3,4 (r≠V)

Z80SYNTAX	Original	Operation	CPUs
			2,3,4 (r=V)
AND A,(rpa)	ANAX rpa	A←A∧(rpa)	1,2,3,4
AND A,byte	ANI A,byte	A←A∧byte	1,2,3,4
AND r,byte	ANI r,byte	r←r∧byte	2,3,4
AND sr2,byte	ANI sr2,byte	sr2←sr2∧byte	1,2,3,4
AND A,(wa)	ANAW wa	A←A∧(wa)	2,3,4
AND (wa),byte	ANIW wa,byte	(wa)←(wa)∧byte	1,2,3,4
AND EA,rp3	DAN EA,rp3	EA←EA∧rp3	3,4
OR A,r	ORA A,r	A←A∨r	1,2,3,4 (r≠V) 2,3,4 (r=V)
OR r,A	ORA r,A	r←A∨r	1,2,3,4 (r≠V) 2,3,4 (r=V)
OR A,(rpa)	ORAX rpa	A←A∨(rpa)	1,2,3,4
OR A,byte	ORI A,byte	A←A∨byte	1,2,3,4
OR r,byte	ORI r,byte	r←r∨byte	2,3,4
OR sr2,byte	ORI sr2,byte	sr2←sr2∨byte	1,2,3,4
OR A,(wa)	ORAW wa	A←A∨(wa)	2,3,4
OR (wa),byte	ORIW wa,byte	(wa)←(wa)∨byte	1,2,3,4
OR EA,rp3	DOR EA,rp3	EA←EA∨rp3	3,4
XOR A,r	XRA A,r	A←A⊕r	1,2,3,4 (r≠V) 2,3,4 (r=V)
XOR r,A	XRA r,A	r←A⊕r	1,2,3,4 (r≠V) 2,3,4 (r=V)
XOR A,(rpa)	XRAX rpa	A←A⊕(rpa)	1,2,3,4
XOR A,byte	XRI A,byte	A←A⊕byte	2,3,4
XOR r,byte	XRI r,byte	r←r⊕byte	2,3,4
XOR sr2,byte	XRI sr2,byte	sr2←sr2⊕byte	2,3,4
XOR A,(wa)	XRAW wa	A←A⊕(wa)	2,3,4
XOR EA,rp3	DXR EA,rp3	EA←EA⊕rp3	3,4
SKGT A,r	GTA A,r	skip if A>r	1,2,3,4

Z80SYNTAX	Original	Operation	CPUs
			(r≠V)
			2,3,4
			(r=V)
SKGT r,A	GTA r,A	skip if r>A	1,2,3,4
			(r≠V)
			2,3,4
			(r=V)
SKGT A,(rpa)	GTAX rpa	skip if A>(rpa)	1,2,3,4
SKGT A,byte	GTI A,byte	skip if A>byte	1,2,3,4
SKGT r,byte	GTI r,byte	skip if r>byte	2,3,4
SKGT sr2,byte	GTI sr2,byte	skip if sr2>byte	2,3,4
SKGT A,(wa)	GTAW wa	skip if A>(wa)	2,3,4
SKGT (wa),byte	GTIW wa,byte	skip if (wa)>byte	1,2,3,4
SKGT EA,rp3	DGT EA,rp3	skip if EA>rp3	3,4
SKLT A,r	LTA A,r	skip if A<r	1,2,3,4
			(r≠V)
			2,3,4
			(r=V)
SKLT r,A	LTA r,A	skip if r<A	1,2,3,4
			(r≠V)
			2,3,4
			(r=V)
SKLT A,(rpa)	LTAX rpa	skip if A<(rpa)	1,2,3,4
SKLT A,byte	LTI A,byte	skip if A<byte	1,2,3,4
SKLT r,byte	LTI r,byte	skip if r<byte	2,3,4
SKLT sr2,byte	LTI sr2,byte	skip if sr2<byte	2,3,4
SKLT A,(wa)	LTAW wa	skip if A<(wa)	2,3,4
SKLT (wa),byte	LTIW wa,byte	skip if (wa)<byte	1,2,3,4
SKLT EA,rp3	DLT EA,rp3	skip if EA<rp3	3,4
SKNE A,r	NEA A,r	skip if A≠r	1,2,3,4
			(r≠V)
			2,3,4
			(r=V)
SKNE r,A	NEA r,A	skip if r≠A	1,2,3,4
			(r≠V)
			2,3,4
			(r=V)
SKNE A,(rpa)	NEAX rpa	skip if A≠(rpa)	1,2,3,4
SKNE A,byte	NEI A,byte	skip if A≠byte	1,2,3,4

Z80SYNTAX	Original	Operation	CPUs
SKNE r,byte	NEI r,byte	skip if r≠byte	2,3,4
SKNE sr2,byte	NEI sr2,byte	skip if sr2≠byte	2,3,4
SKNE A,(wa)	NEAW wa	skip if A≠(wa)	2,3,4
SKNE (wa),byte	NEIW wa,byte	skip if (wa)≠byte	1,2,3,4
SKNE EA,rp3	DNE EA,rp3	skip if EA≠rp3	3,4
SKEQ A,r	EQA A,r	skip if A=r	1,2,3,4 (r≠V) 2,3,4 (r=V)
SKEQ r,A	EQA r,A	skip if r=A	1,2,3,4 (r≠V) 2,3,4 (r=V)
SKEQ A,(rpa)	EQAX rpa	skip if A=(rpa)	1,2,3,4
SKEQ A,byte	EQI A,byte	skip if A=byte	1,2,3,4
SKEQ r,byte	EQI r,byte	skip if r=byte	2,3,4
SKEQ sr2,byte	EQI sr2,byte	skip if sr2=byte	2,3,4
SKEQ A,(wa)	EQAW wa	skip if A=(wa)	2,3,4
SKEQ (wa),byte	EQIW wa,byte	skip if (wa)=byte	1,2,3,4
SKEQ EA,rp3	DEQ EA,rp3	skip if AEA=rp3	3,4
SKON A,r	ONA A,r	skip if (A∧r)≠0	1,2,3,4 (r≠V) 2,3,4 (r=V)
SKON r,A	ONA r,A	skip if (r∧A)≠0	1,2,3,4 (r≠V) 2,3,4 (r=V)
SKON A,(rpa)	ONAX rpa	skip if (A∧(rpa))≠0	1,2,3,4
SKON A,byte	ONI A,byte	skip if (A∧byte)≠0	1,2,3,4
SKON r,byte	ONI r,byte	skip if (r∧byte)≠0	2,3,4
SKON sr2,byte	ONI sr2,byte	skip if (sr2∧byte)≠0	2,3,4
SKON A,(wa)	ONAW wa	skip if (A∧(wa))≠0	2,3,4
SKON (wa),byte	ONIW wa,byte	skip if (wa)∧byte≠0	1,2,3,4
SKON EA,rp3	DON EA,rp3	skip if (EA∧rp3)≠0	3,4
SKOFF A,r	OFFA A,r	skip if (A∧r)=0	1,2,3,4 (r≠V) 2,3,4 (r=V)

Z80SYNTAX	Original	Operation	CPUs
SKOFF r,A	OFFA r,A	skip if (r∧A)=0	1,2,3,4 (r≠V) 2,3,4 (r=V)
SKOFF A,(rpa)	OFFAX rpa	skip if (A∧(rpa))=0	1,2,3,4
SKOFF A,byte	OFFI A,byte	skip if (A∧byte)=0	1,2,3,4
SKOFF r,byte	OFFI r,byte	skip if (r∧byte)=0	2,3,4
SKOFF sr2,byte	OFFI sr2,byte	skip if (sr2∧byte)=0	2,3,4
SKOFF A,(wa)	OFFAW wa	skip if (A∧(wa))=0	2,3,4
SKOFF (wa),byte	OFFIW wa,byte	skip if (wa)∧byte=0	1,2,3,4
SKOFF EA,rp3	DOFF EA,rp3	skip if (EA∧rp3)=0	3,4
INC r2	INR r2	r2←r2+1	1,2,3,4
INC (wa)	INRW wa	(wa)←(wa)+1	1,2,3,4
INC rp	INX rp	rp←rp+1	1,2,3,4
DEC r2	DCR r2	r2←r2-1	1,2,3,4
DEC (wa)	DCRW wa	(wa)←(wa)-1	1,2,3,4
DEC rp	DCX rp	rp←rp-1	1,2,3,4
CPU-Gruppe 1: 78C05, 78C06 CPU-Gruppe 2: 7800, 7801, 7802 CPU-Gruppe 3: 7807, 7808, 7809, 7810 CPU-Gruppe 4: 7810, 78C1x			

Tabelle 4.10: Befehlsvarianten im Z80SYNTAX-Modus

4.63 75K0

Wie bei einigen anderen Prozessoren auch, kennt die Assemblersprache der 75er von NEC Pseudo-Bitoperanden, d.h. man kann einem Symbol eine Kombination aus Adresse und Bitnummer zuweisen, die dann bei bitorientierten Befehlen anstelle direkter Ausdrücke verwendet werden kann. Die drei folgenden Befehle erzeugen daher z.B. identischen Code:

```
ADM     sfr     0fd8h
SOC     bit     ADM.3

        skt     0fd8h.3
        skt     ADM.3
        skt     SOC
```

AS unterscheidet direkte und symbolische Bitzugriffe an einem bei Symbolen fehlenden Punkt; Punkte in Symbolnamen darf man daher nicht verwenden, da es sonst zu Mißverständnissen bei der Auflösung kommt.

Die Ablage von Bitsymbolen orientiert sich dabei weitgehend an der binären Kodierung, die die Prozessorhardware selber verwendet: Es werden 16 Bit belegt, und es existieren ein ,,kurzes" und ein ,,langes" Format. Das kurze Format kann folgende Varianten aufnehmen:

- direkte Zugriffe auf die Bereiche 0FBxH und 0FFxH
- indirekte Zugriffe der Form Adr.@L (0FC0H $\leq$ `Adr` $\leq$ 0FFFH)
- indirekte Zugriffe der Form @H+d4.bit

Das obere Byte ist auf 0 gesetzt, das untere Byte enthält den gemäß [110] kodierten Bitausdruck. Das lange Format kennt im Gegensatz dazu nur direkte Adressierung, kann dafür aber (korrekte Einstellungen von `MBS` und `MBE` vorausgesetzt) den ganzen Adreßraum abdecken. Bei langen Ausdrücken stehen im unteren Byte Bit 7..0 der Adresse, in Bit 8 und 9 die Bitstelle sowie in Bit 10 und 11 konstant 01. Letztere ermöglichen es, langes und kurzes Format einfach durch einen Vergleich des oberen Bytes gegen Null zu unterscheiden. Die Bits 12..15 enthalten Bit 8..11 der Adresse; sie werden zwar nicht zur Generierung des Kodes benötigt, müssen jedoch gespeichert werden, da eine Prüfung auf ein korrektes Banking erst bei der Verwendung des Symboles erfolgen kann.

4.64 78K0

NEC benutzt in seinen Datenbüchern zur Kennzeichnung der Zugriffsweise auf absolute Adressen verschiedene Schreibweisen:

- absolut kurz: kein Präfix
- absolut lang: vorangestelltes `!`
- PC-relativ: vorangestelltes `$`

Bei AS sind diese Präfixe nur notwendig, falls man eine bestimmte Adressierung erzwingen will und der Befehl verschiedene Varianten zuläßt. Setzt man keinen Präfix, so wählt AS automatisch die kürzeste Variante. Es dürfte daher in der Praxis sehr selten notwendig sein, einen Präfix zu verwenden.

4.65 78K2/78K3/78K4

Analog wie beim 78K0 benutzt NEC auch hier wieder Dollar- und Ausrufezeichen für verschiedene Längen von Adreßausdrücken. Zwischen langen und kurzen Adressen (sowohl im RAM- als auch SFR-Bereich) wird wieder automatisch entschieden, nur relative Adressierung muß man manuell anwählen, wenn ein Befehl beides unterstützt (z.B. `BR`).

Noch eine Anmerkung (die im übrigens auch für den 78K0 gilt): Wer mittels `RELAXED` mit Motorola-Syntax arbeitet, muß Hexadezimalkonstanten klammern, weil das führende Dollarzeichen u.U. als relative Adressierung mißverstanden wird...

4.66 uPD772x

Sowohl 7720 als auch 7725 werden von dem gleichen Codegenerator behandelt und sind sich in ihren Befehlssatz extrem ähnlich. Trotzdem sollte man sich nicht zu der Annahme verleiten lassen, sie seien binär kompatibel: Um die längeren Adreßfelder und zusätzlichen Befehle unterbringen zu können, haben sich die Bitpositionen einiger Felder im Instruktionswort verschoben, die Instruktionslänge hat sich auch insgesamt von 23 auf 24 Bit geändert. Im Code-Format sind deshalb auch unterschiedliche Header-Ids für beide reserviert.

Gemeinsam ist beiden, daß sie neben Code- und Datensegment auch noch ein ROM zur Ablage von Konstanten besitzen. Dieses ist bei AS auf das `ROMDATA`-Segment abgebildet!

4.67 uCOM-43

Der uCOM-43-Befehlssatz enthält einen Befehl `CZP`, der es erlaubt, einen Unterprogrammaufruf auf eine der Adressen 0, 4, 8, 12...60 mit einem einzigen Instruktions-Byte zu machen. Leider ist die vorhandene Dokumentation darüber unklar, ob der Befehl im Quelltext die Zieladresse (also einen durch vier teilbaren Wert) oder den im Befehl enthaltenen Vektor (also eine Zahl von 0 bis 15) erwartet. Im Moment betreibt der Assembler ein wenig 'Kaffeesatzleserei':

- Falls das Argument größer als 15 ist, muß es sich um eine Adresse handeln, ansonsten:

- Falls das Argument kein mehrfaches von vier ist, muß es sich um einen Vektor handeln, ansonsten:
- Falls das Argument Null ist, ist keine Unterscheidung notwendig, ansonsten:
- Falls das Argument ein Symbol aus dem `CODE`-Segment ist, z.B. ein Label, muß es sich um eine Adresse handlen, ansonsten:
- ist es ein Vektor.

Die letzten beiden Fälle treffen nur auf die Werte 4, 8 oder 12 zu, und weil die Entscheidung nicht intuitiv ist, wird bei der Entscheidung für einen Vektor eine Warnung ausgegeben. Für den Fall, daß wirklich die Adresse 4, 8 oder 12 gemeint ist, definiere man den Wert als Symbol im Code-Segment:

```
dest   label  4
       .
       .
       .
       czp dest
```

Für den Fall, daß ein Vektor gemeint ist, und die Warnung stört, kann man die Warnung folgender maßen unterdrücken:

```
vector equ    4
       .
       .
       .
       expect 480
       czp    vector
       endexpect
```

Ich gestehe ein, daß diese Situation nicht 100% befriedigend ist. Falls jemand Informationen über das Verhalten des Original-NEC-Assemblers hat, wäre ich für eine Nachricht dankbar.

4.68 F2MC16L

Darauf, daß man bei Anwendungen mit mehr als 64K ROM oder 64K RAM darauf achten sollte, AS die korrekte momentane Belegung der Bank-Register mitzuteilen, wurde bereits in Zusammenhang mit dem `ASSUME`-Befehl erwähnt. AS überprüft

bei jedem absoluten Zugriff anhand dieser Annahmen, ob evtl. ein Zugriff auf eine Speicherstelle erfolgt, die momentan überhaupt nicht greifbar ist. Standardmäßig sind dafür natürlich nur DTB und DPR wichtig, denn ADB bzw. SSB/USB werden nur bei indirekten Zugriffen über RW2/RW6 bzw. RW3/RW7 benutzt, und bei indirekten Zugriffen greift diese Prüfmimik ohnehin nicht. Nun ist es aber so, daß man - ähnlich wie beim 8086 - einem Befehl eine Segmentpräfix voranstellen kann, mit dem DTB für den folgenden Befehl durch ein beliebiges anderes Register ersetzt werden kann. AS führt deswegen über die verwendeten Präfixbefehle Buch und schaltet bei der Prüfung für den nächsten *Prozessorbefehl* um - eine zwischen dem Segmentpräfix und Prozessorbefehl eingestreute Pseudoanweisung löscht den Merker also *nicht*. Dies gilt auch für Pseudobefehle zur Datenablage oder Veränderung des Programmzählers - aber wer macht so etwas schon ;-)

4.69 MN161x

Für dieses Target gilt die Besonderheit, daß man zwischen zwei Code-Generatoren wählen kann: Einer wurde freundlicherweise von Haruo Asano geschrieben und ist über die CPU-Namen `MN1610` bzw. `MN1613` erreichbar; der andere wurde von mir erstellt und ist über die Namen `MN1610ALT` bzw. `MN1613ALT` aktivierbar. Wer den erweiterten Adreßraum von 256 KWorten des MN1613 verwenden will, oder mit dem Gleitkommaformat des MN1613 experimentieren will, muß das `ALT`- Target verwenden.

4.70 CDP180x

Prozessoren dieser Reihe kennen sowohl kurze als auch lange Sprungbefehle: Während ein kurzer Sprung nur innerhalb der gleichen Speicherseite von 256 Byte möglich ist, kann man mit einem langen Sprung jedes beliebige Ziel im Adreßraum von 64 KByte erreichen. Die Assembler-Syntax sieht für beide Varianten unterschiedliche Mnemonics vor (die lange Variante mit vorangestelltem 'L'), aber keine, die dem Assembler gestatten würde, automatisch einen langen oder kurzen Befehl einzusetzen. AS kennt solche 'Pseudobehle' als Erweiterung:

- `JMP` wird zu `BR` oder `LBR`.
- `JZ` wird zu `BZ` oder `LBZ`.
- `JNZ` wird zu `BNZ` oder `LBNZ`.

- JDF wird zu BDF oder LBDF.
- JPZ wird zu BPZ oder LBPZ.
- JGE wird zu BGE oder LBGE.
- JNF wird zu BNF oder LBNF.
- JM wird zu BM oder LBM.
- JL wird zu BL oder LBL.
- JQ wird zu BQ oder LBQ.
- JNQ wird zu BNQ oder LBNQ.

4.71 KENBAK

Der KENBAK-1 wurde 1970 entwickelt, zu einer Zeit, als der erste Mikroprozessor noch drei Jahre entfernt war. Man kann davon ausgehen, daß er für die Hobbyisten, die sich den Bausatz seinerzeit leisten konnten, ihr erster und einziger Computer war. Demzufolge hatten sie auch nichts, auf dem sie einen Assembler für diesen Computer hätten laufen lassen können - der KENBAK-1 mit seinem Speicher von 256 Byte war dafür viel zu klein. Die präferierte Methode waren vorgedruckte Tabellen, in die man die Befehle und ihren Maschinencode eintrug. War man mit dieser ,,Programmierung" fertig, konnte man den Code über die Schalterleiste per Hand in den Computer laden.

Daraus resultiert leider, daß die Assembler-Sprache des KENBAK zwar im Programming Manual beschrieben, aber nicht wirklich formal definiert ist. Als Grant Stockly vor ein paar Jahren neue KENBAK-Kits herausbrachte, hat er eine erste KENBAK-Portierung für meinen Assembler gemacht, die hat ihren Weg jedoch leider nie wieder "upstream" gefunden. In meiner Implementierung habe ich versucht, seine Ideen aufzugreifen, jedoch andererseits auch eine Syntax anzubieten, wie sie Programmierern eines 6502, Z80 oder ähnlichem eher vertraut sein dürfte. In der folgenden Tabelle sind die Syntax-Unterschiede gegenüber gestellt:

Stockly	Alternativ	Bemerkung
Arithmetisch/Logisch (ADD/SUB/LOAD/STORE/AND/OR/LNEG)		

Stockly	Alternativ	Bemerkung
instr `Constant`, *Reg*, *Wert*,	*instr Reg*, *#Wert*	immediate
instr `Memory`, *Reg*, *Addr*,	*instr Reg*, *Addr*	direkt
instr `Indirect`, *Reg*, *Addr*,	*instr Reg*, *(Addr)*	direkt
instr `Indexed`, *Reg*, *Addr*,	*instr Reg*, *Addr*,X	indiziert
instr `Indirect-Indexed`, *Reg*, *Addr*,	*instr Reg*, *(Addr)*,X	indirekt-indiziert
Sprünge		
`JPD` *Reg*, *Cond*, *Addr*	`JP` *Reg*, *Cond*, *Addr*	bedingt-direkt
`JPI` *Reg*, *Cond*, *Addr*	`JP` *Reg*, *Cond*, *(Addr)*	bedingt-indirekt
`JMD` *Reg*, *Cond*, *Addr*	`JM` *Reg*, *Cond*, *Addr*	bedingt-direkt
`JMI` *Reg*, *Cond*, *Addr*	`JM` *Reg*, *Cond*, *(Addr)*	bedingt-indirekt
`JPD Unconditional`, *Cond*, *Addr*	`JP` *Addr*	unbedingt-direkt
`JPI Unconditional`, *Cond*, *Addr*	`JP` *(Addr)*	unbedingt-indirekt
`JMD Unconditional`, *Cond*, *Addr*	`JM` *Addr*	unbedingt-direkt
`JMI Unconditional`, *Cond*, *Addr*	`JM` *(Addr)*	unbedingt-indirekt
Sprungbedingungen		
`Non-zero`	`NZ`	$\neq 0$
`Zero`	`Z`	$= 0$
`Negative`	`N`	< 0
`Positive`	`P`	≥ 0
`Positve-Non-zero`	`PNZ`	> 0
Skips		
`SKP 0`, *bit*, *Addr*	`SKP0` *bit*, *Addr [,Dest]*	
`SKP 1`, *bit*, *Addr*	`SKP1` *bit*, *Addr [,Dest]*	
Bitmanipulation		
`SET 0`, *bit*, *Addr*	`SET0` *bit*, *Addr*	
`SET 1`, *bit*, *Addr*	`SET1` *bit*, *Addr*	
Schiebe/Rotierbefehle		
`SHIFT LEFT`, *cnt*, *Reg*	`SFTL` *[cnt,] Reg*	
`SHIFT RIGHT`, *cnt*, *Reg*	`SFTR` *[cnt,] Reg*	arithm. Shift
`ROTATE LEFT`, *cnt*, *Reg*	`ROTL` *[cnt,] Reg*	
`ROTATE RIGHT`, *cnt*, *Reg*	`ROTR` *[cnt,] Reg*	

Tabelle 4.11: KENBAK-Befehlssyntax

Es gibt keinen Pseudobefehl, um zwischen diesen beiden Syntax-Varianten umzuschalten. Beide dürfen jederzeit und auch in beliebiger Mischung genutzt werden.

Die Zieladresse *[Dest]*, die man optional bei den Skip-Befehlen angeben kann, geht nicht in den erzeugten Code ein. Der Assembler überprüft lediglich, ob der Prozessor zur Laufzeit wirklich zur angegebene Adresse springen würde. Dadurch kann z.B. geprüft werden, ob man nicht versehentlich versucht, einen einzelnen Ein-Byte-Befehl zu überspringen. Ein weggelassenes Schiebeargument *[cnt]* bedeutet, daß um eine Stelle geschoben werden soll.

4.72 HP Nanoprocessor

Der HP Nanoprocessor verfügt über keinerlei Befehle, Daten aus dem ROM- Adreßraum zu lesen und in ein Register zu transferieren. Die diesbezüglichen Befehle `LDR` und `STR` sind eher das, was man bei anderen Prozessoren als ,,immediate-Adressierung" bezeichnen würde. Aus diesen Grund sind für den HP Nanoprocessor keinerlei Pseudobefehen definiert, mit denen sich Konstanten im ROM ablegen oder Platz reservieren läßt.

4.73 IM61x0

Dieser Mikroprozessor ist quasi eine Single-Chip-Implementierung einer PDP/8E, weshalb als ,,Assembler-Referenz" in Quellcode-Beispielen üblicherweise der PAL-III von Digital Equipment benutzt wird. Die AS-Implementierung weicht in einigen Punkten von der PAL-III-Syntax ab, unter anderem auch, weil sich manche Dinge nur mit großem Aufwand hätten bereit stellen lassen. Hier einige Hinweise, wie sich bestehender Code anpassen läßt:

- Bei PAL-III werden Label durch ein angehängtes Komma gekennzeichnet. Bei AS ist diese ein Doppelpunkt, bzw. gar nichts, wenn das Label in der ersten Spalte beginnt.
- Die Ablage von Daten im Speicher erfolgt bei PAL-III einfach dadurch, daß man die Konstante in das Feld des Mnemonics schreibt. AS verwendet zur Datenablage die `DC`-Anweisung, die dafür auch mehrere Worte auf einmal als Argument akzeptiert. Unter Umständen kann man sich die Ablage von Konstanten durch den `LTORG`-Mechanismus aber auch teilweise ersparen.
- Den Programmzähler setzt man mit `ORG address` anstelle *address.

Kapitel 5

Dateiformate

In diesem Kapitel sollen die Formate von von AS erzeugten Dateien beschrieben werden, deren Format sich nicht direkt erschließt.

5.1 Code-Dateien

Das vom Assembler ausgegebene Codedatenformat muß in der Lage sein, die Codeteile für unterschiedliche Prozessoren voneinander zu trennen, und sieht daher etwas anders aus als gängige Formate. Obwohl dem Assembler Tools zur Bearbeitung der Codedateien beiliegen, halte ich es für guten Stil, das Format hier kurz offenzulegen:

Sofern in der Datei Mehrbyte-Integers gespeichert sind, werden sie im Intelformat abgelegt, d.h. mit dem LSB zuerst. Diese Regel gilt bereits für das 16-Bit-Kennungswort mit dem Wert $1489, d.h. jede Codedatei beginnt mit den Bytes $89/$14.

Danach folgt eine Reihe beliebig vieler ,,Records", wobei ein Record entweder ein zusammenhängendes Teilfeld des Codes darstellt oder bestimmte Zusatzinformationen enthält. Eine Datei kann auch ohne Umschaltung des Prozessortyps mehrere Records enthalten, wenn Code- oder Konstantenbereiche durch reservierte (und nicht zu initialisierende) Speicherbereiche unterbrochen werden. Der Assembler versucht auf diese Weise, die Datei nicht länger als nötig werden zu lassen.

Allen Records ist gemein ist ein Header-Byte, das den Typ des Records und die damit folgenden Datenstrukturen festlegt. In einer Pascal-artigen Form läßt sich die Record-Struktur folgendermaßen beschreiben:

```
FileRecord = RECORD CASE RecordTypeOrFamilyHdr : UInt8 OF
               $00:(Creator:ARRAY[] OF Char);
               $01..
               $7f:(StartAddress : UInt32;
                    Length       : UInt16;
                    Data         : ARRAY[0..Length-1] OF UInt8);
               $80:(EntryAddress : UInt32);
               $81:(FamilyHdr    : UInt8;
                    Segment      : UInt8;
                    Gran         : UInt8;
                    StartAddress : UInt32;
                    Length       : UInt16;
                    Data         : ARRAY[0..Length-1] OF UInt8);
             END
```

Was in dieser Schreibweise nicht ganz zum Ausdruck kommt, ist, daß die Länge von Datenfeldern variabel ist und von `Length` abhängt.

Ein Record mit einem Header-Byte von `$81` ist ein Record, der Code oder Daten aus beliebigen Segmenten beinhalten kann. Das erste Byte (Header) gibt an, für welche Prozessorfamilie die folgenden Daten bzw. der folgende Code bestimmt ist (siehe Tabelle 5.1).

FamilyHdr	Familie	FamilyHdr	Familie
$01	680x0, 6833x	$02	ATARI_VECTOR
$03	M*Core	$04	XGATE
$05	PowerPC	$06	XCore
$07	TMS1000	$08	NS32xxx
$09	DSP56xxx	$0a	CP-1600
$0b	HP Nano Processor	$0c	IM6100/6120
$0d	NEC V60	$0e	IBM PALM
$0f	CP-3F	$10	Rockwell PPS-4
$11	65xx/MELPS-740	$12	MELPS-4500
$13	M16	$14	M16C
$15	F^2MC8L	$16	F^2MC16L
$17	IMP-16	$18	IPC-16/INS8900
$19	65816/MELPS-7700	$1a	PDK13
$1b	PDK14	$1c	PDK15
$1d	PDK16	$1e	Renesas RX
$1f	SC61860	$20	SC62015
$21	MCS-48	$22	Konami 052001

FamilyHdr	Familie	FamilyHdr	Familie
$23	PDP-11	$24	WD16
$25	SYM53C8xx	$26	VAX
$27	KENBAK	$28	μCOM-43
$29	29xxx	$2a	i960
$2b	TLCS-42	$2c	WE32xxx
$2d	TMS340xx	$2e	CR16A/B
$2f	CR16C	$30	TBIL
$31	MCS-51	$32	ST9
$33	ST7	$35	Z8000
$35	Super8	$36	MN161x
$37	2650	$38	1802/1805
$39	MCS-96/196/296	$3a	8X30x
$3b	AVR	$3c	XA
$3d	AVR (8-Bit CSeg)	$3e	8008
$3f	4004/4040	$40	H16
$41	8080/8085	$42	8086..V35
$43	SX20	$44	F8
$45	S12Z	$46	78K4
$47	TMS320C6x	$48	TMS9900
$49	TMS370xxx	$4a	MSP430
$4b	TMS320C54x	$4c	80C166/167
$4d	OLMS-50	$4e	OLMS-40
$4f	MIL STD 1750	$50	HMCS-400
$51	Z80/180/380/eZ80	$52	TLCS-900
$53	TLCS-90	$54	TLCS-870
$55	TLCS-47	$56	TLCS-9000
$57	TLCS-870/C	$58	NEC 78K3
$59	eZ8	$5a	TC9331
$5b	KCPSM3	$5c	LatticeMico8
$5d	NEC 75xx	$5e	68RS08
$5f	COP4	$60	78K2
$61	6800, 6301, 6811	$62	6805/HC08
$63	6809	$64	6804
$65	68HC16	$66	68HC12
$67	ACE	$68	H8/300(H)
$69	H8/500	$6a	807x
$6b	KCPSM	$6c	SH7000
$6d	SC14xxx	$6e	SC/MP
$6f	COP8	$70	PIC16C8x

FamilyHdr	Familie	FamilyHdr	Familie
$71	PIC16C5x	$72	PIC17C4x
$73	TMS-7000	$74	TMS3201x
$75	TMS320C2x	$76	TMS320C3x/C4x
$77	TMS320C20x/C5x	$78	ST6
$79	Z8	$7a	μPD78(C)10
$7b	75K0	$7c	78K0
$7d	μPD7720	$7e	μPD7725
$7f	μPD77230		

Tabelle 5.1: Headerbytes für die verschiedenen Prozessorfamilien

Das Segment-Feld gibt an, in welchen Adreßraum des Prozessors der folgende Code gehört. Dabei gilt die in Tabelle 5.2 angegeben Zuordnung. Das Gran-Feld

Nummer	Segment	Nummer	Segment
$00	<undefiniert>	$01	CODE
$02	DATA	$03	IDATA
$04	XDATA	$05	YDATA
$06	BDATA	$07	IO
$08	REG	$09	ROMDATA

Tabelle 5.2: Kodierungen des `Segment`-Feldes

gibt die „Granularität" des Codes an, d.h. die Größe der kleinsten, adressierbaren Einheit im folgenden Datensatz. Dieser Wert ist eine Funktion von Prozessortyp und Segment und ein wichtiges Detail für die Interpretation der beiden folgenden Felder, die Startadresse und Länge angeben: Während die Startadresse sich auf die Granularität bezieht, erfolgt die Längenangabe immer in Bytes! Wäre die Startadresse z.B. $300 und die Länge 12, so wäre die sich ergebende Endadresse bei einer Granularität von 1 $30b, bei einer Granularität von z.B. 4 jedoch $303. Andere Granularitäten als eins sind selten und treten in erster Linie bei Signalprozessoren auf, die nicht auf Einzelbyteverarbeitung ausgelegt sind deren Datenspeicher z.B. aus 64kWorten zu 16 Bit besteht (DSP56K). Der sich ergebende Speicherplatz beträgt dann zwar 128 KByte, er ist aber in 2^{16} Worten organisiert, die mit Adressen von 0,1,2,...65535 adressiert werden. Granularitäten kleiner als 8 Bit werden mit Werten von $f9..$ff kodiert. Falls das kleinste adressierbare Element ein Bit ist, ist die Granularität $ff, ist sie ein Nibble, ist die Granularität $fc.

Die Startadresse ist 32-bittig, unabhängig von der Adreßbreite der jeweiligen Prozessorfamilie. Im Gegensatz dazu ist die Längenangabe nur 16 Bit lang, ein Record kann also maximal (4+4+2+(64K-1)) = 65545 Byte lang werden.

Daten-Records mit den Header-Bytes `$01..$7f` stellen eine Kurzschreibweise dar und stellen die Abwärtskompatibilität mit früheren Definitionen des Dateiformats her: Das Header-Byte gibt direkt den Prozessortyp gemäß der ersten Tabelle an, das Zielsegment ist auf `CODE` festgelegt und die Granularität ergibt sich aus dem Prozessortyp, aufgerundet auf eine Zweierpotenz von Bytes. AS bevorzugt diese Records, wenn Daten bzw. Code für das `CODE`-Segment anstehen.

Der Record mit dem Typ-Byte `$80` legt den Einsprungpunkt fest, d.h. die Adresse, an der mit der Ausführung des Programms begonnen werden soll. Ein solcher Record ist das Ergebnis einer `END`-Anweisung mit einer entsprechenden Adresse als Argument.

Der letzte Record in der Datei trägt das Header-Byte `$00` und besitzt als einziges Datenfeld einen String, dessen Ende durch das Dateiende definiert ist. Dieser String spezifiziert, von welchem Programm diese Datei erzeugt wurde und hat keine weitere Bedeutung.

5.2 Debug-Dateien

Debug-Dateien können optional von AS erzeugt werden und liefern nachgeschalteten Werkzeugen wie Disassemblern oder Debuggern für diese wichtige Informationen. AS kann Debug-Informationen in drei Formaten ausgeben: Zum einen im Objekt-Format der AVR-Tools von Atmel sowie eine zu NoICE kompatible Kommandodatei und zum anderen in einem eigenen Format. Die ersten beiden werden in [5] bzw. der Dokumentation zu NoICE ausführlich beschrieben, deshalb beschränkt sich die folgende Beschreibung auf das AS-eigene MAP-Format:

Diese Informationen in einer MAP-Datei teilen sich in drei Gruppen:

- Symboltabelle
- Speicherberlegung, auf Sektionen verteilt
- Maschinenadressen von Quellzeilen

Letzterer Teil findet sich zuerst in der Datei. Ein einzelner Eintrag in dieser Liste besteht aus zwei, von einem Doppelpunkt getrennten Zahlen:

```
<Zeilennummer>:<Adresse>
```

Ein solcher Eintrag besagt, daß der aus einer bestimmten Quellcodezeile erzeugte Maschinencode auf der angegebenen Adresse (hexadezimal) zu liegen kam. Mit einer solchen Information kann ein Debugger beim Durchsteppen des Programms die entsprechenden Quellcodezeilen anzeigen. Da ein Programm aber auch aus mehreren Include-Dateien bestehen kann, und viele Prozessoren mehr als nur einen Adreßraum besitzen (von dem zugegebenermaßen nur in einem Code liegt), müssen die oben beschriebenen Einträge sortiert werden. AS tut dies in zwei Stufen: Das primäre Sortierkriterium ist das Zielsegment, innerhalb dieser Segmente wird noch einmal nach Dateien sortiert. Einzelne Abschnitte werden dabei durch durch spezielle Zeilen der Form

```
Segment <Segmentname>
```

bzw.

```
File <Dateiname>
```

getrennt.

Die Symboltabelle folgt der Quellzeileninformation und ist wieder primär nach den Segmenten geordnet, aus denen die Symbole stammen. Im Gegensatz zur Zeileninformation kommt hier allerdings auch der Abschnitt `NOTHING` hinzu, der die Symbole beinhaltet, die keinem speziellen Adreßraum zugeordnet sind (z.B. Symbole, die einfach mit `EQU` definiert wurden). Die Einleitung eines Abschnittes in der Symboltabelle erfolgt mit einer Zeile der Form

```
Symbols in Segment <Segmentname>   .
```

Innerhalb eines Abschnittes sind die Symbole nach Namen sortiert, und ein Symboleintrag belegt genau eine Zeile. Eine solche Zeile besteht wiederum aus sechs Feldern, die durch jeweils mindestens ein Leerzeichen getrennt sind:

Das erste Feld ist der Name des Symbols selber, eventuell erweitert um eine in eckigen Klammern eingeschlossene Sektionsnummer, die den Gültigkeitsbereich des Symbols einschränkt. Die zweite Spalte bezeichnet den Typ des Symbols: `Int` für Integerzahlen, `Float` für Gleitkommazahlen und `String` für Zeichenketten. Die dritte Zeile schließlich beinhaltet den eigentliche Wert des Symbols. Falls das Symbol eine Zeichenkette beinhaltet, ist es notwendig, Steuer- und Leerzeichen mit einer gesonderten Notation zu kennzeichnen, damit ein im String enthaltenes Leerzeichen nicht eventuell als Trennzeichen zur nächsten Spalte interpretiert werden kann. AS bedient sich dazu der bereits der in Assemblerquellen üblichen Schreibweise, den ASCII-Zahlenwert mit einem führenden Backslash (\) einzusetzen. Aus dem String

```
Dies ist ein Test
```

wird also z.B.

```
Dies\032ist\032ein\032Test
```

Die Zahlenangabe ist immer dezimal und dreistellig, und der Backslash selber wird ebenfalls in dieser Schreibweise kodiert.

Das vierte Feld gibt - falls vorhanden - die Größe der Datenstruktur an, die an der durch das Symbol gekennzeichneten Adresse abgelegt ist. Ein Debugger kann eine solche Information z.B. nutzen, um symbolisch angesprochene Variablen direkt in der korrekten Länge aufzulisten. Hat AS keine Informationen über die Symbolgröße, so steht in diesem Feld eine schlichte -1.

Das fünfte und letzte Feld gibt an, ob das Symbol während der Assemblierung jemals referenziert wurde. Ein Programm, daß die Symboltabelle liest, kann auf diese Weise z.B. nicht benutzte Symbole automatisch verwerfen, da sie beim folgenden Debugging oder der Disassemblierung mit hoher Wahrscheinlichkeit auch nicht benötigt werden.

das sechste und letzte Feld gibt schlußendlich durch eine 0 oder 1 an, ob es sich bei dem Symbol um eine Konstante (0) oder Variable (1) handelt. Konstanten bekommen einmalig einen Wert zugewiesen (z.B. über die `EQU`-Anweisung oder ein Label), Variablen können ihren Wert beliebig oft ändern. In der MAP-Datei wird der letztgültige Wert aufgeführt.

Der dritte Abschnitt in einer Debug-Datei beschreibt die im Programm benutzten Sektionen näher. Eine solche Beschreibung ist erforderlich, da Sektionen den Gültigkeitsbereich von Symbolen einschränken können. Je nach momentanem Stand des Programmzählers kann z.B. ein symbolischer Debugger einzelne Symboldefinitionen für eine Rückübersetzung nicht nutzen oder muß Prioritäten bei der Symbolnutzung beachten. Die Definition einer Sektion beginnt mit einer Zeile der Form

```
Info for Section nn ssss pp     ,
```

wobei `nn` die Nummer der Sektion angibt (die Nummer, die als Postfix für Symbolnamen in der Symboltabelle genutzt wird), `ssss` der Name der Sektion ist und `pp` die Nummer der Vatersektion darstellt. Letztere Information benötigt ein Rückübersetzer, um sich bei der Auffindung eines Symbols für einen Zahlenwert ausgehend von der aktuellen Sektion im Baum bis zur Wurzel ,,durchhangeln" kann, bis ein passendes Symbol gefunden wird. Auf diese Zeile folgt eine Reihe weiterer Zeilen, die den von dieser Sektion belegten Code-Bereich beschreiben. Jeder einzelne Eintrag (genau einer pro Zeile) beschreibt entweder eine einzelne Adresse oder einen durch zwei

Grenzwerte beschriebenen Bereich (Trennung von Anfangs-und Endwert durch ein Minuszeichen). Die Grenzen sind dabei ,,inklusive", d.h. die Grenzen gehören auch zu dem Bereich. Wichtig ist, daß ein einer Sektion zugehöriger Bereich nicht nochmals für ihre Vatersektionen aufgeführt wird (eine Ausnahme ist natürlich, wenn Bereiche absichtlich mehrfach belegt werden, aber so etwas macht man ja auch nicht, gelle?). Dies dient einer Optimierung der Bereichsspeicherung während der Assemblierung und sollte auch für eine Symbolrückübersetzung keine Probleme darstellen, da durch die einfache Kennzeichnung bereits der Einstiegspunkt und damit der Suchpfad im Sektionsbaum gegeben ist. Die Beschreibung einer Sektion wird durch eine Leerzeile oder das Dateiende gekennzeichnet.

Programmteile, die außerhalb aller Sektionen liegen, werden nicht gesondert ausgewiesen. Diese ,,implizite Wurzelsektion" trägt die Nummer -1 und wird auch als Vatersektion für Sektionen benutzt, die keine eigentliche Vatersektion besitzen.

Es ist möglich, daß die Datei Leerzeilen oder Kommentarzeilen (Semikolon am Zeilenanfang) beinhaltet. Diese sind von einem Leseprogramm zu ignorieren.

Kapitel 6

Hilfsprogramme

Um die Arbeit mit dem Codeformat des Assemblers etwas zu erleichtern, lege ich einige Progamme zu deren Bearbeitung bei. Für diese Programme gilt sinngemäß das gleiche wie in 1.1!

Allen Programmen gemeinsam sind die Returncodes, die sie liefern (Tabelle 6.1).

Returncode	tritt auf bei...
0	kein Fehler
1	Kommandozeilenparameterfehler
2	I/O-Fehler
3	Dateiformatfehler

Tabelle 6.1: Returncodes der Dienstprogramme

Ebenso einträchtig wie AS lesen sie ihre Eingaben von STDIN und schreiben Meldungen auf STDOUT (bzw. Fehlermeldungen auf STDERR). Ein-und Ausgaben sollten sich daher problemlos umleiten lassen.

Sofern Programme im folgenden Zahlen-oder Adreßangaben von der Kommandozeile lesen, dürfen diese auch hexadezimal geschrieben werden, indem man sie mit einem hintan gestellten `h`, einem voranstehenden Dollarzeichen oder `0x` wie in C versieht (z.B. `$10`, `10h` oder `0x10` anstelle von 16).

Unix-Shells ordnen dem Dollarzeichen allerdings eine spezielle Bedeutung zu (Parameterexpansion), weshalb es nötig ist, einem Dollarzeichen direkt einen Backslash voranzustellen. Die `0x`-Variante ist hier sicherlich angenehmer. *UNIX*

Ansonsten folgen die Aufrufkonventionen und -variationen (bis auf PLIST und AS2MSG) denen von AS, d.h. man kann dauernd gebrauchte Schalter in einer Environmentvariablen ablegen (deren Name sich aus dem Anhängen von CMD an

den Programmnamen ergibt, z.B. PBINDCMD für PBIND), Optionen negieren und Groß-bzw. Kleinschreibung erzwingen (näheres zu dem Wie in Abschnitt 2.4).

Sofern Adreßangaben benutzt werden, beziehen sie sich immer auf die Granularität des Adreßraumes des jeweiligen Prozessors; beim PIC bedeutet z.B. eine Adreßdifferenz von 1 nicht ein Byte, sondern ein Wort.

6.1 PLIST

PLIST ist das einfachste Programm der vier mitgelieferten; es dient einfach nur dazu, die in einer Codedatei gespeicherten Records aufzulisten. Da das Programm nicht allzuviel bewirkt, ist der Aufruf ziemlich simpel:

```
PLIST $<$Dateiname$>$
```

Der Dateiname wird automatisch um die Endung P erweitert, falls keine Endung vorhanden ist.

ACHTUNG! An dieser Stelle sind keine Jokerzeichen erlaubt! Falls mit einem Befehl trotzdem mehrere Programmdateien gelistet werden sollen, kann man sich mit folgendem "Minibatch" behelfen:

```
for %n in (*.p) do plist %n
```

PLIST gibt den Inhalt der Codedatei in Tabellenform aus, wobei für jeden Record genau eine Zeile ausgegeben wird. Die Spalten haben dabei folgende Bedeutung:

- Codetyp: die Prozessorfamilie, für die der Code erzeugt wurde.
- Startadresse: absolute Speicheradresse, an die der Code zu laden ist.
- Länge: Länge des Codestücks in Byte.
- Endadresse: letzte absolute Adresse des Codestücks. Diese berechnet sich als Startadresse+Länge-1.

Alle Angaben sind als hexadezimal zu verstehen.

Zuletzt gibt PLIST noch einen Copyrightvermerk aus, sofern er einen solchen in der Datei findet, und die Summe aller Codelängen.

PLIST ist praktisch ein DIR für Codedateien. Man kann es benutzen, um sich den Inhalt einer Datei auflisten zu lassen, bevor man sie weiterbearbeitet.

6.2 PBIND

PBIND ist ein Programm, mit dem man die Records mehrerer Codedateien in eine Datei zusammenkopieren kann. Die dabei vorhandene Filterfunktion erlaubt es aber auch, nur Records eines bestimmten Typs zu übernehmen. Auf diese Weise kann PBIND auch dazu verwendet werden, um eine Codedatei in mehrere aufzuspalten.

Die allgemeine Syntax von PBIND lautet

```
PBIND [Optionen] <Quelldatei(en)> [Zieldatei]
```

Wie auch AS betrachtet PBIND alle nicht mit einem +, - oder / eingeleiteten Parameter als Dateiangaben. Falls keine `-o` Option angegeben wurde, wird die letzte davon als Zieldatei angenommen. Alle anderen Dateiangaben bezeichnen Quellen, diese Angaben dürfen auch wieder Jokerzeichen enthalten.

An Optionen definiert PBIND momentan nur zwei:

- `o <Datei>`: spezifiziert die Ausgabe/Zieldatei.
- `f <Header[,Header...]>`: gibt eine Liste von Header-IDs an, die kopiert werden sollen. Alle anderen Records werden nicht kopiert. Ohne diese Angabe werden alle Records kopiert. Die in der Liste angegebenen entsprechen dem Header-Feld in der Recordstruktur, wie es in Abschnitt 5.1 beschrieben wurden. Die einzelnen Header-Nummern in der Liste werden durch Kommas getrennt.

Um z.B. alle MCS-51-Codeteile aus einer Programmdatei auszusieben, benutzt man PBIND folgendermaßen:

```
PBIND <Quellname> <Zielname> -f $31
```

Fehlt bei einer Dateiangabe eine Endung, so wird automatisch die Endung P angefügt.

6.3 P2HEX

P2HEX ist eine Erweiterung von PBIND. Es besitzt alle Kommandozeilenoptionen von PBIND und hat die gleichen Konventionen bezüglich Quell- und Ziel-Dateinamen. Im Gegensatz zu PBIND wird die Zieldatei aber als Hexfile ausgegeben, d.h. als eine Folge von Zeilen, die den Code als ASCII-Hexzahlen enthalten.

P2HEX kennt zehn verschiedene Zielformate, die über den Kommandozeilenparameter **F** ausgewählt werden können:

- Motorola S-Record (`-F Moto`)
- MOS Hex (`-F MOS`)
- Intel-Hex (Intellec-8, `-F Intel`)
- 16-Bit Intel-Hex (MCS-86, `-F Intel16`)
- 32-Bit Intel-Hex (`-F Intel32`)
- Tektronix Hex (`-F Tek`)
- Texas Instruments DSK (`-F DSK`)
- Atmel AVR Generic (`-F Atmel`, siehe [5])
- Lattice Mico8 prom_init (`-F Mico8`)
- C-Arrays, zum Inkludieren in C(++)-Quelldateien (`-F C`)

Wird kein Zielformat explizit angegeben, so wählt P2HEX anhand des Prozessortyps automatisch eines aus, und zwar S-Records für Motorola- Prozessoren, Hitachi und TLCS-900(0), MOS für 65xx/MELPS, DSK für die 16-Bit-Texas-Signalprozessoren, Atmel Generic für die AVRs und Intel-Hex für den Rest. Je nach Breite der Startadresse kommen bei S-Record Records der Typen 1,2 oder 3 zum Einsatz, jedoch nie in einer Gruppe gemischt. Diese Automatik läßt sich mit der Kommandozeilenoption

```
-M <1|2|3>
```

teilweise unterdrücken: Ein Wert von 2 bzw. 3 sorgt dafür, daß S-Records mit einem Mindesttyp von 2 bzw. 3 benutzt werden, während ein Wert von 0 der vollen Automatik entspricht.

Normalerweise benutzt das AVR-Format immer eine Adreßlänge von 3 Bytes. Manche Programme mögen das leider nicht...deshalb kann man mit dem Schalter

```
-avrlen <2|3>
```

die Länge zur Not auf 2 Bytes reduzieren.

Das Mico8-Format unterscheidet sich insofern aus den anderen Formaten, als daß es keine Adreßfelder besitzt - es ist eine schlichte Auflistung der Instruktionswörter im Programmspeicher. Bei der Benutzung muß darauf geachtet werden, daß der belegte Adreßbereich (der sich z.B. mit PLIST anzeigen läßt) bei Null beginnt und fortlaufend ist.

Die Intel-, Tektronix- und MOS-Formate sind auf 16 Bit-Adressen beschränkt, das 16-Bit Intel-Format reicht 4 Bit weiter. Längere Adressen werden von P2HEX mit einer Warnung gemeldet und abgeschnitten(!). Für die PICs können die drei von Microchip spezifizierten Varianten des Intel-Hex-Formates erzeugt werden, und zwar mit dem Schalter

```
-m <0..3>
```

Das Format 0 ist INHX8M, in dem alle Bytes in Lo-Hi-Ordnung enthalten sind. Die Adreßangaben verdoppeln sich, weil bei den PICs die Adresse sich nur um 1 pro Wort erhöht. Dieses Format ist gleichzeitig die Vorgabe. Im Format 1 (INHX16M) werden alle Worte in ihrer natürlichen Ordnung abgelegt. Dieses Format verwendet Microchip für seine eigenen Programmiergeräte. Format 2 (INHX8L) und 3 (INHX8H) trennen die Worte in ihre oberen und unteren Bytes auf. Um die komplette Information zu erhalten, muß P2HEX zweimal aufgerufen werden, z.B. so:

```
p2hex test -m 2
rename test.hex test.obl
p2hex test -m 3
rename test.hex test.obh
```

Für das Motorola-Format verwendet P2HEX zusätzlich einen in [13] genannten Recordtyp mit der Nummer 5, der die Zahl der folgenden Daten-Records (S1/S2/S3) bezeichnet. Da dieser Typ vielleicht nicht jedem Programm bekannt ist, kann man ihn mit der Option

```
+5
```

unterdrücken.

Das C-Format fällt insofern aus dem Rahmen, als daß es immer explizit ausgewählt werden muß. Die Ausgabedatei stellt im Prinzip ein vollständiges Stück C- oder C++-Code dar, das die Daten als eine Liste von C-Arrays enthält. Neben den eigentlichen Daten wird noch eine Liste von Deskriptoren geschrieben, die Start, Länge und Ende der Datenblöcke beschreiben. Was diese Deskriptoren enthalten, kann mit der Option

```
-cformat <Format>
```

bestimmt werden. Jeder Buchstabe in `Format` legt ein Element des Deskriptors fest:

- Ein `d` oder `D` definiert einen Zeiger auf die Daten. Über den Groß- oder Kleinbuchstaben wird festgelegt, ob die Hex- Konstanten Groß- oder Kleinbuchstaben verwenden sollen.

- Ein `s` oder `S` definiert die Startadresse der Daten, wahlweise vom Typ *unsigned* oder *unsigned long*.
- Ein `l` oder `L` definiert die Länge der Daten, wahlweise vom Typ *unsigned* oder *unsigned long*.
- Ein `e` oder `E` definiert die Endadresse der Daten, d.h. die letzte von den Daten benutzte Adresse, wahlweise vom Typ *unsigned* oder *unsigned long*.

Finden sich Code-Records verschiedener Prozessoren in einer Quelldatei, so erscheinen die verschiedenen Hexformate auch gemischt in der Zieldatei — es empfiehlt sich also dringend, von der Filterfunktion Gebrauch zu machen, oder ein fixes Format über die `-F`-Option festzulegen.

Neben dem Codetypenfilter kennt P2HEX noch ein Adreßfilter, das nützlich ist, falls der Code auf mehrere EPROMs verteilt werden muß:

```
-r <Startadresse>-<Endadresse>
```

Die Startadresse ist dabei die erste Speicherzelle, die im Fenster liegen soll, die Endadresse die der letzten Speicherzelle im Fenster, *nicht* die der ersten außerhalb. Um z.B. ein 8051-Programm in 4 2764-EPROMs aufzuteilen, geht man folgendermaßen vor:

```
p2hex <Quelldatei> eprom1 -f $31 -r $0000-$1fff
p2hex <Quelldatei> eprom2 -f $31 -r $2000-$3fff
p2hex <Quelldatei> eprom3 -f $31 -r $4000-$5fff
p2hex <Quelldatei> eprom4 -f $31 -r $6000-$7fff
```

Anstelle einer festen Adresse kann man als Anfang bzw. Ende auch ein einfaches Dollarzeichen oder ein '0x' angeben. Dies bedeutet, daß die niedrigste bzw. höchste in der Quelldatei gefundene Adresse als Anfang bzw. Ende genommen wird. Der Default für den Bereich ist '0x-0x', d.h. es werden alle Daten aus der Quelldatei übernommen.

ACHTUNG! Die Splittung ändert nichts an den absoluten Adressen, die in den Hexfiles stehen! Sollen die Adressen im Hexfile bei 0 beginnen, so kann man dies durch den zusätzlichen Schalter

```
-a
```

erreichen. Um im Gegenteil die Adreßlage auf einen bestimmten Wert zu verschieben, kann man den Schalter

```
-R <Wert>
```

verwenden. Der dabei angegebene Wert ist ein *Offset*, d.h. er wird auf die in der Code-Datei angegebenen Adressen aufaddiert.

Den Inhalt einer Datei kann man mit einem Offset auf eine beliebige Position verschieben; diesen Offset hängt man einfach in Klammern an den Dateinamen an. Ist der Code in einer Datei z.B. auf Adresse 0 in der P-Datei abgelegt, man möchte ihn jedoch auf Adresse 1000h verschieben, so hängt man an `($1000)` an den Dateinamen (ohne Leerzeichen!) an.

Sofern die P-Datei nicht nur Daten aus dem Code-Segment enthält, kann man mit dem Schalter

```
-segment <name>
```

auswählen, aus welchen Segment Daten extrahiert und ins HEX-Format gewandelt werden sollen. Die als Argument anzugebenden Segmentnamen sind die gleichen wie für den `SEGMENT`-Befehl (3.2.20). Ein Sonderfall ist das TI-DSK-Format, das als einziges Format vermerken kann, ob Daten ins Code- oder Datensegment gehören. In diesem Fall extrahiert P2HEX automatisch beide Segmente, solange kein Segment explizit angegeben ist.

Analog zur `-r` Option kann man mit der Option

```
-d <Start>-<Ende>
```

ein Filter für das Datensegment angeben.

Für das DSK-, Intel- und Motorola-Format relevant ist die Option

```
-e <Adresse> ,
```

mit der man die in die Hex-Datei einzutragende Startadresse festlegen kann. Fehlt diese Angabe, so wird nach einen entsprechenden Eintrag in der Code-Datei gesucht. Ist auch dort kein Hinweis auf einen Einsprungpunkt zu finden, so wird kein Eintrag in die HEX-Datei geschrieben (DSK/Intel) bzw. das entsprechende Feld wird auf 0 gesetzt (Motorola).

Leider ist sich die Literatur nicht ganz über die Endezeile für Intel-Hexfiles einig. P2HEX kennt daher 3 Varianten, einstellbar über den Parameter **i** mit einer nachfolgenden Ziffer:

0 :00000001FF

1 :00000001

2 :0000000000

Defaultmäßig wird die Variante 0 benutzt, die die gebräuchlichste zu sein scheint.

Fehlt der Zieldateiangabe eine Endung, so wird `HEX` als Endung angenommen.

Defaultmäßig gibt P2HEX pro Zeile maximal 16 Datenbytes aus, wie es auch die meisten anderen Tools tun, die Hex-Files erzeugen. Wollen Sie dies ändern, so können Sie dies mit dem Schalter

```
-l <Anzahl>
```

tun. Der erlaubte Wertebereich liegt dabei zwischen 2 und 254 Datenbytes; ungerade Werte werden implizit auf gerade Anzahlen aufgerundet.

Meist werden die temporären, von AS erzeugten Code-Dateien nach einer Umwandlung nicht mehr unbedingt gebraucht. Mit der Kommandozeilen- option

```
-k
```

kann man P2HEX anweisen, diese automatisch nach der Konversion zu löschen.

Anders als PBIND erzeugt P2HEX keine leere Datei, wenn nur ein Dateiname (=Zieldatei) angegeben wurde, sondern bearbeitet die dazugehörige Codedatei. Es ist also ein Minimalaufruf à la

```
P2HEX <Name>
```

möglich, um <Name: >.HEX aus <Name: >.P zu erzeugen.

6.4 P2BIN

P2BIN funktioniert wie P2HEX und bietet die gleichen Optionen (bis auf die a- und i- Optionen, die bei Binärdateien keinen Sinn ergeben), nur wird das Ergebnis nicht als Hexdatei, sondern als einfache Binärdatei abgelegt. Dies kann dann z.B. direkt in ein EPROM gebrannt werden.

Zur Beeinflussung der Binärdatei kennt P2BIN gegenüber P2HEX noch drei weitere Optionen:

- `l` $< 8 - Bit - Zahl >$: gibt den Wert an, mit dem unbenutzte Speicherstellen in der Datei gefüllt werden sollen. Defaultmäßig ist der Wert $ff, so daß ein halbwegs intelligenter EPROM-Brenner sie überspringt. Man kann aber hiermit auch andere Werte einstellen, z.B. enthalten die gelöschten Speicherzellen der MCS-48-EPROM-Versionen Nullen. In einem solchen Falle wäre 0 der richtige Wert.

- `s`: weist das Programm an, eine Prüfsumme über die Binärdatei zu berechnen. Die Prüfsumme wird einmal als 32-Bit-Wert ausgegeben, zum anderen wird das Zweierkomplement der Bits 0..7 in der letzten Speicherstelle abgelegt, so daß die Modulo-256-Summe zu 0 wird.

- `byte-swap`: erlaubt es, die Bytes in der Binärdatei umzusortieren. Mit einer `2` als Argument werden die Bytes paarweise getauscht, mit einer `4` werden Vierergruppen gespiegelt.

- `m`: für den Fall, daß ein Prozessor mit 16- oder 32-Bit-Datenbus eingesetzt wird und die Binärdatei für mehrere EPROMs aufgesplittet werden muß. Das Argument kann folgende Werte annehmen:

  - `ALL`: alles kopieren
  - `ODD`: alle Bytes mit ungerader Adresse kopieren
  - `EVEN`: alle Bytes mit gerader Adresse kopieren
  - `BYTE0`..`BYTE3`: nur alle Bytes kopieren, deren Adresse die Form $4n + 0...4n + 3$ hat.
  - `WORD0`,`WORD1`: nur das untere bzw. obere 16-Bit-Wort der 32-Bit-Worte kopieren.

Nicht wundern: Bei letzteren Optionen ist die Binärdatei um den Faktor 2 oder 4 kleiner als bei `ALL`. Dies ist bei konstantem Adreßfenster logisch!

Falls die Code-Datei keine Startadresse enthält, kann man diese analog zu P2HEX über die `-e`-Kommandozeilenoption vorgeben. Auf Anforderung teilt P2BIN ihren Wert der Ergebnisdatei voran. Mit der Kommandozeilenoption

```
-S
```

wird diese Funktion aktiviert. Sie erwartet als Argument eine Zahlenangabe zwischen 1 und 4, die die Länge des Adressfeldes in Bytes bestimmt. Optional kann dieser Angabe auch noch der Buchstabe L oder B vorangestellt werden, um die Byte-Order dieser Adresse festzulegen. So erzeugt z.B. die Angabe `B4` eine 4-Byte-Adresse in Big-Endian-Anordnung, `L2` oder nur '2' eine 2-Byte-Adresse in Little-Endian-Anordnung.

6.5 AS2MSG

Bei AS2MSG handelt es sich eigentlich um kein Hilfsprogramm, sondern um ein Filter, das (glücklichen) Besitzern von Borland-Pascal 7.0 das Arbeiten mit dem Assembler erleichtern soll. In den DOS-Arbeitsumgebungen existiert ein ,,Tools"-Menü, das man um eigene Programme, z.B. AS erweitern kann. Das Filter erlaubt, die von AS gelieferten Fehlermeldungen mit Zeilenangabe direkt im Editorfenster anzuzeigen. Dazu muß im Tools-Menü ein neuer Eintrag angelegt werden (`Options/Tools/New`). Tragen Sie in die einzelnen Felder folgende Werte ein :

- Title: `~M~akroassembler`
- Program path: `AS`
- Command line: `-E !1 $EDNAME $CAP MSG(AS2MSG) $NOSWAP $SAVE ALL`
- bei Bedarf einen Hotkey zuordnen (z.B. Shift-F7)

Die Option `-E` sorgt dafür, daß Turbo-Pascal nicht mit STDOUT und STDERR durcheinander kommt.

Ich setze dabei voraus, daß sowohl AS als auch AS2MSG sich in einem Verzeichnis befinden, welches in der Pfadliste aufgeführt ist. Nach einem Druck auf dem passenden Hotkey (oder Auswahl aus dem Tools-Menü) wird AS mit dem Namen der Textdatei im aktiven Editorfenster aufgerufen. Die dabei aufgetretenen Fehler werden in ein separates Fenster geleitet, durch das man nun ,,browsen" kann. Mit **Ctrl-Enter** springt man eine fehlerhafte Zeile an. Zusätzlich enthält das Fenster die Statistik, die AS am Ende der Assemblierung ausgibt. Diese erhalten als Dummy-Zeilennummer 1.

Für diese Arbeitsweise sind sowohl TURBO.EXE (Real Mode) als auch BP.EXE (Protected Mode) geeignet. Ich empfehle BP, da in dieser Variante beim Aufruf nicht erst der halbe DOS-Speicher ,,freigeswappt" werden muß.

Anhang A

Fehlermeldungen von AS

Im folgenden findet sich eine halb-tabellarische Auflistung der in AS definierten Fehlermeldungen. Zu jeder Fehlermeldung finden sich folgende Angaben:

- interne Fehlernummer (für den Anwender nur mit der `n`-Option sichtbar);
- Fehlermeldung im Klartext;
- Typ:
  - Warnung: zeigt mögliche Fehler oder ineffizienten Code an. Assemblierung geht weiter.
  - Fehler: echte Fehler. Assemblierung geht weiter, aber keine Code-Datei wird geschrieben.
  - Fatal: schwerwiegende Fehler. Assemblierung wird abgebrochen.
- Ursache: die Situation(en), in denen der Fehler ausgegeben wird;
- Argument: Die Ausgabe, die auf Wunsch als erweiterte Fehlermeldung erfolgt.

5 Displacement=0, überflüssig

Typ:
Warnung

Ursache:
bei 680x0-,6809- und COP8-Prozessoren: Das Displacement in einem Adreßausdruck hat den Wert 0 ergeben. Es wird ein Adreßausdruck ohne Displacement erzeugt. Um keine Phasenfehler zu erzeugen, werden NOP-Befehle eingefügt.

Argument:
keines

10 Kurzadressierung möglich

Typ:
Warnung

Ursache:
bei 680x0-, 6502- und 68xx-Prozessoren können bestimmte Speicherbereiche mit kurzen Adressen erreicht werden. Um keine Phasenfehler zu erzeugen, wird zwar der kürzere Ausdruck erzeugt, der freie Platz wird aber mit NOPs aufgefüllt.

Argument:
keines

20 kurzer Sprung möglich

Typ:
Warnung

Ursache:
Bei 680x0 und 8086-Prozessoren kann der Sprung sowohl mit langem als auch kurzem Displacement ausgeführt werden. Da kein kurzer Sprung angefordert wurde, wurde im ersten Pass Platz für den langen Sprung freigehalten. Es wird ein kurzer Sprung erzeugt, der freie Platz wird mit NOPs aufgefüllt, um Phasenfehler zu vermeiden.

Argument:
keines

25 relativer Sprung möglich

Typ:
Warnung

Ursache:
Sprünge können beim Z80 sowohl absolut als auch relativ ausgeführt werden. In diesem Fall wurde ein absoluter Sprung angefordert, ein(kürzerer) relativer wäre aber möglich.

Argument:
das Sprungargument

30 kein Sharefile angelegt, SHARED ignoriert

Typ:
Warnung

Ursache:
Es wurde eine `SHARED`-Anweisung gefunden, es wurde aber keine Kommandozeilenoption angegeben, um eine Shared-Datei zu erzeugen.

Argument:
keines

40 FPU liest Wert evtl. nicht korrekt ein (>=1E1000)

Typ:
Warnung

Ursache:
Das BCD-Gleitkommaformat der 680x0-Koprozessoren erlaubt zwar vierstellige Exponenten, lt. Datenbuch können solche Werte aber nicht korrekt eingelesen werden. Der vierstellige Wert wird zwar erzeugt, eine Funktion ist aber nicht gewähleistet.

Argument:
keines

50 Privilegierte Anweisung

Typ:
Warnung

Ursache:
Es wurde eine Anweisung benutzt, die nur im Supervisor-Mode zulässig ist, obwohl dieser nicht mittels `SUPMODE ON` vorher explizit angezeigt wurde.

Argument:
keines

60 Distanz 0 nicht bei Kurzsprung erlaubt (NOP erzeugt)

Typ:
Warnung

Ursache:
Ein kurzer Sprung mit der Distanz 0 ist bei 680x0- bzw. COP8-Prozessoren nicht erlaubt, da dieser Sonderwert für lange Sprünge benötigt wird. Stattdessen wurde ein NOP-Befehl eingefügt.

Argument:
keines

70 Symbol aus falschem Segment

Typ:
Warnung

Ursache:
Das in dem Operanden benutzte Symbol ist aus einem Adreßraum, der nicht mit dem benutzten Befehl bearbeitet werden kann.

Argument:
keines

75 Segment nicht adressierbar

Typ:
Warnung

Ursache:
Das in dem Operanden benutzte Symbol ist aus einem Adreßraum, der mit keinem der Segmentregister des 8086 adressiert werden kann.

Argument:
Name des nicht adressierbaren Segments

80 Änderung des Symbolwertes erzwingt zusätzlichen Pass

Typ:
Warnung

Ursache:
Ein Symbol hat einen anderen Wert zugewiesen bekommen als im vorhergehenden Pass. Diese Warnung wird nur ausgegeben, falls die `r`-Option angegeben wurde.

Argument:
Der Name des fraglichen Symbols

90 Überlappende Speicherbelegung

Typ:
Warnung

Ursache:
Bei der Bildung der Belegungsliste wurde festgestellt, daß ein Speicherbereich im Codesegment mehrfach benutzt wurde. Ursache können unüberlegte `ORG`-Anweisungen sein.

Argument:
keines

95 Überlappende Registernutzung

Typ:
Warnung

Ursache:
In der Anweisung wurden Register ganz oder teilweise mehrfach in nicht zulässiger Weise verwendet.

Argument:
Das den Konflikt auslösende Argument

100 keine CASE-Bedingung zugetroffen

Typ:
Warnung

Ursache:
bei einem `SWITCH..CASE`-Konstrukt ohne `ELSECASE`-Zweig traf keiner der `CASE`-Zweige zu.

Argument:
keines

110 Seite möglicherweise nicht adressierbar

Typ:
Warnung

Ursache:
Das in dem Operanden benutzte Symbol liegt nicht in der momentan mit `ASSUME` eingestellten Fenster (ST6,78(C)10).

Argument:
keines

120 Registernummer muß gerade sein

Typ:
Warnung

Ursache:
Die Hardware erlaubt nur ein Registerpaar zu verketten, dessen Startadresse gerade ist (RR0, RR2..., nur Z8).

Argument:
keines

130 veralteter Befehl

Typ:
Warnung

Ursache:
Der verwendete Befehl ist zwar noch definiert, ist in seiner Funktion aber durch andere, neue Befehle ersetzbar und daher in zukünftigen Prozessorversionen eventuell nicht mehr vorhanden.

Argument:
keines

140 Nicht vorhersagbare Ausführung dieser Anweisung

Typ:
Warnung

Ursache:
Die verwendete Adressierungsart ist bei diesem Befehl zwar prinzipiell erlaubt, ein Register wird jedoch in einer Weise doppelt verwendet, daß je nach Ausührungsreihenfolge sich unterschiedliche Ergebnisse einstellen können.

Argument:
keines

141 widerstreitende Benutzung des gleichen Ziels in parallelem Konstrukt

Typ:
Warnung

Ursache:
Zwei Anweisungen in einem parallelen Konstrukt verwenden das gleiche Zielregister. Das Ergebnis davon ist undefiniert.

Argument:
das doppelt verwendete Register

150 Lokaloperator außerhalb einer Sektion überflüssig

Typ:
Warnung

Ursache:
Ein vorangestellter Klammeraffe dient dazu, sich explizit auf zu der Sektion lokale Symbole zu beziehen. Wenn man sich außerhalb einer Sektion befindet, gibt es keine lokalen Symbole, weshalb dieser Operator überflüssig ist.

Argument:
keines

160 sinnlose Operation

Typ:
Warnung

Ursache:
Die Anweisung ergibt entweder überhaupt keinen Sinn oder kann auf andere Weise schneller und kürzer ausgeführt werden.

Argument:
keines

170 unbekannter Symbolwert erzwingt zusätzlichen Pass

Typ:
Warnung

Ursache:
AS vermutet eine Vorwärtsreferenz eines Symbols, d.h. das Symbol wird benutzt, bevor es definiert wurde, und hält einen weiteren Pass für unumgänglich. Diese Warnung wird nur ausgegeben, falls die `r`-Option angegeben wurde.

Argument:
Der Name des fraglichen Symbols

180 Adresse nicht ausgerichtet

Typ:
Warnung

Ursache:
Eine Adresse ist nicht ein mehrfaches der Operandengröße. Das Datenbuch verbietet zwar solche Zugriffe, im Instruktionswort ist aber Platz für diese Adresse, so daß AS es bei einer Warnung belassen hat.

Argument:
keines

190 I/O-Adresse darf nicht verwendet werden

Typ:
Warnung

Ursache:
Der verwendete Adressierungsmodus oder die angesprochene Adresse sind zwar prinzipiell erlaubt, die Adresse liegt aber im Bereich der Peripherieregister, die in diesem Zusammenhang nicht verwendet werden dürfen.

Argument:
keines

200 mögliche Pipeline-Effekte

Typ:
Warnung

Ursache:
Ein Register wird in einer Befehlsfolge so verwendet, daß die Befehlsausführung möglicherweise nicht in der hingeschriebenen Form ablaufen wird. Üblicherweise wird ein Register benutzt, bevor der neue Wert zur Verfügung steht.

Argument:
das die Verklemmung verursachende Register

210 mehrfache Adreßregisterbenutzung in einer Anweisung

Typ:
Warnung

Ursache:
Ein Adreßregister wird in mehreren Adreßausdrücken eines Befehls benutzt. Sofern einer der beiden Ausdrücke das Register modifiziert, sind die Ergebnisadressen nicht eindeutig festgelegt.

Argument:
das mehrfach verwendete Register

220 Speicherstelle ist nicht bitadressierbar

Typ:
Warnung

Ursache:
Mit einer `SFRB`-Anweisung wurde versucht, eine Speicherstelle als bitadressierbar zu deklarieren, die aufgrund der Architektur des 8051 nicht bitadressierbar ist.

Argument:
keines

230 Stack ist nicht leer

Typ:
Warnung

Ursache:
Am Ende eines Durchlaufes ist ein vom Programm definierter Stack nicht leer.

Argument:
der Name des Stacks sowie seine Resttiefe

240 NUL-Zeichen in Strings, Ergebnis undefiniert

Typ:
Warnung

Ursache:
Eine String-Konstante enthält ein NUL-Zeichen. Dies funktioniert zwar mit der Pascal-Version, in Hinblick auf die C-Version von AS ist dies aber ein Problem, da C Strings mit einem NUL-Zeichen terminiert, d.h. der String wäre für C an dieser Stelle zu Ende...

Argument:
keines

250 Befehl überschreitet Seitengrenze

Typ:
Warnung

Ursache:
Ein Befehl steht zu Teilen auf verschiedenen Seiten. Da der Programmzähler des Prozessors aber nicht über Seitengrenzen hinweg inkrementiert wird, würde zur Laufzeit anstelle des Instruktionsbytes von der Folgeseite wieder das erste Byte der alten Seite geholt; das Programm würde fehlerhaft ablaufen.

Argument:
keines

255 Bereichsunterschreitung

Typ:
Warnung

Ursache:
Ein Zahlenwert lag unterhalb des erlaubten Bereichs. AS hat den Wert durch ein Abschneiden der oberen Bitstellen in den erlaubten Bereich gebracht, es ist jedoch nicht garantiert, daß sich durch diese Operation sinnvoller und korrekter Code ergibt.

Argument:
keines

260 Bereichsüberschreitung

Typ:
Warnung

Ursache:
Ein Zahlenwert lag oberhalb des erlaubten Bereichs. AS hat den Wert durch ein Abschneiden der oberen Bitstellen in den erlaubten Bereich gebracht, es ist jedoch nicht garantiert, daß sich durch diese Operation sinnvoller und korrekter Code ergibt.

Argument:
keines

270 negatives Argument für DUP

Typ:
Warnung

Ursache:
Das Wiederholungsargument einer `DUP`-Direktive war kleiner als 0. Es werden (analog zu einem Argument von genau 0) keine Daten abgelegt.

Argument:
keines

280 einzelner X-Operand wird als indizierte und nicht als implizite Adressierung interpretiert

Typ:
Warnung

Ursache:
Ein einzelner X-Operand kann sowohl als Register X als auch X-indizierte Adressierung mit Null-Displacement interpretiert werden, da sich Motorola hier nicht festlegt. AS wählt die letztere Variante, was möglicherweise nicht das erwartete ist.

Argument:
keines

300 Bit-Nummer wird abgeschnitten werden

Typ:
Warnung

Ursache:
Die Instruktion arbeitet nur auf Byte- bzw. Langwort-Operanden, Bitnummern jenseits 7 bzw. 31 werden von der CPU modulo-8 bzw. modulo-32 behandelt werden.

Argument:
keines

310 Ungültiger Wert für Registerzeiger

Typ:
Warnung

Ursache:
Gültige bzw. sinnvolle Werte für den Registerzeiger sind nur Werte von 0x00...0x70 bzw. 0xf0, weil die anderen Registerbereiche unbelegt sind.

Argument:
keines

320 Makro-Argument umdefiniert

Typ:
Warnung

Ursache:
Einem Makroparameter wurden zwei oder mehr verschiedene Werte zugewiesen. Dies kann bei der Verwendung von Schlüsselwortparametern auftreten. Das zuletzt angegebene Argument wird benutzt.

Argument:
Name des Makroparameters

330 veraltete Anweisung

Typ:
Warnung

Ursache:
Dies Anweisung ist veraltet und sollte nicht mehr in neuen Programmen verwendet werden.

Argument:
Die Anweisung, die stattdessen verwendet werden sollte.

340 Quelloperand länger oder gleich Zieloperand

Typ:
Warnung

Ursache:
Der Quelloperand ist länger oder gleich groß wie der Zieloperand, gemessen in Bits. Eine Null- oder Vorzeichenerweiterung ergibt keinen Sinn mit diesen Argumenten. Schlagen Sie im Referenzhandbuch der CPU das Verhalten in diesem Fall nach.

Argument:
keines

350 TRAP-Nummer ist gültige Instruktion

Typ:
Warnung

Ursache:
Ein TRAP mit dieser Nummer benutzt den gleichen Maschinencode wie ein von der CPU unterstützter Maschinenbefehl.

Argument:
keines

360 Padding hinzugefügt

Typ:
Warnung

Ursache:
Die Menge abgelegter Bytes ist ungerade; eine Hälfte des letzten 16-Bit-Wortes bleibt ungenutzt.

Argument:
keines

370 Registernummer-Umlauf

Typ:
Warnung

Ursache:
Die Startregisternummer plus die Anzahl der Register ergibt ein letztes Register jenseits des Endes der Registerbank.

Argument:
das Argument mit der Registeranzahl

380 verwende indizierte anstelle indirekter Adressierung

Typ:
Warnung

Ursache:
Indirekte Adressierung ist an dieser Stelle nicht erlaubt. Es wird stattdessen indizierte Adressierung mit einem (Dummy-)Displacement von Null verwendet.

Argument:
das Argument mit dem Adressierungsausdruck

390 nicht im Normal-Modus erlaubt

Typ:
Warnung

Ursache:
Dieser Maschinenbefehl ist nur im Panel-Modus und nicht im ,,Normalbetrieb" erlaubt.

Argument:
die fragliche Instruktion

400 nicht im Panel-Modus erlaubt

Typ:
Warnung

Ursache:
Dieser Maschinenbefehl ist nur im ,,Nornalbetrieb" und nicht im Panel-Modus erlaubt.

Argument:
die fragliche Instruktion

410 Argument ausserhalb erlauben Bereichs

Typ:
Warnung

Ursache:
Das Argument oder die Summe zweier Argumente liegt außerhalb des für diesen Befehls erlaubten Bereichs, obwohl in der Instruktion prinzipiell Raum für größere Werte ist.

Argument:
das fragliche Argument

420 Versuch, eine Mehrwort-Instruktion zu überspringen

Typ:
Warnung

Ursache:
Die vorherige Anweisung war eine Skip-Anweisung, die nur ein Maschinenwort im Code überspringen kann. Da die aktuelle Anweisung länger als ein Maschinenwort ist, würde der Skip mitten in die Anweisung springen.

Argument:
die fragliche Anweisung

430 implizite Vorzeichenerweiterung

Typ:
Warnung

Ursache:
Der Prozessor führt bei diesem Befehl eine Vorzeichenerweiterung auf die volle Registerbreite aus. Das bedeutet, dass die oberen Bits für das gegebene Argument mit 1 und nicht mit 0 gefüllt werden. Je nach folgender Nutzung kann dies irrelevant sein, muß aber nicht.

Argument:
der fragliche Wert

440 numerischer Wert -128 bedeutet Nutzung des E-Registers (Literal 'E' verwenden, um diese Warnung zu vermeiden)

Typ:
Warnung

Ursache:
Ein Displacement von -128 bedeutet beim SC/MP in diesem Fall, dass das Displacement aus dem E-Register genommen wird. Der Assembler kann in diesem Fall nicht sicher entscheiden, ob das gewollt ist oder der Wert -128 sich unbeabsichtigt aus einer Berechnung ergeben hat. Verwenden Sie stattdessen das Literal 'E' oder ein auf 'E' definiertes Registersymbol, um die Absicht klar zu machen und die Warnung zu vermeiden.

Argument:
das fragliche Displacement-Argument

450 I/O-Adresse muß mit INS/OUTS angesprochen werden

Typ:
Warnung

Ursache:
I/O-Adressen im Bereich des Prozessormoduls (0..3) können nur über die Befehle `INS` und `OUTS` erreicht werden, nicht über `IN` und `OUT`.

Argument:
das fragliche Adress-Argument

460 CASE-Limit passt nicht zur Anzahl der Sprungadressen

Typ:
Warnung

Ursache:
Die CASE-Anweisung erwartet eine Sprungtabelle mit so vielen Einträgen, wie das Limit-Argument plus eins angibt. Limit-Wert und Anzahl der Sprungadressen passen nicht zueinander.

Argument:
das Limit-Argument

470 Befehl als NOP assembliert

Typ:
Warnung

Ursache:
Der dieser Anweisung eigentlich zugewiesene Opcode wird anderweitig verwendet. Da die Anweisung ohnehin nichts bewirkt, wird sie stattdessen als NOP assembliert.

Argument:
keines

480 Argument wird als Vektor behandelt

Typ:
Warnung

Ursache:
Anhand des Arguments kann nicht zweifelsfrei entschieden werden, ob es sich um eine Adresse oder einen Vektor handelt. Da das Argument eine reine Zahl und keinem Segment zugeordnet ist, wird es als Vektor behandelt.

Argument:
das fragliche Argument

490 Zu grosse Integer-Konstante wird als Gleitkomma interpretiert

Typ:
Warnung

Ursache:
Das Argument ist zwar eine syntaktisch korrekte Ganzzahl-Konstante, ihr Wert ist aber außerhalb dessen, was intern als Ganzzahl dargestellt werden kann. Die Zahl wird deswegen als Gleitkomma-Zahl gespeichert.

Argument:
das fragliche Argument

500 Code-Erzeugung ausserhalb des CODE-Segments

Typ:
Warnung

Ursache:
Maschinencode kann nur innerhalb des CODE-Segments erzeugt werden. Die anderen Adressräume verfügen nicht über die dafür erforderliche Wortbreite.

Argument:
keines

510 Befehl wird SP überschreiben

Typ:
Warnung

Ursache:
Dieser Befehl benutzt neben dem explizit genannten noch das Folgeregister für das doppelt lange Ergebnis. Das Folgeregister ist in diesem Fall aber der Stackpointer.

Argument:
das Zielargument der Anweisung

1000 Symbol doppelt definiert

Typ:
Fehler

Ursache:
Einem Symbol wurde durch ein Label oder `EQU`, `PORT`, `SFR`, `LABEL`, `SFRB` oder `BIT` ein neuer Wert zugewiesen, dies ist aber nur bei `SET/EVAL` erlaubt.

Argument:
Name des fraglichen Symbols, bei eingeschalteter Querverweisliste zusätzlich die Zeile der ersten Definition

1010 Symbol nicht definiert

Typ:
Fehler

Ursache:
Ein benutztes Symbol ist auch im 2.Pass noch nicht in der Symboltabelle enthalten.

Argument:
Name des nicht gefundenen Symbols

1011 nicht genügend vorherige lokale Symbole

Typ:
Fehler

Ursache:
Ein rückwärts referenziertes lokales Symbol dieser Tiefe existiert nicht.

Argument:
Das nicht gefundene Symbol

1020 ungültiger Symbolname

Typ:
Fehler

Ursache:
Ein Symbolname entspricht nicht den Bedingungen für einen gültigen Symbolnamen. Beachten Sie, daß für Makro-und Funktionsparameter strengere Regeln gelten!

Argument:
der fehlerhafte Symbolname

1030 reservierter Symbolname

Typ:
Fehler

Ursache:
Der Symbolname ist zwar an sich gültig, dieser spezielle Name ist aber für andere Zwecke reserviert, und kann nicht für selbst definierte Symbole verwendet werden.

Argument:
der reservierte Symbolname

1090 Ungültiges Format

Typ:
Fehler

Ursache:
Das benutzte Befehlsformat existiert bei diesem Befehl nicht.

Argument:
Der Kennbuchstabe des verwendeten Formates

1100 Überflüssiges Attribut

Typ:
Fehler

Ursache:
Der benutzte Befehl (Prozessor oder Pseudo) darf kein mit einem Punkt angehängtes Attribut haben.

Argument:
keines

1105 Attribut darf nur 1 Zeichen lang sein

Typ:
Fehler

Ursache:
Das mit einem Punkt an einen Befehl angehängte Attribut muß genau ein Zeichen lang sein; weder mehr noch weniger ist erlaubt.

Argument:
keines

1107 undefiniertes Attribut

Typ:
Fehler

Ursache:
Das an einem Befehl angefügte Attribut ist ungültig.

Argument:
keines

1110 Unpassende Operandenzahl

Typ:
Fehler

Ursache:
Die bei einem Befehl (Prozessor oder Pseudo) angegebene Operandenzahl liegt nicht in dem für diesen Befehl erlaubten Bereich.

Argument:
Die erwartete Anzahl Argumente bzw. Operanden

1112 Kann Argument nicht in Teile aufspaltenl

Typ:
Fehler

Ursache:
Bei bestimmten Prozessoren (z.B. DSP56000) müssen die kommaseparierten Argumente weiter in Einzeloperanden aufgespalten werden, diese ist fehlgeschlagen.

Argument:
keines

1115 unpassende Optionszahl

Typ:
Fehler

Ursache:
Die bei diesem Befehl angegebene Zahl von Optionen liegt nicht in dem für diesen Befehl erlaubten Bereich.

Argument:
keines

1116 unbekannte Option

Typ:
Fehler

Ursache:
Eine Option dieses Namens existiert nicht.

Argument:
die fragliche Option

1117 doppelte Option

Typ:
Fehler

Ursache:
Optionen dürfen nicht mehfach aufgeführt werden.

Argument:
die fragliche Option

1118 nicht unterstützte Optionsliste

Typ:
Fehler

Ursache:
Optionen sind in dieser Kombination nicht verwendbar.

Argument:
die fragliche Optionsliste

1120 nur immediate-Adressierung erlaubt

Typ:
Fehler

Ursache:
Der benutzte Befehl läßt nur immediate-Operanden (mit vorangestelltem #) zu.

Argument:
keines

1130 Unpassende Operandengröße

Typ:
Fehler

Ursache:
Der Operand hat zwar einen für den Befehl zugelassenen Typ, jedoch nicht die richtige Länge (in Bits).

Argument:
keines

1131 Widersprechende Operandengrößen

Typ:
Fehler

Ursache:
Die angegebenen Operanden haben unterschiedliche Längen (in Bit).

Argument:
keines

1132 Undefinierte Operandengröße

Typ:
Fehler

Ursache:
Aus Opcode und Operanden läßt sich die Operandengröße nicht eindeutig bestimmen (ein Problem des 8086-Assemblers). Sie müssen die Operandengröße durch einen `BYTE`, `WORD`, usw. `PTR`-Präfix festlegen.

Argument:
keines

1133 Ganzzahl oder String erwartet, aber Gleitkommazahl erhalten

Typ:
Fehler

Ursache:
An dieser Stelle kann keine Gleitkommazahl als Argument verwendet werden.

Argument:
das fehlerhafte Argument

1134 Ganzzahl erwartet, aber Gleitkommazahl erhalten

Typ:
Fehler

Ursache:
An dieser Stelle kann keine Gleitkommazahl als Argument verwendet werden.

Argument:
das fehlerhafte Argument

1136 Gleitkommazahl erwartet, aber String erhalten

Typ:
Fehler

Ursache:
An dieser Stelle kann kein String als Argument verwendet werden.

Argument:
das fehlerhafte Argument

1137 Operandentyp-Diskrepanz

Typ:
Fehler

Ursache:
Die beiden Argumente eines Operanden haben nicht den gleichen Datentyp (Integer/Gleitkomma/String).

Argument:
keines

1138 String erwartet, aber Ganzzahl erhalten

Typ:
Fehler

Ursache:
An dieser Stelle kann keine Ganzzahl als Argument verwendet werden.

Argument:
das fehlerhafte Argument

1139 String erwartet, aber Gleitkommazahl erhalten

Typ:
Fehler

Ursache:
An dieser Stelle kann keine Gleitkommazahl als Argument verwendet werden.

Argument:
das fehlerhafte Argument

1140 zu viele Argumente

Typ:
Fehler

Ursache:
Einem Befehl wurden mehr als die unter AS zulässigen 20 Parameter übergeben.

Argument:
keines

1141 Ganzzahl erwartet, aber String erhalten

Typ:
Fehler

Ursache:
An dieser Stelle kann kein String als Argument verwendet werden.

Argument:
das fehlerhafte Argument

1142 Ganz- oder Gleitkommazahl erwartet, aber String erhalten

Typ:
Fehler

Ursache:
An dieser Stelle kann kein String als Argument verwendet werden.

Argument:
das fehlerhafte Argument

1143 String erwartet

Typ:
Fehler

Ursache:
An dieser Stelle kann nur ein (in einfachen Hochkommas eingeschlossener) String als Argument verwendet werden.

Argument:
das fehlerhafte Argument

1144 Ganzzahl erwartet

Typ:
Fehler

Ursache:
An dieser Stelle kann nur eine ganze Zahl als Argument verwendet werden.

Argument:
das fehlerhafte Argument

1145 Ganz-, Gleitkommazahl oder String erwartet, aber Register bekommen

Typ:
Fehler

Ursache:
An dieser Stelle kann kein Registersymbol als Argument verwendet werden.

Argument:
das fehlerhafte Argument

1146 Ganzzahl oder String erwartet

Typ:
Fehler

Ursache:
An dieser Stelle kann keine Gleitkommazahl oder Registersymbol als Argument verwendet werden.

Argument:
das fehlerhafte Argument

1147 Register erwartet

Typ:
Fehler

Ursache:
An dieser Stelle kann nur ein Register als Argument verwendet werden.

Argument:
das fehlerhafte Argument

1148 Registersymbol für anderes Ziel

Typ:
Fehler

Ursache:
Das angesprochene Registersymbol wurde für einen anderen Zielprozessor als den aktuell verwendeten definiert und ist nicht kompatibel.

Argument:
das fehlerhafte Argument

1149 Gleitkomma-Argument erwartet, aber Ganzzahl bekommen

Typ:
Fehler

Ursache:
An dieser Stelle kann kein Ganzzahl-, sondern nur ein Gleitkomma- Argument verwendet werden.

Argument:
das fehlerhafte Argument

1151 Ganz- oder Gleitkommazahl erwartet, aber Register bekommen

Typ:
Fehler

Ursache:
An dieser Stelle kann kein Register, sondern nur eine Ganz- oder Gleitkommazahl als Argument verwendet werden.

Argument:
das fehlerhafte Argument

1152 Ganzzahl oder String erwartet, aber Register bekommen

Typ:
Fehler

Ursache:
An dieser Stelle kann kein Register, sondern nur eine Ganzzahl oder ein String als Argument verwendet werden.

Argument:
das fehlerhafte Argument

1153 Ganzzahl erwartet, aber Register bekommen

Typ:
Fehler

Ursache:
An dieser Stelle kann kein Register, sondern nur eine Ganzzahl als Argument verwendet werden.

Argument:
das fehlerhafte Argument

1154 String zu lang

Typ:
Fehler

Ursache:
Dieser String ist zu lang, als daß er mit vorangestelltem Längen-Byte dargestellt werden könnte.

Argument:
das fehlerhafte Argument

1200 Unbekannter Befehl

Typ:
Fehler

Ursache:
Der benutzte Befehl ist weder ein Pseudobefehl von AS noch ein Befehl des momentan eingestellten Prozessors.

Argument:
keines

1300 Klammerfehler

Typ:
Fehler

Ursache:
Der Formelparser ist auf einen (Teil-)Ausdruck gestoßen, in dem die Summe öffnender und schließender Klammern nicht übereinstimmt.

Argument:
der beanstandete (Teil-)Ausdruck

1310 Division durch 0

Typ:
Fehler

Ursache:
Bei einer Division oder Modulooperation ergab die Auswertung des rechten Teilausdruckes 0.

Argument:
keines

1315 Bereichsunterschreitung

Typ:
Fehler

Ursache:
Der angegebene Integer-Wert unterschreitet den zulässigen Bereich.

Argument:
aktueller Wert und zulässiges Minimum (manchmal, ich stelle das seit Jahren um...)

1320 Bereichsüberschreitung

Typ:
Fehler

Ursache:
Der angegebene Integer-Wert überschreitet den zulässigen Bereich.

Argument:
aktueller Wert und zulässiges Maximum (manchmal, ich stelle das seit Jahren um...)

1322 keine Zweierpotenz

Typ:
Fehler

Ursache:
Hier sind nur Zweierpotenzen (1,2,4,8...) als Wert erlaubt

Argument:
der fragliche Wert

1323 ungültige Dezimalstelle

Typ:
Fehler

Ursache:
Ein String als Argument zu `PACKED` darf nur die Ziffern 0 bis 9 enthalten, sowie ein Plus- oder Minuszeiche nzu Anfang.

Argument:
das fragliche Argument

1324 Dezimalstring zu lang

Typ:
Fehler

Ursache:
Eine gepackte Dezimalzahl darf maximal 31 Stellen lang sein.

Argument:
das fragliche Argument

1325 Adresse nicht ausgerichtet

Typ:
Fehler

Ursache:
Die angegebene direkte Speicheradresse entspricht nicht den Ansprüchen des Datentransfers, d.h. ist nicht ein mehrfaches der Operandengröße. Nicht alle Prozessoren erlauben unausgerichtete Datenzugriffe.

Argument:
keines

1330 Distanz zu groß

Typ:
Fehler

Ursache:
Der in einem Adreßausdruck enthaltene Displacement-Wert ist zu groß.

Argument:
keines

1331 Ziel nicht auf gleicher Seite

Typ:
Fehler

Ursache:
Befehl und Operandenadresse müssen auf der gleichen Seite liegen.

Argument:
das fragliche Adreß-Argument

1340 Kurzadressierung nicht möglich

Typ:
Fehler

Ursache:
Die Adresse des Operanden liegt außerhalb des Speicherbereiches, in dem Kurzadressierung möglich ist.

Argument:
keines

1350 Unerlaubter Adressierungsmodus

Typ:
Fehler

Ursache:
Der benutzte Adressierungsmodus existiert generell zwar, ist an dieser Stelle aber nicht erlaubt.

Argument:
keines

1351 Adresse muß gerade sein

Typ:
Fehler

Ursache:
An dieser Stelle sind nur gerade Adressen erlaubt, da das unterste Bit für andere Zwecke verwendet wird oder reserviert ist.

Argument:
das fragliche Argument

1352 Adresse muß ausgerichtet sein

Typ:
Fehler

Ursache:
An dieser Stelle sind nur ausgerichtete (d.h glatt durch 2, 4, 8... teilbare) Adressen erlaubt, da die untersten Bits für andere Zwecke verwendet werden oder reserviert sind.

Argument:
das fragliche Argument

1355 Adressierungsmodus im Parallelbetrieb nicht erlaubt

Typ:
Fehler

Ursache:
Die verwendeten Adressierungsmodi sind zwar im sequentiellen Modus zulässig, jedoch nicht bei parallelen Instruktionen.

Argument:
keines

1356 ungültiges paralleles Konstrukt

Typ:
Fehler

Ursache:
Die parallele Ausführung dieser Instruktionen ist nicht möglich.

Argument:
keines

1360 undefinierte Bedingung

Typ:
Fehler

Ursache:
Die benutzte Bedingung für bedingte Sprünge existiert nicht.

Argument:
keines

1364 Befehl kann nicht bedingt ausgeführt werden

Typ:
Fehler

Ursache:
Dieser Maschinenbefehl kann nicht mit einer Bedingung verknüfpt werden.

Argument:
der fragliche Maschinenbefehl

1365 inkompatible Bedingungen

Typ:
Fehler

Ursache:
Die benutzte Kombination von Bedingungen kann nicht in einem Befehl verwendet werden.

Argument:
die Bedingung, bei der die Unverträglichkeit entdeckt wurde.

1366 unbekanntes Flag

Typ:
Fehler

Ursache:
Das angegebene Flag existiert nicht.

Argument:
Das Argument mit dem fraglichen Flag

1367 doppeltes Flag

Typ:
Fehler

Ursache:
Das angegebene Flag wurde mehrfach in der Liste verwendet.

Argument:
Das Argument mit dem doppelten Flag

1368 unbekannter Interrupt

Typ:
Fehler

Ursache:
Der angegebene Interrupt existiert nicht.

Argument:
Das Argument mit dem fraglichen Interrupt

1369 doppelter Interrupt

Typ:
Fehler

Ursache:
Der angegebene Interrupt wurde mehrfach in der Liste verwendet.

Argument:
Das Argument mit dem doppelten Interrupt

1370 Sprungdistanz zu groß

Typ:
Fehler

Ursache:
Sprungbefehl und Sprungziel liegen zu weit auseinander, um mit einem Sprung der benutzten Länge überbrückt werden zu können.

Argument:
keines

1371 Sprungdistanz ist Null

Typ:
Fehler

Ursache:
Sprungbefehl und Sprungziel liegen direkt hintereinander, und eine Sprungdistanz von null kann nicht kodiert werden.

Argument:
Die Zieladresse im Quellcode

1375 Sprungdistanz ist ungerade

Typ:
Fehler

Ursache:
Da Befehle nur auf geraden Adressen liegen dürfen, muß eine Sprungdistanz zwischen zwei Befehlen auch immer gerade sein, das Bit 0 der Distanz wird anderweitig verwendet. Diese Bedingung ist verletzt worden. Grund ist üblicherweise die Ablage einer ungeraden Anzahl von Daten in Bytes oder ein falsches `ORG`.

Argument:
keines

1376 Skip-Ziel passt nicht

Typ:
Fehler

Ursache:
Das angegebene Sprungziel ist nicht die Adresse, die der Prozessor bei Ausführung der Skip-Anweisung anspringen würde.

Argument:
die angegebene (beabsichtigte) Sprungadresse

1380 ungültiges Schiebeargument

Typ:
Fehler

Ursache:
als Argument für die Schiebeamplitude darf nur eine Konstante oder ein Datenregister verwendet werden. (nur 680x0)

Argument:
keines

1390 Nur Bereich 1..8 erlaubt

Typ:
Fehler

Ursache:
Konstanten für Schiebeamplituden oder `ADDQ`-Argumente dürfen nur im Bereich 1..8 liegen. (nur 680x0)

Argument:
keines

1400 Schiebezahl zu groß

Typ:
Fehler

Ursache:
(nicht mehr verwendet)

Argument:
keines

1410 Ungültige Registerliste

Typ:
Fehler

Ursache:
Das Registerlisten-Argument von `MOVEM` oder `FMOVEM` hat ein falsches Format. (nur 680x0)

Argument:
keines

1420 Ungültiger Modus mit CMP

Typ:
Fehler

Ursache:
Die verwendete Operandenkombination von `CMP` ist nicht erlaubt. (nur 680x0)

Argument:
keines

1430 Ungültiger Prozessortyp

Typ:
Fehler

Ursache:
Den mit `CPU` angeforderten Zielprozessor kennt AS nicht.

Argument:
der unbekannte Prozessortyp

1431 Ungültiger FPU-Typ

Typ:
Fehler

Ursache:
Den mit `FPU` angeforderten numerischen Koprozessor kennt AS nicht.

Argument:
der unbekannte FPU-Typ

1432 Ungültiger PMMU-Typ

Typ:
Fehler

Ursache:
Die mit `PMMU` angeforderte MMU kennt AS nicht.

Argument:
der unbekannte PMMU-Typ

1437 Ungültiges Prozessorregister

Typ:
Fehler

Ursache:
Ein Prozessorregister dieses Namens existiert nicht.

Argument:
das fragliche Register

1438 Ungültiges Basisregister

Typ:
Fehler

Ursache:
Dieses Register darf an dieser Stelle nicht als Basis benutzt werden.

Argument:
das fragliche Register

1439 Ungültiges Indexregister

Typ:
Fehler

Ursache:
Dieses Register darf an dieser Stelle nicht als Index benutzt werden.

Argument:
das fragliche Register

1440 Ungültiges Steuerregister

Typ:
Fehler

Ursache:
Das bei z.B. `MOVEC` benutzte Kontrollregister kennt der mit CPU gesetzte Prozessor (noch) nicht.

Argument:
keines

1441 unbekannter Vektor

Typ:
Fehler

Ursache:
Ein Vektor dieses Namens existiert nicht.

Argument:
das fragliche Vektor-Argument

1442 Register im aktuellen Ausführungsmodus nicht zugreifbar

Typ:
Fehler

Ursache:
Dieses Register ist nur im Kernel-Modus zugreifbar.

Argument:
das fragliche Register

1443 Register darf im aktuellen Ausführungsmodus nur gelesen werden

Typ:
Fehler

Ursache:
Dieses Register ist nur im Kernel-Modus schreibbar.

Argument:
das fragliche Register

1445 Ungültiges Register

Typ:
Fehler

Ursache:
Das benutzte Register ist zwar prinzipiell vorhanden, hier aber nicht erlaubt.

Argument:
keines

1446 Register mehr als einmal gelistet

Typ:
Fehler

Ursache:
Ein Register taucht in der Liste der zu sichernden bzw. wiederherzustellenden Register mehrfach auf.

Argument:
keines

1447 Register-Bank-Diskrepanz

Typ:
Fehler

Ursache:
In einem Adreßausdruck werden Register aus unterschiedlichen Bänken verwendet.

Argument:
das fragliche Register

1448 Registerlänge undefiniert

Typ:
Fehler

Ursache:
An dieser Stelle können Register verschiedener Länge verwendet werden, und die Registerlänge ist nicht alleine aus der Adresse ableitbar.

Argument:
das fragliche Argument

1449 ungültige Operation auf Register

Typ:
Fehler

Ursache:
Diese Operation kann auf dieses Register nicht angewendet werden, z.B. weil das Register nur gelesen oder nur geschrieben werden darf.

Argument:
das fragliche Register

1450 RESTORE ohne SAVE

Typ:
Fehler

Ursache:
Es wurde ein `RESTORE`-Befehl gefunden, obwohl kein mit `SAVE` gespeicherter Zustand (mehr) auf dem Stapel vorhanden ist.

Argument:
keines

1460 fehlendes RESTORE

Typ:
Fehler

Ursache:
Nach der Assemblierung sind nicht alle `SAVE`-Befehle wieder aufgelöst worden.

Argument:
keines

1465 unbekannte Makro-Steueranweisung

Typ:
Fehler

Ursache:
Eine beim `MACRO`-Befehl zusätzlich angegebene Steueranweisung ist AS unbekannt.

Argument:
die fragliche Anweisung

1470 fehlendes ENDIF/ENDCASE

Typ:
Fehler

Ursache:
Nach der Assemblierung sind nicht alle Konstrukte zur bedingten Assemblierung aufgelöst worden.

Argument:
keines

1480 ungültiges IF-Konstrukt

Typ:
Fehler

Ursache:
Die Reihenfolge der Befehle in einem `IF`- oder `SWITCH`-Konstrukt stimmt nicht.

Argument:
keines

1482 FORWARD-Anweisung muß sich auf aktuelle Sektion beziehen

Typ:
Fehler

Ursache:
`FORWARD`-Direktiven dürfen sich nicht auf Symbole in einer anderen (höheren) Sektion beziehen.

Argument:
der fragliche Symbolname

1483 doppelter Sektionsname

Typ:
Fehler

Ursache:
Es existiert bereits eine Sektion gleichen Namens auf dieser Ebene.

Argument:
der doppelte Name

1484 unbekannte Sektion

Typ:
Fehler

Ursache:
Im momentanen Sichtbarkeitsbereich existiert keine Sektion dieses Namens.

Argument:
der unbekannte Name

1485 fehlendes ENDSECTION

Typ:
Fehler

Ursache:
Nach Ende eines Durchganges sind nicht alle Sektionen wieder geschlossen worden.

Argument:
keines

1486 falsches ENDSECTION

Typ:
Fehler

Ursache:
die bei `ENDSECTION` angegebene Sektion ist nicht die innerste offene.

Argument:
keines

1487 ENDSECTION ohne SECTION

Typ:
Fehler

Ursache:
Es wurde ein `ENDSECTION`-Befehl gegeben, obwohl gar keine Sektion offen war.

Argument:
keines

1488 nicht aufgelöste Vorwärtsdeklaration

Typ:
Fehler

Ursache:
ein mit `FORWARD` oder `PUBLIC` angekündigtes Symbol wurde nicht in der Sektion definiert.

Argument:
der Name des fraglichen Symbols, plus die Position der Vorwärts-Deklaration im Quelltext

1489 widersprechende FORWARD ↔PUBLIC-Deklaration

Typ:
Fehler

Ursache:
Ein Symbol wurde sowohl als privat als auch global definiert.

Argument:
der Name des Symbols

1490 falsche Argumentzahl für Funktion

Typ:
Fehler

Ursache:
Die Anzahl der Argumente für eine selbst definierte Funktion stimmt nicht mit der geforderten Anzahl überein.

Argument:
keines

1491 doppelter Funktions-Argumentenname

Typ:
Fehler

Ursache:
Zwei oder mehrere Argumente einer Funktion haben den gleichen Namen

Argument:
das Argument mit dem doppelten Namen

1495 unaufgelöste Literale (LTORG fehlt)

Typ:
Fehler

Ursache:
Am Programmende oder beim Umachalten zu einem anderen Zielprozessor blieben noch nicht abgelegte Literale übrig.

Argument:
keines

1500 Befehl auf dem ... nicht vorhanden

Typ:
Fehler

Ursache:
Der benutzte Befehl existiert zwar grundsätzlich, das eingestellte Mitglied der Prozessorfamilie beherrscht ihn aber noch nicht.

Argument:
Die Prozessorvarianten, die diesen Befehl unterstützen würden.

1501 FPU-Befehle nicht freigeschaltet

Typ:
Fehler

Ursache:
Die FPU-Befehlssatzerweiterungen müssen erlaubt werden, um diesen Befehl zu benutzen

Argument:
keines

1502 PMMU-Befehle nicht freigeschaltet

Typ:
Fehler

Ursache:
Die PMMU-Befehlssatzerweiterungen müssen erlaubt werden, um diesen Befehl zu benutzen

Argument:
keines

1503 voller PMMU-Befehlssatz nicht freigeschaltet

Typ:
Fehler

Ursache:
Dieser Befehl ist nur im Befehlssatz der 68851-PMMU enthalten, nicht im reduzierten Befehlssatz der integrierten PMMU.

Argument:
keines

1504 Z80-Syntax nicht erlaubt

Typ:
Fehler

Ursache:
Dieser Befehl ist nur erlaubt, wenn die Z80-Syntax für 8080/8085-Befehle freigeschaltet wurde.

Argument:
keines

1505 Adressierungsart auf dem ... nicht vorhanden

Typ:
Fehler

Ursache:
Der benutzte Adressierungsmodus existiert zwar grundsätzlich, das eingestellte Mitglied der Prozessorfamilie beherrscht ihn aber noch nicht.

Argument:
Die Prozessorvarianten, die diesen Adressierungsmodus unterstützen würden.

1506 nicht im Z80-Syntax Exklusiv-Modus erlaubt

Typ:
Fehler

Ursache:
Dieser Befehl ist nicht (mehr) erlaubt, wenn nur noch Z80-Syntax für 8080/8085-Befehle erlaubt wurde.

Argument:
keines

1507 FPU-Befehl auf dem ... nicht unterstützt

Typ:
Fehler

Ursache:
Der benutzte FPU-Befehl existiert zwar grundsätzlich, der eingestellte FPU-Typ beherrscht ihn aber noch nicht.

Argument:
Der fragliche Befehl

1508 Custom-Befehle nicht freigeschaltet

Typ:
Fehler

Ursache:
Die benutzerdefinierten Befehlssatzerweiterungen müssen erlaubt werden, um diesen Befehl zu benutzen.

Argument:
Der fragliche Befehl

1509 Befehls-Erweiterung nicht freigeschaltet

Typ:
Fehler

Ursache:
Dieser Befehl ist Teil einer Befehlssatz-Erweiterung, deren Benutzung nicht erlaubt wurde.

Argument:
Der Name der Erweiterung

1510 Ungültige Bitstelle

Typ:
Fehler

Ursache:
Die angegebene Bitnummer ist nicht erlaubt oder eine Angabe fehlt komplett.

Argument:
keines

1520 nur ON/OFF erlaubt

Typ:
Fehler

Ursache:
Dieser Pseudobefehl darf als Argument nur `ON` oder `OFF` haben.

Argument:
keines

1521 ungültiger LISTING-Wert

Typ:
Fehler

Ursache:
Als Argument für `LISTING` sind nur die Werte `ON`, `OFF`, `NOSKIPPED` oder `PURECODE` erlaubt.

Argument:
das fragliche Argument

1530 Stack ist leer oder nicht definiert

Typ:
Fehler

Ursache:
Es wurde bei einem `POPV` einen Stack anzusprechen, der entweder nie definiert oder bereits leergeräumt wurde.

Argument:
der Name des fraglichen Stacks

1540 Nicht genau ein Bit gesetzt

Typ:
Fehler

Ursache:
In einer Bitmaske, die der `BITPOS`- Funktion übergeben wurde, war nicht genau ein Bit gesetzt.

Argument:
keines

1550 ENDSTRUCT ohne STRUCT

Typ:
Fehler

Ursache:
Eine `ENDSTRUCT`-Anweisung wurde gegeben, obwohl momentan keine Strukturdefinition in Gange war.

Argument:
keines

1551 offene Strukturdefinition

Typ:
Fehler

Ursache:
Nach Ende der Assemblierung waren noch nicht alle `STRUCT`-Anweisungen durch passende `ENDSTRUCT`s abgeschlossen.

Argument:
die innerste, noch nicht abgeschlossene Strukturdefinition

1552 falsches ENDSTRUCT

Typ:
Fehler

Ursache:
Der Namensparameter einer `ENDSTRUCT`-Anweisung entspricht nicht der innersten, offenen Strukturdefinition.

Argument:
keines

1553 Phasendefinition nicht in Strukturen erlaubt

Typ:
Fehler

Ursache:
Was gibt es dazu zu sagen? `PHASE` in einem Record ergibt einfach keinen Sinn und nur Verwirrung...

Argument:
keines

1554 ungültige `STRUCT`-Direktive

Typ:
Fehler

Ursache:
Als Direktive für `STRUCT` ist nur `EXTNAMES`, `NOEXTNAMES`, `DOTS` und `NODOTS` zugelassen.

Argument:
die unbekannte Direktive

1555 Struktur redefiniert

Typ:
Fehler

Ursache:
Eine Struktur dieses Namens wurde bereits definiert.

Argument:
der Name der Struktur

1556 nicht auflösbare Strukturelement-Referenz

Typ:
Fehler

Ursache:
Ein Element bezieht sich auf ein anderes Element in einer Strukturdefintions, dieses ist aber nicht definiert oder dessen Referenz selber is nicht auflösbar.

Argument:
Name des Elements und seine Referenz

1557 Strukturelement doppelt

Typ:
Fehler

Ursache:
Ein Element dieses Namens ist bereits in der Struktur enthalten.

Argument:
Name des Elements

1560 Anweisung nicht wiederholbar

Typ:
Fehler

Ursache:
Diese Maschinenanweisung kann nicht mit Hilfe eines `RPT`-Konstruktes wiederholt werden.

Argument:
keines

1600 vorzeitiges Dateiende

Typ:
Fehler

Ursache:
Es wurde mit einem `BINCLUDE`-Befehl versucht, über das Ende einer Datei hinauszulesen.

Argument:
keines

1700 ROM-Offset geht nur von 0..63

Typ:
Fehler

Ursache:
Das Konstanten-ROM der 680x0-Koprozessoren hat nur max. 63 Einträge.

Argument:
keines

1710 Ungültiger Funktionscode

Typ:
Fehler

Ursache:
Als Funktionscodeargument darf nur SFC, DFC, ein Datenregister oder eine Konstante von 0..15 verwendet werden. (nur 680x0-MMU)

Argument:
keines

1720 Ungültige Funktionscodemaske

Typ:
Fehler

Ursache:
Als Funktionscodemaske darf nur ein Wert von 0..15 verwendet werden. (nur 680x0-MMU)

Argument:
keines

1730 Ungültiges MMU-Register

Typ:
Fehler

Ursache:
Die MMU hat kein Register mit dem angegebenen Namen. (nur 680x0-MMU)

Argument:
keines

1740 Level nur von 0..7

Typ:
Fehler

Ursache:
Die Ebene für `PTESTW` und `PTESTR` muß eine Konstante von 0..7 sein. (nur 680x0-MMU)

Argument:
keines

1750 ungültige Bitmaske

Typ:
Fehler

Ursache:
Die bei den Bit-Feld-Befehlen angegebene Bitmaske hat ein falsches Format. (nur 680x0)

Argument:
keines

1760 ungültiges Registerpaar

Typ:
Fehler

Ursache:
Das angegebene Registerpaar ist hier nicht verwendbar oder syntaktisch falsch. (nur 680x0)

Argument:
keines

1800 offene Makrodefinition

Typ:
Fehler

Ursache:
Eine Makrodefinition war am Dateiende nicht abgeschlossen. Vermutlich fehlt ein `ENDM`.

Argument:
keines

1801 IRP ohne ENDM

Typ:
Fehler

Ursache:
Ein IRP-Block war am Dateiende nicht abgeschlossen. Vermutlich fehlt ein `ENDM`.

Argument:
keines

1802 IRPC ohne ENDM

Typ:
Fehler

Ursache:
Ein IRPC-Block war am Dateiende nicht abgeschlossen. Vermutlich fehlt ein `ENDM`.

Argument:
keines

1803 REPT ohne ENDM

Typ:
Fehler

Ursache:
Ein REPT-Block war am Dateiende nicht abgeschlossen. Vermutlich fehlt ein `ENDM`.

Argument:
keines

1804 WHILE ohne ENDM

Typ:
Fehler

Ursache:
Ein WHILE-Block war am Dateiende nicht abgeschlossen. Vermutlich fehlt ein `ENDM`.

Argument:
keines

1805 EXITM außerhalb eines Makrorumpfes

Typ:
Fehler

Ursache:
`EXITM` bricht die Expansion von Makro-Konstrukten ab. Dieser Befehl macht nur innerhalb von Makros Sinn und es wurde versucht, ihn außerhalb aufzurufen.

Argument:
Der fragliche EXITM-Befehl

1806 ENDM außerhalb eines Makrorumpfes

Typ:
Fehler

Ursache:
`ENDM` schließt die Definition eines Makros oder einen `REPT/IRP/WHILE`-Block ab. Außerhalt davon ergibt er keinen Sinn.

Argument:
Der fragliche ENDM-Befehl

1810 mehr als 10 Makroparameter

Typ:
Fehler

Ursache:
Ein Makro darf höchstens 10 Parameter haben.

Argument:
keines

1811 Schlüsselwortargument nicht in Makro definiert

Typ:
Fehler

Ursache:
Ein Schlüsselwortargument bezog sich auf einen Parameter, den das aufgerufene Makro gar nicht besitzt.

Argument:
verwendetes Schlüsselwort bzw. Makroparameter

1812 Positionsargument nach Schlüsselwortargumenten nicht mehr erlaubt

Typ:
Fehler

Ursache:
Positions- und Schlüsselwortargumente dürfen in einem Makroaufruf gemischt werden, aber nach dem ersten Schlüsselwortargument sind nur noch solche zugelassen.

Argument:
keines

1815 doppelte Makrodefinition

Typ:
Fehler

Ursache:
Ein Makronamne wurde in einer Sektion doppelt vergeben.

Argument:
der doppelt verwendete Name

1820 Ausdruck muß im ersten Pass berechenbar sein

Typ:
Fehler

Ursache:
Der benutzte Befehl beeinflußt die Codelänge, daher sind Vorwärtsreferenzen hier nicht erlaubt.

Argument:
keines

1830 zu viele verschachtelte IFs

Typ:
Fehler

Ursache:
(nicht mehr verwendet)

Argument:
keines

1840 ELSEIF/ENDIF ohne ENDIF

Typ:
Fehler

Ursache:
es wurde ein `ELSEIF`- oder `ENDIF`-Befehl gefunden, obwohl kein offener `IF`-Befehl vorhanden ist.

Argument:
keines

1850 verschachtelter/rekursiver Makroaufruf

Typ:
Fehler

Ursache:
(nicht mehr verwendet)

Argument:
keines

1860 unbekannte Funktion

Typ:
Fehler

Ursache:
Die angesprochene Funktion ist weder eingebaut noch nachträglich definiert worden.

Argument:
der Funktionsname

1870 Funktionsargument außerhalb Definitionsbereich

Typ:
Fehler

Ursache:
Das Argument liegt nicht im Bereich der angesprochenen transzendenten Funktion.

Argument:
keines

1880 Gleitkommaüberlauf

Typ:
Fehler

Ursache:
Das Argument liegt zwar im Bereich der angesprochenen transzendenten Funktion, das Ergebnis wäre aber nicht mehr darstellbar.

Argument:
keines

1890 ungültiges Wertepaar

Typ:
Fehler

Ursache:
Das benutzte Pärchen aus Basis und Exponent kann nicht berechnet werden.

Argument:
keines

1900 Befehl darf nicht auf dieser Adresse liegen

Typ:
Fehler

Ursache:
Die Prozessorhardware erlaubt keine Sprünge von dieser Adresse.

Argument:
keines

1905 ungültiges Sprungziel

Typ:
Fehler

Ursache:
Die Prozessorhardware erlaubt keine Sprünge zu dieser Adresse.

Argument:
keines

1910 Sprungziel nicht auf gleicher Seite

Typ:
Fehler

Ursache:
Sprungbefehl und Sprungziel müssen bei diesem Befehl auf der gleichen Seite liegen.

Argument:
keines

1911 Sprungziel nicht in gleicher Sektion

Typ:
Fehler

Ursache:
Sprungbefehl und Sprungziel müssen bei diesem Befehl in der gleichen (64K-)Sektion liegen.

Argument:
keines

1920 Codeüberlauf

Typ:
Fehler

Ursache:
Es wurde versucht, mehr als 1024 Bytes Code oder Daten in einer Zeile zu erzeugen.

Argument:
keines

1925 Adreßüberlauf

Typ:
Fehler

Ursache:
Der Adreßraum dieses Prozessors wurde überschritten.

Argument:
keines

1930 Konstanten und Platzhalter nicht mischbar

Typ:
Fehler

Ursache:
Anweisungen, die Speicher reservieren und solche, die ihn mit Konstanten belegen, dürfen nicht in einer Pseudoanweisung gemischt werden.

Argument:
keines

1940 Codeerzeugung in Strukturdefinition nicht zulässig

Typ:
Fehler

Ursache:
Ein `STRUCT`-Konstrukt dient nur der Beschreibung einer Datenstruktur und nicht dem Anlegen einer solchen, es sind daher keine Befehle zugelassen, die Code erzeugen.

Argument:
keines

1950 Paralleles Konstrukt nicht möglich

Typ:
Fehler

Ursache:
Entweder sind die beiden Instruktionen prinzipiell nicht parallel ausführbar, oder sie stehen nicht unmittelbar untereinander.

Argument:
keines

1960 ungültiges Segment

Typ:
Fehler

Ursache:
Das angegebene Segment ist an dieser Stelle nicht anwendbar.

Argument:
der benutzte Segmentname

1961 unbekanntes Segment

Typ:
Fehler

Ursache:
Das angegebene Segment existiert bei diesem Prozessor nicht.

Argument:
der benutzte Segmentname

1962 unbekanntes Segmentregister

Typ:
Fehler

Ursache:
Das angegebene Segmentregister existiert nicht (nur 8086).

Argument:
keines

1970 ungültiger String

Typ:
Fehler

Ursache:
Der angegebene String hat ein ungültiges Format.

Argument:
keines

1980 ungültiger Registername

Typ:
Fehler

Ursache:
Das angegebene Register existiert nicht oder darf hier nicht verwendet werden.

Argument:
keines

1985 ungültiges Argument

Typ:
Fehler

Ursache:
Der angegebene Befehl darf nicht mit einem `REP`-Präfix versehen werden.

Argument:
keines

1990 keine Indirektion erlaubt

Typ:
Fehler

Ursache:
in dieser Kombination ist keine indirekte Adressierung erlaubt.

Argument:
keines

1995 nicht im aktuellen Segment erlaubt

Typ:
Fehler

Ursache:
(nicht mehr verwendet)

Argument:
keines

1996 nicht im Maximum-Modus zulässig

Typ:
Fehler

Ursache:
Dieses Register ist nur im Minimum-Modus definiert.

Argument:
keines

1997 nicht im Minimum-Modus zulässig

Typ:
Fehler

Ursache:
Dieses Register ist nur im Maximum-Modus definiert.

Argument:
keines

2000 Anweisungspaket überschreitet Adreßgrenze

Typ:
Fehler

Ursache:
Ein Anweisungspaket darf nicht über eine 32-Byte-Adreßgrenze reichen.

Argument:
keines

2001 Ausführungseinheit mehrfach benutzt

Typ:
Fehler

Ursache:
Eine der Ausführungseinheiten des Prozessors wurde in einem Anweisungspaket mehrfach benutzt.

Argument:
der Name der Funktionseinheit

2002 mehrfache Lang-Leseoperation

Typ:
Fehler

Ursache:
Ein Ausführungspaket enthält mehr als eine Lang-Leseoperation, was nicht erlaubt ist.

Argument:
eine der Funktionseinheiten, auf denen eine Lang-Leseoperation ausgeführt wird

2003 mehrfache Lang-Schreiboperation

Typ:
Fehler

Ursache:
Ein Ausführungspaket enthält mehr als eine Lang-Schreiboperation, was nicht erlaubt ist.

Argument:
eine der Funktionseinheiten, auf denen eine Lang-Schreiboperation ausgeführt wird

2004 Lang-Lese- mit Schreiboperation

Typ:
Fehler

Ursache:
Ein Ausführungspaket enthält sowohl eine Lang-Leseoperation als auch eine Schreiboperation, was nicht erlaubt ist.

Argument:
eine der Funktionseinheiten, deren Operationen im Konflikt stehen.

2005 zu viele Lesezugriffe auf ein Register

Typ:
Fehler

Ursache:
Auf das gleiche Register wurde mehr als viermal im gleichen Anweisungspaket Bezug genommen.

Argument:
der Name des Registers, das zu oft referenziert wurde

2006 überlappende Ziele

Typ:
Fehler

Ursache:
Auf das gleiche Register wurde mehrfach im gleichen Ausführungspaket geschrieben, was nicht erlaubt ist.

Argument:
der Name der fraglichen Registers

2008 zu viele absolute Sprünge in einem Anweisungspaket

Typ:
Fehler

Ursache:
Ein Anweisungspaket beinhaltet mehr als einen direkten Sprung, was nicht erlaubt ist.

Argument:
keines

2009 Anweisung nicht auf diese Funktionseinheit ausführbar

Typ:
Fehler

Ursache:
Diese Anweisung kann nicht auf dieser Funktionseinheit ausgeführt werden.

Argument:
keines

2010 Ungültige Escape-Sequenz

Typ:
Fehler

Ursache:
Das mit einem Backslash eingeleitete Sonderzeichen ist nicht definiert.

Argument:
keines

2020 ungültige Präfix-Kombination

Typ:
Fehler

Ursache:
Die angegebene Kombination von Präfixen ist nicht zulässig oder nicht im Maschinenkode darstellbar.

Argument:
keines

2030 Konstante kann nicht als Variable redefiniert werden

Typ:
Fehler

Ursache:
Ein einmal mit `EQU` als Konstante
definiertes Symbol kann nicht nachträglich mit
`SET` verändert werden.

Argument:
der Name des fraglichen Symbols

2035 Variable kann nicht als Konstante redefiniert werden

Typ:
Fehler

Ursache:
Ein einmal mit `SET` als Variable
definiertes Symbol kann nicht nachträglich als
Konstante deklariert werden (z.B. mit `EQU`.

Argument:
der Name des fraglichen Symbols

2040 Strukturname fehlt

Typ:
Fehler

Ursache:
Bei einer Strukturdefinition fehlt der zugehörende Name für die Struktur.

Argument:
keines

2050 leeres Argument

Typ:
Fehler

Ursache:
In der Argumentenliste dieser Anweisung dürfen keine Leerstrings benutzt werden.

Argument:
keines

2060 nicht implementierte Anweisung

Typ:
Fehler

Ursache:
Der benutzte Maschinenbefehl ist dem Assembler zwar bekannt, ist aber aufgrund fehlender Dokumentation seitens des Prozessorherstellers momentan nicht implementiert.

Argument:
Der benutzte Befehl

2070 namenlose Struktur nicht Teil einer anderen Struktur

Typ:
Fehler

Ursache:
Eine Struktur oder Union, die keinen Namen hat, muß immer Teil einer anderen, benamten Struktur oder Union sein.

Argument:
keines

2080 STRUCT durch ENDUNION beendet

Typ:
Fehler

Ursache:
ENDUNION darf nur zum Beenden der Definition einer Union benutzt werden, nicht einer Struktur.

Argument:
Name der Struktur (falls vorhanden)

2090 Speicheradresse nicht auf aktiver Seite

Typ:
Fehler

Ursache:
Die Zieladresse befindet sich nicht in der durch das Seitenregister aktuell adressierbaren Speicherseite.

Argument:
keines

2100 unbekanntes Makro-Expansions-Argument

Typ:
Fehler

Ursache:
Ein `MACEXP_DFT/OVR` gegebenes Argument konnte nicht interpretiert werden.

Argument:
das unbekannte Argument

2105 zu viele Makro-Expansions-Argumente

Typ:
Fehler

Ursache:
Die Anzahl der Argument zur Makro-Expansion hat ihre Maximalzahl überschritten

Argument:
das Argument, das zu viel ist

2110 widersprüchliche Angaben zur Makro-Expansion

Typ:
Fehler

Ursache:
Eine Angabe zur Makroexpansion und ihr genaues Gegenteil dürfen nicht gleichzeitig als Argument von `MACEXP_DFT/OVR` verwendet werden.

Argument:
keines

2130 erwarteter Fehler nicht eingetreten

Typ:
Fehler

Ursache:
Ein per `EXPECT` angekündigter Fehler oder Warnung ist in dem durch `ENDEXPECT` abgeschlossenen Block nicht aufgetreten.

Argument:
Der erwartete Fehler

2140 Verschachtelung von EXPECT/ENDEXPECT nicht erlaubt

Typ:
Fehler

Ursache:
Durch `EXPECT/ENDEXPECT` eingerahmte Blöcke dürfen keine geschachtelten `EXPECT/ENDEXPECT`-Blöcke enthalten.

Argument:
keines

2150 fehlendes ENDEXPECT

Typ:
Fehler

Ursache:
Ein per `EXPECT` geöffneter Block wurde nicht per `ENDEXPECT` abgeschlossen.

Argument:
keines

2160 ENDEXPECT ohne EXPECT

Typ:
Fehler

Ursache:
Zu einem `ENDEXPECT` gibt es kein vorhergehendes `EXPECT`.

Argument:
keines

2170 kein Default-Checkpoint-Register definiert

Typ:
Fehler

Ursache:
Bei einer Typ-12-Instruktion wurde kein Checkpoint-Register angegeben und es wurde auch keines über die `CKPT`-Anweisung vorher definiert.

Argument:
keines

2180 ungültiges Bitfeld

Typ:
Fehler

Ursache:
Das Bitfeld entspricht nicht der erwarteten `(start,count)`-Syntax.

Argument:
das fragliche Argument

2190 Argument-Wert fehlt

Typ:
Fehler

Ursache:
Argumente müssen die Form 'variable=wert' haben.

Argument:
das fragliche Argument

2200 unbekanntes Argument

Typ:
Fehler

Ursache:
Dieses Variable wird von der gewählten Zielplattform nicht unterstützt.

Argument:
das fragliche Argument

2210 Indexregister muss 16 Bit sein

Typ:
Fehler

Ursache:
Indexregister beim Z8000 müssen eine Länge von 16 Bit (Rn) haben.

Argument:
das fragliche Argument

2211 I/O Adressregister muss 16 Bit sein

Typ:
Fehler

Ursache:
Z8000-Register, um I/O-Adressen zu adressieren, müssen eine Länge von 16 Bit (Rn) haben.

Argument:
das fragliche Argument

2212 Adressregister im segmentierten Modus muss 32 Bit sein

Typ:
Fehler

Ursache:
Z8000-Register, um Speicheradressen im segmentierten Modus zu adressieren, müssen eine Länge von 32 Bit (RRn) haben.

Argument:
das fragliche Argument

2213 Adressregister im nicht-segmentierten Modus muss 16 Bit sein

Typ:
Fehler

Ursache:
Z8000-Register, um Speicheradressen im nicht-segmentierten Modus zu adressieren, müssen eine Länge von 16 Bit (Rn) haben.

Argument:
das fragliche Argument

2220 ungültiges Strukturargument

Typ:
Fehler

Ursache:
Das Argument entspricht keinem der Argumente, die bei der Expansion einer Struktur erlaubt sind.

Argument:
das fragliche Argument

2221 zu viele Array-Dimensionen

Typ:
Fehler

Ursache:
Arrays von Strukturen dürfen maximal drei Dimensionen haben.

Argument:
die Dimension, die zu viel ist

2230 unbekannte Integer-Notation

Typ:
Fehler

Ursache:
Die angegebene Integer-Notation existiert nicht, oder es fehlt das führende Plus/Minuszeichen.

Argument:
das fehlerhafte Argument

2231 ungültige Liste an Integer-Schreibweisen

Typ:
Fehler

Ursache:
Die geforderte Änderung der Liste von benutzbaren Integer-Schreibweisen kann nicht vorgenommen werden, weil sie zu einem Widerspruch führen würde. Aktuell betrifft dies lediglich 0hex und 0oct, die nicht gleichzeitig verwendet werden dürfen.

Argument:
keines

2240 ungültige Skalierung

Typ:
Fehler

Ursache:
Das angegebene Argument kann nicht als Skalierungsfaktor benutzt werden.

Argument:
das fragliche Argument

2250 widersprüchliche String-Optionen

Typ:
Fehler

Ursache:
Das angegebene String-Option steht im Widerspruch zu einer vorherigen in der Liste.

Argument:
die fragliche Option

2251 unbekannte String-Option

Typ:
Fehler

Ursache:
Das angegebene String-Option gibt es nicht.

Argument:
die fragliche Option

2252 ungültiger Cache-Invalidierungs-Modus

Typ:
Fehler

Ursache:
Es können nur Daten-, Instruktions- oder beide Caches invalidiert werden.

Argument:
das fragliche Argument

2253 ungültige Config-Liste

Typ:
Fehler

Ursache:
Die Konfigurationsliste ist entweder insgesamt syntaktisch falsch oder enthält ungültige Elemente.

Argument:
Die fragliche Liste oder ein Element daraus.

2254 widersprüchliche Konfig-Optionen

Typ:
Fehler

Ursache:
Das angegebene Option steht im Widerspruch zu einer vorherigen in der Liste oder ist dazu identisch.

Argument:
die fragliche Option

2255 unbekannte Konfig-Option

Typ:
Fehler

Ursache:
Das angegebene Option gibt es nicht.

Argument:
die fragliche Option

2260 ungültiger Wert für CBAR

Typ:
Fehler

Ursache:
Dieser CBAR-Wert ist nicht erlaubt (CA muß größer als BA sein).

Argument:
keines

2270 Seite nicht zugreifbar

Typ:
Fehler

Ursache:
Die Zieladresse liegt auf einer Speicherseite, die aktuell nicht zugreifbar ist.

Argument:
keines

2271 aktueller Programmzähler in nicht-zugreifbarer Seiter

Typ:
Fehler

Ursache:
Die aktuell assemblierte Instruktion liegt auf einer physischen Adresse, die nicht in den logischen Adreßraum der CPU gemappt ist. Dies ist ein Problem bei PC-relativer Adressierung, weil für die Berechnung der Distanz die logische Adresse gebraucht wird.

Argument:
keines

2280 Feld nicht zugreifbar

Typ:
Fehler

Ursache:
Die Zieladresse liegt in einem Speicherfeld, das aktuell nicht zugreifbar ist.

Argument:
keines

2281 Ziel nicht in gleichem Feld

Typ:
Fehler

Ursache:
Befehl und Operandenadresse müssen im gleichen Speicherfeld liegen.

Argument:
keines

2290 Ungültige Instruktions-Kombination

Typ:
Fehler

Ursache:
Diese Befehle können nicht miteinander kombiniert werden.

Argument:
keines

2300 nicht-gemapptes Zeichen

Typ:
Fehler

Ursache:
Die Zeichenkette enthält einen nicht abbildbares Zeichen.

Argument:
die fragliche Zeichenkette

keines

2310 Multi-Zeichenkonstante mit ungültiger Länge

Typ:
Fehler

Ursache:
Multi-Zeichenkonstanten müssen zwischen ein und vier Zeichen lang sein.

Argument:
keines

2320 kein Target gesetzt ('CPU ...' oder '-cpu ...', um eines zu setzen)

Typ:
fatal

Ursache:
Es wurde noch kein Target gesetzt, der Assembler weiß daher nicht, für welche CPU Code erzeugt werden soll.

Argument:
keines

2330 Ungültige Displacement-Länge

Typ:
Fehler

Ursache:
Bei diesem Adressierungsmodus kann das Displacement diese Länge nicht haben.

Argument:
keines

10001 Fehler bein Öffnen der Datei

Typ:
fatal

Ursache:
Beim Versuch, eine Datei zu öffnen, ist ein Fehler aufgetreten.

Argument:
Beschreibung des E/A-Fehlers

10002 Listingschreibfehler

Typ:
fatal

Ursache:
Beim Schreiben des Assembler-Listings ist ein Fehler aufgetreten.

Argument:
Beschreibung des E/A-Fehlers

10003 Dateilesefehler

Typ:
fatal

Ursache:
Beim Lesen aus einer Quelldatei ist ein Fehler aufgetreten.

Argument:
Beschreibung des E/A-Fehlers

10004 Dateischreibfehler

Typ:
fatal

Ursache:
Beim Schreiben von Code- oder Share-Datei ist ein Fehler aufgetreten.

Argument:
Beschreibung des E/A-Fehlers

10006 Speicherüberlauf

Typ:
fatal

Ursache:
Der verfügbare Speicher reicht nicht mehr, alle Datenstrukturen aufzunehmen. Weichen Sie auf die DPMI- oder OS/2-Version von AS aus.

Argument:
keines

10007 Stapelüberlauf

Typ:
fatal

Ursache:
Der Programmstapel ist wegen zu komplizierter Formelausdrücke oder einer ungünstigen Anlage der Symbol- oder Makrotabelle übergelaufen. Versuchen Sie es noch einmal mit der `-A`-Option.

Argument:
keines

10008 INCLUDE zu tief verschachtelt

Typ:
fatal

Ursache:
Die Include-Verschachtelungstiefe hat das gegebene Limit (im Default 200) überschritten. Dieses Limit kann über den `-maxinclevel`-Schalter herauf gesetzt werden, eine fehlerhafte (rekusive) Verschachtelung ist aber die wahrscheinlichere Ursache.

Argument:
Die INCLUDE-Anweisung, mit der das Limit überschritten wurde.

10010 ungültiger Platzhalter in Listing Pro-Zeile-Präfix Format

Typ:
fatal

Ursache:
Als Platzhalter sind nur %i, %n oder %a zulässig.

Argument:
das ungültige Format

10011 zu oft verwendeter Platzhalter in Listing Pro-Zeile-Präfix Format

Typ:
fatal

Ursache:
Die Platzhalter %i und %n dürfen jeweils nur maximal dreimal im Formatstring enthalten sein. Genügt das nicht?

Argument:
das mehrfach verwendete Format

Anhang B

E/A-Fehlermeldungen

Die hier aufgelisteten Fehlermeldungen werden nicht nur von AS bei E/A- Fehlern ausgegeben, sondern auch von den Hilfsprogrammen PLIST, BIND, P2HEX und P2BIN. Es sind nur die Fehler näher erklärt, die m.E. bei der Arbeit auftreten können. Sollte doch einmal ein nicht erläuterter E/A-Fehler auftreten, so dürfte der Grund in einem Programmfehler liegen. Melden Sie dies unbedingt!!

2 Datei nicht gefunden
Die angegebene Datei existiert nicht oder liegt auf einem anderen Laufwerk.

3 Pfad nicht gefunden
Der Pfad eines Dateinamens existiert nicht oder liegt auf einem anderen Laufwerk.

4 zu viele offene Dateien
DOS sind die Dateihandles ausgegangen. Erhöhen Sie die `FILES=`-Angabe in der CONFIG.SYS.

5 Dateizugriff verweigert
Entweder reichen die Netzwerkrechte für einen Dateizugriff nicht, oder es wurde versucht, eine schreibgeschützte Datei zu überschreiben oder zu verändern. Bei Benutzung in DOS- Fenstern von Multitasking- Systemen ist es überdies möglich, daß ein andere Prozeß die Datei in exklusivem Zugriff hat.

6 ungültiger Dateihandle

12 ungültiger Zugriffsmodus

15 ungültiger Laufwerksbuchstabe
Das angesprochene Laufwerk existiert nicht.

16 aktuelles Verzeichnis kann nicht gelöscht werden

17 RENAME geht nicht über Laufwerke

100 vorzeitiges Dateiende
Eine Datei war zuende, obwohl sie es aufgrund ihrer Struktur noch nicht sein dürfte. Vermutlich ist sie beschädigt.

101 Diskette/Platte voll
Das spricht wohl für sich! Aufräumen!!

102 ASSIGN fehlt

103 Datei nicht offen

104 Datei nicht für Einlesen offen

105 Datei nicht für Ausgaben offen

106 Ungültiges numerisches Format

150 Diskette ist schreibgeschützt
Wenn Sie schon keine Festplatte als Arbeitsmedium verwenden, so sollten Sie wenigstens den Schreibschutz entfernen!

151 Unbekanntes Gerät
Sie haben versucht, ein Peripheriegerät anzusprechen, welches DOS unbekannt ist. Dies sollte normalerweise nicht auftreten, da der Name dann automatisch als Datei interpretiert wird.

152 Laufwerk nicht bereit
Schließen Sie die Klappe des Diskettenlaufwerks.

153 unbekannte DOS-Funktion

154 Prüfsummenfehler auf Diskette/Platte
Ein harter Lesefehler auf der Diskette. Nochmal versuchen; wenn immer noch vorhanden, Diskette neu formatieren bzw. ernste Sorgen um Festplatte machen!

155 ungültiger DPB

156 Positionierfehler
Der Platten/Disketten-Controller hat eine bestimmte Spur nicht gefunden. Siehe Nr. 154!

157 unbekanntes Sektorformat
DOS kann mit dem Format der Diskette nichts anfangen.

158 Sektor nicht gefunden
Analog zu Nr. 158, nur daß hier der angeforderte Sektor auf der Spur nicht gefunden werden konnte.

159 Papierende
Offensichtlich haben Sie die Ausgaben von AS direkt auf einen Drucker umgeleitet. Assembler-Listings können seeehr lang sein...

160 Geräteleseefehler
Nicht näher vom Gerätetreiber klassifizierter Lesefehler.

161 Geräteschreibfehler
Nicht näher vom Gerätetreiber klassifizierter Schreibfehler.

162 allgemeiner Gerätefehler
Hier ist der Gerätetreiber völlig ratlos, was passiert sein könnte.

Anhang C

Programmierbeispiele

Manche Fragen, wie man dieses oder jenes realisiert, werden häufiger gestellt, und es ist vielleicht wert, sie in einer 'Tipps und Tricks' Ecke zu dokumentieren. Hier in diesem Kapitel werde ich sie sammeln und vorstellen:

C.1 16-Bit-Befehle per Makro

Viele 8-Bit-Prozessoren können - wie der Name schon sagt - immer nur acht Bit auf einmal verarbeiten. Oft sind jedoch genug Register vorhanden, um zwei davon zu einem virtuellen '16 Bit Akkumulator' zusammenzufassen. Nun soll dieser per Makro definierte Befehlssatz möglichst die gleichen Adressierungsarten wie die Hardware-Befehle unterstützen. Dazu müßte man die Makro-Argumente irgendwie parsen - aber wie?

Der Motorola 6800 hat zum Beispiel zwei Akkumulatoren A und B, und es liegt nahe, diese als einem 16-Bit-Akku zu behandeln. Die Adressierungsarten sollten die gleichen sein wie bei 8-Bit-Operationen:

- immediate
- direkt (Adresse innerhalb der ersten 256 Byte)
- erweitert (beliebige Adresse)
- indiziert

Ein Makro, das so einen virtuellen 16-Bit-Befehl implementiert, muß die ein oder zwei Argumente also analysieren:

1. Indizierte Adressierung hat als einzige zwei Argumente.
2. Immediate-Adressierung ist am führenden Hash-Zeichen zu erkennen.
3. Ob eine Adresse in den ersten 256 Byte liegt oder nicht, kann man wieder dem Assembler überlassen.

Um Punkt 2 auszuführen, muß ein Argument in einen String überführt werden. Von diesem kann das führende Hash-Zeichen dann auch entfernt werden, um den eigentlichen Wert zu evaluieren. Komplett ausgeschrieben sieht das Makro dann so aus:

```
subd    macro   ARG1,ARG2
  if      "ARG2" != ""            ; indexed?
   suba    (ARG1)+1,ARG2
   sbcb    ARG1,ARG2
  elseif                          ; not indexed?
_SARG1   set     "ARG1"           ; convert to string
  if      substr(_SARG1,0,1)='#' ; immediate?
_SARG1    set     substr(_SARG1,1,strlen(_SARG1)-1) ; yes->remove #
   suba    #lo(VAL(_SARG1))      ; ...and subtract lo/hi bytes
   sbcb    #hi(VAL(_SARG1))
  elseif                          ; no immediate->ext. or direct
   suba    (ARG1)+1               ; and subtract lo/hi bytes
   sbcb    ARG1
  endif
  endif
  endm
```

Makroargumente wurden bewusst in Großbuchstaben geschrieben, so dass das Makro unabhängig davon funktioniert, ob der case-sensitve Modus eingeschaltet ist oder nicht. Verwendet werden kann das Makro z.B. so:

```
        subd    $0007                   ; direct
        subd    $1234                   ; absolute
        subd    #$55aa                  ; immediate
        subd    $12,x                   ; indexed
```

Jetzt ist es aber so, dass man nicht nur 16-bittig subtrahieren möchte. Man könnte für jede Operation eine eigene Version des Makros hinschreiben, es geht aber auch eleganter. Ein Makro kann selber wieder eine Makro-Definition enthalten. Man kann also ein 'Meta-Makro' definieren, das die Befehlsnamen als Argumente erhält:

```
def16   macro   NEWINST,LOINST,HIINST
NEWINST macro   ARG1,ARG2
  if      "ARG2" != ""            ; indexed?
   LOINST  (ARG1)+1,ARG2
   HIINST  ARG1,ARG2
  elseif                          ; not indexed?
_SARG1   set     "ARG1"                 ; convert to string
  if      substr(_SARG1,0,1)='#' ; immediate?
_SARG1    set     substr(_SARG1,1,strlen(_SARG1)-1) ; yes->remove #
   LOINST  #lo(VAL(_SARG1))      ; ...and subtract lo/hi bytes
   HIINST  #hi(VAL(_SARG1))
  elseif                          ; no immediate->ext. or direct
   LOINST  (ARG1)+1              ; ...and subtract lo/hi bytes
   HIINST  ARG1
  endif
  endif
  endm
  endm
```

Dann sind die restlichen Definitionen Einzeiler:

```
        def16   addd,adda,adcb
        def16   subd,suba,sbcb
        def16   andd,anda,andb
        def16   ord,ora,orb
        def16   eord,eora,eorb
```

Anhang D

Häufig gestellte Fragen

In diesem Kapitel habe ich versucht, einige besonders häufig gestellte Fragen mit den passenden Antworten zu sammeln. Die Antworten auf die hier auftauchenden Probleme finden sich zwar auch an anderer Stelle in der Anleitung, jedoch findet man sie vielleicht nicht auf den ersten Blick...

F: Ich bin DOS leid. Für welche Plattformen gibt es AS sonst ?

A: Neben der Protected-Mode-Version, die AS unter DOS mehr Speicher zur Verfügung stellt, existieren Portierungen für OS/2 und Unix-Systeme wie z.B. Linux (im Teststadium). An Versionen, die Softwareherstellern in Redmond beim Geldscheffeln zuarbeiten würden, ist momentan nicht gedacht. Sofern jemand anders in dieser Hinsicht aktiv werden will, stelle ich ihm aber gerne die AS-Quellen zur Verfügung, von denen sich die C-Variante insbesondere eignen dürfte. Über Fragen zu diesen Quellen hinaus sollte er sich aber nicht viel von mir erwarten...

F: Ist eine Unterstützung des XYZ-Prozessors für AS geplant?

A: Es kommen immer neue Prozessoren heraus, und ich bemühe mich, bei Erweiterung von AS Schritt zu halten. Der Stapel mit der Aufschrift ,,Unerledigt" auf meinem Schreibtisch unterschreitet aber selten die 10cm-Grenze... Bei der Planung, welche Kandidaten zuerst abgearbeitet werden, spielen Wünsche von Anwendern natürlich eine große Rolle. Das Internet und die steigende Zahl elektronisch publizierter Dokumentation erleichtern die Beschaffung von Unterlagen, speziell bei ausgefallenen oder älteren Architekturen wird es aber immer wieder schwierig. Wenn sich die fragliche Prozessorfamilie nicht in der

Liste in Planung befindlicher Prozessoren befindet (siehe Kapitel 1), macht es sich sehr gut, der Anfrage auch gleich ein passendes Datenbuch hinzuzupacken (zur Not auch leihweise!).

F: Ein freier Assembler ist ja eine feine Sache, aber eigentlich bräuchte ich jetzt auch noch einen Disassembler...und einen Debugger...ein Simulator wäre auch ganz nett..

A: AS ist ein Freizeitprojekt von mir, d.h. etwas, was ich in der Zeit tue, wenn ich mich nicht gerade um den Broterwerb kümmere. Von dieser Zeit nimmt AS schon einen ganz erheblichen Teil ein, und ab und zu genehmige ich mir auch mal eine Auszeit, um den Lötkolben zu schwingen, mal wieder eine Tangerine Dream-Platte bewußt zu hören, mich vor den Fernseher zu hocken oder einfach nur dringenden menschlichen Bedürfnissen nachzugehen. Ich habe einmal angefangen, einen Disassembler zu konzipieren, der wieder voll reassemblierbaren Code erzeugt und automatisch Daten- und Code-Bereiche trennt, habe das Projekt aber relativ schnell wieder eingestellt, weil die restliche Zeit für so etwas einfach nicht mehr reicht. Ich mache lieber eine Sache gut als ein halbes Dutzend mäßig. Von daher muß die Antwort also wohl ,,nein" heißen...

F: In den Bildschirmausgaben von AS tauchen seltsame Zeichen auf, z.B. Pfeile und eckige Klammern. Warum?

A: AS verwendet zur Bildschirmsteuerung defaultmäßig einige ANSI-Terminal-Steuersequenzen. Haben Sie keinen ANSI-Treiber installiert, so kommen diese Steuerzeichen ungefiltert auf Ihrem Bildschirm heraus. Installieren Sie entweder einen ANSI-Treiber oder schalten Sie die Steuersequenzen mit dem DOS-Befehl `SET USEANSI=N` ab.

F: Während der Assemblierung bricht AS plötzlich mit der Meldung eines Stapelüberlaufes ab. Ist mein Programm zu kompliziert?

A: Ja und Nein. Die Symboltabelle für Ihr Programm ist nur etwas unregelmäßig gewachsen, was zu zu hohen Rekursionstiefen im Zugriff auf die Tabelle geführt hat. Diese Fehler treten insbesondere bei der 16-Bit-OS/2-Version von AS auf, die nur über einen relativ kleinen Stack verfügt. Starten Sie AS noch einmal mit dem `-A`-Kommandozeilenschalter. Hilft dies auch nicht, so kommen als mögliche Problemstellen noch zu komplizierte Formelausdrücke in Frage. Versuchen Sie in einem solchen Fall, die Formel in Zwischenschritte aufzuspalten.

F: AS scheint mein Programm nicht bis zum Ende zu assemblieren. Mit einer älteren Version von AS (1.39) hat es dagegen funktioniert.

A: Neuere Versionen von AS ignorieren das `END`-Statement nicht mehr, sondern beenden danach wirklich die Assemblierung. Insbesondere bei Include-Dateien ist es früher vorgekommen, daß Anwender jede Datei mit einem `END`-Statement beendet haben. Entfernen Sie die überflüssigen `END`s.

F: Weil ich noch ein paar kompliziertere Assemblierfehler im Programm hatte, habe ich mir ein Listing gemacht und es einmal genauer angeschaut. Dabei ist mir aufgefallen, daß einige Sprünge nicht auf das gewünschte Ziel, sondern auf sich selbst zeigen!

A: Dieser Effekt tritt bei Vorwärtssprüngen auf, bei denen der Formelparser von AS im ersten Pass die Zieladresse noch nicht kennen kann. Da der Formelparser ein unabhängiges Modul ist, muß er sich in einem solchen Fall einen Wert ausdenken, der auch relativen Sprüngen mit kurzer Reichweite nicht wehtut, und dies ist nun einmal die aktuelle Programmzähleradresse selber...im zweiten Pass wären die korrekten Werte erschienen, aber zu diesem ist es nicht gekommen, da schon im ersten Pass Fehler auftraten. Korrigieren Sie die anderen Fehler zuerst, so daß AS zum zweiten Pass kommt, und das Listing sollte wieder vernünftiger aussehen.

F: Mein Programm wird zwar korrekt assembliert, bei der Umwandlung mit P2BIN oder P2HEX erhalte ich aber nur eine leere Datei.

A: Dann haben Sie wahrscheinlich das Adreßfilter nicht korrekt eingestellt. Defaultmäßig ist der Filter abgeschaltet, d.h. alle Daten werden übernommen, wenn ein manuell eingestellter Bereichsfilter nicht zu den benutzten Adressen paßt, kann man mit der '-r' Option aber (versehentlich) auch leere Dateien erzeugen.

F: Ich bekomme unter Unix bei der Benutzung von P2BIN oder P2HEX das Dollarzeichen nicht eingegeben. Die automatische Bereichsfestlegung funktioniert nicht, stattdessen gibt es eigenartige Fehlermeldungen.

A: Unix-Shells benutzen das Dollarzeichen zur Expansion von Shell-Variablen. Wollen Sie ein Dollarzeichen an eine Anwendung durchreichen, stellen Sie einen Backslash (\) voran. Im Falle der Adreßangabe bei P2BIN und P2HEX darf aber auch `0x` anstelle des Dollarzeichens benutzt werden, was dieses Problem von vornherein vermeidet.

F: Ich nutze AS auf einem Linux-System, das Ladeprogramm für mein Zielsystem läuft aber auf einem Windows-Rechner. Um das zu vereinfachen, greifen beide System auf das gleiche Netzwerklaufwerk zu. Leider will die Windows-Seite aber die von der Linux-Seite erzeugten Hex-Dateien nicht lesen :-(

A: Windows- und Linux-Systeme benutzen ein etwas abweichendes Format für Textdateien, unter die auch Hex-Dateien fallen. Während Windows jede Zeile mit den Zeichen CR (Carriage Return) und LF (Linefeed) abschließt, verwendet Linux nur ein Linefeed. Es hängt nun von der "Gutmütigkeit" eines Windows-Programms ab, ob es die Dateien im Linux-Format akzeptiert. Falls nicht, kann man die Dateien anstelle über ein Netzwerklaufwerk über FTP im ASCII-Modus übertragen, oder man konvertiert die Dateien unter ins Windows-Format. Das Programm *unix2dos* kann dazu z.B. verwendet werden, oder unter Linux ein kleines Script:

```
awk '{print $0"\r"}' test.hex >test_cr.hex
```

F: Ich habe in meinem Programm eine 16-bittige Adresse und muß die obere und untere Hälfte in getrennte CPU-Register laden. Wie extrahiere ich diese Hälften? Bei anderen Assemblern gibt es dafür eingebaute Funktionen.

A: Das geht zum einen ,,zu Fuß" mit den logischen und Schiebeoperatoren, zum anderen bekommt man die Funktionen `lo()` und `hi()`, indem man die mitgelieferte Datei `bitfuncs.inc` in das Programm inkludiert.

Anhang E

Pseudobefehle und Integer-Syntax, gesammelt

In diesem Anhang finden sich noch einmal als schnelle Referenz alle von AS zur Verfügung gestellten Pseudobefehle. Die Liste ist in zwei Teile gegliedert: Im ersten Teil finden sich Befehle, die unabhängig vom eingestellten Zielprozessor vorhanden sind, danach folgen für jede Prozessorfamilie die zusätzlich vorhandenen Befehle:

Immer vorhandene Befehle

=	:=	ALIGN	BINCLUDE	CASE
CHARSET	CPU	DEPHASE	DOTTEDSTRUCTS	ELSE
ELSECASE	ELSEIF	END	ENDCASE	ENDIF
ENDM	ENDS	ENDSECTION	ENDSTRUCT	ENUM
ENUMCONF	ERROR	EQU	.EQU	EVAL
EXITM	FATAL	FORWARD	FUNCTION	GLOBAL
IF	IFB	IFDEF	IFEXIST	IFNB
IFNDEF	IFNEXIST	IFNSYMEXIST	IFNUSED	IFSYMEXIST
IFUSED	INCLUDE	INTSYNTAX	IRP	LABEL
LISTING	MACEXP	MACECP_DFT	MACEXP_OVR	MACRO
MESSAGE	NEWPAGE	NEXTENUM	ORG	.PAGE
PHASE	POPV	PUSHV	PRTEXIT	PRTINIT
PUBLIC	READ	RELAXED	REPT	.RESTORE
RESTOREENV	RORG	.SAVE	SAVEENV	SECTION
SEGMENT	.SET	SHARED	.SHIFT	STRUC
STRUCT	.SWITCH	TITLE	UNION	WARNING
WHILE				

Zusätzlich existieren:

- `SET` als Alias zu `EVAL`, falls `SET` nicht bereits ein Prozessorbefehl ist.
- `SHIFT` bzw. `SHFT`, falls `SHIFT` bereits ein Prozessorbefehl ist.
- `RESTORE` als Alias zu `RESTOREENV`, falls `RESTORE` nicht bereits ein Prozessorbefehl ist.
- `SAVE` als Alias zu `SAVEENV`, falls `SAVE` nicht bereits ein Prozessorbefehl ist.
- `PAGE` bzw. `PAGESIZE`, falls `PAGE` bereits ein Prozessorbefehl ist.
- `SWITCH` bzw. `SELECT`, falls `SWITCH` bereits ein Prozessorbefehl ist.

Motorola 680x0/MCF5xxx

Default Integer Syntax: Motorola

DC[.<size>]	DS[.<size>]	FULLPMMU	FPU	PADDING
PMMU	REG	SUPMODE		

Motorola 56xxx

Default Integer Syntax: Motorola

DC	DS	PACKING	XSFR	YSFR

PowerPC

Default Integer Syntax: C

BIGENDIAN	BF16	DB	DD	DN
DO	DQ	DS	DT	DW
REG	SUPMODE			

IBM PALM

Default Integer Syntax: IBM

BF16	DB	DD	DN	DO
DQ	DS	DT	DW	PORT
REG				

Motorola M-Core

Default Integer Syntax: Motorola

DC[.<size>]	DS[.<size>]	REG	SUPMODE

Motorola XGATE

Default Integer Syntax: Motorola

ADR	BYT	DC[.<size>]	DFS	DS[.<size>]
FCB	FCC	FDB	PADDING	REG
RMB				

Motorola 68xx/Hitachi 63xx

Default Integer Syntax: Motorola

ADR	BYT	DB	DC[.<size>]	DFS
DS[.<size>]	DW	EXTRACOMMENTS	FCB	FCC
FDB	PADDING	PRWINS(68HC11KA4)	RMB	

Motorola/Freescale 6805/68HC(S)08

Default Integer Syntax: Motorola

ADR	BYT	DB	DC[.<size>]	DFS
DS[.<size>]	DW	EXTRACOMMENTS	FCB	FCC
FDB	PADDING	RMB		

Motorola 6809/Hitachi 6309

Default Integer Syntax: Motorola

ADR	ASSUME	BYT	DB	DC[.<size>]
DFS	DS[.<size>]	DW	EXTRACOMMENTS	FCB
FCC	FDB	PADDING	PLAINBASE	RMB

Konami 052001

Default Integer Syntax: Motorola

ADR	BYT	DB	DC[.<size>]	DFS
DS[.<size>]	DW	EXTRACOMMENTS	FCB	FCC
FDB	PADDING	RMB		

Motorola 68HC12

Default Integer Syntax: Motorola

ADR	BYT	DB	DC[.<size>]	DFS
DS[.<size>]	DW	EXTRACOMMENTS	FCB	FCC
FDB	PADDING	RMB		

NXP S12Z

Default Integer Syntax: Motorola

ADR	BYT	DB	DC[.<size>]	DEFBIT
DEFBITFIELD	DFS	DS[.<size>]	DW	EXTRACOMMENTS
FCB	FCC	FDB	PADDING	RMB

Motorola 68HC16

Default Integer Syntax: Motorola

ADR	ASSUME	BYT	DB	DC[.<size>]
DFS	DS[.<size>]	DW	EXTRACOMMENTS	FCB
FCC	FDB	PADDING	RMB	

Freescale 68RS08

Default Integer Syntax: Motorola

ADR	ASSUME	BYT	DB	DC[.<size>]
DFS	DS[.<size>]	DW	EXTRACOMMENTS	FCB
FCC	FDB	PADDING		

Hitachi H8/300(L/H)

Default Integer Syntax: Motorola

BIT	DC[.<size>]	DS[.<size>]	MAXMODE	PADDING
REG				

Hitachi H8/500

Default Integer Syntax: Motorola

ASSUME	BIT	COMPMODE	DATA	DC[.<size>]
DS[.<size>]	MAXMODE	PADDING	REG	

Hitachi SH7x00

Default Integer Syntax: Motorola

COMPLITERALS	DC[.<size>]	DS[.<size>]	DSP	FPU
LTORG	PADDING	REG	SUPMODE	

Hitachi HMCS400

Default Integer Syntax: Motorola

DATA	RES	SFR

Hitachi H16

Default Integer Syntax: Motorola

BIT	DC[.<size>]	DS[.<size>]	REG	SUPMODE

65xx/MELPS-740

Default Integer Syntax: Motorola

ADR	ASSUME	BYT	DB	DCB
DDB	DFS	DS	DW	FCB
FCC	FDB	RMB		

65816/MELPS-7700

Default Integer Syntax: Motorola

ADR	ASSUME	BF16	BYT	DB
DD	DDB	DN	DO	DQ
DS	DT	DW	DFS	FCB
FCC	FDB	RMB		

Mitsubishi MELPS-4500

Default Integer Syntax: Motorola

DATA	RES	SFR

Rockwell PPS-4

Default Integer Syntax: Intel

DATA	DC	DS	RES

Mitsubishi M16

Default Integer Syntax: Intel

BF16	DB	DD	DN	DO
DQ	DS	DT	DW	REG

Mitsubishi M16C

Default Integer Syntax: Intel

BF16	DB	DD	DN	DO
DQ	DS	DT	DW	REG

DEC PDP-11

Default Integer Syntax: C

BYTE	CIS	EIS	FIS	FLT2
FLT4	FP11	REG	SUPMODE	WORD

WD16

Default Integer Syntax: C

ASCII	ASCIZ	BYTE	FLT3	REG
WORD				

DEC VAX

ACCMODE	ASCIC	ASCII	ASCIZ	BLKB
BLKD	BLKF	BLKG	BLKH	BLKL
BLKO	BLKQ	BLKW	BYTE	D_FLOATING
DOUBLE	F_FLOATING	FLOAT	G_FLOATING	H_FLOATING
LONG	OCTA	REG	QUAD	WORD

WE32xxx

Default Integer Syntax: C

Symbol für Programmzähler: .

BYTE	DOUBLE	DS	EXECMODE	FLT
FPU	HALF	REG	WORD	

Olympia CP-3F/GI LP8000/SGS M380

Default Integer Syntax: Intel

DC	DS	PORT

Intel 4004/4040

Default Integer Syntax: Intel

DATA	DS	REG

Intel 8008

Default Integer Syntax: Intel

BF16	DB	DD	DFB	DN
DO	DQ	DS	DT	DW
PORT				
Z80SYNTAX				

Intel MCS-48

Default Integer Syntax: Intel

ASSUME	BF16	DB	DD	DN
DO	DQ	DS	DT	DW
REG				

Intel MCS-(2)51

Default Integer Syntax: Intel

BIGENDIAN	BIT	BF16	D1	DB
DD	DN	DO	DQ	DS
DT	DW	PORT	REG	SFR
SFRB	SRCMODE (MCS-251)			

Intel MCS-96

Default Integer Syntax: Intel

ASSUME	BF16	DB	DD	DN
DO	DQ	DS	DT	DW

Intel 8080/8085

Default Integer Syntax: Intel

BF16	DB	DD	DN	DO
DQ	DS	DT	DW	PORT
REG				

Intel 8086/80186/NEC V20...V5x

Default Integer Syntax: Intel

ASSUME	BF16	DB	DD	DN
DO	DQ	DS	DT	DW
PORT	REG			

Intel i960

Default Integer Syntax: Intel

BF16	DB	DD	DN	DO
DQ	DS	DT	DW	FPU
REG	SPACE	SUPMODE	WORD	

Signetics 8X30x

Default Integer Syntax: Motorola

LIV	RIV

Signetics 2650

Default Integer Syntax: Motorola

ACON	BF16	BIGENDIAN	DB	DD
DN	DO	DQ	DS	DT
DW	RES			

Philips XA

Default Integer Syntax: Intel

ASSUME	BF16	BIT	DB	DC[.<size>]
DD	DN	DO	DQ	DS[.<size>]
DT	DW	PADDING	PORT	REG
SUPMODE				

Atmel AVR

Default Integer Syntax: C

BF16	BIT	DATA	DB	DD
DN	DO	DQ	DS	DT
DW	PACKING	PORT	REG	RES
SFR				

AMD 29K

Default Integer Syntax: C

ASSUME	BF16	DB	DD	DN
DO	DQ	DS	DT	DW
EMULATED	REG	SUPMODE		

Siemens 80C166/167

Default Integer Syntax: Intel

ASSUME	BF16	BIT	DB	DD
DN	DO	DQ	DS	DT
DW	REG			

Zilog Zx80

Default Integer Syntax: Intel

```
BF16          DB            DD            DEFB          DEFW
DN            DO            DQ            DS            DT
DW            EXTMODE       LWORDMODE     PRWINS(Z(1,2)8E)REG
SUPMODE (Z280)WARNRELATIVE
```

Zilog Z8

Default Integer Syntax: Intel

```
BF16          DB            DEFBIT        DD            DN
DO            DQ            DS            DT            DW
REG           SFR
```

Zilog Z8000

Default Integer Syntax: Intel

```
BF16          DB            DD            DEFBIT        DEFBITB
DN            DO            DQ            DS            DT
DW            PORT          REG
```

Xilinx KCPSM

Default Integer Syntax: Intel

```
CONSTANT      NAMEREG       REG
```

Xilinx KCPSM3

Default Integer Syntax: Intel

BF16	CONSTANT	DB	DD	DN
DO	DQ	DS	DT	DW
NAMEREG	PORT	REG		

LatticeMico8

Default Integer Syntax: C

BF16	DB	DD	DN	DO
DQ	DS	DT	DW	PORT
REG				

Toshiba TLCS-900

Default Integer Syntax: Intel

BF16	DB	DD	DN	DO
DQ	DS	DT	DW	MAXIMUM
SUPMODE				

Toshiba TLCS-90

Default Integer Syntax: Intel

BF16	DB	DD	DN	DO
DQ	DS	DT	DW	

Toshiba TLCS-870(/C)

Default Integer Syntax: Intel

BF16	DB	DD	DN	DO
DQ	DS	DT	DW	

Toshiba TLCS-47(0(A))

Default Integer Syntax: Intel

ASSUME	BF16	DB	DD	DN
DO	DQ	DS	DT	DW
PORT				

Toshiba TLCS-42

Default Integer Syntax: Intel

ASSUME	BF16	DB	DD	DN
DO	DQ	DS	DT	DW
PORT				

Toshiba TLCS-9000

Default Integer Syntax: Intel

BF16	DB	DD	DN	DO
DQ	DS	DT	DW	REG

Toshiba TC9331

Default Integer Syntax: Intel

Microchip PIC16C5x

Default Integer Syntax: Motorola

```
DATA        RES         SFR         ZERO
```

Microchip PIC16C8x

Default Integer Syntax: Motorola

```
DATA        RES         SFR         ZERO
```

Microchip PIC17C42

Default Integer Syntax: Motorola

```
DATA        RES         SFR         ZERO
```

Parallax SX20

Default Integer Syntax: Motorola

```
BIT         DATA        SFR         ZERO
```

SGS-Thomson ST6

Default Integer Syntax: Intel

```
ASCII       ASCIZ       ASSUME      BIT         BYTE
BLOCK       SFR         WORD
```

SGS-Thomson ST7/STM8

Default Integer Syntax: Intel

DC[.<size>] DS[.<size>] PADDING

SGS-Thomson ST9

Default Integer Syntax: Intel

ASSUME	BF16	BIT	DB	DD
DN	DO	DQ	DS	DT
DW	REG			

6804

Default Integer Syntax: Motorola

ADR	BF16	BYT	DB	DFS
DS	DW	FCB	FCC	FDB
RMB	SFR			

Texas Instruments TMS3201x

Default Integer Syntax: Intel

DATA	PORT	RES

Texas Instruments TMS32C02x

Default Integer Syntax: Intel

BFLOAT	BSS	BYTE	DATA	DOUBLE
EFLOAT	TFLOAT	LONG	LQxx	PORT
Qxx	RES	RSTRING	STRING	WORD

Texas Instruments TMS320C3x/C4x

Default Integer Syntax: Intel

ASSUME	BSS	DATA	EXTENDED	PACKING
SINGLE	WORD			

Texas Instruments TM32C020x/TM32C05x/TM32C054x

Default Integer Syntax: Intel

BFLOAT	BSS	BYTE	DATA	DOUBLE
EFLOAT	TFLOAT	LONG	LQxx	PORT
Qxx	RES	RSTRING	STRING	WORD

Texas Instruments TMS320C6x

Default Integer Syntax: Intel

BSS	DATA	DOUBLE	PACKING	SINGLE
WORD				

Texas Instruments TMS340xx

Default Integer Syntax: Intel

BES	BSS	BYTE	COPROC	DOUBLE
EVEN	FIELD	FLOAT	LONG	REG
SPACE	WORD			

Texas Instruments TMS99xx

Default Integer Syntax: Intel

BSS	BYTE	DOUBLE	PADDING	SINGLE
WORD				

Texas Instruments Instruments TMS1000

Default Integer Syntax: Intel

BF16	DB	DD	DN	DO
DQ	DS	DT	DW	

Texas Instruments TMS70Cxx

Default Integer Syntax: Intel

BF16	DB	DD	DN	DO
DQ	DS	DT	DW	

Texas Instruments TMS370

Default Integer Syntax: Intel

BF16	DB	DBIT	DD	DN
DO	DQ	DS	DT	DW

Texas Instruments MSP430

Default Integer Syntax: Intel

BSS	BYTE	PADDING	REG	WORD

National IMP-16

Default Integer Syntax: IBM

ASCII	LTORG	PORT	WORD

National IPC-16/INS8900

Default Integer Syntax: IBM

ASCII	ASSUME	LTORG	WORD

National SC/MP

Default Integer Syntax: C

ASCII	BF16	BIGENDIAN	BYTE	DB
DBYTE	DD	DN	DO	DQ
DS	DT	DW	REG	

National INS807x

Default Integer Syntax: C

BF16	DB	DD	DN	DO
DQ	DS	DT	DW	

National COP4

Default Integer Syntax: C

ADDR	ADDRW	BF16	BYTE	DB
DD	DO	DQ	DS	DSB
DSW	DT	DW	FB	FW
SFR	WORD			

National COP8

Default Integer Syntax: C

ADDR	ADDRW	BF16	BYTE	DB
DD	DQ	DS	DSB	DSW
DT	DW	FB	FW	SFR
WORD				

National SC14xxx

Default Integer Syntax: C

DC	DC8	DS	DS8	DS16
DW	DW16			

National NS32xxx

BF16	BIGENDIAN	BYTE	CUSTOM	DB
DD	DOUBLE	DO	DQ	DS
DT	DW	FLOAT	FPU	LONG
PMMU	REG	SUPMODE	WORD	

National CR16A/B

Default Integer Syntax: C

BLKB	BLKD	BLKF	BLKL	BLKW
BYTE	DC16	DC24	DC32	DC64
DC8	DF32	DF64	DOUBLE	DS16
DS32	DS64	DS8	FLOAT	LONG
REG	SPACE	WORD		

National CR16C

Default Integer Syntax: C

BLKB	BLKD	BLKF	BLKL	BLKW
BYTE	DC16	DC24	DC32	DC64
DC8	DF32	DF64	DOUBLE	DS16
DS32	DS64	DS8	FLOAT	LONG
REG	SPACE	SUPMODE	WORD	

Fairchild ACE

Default Integer Syntax: Intel

BF16	DB	DD	DN	DO
DQ	DS	DT	DW	

Fairchild F8

Default Integer Syntax: Intel

BF16	DB	DD	DN	DO
DQ	DS	DT	DW	PORT

NEC μPD7800...μPD7806, μPD78(C)1x

Default Integer Syntax: Intel

ASSUME	BF16	DB	DD	DN
DO	DQ	DS	DT	DW
Z80SYNTAX				

NEC μPD7807...μPD7809

Default Integer Syntax: Intel

ASSUME	BF16	DB	DEFBIT	DD
DN	DO	DQ	DS	DT
DW				

NEC 75xx

Default Integer Syntax: Intel

BF16	DB	DD	DN	DO
DQ	DS	DT	DW	

NEC μCOM-43

Default Integer Syntax: Intel

BF16	DB	DD	DN	DO
DQ	DS	DT	DW	

NEC 75K0

Default Integer Syntax: Intel

ASSUME	BF16	BIT	DB	DD
DN	DO	DQ	DS	DT
DW	SFR			

NEC 78K0

Default Integer Syntax: Intel

BF16	DB	DD	DN	DO
DQ	DS	DT	DW	

NEC 78K2

Default Integer Syntax: Intel

BF16	BIT	DB	DD	DN
DO	DQ	DS	DT	DW

NEC 78K3

Default Integer Syntax: Intel

BF16	BIT	DB	DD	DN
DO	DQ	DS	DT	DW

NEC 78K4

Default Integer Syntax: Intel

BF16	BIT	DB	DD	DN
DO	DQ	DS	DT	DW

NEC μPD772x

Default Integer Syntax: Intel

DATA	PACKING	RES

NEC μPD77230

Default Integer Syntax: Intel

DS	DW	PACKING

NEC V60

Default Integer Syntax: Intel

DC[.<size>]	DS[.<size>]	PADDING	REG	SUPMODE

Symbios Logic SYM53C8xx

Default Integer Syntax: C

BF16	DB	DD	DN	DO
DQ	DS	DT	DW	

Fujitsu F^{2}MC8L

Default Integer Syntax: Intel

BF16	DB	DD	DN	DO
DQ	DS	DT	DW	

Fujitsu F^{2}MC16L

Default Integer Syntax: Intel

BF16	DB	DD	DN	DO
DQ	DS	DT	DW	

OKI OLMS-40

Default Integer Syntax: Intel

DATA	RES	SFR

OKI OLMS-50

Default Integer Syntax: Intel

DATA	RES	SFR

Panafacom MN161x

Default Integer Syntax: IBM

DC	DS	PACKING

Padauk PMC/PMS/PFSxxx

Default Integer Syntax: C

BIT	DATA	RES	SFR

Intersil 180x

Default Integer Syntax: Intel

BF16	DB	DD	DN	DO
DQ	DS	DT	DW	

XMOS XS1

Default Integer Syntax: Motorola

BF16	DB	DD	DQ	DN
DO	DS	DT	DW	REG

ATARI Vector

Default Integer Syntax: Motorola

MIL STD 1750

Default Integer Syntax: Intel

DATA	EXTENDED	FLOAT

KENBAK

Default Integer Syntax: Intel

BF16	BIT	DB	DD	DN
DO	DQ	DS	DT	DW
REG				

CP-1600

Default Integer Syntax: IBM (hex), C (oct)

BYTE	PACKING	REG	RES	TEXT
WORD	ZERO			

HP Nano Processor

Default Integer Syntax: C

IM61x0

Default Integer Syntax: C

DC	DECIMAL	DS	LTORG	OCTAL
ZERO				

Renesas RX

Default Integer Syntax: Intel

BLKB	BLKW	BLKL	BLKD	BYTE
DOUBLE	ENDIAN	FLOAT	LWORD	WORD

Sharp SC61860

Default Integer Syntax: Motorola

ADR	BYT	DB	DC	DFS
DS	DW	FCB	FCC	FDB
RMB				

Sharp SC62015

Default Integer Syntax: Intel

BF16	DN	DB	DD	DO
DQ	DS	DT	DW	

TBIL

Default Integer Syntax: Motorolal

ADR	BYT	DFS	DW	FCB
FCC	FDB	RMB		

Anhang F

Vordefinierte Symbole

Name	Datentyp	Definition	Bedeutung
ARCHITECTURE	String	vordef.	Zielplattform, für die AS übersetzt wurde, in der Form Prozesor-Hersteller-Betriebs-system
BIGENDIAN	Boolean	normal	Konstantenablage mit MSB first ?
CASESENSITIVE	Boolean	normal	Unterscheidung von Groß- und Kleinbuchstaben in Symbolnamen ?
CONSTPI	Gleitkomma	normal	Kreiszahl Pi (3.1415.....)
DATE	String	vordef.	Datum des Beginns der Assemblierung (1.Pass)
FALSE	Boolean	vordef.	0 = logisch „falsch”
HASFPU	Boolean	dynam.(0)	Koprozessor-Befehle freigeschaltet ?
HASPMMU	Boolean	dynam.(0)	MMU-Befehle frei-geschaltet ?
INEXTMODE	Boolean	dynam.(0)	XM-Flag für 4 Gbyte Adreßraum gesetzt ?
INLWORDMODE	Boolean	dynam.(0)	LW-Flag für 32-Bit-Befehle gesetzt ?
INMAXMODE	Boolean	dynam.(0)	Prozessor im Maximum-Modus ?
INSUPMODE	Boolean	dynam.(0)	Prozessor im Supervisor-

Name	Datentyp	Definition	Bedeutung
			Modus ?
INSRCMODE	Boolean	dynam.(0)	Prozessor im Quellmodus ?
FULLPMMU	Boolean	dynam.(0/1)	voller PMMU-Befehlssatz ?
LISTON	Boolean	dynam.(1)	Listing freigeschaltet ?
MACEXP	Boolean	dynam.(1)	Expansion von Makrokonstrukten im Listing freigeschaltet ?
MOMCPU	Integer	dynam. (68008)	Nummer der momentan gesetzten Ziel-CPU
MOMCPUNAME	String	dynam. (68008)	Name der momentan gesetzten Ziel-CPU
MOMFILE	String	Spezial	augenblickliche Quelldatei (schließt Includes ein)
MOMLINE	Integer	Spezial	aktuelle Zeilennummer in der Quelldatei
MOMPASS	Integer	Spezial	Nummer des laufenden Durchgangs
MOMSECTION	String	Spezial	Name der aktuellen Sektion oder Leerstring, fall außerhalb aller Sektionen
MOMSEGMENT	String	Spezial	Name des mit SEGMENT eingestellten Adreßraumes
NESTMAX	Integer	dynam.(256)	maximale Verschachtelungstiefe für Makros
PADDING	Boolean	dynam.(1)	Auffüllen von Bytefeldern auf ganze Anzahl ?
RELAXED	Boolean	dynam.(0)	Schreibweise von Integer-Konstanten in beliebiger Syntax erlaubt ?
PC	Integer	Spezial	mom. Programmzähler (Thomson)
TIME	String	vordef.	Zeit des Beginns der Assemblierung (1. Pass)
TRUE	Integer	vordef.	1 = logisch „wahr”
VERSION	Integer	vordef.	Version von AS in BCD-Kodierung, z.B. 1331 hex für Version 1.33p1
WRAPMODE	Integer	vordef.	verkürzter Programmzähler angenommen?

Name	Datentyp	Definition	Bedeutung
*	Integer	Spezial	mom. Programmzähler (Motorola, Rockwell, Microchip, Hitachi)
.	Integer	Spezial	mom. Programmzähler (IM61x0)
$	Integer	Spezial	mom. Programmzähler (Intel, Zilog, Texas, Toshiba, NEC, Siemens, AMD)

Boolean-Symbole sind eigentlich normale normale Integer-Symbole, mit dem Unterschied, daß ihnen von AS nur zwei verschiedene Werte (0 oder 1, entsprechend FALSE oder TRUE) zugewiesen werden. Spezialsymbole werden von AS nicht in der Symboltabelle abgelegt, sondern aus Geschwindigkeitsgründen direkt im Parser abgefragt. Sie tauchen daher auch nicht in der Symboltabelle des Listings auf. Während vordefinierte Symbole nur einmal am Anfang eines Passes besetzt werden, können sich die Werte dynamischer Symbole während der Assemblierung mehrfach ändern, da sie mit anderen Befehlen vorgenommene Einstellungen widerspiegeln.

Die hier aufgelistete Schreibweise ist diejenige, mit der man die Symbole auch im case-sensitiven Modus erreicht.

Die hier aufgeführten Namen sollte man für eigene Symbole meiden; entweder kann man sie zwar definieren, aber nicht darauf zugreifen (bei Spezialsymbolen), oder man erhält eine Fehlermeldung wegen eines doppelt definierten Symboles. Im gemeinsten Fall führt die Neubelegung durch AS zu Beginn eines Passes zu einem Phasenfehler und einer Endlosschleife...

Anhang G

Mitgelieferte Includes

Der Distribution von AS liegen eine Reihe von Include-Dateien bei. Neben Includes, die sich nur auf eine Prozessorfamilie beziehen (und deren Funktion sich demjenigen unmittelbar erschließt, der mit dieser Familie arbeitet), existieren aber auch ein paar Dateien, die prozessorunabhängig sind und die eine Reihe nützlicher Funktionen implementieren. Die definierten Funktionen sollen hier kurz beschrieben werden:

G.1 BITFUNCS.INC

Diese Datei definiert eine Reihe bitorientierter Operationen, wie man sie bei anderen Assemblern vielleicht fest eingebaut sind. Bei AS werden sie jedoch mit Hilfe benutzerdefinierter Funktionen implementiert:

- *mask(start,bits)* liefert einen Integer, in dem ab Stelle *start bits* Bits gesetzt sind;
- *invmask(start,bits)* liefert das Einerkomplement zu *mask()*;
- *cutout(x,start,bits)* liefert ausmaskierte *bits* Bits ab Stelle *start* aus x, ohne sie auf Stelle 0 zu verschieben;
- *hi(x)* liefert das zweitniedrigste Byte (Bit 8..15) aus x;
- *lo(x)* liefert das niederwertigste Byte (Bit 0..7) aus x;
- *hiword(x)* liefert das zweitniedrigste Wort (Bit 16..31) aus x;

- *loword(x)* liefert das niederwertigste Wort (Bit 0..15) aus x;
- *odd(x)* liefert TRUE, falls x ungerade ist;
- *even(x)* liefert TRUE, falls x gerade ist;
- *getbit(x,n)* extrahiert das Bit n aus x und liefert es als 0 oder 1;
- *shln(x,size,n)* schiebt ein Wort x der Länge *size* Bits um n Stellen nach links;
- *shrn(x,size,n)* schiebt ein Wort x der Länge *size* Bits um n Stellen nach rechts;
- *rotln(x,size,n)* rotiert die untersten *size* Bits eines Integers x um n Stellen nach links;
- *rotrn(x,size,n)* rotiert die untersten *size* Bits eines Integers x um n Stellen nach rechts;

G.2 CTYPE.INC

Dieser Include ist das Pendant zu dem bei C vorhandenen Header `ctype.h`, der Makros zur Klassifizierung von Zeichen anbietet. Alle Funktionen liefern entweder TRUE oder FALSE:

- *isdigit(ch)* ist TRUE, falls *ch* eine Ziffer (0..9) ist;
- *isxdigit(ch)* ist TRUE, falls *ch* eine gültige Hexadezimal-Ziffer (0..9, A..F, a..f) ist;
- *isascii(ch)* ist TRUE, falls *ch* sich im Bereich normaler ASCII-Zeichen ohne gesetztes Bit 7 bewegt;
- *isupper(ch)* ist TRUE, falls *ch* ein Großbuchstabe ist (Sonderzeichen ausgenommen);
- *islower(ch)* ist TRUE, falls *ch* ein Kleinbuchstabe ist (Sonderzeichen ausgenommen);
- *isalpha(ch)* ist TRUE, falls *ch* ein Buchstabe ist (Sonderzeichen ausgenommen);
- *isalnum(ch)* ist TRUE, falls *ch* ein Buchstabe oder eine Ziffer ist);

- *isspace(ch)* ist TRUE, falls *ch* ein 'Leerzeichen' (Space, Formfeed, Zeilenvorschub, Wagenrücklauf, Tabulator) ist);
- *isprint(ch)* ist TRUE, falls *ch* ein druckbares Zeichen ist (also kein Steuerzeichen bis Code 31);
- *iscntrl(ch)* ist das Gegenteil zu *isprint()*;
- *isgraph(ch)* ist TRUE, falls *ch* ein druckbares und *sichtbares* Zeichen ist;
- *ispunct(ch)* ist TRUE, falls *ch* ein druckbares Sonderzeichen ist (d.h. weder Space, Buchstabe noch Ziffer);

Anhang H

Danksagungen

"If I have seen farther than other men,
it is because I stood on the shoulders of giants."
–Sir Isaac Newton

"If I haven't seen farther than other men,
it is because I stood in the footsteps of giants."
–unknown

Es heißt ja bisweilen, die Programme, die man schreibe, wären so etwas wie eigene Kinder, die man in die Welt setzt. Ich arbeite an diesem Assembler seit über 30 Jahren, und bin mittlerweile der Ansicht, so ein Projekt ist viel mehr eine Reise, auf die man sich begibt: Die Begegnungen mit Menschen, die man auf den Stationen dieses Weges trifft, sind mindestens eben so wichtig wie das vermeintliche Ziel selber. Man lernt Neues kennen, und versteht im besten Falle, daß man Dinge auch aus einer ganz anderen Warte sehen kann. Wenn es gut läuft, bringt das dann beide Seiten weiter.

Einige Menschen sind mir auf diesem Weg besonders in Erinnerung geblieben, weil sie auf ihre Weise dazu beigetragen haben, dieses Projekt dorthin zu bringen, wo es jetzt ist. Die folgende Aufzählung ist notwendigerweise unvollständig, einfach weil ich mich nach den Jahren nicht mehr an jede Begebenheit erinnere. Der zuvorderste Dank gent daher an all die Personen, deren Namen ich in diesem Kapitel unabsichtlich unterschlagen habe. Die Reise geht immer noch weiter, und vielleicht kreuzen sich die Wege wieder!

AS als Universalassembler, wie er jetzt besteht, ist auf Anregung von Bernhard (C.) Zschocke entstanden, der einen ,,studentenfreundlichen", d.h. kostenlosen 8051-Assembler für sein Mikroprozessorpraktikum brauchte und mich dazu bewegt hat,

einen bereits bestehenden 68000-Assembler zu erweitern. Von dortan nahm die Sache ihren Lauf... Das Mikroprozessorpraktikum an der RWTH Aachen hat auch immer die eifrigsten Nutzer der neuesten AS-Features (und damit Bug-Sucher) gestellt und damit einiges zur jetzigen Qualität von AS beigetragen.

Das Internet und FTP haben sich als große Hilfe bei der Meldung von Bugs und der Verbreitung von AS erwiesen. Ein Dank geht daher an die FTP-Administratoren (Bernd Casimir in Stuttgart, Norbert Breidohr in Aachen und Jürgen Meißburger in Jülich). Insbesondere letzterer hat sich sehr engagiert, um eine praxisnahe Lösung im ZAM zu finden.

Ach ja, wo wir schon im ZAM sind: Wolfgang E. Nagel hat zwar nichts direkt mit AS zu tun, immerhin war er aber mein Betreuer und Vorgesetzter und warf ständig vier Augen auf das, was ich tue. Bei AS scheint zumindest ein lachendes dabei zu sein...

Ein Projekt wie AS ist ohne Input und Informationen nicht zu leisten, sei es zu den unterstützten Prozessoren selber, zu Programmiertechniken oder wie Dinge implementiert werden sollten. Ich habe von einer enormen Anzahl von Menschen Rückmeldungen bekommen, die von einem kleinen Tip bis zu ganzen Datenbüchern reichen. Hier eine Aufzählung (wie wie oben gesagt, mit Sicherheit unvollständig ist!):

Ernst Ahlers, Charles Altmann, Marco Awater, Len Bayles, Andreas Bolsch, Rolf Buchholz, Bernd Casimir, Nils Eilers, Jeff Epler, Gunther Ewald, Daniel E. Germann, Michael Haardt, Stephan Hruschka, Fred van Kempen, Peter Kliegelhöfer, Ulf Meinke, Udo Möller, Matthias Paul, Moshe Piekarski, Norbert Rosch, Curt J. Sampson, Steffen Schmid, Leonhard Schneider, Ernst Schwab, Michael Schwingen, Oliver Sellke, Christian Stelter, Patrik Strömdahl, Tadashi G. Takaoka, Oliver Thamm, Thorsten Thiele, Leszek Ulman, Rob Warmelink, Andreas Wassatsch, John Weinrich,

...und ein leicht ironischer Dank geht an Rolf-Dieter-Klein und Tobias Thiel, die mich mit ihren ASM68K dazu angeregt haben, überhaupt einen eigenen Assembler zu schreiben. einige Dinge darin funktionierten nicht so, wie ich es mir vorgestellt hatte, und das war der Anlass, es besser oder zumindest anders zu machen.

So ganz allein habe ich AS auch nicht geschrieben. Die DOS-Version von AS enthielt die OverXMS-Routinen von Wilbert van Leijen, um die Overlay-Module ins Extended Memory verlagern zu können. Eine wirklich feine Sache, einfach und problemlos anzuwenden!

Die TMS320C2x/5x-Codegeneratoren sowie die Datei `STDDEF2x.INC` stammen von Thomas Sailer, ETH Zürich. Erstaunlich, an einem Wochenende hat er es geschafft, durch meinen Code durchzusteigen und den neuen Generator zu implementieren. Entweder waren das reichliche Nachtschichten oder ich werde langsam

alt...gleiches Lob gebührt Haruo Asano für die Targets MN1610/MN1613, IM6100, CP1600, Renesas RX und HP NanoProcessor.

Anhang I

Änderungen seit Version 1.3

- Version 1.31:
  - zusätzlicher MCS-51-Prozessortyp 80515. Die Nummer wird wiederum nur vom Assembler verwaltet. Die Datei STDDEF51.INC wurde um die dazugehörigen SFRs erweitert. **ACHTUNG!** Einige 80515-SFRs haben sich adreßmäßig verschoben!
  - zusätzlich Prozessor Z80 unterstützt;
  - schnellerer 680x0-Codegenerator.
- Version 1.32:
  - Schreibweise von Zeropageadressen für 65xx nicht mehr als Adr.z, sondern wie beim 68xx als <Adr;
  - unterstützt die Prozessoren 6800, 6805, 6301 und 6811;
  - der 8051-Teil versteht jetzt auch `DJNZ`, `PUSH` und `POP` (sorry);
  - im Listing werden neben den Symbolen jetzt auch die definierten Makros aufgelistet;
  - Befehle `IFDEF/IFNDEF` für bedingte Assemblierung, mit denen sich die Existenz eines Symboles abfragen läßt;
  - Befehle `PHASE/DEPHASE` zur Unterstützung von Code, der zur Laufzeit auf eine andere Adresse verschoben werden soll;
  - Befehle `WARNING/ERROR/FATAL`, um anwenderspezifische Fehlermeldungen ausgeben zu können;
  - Die Datei STDDEF51.INC enthält zusätzlich das Makro `USING` zur einfacheren Handhabung der Registerbänke der MCS-51er;

  - Kommandozeilenoption `u`, um Segmentbelegung anzuzeigen.

- Version 1.33:
  - unterstützt den 6809;
  - zusätzlich Stringvariablen;
  - Die Befehle `TITLE`, `PRTINIT`, `PRTEXIT`, `ERROR`, `WARNING` und `FATAL` erwarten jetzt einen Stringausdruck als Parameter, Konstanten müssen demzufolge nicht mehr in Hochkommas, sondern in Gänsefüßchen eingeschlossen werden. Analoges gilt für `DB`, `DC.B` und `BYT`;
  - Befehl `ALIGN` zur Ausrichtung des Programmzählers bei Intel- Prozessoren;
  - Befehl `LISTING`, um die Erzeugung eines Listings ein- und ausschalten zu können;
  - Befehl `CHARSET` zur Definition eigener Zeichensätze.

- Version 1.34:
  - Wenn im ersten Pass Fehler auftreten, wird gar kein zweiter Pass mehr durchgeführt;
  - neues vordefiniertes Symbol `VERSION`, welches die Version von AS enthält;
  - Befehl `MESSAGE`, um Durchsagen und Meldungen programmgesteuert zu erzeugen;
  - Formelparser über Stringkonstanten zugänglich;
  - Bei Fehler in Makroexpansionen wird zusätzlich die laufende Zeile im Makro angezeigt;
  - Funktion `UPSTRING`, um einen String in Großbuchstaben zu wandeln.

- Version 1.35:
  - Funktion `TOUPPER`, um ein einzelnes Zeichen in Großbuchstaben zu wandeln;
  - Befehl `FUNCTION`, um eigene Funktionen definieren zu können;
  - Kommandozeilenoption `D`, um Symbole von außen definieren zu können;
  - Fragt die Environment-Variable `ASCMD` für häufig gebrauchte Optionen ab;
  - bei gesetzter `u`-Option wird das Programm zusätzlich auf doppelt belegte Speicherbereiche abgeprüft;

  - Kommandozeilenoption `C`, um eine Querverweisliste zu erzeugen.

- Version 1.36:
  - unterstützt zusätzlich die Familien PIC 16C5x und PIC17C4x;
  - im Listing wird zusätzlich die Verschachtelungsebene bei Include-Dateien angezeigt;
  - in der Querverweisliste wird zusätzlich die Stelle angezeigt, an der ein Symbol definiert wurde;
  - Kommandozeilenoption `A`, um eine kompaktere Ablage der Symboltabelle zu erzwingen.

- Version 1.37:
  - unterstützt zusätzlich die Prozessoren 8086, 80186, V30, V35, 8087 und Z180;
  - Befehle `SAVE` und `RESTORE` zur besseren Umschaltung von Flags;
  - Operatoren zur logischen Verschiebung und Bitspiegelung;
  - Kommandozeilenoptionen können mit einem Pluszeichen negiert werden;
  - Filter `AS2MSG` zur bequemen Arbeit mit AS unter Turbo-Pascal 7.0;
  - `ELSEIF` darf ein Argument zur Bildung von `IF-THEN-ELSE`-Leitern haben;
  - Zur bequemeren bedingten Assemblierung zusätzlich ein `CASE`-Konstrukt;
  - Selbst definierte Funktionen dürfen mehr als ein Argument haben;
  - P2HEX kann nun auch Hexfiles für 65er-Prozessoren erzeugen;
  - BIND, P2HEX und P2BIN haben jetzt die gleichen Variationsmöglichkeiten in der Kommandozeile wie AS;
  - Schalter `i` bei P2HEX, um 3 Varianten für den Ende-Record einzustellen;
  - Neue Funktionen `ABS` und `SGN`;
  - Neue Pseudovariablen `MOMFILE` und `MOMLINE`;
  - Ausgabemöglichkeit erweiterter Fehlermeldungen;
  - Befehle `IFUSED` und `IFNUSED`, um abzufragen, ob ein Symbol bisher benutzt wurde;
  - Die Environment-Variablen `ASCMD`, `BINDCMD` usw. können auch einen Dateinamen enthalten, in dem für die Optionen mehr Platz ist;
  - P2HEX erzeugt nun die von Microchip vorgegebenen Hex-Formate (p4);

  - mit der Seitenlängenangabe 0 können automatische Seitenvorschübe im Listing vollständig unterdrückt werden (p4);
  - neue Kommandozeilenoption `P`, um die Ausgabe des Makroprozessors in eine Datei zu schreiben (p4);
  - in der Kommandozeile definierte Symbole dürfen nun auch mit einem frei wählbaren Wert belegt werden (p5).

- Version 1.38:
  - Umstellung auf Mehrpass-Betrieb. Damit kann AS auch bei Vorwärtsreferenzen immer den optimalen Code erzeugen;
  - Der 8051-Teil kennt nun auch die Befehle `JMP` und `CALL`;
  - unterstützt zusätzlich die Toshiba TLCS-900-Reihe (p1);
  - Befehl `ASSUME`, um dem Assembler die Belegung der Segmentregister des 8086 mitzuteilen (p2);
  - unterstützt zusätzlich die ST6-Reihe von SGS-Thomson (p2);
  - ..sowie die 3201x-Signalprozessoren von Texas Instruments (p2);
  - Option `F` bei P2HEX, um die automatische Formatwahl übersteuern zu können (p2);
  - P2BIN kann nun auch durch Angabe von Dollarzeichen Anfang und Ende des Adreßfensters selbstständig festlegen (p2);
  - Der 8048-Codegenerator kennt nun auch die 8041/42- Befehlserweiterungen(p2);
  - unterstützt zusätzlich die Zilog Z8-Mikrokontroller(p3).

- Version 1.39:
  - Definitionsmöglichkeit von Sektionen und lokalen Labels;
  - Kommandozeilenschalter `h`, um Hexadezimalzahlenausgabe mit Kleinbuchstaben zu erzwingen;
  - Variable `MOMPASS`, um die Nummer des augenblicklichen Durchganges abfragen zu können;
  - Kommandozeilenschalter `t`, um einzelne Teile des Assembler-Listings ausblenden zu können;
  - kennt zusätzlich die L-Variante der TLCS-900-Reihe von Toshiba und die MELPS-7700-Reihe von Mitsubishi (p1);
  - P2HEX akzeptiert nun auch Dollarzeichen für Start-und Endadresse (p2);

- unterstützt zusätzlich die TLCS90-Familie von Toshiba (p2);
- P2HEX kann Daten zusätzlich im Tektronix- und 16-Bit Intel-Hex-Format ausgeben (p2);
- bei Adreßüberschreitungen gibt P2HEX Warnungen aus (p2);
- Include-Datei STDDEF96.INC mit Adreßdefinitionen für die TLCS-900-Reihe (p3);
- Befehl `READ`, um Werte während der Assemblierung interaktiv einlesen zu können (p3);
- Fehlermeldungen werden nicht mehr einfach auf die Standardausgabe, sondern auf den von DOS dafür vorgesehenen Kanal (STDERR) geschrieben (p3);
- Der beim 6811-Teil fehlende `STOP`-Befehl ist nun da (scusi,p3);
- unterstützt zusätzlich die μPD78(C)1x-Familie von NEC (p3);
- unterstützt zusätzlich den PIC16C84 von Microchip (p3);
- Kommandozeilenschalter `E`, um die Fehlermeldungen in eine Datei umleiten zu können (p3);
- Die Unklarheiten im 78(C)1x-Teil sind beseitigt (p4);
- neben dem MELPS-7700 ist nun auch das „Vorbild" 65816 vorhanden (p4);
- Die ST6-Pseudoanweisung `ROMWIN` wurde entfernt und mit in den `ASSUME`-Befehl eingegliedert (p4);
- unterstützt zusätzlich den 6804 von SGS-Thomson (p4);
- durch die `NOEXPORT`-Option in der Makrodefinition kann nun für jedes Makro einzeln festgelegt werden, ob es in der MAC-Datei erscheinen soll oder nicht (p4);
- Die Bedeutung von `MACEXP` für Expansionen von Makros hat sich wegen der zusätzlichen `NOEXPAND`-Option in der Makrodefinition leicht geändert (p4);
- Durch die `GLOBAL`-Option in der Makrodefinition können nun zusätzlich Makros definiert werden, die durch ihren Sektionsnamen eindeutig gekennzeichnet sind (p4).

• Version 1.40:

- unterstützt zusätzlich den DSP56000 von Motorola;

- P2BIN kann nun auch das untere bzw. obere Wort aus 32-Bit-Wörtern abtrennen;
- unterstützt zusätzlich die TLCS-870- und TLCS-47-Familie von Toshiba(p1);
- mit einem vorangestelltem ! kann man durch Makros ,,verdeckte" Maschinenbefehle wieder erreichen(p1);
- mit der `GLOBAL`-Anweisung lassen sich Symbolnamen nun auch qualifiziert exportieren(p1);
- mit der `r`-Option kann man sich nun eine Liste der Stellen erzeugen lassen, die zusätzliche Durchläufe erzwangen(p1);
- bei der `E`-Option kann nun die Dateiangabe weggelassen werden, so daß ein passender Default gewählt wird(p1);
- mit der `t`-Option kann nun die Zeilennumerierung im Listing abgeschaltet werden(p1);
- Escapesequenzen sind nun auch in in ASCII geschriebenen Integer-Konstanten zulässig(p1);
- Mit dem Pseudobefehl `PADDING` kann das Einfügen von Füllbytes im 680x0-Modus ein- und ausgeschaltet werden (p2);
- `ALIGN` ist nun für alle Zielplattformen erlaubt (p2);
- kennt zusätzlich die PIC16C64-SFRs (p2);
- unterstützt zusätzlich den 8096 von Intel (p2);
- Bei `DC` kann zusätzlich ein Wiederholungsfaktor angegeben werden (r3);
- unterstützt zusätzlich die TMS320C2x-Familie von Texas Instruments (Implementierung von Thomas Sailer, ETH Zürich, r3); P2HEX ist auch entsprechend erweitert;
- statt `EQU` darf nun auch einfach ein Gleichheitszeichen benutzt werden (r3);
- zur Definition von Aufzählungen zusätzlich ein `ENUM`-Befehl (r3);
- `END` hat jetzt auch eine Wirkung (r3);
- zusätzliche Kommandozeilenoption `n`, um zu Fehlermeldungen zusätzlich die internen Fehlernummern zu erhalten (r3);
- unterstützt zusätzlich die TLCS-9000er von Toshiba (r4);
- unterstützt zusätzlich die TMS370xxx-Reihe von Texas Instuments, wobei als neuer Pseudobefehl `DBIT` hinzukam (r5);

- kennt zusätzlich die DS80C320-SFRs (r5);
- der Makroprozessor kann nun auch Includes aus Makros heraus einbinden, wozu das Format von Fehlermeldungen aber leicht geändert werden mußte. Falls Sie AS2MSG verwenden, ersetzen Sie es unbedingt durch die neue Version! (r5)
- unterstützt zusätzlich den 80C166 von Siemens (r5);
- zusätzlich eine `VAL`-Funktion, um Stringausdrücke auswerten zu können (r5);
- Mithilfe von in geschweiften Klammern eingeschlossenen Stringvariablen lassen sich nun selber Symbole definieren (r5);
- kennt zusätzlich die Eigenheiten des 80C167 von Siemens (r6);
- jetzt gibt es für die MELPS740-Reihe auch die special-page-Adressierung (r6);
- mit eckigen Klammern kann man explizit Symbole aus einer bestimmten Sektion ansprechen. Die Hilfskonstruktion mit dem Klammeraffen gibt es nicht mehr (r6)!
- kennt zusätzlich die MELPS-4500-Reihe von Mitsubishi (r7);
- kennt zusätzlich die H8/300 und H8/300H-Prozessoren von Hitachi (r7);
- die mit `LISTING` und `MACEXP` gemachten Einstellungen lassen sich nun auch wieder aus gleichnamigen Symbolen auslesen (r7);
- kennt zusätzlich den TMS320C3x von Texas Instruments (r8);
- kennt zusätzlich den SH7000 von Hitachi (r8);
- der Z80-Teil wurde um die Unterstützung des Z380 erweitert (r9);
- der 68K-Teil wurde um die feinen Unterschiede der 683xx-Mikrokontroller erweitert (r9);
- ein Label muß nun nicht mehr in der ersten Spalte beginnen, wenn man es mit einem Doppelpunkt versieht (r9);
- kennt zusätzlich die 75K0-Reihe von NEC (r9);
- mit dem neuen Kommandozeilenschalter o kann der Name der Code-Datei neu festgelegt werden (r9);
- der `~~`-Operator ist in der Rangfolge auf einen sinnvolleren Platz gerutscht (r9);
- `ASSUME` berücksichtigt für den 6809 jetzt auch das DPR-Register und seine Auswirkungen (pardon, r9);

  - Der 6809-Teil kennt nun auch die versteckten Erweiterungen des 6309 (r9);
  - Binärkonstanten können jetzt auch in C-artiger Notation geschrieben werden (r9).

- Version 1.41:
  - über das Symbol `MOMSEGMENT` kann der momentan gesetzte Adreßraum abgefragt werden;
  - anstelle von `SET` bzw. `EVAL` kann jetzt auch einfach `:=` geschrieben werden;
  - mit der neuen Kommandozeilenoption `q` kann ein ,,stiller" Assemblerlauf erzwungen werden;
  - das Schlüsselwort `PARENT` zum Ansprechen der Vatersektion wurde um `PARENT0...PARENT9` erweitert;
  - der PowerPC-Teil wurde um die Mikrokontroller-Versionen MPC505 und PPC403 erweitert;
  - mit `SET` oder `EQU` definierte Symbole können nun einem bestimmten Adreßraum zugeordnet werden;
  - mit `SET` oder `EQU` definierte Symbole können nun einem bestimmten Adreßraum zugeordnet werden;
  - durch das Setzen der Environment-Variablen `USEANSI` kann die Verwendung von ANSI-Bildschirmsteuersequenzen an-und ausgeschaltet werden (r1);
  - der SH7000-Teil kennt jetzt auch die SH7600-Befehlserweiterungen (und sollte jetzt korrekte Displacements berechnen...) (r1).
  - im 65XX-Teil wird jetzt zwischen 65C02 und 65SC02 unterschieden (r1);
  - neben der Variablen `MOMCPU` gibt es jetzt auch den String `MOMCPUNAME`, der den Prozessornamen im Volltext enthält (r1).
  - P2HEX kennt jetzt auch die 32-Bit-Variante des Intel-Hex-Formates (r1);
  - kennt jetzt auch die Einschränkungen des 87C750 (r2);
  - die Nummern für fatale Fehlermeldungen wurden auf den Bereich ab 10000 verschoben, um Platz für normale Fehlermeldungen zu schaffen (r2);
  - unbenutzte Symbole werden in der Symboltabelle jetzt mit einem Stern gekennzeichnet (r2);
  - unterstützt zusätzlich die 29K-Familie von AMD (r2);

- unterstützt zusätzlich die M16-Familie von Mitsubishi (r2);
- unterstützt zusätzlich die H8/500-Familie von Hitachi (r3);
- die Anzahl von Datenbytes, die P2HEX pro Zeile ausgibt, ist jetzt variierbar (r3);
- der Pass, ab dem durch die `-r`-Option erzeugte Warnungen ausgegeben werden, ist einstellbar (r3);
- der Makroprozessor kennt jetzt ein `WHILE`-Statement, mit dem ein Code-Stück eine variable Anzahl wiederholt werden kann (r3);
- der `PAGE`-Befehl erlaubt es nun auch, die Breite des Ausgabemediums fürs Listing anzugeben (r3);
- Um neue Pseudo-Prozessortypen einführen zu können, lassen sich jetzt CPU-Aliasse definieren (r3);
- unterstützt zusätzlich die MCS/251-Familie von Intel (r3);
- bei eingeschalteter Querverweisliste wird bei doppelt definierten Symbolen die Stelle der ersten Definition angezeigt (r3);
- unterstützt zusätzlich die TMS320C5x-Familie von Texas Instruments (Implementierung von Thomas Sailer, ETH Zürich, r3);
- die OS/2-Version sollte jetzt auch mit langen Dateinamen klarkommen. Wenn man nicht jeden Mist selber kontrolliert... (r3)
- über den Befehl `BIGENDIAN` kann im MCS-51/251-Modus jetzt gewählt werden, ob die Ablage von Konstanten im Big- oder Little-Endian-Format erfolgen soll (r3);
- es wird beim 680x0 jetzt zwischen dem vollen und eingeschränkten MMU-Befehlssatz unterschieden; eine manuelle Umschaltung ist mit dem `FULLPMMU`-Befehl möglich (r3);
- über die neue Kommandozeilenoption `I` kann eine Liste aller eingezogenen Include-Files mit ihrer Verschachtelung ausgegeben werden (r3);
- Beim `END`-Statement kann jetzt zusätzlich ein Einsprungpunkt für das Programm angegeben werden (r3).
- unterstützt zusätzlich die 68HC16-Familie von Motorola (r3);
- P2HEX und P2BIN erlauben es jetzt, den Inhalt einer Code-Datei adreßmäßig zu verschieben (r4);
- einem `SHARED`-Befehl anhängende Kommentare werden jetzt in die Share-Datei mit übertragen (r4);
- unterstützt zusätzlich die 68HC12-Familie von Motorola (r4);

- unterstützt zusätzlich die XA-Familie von Philips (r4);
- unterstützt zusätzlich die 68HC08-Familie von Motorola (r4);
- unterstützt zusätzlich die AVR-Familie von Atmel (r4);
- aus Kompatibilität zum AS11 von Motorola existieren zusätzlich die Befehle `FCB`, `FDB`, `FCC` und `RMB` (r5);
- unterstützt zusätzlich den M16C von Mitsubishi (r5);
- unterstützt zusätzlich den COP8 von National Semiconductor (r5);
- zwei neue Befehle zur bedingten Assemblierung: `IFB` und `IFNB` (r5);
- Mit dem `EXITM`-Befehl ist es nun möglich, eine Makroexpansion vorzeitig abzubrechen (r5);
- unterstützt zusätzlich den MSP430 von Texas Instruments (r5);
- `LISTING` kennt zusätzlich die Varianten `NOSKIPPED` und `PURECODE`, um nicht assemblierten Code aus dem Listing auszublenden (r5);
- unterstützt zusätzlich die 78K0-Familie von NEC (r5);
- BIGENDIAN ist jetzt auch im PowerPC-Modus verfügbar (r5);
- zusätzlich ein `BINCLUDE`-Befehl, um Binärdaten einbinden zu können (r5);
- zusätzliche TOLOWER- und LOWSTRING-Funktionen, um Groß- in Kleinbuchstaben umzuwandeln (r5);
- es ist jetzt möglich, auch in anderen Segmenten als CODE Daten abzulegen. Das Dateiformat wurde entsprechend erweitert (r5);
- der `DS`-Befehl, mit dem man Speicherbereiche reservieren kann, ist jetzt auch im Intel-Modus zulässig (r5);
- Mit der Kommandozeilenoption `U` ist es jetzt möglich, AS in einen casesensitiven Modus umzuschalten, in dem Namen von Symbolen, selbst definierten Funktionen, Makros, Makroparametern sowie Sektionen nach Groß- und Kleinschreibung unterschieden werden (r5);
- `SFRB` berücksichtigt jetzt auch die Bildungsregeln für Bitadressen im RAM-Bereich; werden nicht bitadressierbare Speicherstellen angesprochen, erfolgt eine Warnung (r5);
- zusätzliche Pseudobefehle `PUSHV` und `POPV`, um Symbolwerte temporär zu sichern (r5);
- zusätzliche Funktionen `BITCNT, FIRSTBIT, LASTBIT` und `BITPOS` zur Bitverarbeitung (r5);
- bei den CPU32-Prozessoren ist jetzt auch der 68360 berücksichtigt (r5);

- unterstützt zusätzlich die ST9-Familie von SGS-Thomson (r6);
- unterstützt zusätzlich den SC/MP von National Semiconductor (r6);
- unterstützt zusätzlich die TMS70Cxx-Familie von Texas Instruments (r6);
- unterstützt zusätzlich die TMS9900-Familie von Texas Instruments (r6);
- unterstützt zusätzlich die Befehlssatzerweiterungen des 80296 (r6);
- die unterstützten Z8-Derivate wurden erweitert (r6);
- berücksichtigt zusätzlich die Maskenfehler des 80C504 von Siemens (r6);
- zusätzliche Registerdefinitionsdatei für die C50x-Prozessoren von Siemens (r6);
- unterstützt zusätzlich die ST7-Familie von SGS-Thomson (r6);
- die Intel-Pseudobefehle zur Datenablage sind jetzt auch für 65816 bzw. MELPS-7700 zulässig (r6);
- für 65816/MELPS-7700 kann die Adreßlänge jetzt durch Präfixe explizit festgelegt werden (r6);
- unterstützt zusätzlich die 8X30x-Familie von Signetics (r6);
- `PADDING` ist nur noch für die 680x0-Familie defaultmäßig eingeschaltet (r7);
- über das neu eingeführte, vordefinierte Symbol `ARCHITECTURE` kann ausgelesen werden, für welche Plattform AS übersetzt wurde (r7);
- Zusätzliche Anweisungen `STRUCT` und `ENDSTRUCT` zur Definition von Datenstrukturen (r7);
- Hex- und Objekt-Dateien für die AVR-Tools können jetzt direkt erzeugt werden (r7);
- `MOVEC` kennt jetzt auch die 68040-Steuerregister (r7);
- zusätzliche `STRLEN`-Funktion, um die Länge eines Strings zu ermitteln (r7);
- Möglichkeit zur Definition von Registersymbolen (r7, momentan nur Atmel AVR);
- kennt zusätzlich die undokumentierten 6502-Befehle (r7);
- P2HEX und P2BIN können jetzt optional die Eingabedateien automatisch löschen (r7);
- P2BIN kann der Ergebnisdatei optional zusätzlich die Startadresse voranstellen (r7);

- unterstützt zusätzlich die ColdFire-Familie von Motorola als Variation des 680x0-Kerns (r7);
- `BYT/FCB, ADR/FDB` und `FCC` erlauben jetzt auch den von `DC` her bekannten Wiederholungsfaktor (r7);
- unterstützt zusätzlich den M*Core von Motorola (r7);
- der SH7000-Teil kennt jetzt auch die SH7700-Befehlserweiterungen (r7);
- der 680x0-Teil kennt jetzt auch die zusätzlichen Befehle des 68040 (r7);
- der 56K-Teil kennt jetzt auch die Befehlserweiterungen bis zum 56300 (r7).
- Mit der neuen `CODEPAGE`-Anweisung können jetzt auch mehrere Zeichentabellen gleichzeitig verwaltet werden (r8);
- Die Argumentvarianten für `CHARSET` wurden erweitert (r8);
- Neue String-Funktionen `SUBSTR` und `STRSTR` (r8);
- zusätzliches `IRPC`-Statement im Makroprozessor (r8);
- zusätzlicher `RADIX`-Befehl, um das Default-Zahlensystem für Integer-Konstanten festzulegen (r8);
- statt `ELSEIF` darf auch einfach `ELSE` geschrieben werden (r8);
- statt = darf als Gleichheitsoperator auch == geschrieben werden (r8);
- `BRANCHEXT` erlaubt es beim Philips XA jetzt, die Sprungweite von kurzen Sprüngen automatisch zu erweitern (r8);
- Debug-Ausgaben sind jetzt auch im NoICE-Format möglich (r8);
- unterstützt zusätzlich die i960-Familie von Intel (r8);
- unterstützt zusätzlich die μPD7720/7725-Signalprozssoren von NEC (r8);
- unterstützt zusätzlich den μPD77230-Signalprozssor von NEC (r8);
- unterstützt zusätzlich die SYM53C8xx-SCSI-Prozessoren von Symbios Logic (r8);
- unterstützt zusätzlich den 4004 von Intel (r8);
- unterstützt zusätzlich die SC14xxx-Serie von National (r8);
- unterstützt zusätzlich die Befehlserweiterungen des PPC403GC (r8);
- zusätzliche Kommandozeilenoption `cpu`, um den Zielprozessor-Default zu setzen (r8);
- Key-Files können jetzt auch von der Kommandozeile aus referenziert werden (r8);

  - zusätzliche Kommandozeilenoption `shareout`, um die Ausgabedatei für SHARED-Definitionen zu setzen (r8);
  - neuer Pseudobefehl `WRAPMODE`, um AVR-Prozessoren mit verkürztem Programmzähler zu unterstützen (r8);
  - unterstützt zusätzlich die C20x-Befehlsuntermenge im C5x-Teil (r8);
  - hexadezimale Adreßangaben der Hilfsprogamme können jetzt auch in C-Notation gemacht werden (r8);
  - Das Zahlensystem für Integerergebnisse in `\{...}`- Ausdrücken ist jetzt per `OUTRADIX` setzbar (r8);
  - Die Registersyntax für 4004-Registerpaare wurde korrigiert (r8);
  - unterstützt zusätzlich die F^2MC8L-Familie von Fujitsu (r8);
  - für P2HEX kann jetzt die Minimallänge für S-Record-Adressen angegeben werden (r8);
  - unterstützt zusätzlich die ACE-Familie von Fairchild (r8);
  - `REG` ist jetzt auch für PowerPCs erlaubt (r8);
  - zusätzlicher Schalter in P2HEX, um alle Adressen zu verschieben (r8);
  - Mit dem Schalter `x` kann man jetzt zusätzlich in einer zweiten Stufe bie betroffene Quellzeile ausgeben (r8).

- Version 1.42:
  - Die Default-Zahlensyntax für Atmel AVR ist jetzt C-Syntax;
  - zusätzliche Kommandozeilenoption `olist`, um die Listing-Ausgabedatei zu setzen;
  - unterstützt zusätzlich die F^2MC16L-Familie von Fujitsu;
  - zusätzlicher Befehl `PACKING` für die AVR-Familie;
  - zusätzliche implizite Makroparameter `ALLARGS` und `ARGCOUNT`;
  - zusätzlicher Befehl `SHIFT` zum Abarbeiten variabler Argumentlisten von Makros;
  - unterstützt temporäre Symbole;
  - zusätzlicher Befehl `MAXNEST` zum Einstellen der maximalen Verschachtelungstiefe von Makroexpansionen;
  - zusätzliche Kommandozeilenoption `noicemask`, um die Menge der in einem NoICE-Debuginfofile gelisteten Segmente zu steuern;
  - unterstützt zusätzlich die 180x-Familie von Intersil;

- unterstützt zusätzlich das address windowing des 68HC11K4;
- P2HEX kann jetzt die Adreßfeldlänge von AVR-Hex-Dateien variieren;
- mit der neuen Kommandozeilenoption `-gnuerrors` können Fehlermeldungen in einem GNU-C-artigen Format ausgegeben werden;
- unterstützt zusätzlich die TMS320C54x-Familie von Texas Instruments;
- Neue Makro-Option `INTLABEL`;
- die neuen Instruktionen und Register der MEGA-AVRs 8/16 wurden hinzugefügt;
- `ENDIF/ENDCASE` zeigen im Listing die Zeilennummer des zugehörigen öffnenden Befehls an;
- der 8051-Teil unterstützt jetzt auch den erweiterten Adreßraum des Dallas DS80C390;
- namenlose temporäre Symbole hinzugefügt;
- unterstützt zusätzlich die undokumentierten 8085-Befehle;
- verbesserte Behandlung von Strukturen;
- Funktion EXPRTYPE() hinzugefügt;
- Zeilenfortsetzungszeichen zulassen;
- Unterstützung für KCPSM/PicoBlaze von Andreass Wassatsch integriert;
- unterstützt zusätzlich die 807x-Familie von National Semiconductor;
- unterstützt zusätzlich den 4040 von Intel;
- unterstützt zusätzlich den eZ8 von Zilog;
- unterstützt zusätzlich die 78K2-Familie von NEC;
- unterstützt zusätzlich die KCPSM3-Variante von Xilinx;
- unterstützt zusätzlich den LatticeMico8;
- unterstützt zusätzlich die 12X-Befehlserweiterungen und den XGATE-Kern der 68HC12-Familie;
- unterstützt zusätzlich den Signetics 2650;
- unterstützt zusätzlich die COP4-Familie von National Semiconductor;
- unterstützt zusätzlich die HCS08-Erweiterungen von Freescale;
- unterstützt zusätzlich die RS08-Familie von Freescale;
- unterstützt zusätzlich den 8008 von Intel;
- weitere Syntax für Integer-Konstanten;

- Funktion `CHARFROMSTR` hinzugefügt;
- Q für Oktalkonstanten im Intel-Modus hinzugefügt;
- weitere Variante für temporäre Symbole hinzugefügt;
- der PowerPC-Teil wurde um Unterstützung für den MPC821 erweitert (Beitrag von Marcin Cieslak);
- implizite Makro-Parameter sind immer case-insensitiv;
- das `REG`-Statement ist jetzt auch für den MSP430 erlaubt;
- unterstützt zusätzlich den XS1 von XMOS;
- zusätzliche Parameter `GLOBALSYMBOLS` und `NOGLOBALSYMBOLS` um zu steuern, ob Labels in Makros lokal sind oder nicht;
- kennt zusätzlich die 75xx-Reihe von NEC;
- kennt zusätzlich die TMS1000-Controller von TI;
- unterstützt zusätzlich die 78K2-Familie von NEC;
- alle neueren Anderungen werden nur noch in der separaten changelog-Datei dokumentiert.

Anhang J

Hinweise zum Quellcode von AS

Wie in der Einleitung erwähnt, gebe ich nach Rücksprache den Quellcode von AS heraus. Im folgenden sollen einige Hinweise zu dessen Handhabung gegeben werden.

J.1 Verwendete Sprache

Ursprünglich war AS ein in Turbo-Pascal geschriebenes Programm. Für diese Entscheidung gab es Ende der 80er Jahre eine Reihe von Gründen: Zum einen war ich damit wesentlich vertrauter als mit jedem C-Compiler, zum anderen waren alle C-Compiler unter DOS verglichen mit der IDE von Turbo-Pascal ziemliche Schnecken. Anfang 1997 zeichnete sich jedoch ab, daß sich das Blatt gewendet hatte: Zum einen hatte Borland beschlossen, die DOS-Entwickler im Stich zu lassen (nochmals ausdrücklich keinen schönen Dank, Ihr Pappnasen von Borland!), und Version 7.0 etwas namens 'Delphi' nachfolgen ließen, was zwar wohl wunderbar für Windows-Programme geeignet ist, die zu 90% aus Oberfläche und zufällig auch ein bißchen Funktion bestehen, für kommandozeilenorientierte Progamme wie AS aber reichlich unbrauchbar ist. Zum anderen hatte sich bereits vor diesem Zeitpunkt mein betriebssystemmäßiger Schwerpunkt deutlich in Richtung Unix verschoben, und auf ein Borland-Pascal für Linux hätte ich wohl beliebig lange warten können (an alle die, die jetzt sagen, Borland würde ja an soetwas neuerdings basteln: Leute, das ist *Vapourware*, und glaubt den Firmen nichts, solange Ihr nicht wirklich in den Laden gehen und es kaufen könnt!). Von daher war also klar, daß der Weg in Richtung C gehen mußte.

Nach der Erfahrung, wohin die Verwendung von Inselsystemen führt, habe ich bei der Umsetzung auf C Wert auf eine möglichst große Portabilität gelegt; da AS jedoch

z.B. Binärdateien in einem bestimmten Format erzeugen muß und an einigen Stellen betriebssystemspezifische Funktionen nutzt, gibt es einige Stellen, an denen man anpassen muß, wenn man AS zum ersten Mal auf einer neuen Plattform übersetzt.

AS ist auf einen C-Compiler ausgelegt, der dem ANSI-Standard entspricht; C++ ist ausdrücklich nicht erforderlich. Wenn Sie nur einen Compiler nach dem veralteten Kernighan&Ritchie-Standard besitzen, sollten Sie sich nach einem neuen Compiler umsehen; der ANSI-Standard ist seit 1989 verabschiedet und für jede aktuelle Plattform sollte ein ANSI-Compiler verfügbar sein, zur Not, indem man mit dem alten Compiler GNU-C baut. Im Quellcode sind zwar einige Schalter vorhanden, um den Code K&R-näher zu machen, aber dies ist ein nicht offiziell unterstütztes Feature, das ich nur intern für ein ziemlich antikes Unix benutze. Alles weitere zum 'Thema K&R' steht in der Datei `README.KR`.

Der Sourcenbaum ist durch einige in der Pascal-Version nicht vorhandene Features (z.B. dynamisch ladbare Nachrichtendateien, Testsuite, automatische Generierung der Dokumentation aus *einem* Quellformat) deutlich komplizierter geworden. Ich werde versuchen, die Sache Schritt für Schritt aufzudröseln:

J.2 Abfangen von Systemabhängigkeiten

Wie ich schon andeutete, ist AS (glaube ich jedenfalls...) auf Plattformunabhängigkeit und leichte Portierbarkeit getrimmt. Dies bedeutet, daß man die Plattformunabhängigkeiten in möglichst wenige Dateien zusammenzieht. Auf diese Dateien werde ich im folgenden eingehen, und dieser Abschnitt steht ganz vorne, weil es sicher eines der wichtigsten ist:

Die Generierung aller Komponenten von AS erfolgt über ein zentrales `Makefile`. Damit dies funktioniert, muß man ihm ein passendes `Makefile.def` anbieten, das die plattformabhängigen Einstellungen wie z.B. Compilerflags vorgibt. Im Unterverzeichnis `Makefile.def-samples` finden sich eine Reihe von Includes, die für gängige Plattformen funktionieren (aber nicht zwangsweise optimal sein müssen...). Wenn die von Ihnen benutzte Plattform nicht dabei ist, können Sie die Beispieldatei `Makefile.def.tmpl` als Ausgangspunkt verwenden (und das Ergebnis mir zukommen lassen!).

Ein weiterer Anlaufpunkt zum Abfangen von Systemabhängigkeiten ist die Datei `sysdefs.h`. Praktisch alle Compiler definieren eine Reihe von Präprozessorsymbolen vor, die den benutzten Zielprozessor sowie das benutzte Betriebsystem beschreiben. Auf einer Sun Sparc unter Solaris mit den GNU-Compiler sind dies z.B. die Symbole `__sparc` und `__SVR4`. In `sysdefs.h` werden diese Symbole genutzt, um für die

restlichen, systemunabhängigen Dateien eine einheitliche Umgebung bereitzustellen. Insbesondere betrifft dies Integer-Datentypen einer bekannten Länge, es kann aber auch die Nach- oder Redefinition von C-Funktionen betreffen, die auf einer bestimmten Plattform nicht oder nicht standardgemäß vorhanden sind. Was da so an Sachen anfällt, liest man am besten selber nach. Generell sind die `#ifdef`-Statements in zwei Ebenen gegliedert: Zuerst wird eine bestimmte Prozessorplattform ausgewählt, dann werden in diesem Abschnitt die Betriebssysteme auseinandersortiert.

Wenn Sie AS auf eine neue Plattform portieren, müssen Sie zwei für diese Plattform typische Symbole finden und `sysdefs.h` passend erweitern (und wieder bin ich an dem Ergebnis interessiert...).

J.3 Systemunabhängige Dateien

...stellen den gößten Teil aller Module dar. Alle Funktionen im Detail zu beschreiben, würde den Rahmen dieser Beschreibung sprengen (wer hier mehr wissen will, steigt am besten selbst in das Studium der Quellen ein, so katastrophal ist mein Programmierstil nun auch wieder nicht...), deshalb hier nur eine kurze Auflistung, welche Module vorhanden sind und was für Funktionen sie beinhalten:

J.3.1 Von AS genutzte Module

as.c

Diese Datei ist die Wurzel von AS: Sie enthält die *main()*-Funktion von AS, die Verarbeitung aller Kommandozeilenoptionen, die übergeordnete Steuerung aller Durchläufe durch die Quelldateien sowie Teile des Makroprozessors.

asmallg.c

In diesem Modul werden all die Befehle bearbeitet, die für alle Prozessoren definiert sind, z.B. `EQU` und `ORG`. Hier findet sich auch der `CPU`-Befehl, mit dem zwischen den einzelnen Prozessoren hin- und hergeschaltet wird.

asmcode.c

In diesem Modul befindet sich die Verwaltung der Code-Ausgabedatei. Exportiert wird ein Interface, mit dem sich eine Code-Datei öffnen und schließen läßt, und das Routinen zum Einschreiben (und Zurücknehmen) von Code anbietet. Eine wichtige Aufgabe dieses Moduls ist die Pufferung des Schreibvorgangs, die die Ausgabegeschwindigkeit erhöht, indem der erzeugte Code in größeren Blöcken geschrieben wird.

asmdebug.c

Optional kann AS Debug-Informationen für andere Tools wie Simulatoren oder Debugger erzeugen, die einen Rückbezug auf den Quellcode erlauben, in diesem Modul gesammelt und nach Ende der Assemblierung in einem von mehreren Formaten ausgegeben werden können.

asmdef.c

Dieses Modul enthält lediglich Deklarationen von überall benötigten Konstanten und gemeinsam benutzten Variablen.

asmfnums.c

Intern vergibt AS für jede benutzte Quelldatei eine fortlaufende Nummer, die zur schnellen Referenzierung benutzt wird. Die Vergabe dieser Nummern und die Umwandlung zwischen Nummer und Dateinamen passiert hier.

asmif.c

Hier befinden sich alle Routinen, die die bedingte Assemblierung steuern. Exportiert wird als wichtigste Variable das Flag `IfAsm`, welches anzeigt, ob Codeerzeugung momentan ein- oder ausgeschaltet ist.

asminclist.c

In diesem Modul ist die Listenstruktur definiert, über die AS die Verschachtelung von Include-Dateien im Listing ausgeben kann.

asmitree.c

Wenn man in einer Code-Zeile das benutzende Mnemonic ermitteln will, ist das einfache Durchvergleichen mit allen vorhandenen Befehlen (wie es noch in vielen Codegeneratoren aus Einfachheit und Faulheit passiert) nicht unbedingt die effizienteste Variante. In diesem Modul sind zwei verbesserte Strukturen (Binärbaum und Hash-Tabelle) definiert, die eine effizientere Suche ermöglichen und die einfache lineare Suche nach und nach ablösen sollen...Priorität nach Bedarf...

asmmac.c

In diesem Modul finden sich die Routinen zur Speicherung und Abfrage von Makros. Der eigentliche Makroprozessor befindet sich (wie bereits erwähnt) in `as.c`.

asmpars.c

Hier geht es ins Eingemachte: In diesem Modul werden die Symboltabellen (global und lokal) in zwei Binärbäumen verwaltet. Außerdem findet sich hier eine ziemlich große Prozedur `EvalExpression`, welche einen (Formel-)ausdruck analysiert und auswertet. Die Prozedur liefert das Ergebnis (Integer, Gleitkomma oder String) in einem varianten Record zurück. Zur Auswertung von Ausdrücken bei der Codeerzeugung sollten allerdings eher die Funktionen `EvalIntExpression`, `EvalFloatExpression` und `EvalStringExpression` verwendet werden. Änderungen zum Einfügen neuer Prozessoren sind hier nicht erforderlich und sollten auch nur mit äußerster Überlegung erfolgen, da man hier sozusagen an ,,die Wurzel" von AS greift.

asmsub.c

Hier finden sich gesammelt einige häufig gebrauchte Unterroutinen, welche in erster Linie die Bereiche Fehlerbehandlung und 'gehobene' Stringverarbeitung abdecken.

bpemu.c

Wie am Anfang erwähnt, war AS ursprünglich ein in Borland-Pascal geschriebenes Programm. Bei einigen intrinsischen Funktionen des Compilers war es einfacher, diese zu emulieren, anstatt alle betroffenen Stelle im Quellcode zu ändern. Na ja...

chunks.c

Dieses Modul definiert einen Datentyp, mit dem eine Liste von Adreßbereichen verwaltet werden kann. Diese Funktion wird von AS für die Belegungslisten benötigt, außerdem benutzten P2BIN und P2HEX diese Listen, um vor Überlappungen zu warnen.

cmdarg.c

Dieses Modul implementiert den Mechanismus der Kommdozeilenparameter. Es benötigt eine Spezifikation der erlaubten Parameter, zerlegt die Kommadozeile und ruft die entsprechenden Callbacks auf. Der Mechanismus leistet im einzelnen folgendes:

- Mitbearbeitung von Optionen in einer Environment-Variablen oder entsprechenden Datei;
- Rückgabe einer Menge, welche die noch nicht bearbeiteten Kommandozeilenparameter beschreibt;
- Trenunng von positiven und negativen Schaltern;
- Eine Hintertür, falls die darüber liegende Entwicklungsumgebung die Kommandozeile nur in Groß- oder Kleinschreibung übergibt.

Dieses Modul wird nicht nur von AS, sondern auch von den Hilfsprogrammen `BIND, P2HEX und P2BIN` verwendet.

codepseudo.c

Hier finden sich Pseudobefehle, die von mehreren Codegeneratoren verwendet werden. Dies ist einmal die Intel-Gruppe mit der `DB..DO`-Gruppe, zum anderen die Pendants für die 8/16-Bitter von Motorola oder Rockwell. Wer in diesem Bereich um einen Prozessor erweitern will, kann mit einem Aufruf den größten Teil der Pseudobefehle erschlagen.

codevars.c

Aus Speicherersparnisgründen sind hier einige von diversen Codegeneratoren benutzen Variablen gesammelt.

endian.c

Doch noch ein bißchen Maschinenabhängigkeit, jedoch ein Teil, um den man sich nicht zu kümmern braucht: Ob eine Maschine Little- oder Big-Endianess benutzt, wird in diesem Modul beim Programmstart automatisch bestimmt. Weiterhin wird geprüft, ob die in `sysdefs.h` gemachten Typfestlegungen für Integervariablen auch wirklich die korrekten Längen ergeben.

headids.c

Gesammelt sind hier alle von AS unterstützten Zielprozessorfamilien, die dafür in Code-Dateien verwendeten Kennzahlen (siehe Kapitel 5.1) sowie das von P2HEX defaultmäßig zu verwendende Ausgabeformat. Ziel dieser Tabelle ist es, Das Hinzufügen eines neuen Prozessors möglichst zu zentralisieren, d.h. es sind im Gegensatz zu früher keine weiteren Modifikationen an den Quellen der Hilfsprogramme mehr erforderlich.

ioerrs.c

Hier ist die Umwandlung von Fehlernummern in Klartextmeldungen abgelegt. Hoffentlich treffe ich nie auf ein System, auf dem die Nummern nicht als Makros definiert sind, dann kann ich nämlich dieses Modul komplett umschreiben...

nlmessages.c

Die C-Version von AS liest alle Meldungen zur Laufzeit aus Dateien, nachdem die zu benutzende Sprache ermittelt wurde. Das Format der Nachrichtendateien ist kein einfaches, sondern ein spezielles, kompaktes, vorindiziertes Format, das zur Übersetzungszeit von einem Programm namens 'rescomp' (dazu kommen wir noch) erzeugt wird. Dieses Modul ist das Gegenstück zu rescomp, die den korrekten Sprachenanteil einer Datei in ein Zeichenfeld einliest und Zugriffsfunktionen anbietet.

nls.c

In diesem Modul wird ermittelt, welche nationalen Einstellungen (Datums- und Zeitformat, Ländercode) zur Laufzeit vorliegen. Das ist leider eine hochgradig systemspezifische Sache, und momentan sind nur drei Methoden definiert: Die von MS-DOS, die von OS/2 und die typische Unix-Methode über die *locale*-Funktionen. Für alle anderen Systeme ist leider `NO_NLS` angesagt...

stdhandl.c

Zum einen ist hier eine spezielle open-Funktion gelandet, die die Sonderstrings `!0...!2` als Dateinamen kennt und dafür Duplikate der Standard-Dateihandles *stdin, stdout* und *stderr* erzeugt, zum anderen wird hier festgestellt, ob die Standardausgabe auf ein Gerät oder eine Datei umgeleitet wurde. Das bedingt auf nicht-Unix-Systemen leider auch einige Speziallösungen.

stringlists.c

Dies ist nur ein kleiner „Hack", der Routinen zur Verwaltung von linearen Listen mit Strings als Inhalt definiert, welche z.B. im Makroprozessor von AS gebraucht werden.

strutil.c

Hier sind einige häufig genutzte String-Operationen gelandet.

version.c

Die momentan gültige Version ist für AS und alle anderen Hilfsprogramme hier zentral gespeichert.

code????.c

Dies Module bilden den Hauptteil der AS-Quellen: jedes Modul beinhaltet den Codegenerator für eine bestimmte Prozessorfamilie.

J.3.2 Zusätzliche Module für die Hilfsprogramme

hex.c

Ein kleines Modul zur Umwandlung von Integerzahlen in Hexadezimaldarstellung. In C nicht mehr unbedingt erforderlich (außer zur Wandlung von *long long*-Variablen, was leider nicht alle `printf()`'s unterstützen), aber es ist im Rahmen der Portierung eben auch stehengeblieben.

p2bin.c

Die Quellen von P2BIN.

p2hex.c

Die Quellen von P2HEX.

pbind.c

Die Quellen von BIND.

plist.c

Die Quellen von PLIST.

toolutils.c

Hier sind gesammelt die Unterroutinen, die von allen Hilfsprogrammen benötigt werden, z.B. für das Lesen von Code-Dateien.

J.4 Während der Erzeugung von AS gebrauchte Module

a2k.c

Dies ist ein Minimalfilter, das ANSI-C-Files in Kernighan-Ritchie umwandelt. Um es genau zu sagen: es werden nur die Funktionsköpfe umgewandelt, und auch nur dann, wenn sie ungefähr so formatiert sind, wie es mein Schreibstil eben ist. Es komme also keiner auf die Idee, das wäre ein universeller C-Parser!

addcr.c

Ein kleiner Filter, der bei der Installation auf DOS- oder OS/2-Systemen gebraucht wird. Da DOS und OS/2 den Zeilenvorschub mit CR/LF vornehmen, Unix-Systeme jedoch nur mit LF, werden sämtliche mitgelieferten Assembler-Includes bei der Installation durch diesen Filter geschickt.

bincmp.c

Für DOS und OS/2 übernimmt dieses Modul die Funktion die Funktion des *cmp*-Befehls, d.h. den binären Vergleich von Dateien während des Testlaufes. Während dies prinzipiell auch mit dem mitgelieferten *comp* möglich wäre, hat *bincmp* keine lästigen interaktiven Abfragen (bei denen man erst einmal herausfinden muß, wie man sie auf allen Betriebssystemversionen abstellt...)

findhyphen.c

Dies ist das Untermodul in *tex2doc*, daß für die Silbentrennung von Worten sorgt. Der verwendete Algorithmus is schamlos von TeX abgekupfert.

grhyph.c

Die Definition der Silbentrennungsregeln für die deutsche Sprache.

rescomp.c

Dies ist der 'Resourcencompiler' von AS, d.h. das Werkzeug, das die lesbaren Dateien mit Stringresourcen in ein schnelles, indiziertes Format umsetzt.

tex2doc.c

Ein Werkzeug, daß die LaTeX-Dokumentation von AS in ein ASCII-Format umsetzt.

tex2html.c

Ein Werkzeug, daß die LaTeX-Dokumentation von AS in ein HTML-Dokument umsetzt.

umlaut.c und unumlaut.c

Diese Prograämmchen besorgen die Wandlung zwischen Sonderzeichenkodierung im ISO-Format (alle AS-Dateien verwenden im Auslieferungszustand die ISO8859-1-Kodierung für Sonderzeichen) und Sonderzeichenkodierung im systemspezifischen Format. Neben einer Plain-ASCII7-Variante sind dies im Augenblick die IBM-Zeichensätze 437 und 850.

ushyph.c

Die Definition der Silbentrennungsregeln für die englische Sprache.

J.5 Generierung der Nachrichtendateien

Wie bereits erwähnt, verwendet der C-Quellenbaum von AS ein dynamisches Ladeverfahren für alle (Fehler-)Meldungen. Gegenüber den Pascal-Quellen, in denen alle Meldungen in einem Include-File gebündelt waren und so in die Programme hineinübersetzt wurden, macht es dieses Verfahren überflüssig, mehrere sprachliche Varianten von AS zur Verfügung zu stellen: es gibt nur noch eine Version, die beim Programmstart die zu benutzende Variante ermittelt und aus den Nachrichtendateien die entsprechende Komponente lädt. Kurz zur Erinnerung: Unter DOS und OS/2 wird dazu die gewählte `COUNTRY`-Einstellung zu Rate gezogen, unter Unix werden die Environment-Variablen `LC_MESSAGES, LC_ALL` und `LANG` befragt.

J.5.1 Format der Quelldateien

Eine Quelldatei für den Message-Compiler *rescomp* hat üblicherweise die Endung `.res`. Der Message-Compiler erzeugt aus dieser Datei ein oder zwei Dateien:

- eine binäre Datei, die zur Laufzeit von AS bzw. den Hilfsprogrammen gelesen wird;
- optional eine weitere C-Header-Datei, die allen vorhandenen Nachrichten eine Indexnummer zuweist. Über diese Indexnummern und eine Indextabelle in der binären Datei kann zur Laufzeit schnell auf einzelne Meldungen zugegriffen werden.

Die Quelldatei für den Message-Compiler ist eine reine ASCII-Datei, also mit jedem beliebigen Editor bearbeitbar, und besteht aus einer Reihe von Steueranweisungen mit Parametern. Leerzeilen sowie Zeilen, die mit einem Semikolon beginnen, werden ignoriert. Das Inkludieren anderer Dateien ist über das `Include`-Statement möglich:

```
Include <Datei>
```

Am Anfang jeder Quelldatei müssen zwei Statements stehen, die die im folgenden definierten Sprachen beschreiben. Das wichtigere der beiden Statements ist `Langs`, z.B.:

```
Langs DE(049) EN(001,061)
```

beschreibt, daß zwei Sprachen im folgenden definiert werden. Der erste Nachrichtensatz soll benutzt werden, wenn unter Unix die Sprache per Environment-Variablen auf `DE` gestellt wurde bzw. unter DOS bzw. OS/2 der Landescode 049 eingestellt wurde. Der zweite Satz kommt dementsprechend bei den Einstellungen `EN` bzw. 061 oder 001 zum Einsatz. Während bei den 'Telefonnummern' mehrere Codes auf einen Nachrichtensatz verweisen können, ist die Zuordnung zu den Unix-Landescodes eineindeutig. Dies ist in der Praxis aber kein Beinbruch, weil die `LANG`-Variablen unter Unix Unterversionen einer Sprache als Anhängsel beschreiben, z.B. so:

```
de.de
de.ch
en.us
```

AS vergleicht nur den Anfang der Strings und kommt so trotzdem zur richtigen Entscheidung. Das `Default`-Statement gibt vor, welcher Sprachensatz verwendet werden soll, wenn entweder überhaupt keine Sprache gesetzt wurde oder eine Kennung verwendet wird, die nicht in der Liste von `Langs` vorhanden ist. Typischerweise ist dies Englisch:

```
Default EN
```

Nach diesen beiden Definitionen folgt eine beliebige Menge von `Message`-Statements, d.h. Definitionen von Meldungen:

```
Message ErrName
 ": Fehler "
 ": error "
```

Wurden n Sprachen im `Langs`-Statement angekündigt, so nimmt der Message-Compiler **genau** die folgenden n Zeilen als die zu speichernden Strings. Es ist also nicht möglich, bei einzelnen Nachrichten bestimmte Sprachen fortzulassen, und eine auf die Strings folgende Leerzeile ist keinesfalls als Endemarkierung für die Liste mißzuverstehen; eingefügte Leerzeilen dienen einzig und allein der besseren Lesbarkeit. Was allerdings erlaubt ist, ist, einzelne Meldungen über mehrere Zeilen in der Quelldatei zu verteilen; alle Zeilen bis auf die letzte müssen dann mit einem Backslash als Fortsetzungszeichen enden:

```
Message TestMessage2
 "Dies ist eine" \
 "zweizeilige Nachricht"
 "This is a" \
 "two-line message"
```

Wie bereits erwähnt, handelt es sich bei den Quelldateien um reine ASCII-Dateien; Sonderzeichen können in den Meldungstexten zwar eingetragen werden (und der Compiler wird sie auch so durchreichen), der gravierende Nachteil ist aber, daß eine solche Datei nicht mehr voll portabel ist: Wird sie auf ein anderes System gebracht, das z.B. eine andere Kodierung für Umlaute verwendet, bekommt der Anwender zur Laufzeit nur merkwürdige Zeichen zu sehen...Sonderzeichern sollten daher immer mit Hilfe von speziellen Sequenzen geschrieben werden, die von HTML bzw. SGML entlehnt wurden (siehe Tabelle J.1). Zeilenvorschübe können in eine Zeile wie von C her gewohnt mit `\n` eingebracht werden.

Sequenz...	ergibt...
`ä ö ü`	ä ö ü (Umlaute)
`Ä Ö Ü`	Ä Ö Ü
`ß`	ß (scharfes s)
`à è ì ò ù`	á é í ó ú (Accent
`À È Ì Ò Ù`	Á É Í Ó Ú grave)
`á é í ó ú`	à è ì ò ù (Accent
`Á É Í Ó Ú`	À È Ì Ò Ù agiu)
`â ê î ô û`	â ê î ô û (Accent
`Â Ê Î Ô Û`	Â Ê Î Ô Û circonflex)
`ç Ç`	ç Ç(Cedilla)
`ñ Ñ`	ñ Ñ
`å Å`	åÅ
`æ &Aelig;`	æÆ
`¿ ¡`	umgedrehtes ! oder ?

Tabelle J.1: Sonderzeichenschreibweise des *rescomp*

J.6 Dokumentationserzeugung

In einer Quellcodedistribution von AS ist diese Dokumentation nur als LaTeX-Dokument enthalten. Andere Formate werden aus dieser mit Hilfe von mitgelieferten Werkzeugen automatisch erzeugt. Zum einen reduziert dies den Umfang einer Quellcodedistribution, zum anderen müssen Änderungen nicht an allen Formatversionen eines Dokumentes parallel vorgenommen werden, mit all den Gefahren von Inkonsistenzen.

Als Quellformat wurde LaTeX verwendet, weil...weil...weil es eben schon immer vorhanden war. Zudem ist TeX fast beliebig portierbar und paßt damit recht gut

zum Anspruch von AS. Eine Standard-Distribution erlaubt damit eine 'ordentliche' Ausgabe auf so ziemlich jedem Drucker; für eine Konvertierung in die früher immer vorhandene ASCII-Version liegt der Konverter *tex2doc* bei; zusätzlich einen Konverter *tex2html*, so daß man die Anleitung direkt ins Internet stellen kann.

Die Erzeugung der Dokumentation wird mit einem schlichten

```
make docs
```

angestoßen; daraufhin werden die beiden erwähnten Hilfstools erzeugt, auf die TeX-Dokumentation angewandt und schlußendlich wird noch LaTeX selber aufgerufen. Dies natürlich für alle Sprachen nacheinander...

J.7 Testsuite

Da AS mit binären Daten von genau vorgegebener Struktur umgeht, ist er naturgemäß etwas empfindlich für System- und Compilerabhängigkeiten. Um wenigstens eine gewisse Sicherheit zu geben, daß alles korrekt durchgelaufen ist, liegt dem Assembler im Unterverzeichnis `tests` eine Menge von Test-Assemblerquellen bei, mit denen man den frisch gebauten Assembler testen kann. Diese Testprogramme sind in erster Linie darauf getrimmt, Fehler in der Umsetzung des Maschinenbefehlssatzes zu finden, die besonders gern bei variierenden Wortlängen auftreten. Maschinenunabhängige Features wie der Makroprozessor oder bedingte Assemblierung werden eher beiläufig getestet, weil ich davon ausgehe, daß sie überall funktionieren, wenn sie bei mir funktionieren...

Der Testlauf wird mit einem einfachen *make test* angestoßen. Jedes Testprogramm wird assembliert, in eine Binärdatei gewandelt und mit einem Referenz-Image verglichen. Ein Test gilt als bestanden, wenn Referenz und die neu erzeugte Datei Bit für Bit identisch sind. Am Ende wird summarisch die Assemblierungszeit für jeden Test ausgegeben (wer will, kann mit diesen Ergebnissen die Datei `BENCHES` ergänzen), zusammen mit dem Erfolg oder Mißerfolg. Jedem Fehler ist auf den Grund zu gehen, selbst wenn er bei einem Zielprozessor auftritt, den Sie nie nutzen werden! Es ist immer möglich, daß dies auf einen Fehler hinweist, der auch bei anderen Zielprozessoren auftritt, nur zufällig nicht in den Testfällen.

J.8 Einhängen eines neuen Zielprozessors

Der mit Abstand häufigste Grund, im Quellcode von AS etwas zu verändern, dürfte wohl die Erweiterung um einen neuen Zielprozessor sein. Neben der Ergänzung der

Makefiles um das neue Modul ist lediglich eine Modifikation der Quellen an wenigen Stellen erforderlich, den Rest erledigt das neue Modul, indem es sich in der Liste der Codegeneratoren registriert. Im folgenden will ich kochbuchartig die zum Einhängen erforderlichen Schritte beschreiben:

Festlegung des Prozessornamens

Der für den Prozessor zu wählende Name muß zwei Kriterien erfüllen:

1. Der Name darf noch nicht von einem anderen Prozessor belegt sein. Beim Aufruf von AS ohne Parameter erhält man eine Liste der bereits vorhandenen Namen.
2. Soll der Prozessorname vollständig in der Variablen `MOMCPU` auftauchen, so darf er außer am Anfang keine Buchstaben außerhalb des Bereiches von A..F enthalten. In der Variablen `MOMCPUNAME` liegt aber zur Assemblierzeit immer der volle Name vor. Sonderzeichen sind generell nicht erlaubt, Kleinbuchstaben werden vom CPU-Befehl bei der Eingabe in Großbuchstaben umgewandelt und sind daher auch nicht im Prozessornamen sinnvoll.

Der erste Schritt der Registrierung ist die Eintragung des Prozessors oder der Prozessorfamilie in der Datei `headids.c`. Wie bereits erwähnt, wird diese Datei von den Hilfsprogrammen mitbenutzt und spezifiziert die einer Prozessorfamilie zugeordnete Kenn-ID in Codedateien sowie das zu verwendende Hex-Format. Bei der Wahl der Kenn-ID würde ich mir etwas Absprache wünschen...

Definition des Codegeneratormoduls

Das Modul, das für den neuen Prozessor zuständig sein soll, sollte einer gewissen Einheitlichkeit wegen den Namen `code....` tragen, wobei `.....` etwas mit dem Prozessornamen zu tun haben sollte. Den Kopf mit den Includes übernimmt man am besten direkt aus einem bereits vorhandenen Codegenerator.

Mit Ausnahme einer Initialisierungsfunktion, die zu Anfang der `main()`-Funktion im Modul `as.c` aufgerufen werden muß, braucht das neue Modul keinerlei Funktionen oder Variablen zu exportieren, da die ganze Kommunikation zur Laufzeit über indirekte Sprünge abgewickelt wird. Die dazu erforderlichen Registrierungen müssen in der Initialisierungsfunktion des Moduls vorgenommen werden, indem für jeden von der Unit zu behandelnden Prozessortyp ein Aufruf der Funktion `AddCPU` erfolgt:

```
CPUxxxx = AddCPU("XXXX", SwitchTo_xxxx);
```

'XXXX' ist dabei der für den Prozessor festgelegte Name, der später im Assemblerprogramm verwendet werden muß, um AS auf diesen Zielprozessor umzuschalten. SwitchTo_xxxx (im folgenden kurz als ,,Umschalter" bezeichnet) ist eine parameterlose Prozedur, die von AS aufgerufen wird, sobald auf diesen Prozessor umgeschaltet werden soll. Als Ergebnis liefert AddCPU eine Zahlenwert, der als interne ,,Kennung" für diesen Prozessor fungiert. In der globalen Variablen MomCPU wird ständig die Kennung des momentan gesetzten Zielprozessors mitgeführt. Der von AddCPU gelieferte Wert sollte in einer privaten Variable des Typs CPUVar (hier CPUxxxx genannt) abgelegt werden. Falls ein Codegeneratormodul verschiedene Prozessoren (z.B. einer Familie) verwaltet, kann es so durch Vergleich von MomCPU gegen diese Werte feststellen, welche Befehlsuntermenge momentan zugelassen ist.

Dem Umschalter obliegt es, AS auf den neuen Zielprozessor ,,umzupolen". Dazu müssen im Umschalter einige globale Variablen besetzt werden:

- ValidSegs : Nicht alle Prozessoren definieren alle von AS unterstützten Adreßräume. Mit dieser Menge legt man fest, welche Untermenge für den jeweiligen Prozessor von SEGMENT-Befehl zugelassen wird. Im mindesten muß das Code-Segment freigeschaltet werden. Die Gesamtmenge aller vorhandenen Segmenttypen kann in der Datei fileformat.h nachgelesen werden (Seg.....-Konstanten).

- SegInits : Dieses Feld speichert die initialen (ohne ORG-Befehl) Startadressen in den einzelnen Segmenten. Nur in Ausnahmefällen (physikalisch überlappende, aber logisch getrennte Adreßräume) sind hier andere Werte als 0 sinnvoll.

- Grans : Hiermit kann für jedes Segment die Größe des kleinsten adressierbaren Elements in Bytes festgelegt werden, d.h. die Größe des Elements, für das eine Adresse um eins erhöht wird. Bei den allermeisten Prozessoren (auch 16 oder 32 Bit) ist dies ein Byte, nur z.B. Signalprozessoren und die PICs fallen aus dem Rahmen.

- ListGrans : Hiermit kann wieder für alle Segmente getrennt festgelegt werden, in was für Gruppen die Bytes im Assembler-Listing dargestellt werden sollen. Beim 68000 sind z.B. Befehle immer ein mehrfaches von 2 Bytes lang, weshalb die entsprechende Variable auf 2 gesetzt ist.

- SegLimits : Dieses Feld legt die höchste Adresse für jedes Segment fest, z.B. 65535 für einen 16-Bit-Adreßraum. Dieses Feld braucht nicht ausgefüllt zu werden, wenn der Codegenerator die ChkPC-Methode selber übernimmt.

- `ConstMode` : Diese Variable kann die Werte `ConstModeIntel`, `ConstModeMoto` oder `ConstModeC` haben und bestimmt, in welcher Form Zahlensysteme bei Integer-Konstanten spezifiziert werden sollen (sofern das Programm nicht vom Relaxed-Modus Gebrauch macht).

- `PCSymbol` : Diese Variable enthält den String, mit dem aus dem Assembler-Programm heraus der momentane Stand des Programmzählers abgefragt werden kann. Für Intel-Prozessoren ist dies z.B. ein Dollarzeichen.

- `TurnWords` : Falls der Prozessor ein Big-Endian-Prozessor sein sollte und eines der Elemente von `ListGrans` ungleich eins ist, sollte dieses Flag auf True gesetzt werden, um korrekte Code-Dateien zu erhalten.

- `SetIsOccupiedFnc` : Einige Prozessoren verwenden `SET` als Maschinenbefehl. Ist dieser Callback gesetzt, so kann der Codegenerator darüber melden, daß `SET` nicht als Pseudo-Befehl interpretiert werden soll. Der Rückgabewert kann konstant `True` sein, die Entscheidung kann aber auch z.B. anhand der Anzahl der Argumente fallen.

- `HeaderID` : Dieses Byte enthält die Kennung, mit der in der Codedatei die Prozessorfamilie gekennzeichnet wird (siehe Abschnitt 5.1). Um Zweideutigkeiten zu vermeiden, bitte ich, den Wert mit mir abzusprechen. Bis auf weiteres sollten keine Werte außerhalb des Bereiches $01..$7f benutzt werden, diese sind für Sonderzwecke (wie z.B. eine zukünftige Erweiterung um einen Linker) reserviert. Auch wenn dieser Wert in den meisten älteren Codegeneratoren hart gesetzt wird, ist es die heute bevorzugte Methode, den Wert aus `headids.h` per `FindFamilyByName` zu holen.

- `NOPCode` : In bestimmten Situationen kann es sein, daß AS unbenutzte Bereiche im Code mit NOPs auffüllen muß. Diese Variable beinhaltet den dazu erforderlichen Code.

- `DivideChars` : Dieser String enthält all jene Zeichen, die als Trennzeichen für die Parameter eines Assemblerbefehls zugelassen sind. Nur für extreme Ausreißer (wie den DSP56) sollte sich in diesem String etwas anderes finden als ein Komma.

- `HasAttrs` : Einige Prozessoren wie die 68k-Reihe teilen einen Maschinenbefehl durch einen Punkt noch weiter in Mnemonic und Attribut auf. Ist dies beim neuen Prozessor auch der Fall, so ist dieses Flag auf True zu setzen. AS liefert dann die Einzelteile in den Variablen `OpPart` und `AttrPart`. Setzt man es dagegen auf False, so bleibt der Befehl in `OpPart` zusammen, und `AttrPart` ist immer leer. Sofern der Prozessor keine Attribute verwendet, sollte man

`HasAttrs` auf jeden Fall auf False setzen, da man sich sonst die Möglichkeit nimmt, Makros mit einem Punkt im Namen (z.B. zur Emulation anderer Assembler) zu definieren.

- `AttrChars` : Falls `HasAttrs` gesetzt wurde, müssen in diesem String alle Zeichen eingetragen werden, die das Attribut vom Befehl trennen können. Meist ist dies nur der Punkt.

Gehen Sie nicht davon aus, daß eine dieser Variablen einen vordefinierten Wert hat, sondern besetzen Sie **ALLE** Felder neu!!

Neben diesen Variablen müssen noch einige Funktionszeiger besetzt wird, mit denen der Codegenerator sich in AS einbindet:

- `MakeCode` : Diese Routine wird nach der Zerlegung einer Zeile in Mnemonic und Parameter aufgerufen. Das Mnemonic liegt in der Variablen `OpPart`, die Parameter in dem Feld `ArgStr`. Die Zahl der Parameter kann aus der Variablen `ArgCnt` ausgelesen werden. Das binäre Ergebnis muß in dem Byte-Feld `BAsmCode` abgelegt werden, dessen Länge in der Variablen `CodeLen`. Falls der Prozessor wortorientiert wie der 68000 oder viele Signalprozessoren ist, kann Feld auch wortweise als `WAsmCode` adressiert werden. Für ganz extreme Fälle gibt es auch noch `DAsmCode`... Die Codelänge wird ebenfalls in solchen Einheiten angegeben.

- `SwitchFrom`: Diese parameterlose Prozedur erlaubt dem Codegeneratormodul, noch „Aufräumarbeiten" durchzuführen, wenn auf einen anderen Zielprozessor umgeschaltet wird. So kann man an dieser Stelle z.B. Speicher freigeben, der im Umschalter belegt wurde und nur benötigt wird, während dieses Codegeneratormodul aktiv ist. Im einfachsten Fall zeigt diese Prozedurvariable auf eine leere Prozedur. Ein Beispiel für die Anwendung dieser Prozedur finden Sie im Modul `CODE370`, das seine Codetabellen dynamisch erzeugt und wieder freigibt.

- `IsDef` : Bestimmte Prozessoren kennen neben `EQU` noch weitere Pseudobefehle, bei denen ein in der ersten Spalte stehender Symbolname kein Label darstellt, z.B. `BIT` beim 8051. Diese Funktion muß TRUE zurückliefern, falls ein solcher, zusätzlicher Befehl vorliegt. Im einfachsten Fall braucht nur FALSE zurück geliefert zu werden.

Optional kann ein Codegenerator auch noch folgende weitere Funktionszeiger besetzen:

- `ChkPC` : Obwohl AS die Programmzähler intern durchgängig mit 32 oder 64 Bit verwaltet, benutzen die meisten Prozessoren nur einen kleineren Adreßraum. Diese Funktion liefert AS Informationen, ob der momentane Programmzähler den erlaubten Bereich überschritten hat. Bei Prozessoren mit mehreren Adreßräumen kann diese Routine natürlich deutlich komplizierter ausfallen. Ein Beispiel dafür findet sich z.B. im Modul `code16c8x.c`. Falls alles in Ordnung ist, muß die Funktion TRUE zurückliefern, ansonsten FALSE. Diese Funktion muß ein Codegenerator nur implementieren, wenn er das Feld `SegLimits` nicht belegt. Das kann z.B. notwendig werden, wenn der gültige Adreßbereich eines Segments nicht zusammenhängend ist.

- `InternSymbol` : Manche Prozessoren, z.B. solche mit einer Registerbank im internen RAM, defineren diese 'Register' als Symbole vor, und es würde wenig Sinn machen, diese in einer separaten Include-Datei mit 256 oder möglicherweise noch mehr `EQU`s zu definieren. Mit dieser Funktion erhält man Zugang zum Formel- Parser von AS: Sie erhält den Ausdruck als ASCII-String, und wenn sie eines der 'eingebauten Symbole' erkennt, besetzt sie die übergebene Struktur des Typs *TempResult* entsprechend. Falls die Überprüfung nicht erfolgreich war, muß deren Element `Typ` auf `TempNone` gesetzt werden. Die Routine sollte im Falle eines Mißerfolges *keine* Fehlermeldungen ausgeben, weil dies immer noch anderweitig gültige Symbole sein können. Seien Sie extrem vorsichtig mit dieser Routine, da sie einen Eingriff in den Parser darstellt!

- `DissectBit` : Falls die Zielplattform Bit-Objekte unterstützt, d.h. Objekte, die sowohl eine Register/Speicheradresse als auch eine Bitposition in einer einzelnen Integer-Zahl gepackt abspeichern, ist dies der Callback, über den solche gepackten Objekte fürs Listing wieder in eine Quellcode-artige Form rückübersetzt werden.

- `DissectReg` : Falls die Zielplattform Registersymbole unterstützt, ist dies der Callback, über den Registernummer und -länge fürs Listing wieder in eine Quellcode-artige Form rückübersetzt werden. Falls Registersymbole unterstützt werden, ist üblicherweise auch der `InternSymbol`-Callback auszufüllen.

- `QualifyQuote` : Über diesen optionalen Callback kann für eine bestimmte Zielplattform von Fall zu Fall festgelegt werden, daß ein einzelnes Hochkomma *keine* Zeichenkette einleitet. Ein Beispiel dafür ist die als `AF'` geschriebene alternative Registerbank beim Z80, oder die Hexadezimal-Syntax `H'...` bei manchen Hitachi-Prozessoren.

Wer will, kann sich übrigens auch mit einem Copyright-Eintrag verewigen, indem

er in der Initialisierung des Moduls (bei den `AddCPU`-Befehlen) einen Aufruf der Prozedur `AddCopyright` einfügt, in der folgenden Art:

```
AddCopyright("Intel 80986-Codegenerator (C) 2010 Hubert Simpel");
```

Der übergebene String wird dann nach dem Programmstart zusätzlich zu der Standardmeldung ausgegeben.

Bei Bedarf kann sich das Modul im Initialisierungsteil noch in die Kette aller Funktionen eintragen, die vor Beginn eines Durchlaufes durch den Quelltext ausgeführt werden. Dies ist z.B. immer dann der Fall, wenn die Code-Erzeugung im Modul abhängig vom Stand bestimmter, durch Pseudobefehle beeinflußbarer Flags ist. Ein häufig auftretender Fall ist z.B., daß ein Prozessor im User- oder Supervisor-Modus arbeiten kann, wobei im User-Modus bestimmte Befehle gesperrt sind. Im Assembler-Quelltext könnte dieses Flag, das angibt, in welchem Modus der folgende Code ausgeführt wird, durch einen Pseudobefehl umgeschaltet werden. Es ist aber dann immer noch eine Initialisierung erforderlich, die sicherstellt, daß in allen Durchläufen ein identischer Ausgangszustand vorliegt. Der über die Funktion `AddInitPassProc` angebotene Haken bietet die Möglichkeit, derartige Initialisierungen vorzunehmen. Die übergebene Callback-Funktion wird vor Beginn eines Durchgangs aufgerufen.

Analog zu `AddInitPassProc` funktioniert die über `AddCleanUpProc` aufgebaute Funktionsliste, die es den Codegeneratoren erlaubt, nach dem Abschluß der Assemblierung noch Aufräumarbeiten (z.B. das Freigeben von Literaltabellen o.ä.) durchzuführen. Dies ist sinnvoll, wenn mehrere Dateien mit einem Aufruf assembliert werden, sonst hätte man noch „Müll" aus einem vorigen Lauf in den Tabellen. Momentan nutzt kein Modul diese Möglichkeit.

Schreiben des Codegenerators selber

Nach diesen Präliminarien ist nun endlich eigene Kreativität gefragt: Wie Sie es schaffen, aus dem Mnemonic und den Argumenten die Code-Bytes zu erzeugen, ist weitgehend Ihnen überlassen. Zur Verfügung stehen dafür natürlich über den Formelparser die Symboltabellen sowie die Routinen aus `asmsub.c` und `asmpars.c`. Ich kann hier nur einige generelle Hinweise geben:

- Versuchen Sie, die Prozessorbefehle in Gruppen aufzusplitten, die gleiche Operanden erwarten und sich nur in einigen Kennbits unterscheiden. Alle argumentlosen Befehle kann man z.B. so in einer Tabelle abhandeln.

- Die meisten Prozessoren haben ein festes Repertoire von Adressierungsarten. Verlagern Sie das Parsing eines Adreßausdrucks in eine getrennte Unterroutine.
- Die Routine `WrError` definiert eine Vielzahl von möglichen Fehlermeldungen und ist bei Bedarf leicht erweiterbar. Nutzen Sie das! Bei allen Fehler nur lapidar einen „Syntaxfehler" zu melden, nützt niemandem!

Mit Sicherheit wird auch das Studium der vorhandenen Module weiterhelfen.

Änderungen für die Dienstprogramme

Eine winzige Änderung ist auch noch an den Quellen der Dienstprogramme nötig, und zwar in der Routine `Granularity()` in `toolutils.c`: Falls eines der Adreßräume dieses Prozessors eine andere Granularität als 1 hat, muß dort die Abfrage passend ergänzt werden, sonst verzählen sich PLIST, P2BIN und P2HEX...

J.9 Lokalisierung auf eine neue Sprache

Sie haben Interesse an diesem Thema? Wunderbar! Das ist eine Sache, die von Programmierern gerne außen vor gelassen wird, insbesondere, wenn sie aus dem Land der unbegrenzten Möglichkeiten kommen...

Die Lokalisierung auf eine neue Sprache gliedert sich in zwei Teile: Die Anpassung der Programmmeldungen sowie die Übersetzung der Anleitung. Letzteres ist sicherlich eine Aufgabe herkulischen Ausmaßes, aber die Anpassung der Programmeldungen sollte in ein bis zwei Nachmittagen über die Bühne zu bekommen sein, wenn man sowohl die neue als auch eine der bisher vorhandenen Sprachen gut kennt. Leider ist die Übersetzung auch nichts, was man Stück für Stück machen kann, denn der Ressourcencompiler kann im Moment nicht mit einer variablen Zahl von Sprachen in den verschiedenen Meldungen umgehen, es heißt also 'alles oder nichts'.

Als erstes ergänzt man in `header.res` die neue Sprache. Die für die Sprache passende zweibuchstabige Abkürzung holt man sich vom nächsten Unix-System (wenn man nicht ohnehin darauf arbeitet...), die internationale Vorwahl aus dem nächsten DOS-Handbuch.

Im zweiten Schritt geht man jetzt durch alle anderen `.res`-Dateien und ergänzt die `Message`-Statements. Nocheinmal sei darauf hingewiesen, Sonderzeichen in der HTML-artigen Schreibweise und nicht direkt einzusetzen!

Wenn dies geschafft ist, kann man mit einem *make* alle betroffenen Teile neu bauen und erhält danach einen Assembler, der eine Sprache mehr schickt. Bitte nicht vergessen, die Ergebnisse an mich weiterzuleiten, damit mit der nächsten Release alle etwas davon haben :-)

Literaturverzeichnis

[1] Steve Williams:
68030 Assembly Language Reference.
Addison-Wesley, Reading, Massachusetts, 1989

[2] Advanced Micro Devices:
AM29240, AM29245, and AM29243 RISC Microcontrollers.
1993

[3] Apple Corporation::
6502 Floating Point Routines.
Apple II Reference Manual (Red Book), January 1978, pages 94-95
http://www.6502.org/source/floats/wozfp3.txt

[4] Atmel Corp.:
AVR Enhanced RISC Microcontroller Data Book.
May 1996

[5] Atmel Corp.:
8-Bit AVR Assembler and Simulator Object File Formats (Preliminary).
(part of the AVR tools documentation)

[6] Commodore Semiconductor Group:
65CE02 Microprocessor Preliminary Data Sheet.

[7] CMD Microcircuits:
G65SC802/G65SC816 CMOS 8/16-Bit Microprocessor.
Family Data Sheet.

[8] Freescale Semiconductor:
Digital Signal Processing Libraries Using the ColdFire eMAC and MAC User's Manual. DSPLIBUM, Rev. 1.2, 03/2006

[9] National Semiconductor:
COP410L/COP411L/COP310L/COP311L Single-Chip N-Channel Microcontrollers. RRD-B30M105, March 1992

[10] National Semiconductor:
COPS Family User's Guide.

[11] General Instrument Microelectronics:
Series 1600 Microprocessor System Documentation..
S16DOC-CP-1600-04, May 1975
http://www.bitsavers.org/components/gi/CP1600/
CP-1600_Microprocessor_Users_Manual_May75.pdf

[12] General Instrument Microelectronics:
CP1610 Data Sheet..
https://wiki.console5.com/tw/images/c/ce/CP1610.pdf

[13] Digital Research:
CP/M 68K Operating System User's Guide.
1983

[14] Cyrix Corp.:
FasMath 83D87 User's Manual.
1990

[15] Dallas Semiconductor:
DS80C320 High-Speed Micro User's Guide.
Version 1.30, 1/94

[16] Fairchild Semiconductor:
ACE1101 Data Sheet.
Preliminary, May 1999

[17] Fairchild Semiconductor:
ACE1202 Data Sheet.
Preliminary, May 1999

[18] Fairchild Semiconductor:
ACEx Guide to Developer Tools. AN-8004, Version 1.3 September 1998

[19] Fairchild Micro Systems:
F8 User's Guide. 67095665, 02-13-1976

[20] Fairchild Micro Systems:
F8 Guide to Programming 67095664, 1976

[21] Freescale Semiconductor:
S12XCPUV1 Reference Manual. S12XCPUV1, v01.01, 03/2005

[22] Freescale Semiconductor:
RS08 Core Reference Manual. RS08RM, Rev. 1.0, 04/2006

[23] Freescale Semiconductor:
MC9S12XDP512 Data Sheet. MC9S12XDP512, Rev. 2.11, 5/2005

[24] Fujitsu Limited:
June 1998 Semiconductor Data Book.
CD00-00981-1E

[25] Fujitsu Semiconductor:
F^2MC16LX 16-Bit Microcontroller MB90500 Series Programming Manual.
CM44-00201-1E, 1998

[26] *CPU Comparison with Z80.*
https://gbdev.io/pandocs/CPU_Comparison_with_Z80.html

[27] Hitachi Ltd.:
8-/16-Bit Microprocessor Data Book.
1986

[28] Trevor J.Terrel & Robert J. Simpson:
Understanding HD6301X/03X CMOS Microprocessor Systems.
published by Hitachi

[29] Hitachi Microcomputer:
H8/300H Series Programming Manual.
(21-032, no year of release given)

[30] Hitachi America, Ltd.:
HD641016 User's Manual.
ADE-602-003A, September 1989

[31] *HuC6280 CMOS 8-bit Microprocessor Software Manual.*

[32] Rockwell:
R65C19 Microcomputer Data Sheet.
Document Number 29400N10, January 1992

[33] Hitachi Semiconductor Design & Development Center:
SH Microcomputer Hardware Manual (Preliminary).

[34] Hitachi Semiconductor and IC Div.:
SH7700 Series Programming Manual.
1st Edition, September 1995

[35] Hitachi America Ltd.: *HMCS400 Series Handbook: Users Manual* AD-E00078, March 1988

[36] Hitachi Semiconductor and IC Div.:
H8/500 Series Programming Manual.
(21-20, 1st Edition Feb. 1989)

[37] Hitachi Ltd.:
H8/532 Hardware Manual.
(21-30, no year of release given)

[38] Hitachi Ltd.:
H8/534,H8/536 Hardware Manual.
(21-19A, no year of release given)

[39] Hewlett Packard:
Nano Processor's Users Guide.
Drawing Number A-5955-0331-1
https://www.hp9845.net/9845/downloads/manuals/Nanoprocessor.pdf

[40] IBM Corp.:
PPC403GA Embedded Controller User's Manual.
First Edition, September 1994

[41] IBM Corp.:
IBM5100 ISA Opcodes Micro Instruction Set.
https://github.com/voidstar78/IBM_5100_DOCS/blob/main/
PDFs_5100/IBM5100_ISA_opcodes_micro-instruction_set.pdf

[42] IBM Corp.:
IBM5110 BASIC Reference Manual.
SA21-9306-0, First Edition, January 1978
http://www.bitsavers.org/pdf/ibm/5110/
SA21-9308-0_IBM_5110_BASIC_Reference_Manual_Jan1978.pdf

[43] IBM Corp.:
IBM5100 Maintenance Information Manual.
SY31-0405-3, October 1979

[44] Dokumentation zum IBM5110:
http://computermuseum.informatik.uni-stuttgart.de/dev/ibm_5110/technik/5110.html

[45] Intel Corp.:
Embedded Controller Handbook.
1987

[46] Intel Corp.:
Microprocessor and Peripheral Handbook, Volume I Microprocessor.
1988

[47] Intel Corp. :
MCS-48 Family of Single Chip Microcomputers User's Manual.
September 1980

[48] Intel Corp. :
80960SA/SB Reference Manual.
1991

[49] Intel Corp.:
8XC196NT Microcontroller User's Manual.
June 1995

[50] Intel Corp.:
8XC251SB High Performance CHMOS Single-Chip Microcontroller.
Sept. 1995, Order Number 272616-003

[51] Intel Corp.:
80296SA Microcontroller User's Manual.
Sept. 1996

[52] Intel Corp.:
4040: Single-Chip 4-Bit P-Channel Microprocessor.
(no year of release given)

[53] Intersil:
CDP1802A, CDP1802AC, CDP1802BC CMOS 8-Bit Microprocessors.
March 1997

[54] : RCA Inc.: *CDP1804, CDP1804C Types Objective Data.*
(no year of release given)

[55] Intersil:
CDP1805AC, CDP1806AC CMOS 8-Bit Microprocessor with On-Chip RAM and Counter/Timer.
March 1997

[56] Intersil:
IM6100 CMOS 12 Bit Microprocessor.
(no year of release given)

[57] Hirotsugu Kakugawa:
A memo on the secret features of 6309.
(available via World Wide Web:
http://www.cs.umd.edu/users/fms/comp/CPUs/6309.txt)

[58] KENBAK:
Programming Reference Manual KENBAK-1 Computer.
4/1/1971

[59] Lattice Semiconductor Corporation:
LatticeMico8 Microcontroller Users Guide.
Reference Design RD1026, February 2008

[60] Microchip Technology Inc.:
Microchip Data Book.
1993 Edition

[61] US Department Of Defense:
Military Standard Sixteen-Bit Computer Instruction Set Architecture.
MIL-STD-1750A (USAF), 2 July 1980

[62] Mitsubishi Electric:
Single-Chip 8-Bit Microcomputers.
Vol.2, 1987

[63] Mitsubishi Electric:
Single-Chip 16-Bit Microcomputers.
Enlarged edition, 1991

[64] Mitsubishi Electric:
Single-Chip 8 Bit Microcomputers.
Vol.2, 1992

[65] Mitsubishi Electric:
M34550Mx-XXXFP Users's Manual.
Jan. 1994

[66] Mitsubishi Electric:
7751 Series Software Manual.
Rev. 1.01, 980731

[67] Mitsubishi Electric:
M16 Family Software Manual.
First Edition, Sept. 1994

[68] Mitsubishi Electric:
M16C Software Manual.
First Edition, Rev. C, 1996

[69] Mitsubishi Electric:
M30600-XXXFP Data Sheet.
First Edition, April 1996

[70] documentation about the M16/M32-developer's package from Green Hills Software

[71] Mostek Corporation:
Circuits and Systems Product Guide.
1980, STD No 01009

[72] Mostek Corporation:
3870/F8 Microcomputer Data Book.
1981, Publication Number MK79602

[73] Motorola Inc.:
Microprocessor, Microcontroller and Peripheral Data.
Vol. I+II, 1988

[74] Motorola Inc.:
MC68881/882 Floating Point Coprocessor User's Manual.
Second Edition, Prentice-Hall, Englewood Cliffs 1989

[75] Motorola Inc.:
MC68851 Paged Memory Management Unit User's Manual.
Second Edition, Prentice-Hall, Englewood Cliffs 1989,1988

[76] Motorola Inc.:
CPU32 Reference Manual.
Rev. 1, 1990

[77] Motorola Inc.:
DSP56000/DSP56001 Digital Signal Processor User's Manual.
Rev. 2, 1990

[78] Motorola Inc.:
MC68340 Technical Summary.
Rev. 2, 1991

[79] Motorola Inc.:
CPU16 Reference Manual.
Rev. 1, 1991

[80] Motorola Inc.:
Motorola M68000 Family Programmer's Reference Manual.
1992

[81] Motorola Inc.:
MC68332 Technical Summary.
Rev. 2, 1993

[82] Motorola Inc.:
PowerPC 601 RISC Microprocessor User's Manual.
1993

[83] Motorola Inc.:
PowerPC(tm) MPC505 RISC Microcontroller Technical Summary.
1994

[84] Motorola Inc.:
PowerPC(tm) MPC821 Portable Microprocessor User's Manual.
1996

[85] Motorola Inc.:
CPU12 Reference Manual.
1st edition, 1996

[86] Motorola Inc.:
CPU08 Reference Manual.
Rev. 1 (no year of release given im PDF-File)

[87] Motorola Inc.:
MC68360 User's Manual.

[88] Motorola Inc.:
MCF 5200 ColdFire Family Programmer's Reference Manual.
1995

[89] Motorola Inc.:
*M*Core Programmer's Reference Manual.*
1997

[90] Motorola Inc.:
DSP56300 24-Bit Digital Signal Processor Family Manual.
Rev. 0 (no year of release given im PDF-File)

[91] Motorola Inc.:
MC68HC11K4 Technical Data. 1992

[92] National Semiconductor:
SC/MP Programmier- und Assembler-Handbuch.
Publication Number 4200094A, Aug. 1976

[93] National Semiconductor:
COP800 Assembler/Linker/Librarian User's Manual.
Customer Order Number COP8-ASMLNK-MAN
NSC Publication Number 424421632-001B
August 1993

[94] National Semiconductor:
COP87L84BC microCMOS One-Time-Programmable (OTP) Microcontroller.
Preliminary, March 1996

[95] National Semiconductor:
IMP-16 Programming and Assembler Manual.
Order No. IMP-16S/102YB, Pub. No 4200002B, November 1973

[96] National Semiconductor:
IMP-16L Users Manual.
Order No. IMP-16L/928X, Pub. No 4200028X, April 1974

[97] National Semiconductor:
IPC-16A/520D MOS/LSI Single Chip Microprocessor (PACE).

[98] National Semiconductor:
PACE Microprocessor Assembly Language Programming Manual.
Order No. IPC-16S/969Y, Pub. No 4200130A, January 1977

[99] National Semiconductor:
INS8900 Single-Chip 16-Bit N-Channel Microprocessor.
1977

[100] National Semiconductor:
SC14xxx DIP commands Reference guide.
Application Note AN-D-031, Version 0.4, 12-28-1998

[101] National Semiconductor:
INS8070-Series Microprocessor Family. October 1980

[102] National Semiconductor:
Series 32000 Databook.
1986

[103] National Semiconductor:
NS32CG16-10/NS32CG16-15 High Performance Printer/Display Processor.
Preliminary, November 1995

[104] National Semiconductor and Udo Möller:
Series 32000 Instruction Set Manual.
6. September 2017, available via www.cpu.ns32k.net

[105] National Semiconductor:
CompactRISC CR16A Programmer's Reference Manual.
Part Number: 424521426-006, February 1997

[106] National Semiconductor:
CompactRISC CR16B Programmer's Reference Manual.
Literature Number: 633150-001, September 1997

[107] National Semiconductor:
CompactRISC CR16C Programmer's Reference Manual.
Part Number: 424521772-101, March 2002

[108] NEC Corp.:
μpD70108/μpD70116/μpD70208/μpD70216/μpD72091 Data Book.
(no year of release given)

[109] NEC Electronics Europe GmbH:
User's Manual μCOM-87 AD Family.
(no year of release given)

[110] NEC Corp.:
µCOM-75x Family 4-bit CMOS Microcomputer User's Manual.
Vol. I+II (no year of release given)

[111] NEC Corp.:
78K/II Series 8-Bit Single-Chip Microcontroller User's Manual - Instructions.
Document No. U10228EJ6V0UM00 (6th edition), December 1995

[112] NEC Corp.:
uPD78310/312CW/G 8 Bit CMOS Microcomputer Product Description.

[113] NEC Corp.:
Digital Signal Processor Product Description.
PDDSP.....067V20 (no year of release given)

[114] NEC Corp.:
µPD78070A, 78070AY 8-Bit Single-Chip Microcontroller User's Manual.
Document No. U10200EJ1V0UM00 (1st edition), August 1995

[115] NEC Corp.:
Data Sheet µPD78014.

[116] NEC Corp.:
µPD70616 Programmer's Reference Manual.
Preliminary Information, November 1986

[117] NEC Corp.:
µCOM-43 Single Chip Microcomputer Users' Manual.
January 1978

[118] NXP/Freescale:
CPU S12Z Reference Manual.
CPUS12ZRM, Rev. 1.01, 01/2013

[119] NXP:
MC9S12ZVC-Family Reference Manual and Datasheet.
MC9S12ZVCRMV1, Rev. 1.9, 29-January-2018

[120] Parallax Inc.
SX20AC/SX28AC Data Sheet. Revision 1.7, 4/23/2008

[121] Digital Equipment Corporation:
PDP-11 Architecture Handbook. 1983 Edition, EB-23657-18

[122] Digital Equipment Corporation:
PDP-11 MACRO-11 Language Reference Manual. March 1983, AA-V027A-TC

[123] OKI Semiconductor:
Microcontroller Data Book. Second Edition, December 1986

[124] Philips Semiconductor:
MAB84X1, MAF84X1, MAF84AX1 Family Datasheet.
August 1990

[125] Philips Semiconductor:
16-bit 80C51XA Microcontrollers (eXtended Architecture).
Data Handbook IC25, 1996

[126] Rockwell International Corporation:
Parallel Processing System (PPS-4) Microcomputer Data Sheet.
Document No. 29003 N11, 1973

[127] Renesas Electronics: *SH7205 Group User's Manual: Hardware.*
Rev.3.00, Mar 2014

[128] Renesas Electronics:
SH7615 Group Hardware Manual.
Rev.2.00, 2005.03

[129] Renesas Electronics: *SH7750, SH7750S, SH7750R Group User's Manual: Hardware.*
Rev.7.02, Sep 2013

[130] Renesas Electronics: *SH7780 Hardware Manual.* Rev.1.00, 2005.12

[131] RX Family Product Page,
https://www.renesas.com/us/en/products/
microcontrollers-microprocessors/
rx-32-bit-performance-efficiency-mcus

[132] RX Family Compiler and Assembler Manual:
https://www.renesas.com/us/en/document/mat/
high-performance-embedded-workshop-rx-family-
cc-compiler-package-v101-users-manual-includes-v102

[133] Sharp:
Pocket Computer PC-1350 Machine Language Reference Manual.
https://pockemul.com/wp-content/uploads/2022/12/
PC1350_ML_EN.pdf

[134] Sharp Corporation:
ESR-L Instruction Manual.
https://web.archive.org/web/20060521205349if_/http://www.sarnau.info:80/files/ESR-LInstructionManual.pdf

[135] Signetics Corporation:
Signetics 2650 Microprocessor.
1975

[136] Signetics Corporation:
2650 User's Manual.

[137] SGS-Ates:
M3870 Microcomputer Family Databook.
1st edition, issued January 1983

[138] SGS-Thomson Microelectronics:
8 Bit MCU Families EF6801/04/05 Databook.
1st edition, 1989

[139] SGS-Thomson Microelectronics:
ST6210/ST6215/ST6220/ST6225 Databook.
1st edition, 1991

[140] SGS-Ates: *M380 uP System and Instruction Set.*
6.4.1976

[141] SGS-Thomson Microelectronics:
ST7 Family Programming Manual.
June 1995

[142] SGS-Thomson Microelectronics:
ST9 Programming Manual.
3rd edition, 1993

[143] Siemens AG:
SAB80C166/83C166 User's Manual.
Edition 6.90

[144] Siemens AG:
SAB C167 Preliminary User's Manual.
Revision 1.0, July 1992

[145] Siemens AG:
Telephone Controller (Single-Chip 8-Bit CMOS Microcontroller) SAB80C382/SAB80C482.
May 1987

[146] Siemens AG:
SAB-C502 8-Bit Single-Chip Microcontroller User's Manual.
Edition 8.94

[147] Siemens AG:
SAB-C501 8-Bit Single-Chip Microcontroller User's Manual.
Edition 2.96

[148] Siemens AG:
C504 8-Bit CMOS Microcontroller User's Manual.
Edition 5.96

[149] ST Microelectronics:
STM8 CPU Programming Manual.
PM0044, Doc ID 13590 Rev 3, September 2011

[150] ST Microelectronics:
STM8S Series and STM8AF Series 8-bit Microcontrollers Reference Manual.
RM0016, DocID14587 Rev 14, October 2017

[151] ST Microelectronics:
STM8S003F3 STM8S003K3 Data Sheet.
DS7147 Rev 10, August 2018

[152] C.Vieillefond:
Programmierung des 68000.
Sybex-Verlag Düsseldorf, 1985

[153] Symbios Logic Inc:
Symbios Logic PCI-SCSI-I/O Processors Programming Guide.
Version 2.0, 1995/96

[154] Texas Instruments:
Model 990 Computer/TMS9900 Microprocessor Assembly Language Programmer's Guide.
1977, Manual No. 943441-9701

[155] Texas Instruments:
TMS9995 16-Bit Microcomputer.
Preliminary Data Manual 1981

[156] Texas Instruments:
TMS99105 and TMS99110A 16-Bit Microprocessors.
Preliminary Data Manual 1982

[157] Texas Instruments:
First-Generation TMS320 User's Guide.
1988, ISBN 2-86886-024-9

[158] Texas Instruments:
TMS7000 Family Data Manual.
1991, DB103

[159] Texas Instruments:
TMS320C3x User's Guide.
Revision E, 1991

[160] Texas Instruments:
TMS320C2x User's Guide.
Revision C, Jan. 1993

[161] Texas Instruments:
TMS320C4x User's Guide.
SPRU063C, May 1999

[162] Texas Instruments:
TMS370 Family Data Manual.
1994, SPNS014B

[163] Texas Instruments:
MSP430 Family Software User's Guide.
1994, SLAUE11

[164] Texas Instruments:
MSP430 Metering Application.
1996, SLAAE10A

[165] Texas Instruments:
MSP430 Family Architecture User's Guide.
1995, SLAUE10A

[166] Texas Instruments:
MSP430 MSP430x5xx and MSP430x6xx Family User's Guide.
October 2016, SLAU208

[167] Texas Instruments:
TMS320C62xx CPU and Instruction Set Reference Manual.
Jan. 1997, SPRU189A

[168] Texas Instruments:
TMS320C20x User's Guide.
April 1999, SPRU127C

[169] Texas Instruments:
TMS320C54x DSP Reference Set; Volume 1: CPU and Peripherals.
March 2001, SPRU172C

[170] Texas Instruments:
TMS320C54x DSP; Volume 2: Mnemonic Instruction Set.
March 2001, SPRU172C

[171] Texas Instruments:
TMS 1000 Series MOS/LSI One-Chip Microcomputers Programmer's Reference Manual.
CM122-1 1275, 1975

[172] Texas Instruments:
TMS34010 Assembly Language Tools User's Guide.
SPVU004A, 1604897-9704, 1987

[173] Texas Instruments:
TMS34010 C Compiler User's Guide.
SPVU005, December 1986

[174] Texas Instruments:
TMS34010 User's Guide.
SPVU001A, 1604899-9703, August 1988

[175] Texas Instruments:
TMS34020 User's Guide.
SPVU019, 2564006-9761, August 1990

[176] Toshiba Corp.:
8-Bit Microcontroller TLCS-90 Development System Manual.
1990

[177] Toshiba Corp.:
8-Bit Microcontroller TLCS-870 Series Data Book.
1992

[178] Toshiba Corp.:
16-Bit Microcontroller TLCS-900 Series Users Manual.
1992

[179] Toshiba Corp.:
16-Bit Microcontroller TLCS-900 Series Data Book: TMP93CM40F/ TMP93CM41F.
1993

[180] Toshiba Corp.:
4-Bit Microcontroller TLCS-47E/47/470/470A Development System Manual.
1993

[181] Toshiba Corp.:
TLCS-9000/16 Instruction Set Manual Version 2.2.
10. Feb 1994

[182] Toshiba Corp.:
TC9331 Digital Audio Signal Processor Application Information.

[183] Toshiba Corp.:
Microcomputer LSI Data Book. July 1984

[184] Valvo GmbH:
Bipolare Mikroprozessoren und bipolare LSI-Schaltungen.
Datenbuch, 1985, ISBN 3-87095-186-9

[185] Digital Equipment Corporation:
VAX Architecture Reference Manual.
Order Number EY-3459E-DP, 1987

[186] Western Digital Corporation:
WD16 Microcomputer Programmer's Reference Manual.
4 October 1976

[187] AT&T Technologies Inc.:
WE32100 Microprocessor Information Manual.
1985, 451-000

[188] AT&T Technologies Inc.:
WE32200 Microprocessor Information Manual.
Preliminary, November 1987, 307-705

[189] AT&T Technologies Inc.:
AT&T 3B2/3B5/3B15 Computers Assembly Language Programming Manual.
305-000, March 1986

[190] AT&T Technologies Inc.:
AT&T WE 32-Bit Microprocessors and Peripherals. August 1987

[191] Ken Chapman (Xilinx Inc.):
PicoBlaze 8-Bit Microcontroller for Virtex-E and Spartan-II/IIE Devices.
Application Note XAPP213, Version 2.1, February 2003

[192] Xilinx Inc.:
PicoBlaze 8-bit Embedded Microcontroller User Guide for Spartan-3, Virtex-II, and Virtex-II Pro FPGAs.
UG129 (v1.1) June 10, 2004

[193] David May::
The XMOS XS1 Architecture.
Publication Date: 2009/10/19, Copyright 2009 XMOS Ltd.

[194] data sheets from Zilog about the Z80 family

[195] Zilog Inc.:
Z8 Microcontrollers Databook.
1992

[196] Zilog Inc.:
Discrete Z8 Microcontrollers Databook.
(no year of release given)

[197] Zilog Inc.:
Z280 MPU Microprocessor Unit Technical Manual.
July 1987

[198] Zilog Inc.:
Z380 CPU Central Processing Unit User's Manual.
(no year of release given)

[199] Zilog Inc.:
eZ8 CPU User Manual.
UM01285-0503

[200] Zilog Inc.:
Z88C00 CMOS Super8 ROMless MCU Product Specification
2003, PS014602-0103

[201] Zilog Inc.:
Z8 Encore! F0830 Series Product Specification
2012, PS025113-1212

[202] Zilog Inc.:
Z8000 Technical Manual
January 1983

[203] Zilog Inc.:
eZ80 CPU User Manual
2015, UM007715-0415

[204] Zilog Inc.:
eZ80190 Product Specification
2008, PS006614-1208

[205] Zilog Inc.:
eZ80F91 MCU Product Specification
2022, PS019217-1222

[206] Zilog Inc.:
eZ80F92/eZ80F93 Product Specification
2020, PS015317-0120

[207] Zilog Inc.:
eZ80L92 MCU Product Specification
2016, PS013015-0316

"Ich schlage vor, dem Parlament ein Gesetz vorzulegen,
das einem Autor, der ein Buch ohne Index publiziert,
das Copyright entzieht und ihn außerdem für sein Vergehen
mit einer Geldstrafe belegt."
–Lord John Campbell

Index

.EQU, 73
.PAGE, 182
.RESTORE, 128
.SAVE, 128
.SET, 73
.SHIFT, 172
.SWITCH, 180

ACCMODE, 119
ACON, 153
ADDR, 148
ADDRW, 148
ADR, 152
ALIGN, 160
ASCIC, 156
ASCII, 156
ASCIZ, 156
ASSUME, 129

BF16, 146
BFLOAT, 154
BIGENDIAN, 124
BINCLUDE, 196
BIT, 76
BLKB, 146, 151
BLKD, 146, 151
BLKF, 151
BLKG, 151
BLKH, 151
BLKL, 146, 151
BLKO, 151
BLKQ, 151
BLKW, 146, 151
BLOCK, 158
BSS, 159
BYT, 151
BYTE, 148, 152

CASE, 180
CHARSET, 82
CIS, 119
CKPT, 142
CODEPAGE, 84
CODEPAGE_VAL, 82
COMPMODE, 199
CONSTANT, 73
COPROC, 120
CPU, 93
CUSTOM, 118

D1, 149
D_FLOATING, 150
DATA, 156
DB, 146, 150
DBIT, 77
DC, 144
DC24, 150
DC8, 152
DCM, 153
DD, 146, 150
DDB, 152
DECIMAL, 186
DEFB, 148
DEFBIT, 78
DEFBITB, 78
DEFBITFIELD, 79
DEFW, 148
DEPHASE, 127

DFS, 158
DN, 146, 150
DO, 146
DOTTEDSTRUCTS, 174
DOUBLE, 148, 150, 154
DQ, 146
DS, 145, 151
DS16, 159
DS8, 151
DSB, 159
DSW, 159
DT, 146
DUP, 147
DW, 146, 150
DW16, 153

EFLOAT, 154
EIS, 119
ELSE, 179
ELSECASE, 180
ELSEIF, 179
EMULATED, 142
END, 200
ENDCASE, 180
ENDEXPECT, 144
ENDIAN, 124
ENDIF, 179
ENDM, 161
ENDS, 174
ENDSTRUC, 174
ENDSTRUCT, 174
ENDUNION, 174
ENUM, 84
ENUMCONF, 84
EQU, 73
ERROR, 196
EVEN, 160
EXECMODE, 119
EXITM, 171
EXPECT, 144
EXTENDED, 154
EXTMODE, 123
EXTRACOMMENTS, 123

F_FLOATING, 150
FATAL, 196
FB, 156
FCB, 151
FCC, 158
FDB, 152
FIELD, 155
FIS, 119
FLOAT, 148, 150, 154
FLT, 150
FLT2, 150
FLT3, 150
FLT4, 150
FP11, 119
FPU, 118
FULLPMMU, 120
FUNCTION, 173
FW, 156

G_FLOATING, 150

H_FLOATING, 150
HALF, 148

IF, 179
IFB, 179
IFDEF, 179
IFEXIST, 179
IFNB, 179
IFNDEF, 179
IFNEXIST, 179
IFNSYMEXIST, 179
IFNUSED, 179
IFSYMEXIST, 179
IFUSED, 179
INCLUDE, 195
INT, 148
INTSYNTAX, 198
IRP, 169

IRPC, 170

LABEL, 76
LISTING, 185
LIV, 81
LONG, 148, 153
LQxx, 155
LTORG, 160
LWORD, 148
LWORDMODE, 123

MACEXP, 183
MACEXP_DFT, 183
MACEXP_OVR, 183
MACRO, 161
MAXMODE, 122
MAXNEST, 173
MESSAGE, 196

NAMEREG, 81
NEWPAGE, 183
NEXTENUM, 84

OCTAL, 186
ORG, 86
OUTRADIX, 187

PACKED, 157
PACKING, 122
PADDING, 120
PAGE, 182
PAGESIZE, 182
PANEL, 125
PHASE, 127
PLAINBASE, 124
PMMU, 118
POPV, 85
PORT, 80
PRTEXIT, 185
PRTINIT, 185
PUSHV, 85

QUAD, 148
Qxx, 155

RADIX, 186
RADIX50, 157
READ, 197
REG, 81
Registersymbole, 69
RELAXED, 199
REPT, 170
RES, 159
RESTORE, 128
RESTOREENV, 128
RIV, 81
RMB, 158
RORG, 93
RSTRING, 157

SAVE, 128
SAVEENV, 128
SEGMENT, 126
SELECT, 180
SET, 73
SFR, 75
SFRB, 75
SHARED, 32, 70, 194
SHFT, 172
SHIFT, 172
SINGLE, 154
SPACE, 158
SRCMODE, 123
STRING, 148, 157
STRUC, 174
STRUCT, 174
SUPMODE, 118
SWITCH, 180

TFLOAT, 154
TITLE, 186

UNION, 174

WARNING, 196

WHILE, 171
WORD, 148, 153
WRAPMODE, 125

XSFR, 75

YSFR, 75

Z80SYNTAX, 143
ZERO, 156